Implementação de piloto automático em veículo agrícola utilizando plataforma embarcada Linux de tempo real

Pedro Augusto Ceriotti

Relatório submetido à Universidade Federal de Santa Catarina

como requisito para a aprovação na disciplina

DAS 5511: Projeto de Fim de Curso

Florianópolis, agosto de 2016

Implementação de piloto automático em veículo agrícola utilizando plataforma embarcada Linux de tempo real

Pedro Augusto Ceriotti

Esta monografia foi julgada no contexto da disciplina

DAS5511: Projeto de Fim de Curso

e aprovado na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Rômulo Silva de Oliveira

Assinatura do orientador

Banca Examinadora:

Jonatan Vieira

Orientador na Empresa

Prof. Rômulo Silva de Oliveira

Orientador no Curso

Marcelo de Lellis Costa de Oliveira

Avaliador

Rodrigo da Silva Gesser

Gustavo Rosa Lopes Bodini

Debatedores

Universidade Federal de Santa Catarina

Resumo

Centro Tecnológico

Departamento de Automação e Sistemas

Implementação de piloto automático em veículo agrícola utilizando plataforma embarcada Linux de tempo real

por Pedro Augusto Ceriotti

A produção de grãos e alimentos vem recebendo previsões de crescimento a nível global ao longo dos últimos anos. Nesse contexto a agricultura de precisão vem ganhando cada vez mais espaço, aliando tecnologia, produtividade e sustentabilidade para criar soluções de engenharia completas desenvolvidas especificamente para este propósito. O piloto automático, sistema implementado em máquinas agrícolas que permite maior paralelismo entre as passadas, economia de insumos e precisão na operação, é uma delas. Este trabalho visa à implementação de um sistema de piloto automático utilizando uma plataforma embarcada Linux de tempo real. O objetivo principal é garantir não só o funcionamento lógico, como também o determinismo temporal do sistema devido à concorrência com outros processos rodando sobre o mesmo sistema operacional. As implementações consistem em canais de comunicação com o computador de bordo e interface com sensores e atuadores de forma a atuar no sentido de minimizar o erro de seguimento da trajetória de referência. Além disso, deve-se garantir o cumprimento dos *deadlines* associados às malhas de controle, uma vez que o não-determinismo pode acarretar em falhas severas e irreparáveis. O presente relatório propõe soluções para atingir esses objetivos, bem como apresenta os resultados atingidos através de gráficos e discussões.

Federal University of Santa Catarina

Abstract

Technological Center

Department of Automation and Systems

Implementation of autonomous driving for agricultural vehicles using real-time embedded Linux platform

by Pedro Augusto Ceriotti

Grains and food production has been receiving constantly growing forecasts in a global level during recent years. In this context precision agriculture is becoming even more popular, combining technology, productivity and sustainability to develop complete engineering solutions specifically for this purpose. One of these solutions is the autopilot, a system implemented in agricultural vehicles, which provides better parallelism between the lines, resource savings and precision on the operation. This work aims to the implementation of an autonomous driving system using an embedded real-time Linux platform. The main goal is to guarantee not only the logical behavior of the system, but also to achieve timing constraints due to the concurrency with other processes running on the same operating system. The implementations consist of communication channels with the on-board computer and interface with sensors and actuators in order to minimize the reference tracking error. Besides that, it must be guaranteed that the deadlines associated with the control loops will be achieved, once the non-determinism could lead to severe and irrecoverable failures. This thesis proposes solutions to achieve these goals, as well as presents the results achieved through graphics and discussions.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de símbolos

ADEOS	Adaptative Domain Environment for Operating Systems
CAN	Controller Area Network
CPU	Central Processing Unit
CS	Chip Select
DDR	Double Data Rate
EEPROM	Electrically-Erasable Programmable Read-Only Memory
FIFO	First-In First-Out
GCC	GNU Compiler Collection
GNSS	Global Navigation Satellite System
GLONASS	Global Orbiting Navigation Satellite System
GPS	Global Positioning System
GSM	Global System for Mobile Communication
I2C	Inter-Integrated Circuit
IPC	Inter-Process Communication
LCD	Liquid Crystal Display
MEMS	Micro-Electro-Mechanical Systems
MISO	Master-In Slave-Out
MOSI	Master-Out Slave-In
NED	North-East-Down
PWM	Pulse Width Modulation
RAM	Random Access Memory
SPI	Serial Peripheral Interface
SS	Slave Select
USB	Universal Serial Bus
UTM	Universal Transverse Mercator

Capítulo 1

Introdução

Este capítulo visa apresentar o tema do presente relatório no contexto em que ele está inserido, bem como descrever a motivação e a problemática do tema em questão. Ao final do capítulo será feita uma breve relação das áreas de atuação englobadas pelo projeto e, por fim, será apresentada a estrutura do relatório que se segue.

1.1 Motivação

A agricultura brasileira avançou como nenhuma outra na direção da sustentabilidade. Ao longo dos últimos quarenta anos o país foi capaz de transformar grandes extensões de terras pobres e ácidas em terras férteis. Além disso, fomos capazes de desenvolver uma plataforma de práticas sustentáveis sem igual no planeta - fixação biológica de nitrogênio, controle biológico, plantio direto, sistemas integrados. Esta foi a primeira grande revolução da agricultura brasileira. Agora estamos prestes a entrar na segunda grande revolução.[2]

Esta nova etapa tem suas bases apoiadas fortemente no uso da automação e da agricultura de precisão nas lavouras, de maneira a fazer com que o agronegócio no país evolua constantemente na direção de um sistema produtivo eficaz e sustentável, que consiga extrair o máximo de produtividade sem comprometer o meio ambiente. Para que isso seja possível, é necessário que haja a conscientização do produtor rural e da sociedade em geral, aliada à introdução de novas tecnologias que visem auxiliar e otimizar a aplicação das técnicas de agricultura de precisão. Essas técnicas partem do pressuposto

de que as propriedades do solo não são uniformes dentro de uma área de plantio, tendo que ser considerada sua variabilidade espacial para que haja uma análise mais profunda de suas características e dessa forma a aplicação de insumos, fertilizantes e agrotóxicos, bem como as etapas de plantio e colheita sejam feitas de maneira mais precisa.

Algumas das vantagens do uso correto das técnicas de agricultura de precisão contribuem para o aumento da produtividade, redução da contaminação ambiental pelo uso excessivo de agrotóxicos, economia na utilização de insumos e consequente melhora na lucratividade. Estas vantagens são comprovadas no campo científico e prático. Experimentos comprovaram aumentos de produtividade de 20% a 29%, e economias de 13% a 23% de insumos agrícolas, com relação a médias nacionais.[3]

Para viabilizar a criação de novas máquinas e equipamentos capazes de operar conforme os requisitos impostos pelas técnicas de agricultura de precisão, é necessário destacar o avanço na utilização de sistemas embarcados como aliado fundamental no desenvolvimento de aplicações que necessitam operar em tempo real, de forma a garantir a integridade e bom funcionamento dos serviços. Evoluções constantes e processadores cada vez mais potentes permitem o surgimento de aplicações e produtos mais complexos do ponto de vista da capacidade de processamento, armazenamento e troca de informações. As tecnologias disponíveis indicam que há potencial para gerar sistemas de recomendação de aplicação de insumos (corretivos, fertilizantes e defensivos) e uso de recursos naturais de forma mais eficiente, com alta probabilidade de retorno econômico e baixo impacto ambiental.[2]

Nesse contexto se insere o presente projeto, que visa a implementação de um módulo de piloto automático com controle de trajetória para veículos agrícolas em uma plataforma embarcada Linux operando em tempo real, diferentemente da arquitetura utilizada atualmente, que será explicada com mais detalhes nas seções seguintes. Esse sistema visa uma otimização nas operações de plantio, colheita e aplicação de corretivos e fertilizantes fornecendo um paralelismo preciso entre as passadas, contribuindo para a economia de insumos, redução dos custos de produção e aumento da capacidade operacional. Para tanto, é preciso que o sistema de controle responda em tempo real aos eventos e sinais externos e atue de forma a alterar o ângulo da roda do veículo agrícola, visando minimizar o erro de seguimento da trajetória desejada.

1.2 Objetivo geral

O presente projeto foi desenvolvido numa empresa focada em desenvolvimento de tecnologias agrícolas, com foco em agricultura de precisão, chamada *Hexagon Agriculture*. O trabalho tem por objetivo geral o uso de uma plataforma embarcada Linux de tempo real para cálculo de orientação do veículo, a partir de dados obtidos de sensores como acelerômetro, giroscópio e um módulo GNSS *(Global Navigation Satellite System)*, bem como o processamento do algoritmo responsável pelo cálculo de trajetória do veículo e interface com sensores e atuadores. Dessa forma, o processamento das informações ficariam centralizadas sobre a plataforma Titanium, computador de bordo e produto referência da empresa, de forma a desacoplar o módulo de acionamento dos atuadores e o computador de bordo, permitindo assim um fluxo de processamento mais lógico do que o implementado atualmente. Os detalhes dos dois sistemas e a forma como eles interagem serão descritos detalhadamente nos capítulos seguintes, porém, para que se compreendam os objetivos gerais aqui descritos, será dada uma visão geral do fluxo de dados entre os dois sistemas.

Atualmente, a empresa possui o computador de bordo, o qual é um sistema embarcado operando em um microprocessador sob o sistema operacional Linux, que será chamado de Titanium ao longo do documento, responsável pela interface gráfica com o usuário, através de uma tela LCD *(Liquid Crystal Display) touchscreen* bem como interação com módulos periféricos de GNSS, WiFi e GSM *(Global System for Mobile Communication)*. No modo piloto automático, o operador define uma trajetória a ser seguida pela máquina agrícola, e o Titanium então calcula a rota e a compara com a atitude do veículo e a posição obtida através do módulo GNSS, de forma a gerar o ângulo de referência entre o eixo longitudinal do veículo e a trajetória desejada. Essa informação, bem como a informação referente ao vetor velocidade do veículo, são enviadas via CAN *(Controller Area Network)* para um outro módulo, chamado de Driver, o qual possui um microcontrolador responsável por fazer a leitura de um giroscópio e um acelerômetro e, a partir da referência de ângulo recebida do Titanium e do vetor velocidade, estimar a orientação do veículo, rodar o algoritmo de controle de trajetória e fazer o acionamento dos atuadores.

Atualmente a troca de mensagens entre os dois sitemas é feita via CAN, e há uma forte dependência entre eles, uma vez que os dados provenientes do giroscópio,

acelerômetro e GNSS não são obtidos em uma única plataforma, fazendo com que não seja possível o cálculo da orientação do veículo e desvio da trajetória desejada de forma independente. Essa troca de mensagens, através de um canal físico via rede CAN, necessárias para a correta estimação da orientação do veículo e cálculo dos parâmetros de controle, pode acarretar em latências maiores do que o permitido pelo sistema de controle, prejudicando, assim, a operação correta da máquina agrícola no modo piloto automático. Além disso, a empresa deseja desenvolver um equipamento Titanium com módulo de piloto automático funcionando em modo *stand-alone*, isto é, a placa do Driver não será mais utilizada, e o microcontrolador utilizado para acionamento dos atuadores estará posicionado sobre a mesma placa do Titanium, eliminando, assim, a comunicação via CAN entre os dois dispositivos e criando a necessidade de desenvolvimento de um novo canal de comunicação *on-board*.

Para que o objetivo geral possa ser atingido, será necessário desacoplar as duas plataformas, de forma que o Titanium seja responsável pelo cálculo de atitude do veículo, bem como do controle de trajetória e geração de rota. Em contrapartida, a plataforma microcontrolada seria responsável apenas pela malha de controle dos atuadores, podendo assim ser utilizado um microcontrolador de menor porte e baixo custo para acionamento de PWM *(Pulse Width Modulation)* e leitura dos sensores *hall*. Dessa forma, seria possível uma melhora considerável no fluxo de execução do módulo de piloto automático, bem como o desacoplamento entre as duas plataformas. Além disso, o uso de um sistema operacional de tempo real pode trazer grandes contribuições futuras para o projeto, permitindo maior determinismo, diminuição de latências e garantia de funcionamento temporal para as aplicações.

1.3 Áreas de conhecimento envolvidas

A automação de processos industriais é uma das principais necessidades para o desenvolvimento econômico do país e o sucesso de uma empresa no mercado atual. A carência de engenheiros com visão interdisciplinar, capazes de fornecer uma solução completa para os problemas encontrados, levou a Universidade Federal de Santa Catarina, no ano de 1990, a criar o curso de Engenharia de Controle e Automação.[4]

O presente projeto se enquadra em diversos pilares de conhecimento, proporcionando a oportunidade de aprofundar e experimentar vários conceitos aprendidos em aula durante o curso, nas seguintes áreas de atuação:

- **Eletrônica:** princípio de funcionamento de microcontroladores, sensores e atuadores, noções de eletrônica básica para sistemas embarcados;
- **Controle:** interpretação de modelos matemáticos para controle de trajetória, métodos de filtragem digital de sinais, algoritmos de controle de motores elétricos e válvulas;
- **Informática:** desenvolvimento de *softwares* embarcados, sistemas operacionais de tempo real, comunicação entre processos, rede de área de controladores.

1.4 Estrutura do relatório

O presente documento está organizado da seguinte forma:

- **Capítulo 2:** nesse capítulo será feita a apresentação da empresa em que está sendo realizado o projeto, além de uma contextualização com a área de agricultura de precisão e sistemas embarcados. Por fim, serão apresentados alguns dos produtos da empresa, especialmente aqueles utilizados para a realização deste projeto;
- **Capítulo 3:** nesse capítulo serão apresentados os conceitos básicos necessários para o entendimento do projeto, dentre eles estão sistemas embarcados, sistema operacional Linux, sistemas de navegação inercial, comunicação entre processos e entre dispositivos e uma breve introdução à teoria de controle de trajetória utilizada no piloto automático;
- **Capítulo 4:** esse capítulo tratará de aspectos relacionados ao desenvolvimento e implementação do projeto, abordando os problemas enfrentados, bem como as soluções adotadas para resolvê-los;
- **Capítulo 5:** nessa seção serão apresentados e discutidos os resultados obtidos através da implementação das soluções propostas, bem como os testes realizados para validação final do sistema;

- **Capítulo 6:** por fim, considerações finais do projeto serão apresentadas e trabalhos futuros serão propostos como alternativa para uma possível continuação do projeto.

Capítulo 2

Contextualização

O presente capítulo visa apresentar de forma mais detalhada as vantagens do uso da tecnologia na agricultura de precisão. Após, será feita uma apresentação da empresa em que este trabalho foi realizado, bem como uma breve descrição dos produtos que, direta ou indiretamente, estão envolvidos no escopo deste trabalho, dando ênfase ao computador de bordo Titanium e ao módulo de piloto automático.

2.1 Agricultura de precisão

As tecnologias de agricultura de precisão já são uma realidade no campo para os técnicos e produtores rurais. Está se difundindo progressivamente o conhecimento de que existe uma variabilidade nas áreas de produção, que pode ser devido às variações do relevo, solos, vegetação e também do histórico de uso.

O conhecimento da variabilidade da produção e da sua qualidade é útil para qualquer cultura, sejam aquelas cultivadas em pequenas áreas como aquelas que ocupam grandes extensões de terra. Para isso, basta que o produtor ou o técnico inicie este trabalho de observação, medida e registro destas variações. Estas diferenças fazem com que os produtores e técnicos tratem cada região de modo diferente de acordo com suas potencialidades e necessidades.[2]

Sistemas de posicionamento por satélite, como dispositivos de guia (barra de luz) e piloto automático permitem o deslocamento preciso de máquinas como semeadoras, pulverizadores e colhedoras, contribuindo para maior rendimento operacional e eficiência nas

operações mecanizadas de semeadura, tratos culturais e colheita. Como resultado, criam-se oportunidades para otimização da frota agrícola, economia de tempo, combustível, redução do desperdício de defensivos e controle de tráfego nas lavouras, amenizando os problemas de compactação do solo e de contaminação ambiental.[5]

O ciclo da agricultura de precisão caracteriza-se, de forma bastante simplificada, nas seguintes etapas:

- **Preparação do solo:** consiste primeiramente na análise do solo para identificar as causas da variação de produtividade, tais como falta de nutrientes, nível de pH do solo, teor de matéria orgânica, compactação, dentre outros. Assim que identificada a variabilidade de cada um desses fatores, é feita então uma aplicação de corretivos e fertilizantes a taxa variável, baseada nos mapas de recomendação gerados pela análise do solo;
- **Plantio:** o plantio também pode ser feito a taxas variáveis, caso seja considerado o potencial produtivo de cada seção da lavoura;
- **Acompanhamento:** acompanhamento da lavoura para mapeamento de pragas e doenças, bem como aplicação de defensivos agrícolas localmente, com base na necessidade específica de cada seção, de forma a evitar o desperdício e minimizar a contaminação ambiental;
- **Colheita:** a colheita geralmente ocorre utilizando máquinas capazes de mensurar a produtividade em cada seção, referenciando a quantidade colhida com base na posição geográfica da máquina.

Esse seria o ciclo básico desde a preparação do solo até a colheita, repetindo-se a cada safra. Para cada uma dessas etapas, o uso da tecnologia através de sensores e equipamentos é fundamental para que se atinjam os resultados esperados. Assim, é de extrema importância que se saiba a posição geográfica do veículo agrícola em cada ponto da coleta de dados ou aplicação. Essa informação é obtida através de um sistema de referenciamento via satélite, com precisão na casa dos centímetros para os sistemas mais atuais.

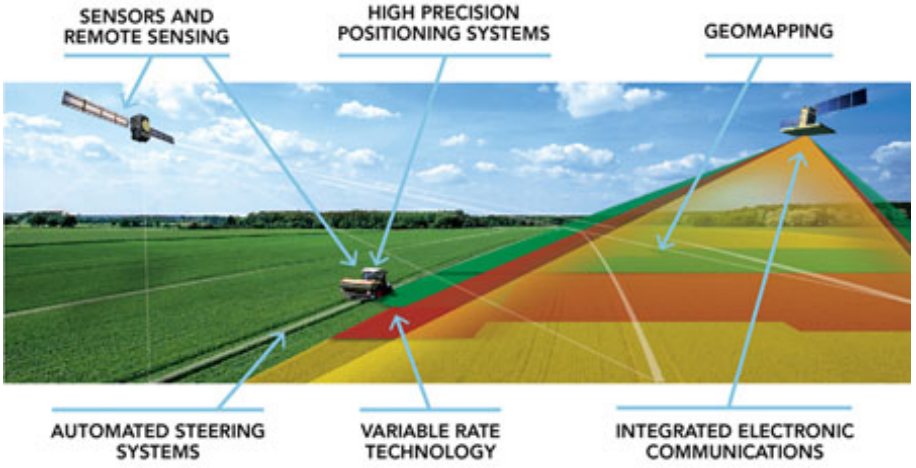


FIGURA 2.1: Esquema básico do uso da tecnologia no campo através de técnicas de agricultura de precisão.
(Fonte: http://cema-agri.org/page/precision-farming-key-technologies-concepts)

A Figura 2.1 ilustra um esquema básico do uso da tecnologia na agricultura de precisão. Conforme a máquina percorre as linhas da lavoura, seja para aplicação de corretivos e fertilizantes, plantio ou colheita, dados de um sistema GNSS são obtidos para que se saiba a posição geográfica do veículo dentro da lavoura. No caso da aplicação de insumos, a lavoura é dividida em seções, com taxas de aplicação independentes entre si, representadas na Figura 2.1 pelas áreas coloridas. Conforme o veículo cruza essas áreas, sistemas reguladores de dosagem atuam no sentido de regular a quantidade aplicada em cada seção de forma a se aproximar ao máximo da recomendação. No caso do plantio, o mesmo princípio pode ser aplicado, caso sejam identificadas áreas com diferentes potenciais de produtividade. Na colheita, a máquina mensura a quantidade colhida dentro de cada uma das seções para, ao final, gerar um mapa de produtividade da lavoura. Todas essas etapas podem ser ainda automatizadas através de um módulo de piloto automático, garantindo uma operação mais precisa e paralelismo entre as passadas, de forma a evitar sobreposições ou falhas na aplicação.

Os sistemas citados anteriormente não são os únicos que compõe o arsenal de ferramentas que podem ser utilizadas na agricultura de precisão, porém merecem mais destaque por serem tecnologias já consolidadas e que vêm ganhando cada vez mais destaque no campo do agronegócio. Além disso, alguns desses equipamentos são desenvolvidos pela empresa em que este trabalho foi realizado e serão citados com mais detalhes ao longo deste documento.

2.2 A Empresa

A Arvus Tecnologia foi fundada em 2004 visando atender, com tecnologia nacional e adaptada a produtores brasileiros, o mercado brasileiro de agricultura de precisão. No início fez parceria com grandes produtores agrícolas de Goiás e teve o apoio da Universidade Federal de Santa Catarina.

Atualmente, a Arvus possui parceiros nas principais fronteiras agrícolas do Brasil. Destaca-se pela assistência técnica personalizada e pelo efetivo resultado para os clientes. Tem portfólio de produtos desenvolvidos especificamente para o mercado nacional – equipamentos e softwares para agricultura e silvicultura de precisão. A empresa atua na venda direta desses equipamentos, bem como na prestação de serviços, sendo pioneira nesta modalidade.

Recentemente a Arvus foi adquirida pelo grupo Hexagon, com sede em Estocolmo, na Suécia. A Hexagon conta atualmente com mais de 16000 empregados, em 40 países, e faturamento anual de 2,4 bilhões de euros. A Hexagon adquiriu 115 empresas em diversos países nos últimos 10 anos e construiu um vasto portfólio de tecnologias que atendem a praticamente todos os setores da indústria, inclusive o agronegócio. Exemplos das empresas do grupo são: Leica Geosystems, NovAtel, Intergraph, Sisgraph e Devex.[6]

A Hexagon Agriculture foi instituída em 2014 pela junção da Arvus Tecnologia, ILab Sistemas e Leica Agriculture pelo grupo Hexagon. A Arvus Tecnologia é uma empresa 100% nacional presente no mercado brasileiro desde 2004 e que sempre trouxe inovações na área de agricultura de precisão e soluções para a indústria de silvicultura. A ILab, presente no mercado desde 1998, atua de forma pioneira no mercado de cana-de-açúcar oferecendo soluções de planejamento e otimização de operações. Por fim, a Leica Agriculture, divisão da reconhecida Leica Geosystems europeia, está fortemente presente no mercado europeu com produtos de altíssima qualidade e tecnologia.

A aquisição da Arvus foi um passo estratégico fundamental no desenvolvimento das soluções de *Smart Agriculture* da empresa europeia. A Arvus promoverá a expansão das ações atuais da Hexagon em agricultura, complementando as suas soluções mundiais de geoprocessamento e controle de máquinas. As tecnologias e *know-how* de cada empresa poderão ser integradas para a criação de soluções de competitividade únicas no mercado

mundial.[6] Atualmente a empresa deixou de se chamar Arvus Tecnologia e passou a se chamar Hexagon Agriculture.

Presente em todas as fronteiras agrícolas do Brasil, a Hexagon Agriculture possui unidades de desenvolvimento e negócio em Ribeirão Preto, São Paulo e Florianópolis, além de escritórios na Espanha, China e Austrália.

A Hexagon Agriculture possui uma diversa gama de produtos para o setor de agricultura e silvicultura de precisão. Esses produtos combinam tecnologias que englobam as áreas de eletrônica, metrologia, posicionamento e automação para atender às necessidades dos produtores rurais. O desenvolvimento das plataformas de *hardware* e *software* é feito quase que em sua totalidade pela própria empresa, permitindo maior flexibilidade e desempenho no desenvolvimento de aplicações específicas.

O produto de maior destaque é o computador de bordo Titanium. O computador de bordo vem equipado de fábrica com a função de barra de luzes para auxiliar o operador a manter uma trajetória pré-definida, a partir das informações obtidas do módulo GNSS. Outros módulos, tais como controlador de plantio, taxa variável, monitor de plantio, piloto automático e corte de seção devem ser adquiridos separadamente. Para o presente trabalho, o foco principal será no computador de bordo Titanium e no módulo de piloto automático, os quais serão descritos com mais detalhes nas seções seguintes.

2.3 Computador de bordo Titanium

O computador de bordo Titanium, ver Figura 2.2, é um sistema completo de orientação para agricultura de precisão. A interface com o usuário é moderna e intuitiva, facilitando a configuração e utilização do sistema. Sua tela principal é bastante semelhante a um dispositivo de GPS convencional, com o veículo posicionado no centro da tela e uma trajetória pré-definida como guia. Dados referentes ao veículo, tais como distância entre eixos e posição da antena devem ser configurados pelo operador, de forma a garantir precisão na operação.

FIGURA 2.2: Computador de bordo Titanium, modelo Ti7.
(Fonte: Hexagon Agriculture)

2.3.1 Visão geral

O modelo Ti7 do Titanium conta com uma tela LCD de 7"sensível ao toque para interface com o usuário, interfaces de comunicação via CAN, USB *(Universal Serial Bus)* e RS232, além de módulo GNSS, WiFi e 3G/4G. Além disso, é necessária a presença de uma antena GNSS externa para sincronização com sinais de satélites. A precisão obtida do sistema GNSS pode chegar a até 2cm no caso do uso de tecnologias de posicionamento mais avançadas, tais como RTK *(Real Time Kinematics)*.

O sistema pode ser alimentado a partir da bateria da máquina agrícola, tipicamente com uma tensão de 12V. Além disso, dados de campo podem ser importados através de interface USB, podendo ser gerados relatórios simples e avançados de mapeamento de dados e aplicações.

2.3.2 Características técnicas

O processador utilizado pelo Titanium é um ARM® CortexTM-A9 com clock de 800MHz e 1GB de DDR3 RAM, embarcado em uma placa de desenvolvimento com vários periféricos. Essa placa está posicionada num *socket* disponível na ATBB, placa de circuito impresso desenvolvida pela empresa contendo os circuitos de alimentação, periféricos, tais como acelerômetro, giroscópio, *buzzer*, módulos GSM *(Global System for Mobile Communication)* e GNSS, além de uma diversidade de circuitos integrados e de condicionamento de sinais responsáveis por fazer o tratamento de sinais de entrada e saída da placa e das interfaces de comunicação.

A grande disponibilidade de componentes faz com que o sistema tenha grande possibilidade de expansão e adição de novas funcionalidades, porém sem perder a flexibilidade inerente à um sistema embarcado, visando o melhor desempenho possível para cada tipo de aplicação.

2.4 Módulo de piloto automático

O operador pode definir uma trajetória a ser seguida, iniciando num ponto A e terminando num ponto B, podendo esta ser reta, curva ou circular. A partir disso, o Titanium irá gerar trajetórias paralelas como referência e então a função de barra de luzes, que pode ser vista na parte central superior da Figura 2.2, passará a funcionar.

A barra de luzes indica em tempo real se a trajetória seguida pelo operador está de acordo com a trajetória pré-definida, caso negativo, ela então orienta para que lado o operador deve esterçar o volante de modo a voltar a operar sobre a linha guia. A automatização desse processo, para garantia de maior desempenho, pode ser feita através do piloto automático.

2.4.1 Visão geral

O piloto automático representa um passo além da barra de luzes na orientação geográfica de equipamentos agrícolas. Trata-se de uma ferramenta moderna que visa a otimização da aplicação de corretivos e fertilizantes, bem como operações de plantio e colheita. Além disso, o operador poderá atentar para outras ações importantes durante a aplicação, como a verificação da quantidade de insumos no depósito e o monitoramento do funcionamento de todo o sistema de aplicação.

O sistema funciona por meio do direcionamento automático da trajetória do veículo agrícola através de monitoramento via satélite. Um dos problemas é a operação em terrenos inclinados, onde há a tendência de o veículo sofrer grandes deslocamentos laterais devido ao ângulo de rodagem. Com a utilização de um sistema de navegação inercial, utilizando-se de acelerômetros e giroscópios em conjunto com os dados obtidos via satélite, esses efeitos podem ser drasticamente reduzidos.

Em suma, o objetivo final do módulo de piloto automático é o seguimento de uma trajetória pré-estabelecida com o menor erro possível, de forma a obter maior eficiência operacional.

2.4.2 Modo de funcionamento

O piloto automático da empresa possui a configuração mostrada na Figura 2.3. De maneira simplificada, o fluxo de execução é o seguinte: o Titanium comunica-se com os sinais de satélite através da antena para obter a posição geográfica e a velocidade do veículo, e então envia essas informações para o Driver via rede CAN.

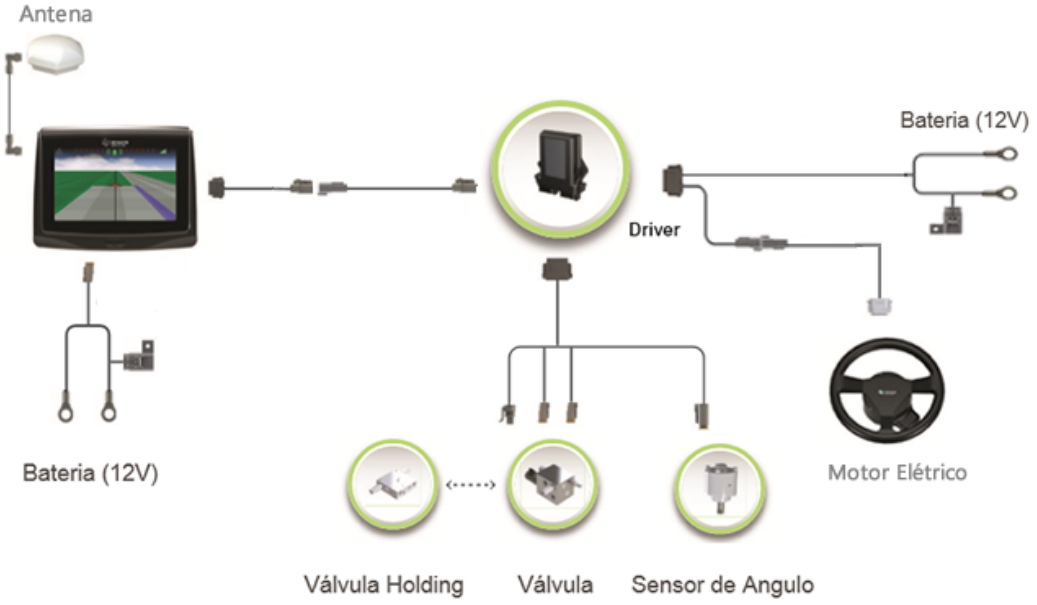


FIGURA 2.3: Módulo de piloto automático da Hexagon Agriculture.
(Fonte: Hexagon Agriculture)

Simultaneamente, o Driver faz a leitura de acelerômetro e giroscópio, e então executa algoritmos para cálculo de orientação do veículo (ângulos de *roll*, *pitch* e *yaw*). Essas informações são enviadas novamente para o Titanium via CAN, que retorna o ângulo de referência e o erro referente à trajetória. Logo, o algoritmo de controle de trajetória passa a executar no Driver, acionando posteriormente os atuadores para correção do ângulo da roda com relação à linha guia.

O atuador pode ser elétrico ou hidráulico. O atuador hidráulico pode ser acionado pelo Driver através da alimentação fornecida pelo Titanium. Já o atuador elétrico necessita de uma potência mais elevada, sendo necessária a ligação direta na bateria para que o

Driver possa fornecer a corrente necessária para o acionamento. Há também a utilização de um sensor de roda para obter a posição da roda com relação ao eixo longitudinal do veículo. Alguns parâmetros de controle podem ser alterados *in loco*, de forma a aumentar ou diminuir a agressividad, bem como modificar parâmetros relacionados ao *overshoot* e a rapidez com que o veículo converge para a trajetória de referência.

Detalhes a respeito do algoritmo de controle de trajetória e do cálculo de orientação serão dados nos Capítulos 3 e 4. Por ora, é importante ter em mente o fluxo de execução do módulo de piloto automático e o papel que o Titanium e o Driver exercem no sistema, bem como a maneira com que eles interagem entre si.

Capítulo 3

Conceitos básicos

Este capítulo visa introduzir alguns dos conceitos básicos necessários para o entendimento deste trabalho. Muitos desses conceitos serão abordados e discutidos mais profundamente no Capítulo 4, que trata do desenvolvimento da solução. Os conceitos serão tratados na ordem que segue: visão geral sobre sistemas Linux embarcados, sistemas de tempo real, protocolos de comunicação entre processos e entre dispositivos e, por fim, sistemas de navegação inercial e teoria de controle de trajetória no contexto do piloto automático.

3.1 Sistemas Linux embarcados

A primeira versão do Linux foi lançada em 1991 e podia ser acessada somente através de código-fonte por pessoas suficientemente capacitadas. O fato de o código-fonte estar disponível abertamente permitiu, substancialmente, que uma vasta gama de entusiastas pudesse contribuir para formar a primeira distribuição de *software*. Ao longo dos anos, o sistema operacional Linux foi ganhando popularidade e hoje pode ser encontrado operando desde os menores e mais simples *gadgets* até os mais complexos supercomputadores, com o suporte e contribuição de uma grande comunidade de desenvolvedores. [7]

Dentre suas principais vantagens podemos citar o fato de ser *open-source*, o que faz com que não seja necessário pagar uma licença para poder usá-lo, além de possuir suporte gratuito através de fóruns e comunidades de desenvolvedores. Outro fator fundamental é

sua versatilidade: devido ao fato de ser *open-source*, seu código fonte pode ser adaptado para as diferentes plataformas de *hardware* sob as quais o sistema irá operar.

A definição de sistemas embarcados (do inglês *Embedded Systems*) é bastante ampla, mas pode-se dizer, de maneira geral, que um sistema embarcado é um sistema computacional com forte integração entre *hardware* e *software*, projetado para desempenhar uma ou mais funções no contexto de uma aplicação específica. Nesses sistemas, recursos como memória, poder de processamento, consumo e espaço em disco podem ser fatores limitantes, o que faz com que grande parte do esforço dispendido no desenvolvimento de um sistema embarcado vise a otimização da utilização de recursos, porém sem comprometer o desempenho do sistema como um todo.

Nos últimos anos, a utilização de Linux em sistemas embarcados vem crescendo significativamente. A natureza *open-source* do Linux significa que os desenvolvedores que trabalham com sistemas embarcados podem tirar vantagem de um aprimoramento contínuo em um ambiente de código aberto, que acontece numa velocidade que nenhuma outra empresa de *software* privada pôde atingir. [8]

3.1.1 Linux kernel

Traduzido do inglês como núcleo, o *kernel* é a parte do sistema operacional responsável pela interface entre a plataforma de *hardware* utilizada no sistema embarcado e as aplicações desenvolvidas pelo usuário. Ele permite e gerencia o acesso de múltiplas aplicações aos recursos do sistema, tais como CPU, memória, rede, disco e interfaces de entrada e saída. Um diagrama simplificado da função do *kernel* no sistema pode ser visto na Figura 3.1.

Dessa forma, o *kernel* abstrai os níveis mais baixos da interação com o *hardware* através de chamadas de sistema. Ou seja, sempre que uma aplicação necessita acessar um recurso de *hardware*, é necessário enviar uma notificação para o *kernel*, para que ele então possa gerenciar o acesso, por muitas vezes concorrente, entre as diversas aplicações. Geralmente, essas chamadas de sistema são feitas através de bibliotecas C, não sendo muito comum que uma aplicação interaja diretamente com o *kernel*. Para que as aplicações possam trocar informações, é utilizado um sistema de arquivos virtual

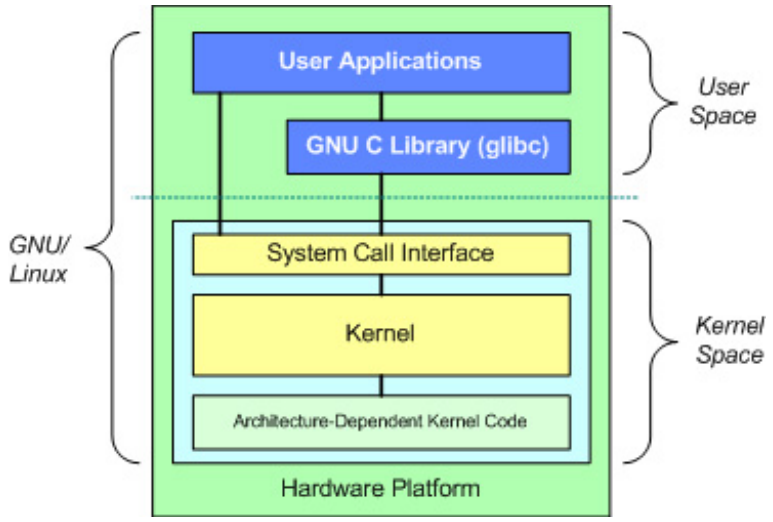


FIGURA 3.1: Interação do *kernel* com as camadas do sistema
(Fonte: https://www.ibm.com/developerworks/library/l-linux-kernel/)

que não é armazenado diretamente no disco, mas sim criado e atualizado virtualmente pelo próprio *kernel* durante sua utilização.

O *kernel* do Linux possui inúmeras versões e permite grande flexibilidade na configuração de suas interfaces e características, podendo se adequar às mais diferentes plataformas. Neste projeto, a versão do *kernel* utilizada é a 3.10.53, sendo atualizada posteriormente para a versão 3.14.52, devido à maior disponibilidade de serviços, principalmente no que se refere à comunicação em rede e sem fio.

3.1.2 O projeto Yocto

O projeto Yocto é um projeto de colaboração de *software* livre que fornece modelos, ferramentas e métodos que suportam sistemas customizados baseados em Linux para produtos integrados, independentemente da arquitetura de hardware.

A cadeia de ferramentas — um compilador, assembler, *linker* e outros utilitários binários necessários para criar arquivos executáveis binários para uma determinada arquitetura — é um item necessário para todos os sistemas de criação. O Poky, sistema de desenvolvimento de referência para o projeto Yocto, usa a GNU Compiler Collection (GCC), mas é possível especificar outras cadeias de ferramentas. O Poky usa uma técnica conhecida como compilação cruzada: o uso de uma cadeia de ferramentas em uma arquitetura para desenvolver arquivos executáveis binários para outra arquitetura (por exemplo, desenvolver uma distribuição ARM em um sistema baseado em x86).

Os desenvolvedores frequentemente usam a compilação cruzada no desenvolvimento de sistemas embarcados para aproveitar o desempenho mais alto do sistema *host*.[9]

A Hexagon Agriculture utiliza-se do projeto Yocto para para gerar uma distribuição Linux compatível e totalmente funcional visando a arquitetura da placa de desenvolvimento utilizada, como comentado no Capítulo 2.

3.2 Sistemas de tempo real

A definição de um sistema de tempo real não está relacionada unicamente com a rapidez com que um sistema responde a um determinado evento, mas sim com o cumprimento de uma determinada tarefa dentro de um período de tempo limitado e previsível. Isso significa que a resposta do sistema dependerá não somente da integridade lógica das operações, mas também será importante estabelecer se ela cumpriu ou não o prazo estipulado para execução.

Diferente dos sistemas de propósito geral, em que usualmente busca-se a execução de várias aplicações simultaneamente, assegurando que cada uma delas possua um tempo de processamento, os sistemas de tempo real geralmente priorizam as tarefas consideradas mais críticas. Dessa forma, os sistemas de tempo real podem ser classificados em duas diferentes categorias: sistemas de tempo real *soft* e sistemas de tempo real *hard*.

Os sistemas de tempo real *soft* são aqueles que possuem uma certa flexibilidade com relação ao *deadline* da tarefa. Como exemplo, pode-se citar um sistema de entretenimento com interface gráfica, no qual a perda de um *deadline* irá afetar apenas o tempo de resposta percebido pelo usuário, porém sem causar nenhum dano crítico com relação à integridade do sistema ou até mesmo das pessoas envolvidas. [10]

Em contrapartida, em um sistema de tempo real *hard* a perda de um *deadline* pode trazer consequências catastróficas, não sendo de maneira alguma tolerável. Como exemplo, podemos citar um sistema de controle de uma aeronave, em que a demora demasiada para enviar os sinais de controle às turbinas pode acarretar em um acidente fatal envolvendo a integridade física não só do sistema, mas de um grande número de pessoas.

Os conceitos apresentados a seguir estão relacionados com sistemas operacionais de maneira geral, mas são de fundamental importância na construção e entendimento de um sistema de tempo real. Além disso, os exemplos apresentados estarão relacionados com a estrutura fornecida pelo Linux, podendo ter implementações diferentes em outros sistemas.

3.2.1 *Threads* e processos

Enquanto executa um programa de usuário, um computador também pode estar lendo um disco e gerando saída de texto em uma tela ou impressora. Em um sistema com multiprogramação, a CPU também alterna de um programa para o outro, executando cada um deles por uma determinada fração de tempo. Rigorosamente falando, o processador pode executar apenas um programa por vez, porém durante um segundo ele pode trabalhar em vários programas, dando aos usuários a ilusão de paralelismo. Assim, todo *software* executável no computador é organizado em um ou mais processos. [11]

Por outro lado, as *threads* representam uma nova concepção, de maneira que um processo determina o momento de sua criação, visando paralelizar a execução de partes do código. Apesar de serem muito semelhantes à processos, elas não possuem identidade própria, estando sempre associadas a um processo pai e compartilhando o mesmo espaço de endereçamento.

De maneira geral, tanto *threads* quanto processos competem pela execução em sistemas *single-core* e possuem três estados principais: em espera, executando e bloqueado. A partir de agora, os conceitos se aplicarão tanto a *threads* quanto a processos, e ambos serão chamados apenas de processos, a menos que seja feita uma clara distinção entre os dois.

Quando em espera, o processo está pronto para executar, porém o processador está momentaneamente ocupado com uma tarefa de maior prioridade. Para que o processo seja bloqueado, algumas situações podem ter ocorrido: o processo solicitou um recurso que não está disponível, solicitou esperar até que um dado evento fosse completado ou entrou em estado de espera por um determinado período. Para que o processo entre em execução, é necessário que este seja o processo de maior prioridade na fila de processos em espera e, além disso, esteja pronto para execução.

3.2.1.1 Chaveamento de contexto

Cada processo tem seu próprio contexto, que é o estado dos registradores em que se encontra antes de ser escalonado pelo processador. Assim, uma troca (ou chaveamento) de contexto ocorre toda vez que o escalonador troca um processo em execução por outro. Toda vez que um processo é criado, é criado também um bloco de controle associado a esse processo, contendo tudo que o *kernel* precisa saber sobre o processo em questão.

Assim, toda vez que um novo processo é escalonado, há um tempo associado ao chaveamento de contexto entre o processo que parou de executar e o que será colocado em execução. Esse tempo geralmente é insignificante, porém, caso a aplicação exija uma grande quantidade de interrupções na sua execução, esse tempo poderá influenciar significativamente o desempenho do sistema.[10]

3.2.1.2 Prioridades

O escalonador geralmente possui algoritmos específicos para determinar qual processo deve executar em um dado momento. Dois dos algoritmos mais populares são o escalonamento baseado em prioridades e o escalonamento por fatia de tempo. Uma combinação dos dois algoritmos também é possível e bastante comum.

O escalonamento baseado em prioridades parte do princípio de que cada processo possui uma prioridade associada, e que processos de maior prioridade devem ter preferência sobre processos de menor prioridade. O funcionamento desse algoritmo pode ser visto na Figura 3.2(a). Nesse contexto, o processo P1, de menor prioridade, começa executando, até que é preemptado por P2, com prioridade maior. Por sua vez, P3 se torna apto a executar e então faz com que P2 seja preemptado, executando até completar sua tarefa. Só então P2 volta a executar de onde parou, até o seu término, devolvendo a possibilidade de execução à P1, de menor prioridade.[10]

A Figura 3.2(b), por outro lado, exemplifica o funcionamento do algoritmo de escalonamento híbrido, combinando as duas técnicas. Nesse caso, processos de mesma prioridade são escalonados a partir de uma fatia de tempo específica dada a cada processo para execução. No caso de um processo de maior prioridade se tornar apto a executar, o processo que estava executando é bloqueado, liberando o processador para que o processo de maior prioridade possa realizar sua tarefa.

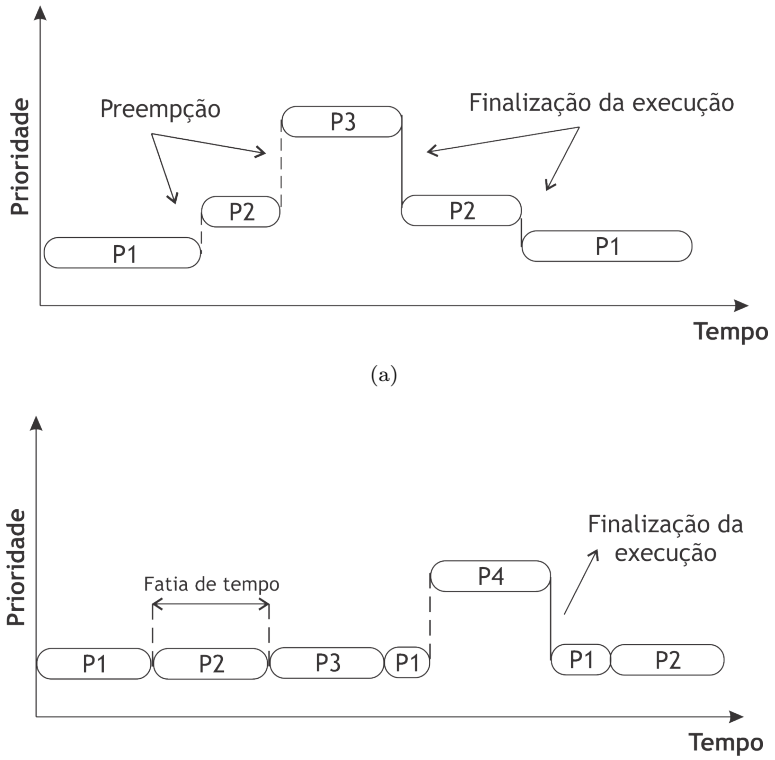


FIGURA 3.2: Funcionamento dos algoritmos de escalonamento.
(Fonte: Autor)

O uso de prioridades em processos é uma ferramenta fundamental no desenvolvimento de sistemas de tempo real, uma vez que dessa forma é possível garantir que processos mais críticos não sejam postergados indefinidamente por processos de menor importância. Porém, deve-se atentar para que os processos de maior prioridade não utilizem uma carga de processamento tão elevada a ponto de nunca permitir que processos de prioridades menores executem, prejudicando assim o funcionamento do sistema como um todo.

3.2.2 Exceções e interrupções

Outro mecanismo bastante importante fornecido pela maioria das arquiteturas em sistemas embarcados são as exceções e interrupções. Esses eventos funcionam de modo a interromper o fluxo de execução de um programa e podem ser acionados de três maneiras diferentes: intencionalmente pela própria aplicação, devido a um erro ou condição incomum do sistema ou através de algum evento externo.

No contexto deste projeto, as interrupções externas se manifestam, por exemplo, através de sinais oriundos dos canais de comunicação via CAN, sempre que uma mensagem é recebida de um dispositivo externo, ou através dos sensores *hall* para medição de posição do motor elétrico do piloto automático, que geram pulsos a uma frequência proporcional ao deslocamento angular do motor. Já no caso das exceções, estas podem ser geradas no caso de uma divisão por zero, ou uma instrução inapropriada e também interrompem o fluxo de execução.

No momento em que a interrupção ocorre, podendo ela ser externa ou interna através de uma excessão no sistema, o processador desvia o fluxo de execução para a rotina de tratamento de interrupção, salvando o contexto do processo anterior à interrupção, e voltando a executá-lo assim que a rotina de tratamento for completada.

As interrupções externas, assim como os processos, podem ser ordenadas por prioridade, de forma a garantir que as interrupções mais críticas tenham preferência na ordem de execução. Além disso, algumas interrupções podem ser desativadas por um determinado período de tempo, quando se deseja obter atomicidade na execução de um conjunto de operações. Por se tratar de um mecanismo que afeta o ciclo de execução dos processos, as interrupções e exceções influenciam diretamente no determinismo de uma aplicação de tempo real.

3.2.3 Latência de interrupção

A definição de latência está associada a uma diferença de tempo entre o início de um evento ou estímulo, e o momento em que seus efeitos tornam-se perceptíveis no sistema. No caso da latência de interrupção, pode-se definí-la como o tempo que decorre do momento em que uma interrupção externa é gerada até o momento em que ela de fato começa a executar sua rotina de tratamento de interrupção.

Esse é um aspecto bastante importante em um *kernel* com performance de tempo real, uma vez que o mesmo deve garantir que a interrupção seja tratada dentro de um limite de tempo pré-estabelecido. A Figura 3.3 exemplifica esse cenário, onde a latência de interrupção está representada por T_L e o tempo de execução da rotina de tratamento da interrupção está representado por T_E.

Nesse caso genérico, o processo que está sendo executado é interrompido por um evento de interrupção externo e o processador então passa a fazer o deslocamento do ponteiro de execução para a seção de código referente ao tratamento da interrupção em questão. Esse chaveamento de contexto engloba a rotina de serviço de interrupção do *kernel*, latência e tempo de execução do escalonador, podendo-se dizer que T_L é a soma de todos esses valores. Só então a rotina de interrupção passa a executar até que seja finalizada, devolvendo ao processo a prioridade de execução. Em casos mais práticos, é possível que haja uma nova interrupção no momento em que a rotina de tratamento de interrupção está sendo executada. Geralmente, quando este tipo de situação ocorre, há um novo chaveamento de contexto, caso a interrupção seja de maior prioridade, criando o que se chama de interrupções em cadeia.

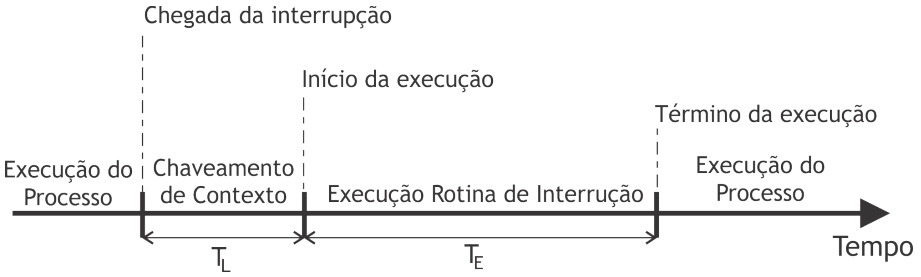


FIGURA 3.3: Latências e tempo de execução da rotina de interrupção externa. (Fonte: Autor)

Para exemplificar o caso acima, considera-se que uma tarefa periódica crítica deve executar um conjunto de instruções a cada intervalo de tempo ΔT. Se $T_L + T_E > \Delta T$, então a tarefa não cumprirá seus requisitos temporais.

Pode-se dizer então que o não-determinismo ocorre quando não é possível assegurar que o tempo decorrido entre o estímulo e a resposta não se enquadram dentro de um limite de tempo ΔT, afetando a capacidade de resposta do sistema e podendo ocasionar em falhas intoleráveis no caso de sistemas de tempo real críticos.

3.2.4 Temporizadores de alta resolução

Como uma ferramenta adicional dependente do *hardware* em que se está trabalhando, os sistemas de tempo real fornecem acesso a temporizadores de alta resolução, usualmente com precisão de nanosegundos. A resolução dos temporizadores está intimamente relacionada com a resolução do *tick* do sistema que, em sistemas de propósito geral, girava em torno de valores como 100Hz, 250Hz ou 1KHz.

Dessa forma não era possível, em termos práticos, usar temporizadores com precisões maiores que $1ms$, o que afetava consideravelmente a temporização exata de malhas de controle com períodos menores que a resolução mínima ou que não fossem múltiplos dela. Com a implementação dos temporizadores de alta resolução, foi possível obter grande precisão em chamadas de sistema como *usleep* no Linux, bastante usada para temporizar tarefas que precisam ser executadas periodicamente, e também no cálculo de unidades de tempo dentro do programa.

3.3 Mecanismos de comunicação

O processamento descentralizado, seja ele dividido entre diferentes processos ou *threads* sob a mesma plataforma, ou até mesmo em plataformas diferentes, abre a necessidade de criação de um canal de comunicação para troca de dados entre as aplicações. Para que uma ou mais aplicações possam se comunicar é necessário que seja definido não só um canal físico de transferência de dados, mas também um protocolo comum de comunicação entre elas com algumas regras que devem ser seguidas para que consigam interpretar corretamente as informações trocadas.

Nesta seção serão abordados alguns protocolos de comunicação entre processos, bem como protocolos de comunicação entre dispositivos, de forma a apresentar conceitos e informações relevantes para o entendimento tanto do fluxo de informação que ocorre no sistema, quanto das soluções adotadas no Capítulo 4.

3.3.1 Comunicação entre processos

A comunicação entre processos, também conhecida como IPC (*Inter-Process Communication)*, tem por objetivo a troca de informações ou coordenação de atividades entre processos executando sobre o mesmo sistema operacional. Neste documento será dada maior ênfase aos protocolos implementados na plataforma Linux, porém muitos deles estão disponíveis também em outras plataformas.

3.3.1.1 D-Bus

D-Bus é um mecanismo de comunicação entre processos e chamada de procedimento remoto, originalmente desenvolvido para Linux. Seu objetivo é o de substituir soluções IPC existentes e concorrentes, em um protocolo unificado. Ele também tem sido projetado para permitir comunicação entre processos a nível de sistema (como serviços de drivers de *hardware* e impressoras) e de processos do usuário normal. Ele usa um protocolo rápido de passagem de mensagens, que é adequado para comunicações internas na máquina. Isso se deve à baixa latência e *overhead*. [12]

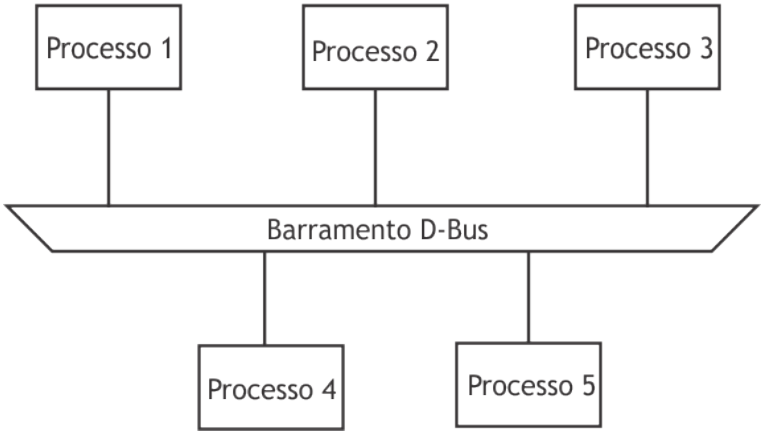


FIGURA 3.4: Protocolo D-Bus implementado através de um barramento de comunicação.
(Fonte: Autor)

A comunicação geralmente ocorre através de uma aplicação de servidor central, chamada de barramento. No entanto, a comunicação aplicação-aplicação direta também é possível. Quando comunicando em um barramento, aplicações podem perguntar quais outras aplicações e serviços estão disponíveis, bem com ativá-las sob demanda. [12]

O fato de existir um barramento, exemplificado pela Figura 3.4, faz com que a estrutura se torne bastante escalável, fazendo uso de um servidor central responsável por manter o controle das aplicações que estão conectadas a esse barramento. No Titanium, a maioria das mensagens e invocação de procedimentos remotos são feitas através do D-Bus, devido à necessidade de comunicação entre vários pontos.

3.3.1.2 Message Queue

Message Queue, também conhecido como fila de mensagens, é um protocolo que permite que dois ou mais processos troquem dados em formatos de mensagens, de forma assíncrona, em que o processo remetente e o processo destinatário não precisam interagir ao mesmo tempo. As mensagens ficam armazenadas numa espécie de *buffer* gerenciado pelo próprio *kernel*, o qual associa uma única identificação para cada fila de mensagens. [13]

O tamanho da fila, bem como o tamanho de cada mensagem, devem ser especificados no momento da criação do objeto e possuem um valor fixo. Para que a fila de mensagens exista, é necessário que um processo faça sua criação. Uma vez criada, outros processos podem acessá-la através de sua identificação única no sistema. Isso significa que vários processos podem ler e escrever ao mesmo tempo.[13]

Além disso, é possível associar prioridades às mensagens no momento do envio, de forma que mensagens de prioridade maior são colocadas no início da fila, e, portanto, serão as primeiras a serem retiradas. Este serviço também possibilita que processos sejam bloqueados caso o processo deseje enviar uma mensagem e a fila esteja cheia, ou o processo queira receber uma mensagem e a fila esteja vazia. Dessa forma, este mecanismo pode também ser facilmente usado para sincronização entre processos.[13]

3.3.1.3 Memória compartilhada

Uma maneira bastante eficiente de trocar dados entre processos é através de memória compartilhada, uma vez que a velocidade de acesso consiste basicamente no tempo de acesso de uma região de memória pré-definida. Essa região precisa ser previamente declarada por um processo, para que o *kernel* saiba que aquela será uma região de memória compartilhada e crie os mecanismos de acesso necessários. A Figura 3.5 mostra como essa região é compartilhada entre os processos, de forma que cada processo possui um mapeamento da região compartilhada dentro de sua própria memória.[14]

Este é certamente o mecanismo de comunicação entre processos mais veloz, porém há algumas limitações. O acesso simultâneo de dois processos na mesma seção de memória

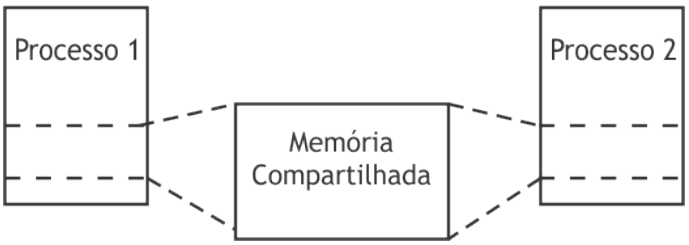


FIGURA 3.5: Representação do mecanismo de comunicação através de memória compartilhada.
(Fonte: Autor)

compartilhada, com um processo escrevendo dados na memória enquanto o outro lê, pode acarretar uma leitura inadequada caso não haja um mecanismo de sincronização de acesso definido.

Além disso, é necessário ter bastante cautela na implementação, uma vez que o acesso através de um ponteiro a um endereço de memória fora da região pode acarretar falhas severas no programa.

3.3.1.4 Pipe

Um mecanismo bastante conhecido e utilizado para comunicação entre processos são os chamados *Pipes*. Esse mecanismo se assemelha bastante e uma fila do tipo FIFO (*First-in, First-out*), assíncrona e unidirecional. Isto significa que para que haja uma comunicação nos dois sentidos, é necessária a criação de dois *pipes* distintos, um para leitura e outro para escrita. [14]

Inicialmente, somente poderiam ser criados *pipes* para comunicação entre processos relacionados, ou seja, processos gerados a partir de um processo pai. Porém, atualmente essa limitação não mais existe com a criação dos *Named Pipes*, os quais possuem funcionalidade similar aos *pipes* tradicionais, mas podem ser acessados por processos que não são relacionados, desde que eles conheçam seu descritor.

3.3.2 Comunicação entre dispositivos

O uso de sensores, atuadores, periféricos e controladores inteligentes em conjunto com um sistema de processamento central geralmente envolve uma necessidade de comunicação entre eles, seja ela para fazer a leitura de um sinal de um sensor, envio e

recebimento de sinais de controle ou simplesmente uma troca de mensagens entre ambos dispositivos.

Atualmente, existem vários protocolos de comunicação disponíveis, proprietários ou não, para que dois ou mais dispositivos, estejam eles sobre a mesma plataforma ou em módulos separados, possam se comunicar. Neste documento, serão abordados três protocolos principais: *Inter-Integrated Circuit* (I^2C), *Serial-Peripheral Interface* (SPI) e *Controller Area Network* (CAN).

3.3.2.1 I^2C

I^2C é um protocolo serial síncrono que utiliza apenas dois fios para conectar dispositivos de baixa velocidade, como microcontroladores, EEPROMs, conversores A/D, interfaces de entrada e saída, e outros periféricos em sistemas embarcados. O protocolo foi inventado pela Philips e agora é usado por quase todos os principais fabricantes de circuitos integrados. O protocolo I^2C ganhou popularidade por ser simples de usar. Apenas a velocidade máxima do barramento precisa ser definida e são utilizados somente dois fios com resistores *pull-up* para conexão de um número quase ilimitado de dispositivos.[15]

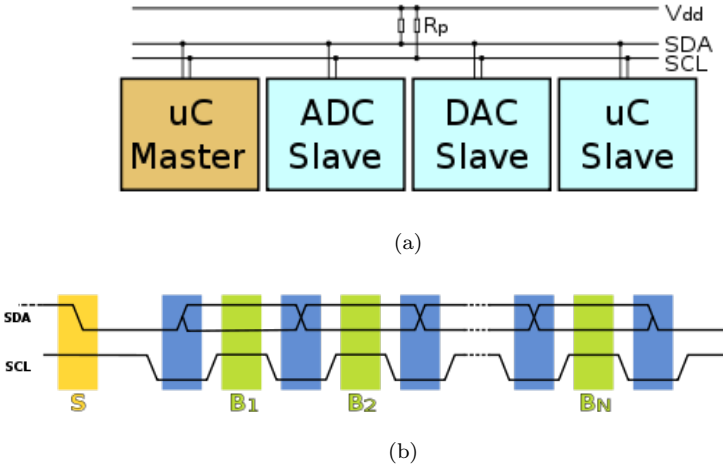


FIGURA 3.6: Funcionamento do protocolo I^2C e esquema de ligação entre os dispositivos
(Fonte: http://i2c.info/)

Como pode ser visto na Figura 3.6(b), a transmissão é iniciada pelo dispositivo mestre através de uma transição na linha SDA. O byte $B1$ consiste num endereço de 7 bits que identifica o dispositivo ao qual o mestre está querendo se comunicar, com o

oitavo bit representando se é uma operação de leitura ou escrita. Os bytes seguintes $B2, B3, ..., Bn$ são os bytes lidos ou enviados para o dispositivo, seguidos de um bit de parada, sinalizando o final da transmissão.

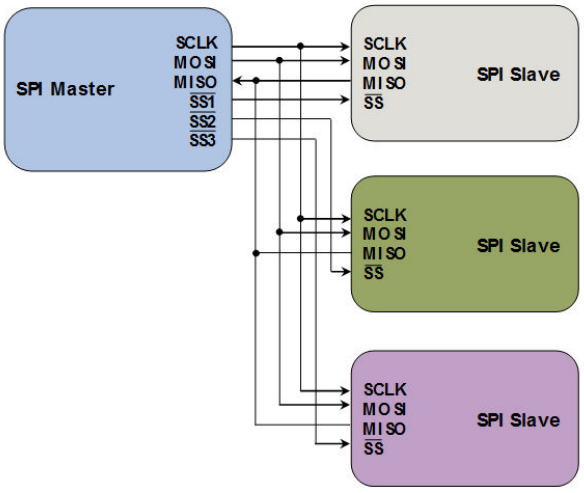
Cada dispositivo escravo numa rede I^2C tem um endereço único. A transferência de dados de e para o dispositivo mestre é feita serialmente e dividida em pacotes de 8 bits. As especificações iniciais deste protocolo definiam uma frequência máxima de clock de 100 kHz. Esta frequência foi posteriormente aumentada para 400 kHz. Atualmente, existem alguns dispositivos que suportam também um modo de alta velocidade que pode chegar a até 5MHz.[15]

3.3.2.2 SPI

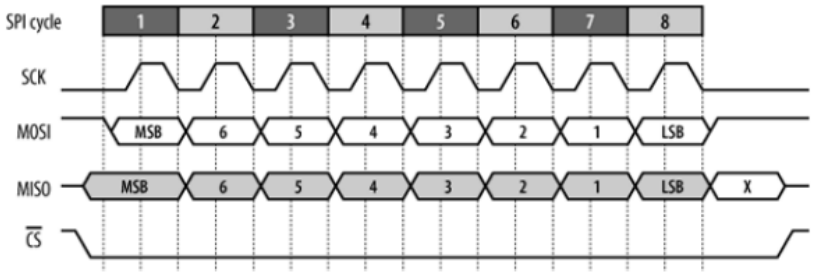
O protocolo SPI foi desenvolvido pela Motorola com intuito de fornecer uma interface simples e de baixo custo entre microcontroladores e periféricos. Diferentemente de uma porta serial padrão, SPI é um protocolo síncrono em que todas as transmissões são feitas com base em um clock de referência, gerado pelo dispositivo mestre. Para selecionar o escravo ao qual deseja se comunicar, o mestre deve ativar seu pino seletor, de forma que os outros dispositivos que por ventura estiverem conectados no mesmo barramento não participem dessa interação.

O protocolo SPI utiliza quatro sinais principais: *Master Out Slave In* (MOSI), *Master In Slave Out* (MISO), *Serial Clock* (SCLK) e *Chip Select* ou *Slave Select* (CS, SS). Na Figura 3.7(a) é possível ver como os dispositivos são conectados ao barramento SPI. Tanto o mestre quanto os escravos possuem um registrador de deslocamento associado às linhas MOSI e MISO, de forma que ao mesmo tempo em que os bits são transmitidos do mestre para o escravo através do sinal MOSI, eles são recebidos de volta pelo sinal MISO. Isso faz com que as operações de leitura e escrita possam ser efetuadas simultaneamente, tornando o protocolo bastante eficiente.[16]

Dessa forma, caso o mestre queira apenas ler um byte do dispositivo, ele irá, inevitavelmente, transferir um *dummy* byte para o escravo. Analogamente, caso o mestre queira apenas enviar um byte para o escravo, irá receber de volta um byte, devendo simplesmente ignorá-lo. [16]



(a)



(b)

FIGURA 3.7: Funcionamento do protocolo SPI e esquema de ligação entre os dispositivos.
(Fonte: http://www.byteparadigm.com/applications/introduction-to-i2c-and-spi-protocols/)

Na Figura 3.7(b) pode-se verificar o funcionamento do protocolo. Alguns dispositivos suportam a transferência de múltiplos bytes sem que a transmissão seja interrompida. Quando esse comportamento for desejado, deve-se manter o pino de *chip select* em nível baixo até o final da transmissão. Usualmente, o primeiro byte a ser transmitido pelo mestre contém o endereço do registrador ou a posição de memória associada ao conteúdo que será lido ou escrito.

Alguns dos pontos positivos associados ao protocolo SPI são: comunicação *full-duplex*, interface de *hardware* bastante simples, possibilidade de transmissão de uma grande quantidade de dados sem interrupção e alta frequência de transmissão, usualmente na casa dos megahertz.

3.3.2.3 CAN

O CAN é um protocolo de comunicação que permite controle distribuído em tempo real, com elevado nível de segurança, bastante utilizado em sistemas automotivos. É um sistema em barramento com capacidades multi-mestre, isto é, vários nós podem pedir acesso ao meio de transmissão simultaneamente. Este protocolo comporta também o conceito de *multicast*, isto é, permite que uma mensagem seja transmitida a um conjunto de receptores.[17]

Nas redes CAN não existe o endereçamento dos destinatários no sentido convencional, em vez disso são transmitidas mensagens que possuem um determinado identificador. Assim, um emissor envia uma mensagem a todos os nós e cada um, por seu lado, decide, com base no identificador recebido, se deve ou não processar a mensagem. O identificador determina também a prioridade intrínseca da mensagem, ao competir com outras pelo acesso ao barramento. A informação transmitida possui tamanho curto. Assim, cada mensagem CAN pode conter um máximo de 8 bytes de informação útil, sendo no entanto possível transmitir blocos maiores de dados recorrendo a segmentação.[17]

A Figura 3.8 mostra o funcionamento do protocolo de transmissão. Os dados são transmitidos através do par diferencial *CAN HI* e *CAN LO*, sempre fazendo com que ambos os sinais sejam opostos, representado o bit 1 quando em nível de tensão baixo e o bit 0 quando em nível de tensão alto. O pacote de dados de uma única mensagem possui um cabeçalho bastante extenso, devido principalmente aos mecanismos de endereçamento e validação das mensagens.

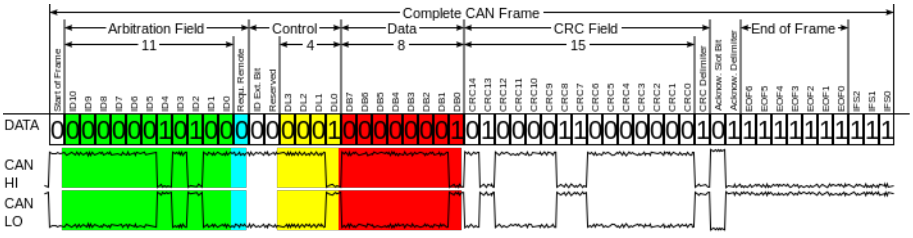


FIGURA 3.8: Protocolo de comunicação da rede CAN
(Fonte: https://en.wikipedia.org/wiki/CAN_bus)

A taxa máxima de transmissão especificada é de 1 Mbit/s, correspondendo este valor a sistemas com comprimento de barramento de até 40m. Para distâncias superiores, a taxa de transmissão diminui. O número de elementos num sistema CAN está, teoricamente,

limitado pelo número possível de identificadores diferentes. Este número limite é, no entanto, significativamente reduzido por limitações físicas do *hardware*. [17]

O protocolo CAN permite flexibilidade uma vez que podem ser adicionados novos nós a uma rede CAN sem requerer alterações do *software* ou *hardware* dos outros nós, se o novo nó não for emissor, ou se o novo nó não necessitar da transmissão de dados adicionais. Outra característica importante é o fato de o controlador CAN de cada estação registrar os erros, avaliando-os estatisticamente, de forma a desencadear ações com eles relacionadas. Estas ações podem corresponder ao desligar, ou não, da estação que provoca os erros, tornando este protocolo eficaz em ambientes ruidosos. [17]

3.4 Controle de trajetória

A função principal do módulo de piloto automático, como comentado nas seções anteriores, é fazer com que o veículo, autonomamente, siga uma trajetória pré-definida. Para tanto, é preciso que os módulos de controle estejam cientes da posição do veículo com relação a um determinado referencial, de forma que possam atuar no sentido de minimizar o erro de seguimento da trajetória.

Nesta seção serão explorados alguns conceitos relacionados ao controle de trajetória. Dentre eles estão os algoritmos de controle utilizados, uma breve descrição do sistema de geração de trajetórias, conceitos relacionados aos sistemas de navegação, tal como módulo GNSS, além de sensores como acelerômetro e giroscópio. Também será abordada brevemente nessa seção uma explicação sobre métodos de filtragem e filtro de Kalman, para integração de dados de vários sensores distintos visando a estimação de um vetor de estados para os ângulos de *roll*, *pitch* e *yaw*.

3.4.1 Algoritmo de controle

O algoritmo utilizado para controle de trajetória é derivado de [18]. Primeiramente, será mostrado o modelo cinemático de uma máquina agrícola genérica e então apresentada a lei de controle. De um ponto de vista prático, o trator e, possivelmente, o implemento que esteja acoplado ao seu eixo traseiro, podem ser modelados como um

triciclo de comprimento l, conforme Figura 3.9, onde as rodas do eixo dianteiro são as rodas que podem ser controladas.

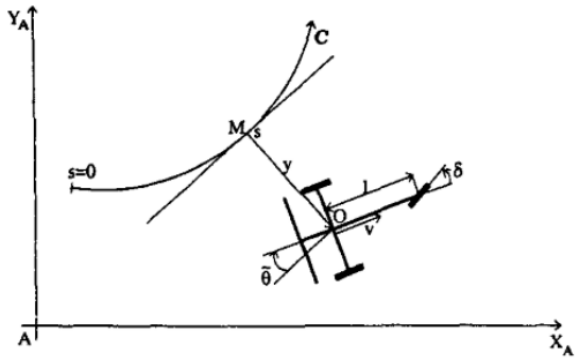


FIGURA 3.9: Modelo cinemático da máquina agrícola

A configuração descrita diz respeito a uma trajetória C a ser seguida pela máquina agrícola, onde O representa o centro do eixo traseiro e M o ponto em C cuja distância é a mais próxima de O. Dessa forma, a configuração do veículo pode ser descrita pelo vetor de estados $X = (s, y, \tilde{\theta})^T$, e as variáveis de controle disponíveis podem ser descritas pelo vetor $U = (v, \delta)^T$, onde cada uma das variáveis representa:

- s : a abscissa curvilinear do ponto M ao longo de C;
- y : o desvio lateral do trator com relação à trajetória C;
- $\tilde{\theta}$: o desvio angular do trator com relação à tangente de C, calculada no ponto M;
- v : velocidade linear no ponto O;
- δ : orientação da roda dianteira.

Duas hipóteses principais foram consideradas para que fosse desenvolvida a lei de controle em questão. Primeiro, que a máquina agrícola e o implemento são um corpo rígido. A segunda, é de que a máquina agrícola move-se horizontalmente sem escorregamento. Dessa forma, tem-se as seguintes equações que representam a dinâmica do sistema:

$$\dot{s} = v\frac{\cos\tilde{\theta}}{1 - c(s)y} \tag{3.1}$$

$$\dot{y} = v\sin\tilde{\theta} \tag{3.2}$$

$$\dot{\tilde{\theta}} = v(\frac{\tan\delta}{l} - \frac{c(s)\cos\tilde{\theta}}{1 - c(s)y}) \tag{3.3}$$

O objetivo da lei de controle é assegurar a convergência do veículo para a trajetória de referência. Ou seja, deseja-se que y e $\tilde{\theta}$ sejam iguais a zero. Além disso, a performance da lei de controle não deve depender da velocidade v. Como as equações de estados apresentadas configuram um sistema não-linear, o autor, em [18], propõe que seja feita uma linearização, transformando o modelo numa aproximação. Esses cálculos fogem do escopo deste trabalho e, portanto, será apresentada apenas a lei de controle obtida, dada pela Equação 3.4 que está implementada no Driver e foi transferida para a plataforma Linux de tempo real:

$$\begin{aligned}\delta(y,\tilde{\theta}) = arctan\Bigg\{\Bigg(l\Bigg[\frac{\cos^3\tilde{\theta}}{(1-c(s)y)^2}\Bigg(\frac{dc(s)}{d(s)}y\tan\tilde{\theta} - K_d(1-c(s)y)\tan\tilde{\theta} \\ -K_p y + c(s)(1-c(s)y)\tan^2\tilde{\theta}\Bigg) + \frac{c(s)\cos\tilde{\theta}}{1-c(s)y} + 1\Bigg]\Bigg\}\end{aligned} \tag{3.4}$$

Para que a lei de controle acima funcione corretamente, é preciso uma medição, ou estimação, em tempo real das variáveis controladas y e $\tilde{\theta}$. O valor de y pode ser facilmente calculado, sabendo-se a posição absoluta do veículo no ponto O, obtida através do GPS, e a trajetória de referência. Em contrapartida, a estimação da variável $\tilde{\theta}$ pode se tornar bastante complexa devido aos ruídos de medição e falta de precisão absoluta dos métodos utilizados. É com foco nesse ponto que serão tratadas as seções seguintes.

3.4.2 Geração de trajetórias

No Titanium, a definição de uma trajetória de referência é feita pelo operador, selecionando o modo de operação através do painel *touchscreen*. As trajetórias possíveis para o piloto automático podem ser vistas na Figura 3.10. Ao escolher uma trajetória, o operador deve configurar o ponto de partida e definir a rota até o ponto final. Assim, o

Titanium fica responsável por identificar a trajetória de referência e atualizá-la conforme o veículo avança sobre o terreno.

FIGURA 3.10: Trajetórias possíveis para o módulo de barra de luzes e piloto automático. (Fonte: Hexagon Agriculture)

Todo esse processamento serve de entrada para o módulo de controle e os dados são enviados atualmente via CAN do Titanium para o Driver, responsável pelo controle de trajetória.

3.4.3 Sistema de navegação

Grande parte das pesquisas relacionadas à implementação de piloto automático em máquinas agrícolas tem como foco principal a utilização de câmeras como elemento sensor principal. Uma das dificuldades é a falta de uma referência fixa, outra é a existência de condições de pouca visibilidade, tal como poeira, neblina e, eventualmente, falta de luz.

O sistema utilizado nesse projeto, em contrapartida, utiliza um referencial absoluto, através de dados de GNSS em conjunto com medidas obtidas de um acelerômetro e um giroscópio. Estes últimos são sensores de baixo custo, confiáveis para curtas distâncias, fazendo uso de propriedades físicas e matemáticas para obter a orientação do veículo através da aceleração e velocidade ângular em cada um dos três eixos de rotação. A configuração do sistema de coordenadas utilizado segue a convenção *North-East-Down*, também chamada de NED, conforme pode ser visto na Figura 3.11.

Como visto nas seções anteriores, é preciso uma estimação precisa do ângulo do veículo com relação à trajetória de referência e do desvio lateral y para que se possa calcular a lei de controle. Caso esses valores não sejam confiáveis, toda a malha de controle pode ser prejudicada, podendo inclusive ocorrer falhas severas no sistema.

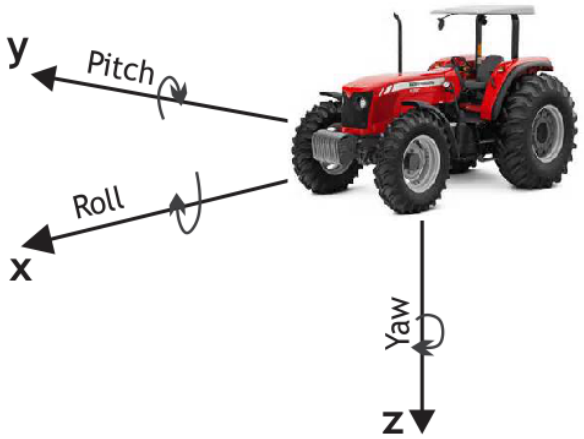


FIGURA 3.11: Ângulos de *roll*, *pitch* e *yaw* com relação ao sistema de coordenas NED. (Fonte: Autor)

3.4.3.1 Acelerômetro

Acelerômetros são dispositivos eletrônicos capazes de medir tanto acelerações lineares quanto a aceleração provocada pelo campo gravitacional terrestre. Assim, caso o acelerômetro esteja sujeito unicamente à aceleração do campo gravitacional terrestre, a soma vetorial de seus três eixos indicará uma aceleração de $1g$. Com essa informação, pode-se calcular os ângulos de *roll* e *pitch* com relação ao seu sistema de referência a partir de matrizes de rotação em x, y e z.

A incapacidade de se obter o ângulo de *yaw* a partir do acelerômetro ocorre porque a rotação em torno do eixo Z está inicialmente alinhada ao eixo gravitacional da Terra, o que faz com que o acelerômetro não apresente nenhuma variação quando rotações ao longo do eixo Z são aplicadas.

Por exemplo, caso a saída do acelerômetro indique *1g* na direção Z, considerando que o eixo Z é ortogonal ao plano terrestre, então os ângulos de *roll* e *pitch* serão iguais a zero. Porém, caso a aceleração gravitacional esteja distribuída entre dois ou mais eixos, pode-se dizer que esses ângulos serão diferentes de zero. Esse princípio é fundamental para que se possa determinar a inclinação de um corpo rígido com relação a um sistema de coordenadas de referência.

Os ângulos de *roll* (ϕ), o qual representa uma rotação em torno do eixo X, e *pitch* (θ), o qual representa uma rotação em torno do eixo Y, podem ser obtidos através da

Equação 3.5[19], para uma sequência de rotação dada por $y - x - z$, onde G_x, G_y e G_z representam a componente de aceleração presente nos eixos X, Y e Z, respectivamente.

$$\phi = \arctan\left(\frac{G_y}{\sqrt{G_y^2 + G_z^2}}\right) \qquad \theta = \arctan\left(\frac{-G_x}{G_z}\right) \tag{3.5}$$

No entanto, é inapropriado utilizar os dados brutos do acelerômetro, uma vez que um veículo agrícola em operação apresentará, inevitavelmente, acelerações diferentes de zero quando em movimento. Para que os cálculos de orientação do veículo sejam mais exatos, é necessária uma fusão de sensores, extraindo o melhor das características estáticas e dinâmicas que cada um apresenta para se obter uma estimação mais próxima possível da realidade.

3.4.3.2 Giroscópio

Giroscópios são dispositivos eletrônicos capazes de medir velocidade angular. Sabe-se que a velocidade angular de um corpo está relacionada com sua posição angular, de forma que ela representa a variação da posição angular ao longo do tempo, dado por $\dot{\theta} = \frac{d\theta}{dt}$. Ou seja, para que se obtenha a posição angular, basta que seja feita a integração da velocidade angular ao longo de um determinado intervalo de tempo.

A Equação 3.6 representa o cálculo utilizado para realizar esse procedimento, onde $\theta(t)$ representa a posição angular, θ_0 a posição angular inicial e $\dot{\theta}(t)$ a velocidade angular.

$$\theta(t) = \theta_0 + \int_0^t \dot{\theta}(t)dt \tag{3.6}$$

Assim, pode-se calcular os ângulos de *roll*, *pitch* e *yaw* simplesmente integrando os valores lidos pelo giroscópio ao longo do tempo em cada um dos eixos de rotação. Porém, como os sistemas computacionais são sistemas discretos, deve-se transformar a integral em um somatório e o tempo em um período de amostragem. Toda vez que os dados lidos pelo giroscópio se alteram a uma frequência maior do que o período de amostragem, será

gerado um acúmulo de erros, fazendo com que a aproximação se distancie cada vez mais do valor real.

Além disso, métodos de filtragem eficientes devem ser empregados para eliminar ao máximo possível o ruído associado na medição, caso contrário a integração será ainda menos confiável. Usualmente, não deve-se confiar nos dados de integração a partir de um certo tempo, uma vez que os erros acumulados podem ser muito maiores do que o valor do ângulo em si. Porém, as características dinâmicas do giroscópio podem ser extremamente úteis se utilizadas em conjunto com os dados do acelerômetro, como será visto na seção 3.4.3.4.

3.4.3.3 GNSS

A navegação por satélite tem se tornado parte fundamental no desenvolvimento de aplicações que requerem conhecimento da posição do corpo com relação a um sistema de referência fixo, de forma rápida, precisa e eficiente.

Geralmente a mobilidade é o foco de tais aplicações, assim como ocorre com o projeto em questão, em que é preciso determinar em tempo real a posição do veículo agrícola na lavoura com relação ao sistema de coordenadas terrestre. Isso é possível através do *Global Navigation Satellite System* (GNSS), o qual é composto por uma constelação de satélites utilizados para se comunicar com uma rede de estações fixas e dispositivos móveis. Atualmente, há dois sistemas principais em operação: o sistema americano GPS (*Global Positioning System*) e o sistema russo GLONASS (*Global Orbiting Navigation Satellite System*).

Estes sistemas utilizam a técnica de triangulação para obter a posição desejada. Cada satélite transmite sinais codificados periodicamente e o receptor converte essa informação em posição, velocidade e uma estimativa de tempo. Usando essas informações, um receptor na superfície terrestre pode calcular a exata posição do satélite e a distância entre ele e o receptor. Coordenando os dados recebidos de 4 ou mais satélites, é possível determinar a posição do receptor com uma exatidão que pode chegar na casa dos centímetros para os sistemas mais avançados.

A informação transmitida pelo módulo GNSS ao módulo de controle, em um sistema eletrônico, geralmente é recebida em *frames*, contendo dados como latitude, longitude,

velocidade, altitude, dentre outros. A informação de latitude e longitude é tipicamente um número decimal com uma quantidade n de casas decimais, onde a primeira casa decimal após a vírgula representa uma posição com exatidão dentro de um raio de 111km. A cada casa decimal adicionada, a exatidão é aumentada em 10 vezes. Portanto, é necessário que o dispositivo sendo utilizado forneça uma resolução adequada, que seja compatível com a aplicação.

3.4.3.4 Integração dos dados

Como comentado nas seções anteriores, o módulo GNSS fornece uma posição absoluta com relação a um referencial terrestre. Isso faz com que seja possível determinar a trajetória do corpo ao longo do tempo. Além disso, é importante também estimar de que forma o veículo está variando sua trajetória a partir daquele ponto. Isso pode ser determinado a partir do cálculo de sua orientação, principalmente do ângulo de *yaw*.

Foi visto que o acelerômetro possui características estáticas bastante interessantes, de onde pode-se extrair os ângulos de *roll* e *pitch* diretamente das leituras em cada um dos eixos, considerando unicamente a ação da força gravitacional. Por outro lado, o giroscópio tem boas características dinâmicas, porém a integração da velocidade angular tende a desviar do valor real ao longo do tempo, devido a ruídos e acúmulo de erros de integração.

Para contornar esses problemas, métodos de filtragem e estimação utilizando um filtro de Kalman se mostram apropriados. Estimação é definida como um método para obter um conjunto único de valores para um conjunto desconhecido de parâmetros, a partir de um conjunto de observações. Para calcular a estimação dos estados, uma relação funcional entre os parâmetros desconhecidos e as quantidades observadas deve ser estabelecida. O filtro de Kalman é um exemplo de estimador que explora as duas informações, a do conhecimento da dinâmica do processo e a relação entre os estados e as medições, para fornecer um estado ótimo do sistema.[20]

Dessa forma, é possível combinar as estimações de ângulo feitas tanto pelo acelerômetro quanto pelo giroscópio, bem como modelar os ruídos de processo e de medição para obter uma estimativa confiável do ângulo real. Os dados do módulo GNSS também fazem parte do processo de compensação e estimação, uma vez que a partir da velocidade

do veículo lida pelo módulo GNSS pode-se obter a aceleração através da derivada em um dado instante de tempo. Essa informação serve para compensar as acelerações lineares presentes no veículo quando em movimento, de forma a restar apenas a aceleração causada pela força gravitacional.

Os conceitos envolvendo a implementação do filtro de Kalman e estimação dos estados são bastante complexos e fogem do escopo deste trabalho, sendo suficiente saber que os dados do módulo INS (*Inertial Navigation System*), composto pelo acelerômetro e giroscópio, são integrados com os dados oriundos do módulo GNSS para obter uma estimação mais exata da orientação e posição real do veículo e garantir o bom funcionamento da malha de controle.

Capítulo 4

Projeto e implementação do sistema

Esse capítulo tem por finalidade apresentar e discutir as soluções propostas para atingir os objetivos mencionados no Capítulo 1, bem como as técnicas utilizadas e os problemas encontrados ao longo das etapas de desenvolvimento. A elaboração de uma solução que resolvesse o problema apresentado consistiu fundamentalmente nas seguintes macro etapas:

1. Definição da arquitetura proposta para o sistema;
2. Estudo sobre a necessidade de uma plataforma de tempo real para atender às especificações temporais do sistema;
3. Leitura de acelerômetro e giroscópio sobre a plataforma Titanium e estimação da orientação do veículo em tempo real;
4. Implementação de um canal de comunicação entre a aplicação do Titanium e o módulo do piloto automático;
5. Desacoplamento entre os módulos implementados no Driver para que seja mantida a compatibilidade entre as duas plataformas e outras que por ventura possam vir a surgir;
6. Desenvolvimento de uma interface para cada submódulo do sistema, visando a troca de dados entre eles;

7. Importação e adequação do sistema de controle de trajetória, juntamente com o módulo de estimação e a interface dos sensores e atuadores para operar sobre a plataforma Titanium.

Todas as etapas acima foram implementadas pelo autor deste relatório, salvo algumas exceções que estarão explícitas no texto, e serão abordadas com maiores detalhes ao longo deste capítulo, dando ênfase às soluções escolhidas, discutindo alternativas e evidenciando os problemas encontrados. As etapas de testes parciais para cada uma das implementações serão tratadas nesta seção, no entanto a validação final do sistema será abordada no Capítulo 5, que trata dos resultados atingidos a partir da solução proposta.

4.1 Definição da arquitetura

O primeiro passo no desenvolvimento do projeto foi o entendimento da arquitetura atual do sistema, envolvendo o Driver, o Titanium e suas interfaces de *hardware* com os sensores e atuadores. Como mencionado no Capítulo 1, a proposta é migrar de uma plataforma complementar microcontrolada responsável pelo controle de trajetória do módulo de piloto automático para a plataforma do Titanium, a qual possui um processador operando um sistema operacional Linux para sistemas embarcados. Para tanto, foi preciso definir uma arquitetura que fornecesse os elementos necessários para o funcionamento do sistema, tal como interface com sensores e atuadores, canais de comunicação entre as aplicações, interface de *hardware* com a placa, além da garantia de funcionamento lógico e temporal do módulo de piloto automático.

4.1.1 Arquitetura original do sistema

O módulo do piloto automático foi apresentado de maneira geral no Capítulo 2, através da Figura 2.3. Nessa seção será dada ênfase a um modelo mais aprofundado da arquitetura do sistema para que se possa compreender melhor o fluxo de informação envolvendo o módulo do piloto automático, a aplicação do Titanium e seus periféricos.

A Figura 4.1 mostra a relação do Driver com o Titanium e as interfaces de *hardware* disponíveis. O Driver é uma plataforma microcontrolada montada em uma placa de circuito impresso desenvolvida pela empresa que contém um processador ARM® Cortex

M4, um circuito dedicado para acionamento de motor trifásico, acelerômetro, giroscópio e portas de entrada e saída para sensores e atuadores, respectivamente, além de um chip de memória externo ao processador. Para se comunicar com outros dispositivos, a placa possui uma porta de comunicação CAN, uma porta serial e uma interface JTAG para facilitar o uso de ferramentas de depuração.

Os sensores externos podem ser tanto encoders incrementais para medir a posição do volante, quanto sensores de roda para medir sua posição angular com relação ao eixo longitudinal da máquina agrícola. Os atuadores podem ser uma válvula ou um motor elétrico, dependendo da configuração do piloto.

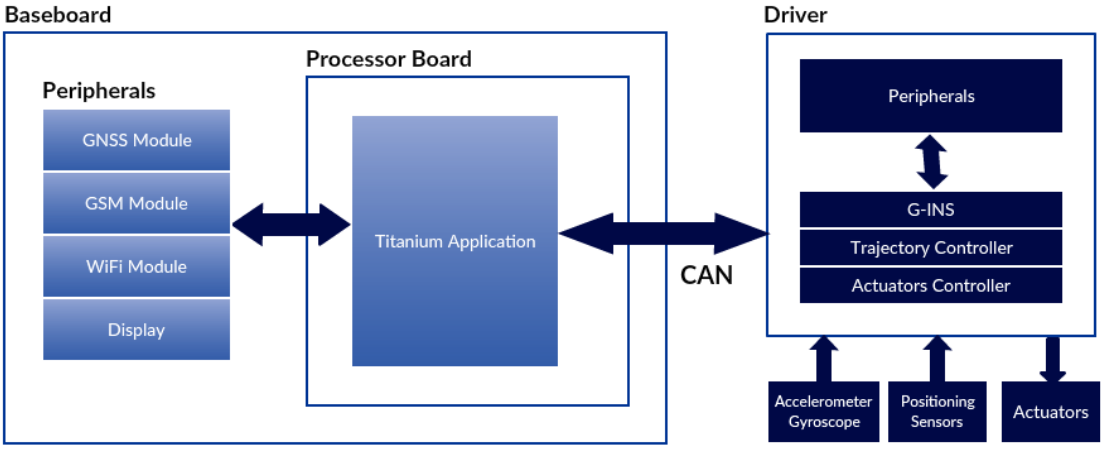


FIGURA 4.1: Arquitetura do sistema original do piloto automático. (Fonte: Autor)

Os módulos G-INS, Trajectory Controller e Actuators Controller estão implementados em *software* numa aplicação que roda sobre o microcontrolador. O módulo G-INS *(GPS-Aided Inertial Measurement System)* é responsável pela leitura do acelerômetro e giroscópio e estimação dos ângulos de *roll*, *pitch* e *yaw* através de um filtro de Kalman que combina os dados obtidos de ambos os sensores. Para uma melhor precisão na estimação, são utilizados os dados oriundos da aplicação do Titanium, que faz a leitura das informações do módulo GNSS, retornando a velocidade, que será então derivada para compensar os efeitos da aceleração linear na estimação do acelerômetro, e a orientação do veículo, a qual é uma medida bastante ruidosa. A diferença angular entre a trajetória de referência e o eixo longitudinal do veículo, $\tilde{\theta}$, e a distância do veículo em relação à trajetória de referência, y, também são enviadas do Titanium para o Driver através da rede CAN.

O módulo Trajectory Controller é responsável por rodar o algoritmo de controle de trajetória, bem como uma rotina de segurança para verificar se os módulos e periféricos estão funcionando corretamente e, caso não estejam, enviar comandos para a aplicação do Titanium, de forma a acionar alarmes para o operador, indicando na tela onde a falha está ocorrendo. A saída do módulo de controle de trajetória é basicamente o ângulo de referência, δ_{ref}, para os atuadores, que será o parâmetro de entrada do módulo Actuators Controller.

A partir dessa informação o módulo Actuators Controller está apto para enviar os comandos necessários aos atuadores, bem como ler a posição da roda de forma direta, ou medindo a posição do volante e fazendo uma relação linear entre a posição angular do volante e a posição da roda. Como visto anteriormente, o sistema de controle funciona de maneira a fornecer uma posição angular de referência, que será então utilizada pela malha de controle dos atuadores.

Entre os módulos presentes no Driver, o módulo G-INS executa a uma frequência de 200Hz, enquanto que o módulo Trajectory Controller executa a 50Hz. Já a frequência de execução do módulo Actuators Controller depende de qual atuador está sendo utilizado, sendo de 2kHz para o atuador elétrico e 20Hz para o atuador hidráulico, além das leituras dos sensores serem ativadas através de interrupções externas.

Por outro lado, a plataforma contendo a aplicação do Titanium é composta de uma *baseboard* e da placa de processamento, sobre a qual está rodando um sistema operacional Linux para sistemas embarcados. Essa placa é responsável por toda a interface de *hardware* com a *baseboard*.

A aplicação do Titanium possui uma extensa gama de funcionalidades, porém como o foco deste trabalho é o módulo de piloto automático, elas não serão abordadas com mais detalhes. Os módulos WiFi e GSM são utilizados para comunicação externa com centrais de recebimento de dados. Já o módulo GNSS é responsável por comunicar-se com sistemas de navegação via satélite, de forma a obter a orientação geográfica do veículo, bem como sua velocidade. O display *touchscreen* serve para receber os comandos do operador, que pode configurar o sistema em tempo real. Esses comandos são processados e interpretados e então enviados ao Driver via rede CAN, caso seja necessário alterar alguma configuração referente ao módulo de piloto automático.

Para se compreender melhor a interação entre cada um dos módulos, pode-se visualizar a Figura 4.2. Nela está uma representação mais alto nível da malha de controle do módulo de piloto automático e do fluxo de informações que são trocadas entre os módulos internos do Driver e também as informações que são trocadas externamente com a aplicação do Titanium via rede CAN. Neste diagrama não estão representadas as mensagens de configuração, somente as mensagens referentes à malha de controle. Além do ângulo de referência e da velocidade, o Driver também envia para o Titanium mensagens de *feedback* de alguns parâmetros de controle. O algoritmo de controle utilizado é o mesmo apresentado na seção 3.4.1.

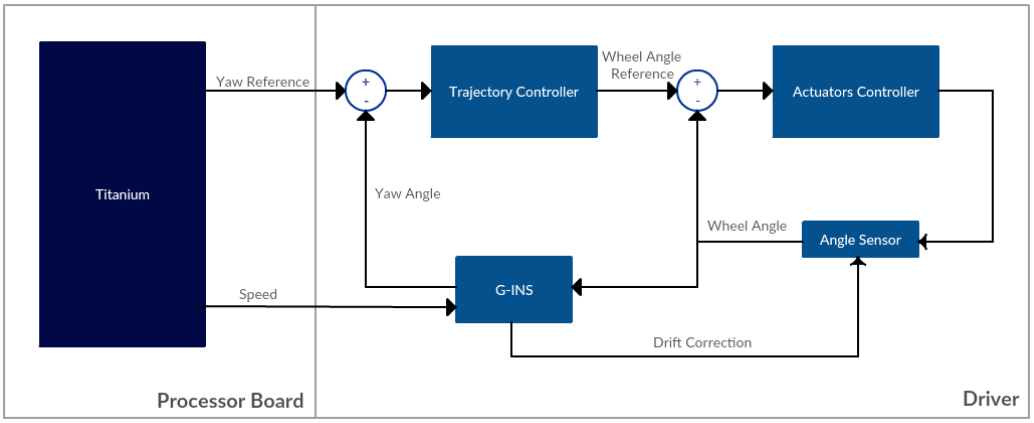


FIGURA 4.2: Malha de controle referente ao módulo de piloto automático. (Fonte: Autor)

Apesar de receber a velocidade do veículo periodicamente a uma frequência de 20Hz (determinada pelo módulo GNSS), o módulo G-INS realiza suas operações a uma frequência de 200Hz para evitar que a estimação da orientação do veículo seja degradada pelo período de amostragem. A correção do *drift* é calculada a partir de alguns dos parâmetros de controle e serve para compensar efeitos como escorregamento mecânico, bem como a relação entre o volante e a roda, que altera-se continuamente ao longo do tempo.

No diagrama da Figura 4.2 estão explícitos apenas os parâmetros que são enviados e recebidos internamente entre os módulos que são mais relevantes para o sistema de controle. No entanto, vários outros dados são trocados entre os módulos, de forma que o nível de acoplamento entre eles pode ser considerado bastante alto. Essa e outras questões serão abordadas nas seções subsequentes, que irão tratar da arquitetura proposta para que o módulo de controle do piloto automático passe a funcionar sobre a

mesma plataforma da aplicação do Titanium.

4.1.2 Arquitetura proposta

A ideia que norteia este trabalho tem por objetivo a migração do módulo do piloto automático do Driver, plataforma onde atualmente está implementado, para a mesma placa sobre a qual a aplicação do Titanium está executando. Assim, será necessário propor uma nova arquitetura para o sistema, que passará a executar sobre uma plataforma de *hardware* diferente. A forma como a aplicação do Titanium e o módulo do piloto automático irão interagir também sofrerá alterações, uma vez que não existirá mais um canal físico de comunicação, pelo fato de ambas estarem agora executando sobre o mesmo processador.

Além disso, considerando-se um processador *single-core* e cada aplicação sendo executada em um processo separado, ambas terão que competir pelos mesmos recursos, tal como memória, processador, objetos específicos do *kernel* e periféricos. Também será preciso modelar o sistema de forma a garantir um certo determinismo na execução dos módulos de controle, evitando assim comportamentos indesejados e até mesmo acidentes que podem ter resultados catastróficos.

A arquitetura proposta para o novo sistema pode ser vista na Figura 4.3. Agora, como pode ser visto, não há mais a presença do Driver como uma plataforma complementar. Todos os módulos estão concentrados sobre a mesma *baseboard*, que é a plataforma principal do produto Titanium. O canal de comunicação entre a aplicação do Titanium e os periféricos e a própria aplicação do Titanium em si já estão implementadas. Os blocos dos periféricos, bem como acelerômetro, giroscópio, atuadores e sensores de posição, ou seja, componentes de *hardware* do sistema, são dispositivos físicos que não sofreram alterações ao longo do trabalho. Já os blocos restantes (G-INS, Trajectory Controller e Actuators Controller) representam aquilo que será tratado e implementado neste trabalho, sendo estes um canal de comunicação entre processos para trocar informações entre o módulo do piloto automático e a aplicação do Titanium, canais de comunicação I^2C e SPI para interface com o acelerômetro e giroscópio, uma interface de comunicação entre o módulo de controle de trajetória e o módulo de controle dos atuadores e a importação e adequação dos blocos G-INS, Trajectory Controller e Actuators Controller para a nova plataforma.

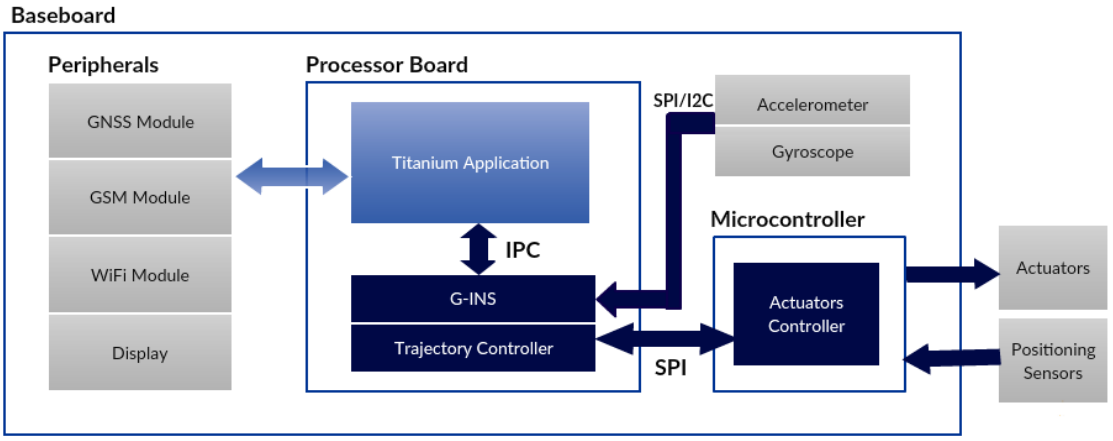


FIGURA 4.3: Proposta de arquitetura para o novo sistema do piloto automático. (Fonte: Autor)

A interface dos periféricos com a aplicação do Titanium continua mantida da mesma forma como era feito anteriormente. No entanto, os módulos presentes no Driver foram separados. O módulo G-INS e o Trajectory Controller fazem parte agora de um único processo que irá executar sobre o mesmo processador da aplicação do Titanium, sendo então necessário um canal de comunicação entre processos entre ambas aplicações.

O módulo Actuators Controller será implementado separadamente em um microcontrolador dedicado de menor porte, devido à necessidade de saídas PWM para acionamento do motor e/ou da válvula. Outro fator que favorece essa abordagem é o período de controle de $400us$, no caso do motor elétrico, e as inúmeras interrupções de *hardware* oriundas dos sensores *hall* para medição da posição angular do volante, fazendo com que um *hardware* dedicado seja mais adequado para essa tarefa.

Outra alteração é a presença de acelerômetro e giroscópio na *baseboard* do Titanium, que serão lidos pelo processo contendo o módulo G-INS, responsável pela estimação da orientação e posição do veículo. A comunicação entre o processo do módulo G-INS e Trajectory Controller com o módulo Actuators Controller será feita através de um protocolo de comunicação *on-board*, tal como o procotolo SPI. Este será de fundamental importância para que o submódulo Trajectory Controller possa enviar o ângulo de referência para a malha de controle dos atuadores e a correção do *drift* do ângulo da roda possa também ser atualizado periodicamente.

O novo modelo proposto elimina a necessidade de utilização de uma placa adicional (nesse caso a do Driver), reduzindo assim os custos do sistema, bem como da instalação

de chicotes elétricos, e permitindo maior centralização dos módulos, que agora estarão todos sobre a mesma plataforma. A eliminação do canal de comunicação via rede CAN entre o módulo de controle de trajetória e a aplicação do Titanium também é uma possível fonte de melhoria de performance temporal, devido ao *overhead* associado ao envio e recebimento de mensagens entre os módulos através da rede.

4.2 Plataforma de tempo real

A arquitetura proposta na seção anterior visa um novo modelo de funcionamento do sistema, que passará agora a executar sobre uma plataforma Linux embarcada. Conforme discutido, o sistema será composto por tarefas responsáveis pelo controle de trajetória do piloto automático, bem como estimação em tempo real da orientação e posição do veículo. Sendo assim, é necessário garantir que o sistema cumprirá seus requisitos temporais, de forma a não perder os *deadlines* associados à execução de tarefas consideradas mais críticas.

Como tarefas críticas no sistema pode-se considerar a estimação da orientação do veículo, uma vez que esse processo está intimamente ligado ao estado atual do veículo e o ângulo de *yaw* serve de entrada ao sistema de controle. Se as iterações, tanto na integração dos dados obtidos do giroscópio, quanto na estimação a partir do acelerômetro e filtro de Kalman, não cumprirem seus requisitos temporais, todo o sistema estará comprometido. Atualmente a malha de estimação opera a uma frequência de 200Hz, sendo que já foi comprovado empiricamente que, quanto menor a frequência da estimação, menor também será a exatidão da orientação estimada. Além disso, a malha de controle de trajetória precisa operar a uma frequência de 20Hz, que é atualmente a frequência de atualização do sinal GNSS, para garantir uma performance aceitável do sistema de controle.

Em ambos os casos, os requisitos temporais não são tão críticos a ponto de a perda de um *deadline* ocasionar uma falha irreversível no sistema. No entanto, à medida em que a perda de *deadlines* se acumula sequencialmente ao longo do tempo, tanto a performance do sistema quanto a segurança das pessoas envolvidas passam a entrar numa zona de risco que não pode ser tolerada.

4.2.1 Requisitos temporais

A partir da discussão feita acima, definiu-se que o requisito temporal para a tarefa de estimação da orientação possui um *deadline* que não pode ser maior que $5ms$, assim como o *deadline* da malha de controle de trajetória não pode ultrapassar os $50ms$. Apesar de eventualmente falhas pontuais no cumprimento desses requisitos serem toleráveis, definiu-se, segundo [1], que o sistema sendo tratado neste trabalho é do tipo *hard real-time*, o que significa que pelo menos 95% das vezes o *deadline* das tarefas de tempo real devem ser atingidos para garantir um nível suficiente de performance e segurança ao sistema.

4.2.2 O sistema Linux

Segundo a arquitetura do módulo de piloto automático definida anteriormente, o módulo será migrado para uma plataforma Linux embarcada, de forma a operar em paralelo juntamente com a aplicação do Titanium. A aplicação possui mais de 5 anos de desenvolvimento utilizando-se a linguagem C++, com versões bastante estáveis e consolidadas, tendo como alvo a plataforma Linux. Sendo assim, é praticamente inviável considerar a utilização de uma outra plataforma que não seja o Linux ou outra linguagem que não seja C++ para a migração do módulo de piloto automático.

Sabe-se que o sistema operacional Linux foi desenhado inicialmente como um sistema *time-sharing*, ou seja, o principal objetivo é otimizar o *throughput* ao mesmo tempo em que se maximiza a utilização de recursos entre os processos, sendo o determinismo temporal uma preocupação secundária. Em contrapartida, sistemas operacionais com características de tempo real priorizam o determinismo, frequentemente diminuindo o *throughput* do sistema. Portanto, determinismo e *throughput* são conceitos inversamente proporcionais quando colocados lado a lado no contexto de sistemas operacionais.

Com isso em mente, foi feito um estudo para verificar o comportamento do sistema operacional Linux quando submetido a requisitos de tempo real. Vários *benchmarkings* estão disponíveis na literatura, indicando a performance temporal de sistemas operacionais Linux de propósito geral sob diferentes óticas (com variações de carga na CPU, acesso a recursos, limitação de memória, dentre outros). Porém, há um certo consenso

no sentido de que o sistema possui diversas características que deterioram o determinismo, gerando latências indesejadas e aleatórias, prejudicando assim o funcionamento correto de aplicações de tempo real.

4.2.3 Alternativas disponíveis

Para contonar esse problema há duas alternativas bastante utilizadas por desenvolvedores que utilizam a plataforma Linux para sistemas de tempo real. A primeira delas é um conjunto de *patches*, chamado de PREEMPT_RT patch, a ser aplicado no *kernel* tradicional do Linux, visando melhorar a performance temporal através de uma série de modificações, porém mantendo a estrutura atual do Linux do ponto de vista das aplicações que irão executar sobre esta plataforma. Este *patch* basicamente transforma o Linux em um *kernel* totalmente preemptível, sendo possível associar prioridades de tempo real aos processos mais críticos, priorizando a execução destes mesmo quando executando seções de código internas ao *kernel* e solucionando problemas como inversão de prioridades, transformando tratadores de interrupção em *threads* que podem ser preemptadas por outras de maior prioridade, permitindo a utilização de temporizadores de alta resolução, dentre outras modificações que visam a diminuição de latências indesejadas. [21]

A segunda alternativa, conhecida como Xenomai, é um *framework* de tempo real que implementa uma espécie de *kernel* secundário responsável por receber as interrupções oriundas das aplicações de tempo real e então tratá-las imediatamente. Caso a interrupção não esteja associada a um processo de tempo real, ela então é transferida ao *kernel* do Linux para ser tratada de forma ordinária, assim como os outros processos concorrentes de prioridade mais baixa. Para que isso seja possível, é implementada uma camada de virtualização dos recursos chamada ADEOS *(Adaptative Domain Environment for Operating Systems)*, que facilita o compartilhamento e uso dos recursos de *hardware* baseado nos conceitos de domínio e canal hierárquico de interrupção. Isso faz com que as interrupções sejam recebidas pelo ADEOS e distribuídas de forma hierárquica, seguindo a ordem de prioridade dos domínios. [22]

4.2.4 Definição da plataforma de tempo real

A metodologia para escolha da plataforma de tempo real a ser utilizada, dentre as duas opções citadas na seção anterior e o próprio Linux tradicional com preempção, foi determinada a partir do fluxograma representado pela Figura 4.4, adaptado de [1].

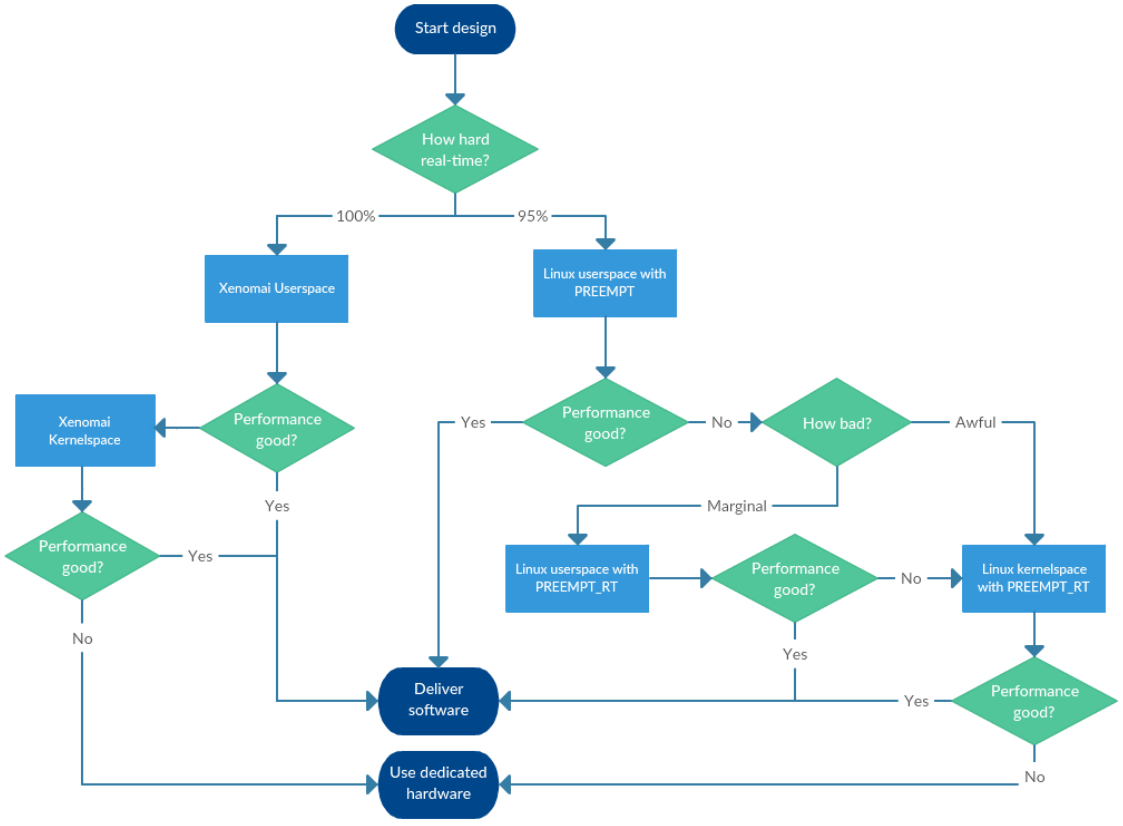


FIGURA 4.4: Fluxograma para determinação da plataforma de tempo real a ser utlizada.
(Fonte: Adaptado de [1])

No fluxograma pode-se identificar 5 opções diferentes de configuração para que os requisitos de tempo real possam ser qualitativamente atingidos. No caso do módulo do piloto automático, já foi definido em seções anteriores que o seu *deadline*, apesar de ser crítico, não configura uma aplicação 100% *hard real-time*, uma vez que a perda de um único *deadline* não implica em uma falha irreversível no sistema. Dessa forma, a escolha seria direcionada para o Linux convencional, podendo ele ser composto pelo *patch* PREEMPT_RT ou não, com a possibilidade, ainda que pouco provável, de que os *deadlines* só sejam cumpridos se a aplicação rodar em *kernelspace*.

Além disso, há outros dois pontos de certa forma mais subjetivos que favorecem ainda mais a utilização do *patch* PREEMPT-RT. O primeiro deles é a possibilidade real de

que esse *patch* passe a fazer parte da linha de desenvolvimento principal do Linux, o que faria com que melhorias e funcionalidades passassem a ser constantemente pensadas e adicionadas. O segundo ponto é que a estrutura principal do *kernel* é mantida, de forma que pouca ou, mais provavelmente, nenhuma alteração precisaria ser feita na aplicação do Titanium para que ela se tornasse compatível com essa nova plataforma.

4.2.5 Validação da plataforma escolhida

Para de fato validar a escolha entre um *kernel* tradicional preemptível e um *patched-kernel* com PREEMPT_RT ativado, foram feitos testes de latência, também chamada nesse caso de *release jitter*, para mensurar o tempo decorrido desde o momento em que uma *thread* de alta prioridade se torna apta a executar até o momento em que ela de fato começa a executar.

Esse tempo pode ser dividido em 4 parcelas distintas, são elas: latência de interrupção, dada por L_I, duração da rotina de serviço de interrupção, dada por L_{DI}, latência do escalonador, dada por L_E e duração do escalonador, dada por L_{DE}. Sendo o tempo de execução dado por T_E e a latência total dada por $L_T = L_I + L_{DI} + L_E + L_{DE}$, para garantir o cumprimento do *deadline*, dado por D, a seguinte relação deve ser respeitada $L_T + T_E < D$. Além disso, caso $L_T + T_E$ seja muito próximo de D, apesar de matematicamente cumprir os requisitos, na prática isso pode ser bastante prejudicial considerando que há outras aplicações de menor prioridade disputando o processador e o tempo hábil restante para que elas executem pode não ser suficiente para garantir um funcionamento adequado do sistema.

Os resultados dos testes de latência podem ser vistos nas Figuras 4.5. Pelos gráficos pode-se perceber que o processo ao qual foi associada uma prioridade de tempo real tem uma latência significativamente menor do que o processo de menor prioridade. Como prioridades de tempo real só podem ser habilitadas quando o *patch* PREEMPT_RT é aplicado, fica claro que há a necessidade de aplicar o *patch* para a aplicação que está sendo tratada neste documento, uma vez que a latência em processos de prioridade normal pode facilmente exceder o *deadline* especificado.

Além disso, o valor máximo quando se utiliza uma prioridade de tempo real gira em torno de $280us$ e a média é de apenas $55us$, podendo ser praticamente desprezível na

maioria das situações. Para reforçar ainda mais a necessidade do *patch* PREEMPT_RT, serão discutidas questões relacionadas ao tempo de execução T_E da tarefa ao longo do documento, de forma a ficar ainda mais visível a necessidade de otimização em cada etapa de desenvolvimento abordada.

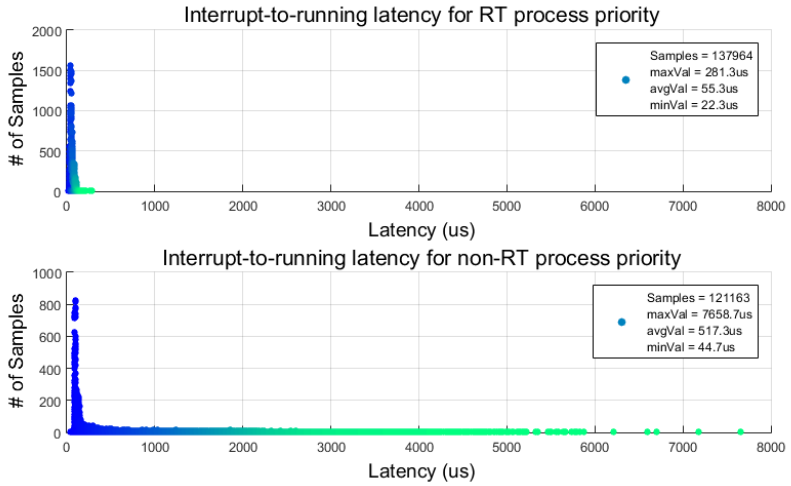


FIGURA 4.5: Testes de latência *(release jitter)* para um processo com prioridade de tempo real, acima, e um processo de prioridade normal, abaixo.
(Fonte: Autor)

Para utilizar essa nova *feature*, foi necessário recompilar o *kernel* do Linux após o *patch* PREEMPT_RT ser aplicado. A versão do *kernel* utilizada neste trabalho foi a 3.10.53, tendo como alvo a plataforma ARM. O *patch* PREEMPT_RT aplicado sobre essa versão do *kernel* foi o 3.10.53-rt56, disponível nos repositórios remotos mantidos pela comunidade Linux através do endereço *kernel.org*. Apesar de terem surgido alguns problemas de compilação, foi possível contorná-los através de correções pontuais em algumas seções do código-fonte.

4.3 Biblioteca para leitura de sensores

Uma vez que a plataforma de tempo real foi definida, o próximo passo do projeto consistiu no desenvolvimento de classes na linguagem C++ para a leitura de acelerômetro e giroscópio, que corresponde ao canal de comunicação I^2C/SPI representado na Figura 4.3, bem como tratamento dos dados de forma a gerar como saída os valores de aceleração em unidades de G, sendo que G corresponde à aceleração da gravidade, dada

por $9,81 m/s^2$, e os valores de velocidade angular em ω, que corresponde a taxa com que a posição angular varia com relação a uma unidade de tempo, dada em $^\circ/s$.

4.3.1 Acelerômetro

O acelerômetro presente na *baseboard* do Titanium é o mesmo utilizado pelo Driver. Esse modelo possui um sensor capacitivo, digital, de baixo consumo, capaz de medir acelerações nos três eixos com uma resolução de até 14 bits por eixo. A escala pode também ser escolhida, variando entre ±2G/±4G/±8G. A comunicação do acelerômetro com o processador central é feita via protocolo I^2C, a uma frequência máxima de 400KHz para o sinal de clock comandado pelo dispositivo mestre.

O Linux contém uma API *(Application Program Interface)* para acessar dispositivos I^2C através do espaço de usuário, desde que o dispositivo esteja fisicamente conectado às portas correspondentes e as mesmas estejam listadas adequadamente na *device tree*. O acesso é feito através de um arquivo de dispositivo que pode ser acessado através de chamadas de sistema padrão do tipo *open*, *read*, *write*, *close* e *ioctl* de forma a abstrair as peculiaridades de cada dispositivo. Por trás dessas chamadas o *kernel* comunica-se com o driver do dispositivo correspondente, que de fato irá realizar as operações de transferência de dados no nível mais baixo da camada de comunicação.

Um conjunto de cerca de 30 registradores está disponível para configuração do modo de operação, frequência de atualização dos dados, configuração de *offset* e *threshold*, geração de interrupções, dentre outros. A maneira como os registradores devem ser acessados para escrita e leitura pode ser visto na Figura 4.6, onde as siglas significam: ST *(Start Bit)*, W *(Write)*, R *(Read)*, SP *(Stop Bit)*, AK *(Acknowledge)*, SR *(Repeated Start Condition* e NAK *(No Acknowledge)*.

Cada dispositivo I^2C possui um endereço de identificação único na rede em que está conectado, de forma que o primeiro byte enviado pelo mestre é sempre o endereço do dispositivo ao qual pretende-se comunicar. Neste trabalho será utilizada a configuração *full-scale* $\pm 4G$ com uma frequência de atualização de 800Hz nos dados de aceleração medidos pelo sensor. Como a resolução dos dados de aceleração medidos é de 14 bits, os dados são armazenados em dois registradores de 8 bits, de forma que a leitura completa dos dados referentes a cada eixo requer a leitura de 2 registradores e a posterior

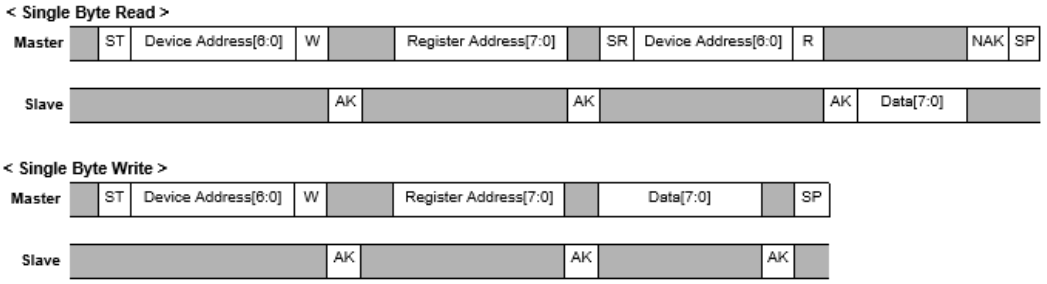


FIGURA 4.6: Operações de escrita e leitura nos registradores do acelerômetro através de protocolo I^2C.
(Fonte: Confidencial)

concatenação dos valores lidos para chegar num valor absoluto que varia entre −8192 a +8192 (para a escala escolhida). A conversão desse valor para a unidade desejada, em G, é feita da seguinte forma:

$$A = \frac{V \times S}{R} \tag{4.1}$$

Na Equação 4.1, A é o valor de aceleração em G, V é o valor concatenado dos dois registradores, e é composto pelo byte mais signficativo concatenado com o byte menos significativo, compondo o valor final de 14 bits, S é a escala e R a resolução. Quanto maior a escala, menor será a resolução, uma vez que a quantidade de bits utilizada para representar os valores lidos se mantém constante.

4.3.2 Giroscópio

O giroscópio presente na *baseboard* do Titanium também é o mesmo disponível na plataforma do Driver. Esse dispositivo é do tipo MEMS *(Micro-Electro-Mechanical Systems)* e possui uma resolução de até 16 bits, em uma escala que varia entre ±250/±500/±2000dps, onde dps significa *degrees per second*. O giroscópio utilizado possui duas interfaces de comunicação disponíveis: I^2C e SPI. Como o protocolo SPI possibilita frequências de comunicação mais elevadas e o barramento I^2C disponível no Titanium já possui um número razoável de dispositivos, optou-se por conectá-lo à interface SPI.

Da mesma forma como acontece com a interface I^2C, o Linux também possui uma API para interface com dispositivos SPI. Apesar de as operações serem um pouco mais limitadas, elas são suficientes para a aplicação em questão. Um grande número de

registradores de 8 bits encontra-se disponível para configuração e, da mesma forma como o acelerômetro, os dados da velocidade angular de cada eixo são armazenados em dois registradores. Há ainda a possibilidade de configuração de filtros internos, de forma a minimizar efeitos indesejados oriundos de ruídos elétricos e vibração mecânica. A forma de acesso aos registradores utilizando o protocolo SPI pode ser visto na Figura 4.7.

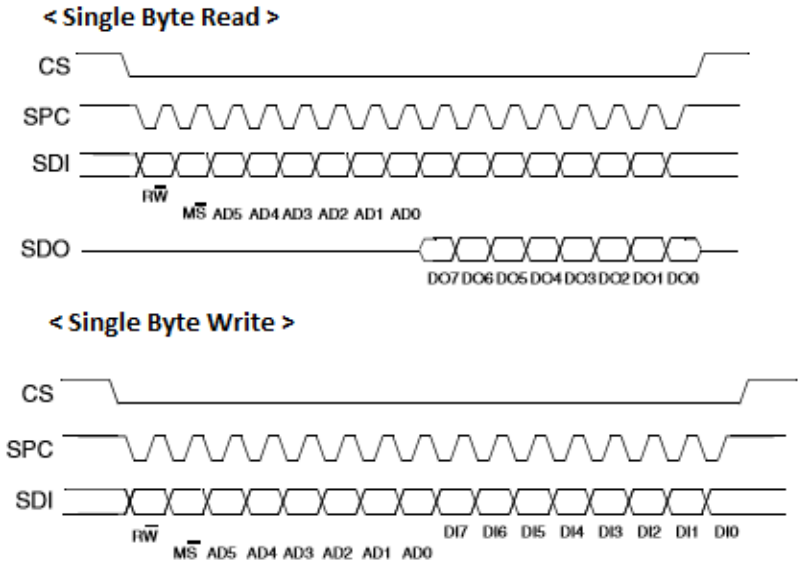


FIGURA 4.7: Operações de escrita e leitura nos registradores do giroscópio através de protocolo SPI.
(Fonte: Confidencial)

Diferentemente do que ocorre no protocolo I^2C, os dispositivos conectados no barramento SPI não necessitam de um endereço único de identificação. Isso acontece porque é o próprio mestre que indica o início de uma transmissão ao dispositivo em questão através do pino CS. A configuração utilizada para o giroscópio é *full-scale* ±250dps, com uma frequência de atualização de 800Hz nos dados. A conversão do valor concatenado obtido pela leitura dos dois registradores de saída segue a mesma regra definida pela Equação 4.1.

4.3.3 Implementação das classes

A implementação da classe do acelerômetro e do giroscópio em C++ deriva de uma classe com métodos virtuais chamada *XYZ Interface* que é comum para ambos. Numa camada mais inferior, foi implementada uma classe genérica para fazer a interface I^2C

(para o acelerômetro) e SPI (para o giroscópio), onde estão implementadas as chamadas de sistema Linux e os métodos de inicialização das interfaces. Dessa forma, os métodos públicos disponíveis pela classe *XYZ Interface* retornam um vetor com os valores de aceleração ou velocidade angular em cada um dos eixos já convertidos para a unidade de interesse, que serão usados pelo módulo de estimação de orientação do veículo.

4.3.4 Validação das bibliotecas

Após a implementação das classes foram feitos testes para verificar a funcionalidade lógica e temporal das implementações. O funcionamento lógico do acelerômetro foi validado comparando-se a saída do sensor com valores conhecidos de aceleração (neste caso a aceleração da gravidade) e rotacionando o sensor em cada um dos eixos para verificar as alterações. Já a validação do giroscópio foi feita rotacionando cada um dos eixos e verificando a saída, porém observando apenas qualitativamente a mudança de direção e velocidade angular, o que já valida a interface de comunicação e leitura dos registradores, uma vez que as etapas de calibração efetivamente serão feitas posteriormente por outros colegas de trabalho e não estão no escopo deste trabalho.

Por outro lado, o funcionamento temporal é de fundamental interesse, uma vez que a leitura desses sensores faz parte da malha de estimação, que deve operar periodicamente a cada $5ms$. Portanto, caso a leitura dos sensores seja demasiadamente longa, todo o processo de estimação será prejudicado. Os primeiros testes indicaram um tempo de leitura bastante elevado, atingindo cerca de $4ms$ para ler ambos os sensores, o que torna praticamente inviável o atendimento do *deadline* e a execução de outros processos em paralelo com uma performance aceitável.

Para minimizar o tempo gasto para leitura dos sensores algumas medidas foram tomadas. Primeiramente, foi verificado que a frequência da I^2C na *device tree* estava configurada para 100KHz. A primeira medida foi alterar para a frequência máxima permitida de 400KHz e verificar se isso não afetaria os outros dispositivos que estavam conectados no mesmo barramento, como a memória *flash*, o painel *touchscreen* e um conversor A/D. Pelo lado do giroscópio, também foi aumentada a frequência do barramento SPI para 10MHz, frequência máxima suportada pelo dispositivo. Além disso, foram feitas implementações visando permitir a leitura dos registradores em modo *burst*, de forma que fosse possível ler os 6 registradores contendo os dados de saída sequencialmente, com

incremento automático dos endereços. Essa alteração foi feita tanto para o giroscópio, quanto para o acelerômetro, pois ambos os dispositivos suportam essa funcionalidade.

Com essas alterações, a velocidade de leitura dos sensores foi aumentada em cerca de 50%. Entretanto, havia um comportamento bastante aleatório no tempo de leitura e processamento dos sensores mesmo definindo o processo responsável pela leitura como sendo de alta prioridade. A razão deste comportamento reside no fato de que os processos responsáveis pelo controle dos *drivers* da I^2C e SPI estavam definidos com baixa prioridade, o que acabava indiretamente ocasionando grandes latências na aplicação de maior prioridade, devido à dependência entre elas. A solução foi então identificar de quais outros processos a aplicação de tempo real depende e então aumentar a prioridade destes processos, de forma a reduzir a latência total.

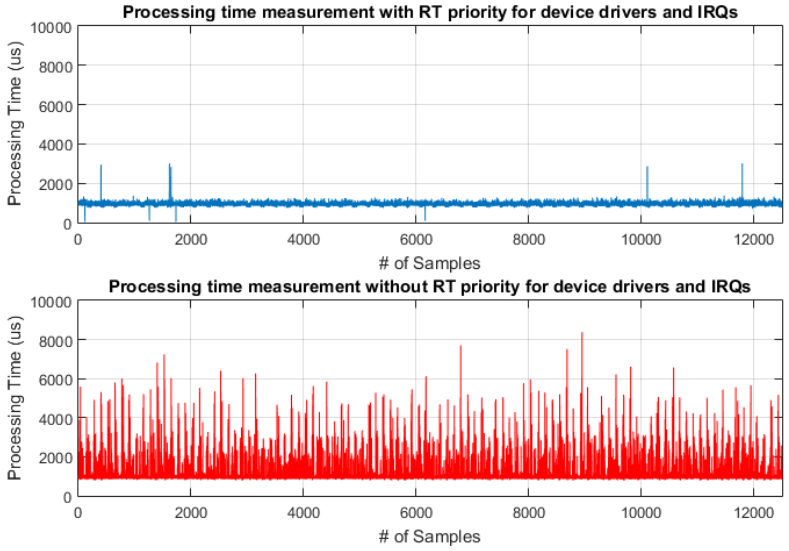


FIGURA 4.8: Comparação entre o tempo de processamento para leitura dos sensores alterando a prioridade dos drivers e tratadores de interrupção para tempo real (em azul) e com prioridade normal (em vermelho).
(Fonte: Autor)

O resultado dessa mudança pode ser visto na Figura 4.8, que representa agora um padrão bem definido no comportamento temporal do processo, reduzindo significativamente os atrasos e garantindo um maior determinismo no tempo de processamento total associado à tarefa de tempo real.

A Tabela 4.1 mostra os valores, em milisegundos, dos dados colhidos com todos os processos relacionados aos drivers e tratadores de interrupção dos barramentos I^2C e SPI configurados com prioridade de tempo real em contraste com os dados colhidos

Tabela 4.1: Comparação entre prioridades dos drivers e tratadores de interrupção associados aos barramentos I^2C e SPI, tempo em milisegundos.
(Fonte: Autor)

Priority	**Samples**	**Min**	**Max**	**Avg**	**StdDev**
RT	12500	0.7	3.2	0.9	0.275
Non-RT	12500	0.8	8.4	1.2	1.153

com os processos configurados com prioridade normal. É possível identificar um valor máximo e um desvio padrão muito maiores do que o desejado quando os processos possuem prioridade normal. Assim, pode-se dizer que a solução encontrada foi bastante adequada, uma vez que o tempo máximo de processamento não ultrapassa os $3.2ms$ e pode-se garantir, considerando uma distribuição normal, que em 95% das vezes o tempo de processamento não será maior que $\bar{x} + 2\sigma$, ou seja, aproximadamente $1.4ms$.

4.4 Interface de comunicação entre aplicação do Titanium e Piloto Automático

Uma vez validada a biblioteca para leitura dos sensores, o próximo passo foi a adequação do módulo G-INS, responsável pela estimação da orientação do veículo, sobre a plataforma Titanium. Além disso, segundo a arquitetura proposta (ver Figura 4.3), a comunicação entre o processo responsável pelo piloto automático com a aplicação do Titanium será através de um mecanismo de comunicação entre processos, representado pelo canal IPC, uma vez que ambas aplicações estarão rodando sobre o mesmo processador.

Esse mecanismo substituiu o protocolo CAN que era utilizado previamente no Driver, responsável por comunicar-se com os módulos de estimação e de controle de trajetória no sentido de distribuir as mensagens recebidas pela aplicação do Titanium entre os módulos, assim como enviar mensagens dos módulos para a aplicação.

Para se ter uma ideia mais precisa do modelo que foi implementado para a nova interface de comunicação, foi feito um mapeamento de todas as mensagens CAN trocadas entre a aplicação do Titanium e o módulo do piloto automático. No total, foram identificadas cerca de 60 mensagens distintas, sendo a grande maioria mensagens de

configuração, tendo algumas mensagens periódicas de controle e *feedback*, bem como mensagens relacionadas à segurança do sistema e geração de alarmes.

As mensagens de configuração são enviadas sempre que o operador adiciona um novo veículo ou altera as configurações do módulo do piloto automático através da interface gráfica. As mensagens de controle e *feedback* são trocadas periodicamente sempre que o módulo de piloto automático está ativado a uma frequência de 20Hz. Já as mensagens de segurança e alarmes são enviadas do Driver para o Titanium por um módulo responsável por fazer a verificação lógica e funcional de todas as unidades associadas ao módulo de piloto automático.

A partir dessa análise, foi possível definir algumas características intrínsecas e outras desejáveis associadas ao sistema de comunicação entre o módulo do piloto automático e a aplicação do Titanium. São elas:

- Grande variedade de mensagens, com diferentes prioridades e frequências de envio e recepção;
- Comunicação *full-duplex*, com envio e recebimento simultâneo de mensagens;
- Modo de operação assíncrono, sendo necessário um *buffer* para armazenar as mensagens de entrada e saída;
- Possibilidade de distribuição, de forma que a mesma mensagem possa ser recebida por submódulos diferentes;
- Performance temporal boa o suficiente de maneira a não aumentar significativamente os tempos de resposta do sistema;
- Utilização de mensagens com tamanho fixo e campos pré-definidos.

Tendo em vista os pontos acima, foi feito um estudo das alternativas disponíveis para a escolha do mecanismo de comunicação entre processos a ser utilizado. Os protocolos aqui tratados foram apresentados de maneira geral no Capítulo 3 e serão discutidos com maiores detalhes na próxima seção.

4.4.1 Estudo das alternativas

Dentre os protocolos de comunicação entre processos aqui abordados estão *Named Pipes*, *Message Queue*, D-BUS e *Shared Memory*. Os critérios definidos foram os descritos na seção anterior e, a partir de estudos sobre cada um desses protocolos, foi possível montar uma matriz de comparação que pode ser vista na Tabela 4.2.

TABELA 4.2: Comparação entre os mecanismos de comunicação entre processos. (Fonte: Autor)

	Mode	Coupling	Bus Interface	Message Boundary	Message Priority	Blocking read/write
Named Pipe (FIFO)	Two-way*	Strong	✘	✘	✘	✓
DBUS	Two-way	Weak	✓	✓	✓	✓
Shared Memory	Two-way	Very Strong	✘	✘	✘	✘
Message Queue	Two-way*	Strong	✓	✓	✓	✓

* Requires instantiating 2 objects/file descriptors

Para se compreender a comparação, é necessário explicar o significado de cada critério utilizado. *Mode* indica se a comunicação é uni ou bidirecional. Nesse sentido, todos os protocolos estudados podem operar de forma bidirecional. Porém, no caso do *Message Queue* e do *Named Pipe* é mais apropriado criar dois objetos distintos, uma vez que é difícil controlar o destino das mensagens caso dois processos estejam utilizando o mesmo *buffer* para leitura e escrita. Se não houver um mecanismo de sincronização apropriado, é possível que o processo que escreveu a mensagem no *buffer* ocasionalmente receba sua própria mensagem quando estiver realizando uma operação de leitura. Para contornar esses problemas de operação recomenda-se a criação de dois objetos distintos, um para leitura e outro para escrita.

Coupling está relacionado com a necessidade de ambos os processos que estão se comunicando conhecerem a priori o tipo de informação que está sendo recebida e de que forma ela está estruturada. No caso do D-BUS, é necessário que o processo conheça a priori o método que estará acessando remotamente e se procupe unicamente com o resultado retornado, isso faz com que o tratamento dos dados seja bastante simplificado. Além disso, um processo qualquer que necessite de alguma informação que está disponível através de acesso a um procedimento remoto pode facilmente acessar o barramento e fazer uma requisição a essa informação. Já os *Named Pipes* e *Message Queue* necessitam de um acoplamento mais forte entre as aplicações. É preciso que ambos estabeleçam

que tipos de dados serão trocados e como eles estão estruturados para que possam interpretar corretamente as mensagens. No caso do *Shared Memory* o acoplamento é ainda mais forte, uma vez que o acesso a uma posição de memória incorreta, que não seja de conhecimento prévio por parte dos processos que estão utilizando esse mecanismo, pode invalidar completamente a informação enviada ou recebida.

Bus Interface indica as funcionalidades de alto nível de cada mecanismo no que diz respeito à maneira como acessar as funções através do espaço de usuário. Nesse sentido o D-BUS, por ser um protocolo de invocação remota de procedimentos e troca de mensagens, tem a maior variedade de funcionalidades disponíveis. *Message Queue* vem logo atrás, sendo possível verificar o número de mensagens na fila, definir alguns atributos como número máximo de mensagens, tamanho das mensagens, dentre outros. Já os protocolos *Named Pipe* e *Shared Memory* se limitam quase que exclusivamente às funções de leitura e escrita e por isso ficam atrás na avaliação desse critério.

Message Boundary e *Message Priority* são conceitos relacionados. Estes avaliam se existe uma fronteira para as mensagens, ou se estas são simplesmente um *stream* de bytes que deve ser tratado apropriadamente dentro de cada aplicação. O D-BUS possui intrinsicamente essa característica, uma vez que cada método irá retornar um tipo de dado específico, que é conhecido pelas aplicações que utilizam o barramento, podendo-se definir prioridades de acesso a este barramento. No caso de *Named Pipes* não é possível armazenar mensagens em uma fila, pois os dados não são estruturados, sendo basicamente um *stream* de bytes do tipo FIFO *(First-in, First-out)*. No *Shared Memory* o conceito é um pouco diferente, uma vez que o que está sendo compartilhado entre os processos é uma região de memória em vez de uma mensagem no sentido mais formal. Assim, caso haja uma mensagem de maior prioridade a ser enviada, esses dois mecanismos não dispõem de uma funcionalidade que possibilite enviá-la diretamente para o início da fila, sendo de responsabilidade do desenvolvedor implementar esse mecanismo nas aplicações. Já o *Message Queue* tem a capacidade de armazenar mensagens com fronteiras definidas, sendo possível enfileirar diferentes tipos de mensagens em uma mesma fila de acordo com suas prioridades.

O último critério é *Blocking Read/Write* que indica se o protocolo possui ou não um mecanismo de verificação para operações de leitura quando o *buffer* está vazio, bem como operações de escrita quando o *buffer* se encontra cheio, além de mecanismo de acesso

exclusivo ao recurso. O único que não possui essa funcionalidade é o *Shared Memory*, onde é necessário que o projetista implemente separadamente um mecanismo de acesso exclusivo utilizando semáforos ou exclusão mútua, de forma a manter a integridade dos dados.

É importante salientar que todos os critérios desejados podem de certa forma ser implementados pelo desenvolvedor como parte da aplicação. Porém, deseja-se aqui que o protocolo escolhido já tenha intrinsicamente algumas dessas características, de forma a reduzir a complexidade da aplicação. Além disso, a utilização de um protocolo já estabelecido garante um nível de segurança e funcionamento correto sem a necessidade de testes e validações.

4.4.1.1 Análise temporal

Baseado na matriz de comparação da Tabela 4.2 é possível notar que o protocolo *Shared Memory* não preenche nenhum dos requisitos iniciais desejados. Com isso, a análise temporal foi feita apenas com *Named Pipes*, *D-BUS* e *Message Queue*.

Para fazer os testes, procurou-se simular as condições de utilização do sistema final. Assim, dois processos executam em paralelo sobre o mesmo processador, um deles com prioridade normal que simula a aplicação do Titanium, e outro com prioridade alta que simula o módulo do piloto automático. A troca de mensagens ocorre a cada $50ms$, simulando as mensagens trocadas contendo os dados do módulo GNSS, por exemplo. No corpo da mensagem é enviado um *timestamp*, representado por T_i, adquirido no momento anterior ao envio. No lado do receptor, logo que a mensagem é recebida é também gerado um *timestamp*, representado por T_f. O intervalo de tempo associado ao tempo que decorreu desde o envio até o recebimento da mensagem é dado por $\Delta T = T_f - T_i$. Esse procedimento é executado repetidamente, de forma a se obter um número considerável de amostras.

Tabela 4.3: Análise temporal dos protocolos de comunicação entre processos D-BUS, *Message Queue* e *Named Pipe* (valores em microsegundos).
(Fonte: Autor)

IPC Protocol	**Mean**	**StdDev**	**Max**	**Min**
Named Pipe	347	434	3446	324
D-BUS	1514	490	11037	868
Message Queue	63	382	3414	57

O resultado dessa análise pode ser visto na Tabela 4.3. É possível verificar que o protocolo *Message Queue* é o mais rápido e com maior determinismo dentre os três testados. Como a análise dos critérios feita na seção anterior favorece bastante os protocolos D-BUS e *Message Queue*, a decisão foi de utilizar o protocolo *Message Queue* no projeto. Porém, para não se limitar ao protocolo utilizado, foi criada uma interface de comunicação que será utilizada pela camadas superiores da aplicação, onde o protocolo utilizado fica abstraído. Dessa forma, é possível implementar diferentes protocolos sem a necessidade de alterar a maneira como os métodos são chamados pelas outras classes. Esses, bem como outros detalhes da implementação e a arquitetura do gerenciador de mensagens utilizado, serão discutidos nas seções seguintes.

4.4.2 Gerenciador de mensagens

A estrutura do gerenciador de mensagens foi pensada a partir de um modelo já existente na empresa utilizado para rede CAN, porém adaptado para o cenário de comunicação entre processos. Uma das preocupações no seu desenvolvimento foi a de manter uma interface simples, de forma que as classes que desejam enviar mensagens precisem apenas passar como parâmetro o corpo da mensagem em si e sua prioridade. Já para o recebimento de mensagens, a estrutura é um pouco mais complexa, como pode ser visto na Figura 4.9.

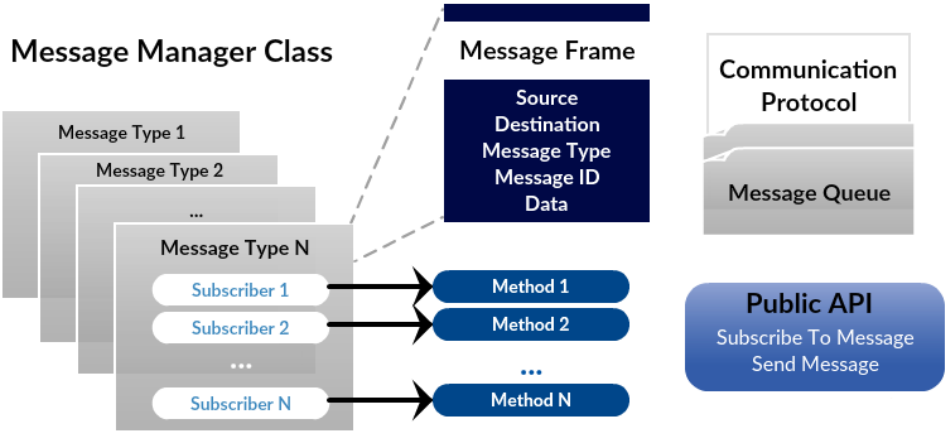


FIGURA 4.9: Estrutura da classe utilizada para gerenciar as mensagens a serem trocadas entre a aplicação do Titanium e o módulo do piloto automático.
(Fonte: Autor)

Para cada mensagem contida numa lista em que cada item possui identificação única é possível associar ponteiros para métodos a serem invocados, implementados em

uma classe que está inscrita para receber aquela determinada mensagem. Dessa forma, toda vez que uma mensagem é retirada do *buffer* pelo gerenciador de mensagens, este verifica qual é o ID dessa mensagem e, baseado na lista de classes inscritas, distribui essa mensagem como parâmetro de entrada na invocação do método. Assim, é possível diretamente executar a seção de código associada ao recebimento de cada mensagem, como se esta seção de código fosse um tratador único daquela mensagem, aumentando assim a responsividade do sistema.

As mensagens são compostas por uma estrutura contendo campos como remetente e destinatário, o tipo da mensagem, indicando se é uma mensagem de configuração ou de controle, o ID da mensagem, que compõe um identificador único para cada mensagem dentro do sistema e, por fim, o *payload* da mensagem em si, contendo as informações que serão trocadas entre os processos.

Uma API está disponível para que as classes do sistema possam se inscrever para receber as mensagens e passar o ponteiro ao método a ser invocado quando do recebimento de cada mensagem. Através da API é também possível enviar mensagens, sendo necessário preencher os dados definidos na estrutura da mensagem e indicar a prioridade da mensagem a ser enviada, onde mensagens de controle geralmente possuem prioridade maior que mensagens de configuração.

Como comentado na seção anterior, é possível também definir qual protocolo está sendo utilizado apenas instanciando um ou outro para compor a camada mais inferior de comunicação, não sendo necessária nenhuma alteração em seções de código das camadas que utilizam as funcionalidades disponíveis através do gerenciador de mensagens.

4.5 Importação dos submódulos

Uma vez que a camada de comunicação entre a aplicação do Titanium e o módulo do piloto automático está implementada, pode-se partir para a importação e adequação dos submódulos utilizados no Driver para a plataforma Titanium. Algumas alterações tiveram que ser feitas para adequar o módulo do piloto automático à nova plataforma, como por exemplo acesso a elementos de *hardware* como temporizadores, memória, porta serial, bem como o acesso ao acelerômetro e giroscópio, já discutido no presente relatório.

Como praticamente toda a parte lógica de processamento já implementada no Driver teve que ser mantida, sendo necessário mudar e adequar apenas o acesso aos periféricos da placa e os canais de comunicação entre as aplicações e dipositivos, pensou-se numa maneira de manter a compatibilidade dos submódulos presentes nas duas plataformas, de maneira que alterações de código ou a implementação de novas funcionalidades sejam sentidas tanto pelo Driver quanto pelo Titanium. Isso é fundamental, uma vez que ambas as plataformas estão ativas como produtos da empresa e são passíveis de serem melhoradas e otimizadas com relação ao seu funcionamento lógico.

Dessa forma, buscou-se evitar código duplicado, o que tornaria a identificação de problemas comuns a ambas as plataformas muito mais complexa, gerando retrabalho desnecessário à equipe, tendo em vista que seria preciso inspecionar o código presente nas duas plataformas.

4.5.1 Criação de interfaces de acesso

Para solucionar esses problemas, pensou-se em desacoplar todos os submódulos e bibliotecas comuns a ambas as plataformas, criando interfaces de acesso às informações que seriam implementadas independentemente em cada uma das plataformas, de forma que o código presente nos submódulos não fosse alterado, uma vez que as informações compartilhadas entre as classes seriam armazenadas em uma interface com *getters* e *setters* acessíveis publicamente. Essa ideia está representada na Figura 4.10.

A partir dessa ideia, foi possível manter o funcionamento lógico do código tanto no Driver, quanto no Titanium, de forma que os submódulos G-INS, Trajectory Controller e Actuators Controller, bem como bibliotecas comuns a ambos sempre apontam para os mesmos repositórios.

No caso do Driver, a criação das interfaces é feita de maneira bastante simples, uma vez que todos os dados estão presentes na memória do processador. Portanto, todos os dados que eram compartilhados entre os submódulos através de *getters* e *setters* diretamente acessíveis por métodos públicos fornecidos pela classe foram transferidos para as interfaces, de forma que sempre que deseja-se acessar um determinado dado que é comum a mais de um submódulo, deve-se utilizar a interface. As interfaces nada

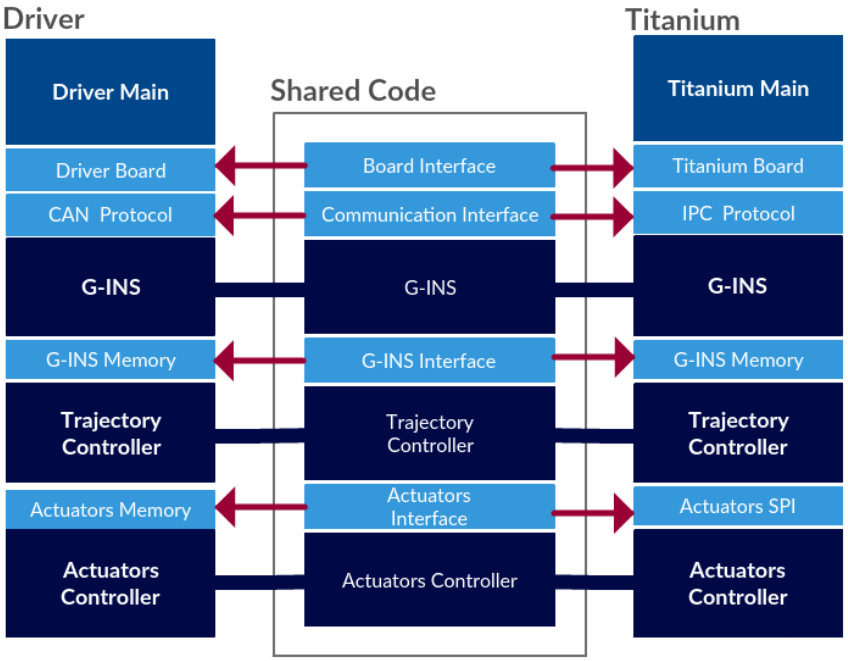


FIGURA 4.10: Desenvolvimento de interface de acesso aos dados e compartilhamento de seções de código comum.
(Fonte: Autor)

mais são do que classes com métodos puramente virtuais que ditam como deve ser o comportamento de classes que derivam delas.

Há ainda a criação de interfaces para a camada de comunicação com a aplicação do Titanium e com o *hardware* disponível pela *baseboard*. Essas interfaces são representados pelas classes Board Interface e Communication Interface.

Já no caso do módulo do piloto automático implementado na plataforma Titanium, há uma diferença nas implementações das classes derivadas das interfaces. Primeiramente, a forma com que o módulo do piloto automático comunica-se com o Titanium é através de um mecanismo de comunicação entre processos em vez do protocolo CAN. O *hardware* presente no Titanium também possui algumas diferenças com relação ao Driver e necessita de uma classe específica para descrever as funções disponíveis. A interface do módulo G-INS é praticamente a mesma para ambas as plataformas, uma vez que ela apenas fornece dados para as classes externas.

A grande diferença encontra-se na interface dos atuadores, chamada na Figura 4.10 de Actuators Interface. No caso do Driver, os dados continuam presentes e acessíveis na memória do programa, uma vez que estão todos sobre o mesmo processador. Por outro

lado, no Titanium o submódulo Actuators Controller será implementado em um microcontrolador dedicado para leitura de sensores *hall* e acionamento PWM dos atuadores (ver Figura 4.3) e deverá se comunicar com o processo do piloto automático através de um mecanismo de comunicação *on-board*, como por exemplo, o protocolo SPI.

O que difere os códigos dos dois módulos de piloto automático são as classes derivadas das interfaces para cada uma delas e a *Main* de cada programa. É na *Main* que decide-se quais interfaces serão utilizadas pelo programa em questão e só então a tarefa responsável pelo controle do módulo do piloto automático passa a executar. Essa ideia pode ser visualizada na Figura 4.11.

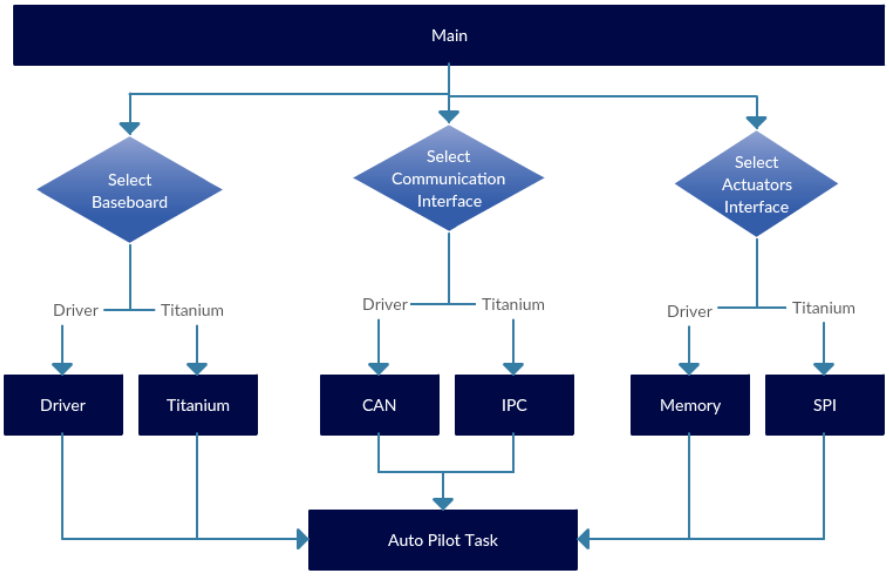


FIGURA 4.11: Escolha das interfaces e do *hardware* a ser utilizado dependendo da plataforma.
(Fonte: Autor)

A importância desse conceito se amplia conforme diferentes plataformas de *hardware* e protocolos de comunicação passarem futuramente a ser utilizados. Dessa forma, ficam transparentes os dados que as interfaces devem fornecer externamente, sendo necessário apenas implementar as classes responsáveis por obter esses dados. Após a criação das interfaces, o foco foi direcionado para importação e adequação dos submódulos G-INS e Trajectory Controller para a plataforma Titanium.

4.5.2 Submódulo G-INS

O primeiro submódulo a ser criado é o G-INS, responsável pela interação com a aplicação do Titanium para receber os dados oriundos do módulo GNSS e então, a partir das leituras do acelerômetro e giroscópio, fazer a estimação da orientação do veículo em tempo real.

Além das alterações de *hardware* já descritas para adequar a plataforma, há também o mecanismo de acesso às informações do acelerômetro e giroscópio que influenciam o resultado lógico gerado por esse submódulo. Por outro lado, a parte lógica e estrutural não sofreu alterações. Há também o desejo da empresa de reutilizar esse módulo de estimação em outros projetos. Para tanto foi desenvolvida uma API, derivada da interface G-INS, de forma a tornar disponível algumas informações que seriam utilizadas por outras aplicações, tal como a orientação estimada do veículo nos três eixos de referência, a velocidade corrigida, informações sobre a configuração do veículo, posição da antena, distância entre eixos, dentre outras.

Para validar algumas das modificações feitas no submódulo G-INS, foi feito um teste em bancada para estimar os ângulos de *roll* e *pitch*. O procedimento utilizado foi o de acoplar a placa do Driver sobre a plataforma Titanium e fazer com que ambas tivessem o mesmo deslocamento angular nos eixos X e Y. Dessa forma foi possível comparar os dados obtidos de ambas as plataformas e verificar se o funcionamento lógico do novo módulo estava adequado. O resultado desses testes pode ser visto na Figura 4.12.

Pelo gráfico é possível perceber que o funcionamento lógico do módulo, mesmo com as mudanças de *hardware* e adaptações feitas, continua apresentando a resposta desejada. O erro presente em alguns picos pode decorrer de várias fontes. Primeiro, porque as leituras do acelerômetro e giroscópio estão sendo feitas em dispositivos diferentes em cada plataforma. Estes podem apresentar pequenas diferenças de calibração e resposta dinâmica. Outro fator que pode influenciar é que o centro de massa de ambas plataformas não se encontra exatamente sobre o mesmo ponto, portanto o efeito medido pelos sensores pode ser ligeiramente diferente. Mesmo considerando esses fatores, pode-se dizer que a resposta se enquadra perfeitamente dentro do que se esperava.

A validação da estimação do ângulo de *yaw* é um pouco mais complexa e deve ser, preferencialmente, feita em campo ou através de um simulador de dados GNSS, uma

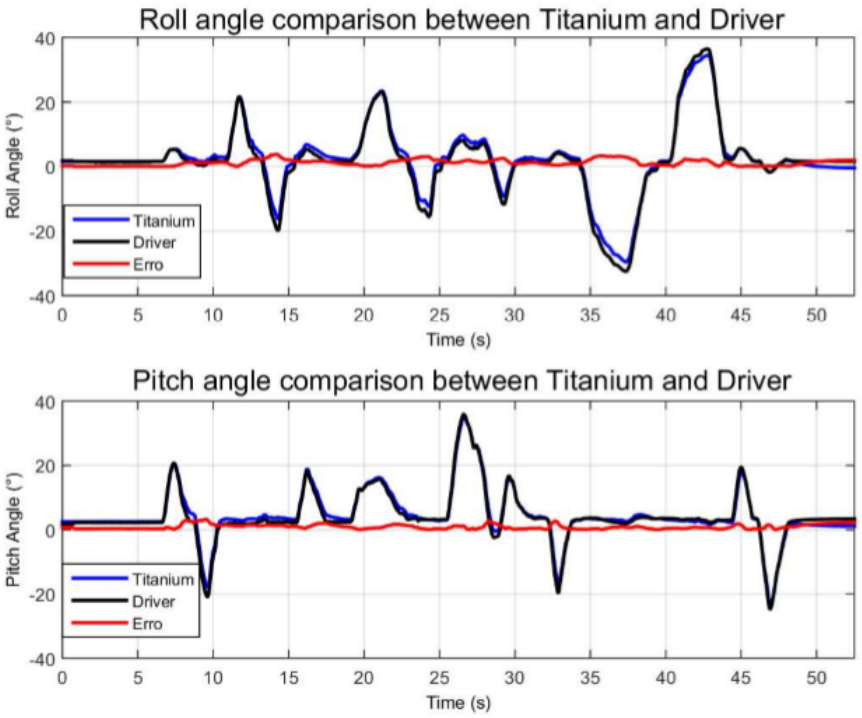


FIGURA 4.12: Validação dos ângulos de *roll* e *pitch* em bancada através de comparação com os ângulos gerados pelo Driver.
(Fonte: Autor)

vez que é preciso conhecer a velocidade do veículo para que a estimação seja feita de maneira correta. Essa etapa será apresentada no Capítulo 5, que discute os resultados finais obtidos.

4.5.3 Submódulo Trajectory Controller

Após validado o submódulo G-INS, o próximo passo foi a importação e adequação do submódulo Trajectory Controller, o qual executa os algoritmos de controle de trajetória e, a partir da referência recebida pela aplicação do Titanium e da estimação da orientação, aplica o algoritmo discutido em [18], fornecendo como saída o ângulo de referência a ser aplicado na roda do veículo.

A adequação do módulo passou principalmente pelo desenvolvimento da interface, onde os dados compartilhados foram movidos para a classe derivada do Actuators Interface. Para validar as alterações foram utilizadas ferramentas de depuração, de forma a verificar o funcionamento lógico do módulo e a comunicação deste com os outros

submódulos e a aplicação do Titanium. Juntamente com o submódulo Trajectory Controller estão implementadas as rotinas de segurança do piloto automático. Foram realizados alguns testes em bancada simulando condições de quebra da rotina normal de execução e verificou-se o disparo de alarmes na tela do Titanium.

O próximo passo para validação completa do módulo do piloto automático, desde a estimação da orientação, cálculo do controle de trajetória e envio e recebimento de comandos para sensores e atuadores, será discutido na seção seguinte.

4.6 Piloto automático

Conforme exposto na Figura 4.3, o submódulo Actuators Controller seria implementado em um microcontrolador dedicado, comunicando-se via SPI com o módulo do piloto automático, controlando os atuadores e fazendo as leituras dos sensores de posição da roda ou do volante. Para que seja possível essa implementação, é necessário o desenvolvimento de uma nova *baseboard*, contendo o novo microcontrolador e a estrutura de comunicação necessária. Este trabalho não faz parte do escopo deste projeto e tem previsão de início apenas após o término das atividades descritas neste relatório.

Sendo assim, a validação do piloto automático utilizando o motor elétrico ou a válvula hidráulica não pôde ser concluída. No entanto, há uma alternativa que possibilita a validação de toda a malha de processamento e controle do módulo do piloto automático implementado no Titanium. Esta é através de um produto desenvolvido pela Leica Geosystems, chamado Leica DirectSteer ES [23] (que será referenciado ao longo deste relatório somente por LeicaDS), empresa pertencente ao grupo Hexagon. O LeicaDS é um sistema que pode ser acoplado a qualquer volante e é transferível facilmente de uma máquina a outra. Esse sistema recebe comandos de acionamento via CAN, mais precisamente uma posição angular de referência, e devolve um *feedback* para o sistema de controle, a cada $100ms$, indicando a posição atual do volante e se este está ou não ativado. A maneira como ele interage com o Titanium pode ser vista na Figura 4.13.

Dessa forma, optou-se por implementar o submódulo Actuators Controller dentro do próprio módulo de piloto automático e a criação de um canal de comunicação via CAN para o envio e recebimento de comandos para o LeicaDS. A arquitetura desenvolvida para adaptar o sistema à essa aplicação pode ser vista na Figura 4.14.

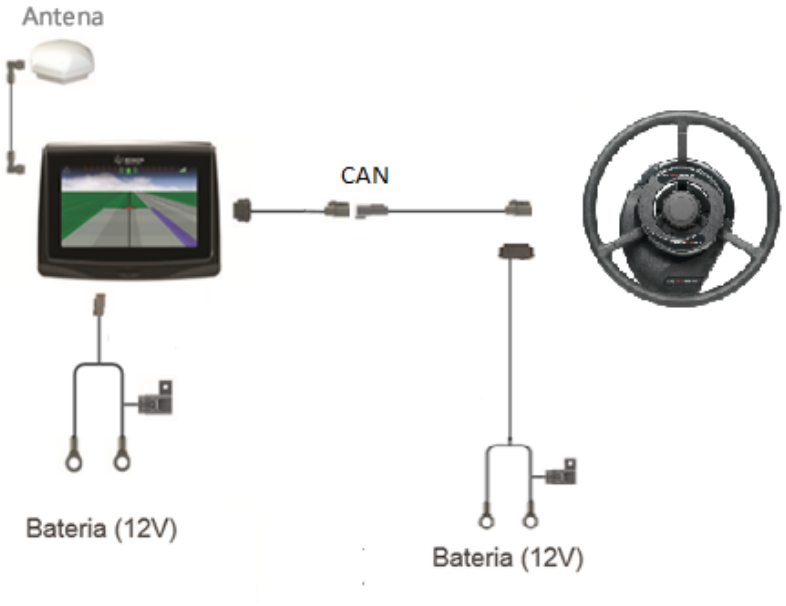


FIGURA 4.13: Esquemático mostrando a interação entre o LeicaDS e o Titanium através de protocolo CAN.
(Fonte: Autor)

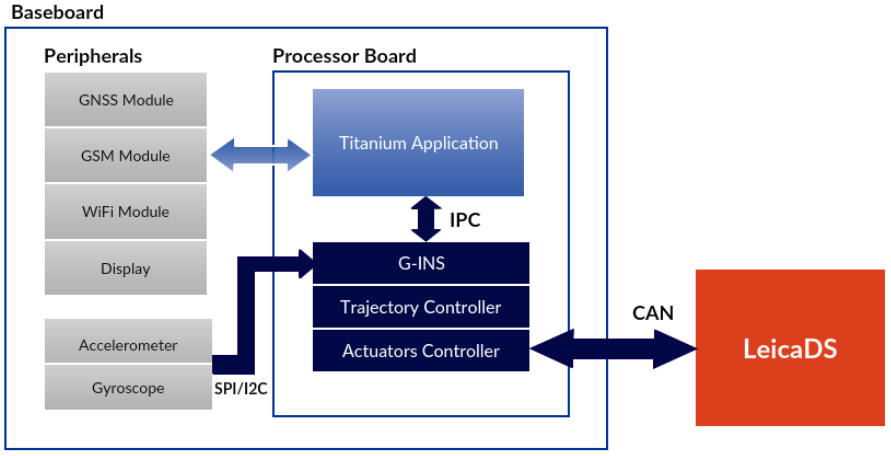


FIGURA 4.14: Arquitetura do sistema utilizando o volante fornecido pela LeicaDS.
(Fonte: Autor)

É importante ressaltar que essa estrutura é temporária, pois o submódulo Actuators Controller será transferido para o microcontrolador dedicado assim que o *hardware* estiver disponível para tal. No entanto, o uso do LeicaDS contribuiu para a validação de todas as implementações feitas anteriormente, fechando todo o ciclo necessário para o funcionamento do piloto automático, desde os canais de comunicação até o funcionamento lógico e temporal. O resultado final das implementações, bem como análises de desempenho e avaliação de requisitos será discutido no próximo capítulo.

Capítulo 5

Resultados

Neste capítulo serão apresentados os resultados do projeto descrito nos capítulos anteriores. Dentre os resultados esperados estão o funcioamento lógico do módulo do piloto automático em conjunto com a aplicação do Titanium e o atuador LeicaDS, que será observado através de testes em bancada e em campo, bem como o funcionamento temporal e garantia do determinismo das malhas de controle que necessitam operar em tempo real. Uma breve análise do barramento de comunicação entre processos também será realizada. Os resultados serão apresentados por meio de gráficos, tabelas e figuras, seguidos de discussões detalhadas sobre cada um dos aspectos analisados.

5.1 Análise do sistema

Nesta seção serão apresentadas as análises temporais das malhas de controle e do módulo do piloto automático como um todo, considerando o *wake-up time* de cada um dos sub-módulos, bem como o tempo de processamento de cada um deles e o tempo de processamento total. Essa mesma análise será feita para o sistema com carga de operação normal e com carga elevada na CPU, memória e nas operações de entrada e saída, de forma a verificar o determinismo temporal em cada uma das situações. Será feita uma análise da dependência do módulo de piloto automático com a aplicação do Titanium e de que forma isso influencia o comportamento temporal das aplicações, além de uma verificação da taxa de utilização do barramento de comunicação entre processos implementado.

5.1.1 Taxa de utilização do barramento de comunicação

Como apresentado nas seções anteriores, o barramento CAN que conectava o módulo do piloto automático com a aplicação do Titanium foi substituido por um mecanismo de comunicação entre processos. Após validado esse mecanismo, verificando se as mensagens estavam sendo trocadas corretamente entre a aplicação do Titanium e a aplicação do piloto automático, foi feito um teste para determinar a taxa de utilização do barramento de comunicação implementado.

Esse teste serve para se ter uma ideia mais precisa da quantidade de mensagens sendo trocadas entre ambas aplicações e se, futuramente, mais processos ou sub-módulos poderiam utilizar o mesmo barramento para comunicação sem que haja uma sobrecarga que faça com que mensagens sejam perdidas ou deixem de ser entregues corretamente a seus respectivos destinatários.

O teste foi realizado medindo-se a quantidade de mensagens enviadas e recebidas por segundo pela aplicação do piloto automático, sendo que esse barramento pode ser considerado ponto a ponto, uma vez que apenas o módulo do piloto automático e a aplicação do Titanium comunicam-se através dele. Os resultados podem ser vistos na Tabela 5.1.

Tabela 5.1: Análise do barramento de comunicação implementado e sua taxa de utilização, do ponto de vista da aplicação do Titanium.
(Fonte: Autor)

Message Direction	**Mean (msg/s)**	**Std (msg/s)**	**Bandwith (KB/s)**	**Buffer Size (bytes)**	**Max Used (bytes)**	**Usage (%)**
Sent	138	2	4.42	1024	224	21.88
Received	81	2	2.60	1024	256	25.00

A tabela mostra a grande quantidade de mensagens trocadas entre o módulo do piloto automático e a aplicação do Titanium quando o piloto automático está ativado. Desse número, grande parte das mensagens são trocadas contendo parâmetros de controle e *feedback*, do lado do módulo do piloto automático para receber os dados oriundos do módulo GNSS e referentes à trajetória de referência, e do lado da aplicação do Titanium para receber *feedback* da operação, se tudo está sendo executado conforme especificado, bem como os parâmetros de controle e estimação calculados. No total, são mais de 200 mensagens trocadas por segundo entre as aplicações, o que se traduz em uma taxa de transmissão média de $4.42KB/s$ e $2.60KB/s$ para envio e recebimento, respectivamente.

O desvio padrão é bastante baixo, o que indica baixa dispersão no envio e recebimento das mensagens, demonstrando funcionamento correto.

Parâmetros como tamanho dos *buffers* de entrada e saída e tamanho de cada mensagem são especificados através de um arquivo *header* comum a ambas aplicações, podendo ser alterado conforme necessário. Atualmente, o tamanho de cada mensagem está fixado em 32 bytes, sendo que tanto o *buffer* de entrada quanto o de saída possuem espaço para armazenamento de até 32 mensagens, o que dá um tamanho total de $1KB$. O número máximo de mensagens simultâneas nos *buffers* utiliza um total de 224 bytes e 256 bytes, correspondendo a 21.88% e 25% de utilização para envio e recebimento, respectivamente.

Dessa forma, pode-se concluir que é possível que outros processos utilizem o barramento, ou até mesmo pode-se aumentar a frequência da troca de mensagens entre as aplicações sem comprometer o funcionamento do mecanismo de comunicação. Também pode-se reduzir o tamanho do *buffer* para que sua taxa de utilização seja maximizada.

5.1.2 Análise temporal

A análise temporal é um passo fundamental na validação do sistema e na avaliação dos objetivos propostos neste trabalho. É preciso que o módulo do piloto automático respeite os *deadlines* impostos, de forma a garantir um funcionamento correto e livre de falhas. Para tanto, sua prioridade foi definida como a mais alta dentre os processos disparados pelo usuário e pretende-se, através dos testes temporais, verificar o cumprimento dos *deadlines* em situações normais de operação e também em situações de sobrecarga no sistema.

Os parâmetros utilizados para fazer essa avaliação foram o *wake-up time* e tempo de processamento tanto da malha de controle de trajetória quanto da malha de estimação da orientação do veículo. Foi avaliada também a dispersão temporal no recebimento das mensagens oriundas da aplicação do Titanium contendo os dados do módulo GNSS.

5.1.2.1 *Wake-up time* das malhas de controle e estimação

O *wake-up time* é um parâmetro que indica se as malhas estão sendo executadas dentro do período estipulado. Este é medido como o tempo decorrido entre duas execuções

consecutivas da seção de código referente à malha em questão. A malha de controle de trajetória deve operar a cada $50ms$, enquanto que a malha de estimação deve operar a cada $5ms$. Os testes em condições normais de operação foram realizados executando a aplicação do piloto automático juntamente com a aplicação do Titanium e todos os outros processos necessários à execução do sistema operacional Linux na plataforma embarcada. Já o teste com carga no sistema foi realizado utilizando-se da ferramenta *stress-ng* [24], que é utilizada para testar o sistema sob diferentes condições de operação. Nos testes aqui apresentados foram utilizados, além dos processos do piloto automático e da aplicação do Titanium, mais 5 processos *CPU bound*, 2 processos *I/O bound* e 1 processo forçando a utilização de memória virtual. Os resultados obtidos podem ser vistos nas Figuras 5.1 e 5.2.

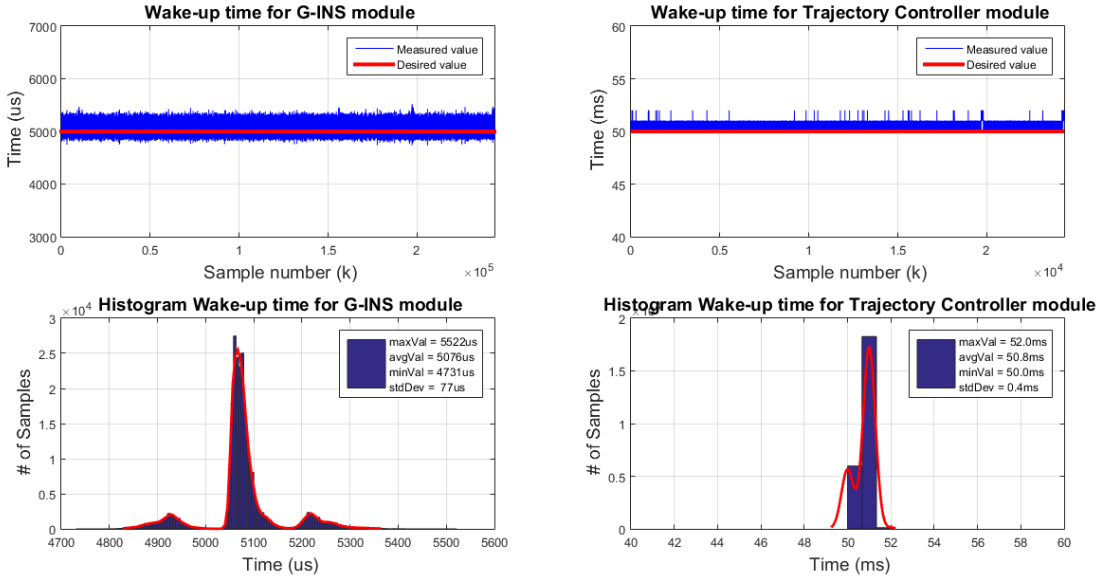


FIGURA 5.1: *Wake-up time* para o submódulo G-INS e o submódulo Trajectory Controller no modo de operação normal.
(Fonte: Autor)

Como pode ser observado nas Figuras 5.1 e 5.2, tanto nos testes realizados no modo de operação normal quanto nos testes com carga no sistema, o comportamento temporal foi adequado e não houve perda de *deadlines* devido a latências excessivas no sistema. Uma síntese dos gráficos apresentados acima pode ser vista na Tabela 5.2.

A partir da análise dos gráficos e da tabela, pode-se concluir que, apesar de haver uma carga excessiva na CPU, na memória e nas operações de entrada e saída, não há qualquer influência significativa na resposta das tarefas de tempo real do sistema, de forma que as latências de ativação continuam seguindo um comportamento padrão, com

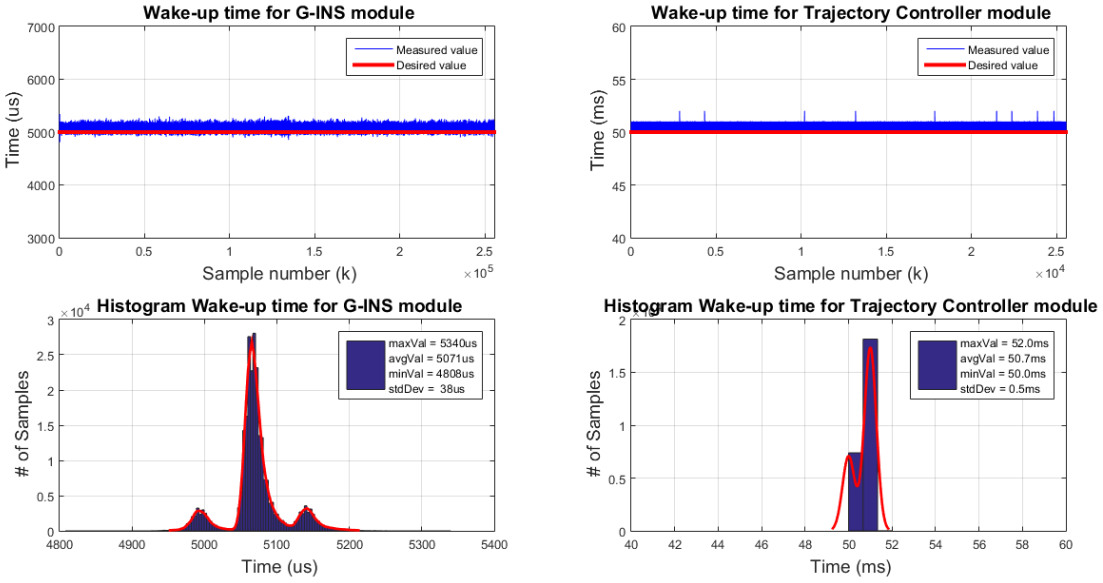


FIGURA 5.2: *Wake-up time* para o submódulo G-INS e o submódulo Trajectory Controller no modo de operação com carga no sistema.
(Fonte: Autor)

baixa dispersão e alta repetibilidade. O número de *deadlines* não cumpridos das tarefas de tempo real em ambos os casos é igual a zero, o que supera a expectativa inicial adotada que permitia que até 5% dos *deadlines* pudessem ser perdidos sem prejudicar de maneira significativa o comportamento do sistema.

TABELA 5.2: Análise do *wake-up time* das tarefas de tempo real no modo de operação normal e com carga (entre parênteses) expressos em milisegundos.
(Fonte: Autor)

Submodule	**Min**	**Mean**	**Max**	**Std**	**Deadlines Lost**
G-INS	4.73 (4.81)	5.08 (5.07)	5.52 (5.34)	0.08 (0.04)	0 (0)
Trajectory Controller	50.0 (50.0)	50.8 (50.7)	52.0 (52.0)	0.4 (0.5)	0 (0)

Entretanto, vale ressaltar que, de um lado, a aplicação do piloto automático se comportou de maneira adequada com excesso de carga no sistema, porém, como era de se esperar, a aplicação do Titanium foi bastante prejudicada, apresentando *lags* na tela que poderiam prejudicar bastante a imagem do produto, mas nunca travando-o por completo. Isso poderia ser resolvido impondo uma prioridade à aplicação do Titanium que fosse maior que a dos outros processos concorrentes, porém menor que da aplicação do piloto automático, já que esta última possui características de tempo real mais críticas.

5.1.2.2 Tempos de processamento do sistema

A partir da análise do *wake-up time* das tarefas de tempo real, foi possível determinar a inexistência de latências indesejadas de ativação. Porém, para garantir que a execução das tarefas esteja sendo feita sem interrupções, foi medido o tempo de execução de cada tarefa, bem como o tempo de execução total do processo do piloto automático. Tempos de processamento acima do desejado podem apontar gargalos no sistema, sejam eles ocasionados pela própria carga de execução da tarefa (operações de entrada e saída, envio e recebimento de mensagens, complexidade de processamento, etc.) ou por interrupções indesejadas, no caso de tarefas de maior prioridade do próprio *kernel* se tornarem aptas a executar. A mesma abordagem foi adotada, primeiramente medindo os tempos de execução no modo de operação normal e, depois, com carga no sistema. Os resultados obtidos podem ser vistos nas Figuras 5.3 e 5.4.

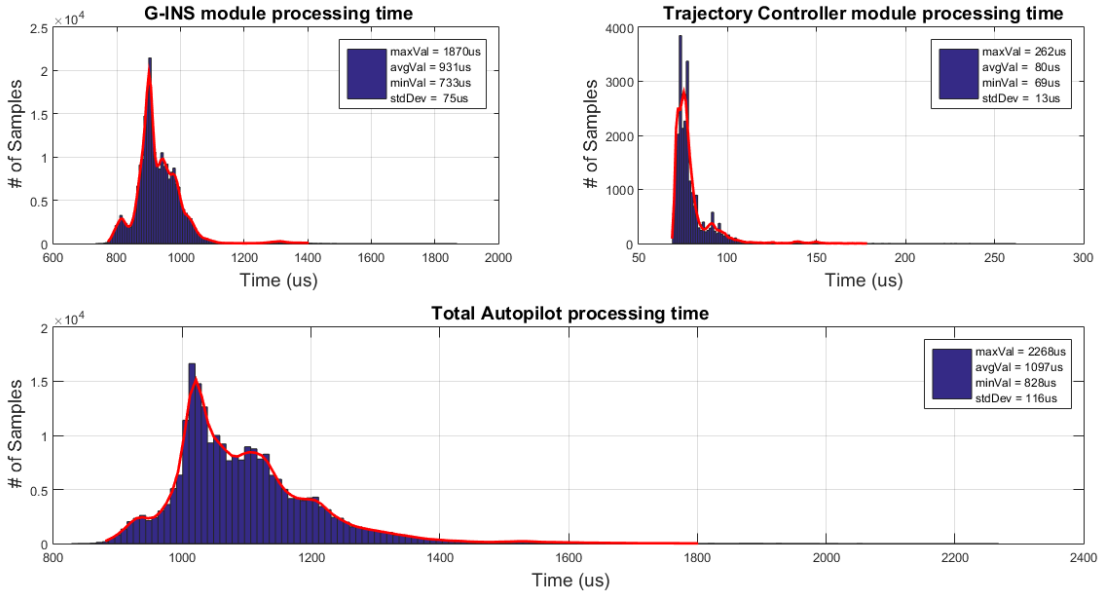


FIGURA 5.3: *T*empo de processamento para os submódulos G-INS e Trajectory Controller e tempo de processamento total no modo de operação normal.
(Fonte: Autor)

Como pode ser visto, o tempo de processamento tanto dos submódulos quanto do processo como um todo sofre uma leve dispersão, porém de amplitude baixa. De maneira geral, o tempo de processamento total do módulo de piloto automático gira em torno de $1ms$, não excedendo um valor máximo de $2.2ms$, o que está dentro do valor desejado. Como o processo é escalonado para rodar a cada $5ms$ e considerando que o tempo de execução médio gira em torno de $1ms$, ainda restam $4ms$ (ou, a grosso modo, 80%

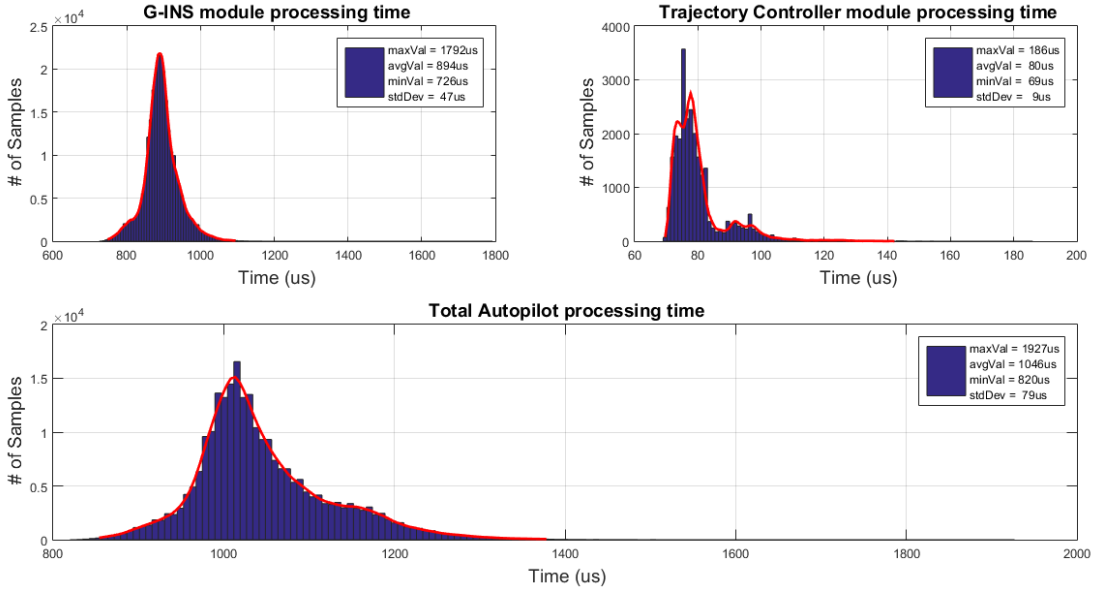


FIGURA 5.4: *T*empo de processamento para os submódulos G-INS e Trajectory Controller e tempo de processamento total com carga no sistema.
(Fonte: Autor)

do tempo disponível da CPU) para os outros processos do sistema poderem executar. Essa fatia de tempo é fundamental para que também a aplicação do Titanium e outros processos relacionados tenham tempo hábil para realizar suas tarefas adequadamente. Além disso, não houve perda de *deadlines* em nenhum dos casos, tanto para o modo de operação normal quanto para o modo de operação com carga no sistema. Os resultados com uma síntese de ambos os testes podem ser vistos na Tabela 5.3.

TABELA 5.3: Análise do tempo de processamento das tarefas de tempo real e o tempo de processamento total do módulo de piloto automático no modo de operação normal e com carga (entre parênteses) expressos em microsegundos.
(Fonte: Autor)

Module	**Min**	**Mean**	**Max**	**Std**
G-INS	733 (726)	931 (894)	1870 (1792)	75 (47)
Trajectory Controller	69 (69)	80 (80)	262 (186)	13 (9)
Total Autopilot	828 (820)	1097 (1046)	2268 (1927)	116 (79)

Ocasionalmente, com os testes realizados com carga no sistema, os desvios padrões foram até menores que com o sistema em operação normal. Isso comprova que, mesmo sobre carga excessiva, o sistema consegue se comportar adequadamente, cumprindo os *deadlines* estipulados e não sofrendo alterações bruscas no tempo de processamento de cada uma das tarefas.

5.1.2.3 Dependência da aplicação do Titanium

Ao longo dos testes de análise temporal e do desenvolvimento do trabalho em si foram sendo identificadas as relações mais críticas entre a aplicação do Titanium e o módulo do piloto automático. Possivelmente a maior delas é o envio da mensagem com dados de controle, sem a qual o submódulo de controle de trajetória ficaria inapto a executar a malha de controle. Assim, para que ele possa de fato calcular o ângulo de referência a ser enviado para os atuadores, de maneira a corrigir a orientação do veículo com relação à trajetória guia, é necessária a chegada dessas informações.

Apesar de a tarefa de controle de trajetória ser acordada periodicamente a cada $5ms$ (por estar implementada no mesmo processo da malha de estimação), ela só passa a executar de fato quando, além de satisfazer a condição imposta pelo período de execução de $50ms$, ela também tenha recebido uma nova mensagem com os dados de controle vinda da aplicação do Titanium. Assim, foi feito um teste de maneira a verificar o tempo de resposta do submódulo de controle de trajetória, baseado no tempo entre a chegada das mensagens de controle. O resultado pode ser visto na Figura 5.5.

A figura mostra claramente que há uma dispersão significativa entre o tempo de chegada das mensagens, o que impossibilita a execução da malha de controle de trajetória no período estipulado. Supostamente, essas mensagens deveriam chegar a cada $50ms$, como a própria média indica. Porém, o valor máximo pode ultrapassar os $100ms$, o que caracteriza uma perda de *deadline* para a tarefa de controle do piloto automático. No teste em questão, 0.8% das amostras estão acima dos $100ms$ e cerca de 4% estão acima de $75ms$, o que pode ser considerado um número bastante indesejável, uma vez que o período deveria ser algo em torno, e muito próximo, de $50ms$, com baixa dispersão.

Esse comportamento está associado à maneira como a aplicação do Titanium trata os dados recebidos do módulo GNSS. Talvez a melhor abordagem para solucionar o problema seria associar uma interrupção toda vez que um novo *frame* do módulo GNSS é recebido e, então, a aplicação do Titanium tratar prioritariamente de decodificar e os dados de controle a serem enviados para o módulo do piloto automático. Isso poderia ser feito instanciando uma *thread* responsável por essa tarefa com prioridade mais alta que a aplicação do Titanium em si. Apesar da variância presente nas amostras não comprometer de modo muito significativo o funcionamento do módulo do piloto automático, a

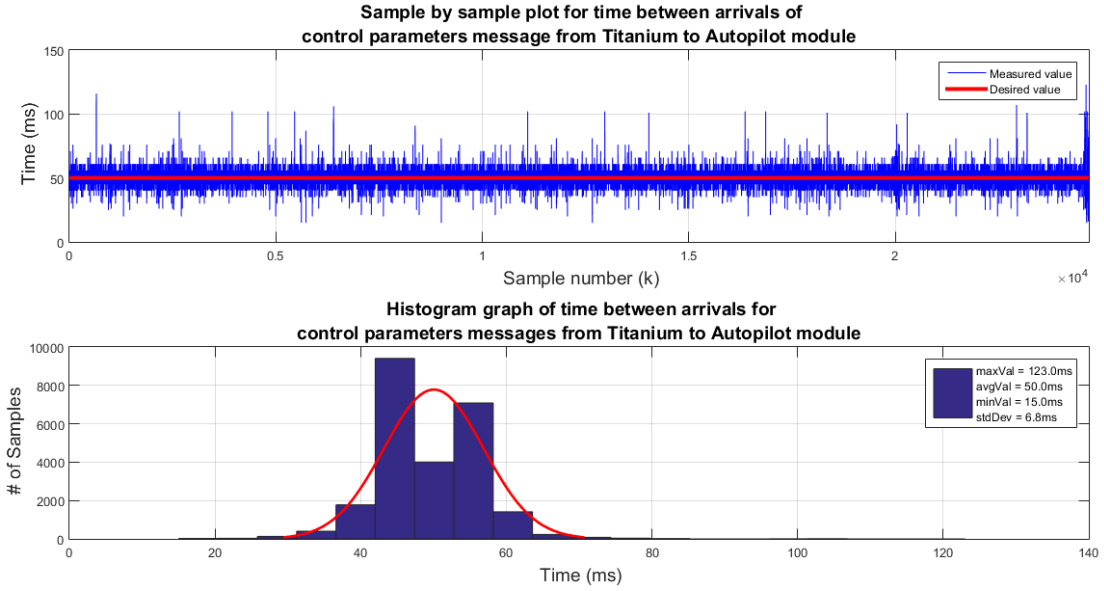


FIGURA 5.5: *T*empo entre chegada das mensagens contendo os parâmetros de controle para o submódulo de controle de trajetória.
(Fonte: Autor)

introdução dessa *thread* poderia contribuir para melhorar ainda mais sua resposta temporal. Além disso, a estimação da orientação feita pelo módulo G-INS também utiliza-se de mensagens oriundas da aplicação do Titanium que dependem dos dados do módulo GNSS, e uma redução nessa variância também traria benefícios na compensação das acelerações lineares do veículo e, consequentemente, uma melhor estimação do ângulo de *yaw*, o qual também é um parâmetro de entrada do sistema de controle de trajetória.

5.2 Validações e testes

Nessa seção serão apresentados os resultados obtidos através de testes realizados primeiramente em bancada e, posteriormente, em campo utilizando um trator da empresa com o sistema de piloto automático instalado. O primeiro teste é para validar o módulo de estimação e a correção do ângulo de *yaw* a partir da utilização do filtro de Kalman e da recepção das mensagens periódicas oriundas da aplicação do Titanium contendo os dados recebidos do módulo GNSS. Após, serão apresentados os testes realizados para validar todo o módulo de piloto automático, segundo a arquitetura apresentada na Figura 4.15, utilizando o atuador LeicaDS. É importante ressaltar que o funcionamento lógico

do módulo do piloto automático depende fortemente do seu correto funcionamento temporal. Porém, como a análise temporal já foi tratada anteriormente, pode-se considerar que o determinismo está presente nos resultados apresentados a seguir.

5.2.1 Validação do cálculo do *Yaw*

Conforme já discutido neste relatório, a validação e estimação do ângulo de *yaw* só é possível a partir do recebimento das mensagens da aplicação do Titanium contendo a velocidade do veículo. Há um *threshold* definido no módulo de estimação, de forma a não realizar os cálculos caso a velocidade seja menor que esse valor. Portanto, os dados só serão válidos se o veículo estiver em movimento com velocidade maior ou igual a esse *threshold*.

Portanto, a validação completa do módulo de estimação foi feita em campo, conectando uma antena GNSS presa no teto do trator pertecente à empresa ao computador de bordo Titanium e então percorrendo uma trajetória em um terreno de teste na região de Florianópolis. Os resultados desse teste podem ser vistos nas Figuras 5.6 e 5.7.

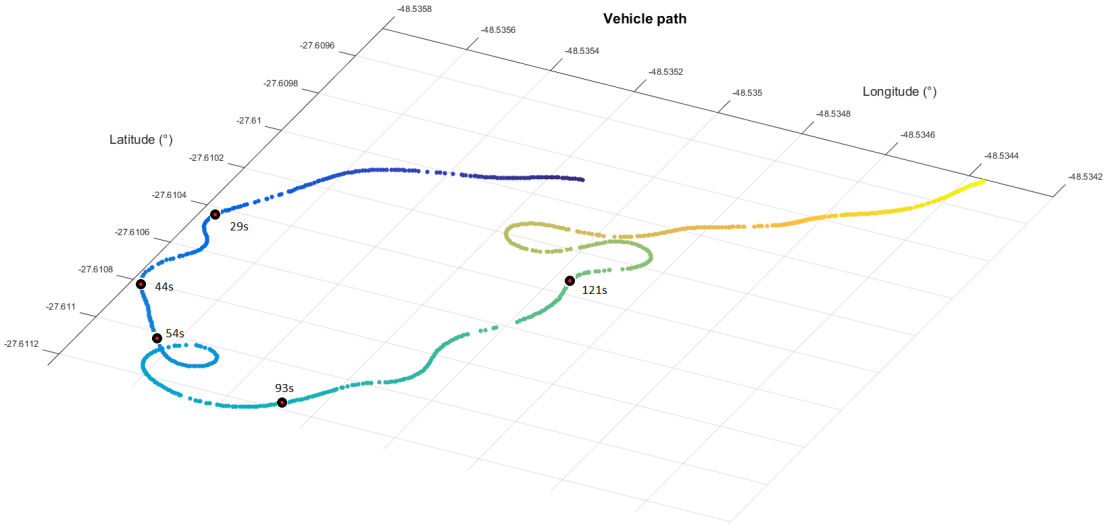


FIGURA 5.6: Trajetória percorrida durante o teste de validação da IMU.
(Fonte: Autor)

A Figura 5.6 mostra a trajetória percorrida pelo trator durante o teste. A tonalidade dos pontos indica a variação do tempo ao longo da trajetória e foram adicionados pontos de referência para poder buscar a informação correspondente no gráfico da Figura 5.7. Os ângulos de *roll* e *pitch* variam conforme as acelerações laterais e longitudinais do trator, respectivamente. Ou seja, em situações de desnível e buracos no terreno, o trator

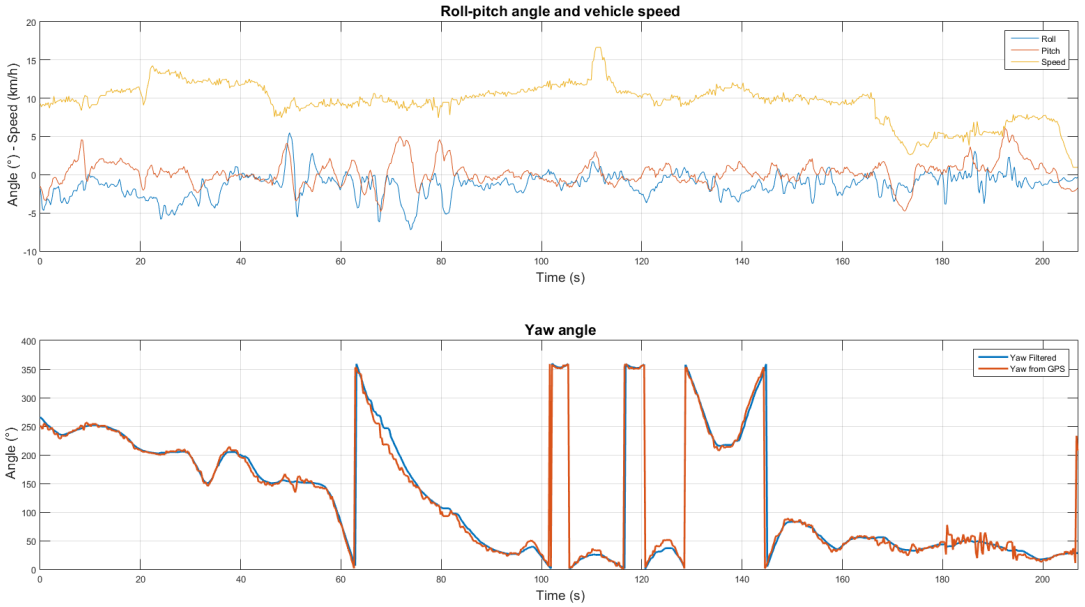


FIGURA 5.7: Ângulos de *roll*, *pitch*, *yaw* e velocidade do veículo durante o teste. (Fonte: Autor)

está bastante suscetível a sofrer alterações nos ângulo de *roll* e *pitch*. Em curvas, quanto mais fechadas estas forem e maior a velocidade do trator, maior será a variação no ângulo de *roll*. Já em situações de frenagem e aceleração, o ângulo de *pitch* sofre maior variação. Uma situação de aceleração e frenagem bastante acentuadas podem ser vistas nos instantes $t = 115s$ e $t = 170s$, respectivamente, com grandes variações na amplitude do ângulo de *pitch*. As demais variações nos ângulos de *roll* e *pitch* são devido a curvas, variações no solo e leves alterações de velocidade, mantendo as amplitudes em torno de $\pm 5°$.

O ângulo de *yaw* está representado no gráfico a partir de duas fontes. O ângulo de *yaw* calculado a partir do filtro de Kalman, bastante suave e sem variações bruscas de amplitude e o ângulo de *yaw* obtido do módulo GNSS, bastante ruidoso. A partir dessa comparação é possível perceber a importância do filtro de Kalman na estimação da orientação do veículo, pois fornece uma medida mais precisa e confiável através da combinação das características estáticas e dinâmicas obtidas do acelerômetro e giroscópio. Uma medida ruidosa seria altamente prejudicial à performance do piloto automático, uma vez que o *yaw* é uma variável controlada do sistema de controle. Pode-se perceber pelo gráfico que os valores dos ângulos estão limitados entre 0° e 360°.

A partir deste teste foi possível validar o funcionamento do módulo de estimação e sua interação através da troca de mensagens com a aplicação do Titanium, bem como as

interfaces de *hardware* com a *baseboard* do Titanium e as interfaces desenvolvidas para se comunicar com o acelerômetro e giroscópio.

5.2.2 Validação do módulo de piloto automático

Uma vez validado em campo o módulo de estimação da orientação, o próximo passo foi a validação final do módulo de piloto automático utilizando o atuador LeicaDS, apresentado no capítulo anterior. Para a validação em bancada foram feitos testes utilizando-se um simulador de dados GNSS e um atuador físico atuando conjuntamente com o computador de bordo Titanium. Já para a validação em campo, foi montado o sistema no trator da empresa e testado em uma área da região de Florianópolis, acoplando o atuador LeicaDS ao volante do trator, recebendo os dados do módulo GNSS através de uma antena e verificando o comportamento do veículo com o piloto automático ativado.

5.2.2.1 Testes em bancada

Os testes em bancada serviram para validar principalmente o funcionamento dos módulos de controle de trajetória, aqui tratado por Trajectory Controller, e de controle dos atuadores, aqui tratado por Actuators Controller. Segundo a arquitetura da Figura 4.15, o módulo de controle dos atuadores foi implementado juntamente com o módulo de estimação e controle de trajetória no mesmo processo. A interface com o atuador LeicaDS é feito via protocolo CAN a cada $100ms$, enviando os sinais de referência para o atuador e recebendo de volta um *feedback* indicando qual a posição atual do volante e se o sistema está funcionando corretamente.

Para simular o veículo em movimento é utilizado um simulador de dados GNSS desenvolvido pela própria empresa, o qual envia os dados para a aplicação do Titanium via porta serial, da mesma forma como ocorre com os módulos físicos GNSS utilizados pela empresa. Dessa forma, pode-se simular uma trajetória a ser percorrida pelo veículo, realimentando os dados a partir da posição do volante, que é enviada diretamente para o simulador de forma a corrigir a posição do veículo na tela do dispositivo. Esse teste serve para validar a estrutura de funcionamento lógico do módulo do piloto automático, para verificar se os dados estão sendo recebidos corretamente da aplicação do Titanium

e se as malhas de controle de trajetória e de controle dos atuadores estão funcionando adequadamente. O ponto negativo é que a malha de estimação de orientação, apesar de estar executando e estimando em tempo real a orientação do veículo, não sofre nenhum tipo de perturbação, vibração e ruído, nem acelerações lineares e angulares, desconsiderando, assim, os efeitos práticos da estimação dos ângulos de *roll*, *pitch* e *yaw*, os quais possuem um papel bastante crítico no sistema. Porém, tendo em vista que o módulo G-INS já foi validado em campo, conforme apresentado na seção anterior, o teste em bancada do módulo do piloto automático se faz necessário para validar os outros módulos do sistema.

Para o teste em questão, foram definidas primeiramente retas A-B e então ativado o piloto automático. Os resultados podem ser vistos nas Figuras 5.8 e 5.9. A Figura 5.8 apresenta a trajetória de referência a ser seguida e a trajetória percorrida pelo veículo. Os dados foram coletados a cada $50ms$. Os pontos em vermelho representam o piloto automático acionado, enquanto que os pontos em azul representam operação manual, com o objetivo de retirar o veículo da trajetória de referência para verificar o tempo de resposta até convergir novamente para a trajetória guia.

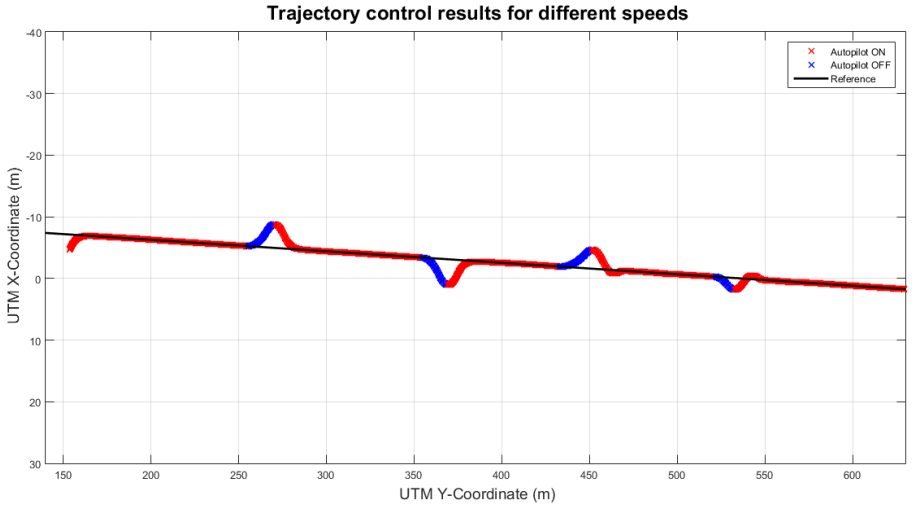


FIGURA 5.8: Trajetória percorrida e trajetória de referência.
(Fonte: Autor)

Como pode ser visto na Figura 5.9, o sistema convergiu rapidamente com erro nulo em regime permanente em todas as situações. Do início do teste até a amostra de número 2000, a velocidade do veículo permaneceu constante e igual a $8km/h$. A partir da amostra de número 2000, a velocidade foi alterada para $15km/h$, o que acarretou em um pequeno sobresinal, que é diretamente proporcional à velocidade do veículo. O

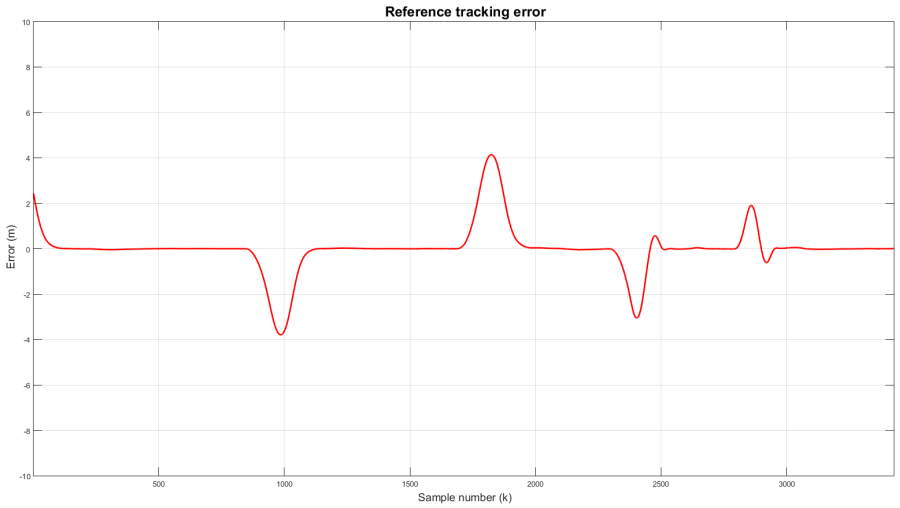


FIGURA 5.9: Erro de seguimento de trajetória.
(Fonte: Autor)

tempo de acomodação a $8km/h$ é de cerca de $7.1s$, considerando uma distância inicial de cerca de $4m$ da trajetória de referência. Já a $15km/h$, o tempo de acomodação gira em torno de $8.5s$, a uma distância inicial de cerca de $3m$ da trajetória de referência.

Além desse teste, no qual foi utilizada uma reta A-B, foram realizados outros testes com trajetórias curvas e circulares, comprovando o correto funcionamento do piloto automático também nestas situações, apesar de o erro de seguimento de trajetória em curvas ser mais acentuado. As trajetórias percorridas pelo piloto automático podem ser vistas na Figura 5.10.

Na tela de cima da Figura 5.10 é possível visualizar a tela do computador de bordo Titanium no modo 3D e notar que a trajetória percorrida pelo veículo segue a linha de referência. O rastro deixado em cinza corresponde a primeira passada, já o rastro em azul indica que houve sobreposição de passadas. Na tela de baixo da Figura 5.10 é possível visualizar a tela do computador de bordo Titanium no modo 2D a partir de uma vista superior. De maneira geral pode-se dizer que o piloto automático funcionou corretamente para reta A-B, curva A-B e pivô, o que compõe todas as classes de trajetórias possíveis de serem geradas com o equipamento, indicando compatibilidade na adequação dos módulos importados do Driver e na implementação das interfaces com a aplicação do Titanium, sensores e atuadores.

A partir desse teste foi possível validar a interface com os atuadores e o funcionamento

FIGURA 5.10: Trajetórias de referência em curva e trejetória do veículo a partir de uma visão 3D, acima, e 2D, abaixo.
(Fonte: Autor)

lógico de toda a malha de controle do piloto automático. Os testes em bancada serviram de base para o teste a ser realizado em campo, que será abordado na seção seguinte.

5.2.2.2 Testes em campo

Os testes em campo são a etapa final de validação do sistema, onde ele é exposto a condições reais de operação que, por muitas vezes, não são possíveis de serem reproduzidas em sua totalidade nos testes em bancada.

Para o teste em questão foi projetado um suporte, de forma a acoplar o computador de bordo Titanium na estrutura do trator, para reduzir possíveis vibrações e oscilações indesejadas que seriam bastante prejudiciais para a estimação da orientação do veículo. Uma antena GNSS foi posicionada no teto do trator e ligada diretamente no computador de bordo. Também foi preciso acoplar o atuador LeicaDS ao volante do trator e fazer

as conexões necessárias dos chicotes elétricos. O esquema de montagem do sistema no trator pode ser visto na Figura 5.11.

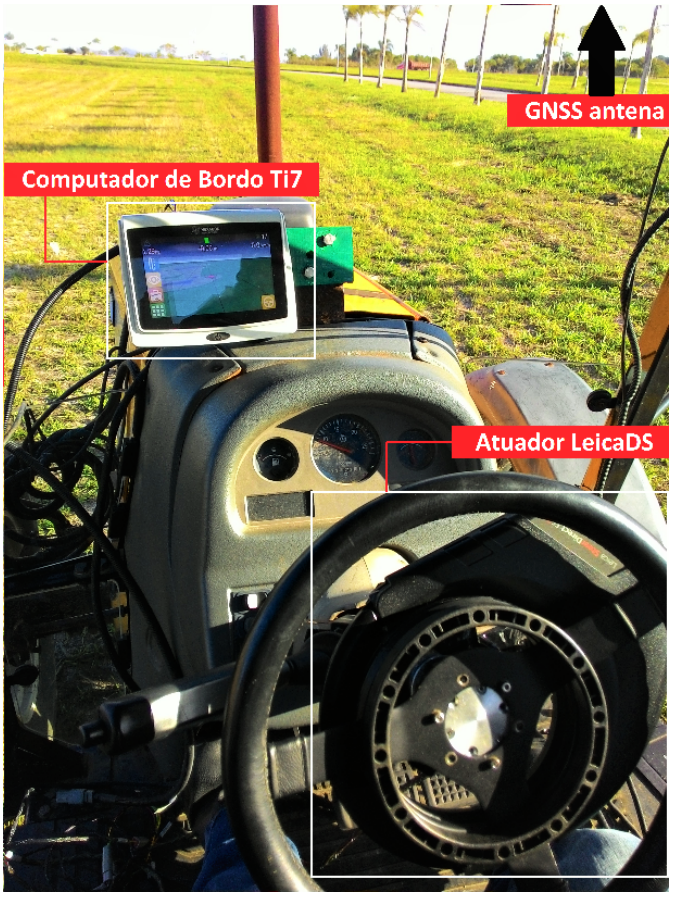
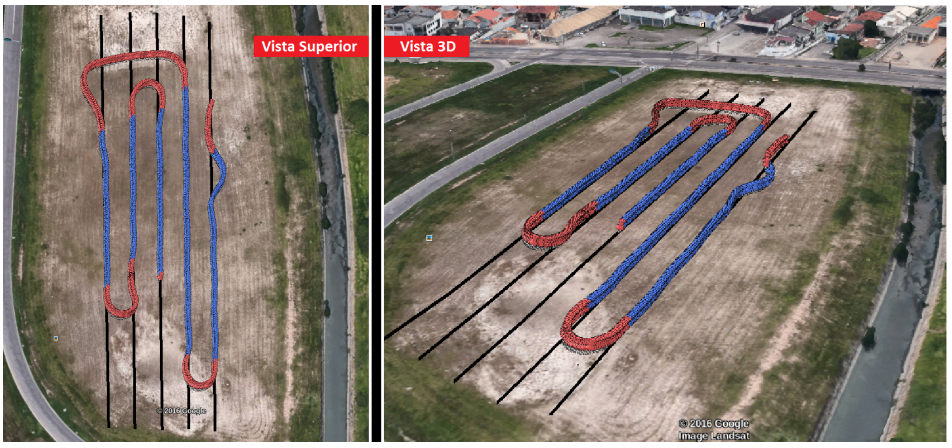


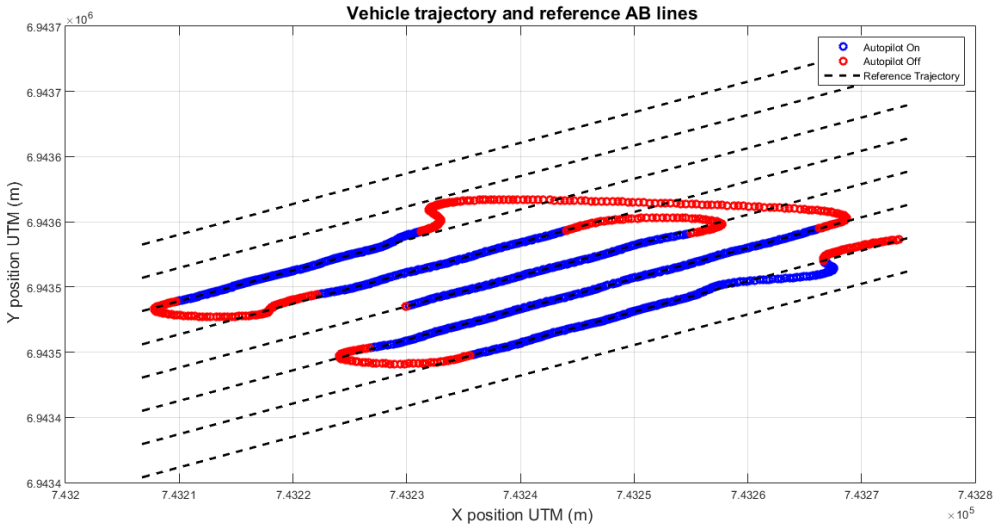
FIGURA 5.11: Montagem do sistema de piloto automático com o atuador LeicaDS no trator.
(Fonte: Autor)

Antes de testar de fato o piloto automático, é necessário fazer uma calibração do *bias* do giroscópio e do *offset* dos ângulos de *roll*, *pitch* e *yaw* através da interface presente no computador de bordo. É também preciso configurar os dados do veículo tal como distância entre rodas, posição, altura e deslocamento longitudinal da antena GNSS. Por fim, é preciso calibrar a relação volante-roda através de uma sequência de operações indicadas pelo computador de bordo. Essas calibrações são específicas para cada veículo e devem ser feitas uma única vez, após a instalação do sistema.

Para realizar o teste foram geradas retas A-B com espaçamento de 10 metros entre elas. A área possui um comprimento de cerca de 160 metros por 85 metros de largura e está localizada na região de Florianópolis. O terreno é em grande parte plano, com solo um pouco arenoso e algumas irregularidades. A trajetória percorrida pelo trator pode ser vista nas Figuras 5.12(b) e 5.12(a).



(a)



(b)

FIGURA 5.12: Trajetória do trator com o piloto automático ativado, em azul, e com operação manual, em vermelho.
(Fonte: Autor)

A Figura 5.12(a) foi gerada a partir dos pontos coletados pelo sistema GNSS com o auxílio do *software* Google Earth. Os pontos em azul significam que o piloto automático está ativado, enquanto que os pontos em vermelho indicam operação manual. O teste iniciou-se no ponto ao lado direito das figuras. Atualmente o sistema não está programado para efetuar manobras de cabeceira, ficando estas a critério do operador. Quando um movimento no volante feito pelo operador é detectado pelo sistema, o piloto automático é desativado e o trator passa a operar em modo manual. Assim que o piloto automático é ativado novamente pelo operador, o sistema irá buscar a linha de referência mais próxima para ser sua linha guia. A ativação do piloto automático pode ser feita através de um botão na tela do computador de bordo Titanium ou através de

um pequeno pedal que pode ser acionado com os pés.

A Figura 5.12(b) corresponde ao mesmo teste, porém gerada com o auxílio do *software* MATLAB e dá uma ideia melhor dos pontos percorridos pelo veículo, representados em metros através do sistema de coordenadas UTM (*Universal Transverse Mercator*). Pela análise das figuras, é possível verificar que o módulo de piloto automático funcionou corretamente, mantendo o trator na trajetória de referência com erro médio próximo de zero.

Os ângulos de *roll*, *pitch* e *yaw* correspondentes ao teste em questão podem ser vistos na Figura 5.13. Através dos gráficos é possível perceber que o ângulo de *yaw* mantém-se com pequenas oscilações, indicando o alinhamento do trator e manutenção deste sobre a linha guia quando o piloto está ativado. Pode-se notar as variações de $\pm 180°$ representando as manobras de cabeceira, seguidas por um novo alinhamento à trajetória de referência.

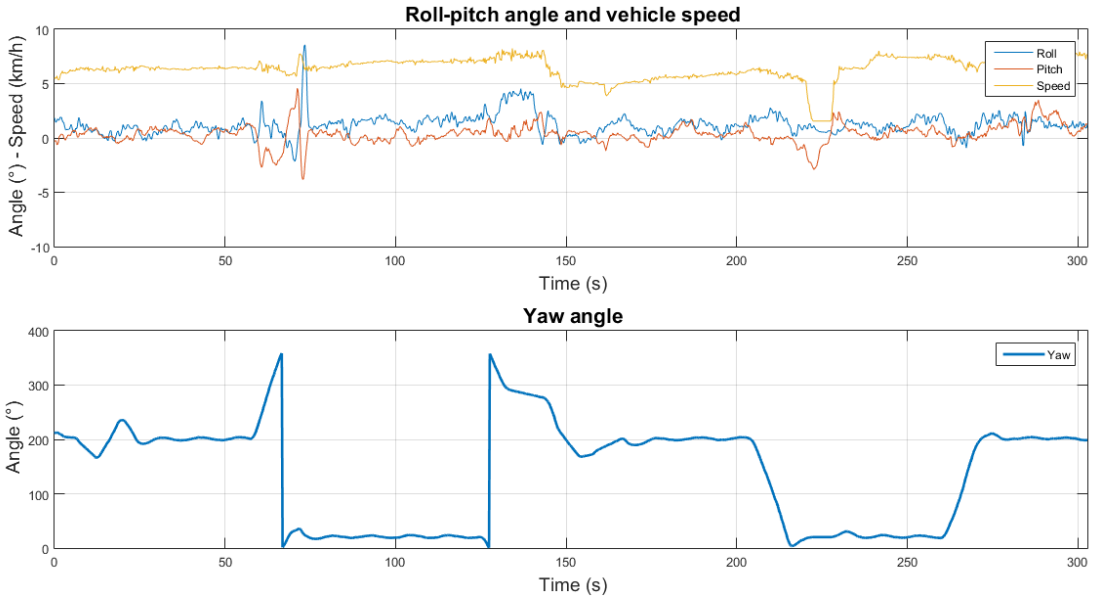


FIGURA 5.13: Montagem do sistema de piloto automático com o atuador LeicaDS no trator.
(Fonte: Autor)

É possível perceber também, em torno de $70s$, uma manobra mais brusca provocada por uma aceleração e uma manobra de curva realizadas simultaneamente, o que fez com que os ângulos de *roll* e *pitch* sofressem grandes variações.

A partir deste teste foi possível validar todo o sistema, de forma a representar grande parte das variáveis e particularidades envolvidas em uma situação real de operação.

Assim, pode-se dizer que o objetivo final do trabalho foi atingido, onde a implementação do módulo de piloto automático em uma plataforma Linux embarcada de tempo real se mostrou uma alternativa bastante interessante e funcional para o desenvolvimento de produtos futuros da empresa.

Capítulo 6

Considerações finais

Este relatório apresentou o projeto e implementação de um sistema de controle de trajetória para veículos agrícolas utilizando uma plataforma embarcada Linux de tempo real. Os desafios encontrados ao longo do trabalho passaram principalmente pela otimização do desempenho temporal e determinístico da aplicação, uma vez que esta consiste em um processo crítico em que falhas podem acarretar problemas indesejados e até mesmo acidentes com vítimas.

O projeto serviu para validar a possibilidade de desacoplar totalmente o módulo de piloto automático existente na empresa, que agora funciona sem a dependência do Driver. Essa ideia permite a economia no sentido de que não é mais necessário fabricar e montar uma placa adicional, reduzindo os custos de fabricação, montagem e a quantidade de chicotes elétricos para instalação do sistema nos tratores. Além disso, agora a empresa possui o *know-how* necessário para utilizar um *kernel* de tempo real, o que abre grandes possibilidades de otimização de tarefas já existentes na aplicação do Titanium e a criação de novos processos com requisitos temporais mais críticos.

Uma das grandes contribuições deste trabalho foi a criação de submódulos, tornando as seções de códigos genéricas e compartilhadas entre o módulo do piloto automático implementado no Driver e no Titanium. Dessa forma, alterações no funcionamento lógico, correções de bugs ou adição de funcionalidades se refletem em ambos os códigos. Futuramente, a criação de novas placas de *hardware* ou substituição dos canais de comunicação pode se dar sem maiores alterações, uma vez que o acesso às funcionalidades de *hardware* e da camada de comunicação está implementado através de interfaces.

A validação em bancada e em campo serviu para comprovar a real possibilidade de uso deste projeto nos produtos futuros da empresa, pois cumpriu integralmente seus requisitos lógicos e temporais, funcionando da maneira como se esperava. De maneira geral pode-se dizer que os objetivos propostos inicialmente foram alcançados. Apesar disso, trabalhos futuros ainda são necessários para a completa implementação da arquitetura proposta, uma vez que é necessário desenvolver e fabricar a nova *baseboard* para que se possa fazer a interface com os atuadores elétricos e hidráulicos.

Ao longo do desenvolvimento do trabalho a multidisciplinareidade envolvida foi ficando cada vez mais evidente, uma vez que é possível identificar aspectos teóricos e práticos envolvendo as áreas de informática, automação, eletrônica, controle e mecânica. Algumas das disciplinas do curso que foram fundamentais para o desenvolvimento deste trabalho são Arquitetura e Programação de Sistemas Microcontrolados, Metrologia Industrial, Sistemas de Controle, Redes de Computadores para Automação Industrial, Instrumentação em Controle, Sistemas Dinâmicos, Programação Concorrente e Sistemas de Tempo Real, dentre outras. Espera-se que este trabalho seja de grande utilidade para a empresa e também para alunos das engenharias que venham a trabalhar ou pesquisar dentro do contexto em que este trabalho está inserido, e que os desafios que o cercam possam ser bastante questionados e discutidos de maneira a buscar sempre o aperfeiçoamento e a melhor solução para os problemas encontrados.

Referências Bibliográficas

[1] J. H. Brown and B. Martin. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. *Proceedings of the Real Time Linux Workshops*, 2010.

[2] Alberto Carlos de Campo et al Bernardi. *"Agricultura De Precisão: Resultados de um Novo Olhar"*. Embrapa, 2014.

[3] Arvus: Agricultura de precisão, Ano 2007. Acesso em: abril de 2016 . URL `http://www.cnps.embrapa.br/`.

[4] "Por que um curso de Engenharia de Controle e Automação?", Acesso em: abril de 2016. URL `http://automacao.ufsc.br/`.

[5] Álvaro Vilela et al de Resende. *"Aplicações Da Agricultura de Precisão em Sistemas de Produção de Grãos no Brasil"*. Embrapa, 2014.

[6] Hexagon adquire arvus tecnologia, 2014. URL `http://www.arvus.com.br/noticias_exibe.html?id=83`.

[7] Gilad Ben-Yossef Karim Yaghmour, Jon Masters and Philippe Gerum. *"Building Embedded Linux Systems*. O'Reilly Media, 2nd edition, 2008.

[8] Gene Sally. *"Pro Linux Embedded Systems*. Apress, 2010.

[9] Jeffrey M. Osier-Mixon. Desenvolva Distribuições Integradas e Customizadas do Linux com o Projeto Yocto, Ano 2012. Acesso em: maio de 2016. URL `http://www.ibm.com/developerworks/br/library/l-yocto-linux/`.

[10] Qing Li. *"Real-Time Concepts for Embedded Systems"*. CRC Press, 1 edition, 2003.

[11] Andrew S. Tanenbaum e Albert S. Woodhull. *"Sistemas Operacionais: Projeto e Implementação"*. Bookman, 3 edition, 1999.

[12] Linet Wikia. Qt d-bus, Acesso em: maio de 2016. URL `http://linet.wikia.com/wiki/Qt_D-Bus`.

[13] Michael Kerrisk. *"The Linux Programming Interface"*. No Starch Press, 2010.

[14] Mark Mitchell. *"Advanced Linux Programming"*. No Starch Press, 2010.

[15] I2c bus, interface and protocol, Acesso em: maio de 2016. URL `http://i2c.info/`.

[16] John Catsoulis. *"Designing Embedded Hardware"*. O'Reilly Media, 2 edition, 2005.

[17] Universidade do Porto. Capítulo 3: Protocolo de comunicação CAN, Acesso em: maio de 2016. URL `http://i2c.info/`.

[18] B. Thuilot et al. Automatic guidance of a farm tractor along curved paths, using a unique CP-DGPS. *International Conference on Inteligent Robots and Systems*, 2001.

[19] Mark Pedley. *"Tilt Sensing Using a Three Axis Accelerometer"*. Freescale Semiconductor, March 2013. AN3461.

[20] S. Godha. "Performance Evaluation of Low Cost MEMS-Based IMU Integrated with GPS for Land Vehicle Navigation Integration", Fevereiro 2006. Dissertação de Mestrado.

[21] Luotau Fu and Robert Schwebel. RT PREEMPT HOWTO, May 2016. URL `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`.

[22] George Lima e Luciano Barreto Paul Regnier. Avaliação do determinismo temporal no tratamento de interrupções em plataformas de tempo real Linux. *Pós-graduação em Mecatrônica, Universidade Federal da Bahia.*

[23] Leica Geosystems. *"Product Specification - Leica SteerDirect ES Plus"*, 2016.

[24] Colin King. Stress-ng, Acesso em: julho de 2016. URL `http://kernel.ubuntu.com/~cking/stress-ng/`.