

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RÔMULO DA ROSA CORRÊA

Implementação de um controle de missões para Veículos Aéreos Não-Tripulados utilizando a plataforma Raspberry Pi

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do Curso de Engenharia de Computação: Prof. Raul Fernando Weber

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço aos meus pais que sempre me deram toda força e incentivo para batalhar e ultrapassar os obstáculos impostos pela vida. Foi com eles que aprendi os valores essenciais que uso como norteadores de moral e ética.

Também agradeço aos meus amigos que sempre estiveram comigo em cada fase da vida até aqui. Todos fizeram parte das minhas maiores vitórias.

Sou grato a todos professores que já tive. Cada um foi responsável por acrescentar uma pedra aos pilares fundamentais de todo conhecimento que adquiri. Em especial, um agradecimento ao meu orientador neste trabalho.

Em seguida, agradeço à Universidade Federal do Rio Grande do Sul por proporcionar a formação de altíssimo nível. E também à Technische Universiteit Eindhoven por me receber como intercambista por um ano durante a graduação.

Gostaria de agradecer a toda sociedade brasileira que contribui para que possamos ter educação superior pública. E finalmente, agradecer ao governo federal e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico pela concessão das bolsas de pesquisa e intercâmbio acadêmico durante este período.

RESUMO

O uso de múltiplas plataformas computacionais é uma forma de aumentar as capacidades de Veículos Aéreos Não-Tripulados (VANTs), visto que habilita o processamento de dados como imagens de câmeras e dados sensoriais a bordo da aeronave, e influencia o planejamento da missão. Este trabalho propõe a implementação de uma aplicação de controle de missão para Veículos Aéreos Não-Tripulados em uma plataforma Raspberry Pi que seja capaz de configurar e controlar algumas tarefas de uma missão como decolagem, pouso, navegação entre posições e envio de itens de missão; e estabelecer a integração dessa plataforma embarcada com o piloto automático Pixhawk através do uso de uma interface de comunicação serial. Os principais objetivos desse estudo são: proporcionar um controle com certo grau de autonomia que seja capaz de executar algumas tarefas responsáveis por estágios iniciais de uma missão sem necessitar de intervenção humana; e permitir o uso de mais módulos de processamento a partir do Raspberry Pi através de uma interface de comunicação serial padrão, sem causar um excesso de processamento no piloto automático. Esse controle é projetado como uma máquina de estado onde cada um de seus estados é uma tarefa a ser executado na sequência de eventos de uma missão. A comunicação entre o computador embarcado e o piloto automático se dá através de mensagens MAVLINK que podem conter comandos ou informações de status. Os resultados mostram que esse controle de missão é funcional e capaz de enviar itens de missão e outros comandos de navegação ao piloto automático. O módulo de controle também se mostrou capaz de tratar as mensagens recebidas e atribuir tarefas pertinentes ao estágio da missão em curso.

Palavras-chave: VANTs. Processamento de dados. Controle de missão. Interface de comunicação.

Implementation of a mission control for Unmanned Aerial Vehicles using a Raspberry Pi

ABSTRACT

The usage of multiple computational platforms is a way to increase Unmanned Aerial Vehicles (UAVs) capabilities. It enables on-board payload data processing, such as camera footage and sensory data, and influences the mission planning. This work proposes the implementation of an application for mission control for Unmanned Aerial Vehicles in a Raspberry Pi platform, that is capable of configuring and controlling some tasks in a mission such as taking off, landing, navigation between locations and the delivery of mission items; and establishing the integration between the embedded computer and the autopilot Pixhawk through the use of a serial communication interface. The main goals of this study are: providing a control with some degree of autonomy in which the execution of some tasks from early stages of a mission is possible without the need for human intervention; and allowing more processing modules to be executed on the Raspberry Pi without an autopilot overhead. This automated control is designed as a state machine in which each of its states is a task to be executed during the sequence of events in a mission. The communication between the embedded computer and the autopilot is through MAVLINK messages containing commands or status information. The results show that such control is functional and capable of delivering mission items as well as other navigation commands to the autopilot. The mission control module is also shown as capable of handling incoming messages and assigning pertinent tasks according to the mission stage.

Keywords: UAVs. Data processing, Mission control. Communication interface.

LISTA DE FIGURAS

Figura 2.1 – Estrutura geral de um sistema aéreo não-tripulado	13
Figura 2.2 – Classificação geral dos VANTs	15
Figura 2.3 – VANT de asa fixa	16
Figura 2.4 – VANT de asa rotativa	16
Figura 2.5 – Graus de autonomia entre autoridade total do piloto e autonomia total do VANT... 16	
Figura 2.6 – Estrutura física de VANTs	20
Figura 2.7 – Graus de liberdade em um quadróptero.....	20
Figura 2.8 – Giroscópio	23
Figura 2.9 – Acelerômetro.....	23
Figura 2.10 – Configuração de motores e hélices de um quadróptero.....	24
Figura 2.11 – Servomotor em um VANT.....	25
Figura 2.12 - Exemplo de pontos de interesse em um espaço de missão	28
Figura 2.13 – Sistema de coordenadas WGS84	29
Figura 4.1 – Arquitetura de integração de sistemas.....	33
Figura 4.2 – Plataforma de voo IRIS+.....	34
Figura 4.3 – Raspberry Pi	36
Figura 4.4 – Diagrama de comunicação entre Raspberry Pi e Pixhawk.....	38
Figura 4.5 – Conexão física entre Pixhawk e Raspberry Pi	40
Figura 4.6 – Função dos pinos de entrada e saída no Raspberry Pi	40
Figura 4.7 – Módulos da implementação do controle de missão	41
Figura 4.8 – Classes da interface de comunicação	42
Figura 4.9 – Diagrama de comunicação de mensagens MAVLINK.....	43
Figura 4.10 – Classe e <i>struct</i> da estrutura de máquina de estado	43
Figura 4.11 – Máquinas de estado do controle de missão	45
Figura 4.12 – Estrutura de variáveis de controle	46
Figura 5.1 – Estrutura de teste com computador e IRIS+	48
Figura 5.2 – Estrutura de teste com Raspberry Pi e IRIS+.....	49
Figura 5.3 – Topologia de rede utilizada nos testes	49
Figura 5.4 – Instruções de uso da interface	50
Figura 5.5 – Máquina de estado teste	51
Figura 5.6 – Primeira parte da saída no console da máquina virtual.....	54
Figura 5.7 – Segunda parte da saída no console da máquina virtual.....	55
Figura 5.8 – Terceira parte da saída no console da máquina virtual	56
Figura 5.9 – Protocolo de envio de <i>waypoints</i>	57
Figura 5.10 – Acesso e compilação no Raspberry Pi	58
Figura 5.11 – Primeira parte da saída no console do Raspberry Pi	59
Figura 5.12 – Segunda parte da saída no console do Raspberry Pi	60
Figura 5.13 – Terceira parte da saída no console do Raspberry Pi	60
Figura 5.14 – Visualização da missão pelo Mission Planner	61

LISTA DE TABELAS

Tabela 2.1 – Classificação dos VANTs por aplicação	17
Tabela 2.2 – Classificação de VANTs por peso	18
Tabela 4.1 – Especificações técnicas da plataforma IRIS+	35
Tabela 4.2 – Comparativo de computadores embarcados	36
Tabela 4.3 – Formato do pacote MAVLINK	39
Tabela 5.1 – Sequência de eventos para teste do controle de missão.....	52
Tabela 5.2 – Exemplo de temporização nos estados da máquina.....	58
Tabela 5.3 – Métricas sobre os módulos	62

LISTA DE ABREVIATURAS E SIGLAS

ARM	Máquinas RISC Avançadas (<i>Advanced RISC Machines</i>)
CPU	Unidade de Processamento Central (<i>Central Processing Unit</i>)
CRC	Verificação Cíclica de Redundância (<i>Cyclic Redundancy Check</i>)
EUA	Estados Unidos da América
GCC	Grande Conjunto Canônico
GNU	GNU Não É Unix (<i>GNU's Not Unix</i>)
GPRS	Serviço de Pacote de Rádio Geral (<i>General Packet Radio Service</i>)
GPS	Sistema de Posicionamento Global (<i>Global Positioning System</i>)
GPU	Unidade de Processamento Gráfico (<i>Graphic Processing Unit</i>)
HDMI	Interface Multimídia de Alta Definição (<i>High Definition Multimedia Interface</i>)
ID	Identificador (<i>Identifier</i>)
IERS	Sistema Internacional de Rotação e Referência da Terra (<i>International Earth Rotation and Reference System</i>)
IP	Protocolo de Internet (<i>Internet Protocol</i>)
IRM	Meridiano de Referência IERS (<i>IERS Reference Meridian</i>)
IRP	Polo de Referência IERS (<i>IERS Reference Pole</i>)
ISM	Industrial, Científico e Médico (<i>Industrial, Scientific and Medical</i>)
LED	Diodo Emissor de Luz (<i>Light-Emitting Diode</i>)
MAVLINK	Enlace de Micro Veículo Aéreo (<i>Micro Air Vehicle Link</i>)
MOSA	Conjunto de Sensores Orientado à Missão (<i>Mission-Oriented Sensors Array</i>)
NUC	Próxima Unidade de Computação (<i>Next Unit of Computing</i>)
POI	Ponto de Interesse (<i>Point of Interest</i>)
PWM	Modulação por Largura de Pulso (<i>Pulse Width Modulation</i>)
RAM	Memória de Acesso Aleatório (<i>Random Access Memory</i>)
SSH	Cápsula Segura (<i>Secure Shell</i>)
SSP	Protocolo de Sensores Inteligente (<i>Smart Sensor Protocol</i>)
UFRGS	Universidade Federal do Rio Grande do Sul
USB	Barramento Serial Universal (<i>Universal Serial Bus</i>)
VANT	Veículo Aéreo Não-Tripulado
WGS84	Sistema Geodético Mundial 1984 (<i>World Geodetic System 1984</i>)

SUMÁRIO

1 INTRODUÇÃO	11
2 CONTEXTUALIZAÇÃO	13
2.1 Sistemas Aéreos Não-Tripulados	13
2.2 Veículos Aéreos Não-Tripulados	14
2.2.1 Classificação Geral	14
2.2.1.1 Sustentação	15
2.2.1.2 Voo	16
2.2.1.3 Aplicação	16
2.2.1.4 Escopo de Tecnologia e Disponibilidade	17
2.2.1.5 Peso	18
2.2.2 Componentes de Genéricos de Arquitetura	19
2.2.2.1 Corpo da Aeronave	19
2.2.2.2 Fonte de Alimentação	21
2.2.2.3 Computação Embarcada	21
2.2.2.4 Sensores	22
2.2.2.5 Atuadores	24
2.2.2.6 Comunicação	25
2.3 Cenário de Utilização	26
2.3.1 Missão	27
2.3.2 Controle de Missão	27
2.3.3 Pontos de Interesse	28
2.3.3.1 Coordenadas Geográficas	28
3 TRABALHOS RELACIONADOS	30
4 CONTROLE DE MISSÃO EMBARCADO	32
4.1 Apresentação	32
4.2 Arquitetura de Integração	33
4.3 Especificação da Aeronave	34
4.4 Especificação do Computador Embarcado	35
4.5 Comunicação	38
4.5.1 Protocolo MAVLINK	39
4.5.1.1 Formato do Pacote MAVLINK	39
4.5.2 Conexão Física	39
4.6 Implementação do Controle de Missão	41
4.6.1 Interface de Comunicação Serial	41
4.6.2 Estrutura de Máquinas de Estados	43
4.6.2.1 Arquitetura da Máquina de Estados Proposta	44
5 TESTES E RESULTADOS	48
5.1 Estruturas de Teste	48
5.2 Descrição dos Testes	50
5.2.1 Verificação da Biblioteca MAVLINK	50
5.2.2 Verificação da Estrutura de Desenvolvimento de Máquinas de Estado	51
5.2.3 Verificação do Controle de Missão na Máquina Virtual	52
5.2.4 Verificação do Controle de Missão no Raspberry Pi	56
5.2.5 Visualização no Aplicativo Mission Planner	61
5.2.6 Métricas	62
6 CONCLUSÕES	63

REFERÊNCIAS	65
APÊNDICE A - Arquivo de cabeçalho para estrutura de máquina de estado (fsm.h)....	68
APÊNDICE B - Arquivo da estrutura de máquina de estado (fsm.cpp).....	69
APÊNDICE C - Arquivo de cabeçalho para o controle de missão (missioncontrol.h)	72
APÊNDICE D - Arquivo do controle de missão (missioncontrol.cpp).....	74
ANEXO A - Mensagens de missão mavlink utilizadas.....	88

1 INTRODUÇÃO

A capacidade de interação do sistema em ambientes de difícil acesso, a velocidade na coleta de dados, a redução do risco de operadores humanos a bordo, assim como os custos reduzidos para aquisição e manutenção de veículos aéreos não-tripulados (VANTs) tem tornado esta classe de aeronaves como uma opção cada vez mais viável para diversas aplicações. Jenkins (2013) prevê que a adoção de novas regras que habilitam o uso comercial de VANTs no espaço aéreo vai influenciar diretamente na rápida absorção desses dispositivos pelo mercado. Além disso, Jenkins (2013) também conclui que o maior segmento a se beneficiar do uso de VANTs é o mercado agrícola, o qual possui potencial para se criar métodos mais confiáveis, seguros e com melhor custo benefício não apenas para grandes plantações, mas também para uma variedade de soluções na agricultura familiar.

O uso de VANTs também pode ser qualificado para muitos outros propósitos como monitoramento de área desmatada, operações de busca e resgate, entrega de carga, inspeção de áreas de desastre que apresentam risco ao homem, monitoramento de tráfego urbano e inspeção de redes de alta tensão. Muitas vezes são escolhidos VANTs de pequeno porte para executar tais tarefas de forma que essas aeronaves, que operam em pequenas distâncias e possuem limite de carga, devem possuir grande disponibilidade, robustez de operação, reponsividade a informações de seus sensores, além da necessidade de coleta e processamento de dados. Entretanto, esses VANTs geralmente são controlados manualmente por operadores humanos, os quais são responsáveis por tratar o plano de voo e as respostas a erros, falhas de sistema e fatores externos. Portanto, a performance geral de um VANT para uma aplicação é limitada pelas capacidades do operador humano, o que pode não ser a melhor solução para atender os requisitos de tal aplicação.

Dessa forma, este trabalho de graduação propõe a implementação de um módulo de controle de missão para VANTs que será executado a partir de um computador embarcado acoplado ao VANT e que não necessite de intervenção de um controlador humano durante sua execução. E assim, estabelecer um gerenciamento inicial sobre as missões a fim de se obter maior qualidade nas decisões de planejamento de voo que aquelas obtidas por um controle completamente manual vindo de um operador humano. Nesse sentido, esse controle de missão é um algoritmo que é capaz de tratar mensagens do formato MAVLINK e iniciar tarefas

pertinentes a uma missão como decolagem, pouso e navegação entre posições, bem como enviar itens de missão ao piloto automático de um VANT.

No que se segue, esse trabalho apresenta: no capítulo 2, uma contextualização sobre veículos aéreos não-tripulados, seus componentes, sua utilização e suas classificações; no capítulo 3, um breve estudo sobre trabalhos relacionados; no capítulo 4, as especificações dos dispositivos utilizados para o desenvolvimento do módulo proposto, bem como a descrição da implementação geral desenvolvida; no capítulo 5, uma discussão sobre os resultados obtidos; e o capítulo 6, as conclusões finais sobre o trabalho realizado.

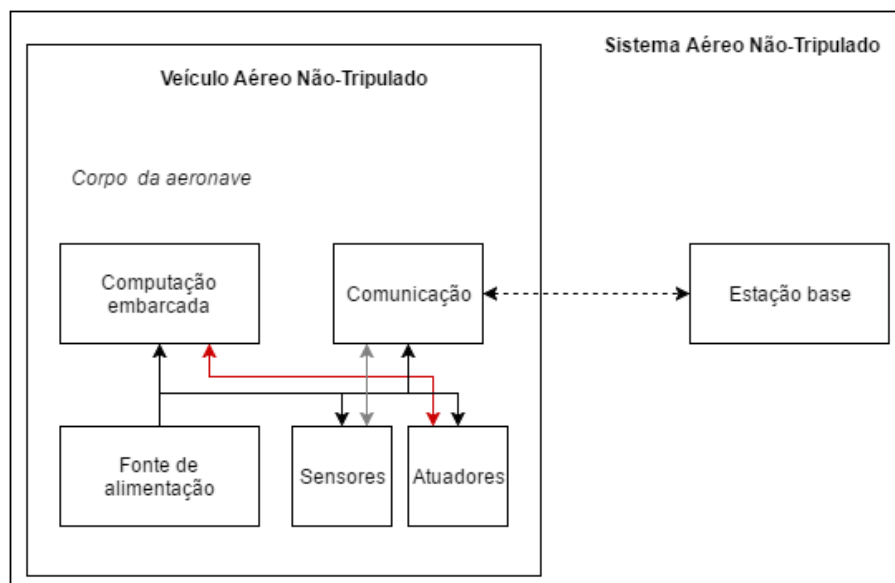
2 CONTEXTUALIZAÇÃO

Este capítulo apresenta referências e conceitos teóricos sobre nomenclaturas, ideias e características utilizadas ou mencionadas neste trabalho. Iniciando na descrição de um sistema aéreo não-tripulado e a posterior descrição e discussão sobre seus componentes e elementos característicos.

2.1 Sistemas Aéreos Não-Tripulados

Um sistema aéreo não-tripulado é composto basicamente por dois elementos principais (Figura 2.1): um veículo aéreo não-tripulado e uma estação base de comunicação. Este tipo de sistema é utilizado em situações em que é necessário uma avaliação de uma localidade de difícil de acesso, que pode representar perigo ao homem ou até mesmo em áreas que uma aeronave tripulada seria grande demais para realizar a tarefa designada. As tarefas a serem realizadas pelo VANT parte deste sistema são definidas pelo controlador humano que fica na estação base, enviando comando à aeronave. Esta aeronave pode também executar comandos com certo grau de autonomia, de acordo com o que for definido pelo controlador humano.

Figura 2.1 – Estrutura geral de um sistema aéreo não-tripulado



Fonte: Elaborada pelo autor

Como apresentado em Gertler (2012), apesar do maior número de pesquisas sobre estes sistemas e a sua utilização em mais larga escala terem ocorrido recentemente, os sistemas aéreos não-tripulados foram testados pela primeira vez durante a Primeira Guerra

Mundial. E foram utilizados em combate pela primeira vez, de fato, na guerra do Vietnã quando os EUA utilizaram o AQM-34 Firebee. Este modelo de sistema foi utilizado durante os anos 1950 como alvo de artilharia aérea, enquanto nos anos 1960 como coletor de inteligência.

Outros usos de sistemas aéreos não-tripulados para fins militares como os conflitos em Kosovo (1999), no Iraque (2003) e no Afeganistão (2001) demonstraram suas principais vantagens e desvantagens. As duas vantagens mais abordadas são a eliminação do risco de vida a um piloto e suas capacidades aeronáuticas que não ficam limitadas aos limites humanos. Já as principais desvantagens ilustradas pelo uso destes sistemas são a tendência a falhas e os riscos e complicações inerentes a voos em espaço aéreo compartilhado com aeronaves tripuladas.

Avanços recentes em tecnologias de navegação e comunicação tem tornado o uso de sistemas aéreos não-tripulados mais popular. O mundo civil moderno também tem se beneficiado do uso destes dispositivos, possibilitando aplicações em diversas áreas que careciam de soluções de mais baixo custo, risco e tempo necessário para a execução de tarefas.

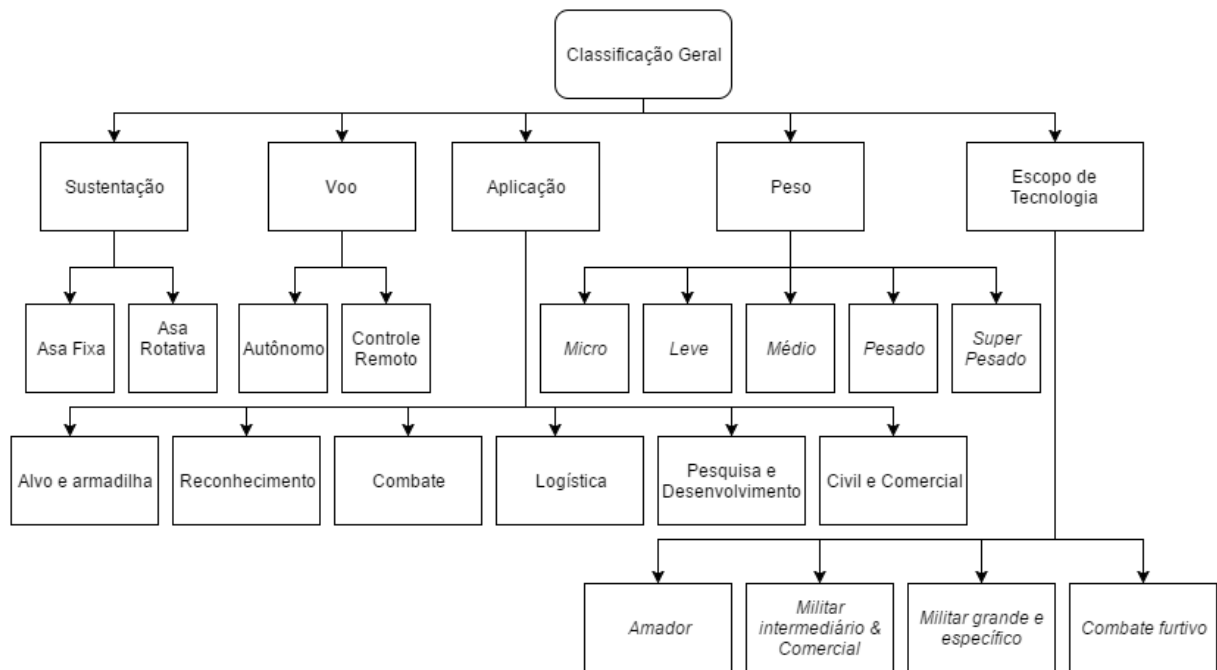
2.2 Veículos Aéreos Não-Tripulados

Um veículo aéreo não-tripulado (VANT) é definido (Department of Defense, 2005) como um veículo aéreo motorizado que não carrega um operador humano, usa forças aerodinâmicas para proporcionar ascensão ao veículo, pode voar autonomamente ou ser pilotado remotamente, pode ser descartável ou recuperável, e pode carregar uma carga letal ou não-letal. Veículos balísticos ou semibalísticos, mísseis cruzeiros, e projéteis de artilharia não são considerados veículos aéreos não-tripulados.

2.2.1 Classificação Geral

Os VANTS podem ser categorizados em cinco grandes grupos: tipo de sustentação (BEHNCK, 2014), autonomia de voo (ARNING, 2006), aplicações principais (GERTLER, 2012), escopo de tecnologia e disponibilidade (SAYLER, 2015), e peso (AGOSTINO, 2006). A Figura 2.2 ilustra a classificação proposta dos VANTS.

Figura 2.2 – Classificação geral dos VANTs



Fonte: Elaborada pelo autor

2.2.1.1 Sustentação

De forma geral, existem dois tipos principais de sustentação para VANTs: asas fixa (Figura 2.3) e asas rotativas (Figura 2.4). O primeiro funciona com a passagem de ar de forma rápida ao longo das asas que são fixas à fuselagem da aeronave. Esse tipo de VANT necessita de uma pista para pouso e decolagem (alguns modelos podem ser lançados manualmente). Este tipo de VANT geralmente é utilizado para aplicações de varredura ou reconhecimento, em que uma grande área deve ser coberta em um curto espaço de tempo, visto que essa arquitetura física permite ao VANT atingir velocidades muito maiores que aqueles de asas rotativas.

O segundo opera com a propulsão de pás dos rotores, de forma que não é necessária uma pista de pouso e decolagem pois estas operações ocorrem verticalmente, esta configuração também possibilita que este tipo de aeronave paire no ar. VANTs de asa rotativa ainda podem ser subclassificados de acordo com o número de rotores em sua configuração. As configurações mais usuais são 3, 4, 6 e 8, sendo denominados tricópteros, quadcópteros, hexacópteros e octacópteros, respectivamente. Este tipo de aeronave é mais utilizada para aplicações de monitoração e inspeção, em que é necessário um maior detalhamento da região monitorada, visto que esta arquitetura permite a manutenção e controle da aeronave em certa posição no ar.

Figura 2.3 – VANT de asa fixa



Fonte: Wikimedia Commons

Figura 2.4 – VANT de asa rotativa

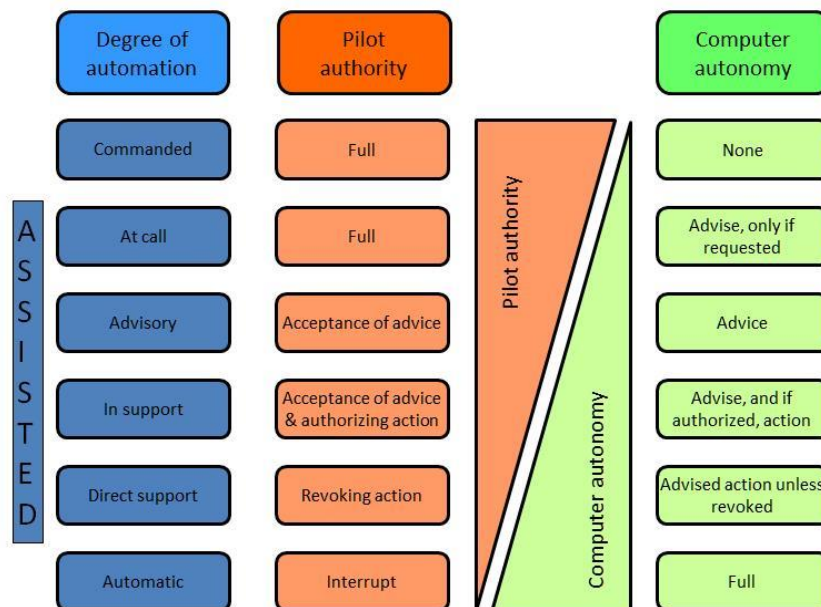


Fonte: Wikimedia Commons

2.2.1.2 Voo

No escopo deste capítulo a palavra autonomia se refere à autonomia de controle. Uma aeronave pode ser classificada por grau de autonomia do sistema e autoridade do piloto. Os possíveis graus de autonomia são caracterizados na Figura 2.5, a qual demonstra que para cada grau maior de autonomia computadorizada, menor será o grau de autoridade do piloto.

Figura 2.5 – Graus de autonomia entre autoridade total do piloto e autonomia total do VANT



Fonte: Arning, Heinzinger, Langmeier, Schertler, Sobotta (2006).

2.2.1.3 Aplicação

O uso de VANTs pode ser dividido em duas grandes áreas: militar e civil. Na área militar existem algumas missões em que é possível reduzir o risco humano ao se utilizar VANTs, tais como vigilância e reconhecimento de determinada área de risco, perseguição de alvos, detecção de ameaças no ar, no mar ou na terra, busca de sobreviventes em locais de

desastres naturais de difícil acesso ou ainda, monitoramento de vazamentos de óleo ou outras substâncias perigosas ao homem.

Já na área civil existem diversas aplicações comerciais como monitoramento de redes de alta tensão, agricultura de precisão (DOERING, 2014), monitoramento de cabeças de gado e transmissão de imagens aéreas. Além disso, existem possíveis aplicações científicas como modelagem e teste de voos e redes de internet móveis (GOOGLE, 2013). A Tabela 2.1 detalha a classificação por tipo de aplicação como foi descrito na Figura 2.1.

Tabela 2.1 – Classificação dos VANTs por aplicação

<i>Tipo</i>	<i>Aplicações de exemplo</i>
Alvo e armadilha	Agir como isca para estimular a ação de uma aeronave ou míssil inimigo
Reconhecimento	Reconhecimento de área, alvos ou inimigos
Combate	Ataque a alvos Defesa contra inimigos Alvos para treinamentos de combate
Logística	Entrega de medicamentos e alimentos em áreas remotas Entrega de encomendas
Pesquisa e desenvolvimento	Desenvolvimento de novas tecnologias de VANTs a serem futuramente integradas em veículos reais
Civil e comercial	Vigilância de plantações Filmagem acrobática Operações de busca e resgate Inspeção de redes de elétricas e oledutos Estimação de número de animais selvagens Monitoração de segurança em áreas urbanas Monitoração de poluição Pesquisas para exploração de óleo, gás e mineral Proteção de sítios arqueológicos

Fonte: Saylor (2015)

2.2.1.4 Escopo de Tecnologia e Disponibilidade

Nesta seção os VANTs são categorizados de acordo com suas tecnologias e sua disponibilidade de acesso aos usuários. A Figura 2.2 ilustra quatro categorias deste escopo para aeronaves sem piloto definidas por Saylor (2015).

A primeira é denominada *Amador* e inclui os VANTs prontamente disponíveis para venda a qualquer tipo de usuário. Estes sistemas são pré-montados ou montados a partir de componentes e não requerem uma infraestrutura formal ou treinamento para sua operação. A segunda nomeada *Militar intermediário & Comercial* caracteriza as aeronaves que não estão usualmente disponíveis a pessoas comuns devido ao seu custo ou requerimentos de

infraestrutura. Estes dispositivos podem, no entanto, ser transferidos a exércitos estrangeiros ou a atores privados.

A terceira categoria é chamado de *Militar grande e específico* e geralmente inclui aeronaves armadas, que requerem infraestrutura militar substancial e não são acessíveis ou operadas por atores fora de grandes exércitos. A última é denominada *Combate furtivo* e engloba VANTs com tecnologias altamente sofisticadas, como características de baixa visibilidade (menos visíveis a radares e sonares, por exemplo), e que não são vendáveis a estrangeiros. Os Estados Unidos são a única nação conhecida que opera este tipo de veículo aéreo.

2.2.1.5 Peso

Os veículos aéreos não-tripulados também podem ser classificados de acordo com seu peso e esta classificação geralmente é associada com o tipo de aplicação em que um VANT pode ser utilizado. A Tabela 2.2 traduz a nomenclatura destas categorias e exemplifica com aeronaves conhecidas no mercado de VANTs.

Tabela 2.2 – Classificação de VANTs por peso

<i>Classe</i>	<i>Peso</i>	<i>Exemplo</i>
Micro	< 5 kg	Dragon Eye
Leve	5 – 50 kg	RPO Midget
Médio	50 – 200 kg	Raven
Pesado	200 – 2000 kg	A-160
Super Pesado	> 2000 kg	Global Hawk

Fonte: Agostino, Mammone, Nelson, Zhou.

A categoria de peso de uma aeronave influencia muitas características como tipo de motor de propulsão, capacidade de carga, persistência e duração de voo e aplicações fim. A seguir são descritas como estas características podem ser influenciadas pelo peso da aeronave.

I. Tipo de motor de propulsão: a classe *micro* utiliza primordialmente motores elétricos, enquanto as classes de maior peso utilizam motores de combustão, em especial a classe *super pesado* normalmente utiliza turbinas a jato;

II. A carga que é capaz de carregar: quanto mais leve, proporcionalmente menor é a capacidade de carga do VANT;

III. Persistência e distância de voo: quanto mais leve, menor é a capacidade de carga de bateria ou combustível de uma aeronave e isto influencia a capacidade de manutenção desta no ar, bem como a distância máxima que se pode percorrer sem recarregamento ou reabastecimento;

IV. Aplicação: o conjunto das características anteriormente mencionadas define o melhor tipo de aplicação para cada classe de VANT, aeronaves de classe *micro* e *leve* geralmente realizam missões com menor área de atuação (poucos quilômetros quadrados) e de menor duração (poucos minutos), enquanto aquelas de classes mais pesadas realizam missões em áreas muito maiores (podendo atravessar países) e podem durar de algumas horas até alguns dias.

No escopo deste trabalho é utilizado um VANT da classe *micro* devido a sua acessibilidade, baixo custo, pelo objetivo de utilizá-lo para missões de curta duração (até vinte minutos) em um espaço de poucos quilômetros de distância e pelo baixa necessidade de capacidade de carga.

2.2.2 Componentes Genéricos de Arquitetura

Veículos aéreos que pertencem a uma mesma categoria possuem, no geral, características físicas bem semelhantes. E assim, no caso dos VANTs, é possível definir uma arquitetura generalizada (Figura 2.1) que representa a grande maioria dos sistemas hoje existentes.

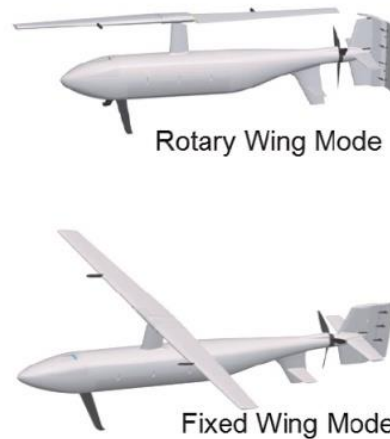
Nesta sub-seção são apresentados os conceitos gerais da arquitetura de um veículo aéreo não-tripulado ilustrada na Figura 2.1 divididos em corpo da aeronave, fonte de alimentação, computação embarcada, sensores, atuadores e comunicação. Um sistema aéreo não-tripulado completo também possui uma estação base, na qual existe um módulo de comunicação como o do VANT. Esta estação base pode ser representada por uma estação fixa em terra (controle remoto ou aplicativo móvel, por exemplo) ou ainda outra aeronave que envia comandos àquele VANT representado na Figura 2.1.

2.2.2.1 Corpo da Aeronave

O corpo da aeronave é representado pela estrutura física que a sustenta. Esta estrutura é composta por fuselagem, asas para aeronaves de asa fixa ou braços para multirrotores e

cobertura. A Figura 2.6 ilustra esta estrutura física para dois tipos de aeronaves: um de asa fixa e um de uma asa rotativa.

Figura 2.6 – Estrutura física de VANTs

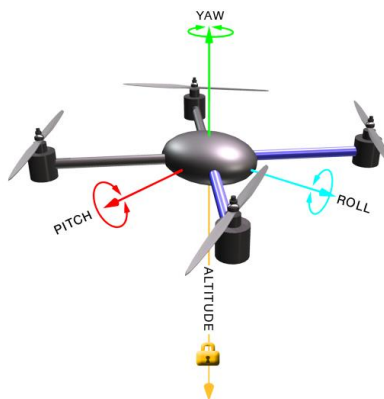


Fonte: Trenton Systems

A partir da arquitetura mecânica básica de uma aeronave se tem os graus de liberdade de movimento desta: um VANT pode rotacionar no entorno de três eixos (x , y e z) do seu centro de massa. Contudo, o controle de posicionamento geralmente é convertido para controle angular, no qual existem três ângulos que representam os graus de liberdade possíveis: ψ (*yaw*), θ (*pitch*) e ϕ (*roll*).

Yaw representa o movimento angular no eixo z (aquele perpendicular ao corpo da aeronave e, em posição de repouso do VANT, normal ao solo); *pitch* representa o movimento angular no eixo y (aquele perpendicular à frente da aeronave e, para VANTs de asa fixa, paralelo ao sentido das asas); e *roll* representa o movimento angular no eixo x (aquele paralelo à frente da aeronave, com sentido igual à frente do VANT). A Figura 2.7 apresenta a configuração dos graus de liberdade de um VANT.

Figura 2.7 – Graus de liberdade em um quadróptero



Fonte: Spswarm UAV

Estes graus de liberdade são o que possibilitam a movimentação geral, execução de curvas e inclinações no corpo do VANT, permitindo a realização de manobras comuns e outras mais ousadas que em aeronaves tripuladas poderiam oferecer grande risco a tripulação.

2.2.2.2 Fonte de Alimentação

Plataformas de VANTs são altamente adaptáveis, permitindo o uso de diferentes tipos de sensores para as mais diversas análises. A grande variedade de requerimentos resulta em uma grande quantidade de escolhas necessárias quando é feita uma análise de requerimentos de energia e as formas em que a energia é entregue aos componentes de um VANT.

Para tarefas de processamento embarcado ao VANT são utilizados CPUs e GPUs que proporcionam computação efetiva a baixo custo e potência. Estes tipos de processadores podem fornecer *gigaflops* de capacidade computacional consumindo menos de 10W de potência, conforme Piscione (2014). Contudo, estes dispositivos possuem demandas de energia que podem dificultar o projeto de uma arquitetura de VANT, visto que seus núcleos geralmente operam em baixos níveis de tensão, até 1 V, mas o sistema deve suportar também outros níveis de tensão para comunicação periféricos e acessos à memória. Processadores modernos possuem múltiplos núcleos e isso implica que suas arquiteturas tenham uma demanda de energia muito variável de acordo com a quantidade de unidades de processamento sendo utilizadas a cada instante.

O peso dos veículos aéreos é um fator crítico no projeto de alimentação de energia. É necessário um esforço para reduzir o máximo de peso possível na geração de energia e carga útil a fim de prolongar o tempo de voo, facilitar o controle e estabilidade da aeronave, e até mesmo possibilitar a decolagem em uma pista menor – no caso de VANTs de asa fixa. Células de energia solar tem grande potencial para voos de longa duração e grande altitude. Células combustíveis são fontes de energia com potencial para VANTs que necessitam atingir grandes velocidades e altitudes, e pertencem a categorias que suportam cargas mais pesadas. Já baterias a base de lítio são utilizadas primordialmente como fonte de energia para VANTs de pequeno porte (geralmente aqueles da categoria de peso micro).

2.2.2.3 Computação Embarcada

A capacidade de processamento de dados de veículos aéreos não-tripulados é baseada no uso de sistemas embarcados, os quais são responsáveis por tarefas de controle e

estabilidade do voo, leitura de informações de sensores, comunicação entre componentes da aeronave e com outros dispositivos. O computador embarcado principal que é integrado a um VANT também é referenciado como piloto automático ou controlador de voo.

Sistemas embarcados são desenvolvidos e implantados para executar tarefas específicas e geralmente são compostos por um computador encapsulado pelo dispositivo que ele controla. Quando comparados com computadores de propósito geral, os computadores embarcados possuem uma tendência a consumir menos energia, ter tamanho reduzido e ainda ter um custo por unidade menor. No entanto, possuem limitações de recursos e de capacidade de processamento. Alguns desses sistemas podem possuir restrições de performance de tempo real que devem ser obedecidas devido a questões de segurança e usabilidade. Enquanto outros podem não ter requisitos de performance, possibilitando uma simplificação da sua arquitetura de hardware a fim de reduzir custos de projeto.

O uso de computadores embarcados em VANTs possibilita a extensão das aplicações possíveis para essas aeronaves, visto que facilita a utilização de sensores, módulos de comunicação e até mesmo o acoplamento de outros computadores embarcados, ampliando a capacidade de processamento de informações, e habilita a implementação de algoritmos de mais alto nível para realizar diversos tipos de controle e automação dos voos. Essa complexidade e diversidade de aplicações de VANTs modernos e seus sistemas computacionais subjacentes tem apresentado a desenvolvedores alguns dos mais difíceis desafios de projeto. Desenvolvedores tem de lidar com uma lista de requisitos que desafiam os limites das tecnologias de computação embarcada.

De acordo com MacLaren (2012), cada tipo de fuselagem de aeronave possui um conjunto de requisitos específicos aos objetivos operacionais daquela categoria de VANT. Contudo, um grupo de requisitos chave para plataformas de computação embarcada deve ser compartilhado entre os VANTs para assegurar o sucesso de operações de *software*. Essas demandas de interoperabilidade e conectividade são dispostas na forma de interfaces padronizadas e cargas interoperáveis do tipo plugue e execute (*plug and play*). Esse tipo de padronização requer um aumento nas capacidades de computação e comunicação de um VANT, além de um maior controle de operações em tempo real.

2.2.2.4 Sensores

Sensores são elementos essenciais para o controle de voo, navegação e comunicação de um veículo aéreo. Existem dois tipos principais de sensores utilizados em VANTs e eles

são organizados pelo tipo de estímulo sentido. O primeiro corresponde aos sensores que percebem estímulos dentro da estrutura do VANT, geralmente relacionados à posição ou movimento. O segundo corresponde aos sensores que percebem estímulos externos à estrutura do VANT.

O conjunto de sensores internos ao VANT pode incluir:

I. Giroscópio (Figura 2.8), é um disco giratório em que o eixo de rotação é livre para assumir uma orientação. É utilizado como sistema de navegação inercial e para estabilização de VANTs;

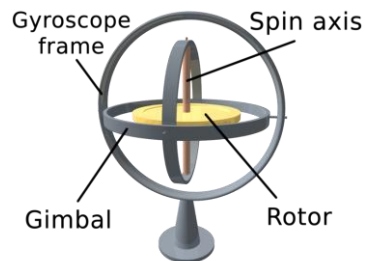
II. Acelerômetro (Figura 2.9), é um dispositivo que mede a aceleração do tipo força G e compõe o sistema de navegação inercial;

III. Compasso, é utilizado para navegação e orientação, apontando o sentido do norte magnético da Terra;

IV. Altímetro, é utilizado para medir a altitude da aeronave sobre superfícies ou sobre o solo, podendo realizar a medição por radar ou pela medida da pressão atmosférica;

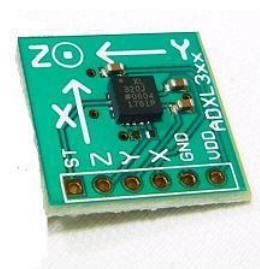
V. Módulo de GPS, mede a posição geográfica do VANT através da comunicação com satélites especializados para esta função.

Figura 2.8 - Giroscópio



Fonte: Wikimedia Commons

Figura 2.9 - Acelerômetro



Fonte: Pyroelectro.com

Enquanto o conjunto de sensores externos podem incluir:

I. Câmera, é um dispositivo óptico utilizado para capturar imagens que são posteriormente processadas por um computador embarcado ao VANT;

II. Radar, é um sistema de detecção de objetos que utiliza ondas de rádio para determinar distância, ângulo ou velocidade de objetos;

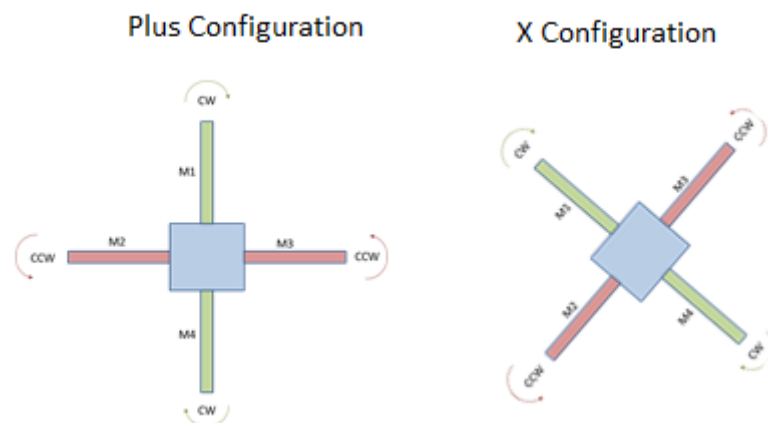
III. Sonar, é um sistema de detecção de objetos semelhante a um radar, mas que utiliza ondas sonoras para executar a detecção.

2.2.2.5 Atuadores

Os atuadores variam de acordo com o tipo de aeronave e podem ser de diferentes tipos como controladores eletrônicos de velocidade, hélices, servomotores, atuadores de carga, LEDs, alto-falantes, entre outros.

De acordo com Chao et al. (2010), superfícies de controle de VANTs de asa fixa podem incluir: ailerons para controlar o ângulo *roll*; elevadores para controlar o ângulo *pitch*; aceleradores usados para controlar a velocidade dos motores; lemes utilizados para controlar o ângulo *yaw*. No caso de VANTs de asa rotativa, superfícies de controle também podem incluir hélices que de acordo com a velocidade de cada uma e a sua configuração (exemplo de configurações para VANTs de 4 rotores na Figura 2.10) podem controlar todos os ângulos de movimento.

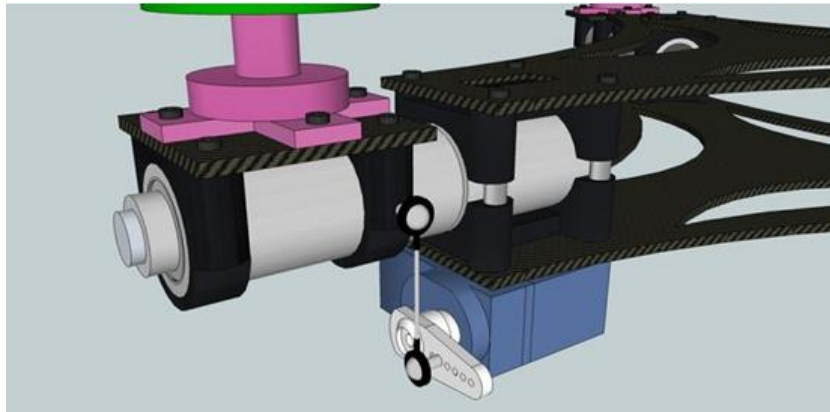
Figura 2.10 – Configuração de motores e hélices de um quadróptero



Fonte: Technical Adventures (2012)

Servomotores são os dispositivos que controlam as superfícies de controle, visto que este tipo de motor DC possui um mecanismo servo para controle preciso de posição angular. A rotação destes dispositivos é limitada entre ângulos fixos, uma vez que este tipo de motor não rotaciona continuamente. A Figura 2.11 ilustra um servomotor acoplado a um braço de um VANT de asa rotativa.

Figura 2.11 – Servomotor em um VANT



Fonte: UAV Society.

Um servomotor é colocado em uma certa posição angular através do envio de um sinal PWM no seu fio de controle. Um pulso de largura variando de um a dois milissegundos repetidamente em um quadro de tempo é enviado ao servo em torno de 50 vezes por segundo e a largura deste pulso define a posição angular.

Outros tipos de atuadores como LEDs e alto-falantes são utilizados para sinalização de status. Diferentes conjuntos de cores exibidas nos LEDs ou sons transmitidos nos alto-falantes podem representar avisos sobre as condições da aeronave como aceite ou recusa de comandos originados na estação base, alerta de bateria fraca, alerta de componentes com problemas, entre outros.

2.2.2.6 Comunicação

A necessidade de uso de sistemas de comunicação com alta confiabilidade, segurança e disponibilidade tem aumentado, visto que as aplicações de veículos aéreos não-tripulados tem se tornado cada vez mais complexas. Muitos VANTs utilizam módulos de rádio-frequência, conectando um conversor analógico-digital a uma antena que se comunica com o controlador de voo. Este sistema de comunicação pode ser usado para comunicações com uma estação base ou ainda com outros VANTs e serve para controlar remotamente a aeronave e trocar dados com informações pertinentes às missões.

A escolha de protocolos de comunicação para determinados VANTs deve levar em consideração a expectativa de uso das aeronaves, isto é, quais serão suas aplicações principais, bem como a topologia da rede esperada, o cenário de utilização e a quantidade de VANTs operando conjuntamente. Todos esses fatores influenciam características de

performance como largura de banda disponível, segurança, confiabilidade e velocidade necessária e efetiva de transmissão de dados.

Em Barnard (2007) são expostas algumas características para uso e implementação de sistemas de comunicação para VANTs: os enlaces de controle e comando de envio (*uplink*) aos VANTs necessitam levar em conta a necessidade de prevenir o uso não autorizado da conexão, de forma que o caminho de dados deve ser robusto e criptografado; enlaces de comando e telemetria devem ser considerados separadamente dos enlaces de carga útil; o enlace de reposta (*downlink*) de *status* de um VANT é relativamente simples e pode ser gerenciado por um modem GPRS ou por uma conexão via satélite; o enlace de carga útil geralmente necessita de muito mais largura de banda do que aqueles de comando e telemetria e como algumas larguras de banda necessárias para determinadas aplicações (como transmissão de vídeo) não estão prontamente disponíveis, são utilizadas bandas do padrão ISM (*Industrial, Scientific and Medical*) como 2.4GHz, 5.8GHz e 20GHz.

Outro tema discutido por Barnard (2007) é o uso compartilhado de espaço aéreo entre veículos aéreos tripulados e não-tripulados. Controladores de tráfego aéreo requerem que VANTs que compartilhem o espaço aéreo comum com aeronaves tripuladas sejam capazes de responder a comandos de voz originados nas estações de tráfego aéreo, uma vez que VANTs devem ser tratados da mesma maneira que aeronaves comuns da perspectiva de tráfego aéreo. Essa condição apresentada ainda demonstra um grande desafio a desenvolvedores e projetistas de VANTs comerciais, haja vista a latência de transmissão de voz digitalizada entre estação base, VANT e estação de controle de tráfego aéreo. Um objeto para estudos futuros proposto, nesse sentido, é implementar o uso de programas de reconhecimento de voz para automação de comandos entre VANTs e controladores de tráfego aéreo.

2.3 Cenário de Utilização

Para a melhor compreensão do cenário de utilização de um VANT é necessária a apresentação de alguns dos elementos principais deste contexto e que são fontes de muitos estudos presentes na literatura. Estes elementos são representados na execução de uma tarefa designada a uma aplicação.

2.3.1 Missão

A missão para um VANT é um conjunto de tarefas sequenciais a serem realizadas pela aeronave. Essas tarefas podem ser executadas normalmente ou podem entrar em espera caso algum componente do sistema do VANT envie notificações de emergência que precisem ser tratadas com prioridade mais alta que aquela das tarefas comuns da missão.

O conjunto de tarefas de uma missão pode ser enviado pelo controlador humano a partir da estação base ou por um controle autônomo embarcado na aeronave. Cada tarefa é um item da missão e representa um comando específico, com diversos parâmetros, a ser seguido pelo VANT. Os comandos possíveis variam de acordo com o protocolo de comunicação utilizado entre o piloto automático e o controlador da missão, bem como de acordo com o tipo veículo aéreo utilizado.

Os itens mais comuns de uma missão são: decolar, pairar sobre uma região, pousar, ir até um local, mudar posição (*yaw*, *pitch* e *roll*), alterar velocidade, voltar ao ponto de decolagem e armar ou desarmar componente. Entretanto, uma série de outros controles sobre o voo podem ser especificados em um item de missão.

2.3.2 Controle de Missão

Esta subseção apresenta o conceito utilizado como um dos objetos de estudo deste trabalho. O controle de missão de VANTs é o elemento responsável por definir e gerenciar os itens de uma missão, definir ações responsivas a estímulos oriundos de sensores ou outros dispositivos externos, bem como corrigir eventuais erros e falhas quando possível.

Esse elemento que controla missões pode ser representado por um operador humano em uma estação base ou ainda por um algoritmo que automatiza o controle. O primeiro se comunica com os VANTs através de conexões sem fio e, por se tratar de um operador humano, pode controlar a movimentação das aeronaves de forma completamente manual. O segundo pode estar executando a partir de um computador embarcado acoplado ao VANT ou ainda, executando a partir de um computador localizado na estação base.

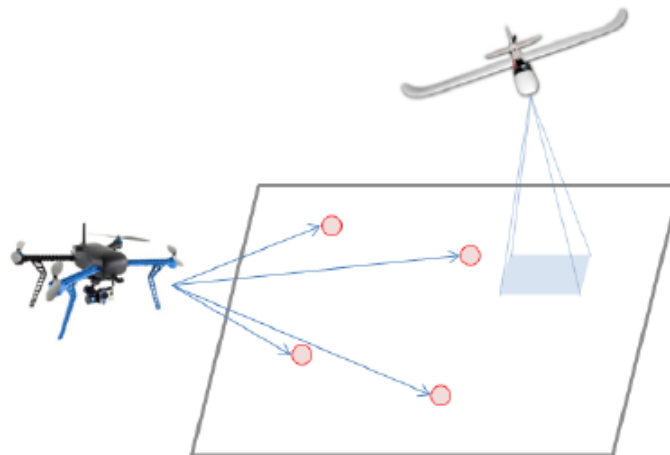
As duas representações de controle de missão citadas podem ainda ser combinadas de forma que o nível de autonomia de um algoritmo sobre os comandos do VANT é inversamente proporcional ao grau de autoridade que o operador humano possui, como apresentado na Figura 2.5.

2.3.3 Pontos de Interesse

No escopo deste trabalho, pontos de interesse (ou *points of interest*, PoI) se referem a pontos dentro do espaço de missão de um VANT, descritos na forma de coordenadas geográficas, em que existem informações úteis ao controlador e que devem ser identificadas e processadas.

Na Figura 2.12 é ilustrado um exemplo de pontos de interesse, os quais são representados por pequenos círculos e o espaço de missão que é representado por um grande paralelogramo. A imagem também ilustra a varredura do espaço de missão por um VANT de asa fixa para a identificação dos pontos de interesse, os quais serão posteriormente visitados por um VANT de asas rotativas.

Figura 2.12 – Exemplo de pontos de interesse em um espaço de missão



Fonte: 3DR Robotics

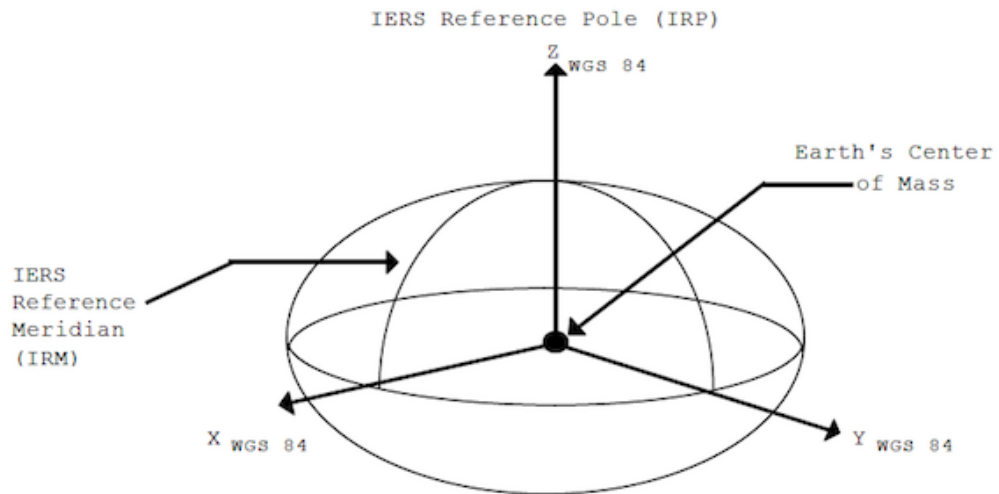
2.3.3.1 Coordenadas Geográficas

O uso de coordenadas geográficas é um dos pontos essenciais na execução de missões para veículos aéreos não-tripulados, visto que parte da autonomia da aeronave se baseia no conhecimento de pontos de interesse expressados na forma de coordenadas geográficas.

O formato padrão utilizado na maioria de sistemas de GPS comerciais (MATSUMOTO, 1993) é o WGS84 (World Geodetic System 1984). Esta norma define um sistema de coordenadas tridimensionais (Figura 2.13) e considera como coordenada de origem a localização do centro de massa da Terra. Além disso, a norma utiliza o meridiano

internacional de referência como meridiano de longitude zero e utiliza um modelo no formato de elipsoide como superfície de dados.

Figura 2.13 – Sistema de coordenadas WGS84



Fonte: National Imagery and Mapping Agency (2000)

3 TRABALHOS RELACIONADOS

Essa seção apresenta alguns dos trabalhos relacionados ao estudo de integração de plataformas e controladores de missão para veículos aéreos não-tripulados presentes na literatura.

Kannan et al. (1999) apresentam um *software* que utiliza a infraestrutura do *Open Control Platform*. Seus principais objetivos são: acomodar facilmente os requisitos de aplicações; incorporar novas tecnologias como plataformas de *hardware* e sensores; operar em ambientes heterogêneos; e manter a viabilidade em ambientes não-determinísticos. Essa proposta habilita a comunicação assíncrona em tempo real entre componentes heterogêneos distribuídos.

Além disso, é utilizada a *Common Object Request Broker Architecture* para estabelecer comunicação contínua entre objetos em diferentes computadores ou através de protocolos de rede. Essa arquitetura é utilizada para padronizar as interfaces de cada componente dos VANTs. Em um ambiente de teste os resultados obtidos mostraram a simulação de reconfiguração de um modelo de VANT com um controle secundário tolerante a falhas.

Dobrokhodov et al. (2013) demonstraram o uso do *Rapid Flight Control Prototyping System*. Esse sistema foi construído ao redor das capacidades de um piloto automático avançado com um *link* de comunicação de tempo real e um conjunto secundário de controladores, responsáveis por diversas tarefas como controle de voo, pré-processamento de dados e tarefas de comunicação robusta. Essa solução é baseada no uso de sistemas operacionais, utilizando *design* de algoritmos de controle, geração automática de código através do Simulink, verificação e validação formal. Nenhuma modificação manual da biblioteca gerada é necessária, porém é preciso definir explicitamente a taxa de amostragem de cada componente de código e mecanismos de mensagem.

Na arquitetura de teste foi utilizado um VANT com o piloto automático Piccolo, um computador para processamento de dados, outro para controle de voo e uma estação base que habilita a operação simultânea de múltiplos VANTs. A partir dos diversos voos realizados, o ambiente operacional maturou de forma que os códigos gerados formaram bibliotecas de funções testadas e validadas, visto que os dados coletados durante os voos definem características para a geração automática de código a ser modificado e executado em tempo de voo. Essa solução foi desenvolvida para VANTs de maior peso que aqueles pertencentes à categoria central de estudo dessa monografia.

Já Koeners et al. (2006) estudaram a integração de múltiplas plataformas como estação, controle e comando, e controle de tráfego aéreo. O foco foi dado na conectividade e interoperabilidade em rede, utilizando predição de conflitos com outros elementos e realizando simulações em níveis de missões militares. Os testes utilizaram operadores do comando da força aérea holandesa, bem como de centros militares de controle de tráfego aéreo e outros institutos de pesquisa e demonstraram a viabilidade de integração das diversas plataformas. A abordagem proposta de integração de VANTs com certa autonomia ao espaço aéreo comum leva em conta apenas VANTs de classe militar.

Em outro estudo, Pires et al. (2014) apresentam um novo padrão de integração entre VANTs e sua carga útil. Foi desenvolvido um protocolo baseado no modelo MOSA (do inglês, *Mission-Oriented Sensors Array*) chamado *Smart Sensor Protocol* (SSP), no qual o processamento da missão é separado do sistema de voo do piloto automático. Esse protocolo é simples, contém algumas diretivas para criação de mensagens e foi validado através de verificação formal.

Contudo, é necessário o uso de dispositivos intermediários para realizar a tradução de mensagens SSP para mensagens do formato utilizado no piloto automático e vice-versa. Além de ser um protocolo novo, o SSP não foi testado em missões reais e também demonstra ser bem limitado nas possibilidades de comandos e parâmetros de navegação se comparado a outros protocolos já bem estabelecidos e amplamente utilizados como o MAVLINK.

4 CONTROLE DE MISSÃO EMBARCADO

Nesta seção é apresentada a definição, motivação e desenvolvimento do estudo realizado, bem como a descrição das plataformas escolhidas para a execução do trabalho: o computador embarcado Raspberry Pi e o veículo aéreo não-tripulado IRIS+.

4.1 Apresentação

O fato do planejamento de missões ser feito por um operador humano de forma manual pode implicar na limitação do escopo das aplicações de VANTs. A necessidade de acompanhamento do operador humano durante toda a missão sob o ponto de vista do planejamento de rotas e tomada de decisões devido a influência de fatores como vento, carga da bateria, obstáculos e falhas pode acarretar na diminuição da qualidade de decisões de acordo com a situação da missão, pois a grande quantidade de variáveis influenciadoras na tomada de decisão gera um aumento da complexidade de gerenciamento da missão

Neste contexto, a proposta deste trabalho é realizar a integração de um computador embarcado com um VANT, implementando uma interface de comunicação entre estes sistemas para que esta seja uma plataforma base para executar um controle de missão autônomo e habilitar a execução de algoritmos de mais alto nível.

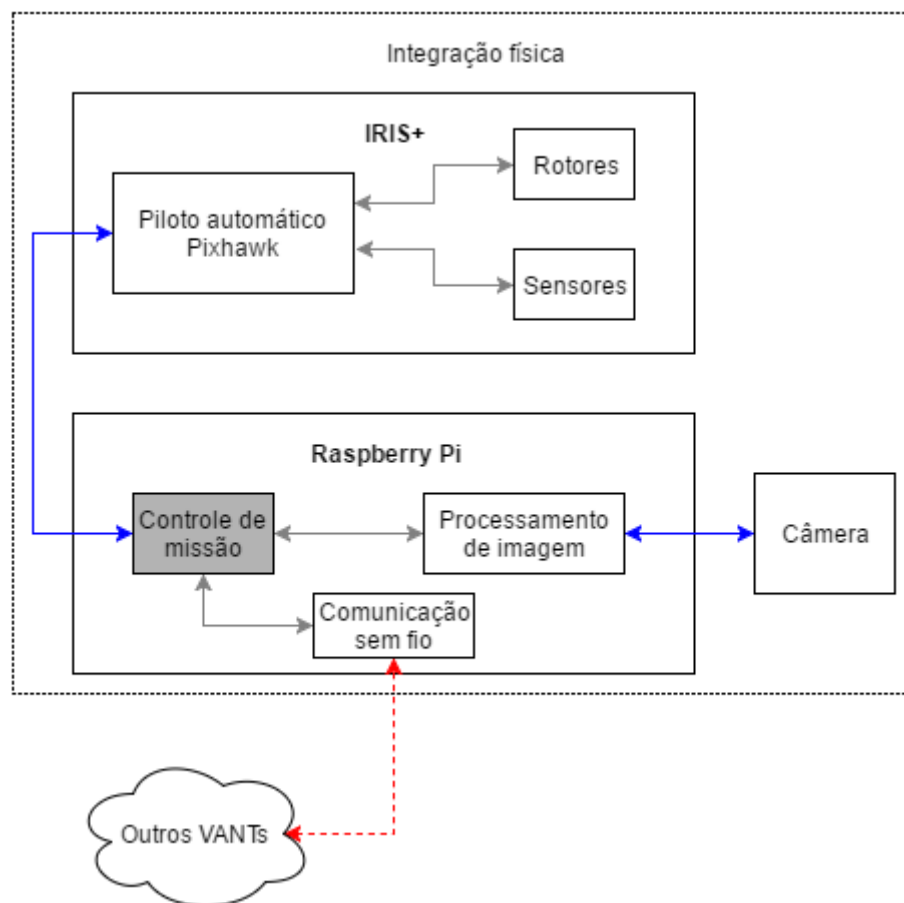
O objetivo principal deste estudo é tornar o controle de missão autônomo e otimizado de forma que o operador humano não seja sobrecarregado. Assim, as decisões de melhores rotas e deliberações sobre fatores externos serão otimizadas e gerenciadas pelo computador embarcado integrado ao veículo aéreo não-tripulado.

Para atingir soluções ótimas para o planejamento de rotas dos VANTs é necessário o uso de um algoritmo de planejamento de rotas que seja executado durante a inspeção dos pontos de interesse de uma missão e que seja capaz de levar em conta alguns fatores como a disponibilidade e qualidade específica de cada VANT. Além disso, deve ser utilizado um protocolo padrão para a comunicação entre os dispositivos que integram a arquitetura proposta.

4.2 Arquitetura de Integração

A arquitetura proposta para a resolução deste trabalho (Figura 4.1) se constitui de dois componentes imprescindíveis: VANT (IRIS+) e computador embarcado (Raspberry Pi); e outros componentes designados a trabalhos futuros: câmera e comunicação sem fio. O escopo deste trabalho se restringe a implementação de parte do controle de missão (item hachurado na Figura 4.1) e o uso de uma interface de comunicação serial para estabelecer a comunicação entre o módulo de controle de missão e o piloto automático da aeronave.

Figura 4.1 – Arquitetura de integração de sistemas



Fonte: Elaborada pelo autor

A interface de comunicação entre os módulos de controle de missão do Raspberry Pi e o piloto automático Pixhawk utiliza o protocolo MAVLINK sobre serial. O controle de missão utiliza as informações de rotas geradas a partir de um algoritmo de planejamento de rotas (BEHNCK, 2015) e as traduz em *waypoints* enviados ao piloto automático.

A sequência ideal de operações para a execução da missão é apresentada a seguir:

- I. Um voo de varredura, geralmente feito por VANTs de asa fixa devido à sua velocidade e raio de operação, identifica os pontos de interesse de uma região.

- II. O algoritmo de planejamento de rotas (BEHNCK, 2015), rodando no Raspberry Pi, utiliza todos os pontos de interesse identificados de acordo com suas características e calcula as rotas de menor custo.
- III. O controle de missão, rodando no Raspberry Pi, encapsula as rotas calculadas utilizando o protocolo MAVLINK e envia os pacotes por comunicação serial ao piloto automático Pixhawk.
- IV. O controle de missão envia comandos MAVLINK para armar os motores e executar a decolagem.
- V. O controle de missão gerencia o status de voo e interfere, se necessário, enviando comando para mudança de rotas ou pouso.
- VI. Após todos pontos de interesse terem sido visitados, o VANT volta ao ponto de decolagem e pouso, o controle de missão então envia comandos MAVLINK para desarmar os motores e terminar a missão.

4.3 Especificação da Aeronave

O veículo aéreo utilizado neste trabalho é o 3DR IRIS+ (3DR ROBOTICS, 2015) ilustrado na Figura 4.2. Esta plataforma é um VANT pronto para voar, que consiste de um “quadróptero” (um helicóptero impulsionado por quatro rotores), bateria e controlador via rádio. Além disso possui planejador de voo, sistema de retorno à base e pouso automático. As suas especificações técnicas estão descritas na Tabela 4.1

Figura 4.2 – Plataforma de voo IRIS+



Fonte: 3DR Robotics (2015)

Tabela 4.1 – Especificações técnicas da plataforma IRIS+

	<i>Descrição</i>
Piloto automático	Pixhawk v2.4.5
Firmware	ArduCopter 3.2
GPS	uBlocks GPS
Frequência	915MHz/433MHz
Motores	950 kV
Bateria	3S 5.1 Ah 8C lítio
Tensão bateria máxima	12.6 V
Tensão bateria mínima	10.5 V
Peso com bateria	1282 g
Altura	100 mm
Largura	500 mm
Comprimento	500 mm
Capacidade de carga	400 g
Campo de operação	1 km
Tempo de voo	16-22 minutos
Velocidade máxima	64 km/h
Preço	599 dólares americanos

Fonte: 3DR Robotics (2015)

Este VANT também contém mecanismos “failsafe”. Em caso de perda do sinal de rádio: se o contato com o controlador é perdido durante o voo, IRIS retornará ao ponto de decolagem e pousará exibindo um LED piscante amarelo. Em caso de perda de sinal do GPS: se o sinal de GPS é perdido durante o voo, IRIS mudará automaticamente seu controle para o modo manual indicando com LEDs piscantes azul e amarelo. Em caso de pouca bateria: quando a bateria atinge 25% de sua capacidade, IRIS pousará exibindo LED piscante amarelo com um padrão rápido de repetição. E em caso este limite de bateria seja atingido durante uma missão, IRIS retornará ao de decolagem antes de pousar.

4.4 Especificação do Computador Embarcado

Neste estudo foi proposto o uso do Raspberry Pi (Figura 4.3) como plataforma de controle de missão, isto é, o computador embarcado responsável pelo gerenciamento da missão em nível mais alto àquele do piloto automático do VANT. Além disso, Raspberry Pi pode ser responsável por executar algoritmos de processamento de imagens, planejamento de rotas e gerenciamento de outros sensores externos.

Figura 4.3 – Raspberry Pi



Fonte: Raspberry Pi Foundation (2016)

O Raspberry Pi (ELEMENT14, 2014) é um computador de baixo custo, de tamanho pequeno e que é capaz de executar funções complexas como um computador de mesa. Devido ao seu tamanho extremamente reduzido, é muito utilizado em projetos que necessitam alto processamento, portabilidade física e um baixo custo energético. Na Tabela 4.2 é feita uma análise comparativa das características desta plataforma com outros sete computadores embarcados disponíveis no mercado atualmente.

Tabela 4.2 – Comparativo de computadores embarcados

	<i>Raspberry Pi</i>	<i>Beagle Board</i>	<i>Banana Pi</i>	<i>Panda Board</i>	<i>pcDuino3B</i>	<i>Intel NUC</i>	<i>Cotton Candy</i>	<i>Odroid-XU4</i>
Processador	ARM Cortex-A7 quad-core 900MHz	ARM Cortex-A8 720MHz	ARM Cortex-A7 dual-core 1GHz	ARM Cortex-A9 dual-core 1.2GHz	ARM Cortex A-7 dual-core 1GHz, Allwinner A20	Intel Core i5-4250U	Samsung Exynos 4210 dual-core 1.2GHz	Samsung Exynos 5422 octa-core (4xA15, 4xA7) 2GHz
Memória RAM	1 GB	256 MB	1 GB	1 GB	1 GB	Até 32 GB	1 GB	2 GB
Portas USB	4	1	4	3	2	6	2	3
Porta HDMI	Sim	Não	Sim	Sim	Sim	Sim	Sim	Sim
Porta Ethernet	Sim	Não (RS-232)	Sim	Sim	Sim	Sim	Não	Sim
Leitor de cartão de memória	Sim	Sim	Sim	Sim	Sim	Não	Sim	Sim
Placa gráfica	GPU VideoCore IV 3D	PowerVR SGX 2D/3D	GPU Mali400MP2	PowerVR SGX540	Mali400 Dual Core	Intel HD 5000	Mali400	MaliT628MP 6
Sistemas operacionais suportados	Todas distribuições ARM GNU/Linux, Windows 10	Android, distribuições Linux, VxWorks, FreeBSD, Windows CE, QNX,	Android 4.2.2 e 4.4, distribuições Linux	Android, distribuições Linux, OpenBSD, FreeBSD, RISC OS, QNX	Lbuntu, Android	Windows, distribuições Linux, Mac OS X	Android, Ubuntu, Android 4.0	Android, distribuições Linux

		Symbian, RISC OS						
Preço médio	35 USD	125 USD	45 USD	180 USD	59 USD	400 USD	200 USD	74 USD
Pinos	40 GPIO	28 GPIO	26 GPIO	26 GPIO	14 GPIO			42 GPIO
Peso	45g	37g	48g	81.5g	90g	476g	21g	60g

Fonte: Elaborada pelo autor

O uso de um dispositivo de processamento extra ao já existente no VANT é justificado quando o seu custo benefício é positivo. Assim, esta plataforma de uso geral deve conter algumas características relevantes e imprescindíveis para aplicações em VANTs, tais como baixo custo, baixo peso, alto poder de processamento e facilidade de comunicação com outros periféricos.

A partir das opções de computadores embarcados disponíveis e descritas na Tabela 4.2 e da análise dos requisitos citados, compreende-se que a plataforma Raspberry Pi contém as principais características necessárias para justificar seu uso em um VANT. Este computador é a opção mais econômica dentre as pesquisadas, é o terceiro mais leve, sendo seu peso equivalente a 11.25% do limite de carga do VANT escolhido para este projeto (Tabela 4.1), possibilitando portanto, o uso de outros periféricos como câmera e sensores. Nas questões de capacidade de processamento e comunicação com periféricos se equipara as outras plataformas pesquisadas. E finalmente, outra vantagem do uso desta plataforma é o fato de esta ser uma plataforma de fácil utilização e ser vastamente testada e documentada com diversos projetos.

A análise das outras plataformas embarcadas demonstra que o Raspberry Pi ainda é a melhor das opções pesquisadas. Os pontos mais críticos para a escolha de uma plataforma embarcada para aplicações em VANTs, são o peso, o custo, a capacidade de processamento e a capacidade de uso de outros componentes acoplados. O primeiro é um fator que influencia a capacidade de carga do VANT e a sua autonomia de voo. O segundo delimita a viabilidade da aplicação e a capacidade de uso deste tipo de aplicação em maior escala. O terceiro representa a capacidade de otimização das soluções para cada caso específico. E o quarto demonstra o quanto é possível utilizar outros componentes como câmeras e sensores junto à plataforma embarcada. A seguir são explicitados os principais fatores negativos dos demais computadores embarcados, em comparação direta com o escolhido para a implementação deste trabalho.

O Intel NUC não é um dispositivo compatível com aplicações em VANTs, visto o seu alto custo (400 dólares americanos) e o seu peso excessivo (476g), fatos que tornam o seu uso

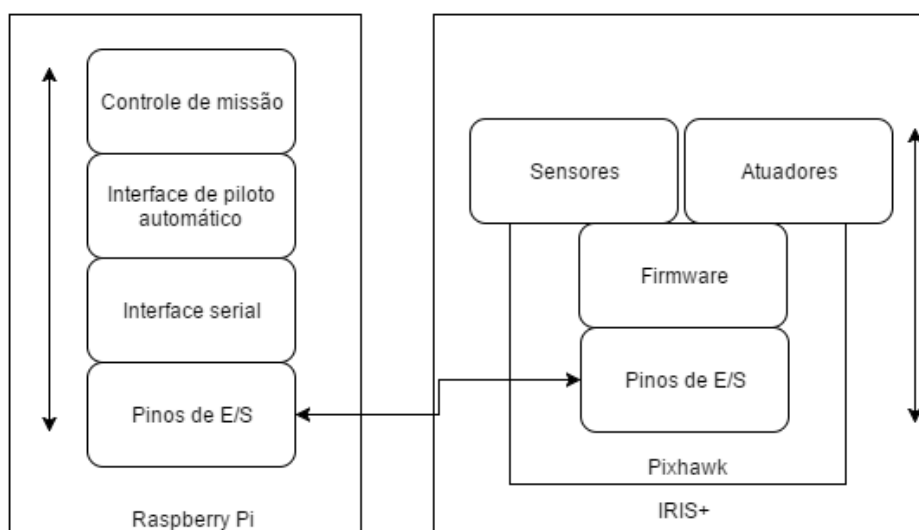
inviável para este tipo de solução. O Cotton Candy também é uma plataforma de alto custo (200 dólares americanos) e sua principal desvantagem é o fato de ainda ser um computador em teste, além de não disponibilizar pinos de expansão para o uso de outros componentes que não utilizem comunicação USB. O PandaBoard é outra opção de alto custo (180 dólares americanos) e é cerca de 81% mais pesado que o Raspberry Pi.

O BeagleBoard se apresenta como outra opção de alto custo (125 dólares americanos) e oferece apenas 25% da memória RAM disponível na plataforma utilizada neste trabalho. O Odroid-XU4, possui uma maior capacidade de processamento que os demais, visto a sua arquitetura octa-core de processadores, porém custa 110% a mais que o Raspberry Pi e pesa 33% mais que esse. O PcDuino custa 68% e tem o dobro do peso do Raspberry Pi. E finalmente, o Banana Pi é a opção mais próxima da plataforma escolhida para este trabalho, visto que é baseada no próprio Raspberry Pi. Contudo, o Banana Pi custa 28% mais e pesa 3 gramas a mais que a sua plataforma base.

4.5 Comunicação

A comunicação entre o computador embarcado e o piloto automático do VANT ocorre sobre um enlace serial e a nível da camada de aplicação se utiliza como padrão o protocolo MAVLINK (MEIER, 2009). A Figura 4.4 ilustra a interação entre o controle de missão, as interfaces de comunicação e o piloto automático .

Figura 4.4 – Diagrama de comunicação entre Raspberry Pi e Pixhawk



Fonte: Elaborada pelo autor

4.5.1 Protocolo MAVLINK

O protocolo de comunicação MAVLINK é utilizado sobre os canais seriais do piloto automático Pixhawk e se baseia em mensagens padrão encapsuladas em pacotes e enviadas através dos canais disponíveis. Este protocolo é um padrão de código aberto utilizado em vários projetos de veículos aéreos (de asa fixa e de asas rotativas) e terrestres.

A biblioteca MAVLINK é distribuída no formato de arquivos de cabeçalho na linguagem C e é amplamente utilizada com aplicações compiladas nesta linguagem e em aplicações de mais alto nível programadas em Python.

4.5.1.1 Formato do Pacote MAVLINK

A biblioteca MAVLINK na versão 1.0 possui um formato de mensagens encapsuladas em pacotes que é descrito na Tabela 4.3.

Tabela 4.3 – Formato do pacote MAVLINK

<i>Campo</i>	<i>Índice (bytes)</i>	<i>Propósito</i>
Start-of-frame	0	Denota o início da transmissão de quadro (0xFE)
Payload-length	1	Tamanho da carga do pacote (n)
Packet Sequence	2	Cada componente conta sua sequência de envio. Permite detectar perda de pacotes
System ID	3	Identificação do sistema transmissor. Permite diferenciar sistemas na mesma rede
Component ID	4	Identificação do componente transmissor. Permite diferenciar componentes de um mesmo sistema
Message ID	5	Identificação da mensagem – o ID define o que a carga representa e como deve ser corretamente decodificada
Payload	6 a (n+5)	Os dados da mensagem
CRC	(n+6) a (n+7)	Verificador CRC da mensagem

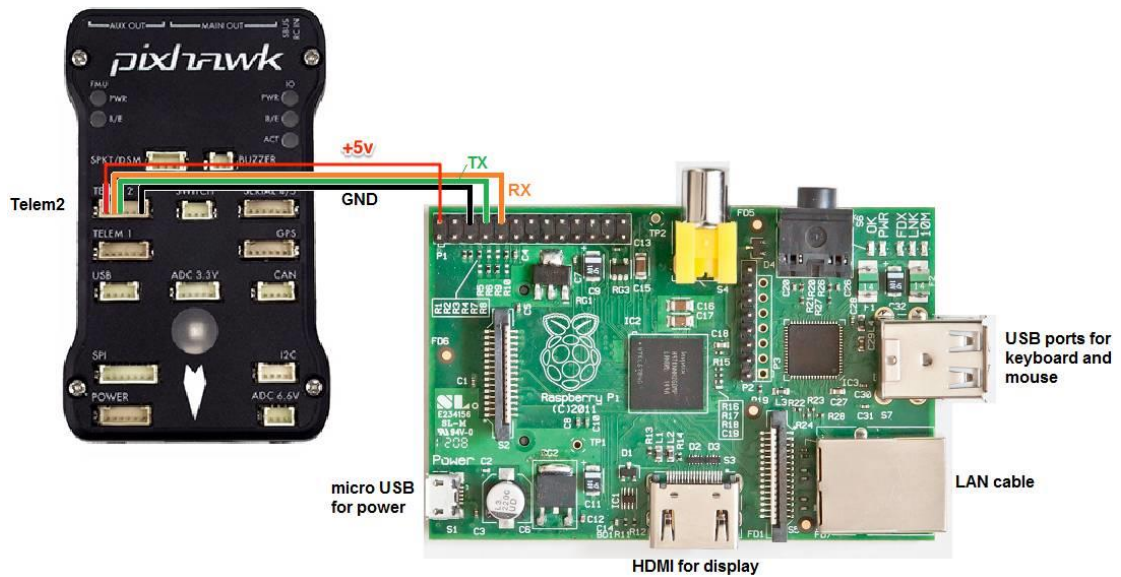
Fonte: QGroundControl.

4.5.2 Conexão Física

A comunicação entre o piloto automático Pixhawk e o Raspberry Pi a nível físico ocorre pela conexão de cabos entre os pinos de entrada e saída de cada dispositivo. A Figura 4.5 ilustra a ordem de conexão dos pinos em cada um dos dispositivos. As portas seriais do Pixhawk possuem o padrão DF13 de 6 vias, enquanto nos pinos do Raspberry Pi pode ser

utilizado um cabo do padrão FTDI. Assim, foi necessário uma adaptação para o cabeamento entre os dois padrões mencionados.

Figura 4.5 – Conexão física entre Pixhawk e Raspberry Pi



Fonte: Ardupilot.org (2016)

A conexão demonstrada na Figura 4.5 utiliza a porta Telem2 (telemetria) do piloto automático Pixhawk que utiliza um protocolo serial comum com pinos de +5V, TX, RX e GND que são conectados no Raspberry Pi nos pinos 2, 6, 8 e 10, respectivamente. A utilização dos pinos de entrada e saída do Raspberry Pi ainda podem ser utilizados para outros sensores e para comunicação com outras plataformas. A Figura 4.6 demonstra a utilização dos respectivos pinos no Raspberry Pi.

Figura 4.6 – Função dos pinos de entrada e saída no Raspberry Pi

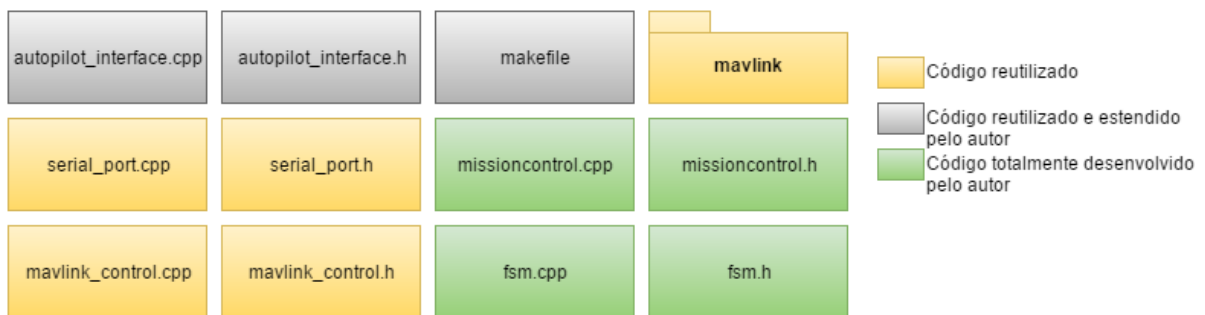
	Pin 1	Pin 2	
			+5V
			+5V
			GND
			TXD0 / GPIO 14
			RXD0 / GPIO 15
			GPIO 18
			GND
			GPIO 23
			GPIO 24
			GND
			GPIO 25
			CE0# / GPIO8
			CE1# / GPIO7
	Pin 25	Pin 26	

Fonte: Elinux.org (2015)

4.6 Implementação do Controle de Missão

O projeto da plataforma de controle de missão utilizou como base a biblioteca de mensagens MAVLINK, bem como sua interface pré-desenvolvida de comunicação serial. A partir destes elementos foi desenvolvida uma metodologia para controlar missões a partir do Raspberry Pi utilizando o conceito de máquinas de estado e concorrência (uso de *threads* com a biblioteca POSIX Threads) utilizando a linguagem C++, também foi estendida a interface de comunicação serial para o tratamento de mais tipos de mensagens MAVLINK. A organização dos arquivos de implementação é ilustrada na Figura 4.7 e a descrição dos módulos dessa implementação é apresentada nas subseções seguintes.

Figura 4.7 – Módulos da implementação do controle de missão



Fonte: Elaborada pelo autor

4.6.1 Interface de Comunicação Serial

A biblioteca MAVLINK disponibiliza uma interface de comunicação serial de código aberto e desenvolvida em C++. Essa interface é dividida em duas classes principais `Serial_Port` e `Autopilot_Interface`, em que o diagrama com seus atributos e métodos é ilustrado na Figura 4.8, além de funções comuns que não são membros de alguma classe. Todas esses elementos são declarados nos seguintes arquivos: `serial_port.h`, `autopilot_interface.h` e `mavlink_control.h`.

Figura 4.8 – Classes da interface de comunicação

Autopilot_Interface	Serial_Port
<pre> + reading_status : char + writing_status : char + control_status : char + write_count : uint64_t + system_id : int + autopilot_id : int + companion_id : int + current_messages : Mavlink_Messages + initial_position : mavlink_set_position_target_local_ned_t - serial_port : Serial_Port* - time_to_exit : boolean - read_tid : pthread_t - write_tid : pthread_t - current_setpoint : mavlink_set_position_target_local_ned_t + update_setpoint(setpoint : mavlink_set_position_target_local_ned_t) : void + read_messages() : void + write_message(message : mavlink_message_t) : int + enable_offboard_control() : void + disable_offboard_control() : void + start() : void + stop() : void + start_read_thread() : void + start_write_thread() : void + handle_quit(sig : int) : void - read_thread() : void - write_thread() : void - toggle_offboard_control(flag : boolean) : int - write_setpoint() : void </pre>	<pre> + initialize_defaults : int + debug : boolean + uart_name : char* + baudrate : int + status : int - fd : int - lastStatus : mavlink_status_t - lock : pthread_mutex_t + read_message(&message : mavlink_message_t) : int + write_message(&message : mavlink_message_t) : int + open_serial() : void + close_serial() : void + start() : void + stop() : void + handle_quit(sig : int) : void - _open_port(port : char*) : int - _setup_port(baud : int, data_bits : int, stop_bits : int, parity : boolean, hardware_control : boolean) : boolean - _read_port(&cp : uint8_t) : int - _write_port(buf : char*, len : unsigned) : void </pre>

Fonte: Elaborada pelo autor

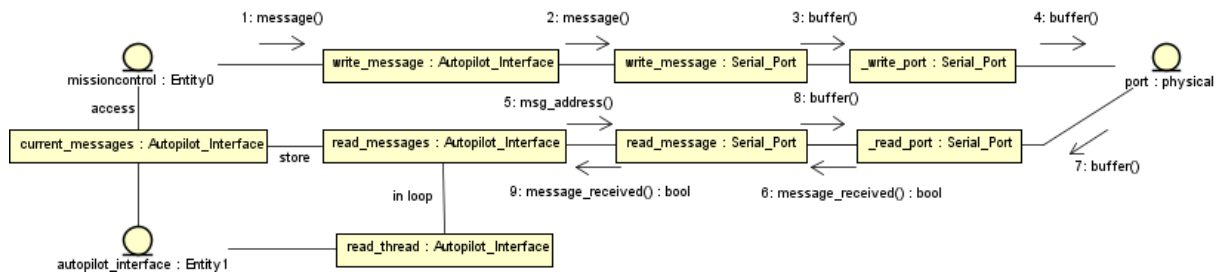
A classe `Serial_Port` é responsável por gerenciar a abertura e fechamento de portas seriais, e realiza a leitura e escrita de dados nas portas seriais. No método de leitura de dados, são lidos bytes da porta serial que está aberta e comunicando com o Pixhawk, então é feito um *parsing* dos bytes para codificá-los em uma mensagem no formato `mavlink_message_t`, que é o tipo genérico utilizado para transmissão de qualquer tipo de mensagem MAVLINK, e essa mensagem é retornada ao objeto da classe `Autopilot_Interface` que faz o tratamento posterior. Já a escrita de dados realiza o caminho contrário, transformando uma mensagem no formato `mavlink_message_t` em blocos de bytes a serem escritos na porta serial. A porta utilizada para comunicação não pode ser lida e escrita simultaneamente, para evitar possíveis erros decorrentes disso é utilizado um controle de mutex para o uso da porta.

A classe `Autopilot_Interface` é responsável pelo gerenciamento do envio e recebimento de mensagens em um nível mais alto que aquele da classe `Serial_Port`. O gerenciamento é realizado por meio de duas *threads* que ficam executando em *loop* até que seja configurado o fim da tarefa, uma que realiza *thread* gerencia os envios por meio de um método para envio de mensagens e outra os recebimentos, com um método para tratar mensagens recebidas.

O método de recebimento de mensagens identifica qual o tipo da mensagem, recebida do método de leitura de dados da classe `Serial_Port`, por sua ID e traduz a mensagem do formato `mavlink_message_t` para o formato específico da mensagem e a armazena na estrutura de dados `current_messages` (é um conjunto que contém cada tipo de mensagem

MAVLINK tratada no escopo deste trabalho) . Por sua vez, o método de envio de mensagens recebe uma mensagem do tipo *mavlink_message_t* e a transmite para o método de escrita de dados da classe *Serial_Port*. As operações de recebimento e envio de mensagens apresentadas nesta subsecção são ilustradas no diagrama de comunicação na Figura 4.9.

Figura 4.9 – Diagrama de comunicação de mensagens MAVLINK

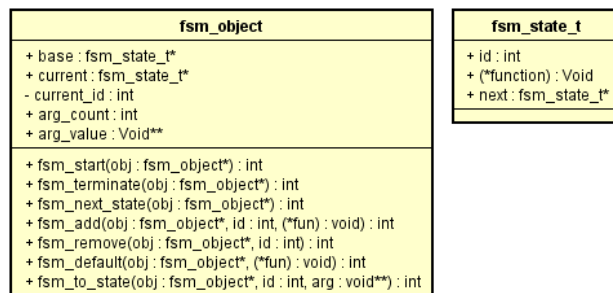


Fonte: Elaborada pelo autor

4.6.2 Estrutura de Máquina de Estados

Durante o projeto da arquitetura de implementação, foi pensada a criação de uma estrutura que permitisse a implementação do gerenciamento de missões de tal forma que fosse possível em estudos futuros estender a arquitetura existente de forma fácil e que possua alguma documentação. Desta forma, foi implementado um *framework* de criação de máquinas de estado genéricas, que também pode ser utilizado para outras aplicações, com fácil usabilidade de forma que tarefas de adição ou remoção de estados, bem como criação de transições entre estados são claras e objetivas. O diagrama da Figura 4.10 apresenta a classe e a estrutura responsáveis pelo *framework* descrito.

Figura 4.10 – Classe e *struct* da estrutura de máquina de estado



Fonte: Elaborada pelo autor

A classe *fsm_object* possui atributos utilizados para gerenciar os estados de uma máquina: um ponteiro que armazena o endereço do primeiro estado adicionado à lista de estados, um ponteiro que aponta ao estado atual sendo executado pela máquina, um identificador do estado atual, um contador de argumentos e um ponteiro com argumentos a serem passados por parâmetro entre os estados.

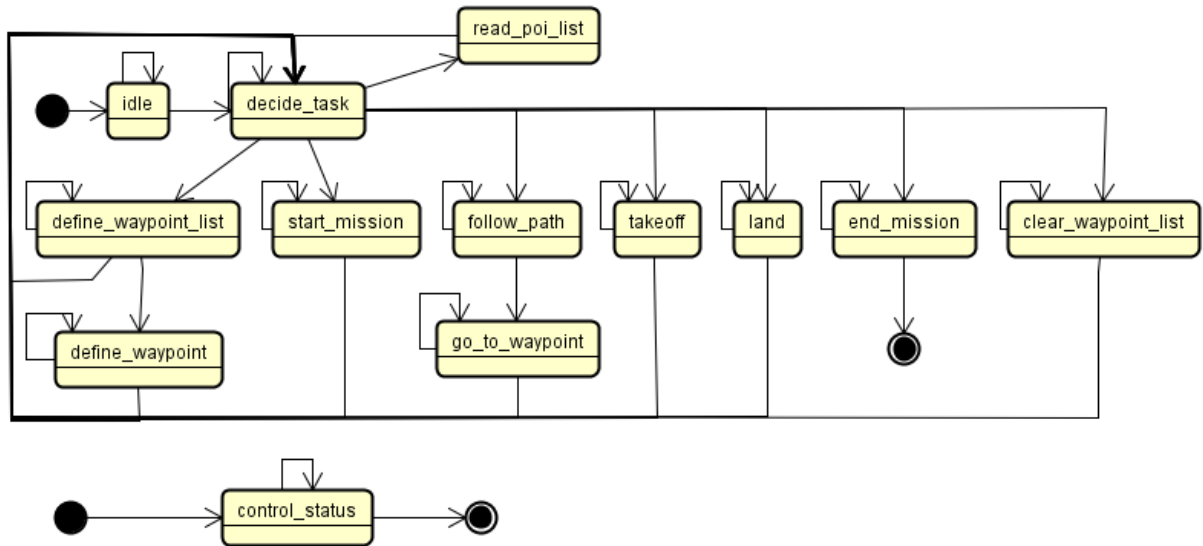
Além disso, possui métodos responsáveis por: iniciar a execução da máquina de estados (*fsm_start*), terminar a execução dessa (*fsm_terminate*), apontar o próximo estado da lista (*fsm_next_state*), adicionar novos estados (*fsm_add*), remover estados (*fsm_remove*), instanciar um estado padrão (*fsm_default*) e apontar o próximo estado a ser executado (*fsm_to_state*).

Já a estrutura do tipo *fsm_state_t* é utilizada para o instanciamento de um estado, com um identificador, sua função correspondente, que é passada por parâmetro no momento de sua criação através do método *fsm_add* da classe *fsm_object*, e o ponteiro que aponta para o próximo estado da lista.

4.6.2.1 Arquitetura da Máquina de Estados Proposta

A estrutura do controle de missão proposta (Figura 4.11) se constitui de duas máquinas de estado criadas utilizando a metodologia apresentada anteriormente, cada uma rodando a partir de uma *thread* diferente. A primeira máquina controla o fluxo da missão em si, desde sua definição até sua execução, isto é, trata a leitura dos pontos de interesse e o envio desses pontos ao piloto automático em forma de itens de missão e também é capaz de gerenciar a missão durante sua execução através do tratamento de informações de status recebidas do piloto automático. Enquanto a segunda máquina é responsável por verificar a situação da missão por meio da leitura de mensagens de status a fim de apontar o término do processamento da primeira máquina de estado no momento correto.

Figura 4.11 – Máquinas de estado do controle de missão



Fonte: Elaborada pelo autor

A Figura 4.11 ilustra os estados implementados no controle de missão proposto, esta arquitetura de máquina de estado foi baseada no caso de uso apresentado por Behnck (2014), no qual cada tarefa representa um estágio da missão e um controlador é responsável por decidir qual tarefa deve ser realizada. O controle de missão proposto gerencia as tarefas de uma missão através da comunicação entre o Raspberry Pi e o piloto automático Pixhawk por meio de mensagens MAVLINK.

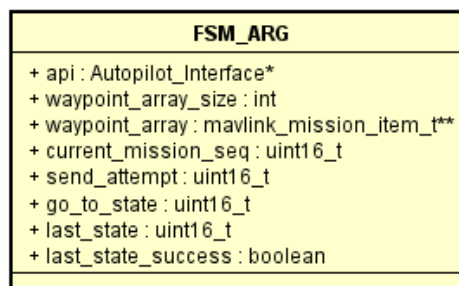
Inicialmente, no estado *idle*, algumas variáveis de controle que são utilizadas por todos os estados são inicializadas e se fica nesse estado até que o piloto automático informe que o VANT está em *standby* através de suas mensagens de status com a informação de *MAV_STATE*. Então, as coordenadas dos pontos de interesse ordenadas pelo algoritmo de planejamento de rotas (BEHNCK, 2015) são lidas no estado *read_poi_list* e traduzidas para itens de missão (*mavlink_mission_item_t*). As coordenadas dos pontos de interesse ordenados são lidas de um arquivo binário “path.poi”, o qual está estruturado como uma sequência de valores em ponto flutuante que devem ser lidos em pares (o primeiro valor é a latitude e o segundo a longitude do respectivo ponto de interesse). A tradução desses pontos de interesse é simulada da seguinte forma: a primeira mensagem de item de missão recebe o comando de decolagem com as coordenadas do primeiro ponto de interesse lido, as mensagens seguintes recebem o comando de navegação *Waypoint* (navegar até a localidade configurada) com cada respectiva coordenada geográfica lida do arquivo “path.poi” e a última mensagem de item de

missão recebe o comando de pouso e as coordenadas do último ponto de interesse existente no arquivo binário.

A definição da missão ocorre no estado *define_waypoint_list* com o envio de mensagens do tipo *mavlink_mission_item_t*, as quais podem conter comandos de navegação, comandos auxiliares que não afetam a posição do veículo ou comandos condicionais, que são executados quando as respectivas condições são encontradas. Esses comandos e seus respectivos parâmetros são armazenados pelo piloto automático em uma lista e são executados sequencialmente quando enviada uma mensagem do tipo *mavlink_command_long_t* no estado *start_mission* com o comando para iniciar a missão.

O estado *decide_task* é o operador central da máquina de estado, isto é, nesse estado que é tomada a decisão de qual próxima tarefa será executada. A próxima tarefa é escolhida a partir do valor das variáveis *last_state*, *go_to_state* e *last_state_success* da estrutura *FSM_ARG* (Figura 4.12). Essa estrutura é lida por todos estados e contém o objeto da interface do piloto automático (*api*), a lista de itens de missão com os respectivos pontos de interesse (*waypoint_array*) e algumas variáveis de controle: para número de tentativas de reenvio de mensagem (*send_attempt*), próximo estado esperado para executar (*go_to_state*), último estado executado (*last_state*) e resultado do último estado (*last_state_success*).

Figura 4.12 – Estrutura de variáveis de controle



Fonte: Elaborada pelo autor

A maioria dos comandos também podem ser enviados ao piloto automático, de forma independente daqueles atrelados a sequência de uma missão, por meio de mensagens do tipo *mavlink_command_long_t*. Através desse tipo de mensagem é possível tratar alguns problemas e falhas durante o voo, isto é, se uma determinada situação ocorrer durante a execução da missão – como baixa carga de bateria, por exemplo – uma mensagem comandando o retorno à base ou a um ponto de encontro (*rally point*) pode ser enviada ao piloto automático de forma que a execução do item atual da missão seja pausada e o VANT

possa retornar com segurança à localização designada. Assim, os estados *follow_path*, *takeoff* e *land* foram implementados para, em casos de emergência detectados, solicitar a navegação até certa localização, decolagem e pouso, respectivamente.

O fim da execução de uma missão ou um pouso de emergência podem representar o fim da necessidade do controle de missão. Dessa forma, o estado *control_status* da segunda máquina de estado lê as mensagens de status recebidas do piloto automático e quando alguma das condições citadas for atingida, esse estado armazena a informação em uma variável a fim de avisar à outra máquina de estado que sua execução pode ser terminada. E então, quando a primeira máquina de estado entrar novamente em *decide_task*, aquela variável será lida e o processamento irá ser terminado. Posteriormente, a porta serial utilizada será fechada e a comunicação entre o piloto automático e o Raspberry Pi será interrompida.

Essa máquina estado foi desenvolvida dessa forma para que também fosse fácil acrescentar novas tarefas futuramente, podendo assim realizar a integração com outros algoritmos desenvolvidos para processar outras atividades como comunicação e sistema de imagem. Cada novo módulo de processamento pode ser adicionado através da inserção de um novo estado nessa máquina de estado, tal que nesse novo estado é feita a chamada das funções do módulo adicionado.

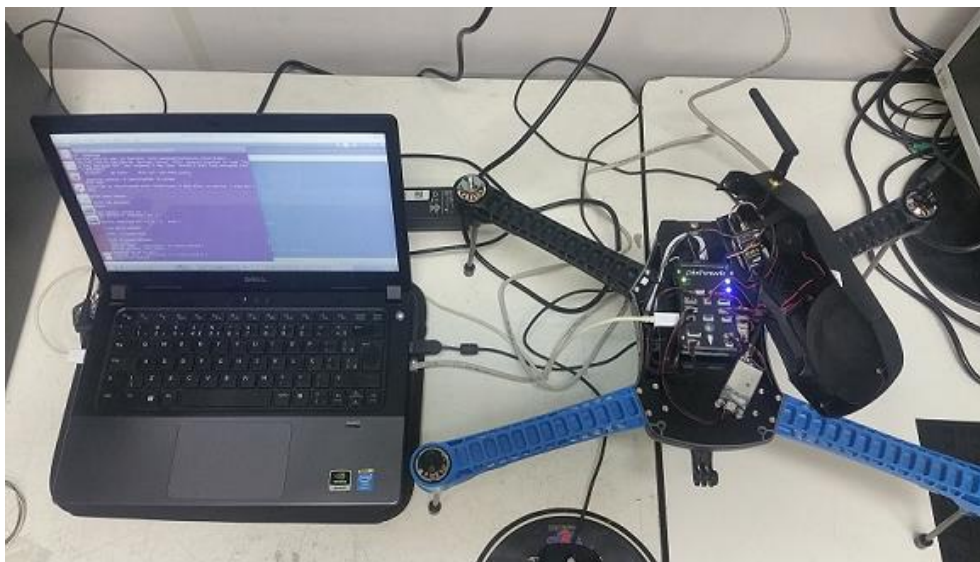
5 TESTES E RESULTADOS

Nesta seção são apresentados as estruturas montadas, a descrição dos testes realizados para verificar o funcionamento dos módulos implementados, bem como a discussão dos resultados obtidos neste estudo.

5.1 Estruturas de Teste

Os primeiros testes completos foram realizados a partir de um computador conectado ao piloto automático Pixhawk via USB (Figura 5.1). Esse ambiente de teste foi operado em uma máquina virtual Ubuntu 12.04, com processador Intel Core i7-4500U e 8 GB de memória RAM. A comunicação foi efetivada através do dispositivo *ttyACM0* identificado pelo sistema operacional do computador. E por questões de segurança, todos testes realizados em laboratório fizeram uso do VANT sem suas hélices.

Figura 5.1 – Estrutura de teste com computador e IRIS+

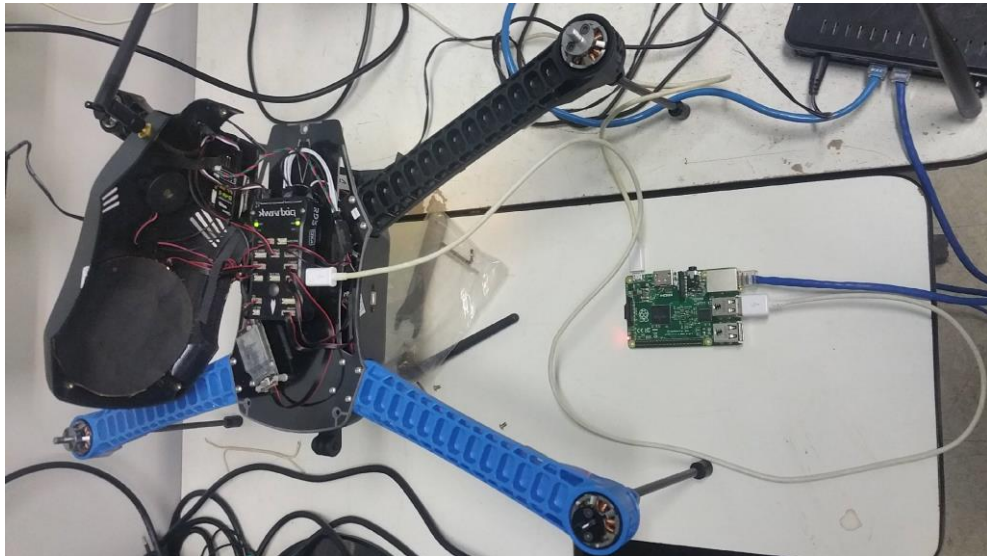


Fonte: Elaborada pelo autor

Posteriormente, os testes finais foram realizados a partir da arquitetura originalmente proposta para este trabalho, efetuando a comunicação entre a plataforma Raspberry Pi e o piloto automático Pixhawk (Figura 5.2) conforme apresentado na seção 4.5.2 desse estudo. Também foram executados testes comunicando o computador embarcado com o piloto

automático através das suas respectivas portas USB. O controle do Raspberry Pi foi feito através de um acesso por SSH (*Secure shell*), visto que o computador embarcado foi conectado à rede através de um cabo Ethernet. Esse dispositivo roda o sistema operacional Raspbian e contém o compilador g++ 4.9.2.

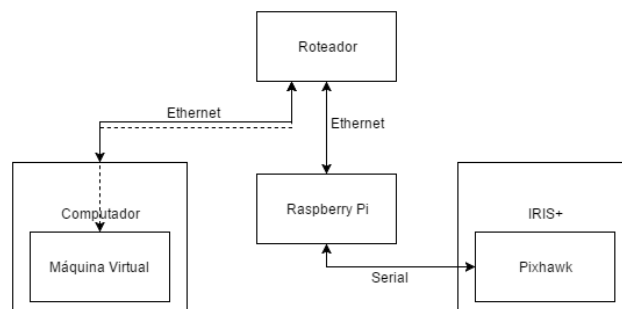
Figura 5.2 – Estrutura de teste com Raspberry Pi e IRIS+



Fonte: Elaborada pelo autor

A topologia da rede criada para realizar a comunicação entre os dispositivos utilizados nos testes é apresentada na Figura 5.3. Nessa topologia o computador e o Raspberry Pi são conectados ao roteador por meio de cabos Ethernet e são identificados por endereços IP (*Internet Protocol*). Já a comunicação entre o Raspberry Pi e o Pixhawk se dá pela conexão com um cabo serial e os respectivos dispositivos são identificados através dos seus arquivos de dispositivos criados pelo sistema. Por meio da virtualização da rede, a máquina virtual também recebe um endereço de IP pertencente a mesma rede centralizada pelo roteador.

Figura 5.3 – Topologia de rede utilizada nos testes



Fonte: Elaborada pelo autor

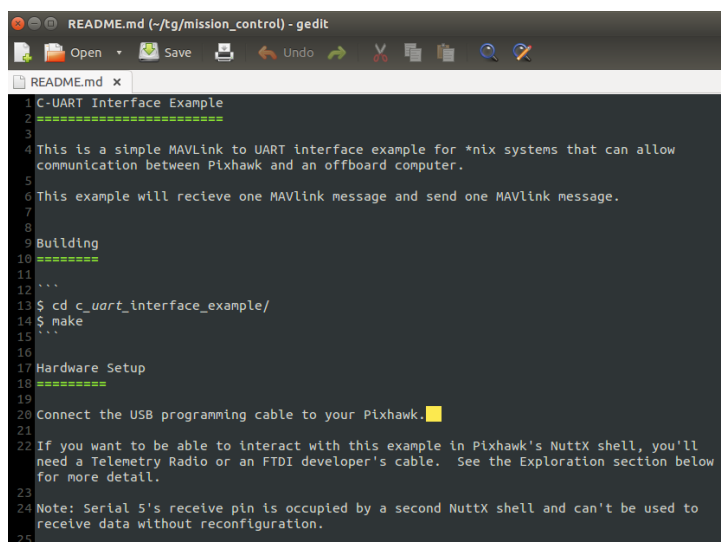
5.2 Descrição dos Testes

Nas diversas fases do desenvolvimento deste trabalho foram realizados testes para validar corretude, modularidade, usabilidade e os respectivos requisitos funcionais do sistema. Todos os módulos apresentados na Figura 4.7 foram testados, sendo inicialmente verificações em tempo de compilação e posteriormente verificações funcionais.

5.2.1 Verificação da Biblioteca MAVLINK

Como o módulo de controle de missão foi desenvolvido a partir da biblioteca MAVLINK, o primeiro teste executado foi a verificação da corretude do código das interfaces de piloto automático e portal serial disponibilizadas com a biblioteca de código aberto. Esse teste basicamente foi a compilação do código já existente utilizando o compilador g++ 4.8.4 na máquina virtual com Ubuntu 14.04, e sua execução de acordo com as instruções existentes no arquivo *README* (Figura 5.4) escritas pelos autores da biblioteca. Nesse arquivo é demonstrado a forma de compilação do exemplo, o comando para executar o arquivo binário compilado “./mavlink_control -d /dev/ttyACM0 -b 115200” (onde “-d” configura o piloto automático a ser utilizado e “-b” o *baudrate*), instruções sobre como efetuar a conexão física e a saída esperada no console.

Figura 5.4 – Instruções de uso da interface



```
README.md (-/tg/mission_control) - gedit
README.md x
1 C-UART Interface Example
2 =====
3
4 This is a simple MAVLink to UART interface example for *nix systems that can allow
5 communication between Pixhawk and an offboard computer.
6
7 This example will receive one MAVlink message and send one MAVlink message.
8
9 Building
10 =====
11 ***
12 ***
13 $ cd c_uart_interface_example/
14 $ make
15 ***
16
17 Hardware Setup
18 =====
19
20 Connect the USB programming cable to your Pixhawk.
21
22 If you want to be able to interact with this example in Pixhawk's NuttX shell, you'll
23 need a Telemetry Radio or an FTDI developer's cable. See the Exploration section below
24 for more detail.
25
26 Note: Serial 5's receive pin is occupied by a second NuttX shell and can't be used
27 to receive data without reconfiguration.
```

Fonte: Elaborada pelo autor

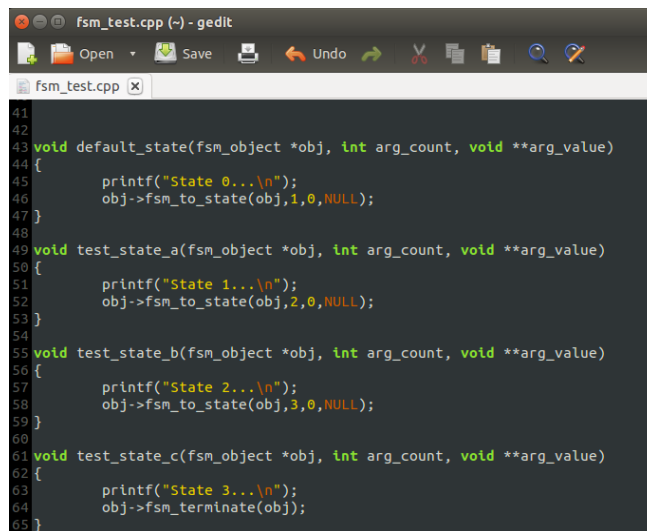
A codificação das mensagens MAVLINK, declaradas como estruturas, e suas respectivas funções, declaradas como *static inline*, são definidas em arquivos do tipo *header*. Dessa forma, foi necessário apenas referenciar esses arquivos no projeto para validar a corretude de seus códigos em tempo de compilação. O roteiro apresentado no arquivo *README* foi seguido e assim foram validadas as funcionalidades de leitura e escrita da interface de piloto automático, bem como a abertura e fechamento de portas seriais realizada pela interface de comunicação serial.

5.2.2 Verificação da Estrutura de Desenvolvimento de Máquinas de Estado

O segundo passo foi desenvolver uma estrutura que habilitasse a implementação de máquinas de estado como apresentado na seção 4.6.2 deste trabalho. Assim, antes de iniciar a implementação final da máquina de estado para o controle de missão, foi necessário verificar o comportamento esperado dessa estrutura de máquinas de estado. Essa verificação ocorreu utilizando o compilador g++ 4.8.4 na máquina virtual com Ubuntu 14.04.

O teste para essa estrutura consistiu em observar o funcionamento de todos os métodos da classe *fsm_object* (Figura 4.10) a partir da implementação de uma máquina de estado simples. Essa máquina de teste (Figura 5.5) possui quatro estados (0, 1, 2 e 3) que, quando executados, imprimem uma mensagem na tela e então transicionam para o próximo estado.

Figura 5.5 – Máquina de estado teste



```

41
42
43 void default_state(fsm_object *obj, int arg_count, void **arg_value)
44 {
45     printf("State 0...\n");
46     obj->fsm_to_state(obj,1,0,NULL);
47 }
48
49 void test_state_a(fsm_object *obj, int arg_count, void **arg_value)
50 {
51     printf("State 1...\n");
52     obj->fsm_to_state(obj,2,0,NULL);
53 }
54
55 void test_state_b(fsm_object *obj, int arg_count, void **arg_value)
56 {
57     printf("State 2...\n");
58     obj->fsm_to_state(obj,3,0,NULL);
59 }
60
61 void test_state_c(fsm_object *obj, int arg_count, void **arg_value)
62 {
63     printf("State 3...\n");
64     obj->fsm_terminate(obj);
65 }

```

Fonte: Elaborada pelo autor

Nesse teste foram observadas: a criação de cada estado através do método *fsm_add* e sua correta alocação na lista de estados; a transição de lógica de estados pelo método *fsm_to_state* que altera o atributo *current_state* do objeto pertencente à classe *fsm_object*; a transição de estados de fato, quando o método *fsm_next_state* procura o nó da lista que corresponde à identificação configurada no atributo *current_state* e executa sua respectiva função; o início da execução da máquina de estado através do *loop* criado no método *fsm_start*; e o fim da execução da máquina com o método *fsm_terminate*, o qual remove todos os nós da lista de estados e sinaliza o fim do processamento.

A criação correta de todos os estados, o funcionamento das transições de estado, bem como o início e fim de execução da máquina foram verificadas apresentando o comportamento esperado durante a execução do algoritmo. O comportamento do algoritmo foi analisado através da impressão em tela dos resultados obtidos a partir de cada método mencionado e da sequência de execução correta dos estados como eles foram implementados (iniciando no estado 0 e posteriormente transicionando para os estados 1, 2 e 3, respectivamente).

5.2.3 Verificação do Controle de Missão na Máquina Virtual

Finalmente, a partir do desenvolvimento do módulo de controle de missão foram realizados testes para validar a proposta apresentada no corpo deste trabalho. O desenvolvimento desse módulo utilizou o compilador g++ tanto na máquina virtual quanto no Raspberry Pi. Dessa forma, os testes funcionais foram realizados na máquina virtual e posteriormente no Raspberry Pi.

O principal objetivo desse teste foi verificar o comportamento dos estados criados na máquina de estado do controle de missão, a fim de validar a troca de mensagens MAVLINK e a atribuição de tarefas de acordo com o estágio da missão. A sequência de tarefas da máquina de estado verificada é ilustrada na Tabela 5.1 e equivalente para as execuções na máquina virtual e no Raspberry Pi.

Tabela 5.1 – Sequência de eventos para teste do controle de missão

Ordem	Ação
1	Estado <i>IDLE</i> : o controle aguarda o piloto automático ficar no modo <i>standby</i> e então aloca espaço na memória para registrar a lista de pontos de interesse e

	inicializa outras variáveis de controle da estrutura <i>FSM_ARG</i> (Figura 4.12).
2	Estado <i>READ Poi LIST</i> : o controle lê um arquivo contendo as coordenadas dos pontos de interesse e então as traduz em itens de missão, atribuindo comandos e parâmetros de navegação.
3	Estado <i>DEFINE WAYPOINT LIST</i> : é iniciado o protocolo de envio de itens de missão: é enviado o número de itens ao piloto automático através de uma mensagem do tipo <i>mavlink_mission_count_t</i> , o qual responde com uma mensagem do tipo <i>mavlink_mission_request_t</i> requisitando o primeiro item de missão com sequência igual a zero .
4	Estado <i>DEFINE WAYPOINT (n)</i> : para cada requisição de item de missão recebido, um item de missão com o respectivo valor de sequência é enviado ao piloto automático através de uma mensagem do tipo <i>mavlink_mission_item_t</i> ; assim que todos itens forem recebidos, o piloto automático responde com uma mensagem do tipo <i>mavlink_mission_ack_t</i> com o parâmetro <i>type</i> como <i>MAV_MISSION_ACCEPTED</i> .
5	Estado <i>START MISSION</i> : o controle envia uma mensagem do tipo <i>mavlink_command_long_t</i> com o parâmetro <i>command</i> configurado como <i>MAV_CMD_MISSION_START</i> , requisitando o início da missão .
*	Estado <i>DECIDE TASK</i> : esse estado analisa qual deve ser a próxima tarefa a ser executada levando em conta três parâmetros: o último estado executado (<i>last_state</i>), o resultado da última tarefa (<i>last_state_success</i>) e o próximo estado que é esperado para ser executado (<i>go_to_state</i>). Essas variáveis de controle são atualizadas por todos os estados executados e estão na estrutura <i>FSM_ARG</i> (Figura 4.12).

Fonte: Elaborada pelo autor

A execução da aplicação apresenta os diversos módulos desenvolvidos sendo utilizados. A sequência é iniciada pelo módulo “mavlink_control” que realiza um *parse* do argumentos do comando no terminal, em seguida é inicializado o objeto da classe *Autopilot_Interface* e o objeto da classe *Serial_Port*. Então, através do objeto da interface de piloto automático é iniciada a comunicação com o Pixhawk e na função *commands* é chamada a função *start_mission_control* que executa a máquina de estado. Após o fim do processamento da máquina de estado, o algoritmo retorna a função *commands* da interface de piloto automático. Nesse ponto são realizadas algumas leituras de posição do VANT e de dados dos seus sensores.

A Figura 5.6 ilustra a primeira parte do teste na máquina virtual, nessa figura são mostrados a compilação utilizando o compilador g++ e o início da execução do módulo de controle de missão a partir do comando “./mavlink_control -d /dev/ttyACM1 -b 115200”. Esse comando inicializa o programa e configura o dispositivo do piloto automático como sendo *ttyACM1* e também configura a *baudrate* com velocidade de 115200 bits por segundo.

Figura 5.6 – Primeira parte da saída no console da máquina virtual

```

romulo@romulo-VirtualBox: ~/tg/mission_control
rm -rf *o mavlink_control missioncontrol
g++ -I mavlink/include/mavlink/v1.0 fsm.cpp messages_test.cpp missioncontrol.cpp
mavlink_control.cpp serial_port.cpp autopilot_interface.cpp -o mavlink_control
-lpthread
mavlink_control.cpp: In function 'void commands(Autopilot_Interface&)':
mavlink_control.cpp:259:50: warning: format '%llu' expects argument of type 'long
long unsigned int', but argument 2 has type 'uint64_t {aka long unsigned int}'
[-Wformat=]
    printf("    ap time:    %llu \n", imu.time_usec);
                                   ^
./mavlink_control -d /dev/ttyACM1 -b 115200
OPEN PORT
Connected to /dev/ttyACM1 with 115200 baud, 8 data bits, no parity, 1 stop bit (
8N1)

START READ THREAD

CHECK FOR MESSAGES
Found

GOT VEHICLE SYSTEM ID: 1
GOT AUTOPILOT COMPONENT ID: 1

INITIAL POSITION XYZ = [ 0 , 0 , 1400 ]

START WRITE THREAD

ENABLE OFFBOARD MODE

SEND OFFBOARD COMMANDS
IDLE state...
>DECIDE TASK... last_state = 0, state_success 1
DEFINE WAYPOINT LIST state...
Mission count message sent
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0

```

Fonte: *print screen* da tela na máquina virtual

Posteriormente, é escrita uma mensagem informando que foi possível se conectar àquele dispositivo configurado. Então, são iniciadas as *threads* de leitura – que identifica o veículo aéreo e o piloto automático e periodicamente lê mensagens MAVLINK enviadas do Pixhawk, e de escrita – que habilita o envio de comandos externos ao piloto automático. Depois disso, as máquinas de estado são inicializadas e executadas seguindo a sequência de eventos proposta na Tabela 5.1.

A Figura 5.7 apresenta a continuação do envio dos itens de missão seguindo o protocolo padrão de definição de missão: inicialmente se envia uma mensagem do tipo *mavlink_mission_count_t* com o número de itens a serem enviados ao piloto automático, o qual responde com uma mensagem do tipo *mavlink_mission_request_t* com o parâmetro *seq* (sequência) igual a zero. A cada respectivo item de missão enviado com sucesso, o piloto automático responde com uma mensagem do tipo *mavlink_mission_request_t* incrementando o parâmetro *seq* até que o último item esperado tenha sido enviado. Finalmente, o piloto automático responde com uma mensagem do tipo *mavlink_mission_ack_t* (Figura 5.8) com o parâmetro *type* configurado como *MAV_MISSION_ACCEPTED* (enumerado como zero).

Figura 5.7 – Segunda parte da saída no console da máquina virtual

```

romulo@romulo-VirtualBox: ~/tg/mission_control
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
sending mission item 0
command 22, lat 0.00000, lon 0.00000
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 1
sending mission item 1
command 16, lat -30.06898, lon -51.12179
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 1
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 2
sending mission item 2
command 16, lat -30.06990, lon -51.12096
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 2
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 3
sending mission item 3
command 16, lat -30.06990, lon -51.12096
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 3
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
sending mission item 4
command 16, lat -30.07020, lon -51.12475
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5
sending mission item 5
command 16, lat -30.06775, lon -51.12594
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 6
sending mission item 6
command 16, lat -30.06775, lon -51.12594
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 6
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
sending mission item 7
command 16, lat -30.06775, lon -51.12594
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 8
sending mission item 8

```

Fonte: *print screen* da tela na máquina virtual

A missão é iniciada a partir do envio de uma mensagem do tipo *mavlink_command_long_t* com o comando *MAV_CMD_MISSION_START*, com parâmetros de primeiro e último itens de missão a serem executados (Figura 5.8). O fim da missão ocorre quando o último item da missão é atingido e geralmente esse item contém um comando de pouso. Após o fim da execução das máquinas de estado, a interface de piloto automático realiza algumas leituras de posição atual do VANT e de alguns de seus sensores.

Como o VANT testado estava em repouso, a Figura 5.8 ilustra que foram lidos valores zerados de aceleração e velocidade angular. Os valores lidos de mensagens de *status* também dependem dos sensores que estão, de fato, conectados ao piloto automático. No teste também são lidos mensagens de magnetometria, altimetria e barimetria.

Os sensores presentes no VANT são detectados pelo controle de missão através de mensagens do tipo *mavlink_sys_status_t* recebidas do piloto automático. Esse tipo de mensagem contém um parâmetro que é uma máscara de bits em que cada bit representa um sensor, se esse sensor estiver presente no VANT o bit correspondente terá o valor um senão terá o valor zero.

Figura 5.8 – Terceira parte da saída no console da máquina virtual

```

romulo@romulo-VirtualBox: ~/tg/mission_control
sending mission item 8
command 16, lat -30.06610, lon -51.12431
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
sending mission item 9
command 21, lat -30.06711, lon -51.12207
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
Reply: MAVLINK_MSG_ID_MISSION_ACK type 0
Mission items accepted!
>DECIDE TASK... last_state = 3, state_success 1
START MISSION state...
start mission message sent
>DECIDE TASK... last_state = 11, state_success 1
End Mission
POSITION SETPOINT XYZ = [ -5.0000 , -5.0000 , 0.0000 ]
POSITION SETPOINT YAW = 0.0000
0 CURRENT POSITION XYZ = [ 0 , 0 , 960 ]
1 CURRENT POSITION XYZ = [ 0 , 0 , 960 ]
2 CURRENT POSITION XYZ = [ 0 , 0 , 960 ]
3 CURRENT POSITION XYZ = [ 0 , 0 , 930 ]
4 CURRENT POSITION XYZ = [ 0 , 0 , 940 ]
5 CURRENT POSITION XYZ = [ 0 , 0 , 940 ]
6 CURRENT POSITION XYZ = [ 0 , 0 , 930 ]
7 CURRENT POSITION XYZ = [ 0 , 0 , 930 ]

DISABLE OFFBOARD MODE

READ SOME MESSAGES
Got message LOCAL_POSITION_NED (spec: https://pixhawk.ethz.ch/mavlink/#LOCAL_POSITION_NED)
  pos (NED):  0.000000 0.000000 0.000000 (m)
Got message HIGHRES_IMU (spec: https://pixhawk.ethz.ch/mavlink/#HIGHRES_IMU)
  ap time:    47288453611576
  acc (NED):  0.000000 0.000000 0.000000 (m/s^2)
  gyro (NED): 0.000000 0.000000 0.000000 (rad/s)
  mag (NED):  0.000000 0.000000 0.000000 (Ga)
  baro:       0.000000 (mBar)
  altitude:   0.000000 (m)

```

Fonte: *print screen* da tela na máquina virtual

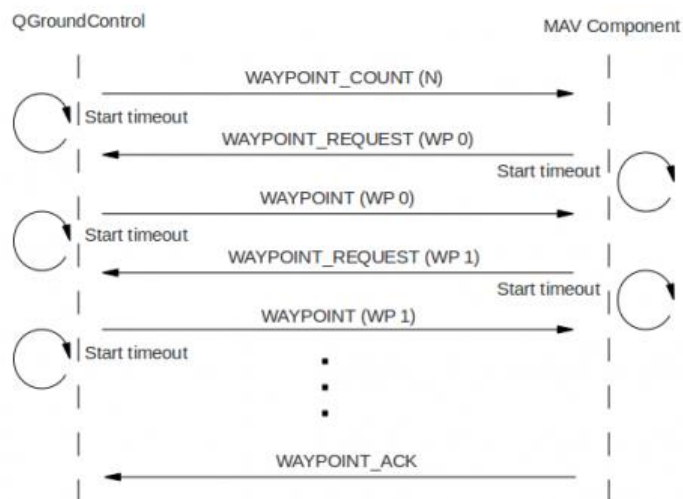
5.2.4 Verificação do Controle de Missão no Raspberry Pi

A última fase de testes ocorreu após o controle de missão ter suas funcionalidades validadas a partir da comunicação entre a máquina virtual e o piloto automático Pixhawk como apresentado na seção anterior. Nessa fase foram realizados os ajustes de portabilidade necessários para a execução bem sucedida do algoritmo a partir do Raspberry Pi. As pequenas modificações ocorreram apenas devido a uma inconsistência na alocação de memória ao se utilizar o mesmo código fonte testado anteriormente na outra plataforma. O código foi recompilado no Raspberry Pi sem o uso de um *cross-compiler* já que foi escolhido apenas esse computador embarcado como plataforma alvo e como sua memória é do tipo *flash* não existe uma limitação em capacidade de memória que impossibilitasse a instalação de um compilador.

O Raspberry Pi utilizado nos testes opera com o sistema operacional Raspbian e não contém um *kernel* modificado para atender requisitos de tempo real. Algumas garantias de temporização são encontradas através de funções de espera utilizadas para que a troca de mensagens entre as plataformas ocorresse na sequência esperada. Uma das melhorias que devem ser desenvolvidas no futuro deste projeto está no uso de um *kernel* de tempo real que é necessário para atender os requisitos de aplicações para veículos aéreos.

Durante a execução do protocolo de envio de mensagens *mavlink_mission_item_t*, baseado no exemplo da Figura 5.9, a temporização é feita através da função *usleep()*. Nos estados *define_waypoint_list* e *define_waypoint* são realizados os envios de mensagens do tipo *mavlink_mission_count_t* e *mavlink_mission_item_t* respectivamente, e as respostas do piloto automático são com mensagens do tipo *mavlink_mission_request_t* contendo a sequência do item de missão esperado. A Tabela 5.2 apresenta os tempos utilizados para esperar mensagens de resposta e para tentativas de reenvio de mensagens. Esses tempos foram configurados depois da análise de diversas execuções com variadas configurações de temporização. O principal objetivo da temporização entre envio de mensagens é não fazer sobreposição de mensagens, dando tempo suficiente para tanto o piloto automático quanto o módulo de controle de missão fazerem seus respectivos processamentos.

Figura 5.9 – Protocolo de envio de *waypoints*



Fonte: QGroundControl

Tabela 5.2 – Exemplo de temporização nos estados da máquina

		Tempo de espera (µs)	Comentário
Estado	<i>define_waypoint_list</i>		
Caso 1	Erro no envio de mensagem	1000	Tenta reenviar 5 vezes, caso não consiga retorna ao estado <i>decide_task</i> com parâmetro <i>last_state_success = false</i>
Caso 2	Enquanto o piloto automático não responde com mensagem <i>mavlink_mission_request_t</i> com sequência igual a 0	10000	Execução no trecho: <pre>while(messages.mission_request.seq != 0 && messages.time_stamps.mission_request < send_time) { usleep(10000); messages = api.current_messages; }</pre>
Estado	<i>define_waypoint</i>		
Caso 1	Espera para testar se o item pedido pelo piloto automático é o mesmo esperado pelo controle de missão	300000	Espera para executar o trecho: <pre>if(api.current_messages.mission_request.seq == fsm_arg->current_mission_seq) { ... }</pre>
Caso 2	Espera para receber a sequência correta do piloto automático (caso o controle tenha processado mais rápido que a resposta do piloto)	200000	Execução no trecho: <pre>if(api.current_messages.mission_request.seq < fsm_arg->current_mission_seq && fsm_arg->current_mission_seq < fsm_arg-> waypoint_array_size && fsm_arg->send_attempt < 5) { usleep(200000); fsm_arg->send_attempt++; obj->fsm_to_state(obj, FSM_STATE_DEFINE_WAYPOINT_ID, arg_count, arg_value); return; }</pre>

Fonte: Elaborada pelo autor

A sequência de eventos na máquina de estado testada é a mesma daquela utilizada na máquina virtual e descrita na Tabela 5.1. Dessa forma, o comportamento esperado da execução do módulo de controle de missão é o mesmo daquele obtido com os testes na máquina virtual. A Figura 5.10 ilustra a primeira parte do teste, na qual é realizado o acesso ao Raspberry Pi via SSH e a compilação do projeto utilizando o compilador g++ 4.9.2. A topologia da rede em que todos os dispositivos estão é aquela apresentada na Figura 5.3.

Figura 5.10 – Acesso e compilação no Raspberry Pi

```

pi@raspberrypi:~/tg/mission_control
romulo@romulo-VirtualBox:~$ ssh -X pi@192.168.0.104
pi@192.168.0.104's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Jun  2 18:19:11 2016 from 192.168.0.101
pi@raspberrypi:~$ cd tg/mission_control/
pi@raspberrypi:~/tg/mission_control $ make clean && make && make run
rm -rf *o mavlink_control missioncontrol
g++ -I mavlink/include/mavlink/v1.0 fsm.cpp messages_test.cpp missioncontrol.cpp mavlink_control.cpp serial_port.cpp autopilot_interface.cpp
-o mavlink_control -lpthread

```

Fonte: *print screen* da tela na máquina virtual

As Figuras 5.11, 5.12 e 5.13 ilustram a execução completa do módulo de controle de missão. Na sequência de imagens é possível observar o comportamento semelhante àquele dos testes na máquina virtual, de forma que as funcionalidades implementadas foram novamente validadas no computador embarcado. Como o mesmo roteiro de eventos da máquina virtual foi utilizado no teste com o Raspberry Pi, é possível estabelecer que o algoritmo também é funcional no computador embarcado, haja vista os resultados semelhantes observados nos testes na máquina virtual e no computador embarcado.

As tarefas de leitura e escrita de mensagens MAVLINK foram executadas com sucesso, da mesma maneira foram obtidos resultados bem sucedidos nas tarefas de controle de itens de missão e partida e finalização de missão. Outras mensagens do tipo *mavlink_command_long_t* também foram enviadas do Raspberry Pi ao Pixhawk e se pode observar a resposta com mensagens do tipo *mavlink_command_ack_t* representando o resultado do tratamento dos comandos enviados.

Figura 5.11 – Primeira parte da saída no console do Raspberry Pi

```

pi@raspberrypi:~/mission_control
pi@raspberrypi:~/mission_control $ make clean && make && make run
rm -rf *o mavlink_control missioncontrol
g++ -I mavlink/include/mavlink/v1.0 fsm.cpp messages_test.cpp missioncontrol.cpp mavlink_control.cpp serial_port.cpp autopilot_interface.cpp
-o mavlink_control -lpthread
./mavlink_control -d /dev/ttyACM0 -b 115200
OPEN PORT
Connected to /dev/ttyACM0 with 115200 baud, 8 data bits, no parity, 1 stop bit (8N1)

START READ THREAD

CHECK FOR MESSAGES
Found

GOT VEHICLE SYSTEM ID: 1
GOT AUTOPILOT COMPONENT ID: 1

INITIAL POSITION XYZ = [ 0 , 0 , 0 ]

START WRITE THREAD

ENABLE OFFBOARD MODE

SEND OFFBOARD COMMANDS
IDLE state...
>DECIDE TASK... last_state = 0, state_success 1
READ Poi LIST state...
>DECIDE TASK... last_state = 12, state_success 1
DEFINE WAYPOINT LIST state...
Mission count (10) message sent

```

Fonte: *print screen* da tela na máquina virtual

Figura 5.12 – Segunda parte da saída no console do Raspberry Pi

```

pi@raspberrypi: ~/tg/mission_control
IDLE state...
>DECIDE TASK... last_state = 0, state_success 1
READ POI LIST state...
>DECIDE TASK... last_state = 12, state_success 1
DEFINE WAYPOINT LIST state...
Mission count (10) message sent
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
sending mission item 0
command 22, lat 0.00000, lon 0.00000
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 0
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 1
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 1
sending mission item 1
command 16, lat -30.06898, lon -51.12179
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 1
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 2
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 2
sending mission item 2
command 16, lat -30.06990, lon -51.12096
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 2
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 3
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 3
sending mission item 3
command 16, lat -30.06990, lon -51.12096
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 3
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
sending mission item 4
command 16, lat -30.07020, lon -51.12475
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5

pi@raspberrypi: ~/tg/mission_control
command 16, lat -30.07020, lon -51.12475
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 4
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5
sending mission item 5
command 16, lat -30.06775, lon -51.12594
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 5
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 6
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 6
sending mission item 6
command 16, lat -30.06775, lon -51.12594
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 6
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
sending mission item 7
command 16, lat -30.06775, lon -51.12594
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
sending mission item 7
command 16, lat -30.06775, lon -51.12594
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 7
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 8
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 8
sending mission item 8
command 16, lat -30.06610, lon -51.12431
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 8
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
DEFINE WAYPOINT (n) state...
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
sending mission item 9
command 21, lat -30.06711, lon -51.12207
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
Reply: MAVLINK_MSG_ID_MISSION_ACK type 0

```

Fonte: *print screen* das telas na máquina virtual

Figura 5.13 – Terceira parte da saída no console do Raspberry Pi

```

pi@raspberrypi: ~/tg/mission_control
command 21, lat -30.06711, lon -51.12207
Reply: MAVLINK_MSG_ID_MISSION_REQUEST, sequence: 9
Reply: MAVLINK_MSG_ID_MISSION_ACK type 0
DEFINE WAYPOINT (n) state...
Mission items accepted!
>DECIDE TASK... last_state = 3, state_success 1
START MISSION state...
Start mission message sent
>DECIDE TASK... last_state = 11, state_success 1
End Mission
POSITION SETPOINT XYZ = [ -5.0000 , -5.0000 , 0.0000 ]
POSITION SETPOINT YAW = 0.0000
0 CURRENT POSITION XYZ = [ 0, 0, -760 ]
1 CURRENT POSITION XYZ = [ 0, 0, -760 ]
2 CURRENT POSITION XYZ = [ 0, 0, -760 ]
3 CURRENT POSITION XYZ = [ 0, 0, -760 ]
4 CURRENT POSITION XYZ = [ 0, 0, -760 ]
5 CURRENT POSITION XYZ = [ 0, 0, -760 ]
6 CURRENT POSITION XYZ = [ 0, 0, -760 ]
7 CURRENT POSITION XYZ = [ 0, 0, -760 ]

DISABLE OFFBOARD MODE

READ SOME MESSAGES
Got message LOCAL_POSITION_NED (spec: https://pixhawk.ethz.ch/mavlink/#LOCAL_POSITION_NED)
pos (NED): 0.000000 0.000000 0.000000 (m)
Got message HIGHRES_IMU (spec: https://pixhawk.ethz.ch/mavlink/#HIGHRES_IMU)
ap ttime: 856509034774765568
acc (NED): 0.000000 0.000000 0.000000 (m/s^2)
gyro (NED): 0.000000 0.000000 0.000000 (rad/s)
mag (NED): 0.000000 0.000000 0.000000 (Ga)
baro:
altitude: 0.000000 (m)

CLOSE THREADS

CLOSE PORT

pi@raspberrypi: ~/tg/mission_control $

```

Fonte: *print screen* da tela na máquina virtual

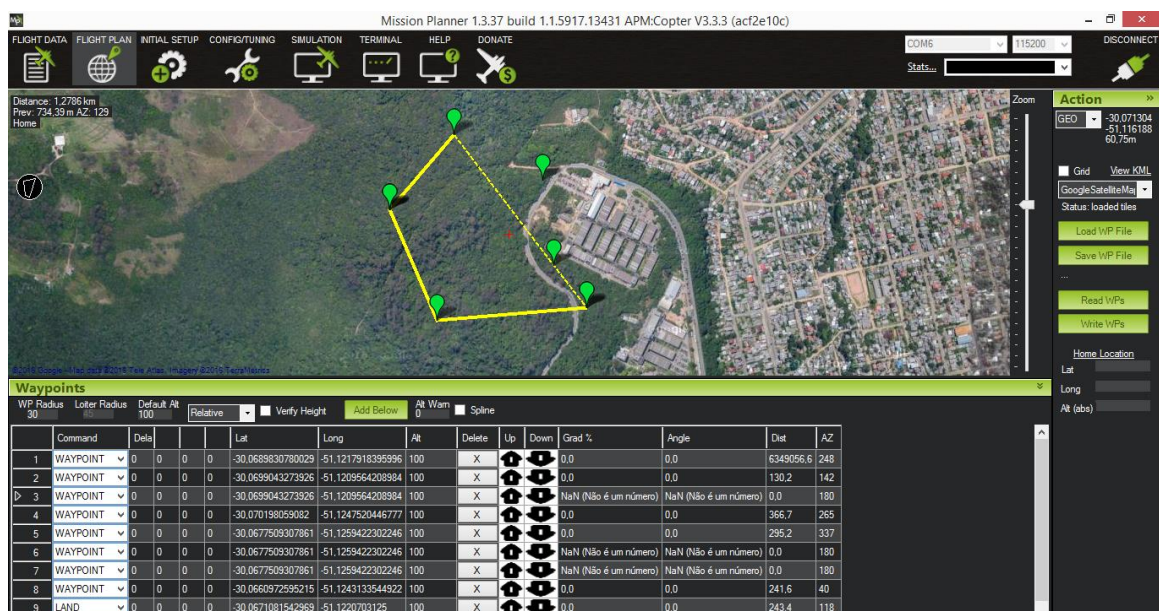
5.2.5 Visualização no Aplicativo Mission Planner

Os itens de missão enviados ao piloto automático Pixhawk são armazenados na sua memória interna e podem ser visualizados por outras aplicações que também são capazes de se comunicar com o piloto automático. Nesse sentido, o aplicativo *Mission Planner* (ARDUPILOT, 2016) foi utilizado para verificar os itens de missão e seus respectivos comandos e parâmetros armazenados na Pixhawk do IRIS+.

Esse aplicativo é um controlador do tipo estação base e possui uma interface gráfica que habilita a fácil visualização de comandos, *status* e localização do VANT controlado. Ele permite o envio de comandos e parâmetros de voo, e a visualização da posição do VANT no mapa. O *Mission Planner* além de ser usado para controlar VANTs de asas fixas ou rotativas, também pode ser utilizado como estação base para veículos terrestres que utilizam o mesmo padrão de comunicação.

A Figura 5.14 ilustra a leitura dos itens de missão e sua visualização no mapa, mostrando os comandos de navegação de cada item criados pelo módulo de controle de missão desenvolvido (o primeiro de item, com o comando de decolagem, não é armazenado pelo piloto automático). A rota traçada no mapa é calculada pelas coordenadas de cada item ordenado sequencialmente e armazenado na lista do piloto automático.

Figura 5.14 – Visualização da missão pelo Mission Planner



Fonte: *print screen* da tela no computador de teste

5.2.6 Métricas

Na Tabela 5.3 são apresentadas algumas medidas sobre os códigos desenvolvidos e suas respectivas execuções durante os testes em laboratórios descritos no capítulo 5. As métricas são divididas pelos principais módulos presentes no projeto apresentado neste trabalho. Nessas métricas são apresentados: o número de linhas de código de cada módulo – considerando tanto os arquivos de cabeçalho (extensão “.h”) e os arquivos principais (extensão “.cpp”), com linhas de comentário inclusas; o número de funções/métodos implementados em cada módulo; a porcentagem de funções de cada módulo que foi verificada durante os testes, isto é, a cobertura das funções que foram executadas durante os testes; a porcentagem de utilização dos estados da máquina de estado desenvolvida durante os testes; o número de classes presentes em cada módulo; e o número de estruturas declaradas no arquivos de cabeçalho de cada módulo e definidas com um tipo (através da diretiva *typedef*).

Tabela 5.3 – Métricas sobre os módulos

Módulo	missioncontrol	fsm	autopilot_interface	serial_port
Linhas de código	788	174	1226	693
Número de funções	20	7	22	12
Cobertura de funções nos testes	75%	100%	90,9%	100%
Cobertura dos estados da máquina de estado nos testes	61,53%	-	-	-
Número de classes	0	1	1	1
Número de <i>structs</i>	1	1	2	0

Fonte: Elaborada pelo autor

6 CONCLUSÕES

Este trabalho apresentou a implementação de um módulo de controle de missão que trata um conjunto de mensagens MAVLINK, atribui missões a VANTs através do envio de itens de missão e capaz de enviar comandos do tipo *mavlink_command_long_t*. Esse módulo é um conjunto de algoritmos executados a partir de um computador embarcado Raspberry Pi que se comunica com o piloto automático Pixhawk. Os objetivos de habilitar certa autonomia de voo e configurar missões em plataformas reais foi alcançado e validado através de testes bem sucedidos nas plataformas de computação embarcada mencionadas.

O módulo proposto foi desenvolvido através de diversas fases deste projeto. Primeiramente, as plataformas de piloto automático Pixhawk e computação embarcada Raspberry Pi foram estudadas a fim de estabelecer a compreensão de seu uso, de vantagens e de desvantagens. A partir do conhecimento adquirido sobre as plataformas, foi proposta uma arquitetura de integração entre elas. Na próxima etapa, a biblioteca MAVLINK foi estudada no intuito de entender o funcionamento de suas mensagens e seu protocolo de comunicação, e de habilitar o desenvolvimento efetivo da arquitetura proposta. Dessa forma, foi possível implementar e testar o uso de novas mensagens MAVLINK na interface de piloto automático disponibilizada com a biblioteca.

A etapa seguinte consistiu na criação de uma arquitetura para o controle de missão utilizando o conceito de máquinas de estado. Para possibilitar a implementação dessa arquitetura, foi criada uma estrutura de desenvolvimento de máquinas de estado genéricas em C++. A proposta desse controle de missão foi focada na representação dos estados da máquina como tarefas pertinentes a cada estágio de uma missão de veículos aéreos não-tripulados. Finalmente, os testes em laboratório na máquina virtual e no Raspberry Pi demonstraram que as tarefas propostas para esse controle de missão são funcionais, capacitando o módulo de controle de missão a configurar missões, ler dados de sensores e comandar pousos de emergência ao piloto automático Pixhawk.

A implementação bem sucedida do módulo de controle de missão proposto foi o primeiro passo no desenvolvimento de uma metodologia para automatizar o voo de VANTs. Os futuros passos no rumo da autonomia de VANTs estão voltados na integração do controle de missão com outros módulos de processamento como um sistema de comunicação entre VANTS – que possibilitará o desenvolvimento de aplicações para uso de múltiplas aeronaves

simultaneamente em um missão. Outro foco importante para o futuro é o desenvolvimento de um protocolo de resposta a falhas e adversidades como falha de comunicação, pouca bateria e replanejamento de rotas.

Como apresentado ao longo desse trabalho, o desenvolvimento destas tecnologias para veículos aéreos não-tripulados de baixo custo poderão beneficiar diversos setores econômicos. Haja vista a capacidade e necessidade que esses setores possuem para a rápida absorção de soluções que podem significar a diminuição de custos e riscos, e o aumento de confiabilidade e produtividade. Além disso, a sociedade civil também poderá se beneficiar a ativamente dessas novas tecnologias, possibilitando o desenvolvimento de aplicações para diversas áreas ainda deficientes como monitoramento urbano e ambiental.

REFERÊNCIAS

3D Robotics . **IRIS+ Operation Manual vH**. 2015. Disponível em: <<https://3drobotics.com/wp-content/uploads/2015/02/IRIS-Plus-Operation-Manual-vH-web.pdf>>. Acesso em: 20 out. 2015.

AGOSTINO, S.; MAMMONE, M.; NELSON, M.; TONG, Z. **Classification of Unmanned Aerial Vehicles**. [S.l:s.n], 2006.

ARDUPILOT. **Mission Planner Overview**. 2016. Disponível em: <<http://ardupilot.org/planner/docs/mission-planner-overview.html>>. Acesso em: 07 jun. 2016.

ARNING; HEINZINGER; LANGMEIER; SCHERTLER; SOBOTTA. **A View on Research Perspectives in Autonomous Systems**. Corporate Research Centre. Alemanha [s.n.], 2006. Disponível em: <http://www.dglr.de/veranstaltungen/archiv/2006_uav/dglr_uav2006_V9-2_Arning_AutonomousSysResearch%20Presentation_public_release.pdf>. Acesso em: 10 mar. 2016.

BARNARD, J. **Small UAV Command, Control and Communication Issues**. Barnard Microsystems. Reino Unido [s.n.], 2007. Disponível em: <http://www.barnardmicrosystems.com/media/presentations/IET_UAV_C2_Barnard_DEC_2007.pdf>. Acesso em : 19 mai. 2016.

BEHNCK, L. **Controle de Missão de Voo de Veículo Aéreo Não-Tripulado**. Porto Alegre: [s.n.], 2014.

BEHNCK, L.; DOERING, D.; PEREIRA, C.E.; RETTBERG, A. A Modified Simulated Annealing Algorithm for SUAVs Path Planning. **IFAC-PapersOnLine**, 48(10), p. 63-88. [S.l.], 2015.

CHAO, H.; CAO, Y.; CHEN, Y. Autopilots for Small Unmanned Aerial Vehicles: A Survey. **International Journal of Control, Automation and Systems** 8(1):36-44. [S.l.:s.n.], 2010.

CLOUGH, B. **Metrics, Schmetrics! How The Heck Do You Determine A UAV's Autonomy Anyway?** Air Force Research Laboratory, [S.l:s.n.]. Disponível em <<http://www.dtic.mil/dtic/tr/fulltext/u2/a515926.pdf>>. Acesso em: 12 mai. 2016.

DEPARTMENT OF DEFENSE. **Dictionary of Military and Associated Terms**. [S.l.:s.n.], 2001. Disponível em <www.dtic.mil/doctrine/new_pubs/jp1_02.pdf>. Acesso em: 20 out. 2015.

DOBROKHODOV, V.; JONES, K.; KAMINER, I. Rapid Flight Control Prototyping – Steps Towards Cooperative Mission-Oriented Capabilities. **2013 American Control Conference**, p. 680-685. AACC, Washington, 2013.

DOERING, D.; BENENMANN, A.; LERM, R.; FREITAS, E.P.; MULLER, I., WINTER, J.; PEREIRA, C.E. Design and Optimization of a Heterogeneous Platform for multiple UAV use in Precision Agriculture Applications. **Proceedings of the 19th IFAC World Congress**, 19(1), p. 12272-12277. [S.l.], 2014.

Element14. **Raspberry-Pi Technical Data Sheet**. 2014. Disponível em: <<http://www.element14.com/community/servlet/JiveServlet/downloadBody/65470-102-1-287848/Raspberry-Pi%20Technical%20Data%20Sheet.pdf>>. Acesso em: 25 out. 2015.

GERTLER, J. U.S. Unmanned Aerial Systems. **Congressional Research Service**. [S.l.:s.n], 2012.

GOOGLE. **Loon para todos**. Disponível em <<https://www.google.com/loon/>>. Acesso em: 12 mai. 2016.

JENKINS, B. **The Economic Impact of Unmanned Aircraft Systems Integration in the United States**. Association for Unmanned Vehicle Systems International. [S.l.:s.n], 2013. Disponível em <http://robohub.org/_uploads/AUVSI_New_Economic_Report_2013_Full.pdf>. Acesso em: 29 mai. 2016.

KANNAN, S.; RESTREPO, C.; YAVRUCUK, I.; WILLS, L.; SCHRAGE, D.; PRASAD, J. Control Algorithm and Flight Simulation Integration using The Open Control Platform for Unmanned Aerial Vehicles. **Proceedings of the 18th Digital Avionics Systems Conference**, p. 6.A.3-1 – 6.A.3-10. IEEE, St. Louis. 1999.

KOENERS, G.; DE VRIES, M.; GOOSSENS, A.; TADEMA, J.; THEUNISSEN, E. Exploring Network Enabled Airspace Integration Functions for a UAV Mission Management Station. **25th Digital Avionics Systems Conference**, p. 1-11. IEEE, [S.l.], 2006.

MacLAREN, R. **UAV Applications Challenge the Limits of Embedded Computing Technologies**. Poway: [s.n.], 2012. Disponível em <www.kontron.com/resources/collateral/articles/defensetechbriefs_article.pdf>, 2012. Acesso em: 15 mai. 2016.

MASUMOTO, Y. **Global positioning system**. Patente US5210540 A. [S.l.:s.n], 1993.

MEIER, L. **MAVLink Micro Air Vehicle Marshalling Library**. Disponível em: <<https://github.com/mavlink/mavlink/commit/a087528b8146ddad17e9f39c1dd0c1353e5991d>>

5>. Acesso em: 06 nov. 2015.

OBERLIN, P.; RATHINAM, S.; DARBHA, S. Today's traveling salesmen problem. **Robotics & Automation Magazine**, IEEE, 17(4), p. 70-77. 2010.

PIRES, R.; CHAVES, A.; BRANCO, K. Smart Sensor Protocol – a new standard for UAV and payload integration. **2014 International Conference on Unmanned Aircraft Systems**, p. 1300-1310. IEEE, Orlando, 2014.

PISCIONE, P.; VILLARDITA, A. **Power Consumption and Performance Trends on GPUs**. Università di Pisa. Pisa [s.n.], 2014. Disponível em <<http://pt.slideshare.net/AlessioVillardita/ca-1st-presentation-final-published>>. Acesso em: 15 mai. 2015.

SAYLER, K. **A world of proliferated drones: A Technology Primer**. Center for a New American Security. Washington: [s.n.], 2015. Disponível em <http://www.cnas.org/sites/default/files/publications-pdf/CNAS%20World%20of%20Drones_052115.pdf>. Acesso em: 12 mai. 2016.

SHIRAFKAN, M.T.; SEIDGAR, H.; REZAIAN-ZEIDI, J.; JAVADIAN, N. Using a Hybrid Simulated Annealing and Genetic Algorithms for Non Fixed Destination Multi-depot Multiple Traveling Salesmen Problem with Time Window and Waiting Penalty. **The Journal of Mathematics and Computer Science**, Vol. 4, No. 3, p. 428-435. [S.l.], 2012.

SONG, C.H., LEE, K.; LEE, W.D. Extended simulated annealing for augmented tsp and multi-salesmen tsp. **Neural Networks**. Proceedings of the International Joint Conference, Vol. 3, p. 2340-2343. [S.l.], 2003.

YU, Z., JINHAI, L., GUOCHANG, G., RUBO, Z., HAIYAN, Y. An implementation of evolutionary computation for path planning of cooperative mobile robots. **Intelligent Control and Automation. Proceedings on the 4th World Congress**, Vol. 3, p. 1798-1802. 2002.

APÊNDICE A – Arquivo de cabeçalho para estrutura de máquina de estado (fsm.h)

```

#include <iostream>
#include <stdio.h>
#include <cstdlib>
#include <unistd.h>
#include <cmath>
#include <string.h>
#include <inttypes.h>
#include <fstream>
#include <signal.h>
#include <time.h>
#include <sys/time.h>

#define DEFAULT_STATE_ID 0

class fsm_object;

typedef struct fsm_state
{
    int id;
    void (*function)(fsm_object*, int, void**);
    struct fsm_state *next;
}fsm_state_t;

class fsm_object
{
public:
    fsm_object();
    fsm_object(fsm_state_t *);
    fsm_object(void (*fun)(fsm_object *, int, void **));

    fsm_state_t *base;
    fsm_state_t *current;
    int current_id;
    int arg_count;
    void **arg_value;

    int fsm_start(fsm_object *obj);
    int fsm_next_state(fsm_object *obj);
    int fsm_add(fsm_object *obj, int id, void (*fun)(fsm_object *,
int, void **) );
    int fsm_default(fsm_object *obj, void (*fun)(fsm_object *, int,
void **) );
    int fsm_remove(fsm_object *obj, int id);
    int fsm_to_state(fsm_object *obj, int id, int num, void** arg);
    void fsm_terminate(fsm_object *obj);
};

```

APÊNDICE B – Arquivo da estrutura de máquina de estado (fsm.cpp)

```

#include "fsm.h"

fsm_object::
fsm_object()
{
    this->current = NULL;
    this->current_id = -1;
    this->base = NULL;
    this->arg_count = 0;
    this->arg_value = NULL;
}

fsm_object::
fsm_object(fsm_state_t *obj)
{
    this->current = obj;
    this->current_id = obj->id;
    this->base = obj;
    this->arg_count = 0;
    this->arg_value = NULL;
}

fsm_object::
fsm_object(void (*fun)(fsm_object *, int, void **))
{
    //printf("DEBUG: entered constructor for default_state\n");
    fsm_default(this, fun);
}

int fsm_object::
fsm_start(fsm_object *obj)
{
    while (!fsm_next_state(obj));
    return 0;
}

int fsm_object::
fsm_next_state(fsm_object *obj)
{
    fsm_state_t *tmp = obj->base;
    if(obj->base == NULL || obj->current_id == -1)
    {
        //printf("WARNING: FSM terminated or state not
initialized\n");
        return -1;
    }
    while(tmp->id != obj->current_id && tmp != NULL)
        tmp = tmp->next;
    if(tmp == NULL)
        return -1;
    tmp->function(obj, obj->arg_count, obj->arg_value);
    return 0;
}

```

```

int fsm_object::
fsm_add(fsm_object *obj, int id, void (*fun)(fsm_object *, int, void
**) )
{
    fsm_state_t *tmp = obj->base;
    fsm_state_t *new_state = new fsm_state_t;
    while(tmp->next)
        tmp = tmp->next;
    new_state->id = id;
    new_state->function = fun;
    new_state->next = NULL;
    tmp->next = new_state;
    return 0;
}

int fsm_object::
fsm_default(fsm_object *obj, void (*fun)(fsm_object *, int, void **) )
{
    obj->base = new fsm_state_t;
    obj->base->id = DEFAULT_STATE_ID;
    obj->base->function = fun;
    obj->base->next = NULL;
    //if(obj->base == NULL) printf("DEBUG: obj->base is NULL\n");
    //else printf("DEBUG: obj->base->id is %d\n",obj->base->id);
    obj->current = obj->base;
    obj->current_id = obj->base->id;
    return 0;
}

int fsm_object::
fsm_remove(fsm_object *obj, int id)
{
    if(id == DEFAULT_STATE_ID)
        return -1;
    fsm_state_t *to_del;
    fsm_state_t *tmp = obj->base;
    while(tmp->next != NULL && tmp->next->id != id)
        tmp = tmp->next;
    if(tmp == NULL)
        return -1;
    to_del = tmp->next;
    tmp->next->next;
    delete to_del;
    return 0;
}

int fsm_object::
fsm_to_state(fsm_object *obj, int id, int num, void** arg)
{
    fsm_state_t *tmp = obj->base;
    while(tmp != NULL & tmp->id != id)
        tmp = tmp->next;
    if(tmp == NULL)
        return -1;
    obj->current = tmp;
    obj->current_id = tmp->id;
    obj->arg_count = num;
    obj->arg_value = arg;
}

```

```
        return 0;
    }

void fsm_object::
fsm_terminate(fsm_object *obj)
{
    fsm_state_t *tmp = obj->base;
    fsm_state_t *to_del = tmp;
    while(tmp)
    {
        to_del = tmp;
        tmp = tmp->next;
        delete to_del;
    }
    obj->current = NULL;
    obj->current_id = -1;
    obj->base = NULL;
    //printf("DEBUG: terminating FSM...\n");
}
```

APÊNDICE C – Arquivo de cabeçalho para o controle de missão (missioncontrol.h)

```

#include <iostream>
#include <stdio.h>
#include <cstdlib>
#include <unistd.h>
#include <cmath>
#include <string.h>
#include <inttypes.h>
#include <fstream>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>
#include "fsm.h"
#include "autopilot_interface.h"
#include "serial_port.h"
#include "messages_test.h"
#include <common/mavlink.h>

#define FSM_STATE_DEFAULT_ID -1
#define FSM_STATE_IDLE_ID 0
#define FSM_STATE_DECIDE_TASK_ID 1
#define FSM_STATE_DEFINE_WAYPOINT_LIST_ID 2
#define FSM_STATE_DEFINE_WAYPOINT_ID 3
#define FSM_STATE_FOLLOW_PATH_ID 4
#define FSM_STATE_GO_TO_WAYPOINT_ID 5
#define FSM_STATE_GO_TO_HOME_ID 6
#define FSM_STATE_TAKEOFF_ID 7
#define FSM_STATE_END_MISSION_ID 8
#define FSM_STATE_LAND_ID 9
#define FSM_STATE_CLEAR_WAYPOINT_LIST_ID 10
#define FSM_STATE_START_MISSION_ID 11
#define FSM_STATE_READ_POI_LIST_ID 12

/*CUSTOM FSM STATES*/
void default_state(fsm_object *, int, void **);
void test_state_a(fsm_object *, int, void **);
void test_state_b(fsm_object *, int, void **);
void test_state_c(fsm_object *, int, void **);
/*Flight Control*/
void idle(fsm_object *, int, void **);
void decide_task(fsm_object *, int, void **);
void define_waypoint_list(fsm_object *, int, void **);
void define_waypoint(fsm_object *, int, void **);
void follow_path(fsm_object *, int, void **);
void go_to_waypoint(fsm_object *, int, void **);
void go_to_home(fsm_object *, int, void **);
void takeoff(fsm_object *, int, void **);
void end_mission(fsm_object *, int, void **);
void land(fsm_object *, int, void **);
void clear_waypoint_list(fsm_object *, int, void **);
void start_mission(fsm_object *, int, void **);

/*Settings Control*/
void set_velocity(fsm_object *, int, void **);

```



```

void get_status(fsm_object *, int, void **);
void define_waypoint_control(fsm_object *, int, void **);
void read_poi_list(fsm_object *, int, void **);

/*Helper functions*/
int start_mission_control(Autopilot_Interface *api);
fsm_object* create_flight_control_fsm(fsm_object *);
fsm_object* create_settings_control_fsm(fsm_object *);
void start_threads(fsm_object **, int);
void *thread_test(void *);

/****test zone starts here****
int main();
Autopilot_Interface *autopilot_interface_quit;
Serial_Port *serial_port_quit;
void quit_handler( int sig );
est zone ends here*****/
typedef enum
{
    TASK_ID_CALIBRATE,
    TASK_ID_DEFINE_WAYPOINT_LIST,
    TASK_ID_TAKEOFF,
    TASK_ID_LAND,
    TASK_ID_ENDMISSION,
}TASK_ID;

typedef struct _fsm_arg
{
    Autopilot_Interface *api;
    int waypoint_array_size;
    mavlink_mission_item_t **waypoint_array;
    uint16_t current_mission_seq;
    uint16_t send_attempt;
    uint16_t go_to_state;
    uint16_t last_state;
    bool last_state_success;
}FSM_ARG;

```

APÊNDICE D – Arquivo do controle de missão (missioncontrol.cpp)

```

#include "missioncontrol.h"

#ifdef GLOBAL_VARS_MISSIONCONTROL_H
#define GLOBAL_VARS_MISSIONCONTROL_H
Autopilot_Interface *globalApi;
bool terminateFSM = false;
pthread_mutex_t mutex;
#endif // GLOBAL_VARS_MISSIONCONTROL_H

int start_mission_control(Autopilot_Interface *api) //get *api
pointer
{
    if(api == NULL) printf("api from autopilot_interface is
null\n");
    else globalApi = api;
    fsm_object **obj = (fsm_object**)malloc(2*sizeof(fsm_object*));
    fsm_object *obj0 = new fsm_object(default_state);
    obj0 = create_flight_control_fsm(obj0);
    fsm_object *obj1 = new fsm_object(default_state);
    obj1 = create_settings_control_fsm(obj1);
    obj[0] = obj0;
    obj[1] = obj1;
    fsm_state_t *tmp = obj[1]->base;
    start_threads(obj,2);
    return 0;
}

void default_state(fsm_object *obj, int arg_count, void **arg_value)
{
    obj->fsm_to_state(obj,1,0,NULL);
}

/*          DUMMY STATES          */
void test_state_a(fsm_object *obj, int arg_count, void **arg_value)
{
    obj->fsm_to_state(obj,2,0,NULL);
}

void test_state_b(fsm_object *obj, int arg_count, void **arg_value)
{
    obj->fsm_to_state(obj,3,0,NULL);
}

void test_state_c(fsm_object *obj, int arg_count, void **arg_value)
{
    usleep(5000000);
    pthread_mutex_lock(&mutex);
    //printf(">>main FSM set to terminate when next error
happens...\n");
    terminateFSM = true; //this is used to terminate the main state
machine
    pthread_mutex_unlock(&mutex);
    obj->fsm_terminate(obj);
}

```

```

//Associate the states (functions) into one FSM
fsm_object* create_flight_control_fsm(fsm_object *obj)
{
    obj = new fsm_object(idle);
    //obj->fsm_add(obj,FSM_STATE_IDLE_ID,idle);
    obj->fsm_add(obj,FSM_STATE_DECIDE_TASK_ID,decide_task);
    obj-
>fsm_add(obj,FSM_STATE_DEFINE_WAYPOINT_LIST_ID,define_waypoint_list);
    obj->fsm_add(obj,FSM_STATE_DEFINE_WAYPOINT_ID,define_waypoint);
    obj->fsm_add(obj,FSM_STATE_START_MISSION_ID,start_mission);
    obj->fsm_add(obj,FSM_STATE_READ_POI_LIST_ID,read_poi_list);
    /*fsm_state_t *tmp = obj->base;
    if(obj == NULL) printf("DEBUG: obj is NULL\n");
    else if(obj->base == NULL) printf("DEBUG: obj->base is NULL\n");
    else if(tmp == NULL) printf("DEBUG: tmp->base is NULL\n");
    else if(tmp->next == NULL) printf("DEBUG: tmp->next is NULL\n");
    while(tmp->next != NULL) {printf("DEBUG: creating state
%d\n",tmp->id); tmp = tmp->next;}
    printf("DEBUG: end of create_flight_control\n");*/
    return obj;
}

fsm_object* create_settings_control_fsm(fsm_object *obj)
{
    obj = new fsm_object(default_state);
    obj->fsm_add(obj,1,test_state_a);
    obj->fsm_add(obj,2,test_state_b);
    obj->fsm_add(obj,3,test_state_c);

    /*fsm_state_t *tmp = obj->base;
    while(tmp->next != NULL) {printf("DEBUG: creating state
%d\n",tmp->id); tmp = tmp->next;}
    printf("DEBUG: end of create_settings_control\n");*/
    return obj;
}

//create the threads for running the FSM
void start_threads(fsm_object **obj, int count)
{
    if(pthread_mutex_init(&mutex,NULL) != 0) printf("DEBUG: Mutex
init failed\n");
    pthread_t threads[count];
    int i;
    int rc; //count = 1;
    for(i = 0; i < count; i++)
    {
        //cout << "DEBUG: creating thread: " << i << endl;
        rc = pthread_create(&threads[i], NULL, thread_test, (void
*)obj[i]);
        if (rc)
        {
            cout << "ERROR: in start_threads(), unable to
create thread," << rc << endl;
            exit(-1);
        }
    }
    for(i = 0; i < count; i++)
    {

```

```

        pthread_join(threads[i],NULL);
        //cout << "DEBUG: joined thread: " << i << endl;
    }
    pthread_mutex_destroy(&mutex);
}

void *thread_test(void *obj)
{
    //printf("DEBUG: new thread...\n");
    fsm_object *thread_obj = (fsm_object *)obj;
    /*if(thread_obj == NULL) printf("DEBUG: thread_obj is NULL\n");
    if(thread_obj->base == NULL) printf("DEBUG: thread_obj->base is
NULL\n");
    if(thread_obj->current == NULL) printf("DEBUG: thread_obj-
>current is NULL\n");
    if(thread_obj->base->next == NULL) printf("DEBUG: thread_obj-
>base->next is NULL\n");
    fsm_state_t *tmp = thread_obj->base;
    if(tmp == NULL)
    {
        printf("ERROR: in thread_test(), tmp is null\n");
pthread_exit(NULL);
    }
    while(tmp != NULL)
    {
        printf("DEBUG: state %d\n",tmp->id);
        tmp = tmp->next;
    }*/
    thread_obj->fsm_start(thread_obj);
    pthread_exit(NULL);
}

//-----//
//                               FSM STATES                               //
//-----//
/**
    @param obj: state machine object
    @param arg_count: number of args      (not used yet)
    @param arg_value: is struct FSM_ARG
*/
void idle(fsm_object *obj, int arg_count, void **arg_value)
{
    printf("IDLE state...\n");
    if(globalApi == NULL)
    {
        fprintf(stderr,"autopilot_api pointer is null in idle
state\n");
        obj->fsm_terminate(obj); //should handle (somehow) the
error
        return;
    }

    FSM_ARG *fsm_arg = (FSM_ARG*)malloc(sizeof(FSM_ARG*));
    //initialize FSM control variables
    fsm_arg->api = globalApi;    //comes from mavlink_control
    fsm_arg->waypoint_array_size = 0;
    fsm_arg->waypoint_array = NULL;

```

```

    /**test only**/
    int new_size = 10;
    fsm_arg->waypoint_array = (mavlink_mission_item_t **)
malloc(new_size*sizeof(mavlink_mission_item_t *));
    for(int i = 0; i < 10; i++)
    {
        fsm_arg->waypoint_array[i] = (mavlink_mission_item_t *)
malloc(sizeof(mavlink_mission_item_t *));
    }
    create_mission_items_test(fsm_arg->waypoint_array, new_size,
fsm_arg->api->system_id, fsm_arg->api->autopilot_id);
    fsm_arg->waypoint_array_size = new_size;
    /*for(int i = 0; i < 10; i++)
    {
        printf("mission item %d, lat: %.5f, lon: %.5f, command:
%d\n",fsm_arg->waypoint_array[i]->seq, fsm_arg->waypoint_array[i]->x,
fsm_arg->waypoint_array[i]->y, fsm_arg->waypoint_array[i]->command);
    }*/
    /**end of test code**/
    fsm_arg->last_state_success = false;
    fsm_arg->go_to_state = FSM_STATE_IDLE_ID;
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_IDLE_ID;
    Autopilot_Interface &api = (*fsm_arg->api);
    Mavlink_Messages messages = api.current_messages;

    if(messages.heartbeat.system_status != MAV_STATE_STANDBY)
    //while MAV not on standby, stay on idle (wait for preflight op)
    {
        arg_value = (void**) fsm_arg;
        obj-
>fsm_to_state(obj,FSM_STATE_IDLE_ID,arg_count,arg_value);
        return;
    }
    else
    {
        fsm_arg->last_state_success = true;
        fsm_arg->go_to_state = FSM_STATE_READ_POI_LIST_ID;
        arg_value = (void**) fsm_arg;
        obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
    }
}

//Central Control of FSM
void decide_task(fsm_object *obj, int arg_count, void **arg_value)
{
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    Autopilot_Interface &api = (*fsm_arg->api);
    bool_terminate = false;
    printf(">DECIDE TASK... last_state = %d, state_success
%d\n",fsm_arg->last_state, (int)fsm_arg->last_state_success);

    switch(fsm_arg->last_state)
    {
        case FSM_STATE_IDLE_ID:
            {
                if(fsm_arg->last_state_success)

```

```

obj->fsm_to_state(obj, fsm_arg-
>go_to_state, arg_count, arg_value);
else
{
    printf("last state failed...\n");
    fsm_arg->last_state =
FSM_STATE_DECIDE_TASK_ID;
    obj-
>fsm_to_state(obj, (int) FSM_STATE_DECIDE_TASK_ID, arg_count, arg_value);
}
break;
}
case FSM_STATE_DECIDE_TASK_ID:
{
    pthread_mutex_lock(&mutex);
    _terminate = terminateFSM;
    pthread_mutex_unlock(&mutex);
    if(_terminate)
    {
        //printf("main FSM
terminating...\n");
        obj->fsm_terminate(obj); return;
    }
    else
    {
        fsm_arg->last_state =
FSM_STATE_DECIDE_TASK_ID;
        printf("waiting...\n");
        usleep(1000000); //adjust this
timer
        obj-
>fsm_to_state(obj, (int) FSM_STATE_DECIDE_TASK_ID, arg_count, arg_value);
    }
}
case FSM_STATE_DEFINE_WAYPOINT_LIST_ID:
{
    fsm_arg->last_state =
FSM_STATE_DECIDE_TASK_ID;
    if(fsm_arg->last_state_success)
        //shouldn't come back to decide task state, just in case
        obj->fsm_to_state(obj, fsm_arg-
>go_to_state, arg_count, arg_value);
    else
    {
        printf("last state failed...\n");
        obj-
>fsm_to_state(obj, (int) FSM_STATE_DECIDE_TASK_ID, arg count, arg value);
        //handle failure (couldn't send
message or ACK timed out)
    }
    break;
}
case FSM_STATE_DEFINE_WAYPOINT_ID:
{
    if(fsm_arg->last_state_success)
        obj->fsm_to_state(obj, fsm_arg-
>go_to_state, arg_count, arg_value);
    else

```

```

        {
            printf("last state failed...\n");
            obj->fsm_terminate(obj);
            return;
        }
        break;
    }
    case FSM_STATE_START_MISSION_ID:
    {
        fsm_arg->last_state =
FSM_STATE_DECIDE_TASK_ID;
        //what to do next?
        printf("End Mission\n");
        obj->fsm_terminate(obj);
        break;
    }
    case FSM_STATE_READ_POI_LIST_ID:
    {
        if(fsm_arg->last_state_success)
        {
            obj-
>fsm_to_state(obj, FSM_STATE_DEFINE_WAYPOINT_LIST_ID, arg_count, arg_value
);
        }
        else
        {
            printf("last state failed...\n");
            obj-
>fsm_to_state(obj, FSM_STATE_DECIDE_TASK_ID, arg_count, arg_value);
            return;
        }
        break;
    }
    default:
    {
        //should handle error here ...
        obj->fsm_terminate(obj);
        break;
    }
}
return;
}

```

```

void define_waypoint_list(fsm_object *obj, int arg_count, void
**arg_value)
{
    printf("DEFINE WAYPOINT LIST state...\n");
    if(arg_count == -1 || arg_value == NULL)
    {
        fprintf(stderr, "FSM ERROR: invalid arguments \n");
        //do something
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_DEFINE_WAYPOINT_LIST_ID;
    if(fsm_arg->waypoint_array == NULL)
    {

```

```

        printf("waypoint array is null\n");
        fsm_arg->last_state_success = false;
        //obj-
>fsm_to_state(obj,FSM_STATE_IDLE_ID,arg_count,arg_value);
        obj-
>fsm_to_state(obj,(int)FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
        return;
    }
    else
    {
        Autopilot_Interface &api = (*fsm_arg->api);
        api.enable_offboard_control();
        usleep(100);

        mavlink_mission_count_t mission_count;
        mission_count.target_system = api.system_id;
        mission_count.target_component = api.autopilot_id;
        mission_count.count = fsm_arg->waypoint_array_size;
        mavlink_message_t message;

        mavlink_msg_mission_count_encode(api.system_id,api.companion_id,
        &message, &mission_count);
        uint64_t send_time = get_time_usec();
        int len = api.write_message(message);          //send
message
        if ( len <= 0 )
        {
            printf("WARNING: could not send MISSION_COUNT
\n");
            usleep(1000);
            if(fsm_arg->send_attempt < 5)
            {
                //try again
                printf("retry (%d)... \n",fsm_arg-
>send_attempt+1);
                fsm_arg->send_attempt++;
                obj-
>fsm_to_state(obj,FSM_STATE_DEFINE_WAYPOINT_LIST_ID,arg_count,arg_value
); return;
            }
            else
            {
                printf("should go to decide task...\n");
                fsm_arg->send_attempt = 0;
                fsm_arg->last_state_success = false;
                obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
                return;
            }
        }
        }else{printf("Mission count message sent\n");}
        fsm_arg->current_mission_seq = 0;
        Mavlink_Messages messages = api.current_messages;
        while(messages.mission_request.seq != (uint16_t)0 &&
            messages.time_stamps.mission_request < send_time) //what w
        {
            usleep(10000);
            messages = api.current_messages;
        }
    }
}

```



```

        fsm_arg->send_attempt = 0;
        obj-
>fsm_to_state(obj,FSM_STATE_DEFINE_WAYPOINT_ID,arg_count,arg_value);
return;
    }
}

void define_waypoint(fsm_object *obj, int arg_count, void **arg_value)
{
    printf("DEFINE WAYPOINT (n) state...\n");
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    Autopilot_Interface &api = (*fsm_arg->api);
    fsm_arg->last_state = FSM_STATE_DEFINE_WAYPOINT_ID;
    usleep(300000);

    if(api.current_messages.mission_request.seq == fsm_arg-
>current_mission_seq)
    {
        mavlink_mission_item_t **array = fsm_arg->waypoint_array;
        mavlink_mission_item_t waypoint;
        Mavlink_Messages messages = api.current_messages;
        uint16_t sequence = messages.mission_request.seq;
        //waypoint = (*(fsm_arg-
>waypoint_array[messages.mission_request.seq])); //get waypoint at
'seq' index

        waypoint = (*array[sequence]);
        if(sequence == 0)
            { //due to some (yet) unhandled error in the
waypoint_array pointer, the first item in the list has to be put
manually

                mavlink_mission_item_t mission_item;
                mission_item.target_system = api.system_id;
                mission_item.target_component = api.autopilot_id;
                mission_item.frame = MAV_FRAME_MISSION;
                mission_item.seq = 0;
                mission_item.param1 = 0;
                mission_item.param4 = 30;
                mission_item.x = 0;
                mission_item.y = 0;
                mission_item.z = 40;
                mission_item.command = MAV_CMD_NAV_TAKEOFF;
                waypoint = mission_item;
            }
        mavlink_message_t message;
        printf("sending mission item %d\ncommand %d, lat %.5f,
lon %.5f\n", sequence,waypoint.command,waypoint.x,waypoint.y);

        mavlink_msg_mission_item_encode(api.system_id,api.companion_id,&
message,&waypoint);
        uint64_t send_time = get_time_usec();
        int len = api.write_message(message);
        if ( len <= 0 )
        {

```

```

        fprintf(stderr,"WARNING: could not send
MISSION_ITEM %d \n",messages.mission_request.seq);
        if(fsm_arg->send_attempt < 5)
        {
            //try again
            fsm_arg->send_attempt++;
            obj-
>fsm_to_state(obj,FSM_STATE_DEFINE_WAYPOINT_ID,arg_count,arg_value);
            return;
        }
        else
        {
            fsm_arg->send_attempt = 0;
            fsm_arg->last_state_success = false;
            obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
            return;
        }
    }
    else
    {
        fsm_arg->current_mission_seq++;
        //usleep(500000);
        if(fsm_arg->current_mission_seq >= fsm_arg-
>waypoint_array_size &&
MAV_MISSION_ACCEPTED &&
send_time)
        {
            printf("Mission items accepted!\n");
            fsm_arg->last_state_success = true;
            fsm_arg->go_to_state =
FSM_STATE_START_MISSION_ID;
            fsm_arg->send_attempt = 0;
            obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
        }
        else
        {
            obj-
>fsm_to_state(obj,FSM_STATE_DEFINE_WAYPOINT_ID,arg_count,arg_value);
            return;
        }
    }
}
else
{
    if(api.current_messages.mission_request.seq < fsm_arg-
>current_mission_seq &&
        fsm_arg->current_mission_seq < fsm_arg-
>waypoint_array_size && fsm_arg->send_attempt < 5)
    {
        usleep(200000);
        fsm_arg->send_attempt++;
        obj-
>fsm_to_state(obj,FSM_STATE_DEFINE_WAYPOINT_ID,arg_count,arg_value);
        return;
    }
}
}

```

```

        else if(api.current_messages.mission_ack.type ==
MAV_MISSION_ACCEPTED)
        {
            printf("Mission items accepted!\n");
            fsm_arg->last_state_success = true;
            fsm_arg->go_to_state = FSM_STATE_START_MISSION_ID;
            fsm_arg->send_attempt = 0;
            obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
            return;
        }
        else
        {
            fsm_arg->last_state_success = false;
            fsm_arg->send_attempt = 0;
            obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
            return;
        }
    }

}

//NOT IMPLEMENTED YET
void follow_path(fsm_object *obj, int arg_count, void **arg_value)
{
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_FOLLOW_PATH_ID;
    Autopilot_Interface &api = (*fsm_arg->api);

    /*do something*/

    fsm_arg->send_attempt = 0;
    fsm_arg->last_state_success = true;
    obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

//NOT IMPLEMENTED YET
void go_to_waypoint(fsm_object *obj, int arg_count, void **arg_value)
{
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    Autopilot_Interface &api = (*fsm_arg->api);
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_GO_TO_WAYPOINT_ID;
    //which msg type to send?

    /**/

```

```

        fsm_arg->last_state_success = true;
        obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

//NOT IMPLEMENTED YET
void go_to_home(fsm_object *obj, int arg_count, void **arg_value)
{
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    Autopilot_Interface &api = (*fsm_arg->api);
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_GO_TO_HOME_ID;
    api.update_setpoint(api.initial_position);
    //api.write_setpoint();
    fsm_arg->last_state_success = true;
    obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

/**
    this state sends a command long message with a takeoff command,
    it is not a mission item message
*/
void takeoff(fsm_object *obj, int arg_count, void **arg_value)
{
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_TAKEOFF_ID;
    Autopilot_Interface &api = (*fsm_arg->api);
    mavlink_command_long_t takeoff_command =
create_takeoff_msg(api.system_id,api.autopilot_id);
    mavlink_message_t message;
    mavlink_msg_command_long_encode(api.system_id,api.companion_id,&
message,&takeoff_command);
    int len = api.write_message(message);
    if(len <= 0)
    {
        fprintf(stderr,"WARNING: could not send TAKEOFF
message\n");
        fsm_arg->last_state_success = false;
    }
    else
        fsm_arg->last_state_success = true;
    obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

void clear_waypoint_list(fsm_object *obj, int arg_count, void
**arg_value)

```

```

{
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    Autopilot_Interface &api = (*fsm_arg->api);
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_CLEAR_WAYPOINT_LIST_ID;
    mavlink_mission_clear_all_t clear_msg;
    mavlink_message_t message;
    mavlink_msg_mission_clear_all_encode(api.system_id,api.companion
_id,&message,&clear_msg);
    uint64_t send_time = get_time_usec();
    int len = api.write_message(message);
    if(len <= 0)
    {
        fprintf(stderr,"WARNING: could not send CLEAR WAYPOINT
LIST message\n");
        fsm_arg->last_state_success = false;
    }
    else
    {
        int timeout = 0;
        while(api.current_messages.mission_ack.type !=
MAV_MISSION_ACCEPTED  &&
        api.current_messages.time_stamps.mission_ack <
send_time)
        {
            usleep(100000);
            if(timeout++ > 5)
            {
                printf("TIMEOUT: send clear waypoint
list\n");
                fsm_arg->last_state_success =
false;
            }
        }
        if(timeout <= 5) fsm_arg->last_state_success = true;
    }
    obj->
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

void start_mission(fsm_object *obj, int arg_count, void **arg_value)
{
    printf("START MISSION state...\n");
    if(arg_count == -1 || arg_value == NULL)
    {
        //DO SOMETHING
    }
    FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
    fsm_arg->send_attempt = 0;
    fsm_arg->last_state = FSM_STATE_START_MISSION_ID;
    Autopilot_Interface &api = (*fsm_arg->api);
    fsm_arg->send_attempt = 0;
    mavlink_command_long_t start_message;

```

```

        start_message.param1 = 0; //mission item to be first run (CHECK
IF IT'S THE RIGHT FORMAT/now considering it's the item seq number)
        start_message.param2 = fsm_arg->waypoint_array_size - 1; //item
to be last run (CHECK THE FORMAT HERE TOO)
        start_message.command = MAV_CMD_MISSION_START;
        start_message.target_system = api.system_id;
        start_message.target_component = api.autopilot_id;
        mavlink_message_t message;
        mavlink_msg_command_long_encode(api.system_id,api.companion_id,&
message,&start_message);
        uint64_t send_time = get_time_usec();
        int len = api.write_message(message);
        if(len <= 0)
        {
                fprintf(stderr,"WARNING: could not send START MISSION
message\n");
                fsm_arg->last_state_success = false;
        }
        else
        {
                int timeout = 0;
                //is the response for this command an ACK?
                while(api.current_messages.mission_ack.type !=
MAV_MISSION_ACCEPTED &&
                api.current_messages.time_stamps.mission_ack <
send_time)
                {
                        usleep(100000);
                        if(timeout++ > 5)
                        {
                                printf("TIMEOUT: send clear waypoint
list\n");
                                fsm_arg->last_state_success = false;
                        }
                }
                if(timeout <= 5)
                {
                        fsm_arg->last_state_success = true;
                        printf("start mission message sent\n");
                }
        }
        obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
}

void read_poi_list(fsm_object *obj, int arg_count, void **arg_value)
{
        printf("READ PoI LIST state...\n");
        if(arg_count == -1 || arg_value == NULL)
        {
                //DO SOMETHING
        }
        FSM_ARG *fsm_arg = (FSM_ARG *) arg_value;
        fsm_arg->last_state = FSM_STATE_READ_POI_LIST_ID;
        int size = 10; //it has 10 pre-dev coordinates in
messages_test global var
        int result = read_from_poi_list(size, 2);

```

```
    if(result == 1)
    {
        fsm_arg->last_state_success = true;
        obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
        return;
    }
    else
    {
        fsm_arg->last_state_success = false;
        obj-
>fsm_to_state(obj,FSM_STATE_DECIDE_TASK_ID,arg_count,arg_value);
        return;
    }
}
```

ANEXO A – Mensagens de missão MAVLINK utilizadas

Tipo da mensagem	Parâmetros	Descrição
mavlink_mission_count_t	count	Número de itens de missão
	target_system	Veículo alvo
	target_component	Componente alvo
mavlink_mission_request_t	seq	Sequência do item
	target_system	Veículo alvo
	target_component	Componente alvo
mavlink_mission_ack_t	type	Tipo de resposta enumerada - <i>MAV_MISSION_RESULT</i>
	target_system	Veículo alvo
	target_component	Componente alvo
mavlink_mission_item_t	param1	Parâmetro de acordo com o comando enumerado – <i>MAV_CMD</i>
	param2	Parâmetro de acordo com o comando enumerado – <i>MAV_CMD</i>
	param3	Parâmetro de acordo com o comando enumerado – <i>MAV_CMD</i>
	param4	Parâmetro de acordo com o comando enumerado – <i>MAV_CMD</i>
	x	Posição x local ou latitude global
	y	Posição y local ou longitude global
	z	Posição z local ou altitude global
	seq	Sequência do item
	command	Comando, ação enumerada – <i>MAV_CMD</i>
	target_system	Veículo alvo
	target_component	Componente alvo
	frame	Sistema de coordenadas da missão
	current	Item atual da missão
	autocontinue	Continuar automaticamente ao próximo <i>waypoint</i>
mavlink_mission_item_reached_t	seq	Sequência do item alcançado
mavlink_mission_current_t	seq	Sequência do item sendo executado