

# Sistema de Aquisição de Dados de Vôo

Arthur Evangelista dos Santos  
Universidade de Brasília,  
Faculdade do Gama — UnB, FGA  
Matrícula: 14/0016686  
Email: arthuevangelista@hotmail.com

**Abstract**—Este projeto visa criar um sistema de aquisição de dados, com o uso de uma Raspberry Pi 3 Model B, para uma aeronave não tripulada, radiocontrolada e de pequeno porte. Os dados a serem adquiridos e processados serão relacionados à aeroelasticidade e referência-atitude da aeronave. Serão utilizadas três unidades de medição inercial (IMU), sendo uma em cada meia asa e uma no centro da aeronave, um módulo GPS e um *display* para apresentação dos dados obtidos por meio de uma Interface Gráfica para o Usuário (GUI).

**Keywords**—Sistemas Operacionais Embarcados, Raspberry Pi, VANT, IMU, AHRS, Aeroelasticidade.

## I. INTRODUÇÃO

A indústria aeronáutica é uma das mais vastas do mundo. A quantia de inovações tecnológicas que visam a melhoria de desempenho de uma aeronave, bem como a segurança de sua tripulação, agregam alto valor de mercado a este segmento. Para concepção de um projeto aeronáutico, se faz necessária a validação de modelos teóricos de engenharia aeronáutica e aeroespacial. Entretanto, a construção de um protótipo de uma aeronave requer grande investimento. Caso o modelo teórico não seja coerente ou condizente com a realidade de operação desta aeronave danos com insumos, recursos humanos e recursos financeiros podem acarretar na falência de uma empresa deste ramo.

Portanto, uma opção viável é a realização de um modelo em pequena escala deste projeto aeronáutico e extrapolar alguns dados obtidos nos testes em túnel de vento e em voo. Por motivos de segurança, é comum que este modelo seja radiocontrolado. A este modelo é dado o nome de *Aerodesign, Unmanned Aerial Vehicle* (UAV) ou Veículo Aéreo Não Tripulado (VANT).

A aquisição de dados pode ser realizada por um simples microcontrolador (MCU). Entretanto, separar os dados e processá-los deve ser realizado com um computador numa *groundstation* e consome tempo e recursos humanos. A automatização destes processos pode ser realizada com um *System-on-Chip* (SoC) durante a operação de voo. Com o auxílio de um GPS e um IMU pode ser traçada a trajetória de um VANT. Para automatização de voo deste veículo, os dados adquiridos pelo conjunto proposto (GPS, IMU e SoC) podem ser úteis para os algoritmos utilizados no sistema de controle da aeronave.

Um sistema utilizado na indústria é o *Attitude-Heading Reference System* (AHRS) que consiste em sensores, em conjunto com um MCU, para aquisição de dados de ângulo de atitude e referencial inercial da aeronave.

Este trabalho visa projetar e implementar um sistema de aquisição de dados de voo para um VANT que consiga adquirir

os dados de um AHRS, torção, flexão e vibração da asa e velocidade e altitude da aeronave. Alguns dos dados a serem adquiridos podem ser derivados, por meio de manipulações matemáticas, de leituras de sensores implementados para outros propósitos no sistema. Para rápida referência, é proposta a apresentação destes dados para um usuário ao final do procedimento de voo por meio de uma GUI. A aeronave a ser utilizada para testes do sistema é radiocontrolada por um piloto em *groundstation*. Este piloto seguirá o procedimento de avaliação proposto por Cooper e Harper (Seção V-B). Um trabalho a ser realizado no futuro é a integração do sistema proposto com um sistema de controle autônomo de uma aeronave.

## II. OBJETIVOS

- Adquirir dados de flexão e torção da asa;
- Adquirir dados de vibração da asa;
- Adquirir velocidade da aeronave;
- Adquirir altitude, ângulo de atitude e ângulo de ataque;
- Fusão dos dados dos acelerômetros e dos giroscópios;
- Processamento dos dados adquiridos;
- *Plot* da FFT, PDS e espectrograma (FFT/tempo);
- Fusão dos dados do IMU e do GPS com o Filtro de Kalman;
- Organizar dados de acordo com o procedimento de voo realizado;
- Apresentar resultados em uma GUI para o usuário.

## III. REQUISITOS

- Velocidade, aceleração e posição (linear e angular) da aeronave e de cada meia asa;
- Ângulo de atitude e ângulo de ataque da aeronave;
- Altitude, posição e trajetória da aeronave;
- Operações matemáticas (FFT, arctg, plot de gráficos);
- Implementação do Filtro de Kalman;
- GUI apresentando trajetória e dados adquiridos;

#### IV. BENEFÍCIOS

O benefício de se utilizar o conjunto proposto em relação ao uso de um microcontrolador é a possibilidade de se automatizar a etapa de processamento de dados durante o voo de um VANT. Isto reduz os custos com recursos humanos alocados no trabalho com estes dados. Também é viabilizado o envio destes dados para um sistema de controle afim de se automatizar o procedimento de voo. Ademais, a implementação aqui proposta possui baixo custo, quando comparado a implementações encontradas no mercado, tornando viável sua reprodução em protótipos funcionais ou outras pesquisas.

#### V. REVISÃO BIBLIOGRÁFICA

##### A. Veículo aéreo não tripulado (VANT)

Não há consenso quanto ao surgimento de VANT<sup>1</sup> por parte dos historiadores. Considera-se que o primeiro veículo aéreo não tripulado tenha surgido na Áustria em 1849 para transportar explosivos [1]. Pesquisas relacionadas a VANT para o segmento militar foram intensificadas com o advento da Primeira e Segunda Guerra Mundial. VANT foram utilizados para validar modelos de aerodinâmica e de aeroelasticidade como no projeto DAST (sigla, do inglês, *Drones for Aerodynamic and Structural Testing*) da NASA que ocorreu de 1977 a 1983 [2]. O uso de VANT com propósito militar se popularizou com a repercussão do programa classificado dos Estados Unidos desmascarado pelo governo Chinês por volta de 1982 [3].



Fig. 1. Técnicos instalam um drone BQM-43 Firebee II no pilone da asa de uma aeronave B-52B para testes no projeto DAST [2].

Atualmente, VANT são utilizados para fins militares, como monitoramento de divisas, reconhecimento territorial ou transporte de suprimentos, ou para fins civis, como em projetos de pesquisa, agricultura, transporte de bens ou simples recreação. Sua introdução no espaço aéreo não segregado ainda não é regulamentada [4] sendo uma tecnologia recentemente em ascensão quando comparada a aviação tripulada. A regulamentação sobre a operação de VANT autônomos ainda está em discussão por órgãos responsáveis como a ANAC,

<sup>1</sup>No presente trabalho, a sigla VANT será utilizada para veículo aéreo não tripulado no singular ou no plural

FAA, entre outros. Em um futuro próximo, a existência de um AHRS e uma caixa preta podem ser condições mínimas e necessárias para operação de VANT.

##### B. Avaliação Cooper-Harper

A Escala de Avaliação de Qualidade do Manuseio de Aeronaves Cooper Harper (Cooper Harper Handling Qualities Rating Scheme), comumente designada Escala Cooper-Harper, é uma escala de avaliação quantitativa e qualitativa de uma aeronave quanto à sua controlabilidade e manobrabilidade. A avaliação segue o algoritmo apresentado no Apêndice A, retirado do relatório original disponível em [5]. O piloto, após executar procedimentos de voo pré-estabelecidos, deve avaliar se houve carga de trabalho para manter a aeronave estável e em bom estado de desempenho. Seguindo o algoritmo, o avaliador deve escolher as opções que melhor representam sua experiência de manuseio. Caso tenha sido necessária excessiva interferência, o veículo necessita de alterações de projeto para que haja menor carga de trabalho e redução do risco de acidentes. Do contrário, a aeronave está em condições excelentes e não necessita de melhorias. O piloto tem ainda a possibilidade de incluir notas e observações quanto à missão executada para melhor levantamento de ameaças à qualidade do produto em estudo.

Esta escala é utilizada em VANT para verificar o grau de controlabilidade requerida pelo sistema autônomo ou semi-autônomo [?]. É como uma avaliação preliminar para levantamento de requisitos para o projeto do sistema de controle. Os níveis, a escolha da característica e a nota da avaliação podem dizer muito quanto a um sistema de controle implementado ou quanto ao grau de interferência que o projeto deste sistema deve realizar na aeronave para que haja menor, ou nenhuma, carga de trabalho por parte do condutor. Com o sistema de aquisição de dados proposto será possível avaliar as condições de voo em conjunto com a Escala Cooper-Harper. A aquisição de dados torna a avaliação menos subjetiva e otimiza tempo no momento de implementar um sistema de controle. Com os dados em mãos, a equipe de trabalho e o piloto avaliador podem buscar a melhor maneira de estudar os algoritmos a serem desenhados para determinadas trajetórias, circunstâncias de voo, procedimentos e manobras de teste realizadas durante a operação conduzida.

##### C. Aeroelasticidade

**Torção** é um tipo de deformação que ocorre quando um *torque*, também chamado de momento torsor, é aplicado no componente estrutural de modo a torcê-lo [6]. Quando da ocorrência deste torque, linhas longitudinais no componente são distorcidas. Se o componente tiver geometria circular, seções transversais ao longo do componente continuarão as mesmas após a deformação por torção, o que não ocorre com componentes de geometrias distintas como retangular, prismática e entre outras. A Fig. 2 representa um componente estrutural de geometria retangular sofrendo torção. Podem ser observadas as linhas longitudinais e as seções transversais se deformando devido ao torque aplicado.

O **ângulo de torção** pode ser definido como sendo o ângulo que um elemento do material em uma dada posição será rotacionado em relação a outro elemento do componente

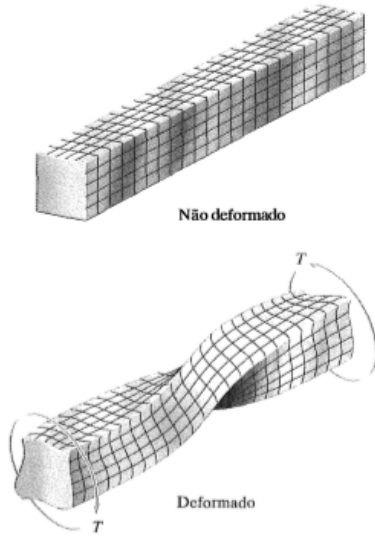


Fig. 2. Deformação por cisalhamento de um componente estrutural de geometria retangular [6].

estrutural. A expressão para o cálculo do ângulo de torção encontra-se na Equação 1 a seguir.

$$\phi(x) = \int_0^L \frac{T(x)}{J(x)G} dx \quad (1)$$

Em que  $\phi(x)$  é o ângulo de torção,  $L$  é o comprimento do componente estrutural,  $T(x)$  é o torque interno que age na seção transversal,  $J(x)$  é o momento polar de inércia da área da seção transversal,  $G$  é o módulo de rigidez ou módulo de cisalhamento do material e  $x$  é uma posição arbitrária.

A Equação 2 retirada de [6] apresenta o resultado da análise de torção para o cálculo do ângulo de torção de um componente estrutural de seção transversal quadrada de lado  $l$  em uma posição arbitrária  $x$ .

$$\phi(x) = \frac{7.10T(x)}{l^4G} \quad (2)$$

O termo  $T(x)$  pode ser isolado nesta expressão para obter a Equação 3.

$$T(x) = \frac{\phi(x)l^4G}{7.10} \quad (3)$$

Com esta equação é possível calcular o torque interno que age na seção transversal em um ponto arbitrário  $x$  a partir do ângulo de torção. Na prática, o cálculo do ângulo de torção e torque interno é realizado por uma simulação numérica utilizando o Método de Elementos Finitos (MEF). Os dados resultantes da simulação são posteriormente comparados com os valores adquiridos pelos ensaios em laboratório.

No sistema proposto, os sensores IMU medirão o ângulo de torção na longarina da asa do VANT. A diferença de posição angular entre um sensor IMU posicionado num ponto  $x_o$  da envergadura da asa e um sensor IMU posicionado no centro

da aeronave (ponto de apoio central da longarina) resultará neste ângulo de torção. Com este dado, pode ser calculado, de maneira aproximado com a Equação 3, o torque interno no ponto  $x_o$ .

O VANT MMT003, no qual serão conduzidos os testes deste sistema, possui longarina de seção transversal retangular de fibra de carbono composta com alma de divynycell<sup>®</sup> de acordo com a Fig. 3. Observa-se que a seção transversal da longarina varia na extremidade. Sendo assim, os sensores serão instalados em um ponto anterior a esta mudança de geometria. A justificativa é reduzir a complexidade computacional para este protótipo do sistema.

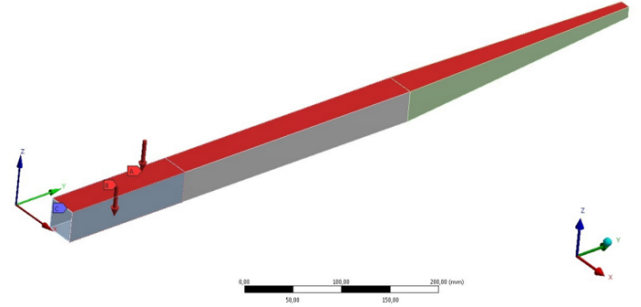


Fig. 3. Longarina do VANT MMT003. [?]

**Flexão**, assim como a torção, é um tipo de deformação que ocorre quando um momento fletor é aplicado a um componente estrutural. Sendo assim, as linhas longitudinais do objeto ficam curvadas e as linhas transversais se deformam de modo que um lado comprime e o outro alonga. Na Fig. 4 encontra-se uma ilustração deste fenômeno.

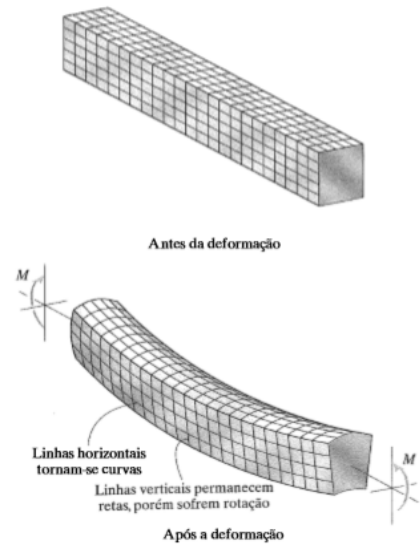


Fig. 4. Deformação por momento fletor de um componente estrutural de geometria retangular [6].

#### D. Sensores IMU

Os sensores utilizados neste trabalho são o MPU-6050 e o MPU-9250 da InvenSense. O MPU-6050, Fig. 5, é um conjunto de giroscópio de 3 eixos, acelerômetro de 3 eixos e um micro-processador integrados em um único Circuito Integrado (CI).

O MPU-9250, componente primo do MPU-6050, Fig. 6, conta com um módulo magnetômetro na segunda camada do chip que permite a realização de medidas de altitude do componente para combinar com os dados adquiridos pelo acelerômetro e giroscópio. Ambos módulos possuem a tecnologia *Digital Motion Processor™* (DMP) inclusa que permite uma etapa de pré-processamento dos dados colhidos pelos sensores antes de enviá-los pelo protocolo I<sup>2</sup>C.

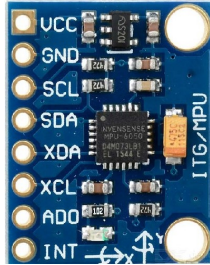


Fig. 5. Sensor MPU-6050 da InvenSense



Fig. 6. Sensor MPU-9250 da InvenSense

**Giroscópios** são unidades de medição inercial (IMU) que respondem a uma mudança de posição angular em relação ao tempo. Portanto, os dados adquiridos por um giroscópio correspondem a uma derivada da posição angular em relação ao tempo (velocidade angular). Nos sensores utilizados, a escala do giroscópio pode ser ajustada para  $\pm 250$ ,  $\pm 500$ ,  $\pm 1000$ , ou  $\pm 2000$  ° por segundo.

Para adquirir a posição angular do giroscópio deve ser realizada uma integral do dado obtido pelo tempo de acordo com a Equação 4 onde  $\theta(t)$  é a posição angular,  $\omega(t)$  é a velocidade angular medida pelo giroscópio,  $T$  é um intervalo de tempo e  $dt$  é um valor infinitesimal de tempo.

$$\theta(t) = \int_T \omega(t) * dt \quad (4)$$

Por se tratar de um sistema digital, a integração realizada deve ser numérica com valores discretos. O resultado será aproximado do valor da integral contínua cuja estimativa do erro será avaliada mais adiante. A equação utilizada na Raspberry Pi deverá ser a Equação 5 abaixo.

$$\theta(t) \approx \sum_{n=0}^N \frac{\omega[n]}{fs} + \varepsilon[n] \quad (5)$$

Em que  $n$  é o número de amostras,  $fs$  é a frequência de amostragem utilizada para adquirir os dados do sensor,  $\varepsilon[n]$  é a estimativa do erro e  $N$  é o período em que foram adquiridas as amostras.

Em geral, recomenda-se o uso de  $fs$  na ordem de 100 Hz a 200 Hz devido a lenta resposta de um sistema mecânico [7]. Entretanto, dado o propósito deste trabalho, será atualizado o valor de  $fs$  de acordo com a medição a ser efetuada. Isto é, caso a medida efetuada tenha como objetivo a **torção** ou **flexão** da asa, a frequência de amostragem pode estar entre 100 Hz e 200 Hz; Caso a medida efetuada tenha como objetivo a **vibração**, a frequência de amostragem utilizada deve estar de acordo com o teorema de amostragem de Nyquist-Shannon para a captura das frequências naturais da vibração na aeronave.

Choques e vibrações no sistema irão influenciar na medição de torção e flexão da asa. Na ocorrência de perturbações no sistema de forma que  $fs$  não respeite o teorema de Nyquist-Shannon ocorrerá o fenômeno de *aliasing* e a medição apresentará erro por deriva, também chamado de *drift* [7]. Uma maneira de lidar com o erro por *drift* é utilizar uma espécie de filtro que seja capaz de aplicar correções à deriva do sinal sem que seja alterada a informação de interesse. Uma proposta a ser avaliada é a implementação do filtro de Kalman [8] ou um filtro complementar como sugerido em [?], [9] e [10].

**Acelerômetros**, assim como Giroscópios, são IMU capazes de medir a aceleração a que estão submetidos. Consistem de uma massa de prova e sensores capacitivos. Quando a massa de prova é deslocada a diferença de capacitância é detectada e convertida. A saída deste sensor é a aceleração e são necessárias duas etapas de integração numérica para adquirir o deslocamento ao qual o sensor foi submetido. Por ser suscetível a acelerações, é comum que os dados de um acelerômetro sejam expressos em função da aceleração da gravidade  $g$ . Para testes em Brasília, será utilizada a aceleração da gravidade com valor de  $9,7808 \text{ m/s}^2$ . Nos sensores utilizados, a escala do acelerômetros pode ser ajustada para  $\pm 2g$ ,  $\pm 4g$ ,  $\pm 8g$  ou  $\pm 16g$ .

#### E. Global Positioning System (GPS)

O Sistema de Posicionamento Global, do inglês *Global Positioning System* (GPS), é um sistema de geoposicionamento desenvolvido pelo Departamento de Defesa (DoD) dos Estados Unidos. Este sistema foi desenvolvido para determinar, de maneira acurada, a posição, velocidade e tempo em um sistema comum de referências [11]. Este sistema utiliza ao menos quatro satélites em órbita e um receptor dos sinais vindos destes satélites. O dispositivo receptor calcula a distância que os satélites estão e estima sua posição utilizando um algoritmo chamado trilateração [?]. Os dados recebidos pelo GPS seguem o protocolo NMEA 0183 [?] e dizem respeito ao tempo (os satélites possuem um relógio atômico interno para ajustes devido à relatividade do tempo na órbita da Terra) e sua posição a uma constante taxa de amostragem.

Trilateração é um método de estimar a localização de um vetor usando a geometria de circunferências, para o caso bidimensional, ou esferas, para o caso tridimensional [?]. Quando os dados são recebidos, calcula-se as esferas aproximadas nas quais é possível que o receptor se encontre. A interseção



entre estas esferas, a grosso modo, será a posição na Terra aproximada do receptor.

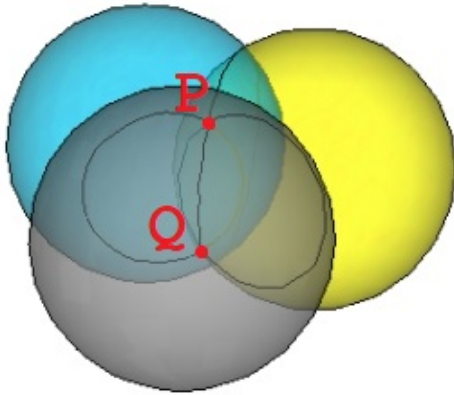


Fig. 7. Exemplo gráfico tridimensional da localização por trilateração. As interseções das esferas (pontos P e Q) são possíveis localizações do receptor GPS.

São utilizados, no mínimo, quatro satélites para calcular a posição do receptor. Desta forma, a posição calculada é muito próxima da localização exata. Quanto maior o número de satélites identificados, mais informações serão recebtadas e com maior acurácia ocorrerá a estimativa da localização do dispositivo. Outros dados são derivados destes cálculos como a velocidade, altitude e direcionamento. Uma falha no sistema de GPS é sua indisponibilidade em ambientes fechados como em prédio e túneis, ou obstáculos como árvores e pontes. Sendo assim, sistemas que necessitam estimar sua localização utilizam a fusão dos dados de um GPS e de acelerômetros e giroscópios [8].

O Módulo GPS utilizado neste trabalho será o módulo da u-blox NEO-6M apresentado na Fig. 8. De acordo com [?], a alimentação deste módulo é 3.3V e possui capacidade para interfaceamento com a Raspberry Pi por meio dos protocolos UART, SPI, USB e I<sup>2</sup>C. Para o projeto foi escolhida a comunicação por meio de UART por não depender do endereçamento da I<sup>2</sup>C, consumido com os três sensores IMU, não utilizar muitos fios para conexão e por não estar sujeito ao congestionamento da linha de comunicação entre os dispositivos conectados na I<sup>2</sup>C.

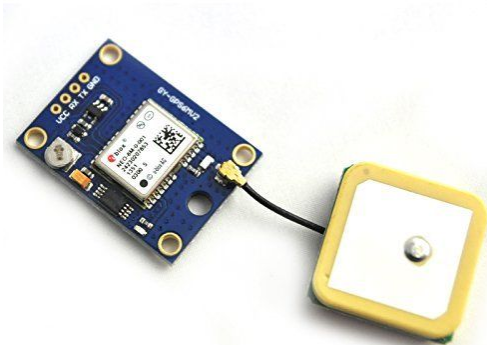


Fig. 8. Módulo GPS NEO-6M da u-blox.

## VI. HARDWARE

Os componentes a serem utilizados estão listados na Tabela I. O GPS está conectado à SoC pela porta serial UART ttyS0. Os sensores MPU-6050 conectam-se à Raspberry Pi por meio do protocolo I<sup>2</sup>C. O sensor posicionado na asa direita possui endereço I<sup>2</sup>C 0x68 e o sensor posicionado na asa esquerda possui endereço I<sup>2</sup>C 0x69. O sensor MPU-9250 posicionado no centro da aeronave possui endereço padrão SPI. O buzzer está na porta GPIO 7, a saída para modificação do endereço do sensor MPU-6050 da asa esquerda está na porta GPIO 18 e a chave de controle está na porta GPIO 20.

TABLE I. BOM (BILL-OF-MATERIALS)

Quantidade	Componente
1	Raspberry Pi 3 Model B+
2	MPU-6050
1	MPU-9250
1	GPS ublox NEO-6M
1	Buzzer Ativo
1	Switch
2	Resistor 1K $\Omega$
9	Jumper fêmea-fêmea
16	Jumper macho-fêmea
3	Jumper macho-macho

### A. Esquemático

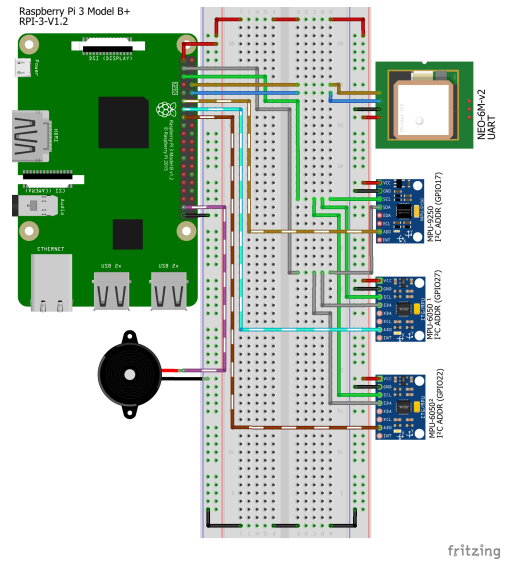


Fig. 9. Esquemático do sistema realizado no software fritzing.

## VII. SOFTWARE

O código foi construído na linguagem C, por exceção dos módulos de interface com os sensores IMU, que utiliza recursos do C++. A compilação deste código é realizada por meio de um makefile que cria os objetos para cada biblioteca e módulo utilizado, realiza o link destes objetos e cria um arquivo executável.

Neste trabalho, o arquivo executável chama-se "final-PROG". Para alterar o nome do executável basta alterar o valor da macro `MAKE_TARGET` no arquivo makefile para o nome desejado. Este executável é salvo na pasta `"/usr/local/bin/"` e pode ser executado em qualquer posição

do terminal. As referências dos arquivos são diretas, evitando referências por links que podem ser inválidos a depender a posição do terminal.

As bibliotecas utilizadas ficam salvas na pasta `./libs`, os arquivos de inicialização na pasta `./initFiles`, os objetos na pasta `./objects` e uma pasta `./Output` contém uma cópia do arquivo executável criado na pasta de binários do sistema. É aconselhável manter a organização das pastas como a contida no repositório do github para evitar falhar de referência de arquivos por parte do compilador, a menos que sejam alterados os caminhos para acesso destes no arquivo `makefile`.

### A. Algoritmo

1) *Sensores IMU*: Para interfaceamento dos sensores IMU com a Raspberry Pi foi utilizada a biblioteca RTIMULib, do usuário do github Richard Barnett. Esta realiza a calibração e leitura dos sensores e, posteriormente, o processamento dos dados de cada sensor individual de acordo com um arquivo de configuração. O arquivo de configuração é criado no momento da calibração dos sensores ou é gerado automaticamente caso não sejam encontrados arquivos de configuração.

Foi criada uma pasta para cada IMU com seu respectivo arquivo de configuração, uma vez que esta biblioteca apenas reconhece `RTIMULib.ini` como nome válido para o arquivo. `RTIMULib.ini` possui o protocolo e endereço a ser acessado para aquisição dos dados do sensor, e os parâmetros de configuração, calibração e processamento. Como referido na Seção VI, um MPU-6050 é posicionado na asa direita com endereço I<sup>2</sup>C 0x68, outro é posicionado na asa esquerda com endereço I<sup>2</sup>C 0x69 e um MPU-9250, posicionado no centro da aeronave, possui endereço padrão SPI.

No módulo principal são criados três ponteiros para objetos da classe RTIMU e um ponteiro para uma struct local do tipo `imuDataAngulo` que receberá os dados processados pela biblioteca RTIMULib. Um contador também é declarado para identificação de qual sensor IMU estará sendo lido no momento. Estes objetos, a struct e o contador serão passados para as funções do módulo de implementação dos sensores IMU que funciona como um *wrapper* para as funções da biblioteca utilizada.

Um teste foi realizado utilizando o aplicativo padrão RTIMULibDemo para o sensor MPU-6050, Fig. 10, e para o sensor MPU-9250, Fig. 11. Posteriormente a estes testes fora realizada a calibração de cada sensor e suas configurações de inicialização salvas em uma pasta separada, como citado anteriormente.

O módulo de implementação dos sensores é composto por duas funções, uma de inicialização e outra de leitura. A função de inicialização recebe o contador para identificação do sensor que é usado como argumento de uma rotina condicional. A depender do valor do contador, o arquivo de configuração será aberto e associado ao objeto da classe RTIMU. Caso a identificação do sensor e a associação de suas configurações ao objeto tenham sido bem sucedidas, o buzzer apitará. O retorno da função de inicialização é o próprio objeto.

A função de leitura recebe um ponteiro para a struct `imuDataAngulo` e o objeto, realiza um pedido de *poll* dos dados do sensor, aguarda o intervalo de polling, realiza a leitura

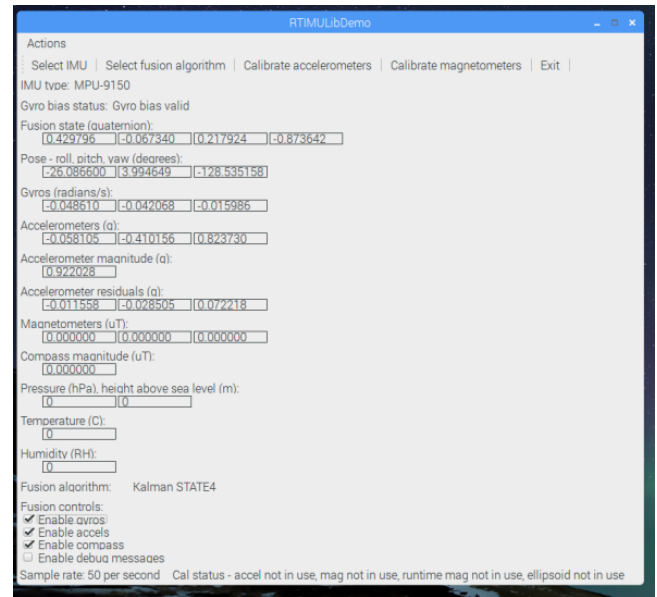


Fig. 10. Teste do sensor MPU-6050 utilizando aplicativo padrão.

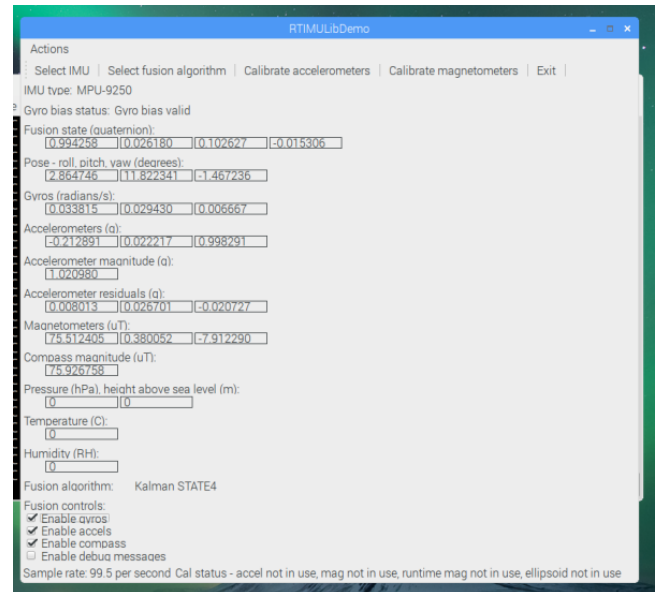


Fig. 11. Teste do sensor MPU-9250 utilizando aplicativo padrão.

destes dados, e os salva nos respectivos elementos da struct. Após uma leitura bem sucedida de dados, caso o sensor lido seja o MPU-9250, o módulo principal solicita a chave mutex destinada à struct `uav` e guarda os dados de *roll*, *pitch* e *yaw*. Este procedimento está ilustrado na Fig. 12 a seguir.

Após a leitura dos três sensores, são criadas duas threads para que os dados sejam processados e salvos na struct `uav`. Nesta thread são calculados o ângulo de torção e flexão, Seção V-C, em relação ao sensor no centro da aeronave. O módulo principal aguarda até que as threads retornem para chamar a função que armazena os dados calculados e adquiridos em um arquivo de log.

2) *Receptor GPS*: Foi utilizado o *daemon* GPSD para interfaceamento da Raspberry Pi com o GPS. Este serviço

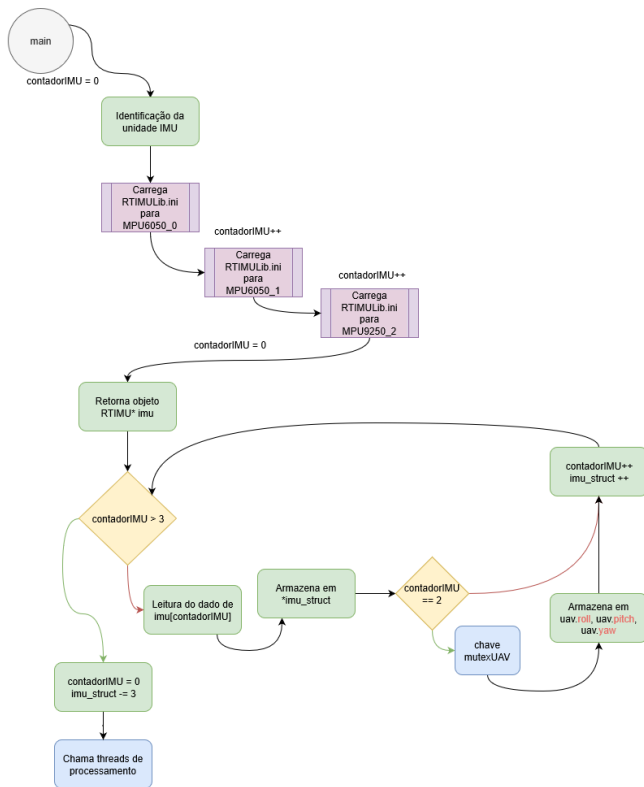


Fig. 12. Fluxograma do algoritmo de leitura dos sensores IMU.

monitora os receptores GPS e traduz suas informações de padrão NMEA para variáveis, utilizando uma biblioteca em C para uso em aplicações externas, como a proposta por este projeto. A biblioteca possui a opção de se realizar o *polling* do GPS por meio do protocolo TCP ou por meio de variáveis locais no sistema operacional. Prezando pela redução de *overhead* no sistema, foi utilizada a opção de *polling* com a variável local. Esta opção é escolhida pela macro `GPSD_SHARED_MEMORY` enviada como primeiro argumento da função `gps_open()` da biblioteca.

Executou-se um teste utilizando um aplicativo chamado "cgps", padrão da distribuição do serviço GPSD. O resultado deste teste pode ser observado na Fig. 13. As informações de tempo, latitude e longitude foram desfocadas para evitar rastreamento da localização em que foi realizado este teste. Foi passado "-" como argumento para suprimir as informações das strings no padrão NMEA recebidas pelo gps e concatenadas para a saída do console do terminal. Nesta e demais aplicações o tipo de FIX é importante para escolher quais informações possuem 95% de confiabilidade.

- FIX 0 - Não existem informações suficientes dos satélites para compor alguma variável;
- FIX 2 - Apenas informações de tempo, velocidade horizontal, latitude, longitude e seus respectivos erros possuem confiabilidade suficiente; e
- FIX 3 - Todas as informações repassadas pelo receptor GPS são confiáveis e podem ser utilizadas.

Na aplicação fora separada uma thread para o constante monitoramento do GPS, em paralelo com a aplicação principal.

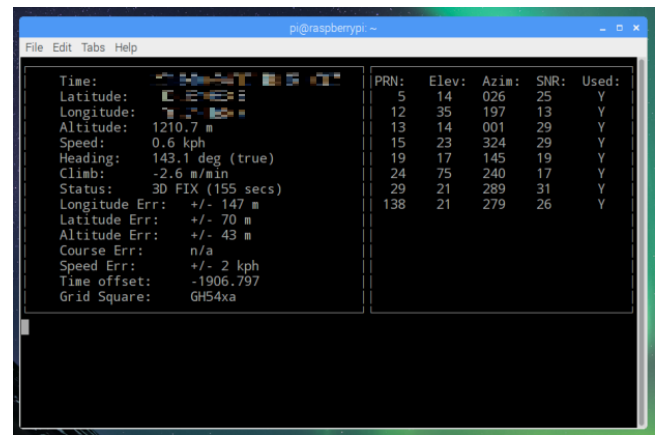


Fig. 13. Resultado do teste do GPS com o *daemon* GPSD.

O fluxograma da Fig. 14 explica de maneira gráfica o funcionamento desta thread e sua interação com a função principal. Para utilização do *daemon* GPSD se faz necessária a declaração de uma struct global do tipo `gps_data_t*`, neste trabalho chamada de "dataGPS", que será passada para as funções de abertura, leitura e encerramento do GPS. A alocação de memória desta variável foi realizada no interior da thread para evitar *overhead* do sistema.

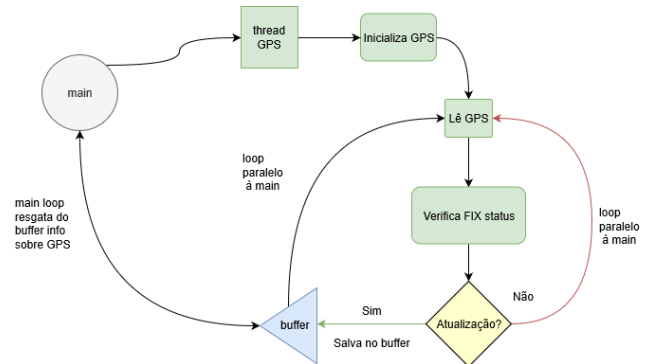


Fig. 14. Fluxograma do algoritmo da thread do GPS.

Alguns comandos devem ser passados para o console do terminal todo momento que uma aplicação externa deve requisitar o serviço GPSD. Estes comandos são executados pelo código utilizando a chamada do sistema `system()`. Quando a inicialização do GPS e alocação de suas respectivas variáveis é realizada com sucesso, o buzzer de sinalização de status apita duas vezes e o LED interno do módulo GPS ficará piscando enquanto o receptor GPS estiver recebendo informações de um satélite.

Esta variável global está atrelada a uma chave mutex que será utilizada para proteção de dataGPS afim de evitar erros de segmentação. O buffer referido pela Fig. 14 são elementos da struct "uav" que poderão ser acessados por outras threads sem a necessidade de se interromper a thread separada para o GPS. Os dados recolhidos do GPS são:

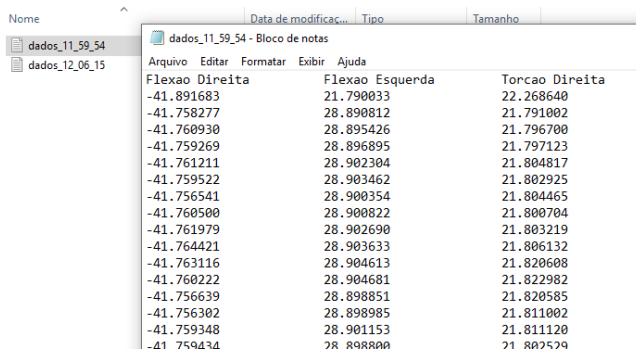
- latitude;
- longitude;
- altitude;

- velocidade nos eixos X e Y;
- velocidade de subida (eixo Z);
- Tempo atual; e
- Tipo de FIX.

O encerramento da thread e do GPS são realizados quando o sistema recebe o sinal de interrupção, como descrito na Seção VII-A. A memória alocada para as variáveis globais também é liberada.

3) *Arquivo de log do teste*: O arquivo de log é criado no módulo principal antes do loop principal controlado pelo switch. Este arquivo consiste em um arquivo de texto dados\_\_TIME\_\_.txt, onde \_\_TIME\_\_ é uma constante do sistema operacional que remete ao tempo atual do sistema em horas, minutos e segundos. Este arquivo possui formato de tabela para posterior visualização e construção de gráficos.

Sua inicialização é realizada com os campos de cada coluna de dados e, no loop principal, é realizada uma chamada para função "fileHandler" que insere as informações adquiridas e armazenadas na struct "uav". A divisão de colunas é realizada com "tab" para que programas de manipulação de tabelas como Microsoft Excel e derivados possam importá-la. Um exemplo deste arquivo de log encontra-se na Fig. 15 a seguir. Observa-se que os logs de dados podem ser facilmente identificados com relação ao tempo de início do teste.



Flexao Direita	Flexao Esquerda	Torcao Direita
-41.891683	21.790033	22.268640
-41.758277	28.890812	21.791002
-41.760930	28.895426	21.796700
-41.759269	28.896895	21.797123
-41.761211	28.902304	21.804817
-41.759522	28.903462	21.802925
-41.756541	28.900354	21.804465
-41.760500	28.900822	21.800704
-41.761979	28.902690	21.803219
-41.764421	28.903633	21.806132
-41.763116	28.904613	21.820608
-41.760222	28.904681	21.822982
-41.756639	28.898851	21.820585
-41.756302	28.898985	21.811002
-41.759348	28.901153	21.811120
-41.759434	28.898800	21.807529

Fig. 15. Exemplo de arquivo de log aberto para análise.

Um contador "tamanhoFFT" é passado como argumento desta função para que seja incrementado toda vez que os dados são inseridos na tabela. O contador é retornado pela função "fileHandler" para atualização do contador no loop principal. A variável, como o nome indica, será utilizada no processamento da Transformada Rápida de Fourier (FFT) indicando o tamanho do vetor de dados que será passado.

4) *Tratamento do Sinal de Interrupção*: Uma função de tratamento do sinal de interrupção fora implementada. É utilizado *sigaction* POSIX para o tratamento de sinais. Esta função encerra as threads em andamento, encerra o GPS, libera a memória alocada pelas variáveis com alocação dinâmica, libera e destrói as chaves mutex utilizadas e finaliza o programa com `EXIT_SUCCESS`. Esta função é de vital importância para que o programa seja encerrado de maneira adequada e não existam processos zumbis ou variáveis fantasmas consumindo memória.

No módulo principal, enquanto o switch de controle está ativado, ocorre o loop principal que realiza a leitura e processamento dos dados dos sensores e do GPS. Quando este switch é desativado, a função main realiza a chamada POSIX `sigqueue` para colocar o sinal de interrupção SIGINT na fila de sinais para o processo com o PID da própria função main. Deste modo, a função main executará o tratamento deste sinal e encerrará o programa.

## VIII. RESULTADOS

Para atestar o funcionamento do sistema, foi montado um protótipo funcional do sistema de acordo com o esquemático da Seção VI-A. Quando finalizada a produção do código e montagem do protótipo, a aeronave MMT-003 não estava disponível para testes. Portanto, no lugar da longarina da aeronave utilizou-se uma barra metálica. Uma imagem do protótipo funcional pode ser observado na Fig. 16 a seguir. O propósito dos testes executados foi aferir a qualidade do código produzido e a veracidade dos dados adquiridos.

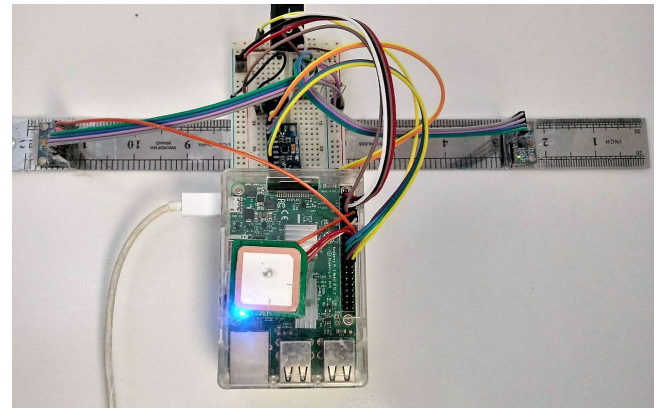


Fig. 16. Protótipo funcional para testes.

Após o início da execução do programa, é apresentada os status de inicialização dos sensores em conjunto com o toque do buzzer. Logo em seguida, a tela do terminal é limpa utilizando a chamada de sistema `system("clear")` e os dados começam a ser apresentados na tela para o usuário, Fig. 17. Para visualização da tela da Raspberry Pi, foi utilizado o serviço VNC Viewer que acessa por wlan a tela da SoC e envia dados do mouse e teclado para a placa.





Fig. 17. Sistema apresenta dados no terminal durante execução de testes.

Calibrou-se os acelerômetros e giroscópios dos três sensores IMU. O MPU-9250 teve seu magnetômetro calibrado e gerou-se um ajuste elipsoidal, Fig. 18, para melhor precisão dos dados. O GPS fora testado em outras localidades para assegurar seu funcionamento. Dessa forma fora delimitado que não seria possível adquirir dados do GPS no interior de prédios, túneis ou embaixo de pontes. Uma mensagem no terminal é então exibida quando isto ocorre.



Fig. 18. Pontos adquiridos para ajuste elipsoidal. Gráfico construído no software Octave.

Após esta etapa de calibração, foram realizados testes estáticos para aferir o erro de medida dos equipamentos. Nos

gráficos das Fig. ?? e ?? observa-se a flutuação da medida estática de roll, pitch e yaw para os sensores MPU-6050 e MPU-9250, respectivamente. As medidas de roll, pitch e yaw estão em graus. Esta flutuação diz respeito ao drift e bias na situação estática, em que os ângulos deveriam apresentar zero ou uma medida constante.

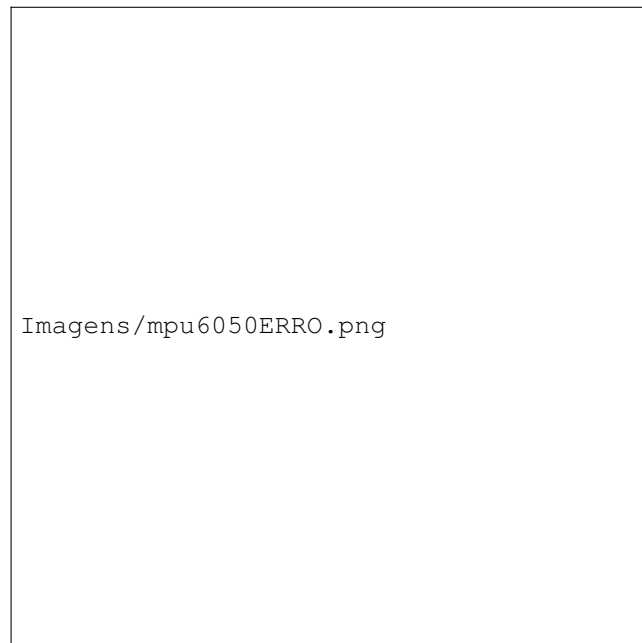


Fig. 19. Flutuação da medida estática de roll, pitch e yaw do MPU-6050.



Fig. 20. Flutuação da medida estática de roll, pitch e yaw do MPU-9250.

O gráfico da Fig. ?? representa o erro do GPS em modo de FIX 3 para latitude, longitude, altitude e velocidade de subida. As medidas de latitude e longitude estão em graus. Altitude é medido em metros e velocidade de subida em metros por segundo. Este erro é ocasionado pelo número de satélites

encontrados, o erro de medida de cada satélite observado, o efeito Sagnac, gravidade e a rotação da Terra.



Fig. 21. Erro do GPS em modo de FIX 3.

Em seguida foram realizados testes para aferir a precisão da medida de flexão e torção em cada longarina. Nas Fig. ?? e ?? são apresentadas as medidas na longarina direita e esquerda, respectivamente, quando estas são flexionadas e torcionadas.



Fig. 22. Medida de flexão e torção na longarina Direita.



Fig. 23. Medida de flexão e torção na longarina Esquerda.

## IX. CONCLUSÃO

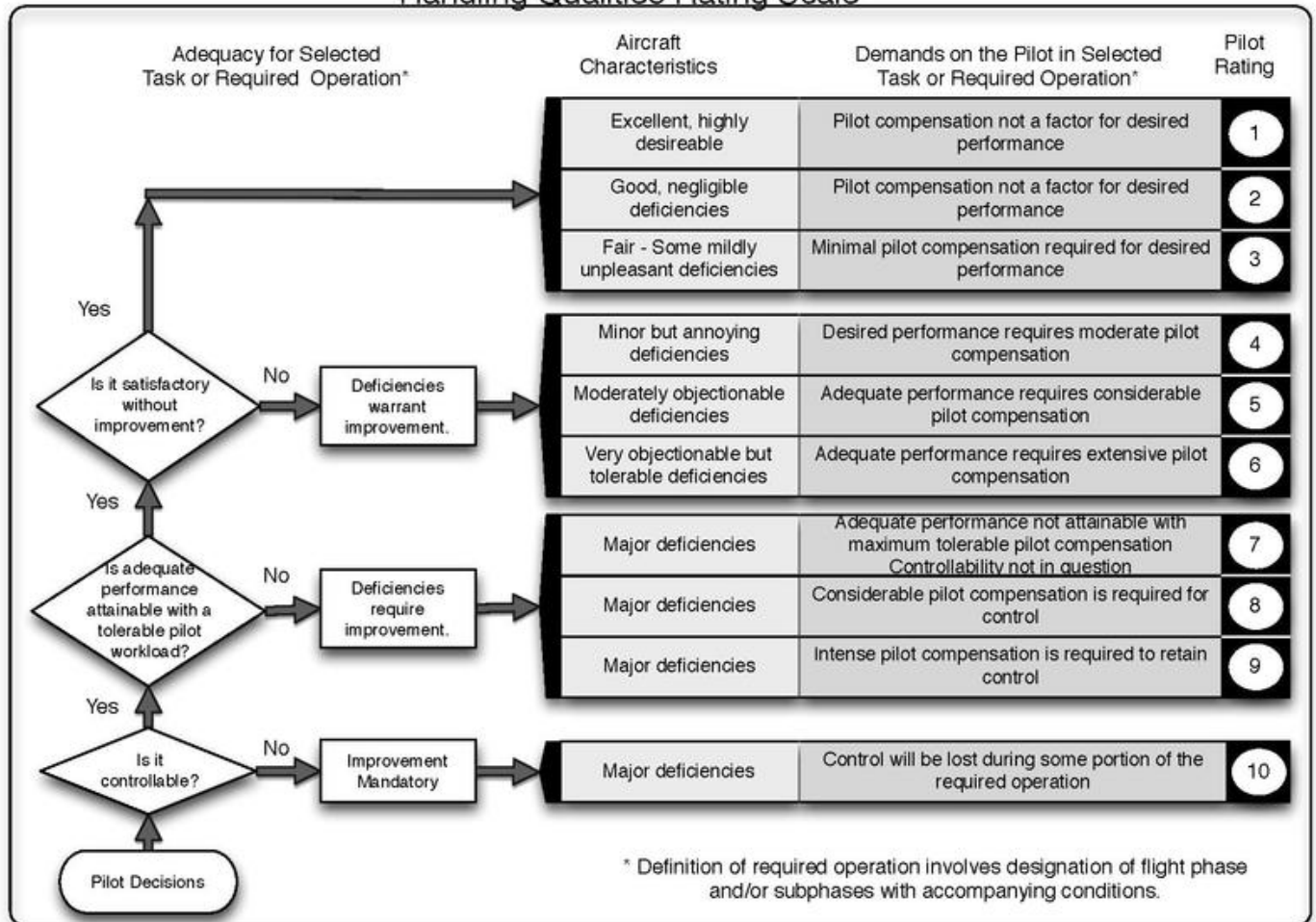
### REFERENCES

- [1] B. Custers, *The Future of Drone Use: Opportunities and Threats from Ethical and Legal Perspectives*, ser. Information Technology

- and Law Series. T.M.C. Asser Press, 2016. [Online]. Available: <https://books.google.co.il/books?id=WytEDQAAQBAJ>
- [2] H. Murrow and C. Eckstrom, “Drones for aerodynamic and structural testing,” in *Aircraft Systems and Technology Conference*. NASA Langley Research Center: National Aeronautics and Space Administration, Aug 1978.
- [3] W. Wagner, *Lightning Bugs and Other Reconnaissance Drones*. Armed Forces Journal, 1982. [Online]. Available: <https://books.google.com.br/books?id=L-xzQgAACAAJ>
- [4] R. A. V. Gimenes, “Método de avaliação de segurança crítica para a integração de veículos aéreos não tripulados no espaço aéreo controlado e não segregado,” Escola Politécnica, 2015.
- [5] G. E. Cooper and R. P. Harper, “The use of pilot rating in the evaluation of aircraft handling qualities,” National Aeronautics and Space Administration, Washington D.C., Tech. Rep. NASA-TN-D-5153, April 1969.
- [6] R. C. Hibbeler, *Resistência dos Materiais*, 7th ed. Pearson Prentice Hall, 2010.
- [7] P.-J. V. de Maele, “Getting the angular position from gyroscope data,” Disponível em: <https://www.pieter-jan.com/node/7>, Sept 2012, acessado em: 24/07/2018.
- [8] M. S. Grewal and A. P. Andrews, *Kalman Filtering: Theory and Practice Using MATLAB*, 2nd ed., ser. Wiley-Interscience. John Wiley & Sons, Inc., 2001.
- [9] M. Euston, P. Coote, R. Mahony, J. Kim, and T. Hamel, “A complementary filter for attitude estimation of a fixed-wing uav,” in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sept 2008, pp. 340–345.
- [10] P.-J. V. de Maele, “Reading a imu without kalman: The complementary filter,” Disponível em: <https://www.pieter-jan.com/node/11>, April 2013, acessado em: 24/07/2018.
- [11] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins, *Global Positioning System: Theory and Practice*. Springer Vienna, 2012. [Online]. Available: <https://books.google.com.br/books?id=F7jrCAAAQBAJ>

APPENDIX A  
ESCALA DE AVALIAÇÃO COOPER-HARPER

## Handling Qualities Rating Scale



## APPENDIX B

### QUADRO PLANEJAMENTO

#Hashtag					Sistema de Aquisição de Dados				
Equipe		Datas		Por que?		Objetivos		Ameaças ao Projeto	
				Por que não?					
Arthur Evangelista dos Santos (14/0016686)		PC1 - 29/03 PC2 - 29/04 PC3 - 27/05 PC4 - 10/06 FINAL - 05/07		A aquisição de dados pode ser realizada por um simples MCU. Entretanto, separar os dados e processá-los deve ser realizado com um computador numa groundstation e consome tempo e recursos humanos. A automatização destes processos pode ser realizada com um SoC (no presente trabalho uma raspberry pi) durante o voo. Com o auxílio de um GPS e um IMU pode ser traçada a trajetória de um VANT. Para automatização de voo deste VANT, os dados adquiridos pelo AHRS (conjunto do GPS, IMU e SoC para aquisição de dados de voo) podem ser utilizados para o algoritmo utilizado no sistema de controle da aeronave.		- Adquirir dados de flexão e torção da asa; - Adquirir velocidade da aeronave; - Adquirir dados de vibração da asa; - Adquirir altitude, ângulo de atitude e ângulo de ataque; - Fusão dos dados dos acelerômetros e dos giroscópios; - Processamento dos dados adquiridos; - Plot da FFT, PDS e espectrograma (FFT/tempo); - Fusão dos dados do IMU e do GPS com o Filtro de Kalman; - Organizar dados de acordo com o procedimento de voo realizado; - Apresentar resultados em uma GUI para o usuário;			
Fábio Barbosa Pinto (11/0116356)				Talvez não seja possível implementar todas as características propostas para o projeto até a data limite. Ademais, a ideia deste sistema de aquisição de dados não é original, sendo algo implementado e estudado pela indústria aeroespacial desde o lançamento da missão Apollo à Lua. Ou seja, já existem soluções na indústria para o problema apresentado. Outro impecilho são os custos elevados para obtenção dos sensores e componentes para o projeto. Talvez o barateamento dos custos afete a precisão dos dados adquiridos.					
Custos					Requisitos				
- Raspberry pi 3 Model B+ [R\$ 190,00] - Case impresso 3D [Indefinido] - 2 x MPU-6050 [R\$ 19,90] - Tela para raspi (touch ou não) [Indefinido]					- Módulo GPS [R\$ 119,90] - Jumpers [Indefinido] - MPU-9250 [R\$ 99,90] - Velocidade, aceleração e posição (linear e angular) da aeronave e de cada meia asa [accel] - Ângulo de atitude e ângulo de ataque da aeronave [gyro] - Altitude, posição e trajetória da aeronave (fusão dos dois últimos requisitos com módulo gps) [accel + gyro + gps] - Operações matemáticas (FFT, arctg, plot de gráficos) [octave, matlab, scilab] - Implementação do Filtro de Kalman [octave, matlab, scilab] - GUI apresentando trajetória e dados adquiridos [GTK+, Visual Studio, Processing]				



# APPENDIX C DIAGRAMA LÓGICO DO SISTEMA

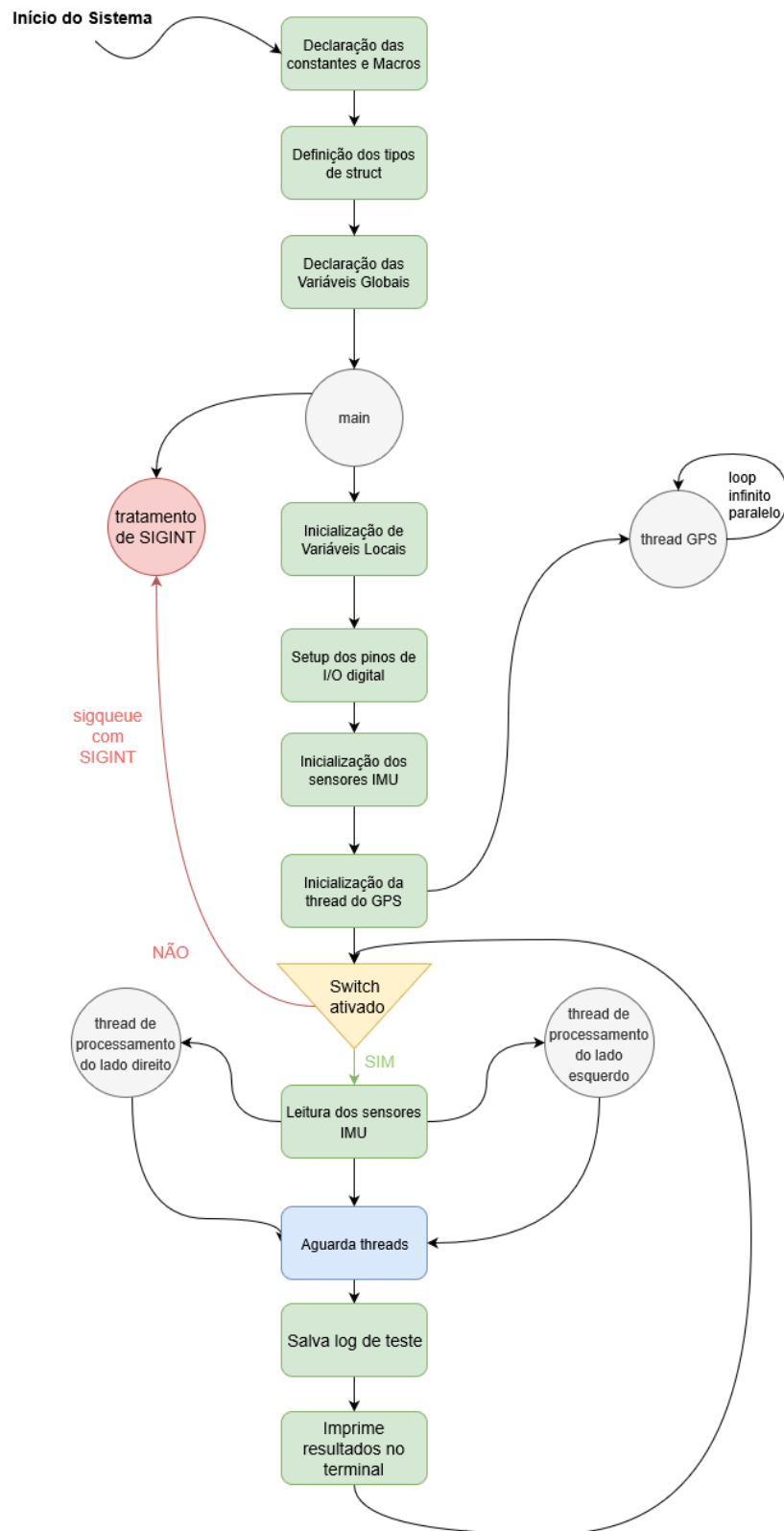


Fig. 24. Diagrama lógico do sistema.

## APPENDIX D

### CÓDIGO DA RASPBERRY

#### A. Módulo de implementação do GPS

- Arquivo .c

```
1  /* =====
2  * Pacote de implementacao das funcoes do GPS
3  * =====
4  * Devem ser adicionadas as flags a seguir no momento da compilacao:
5  * -lm -lgps
6  *
7  * Exemplo:
8  * g++ -o output filename.cpp -lm -lgps
9  * =====
10 */
11
12 #include "implementacaoGPS.h"
13
14 // =====
15 // Funcao de inicializacao do GPS
16 // =====
17 void initGPS(gps_data_t* dataGPS){
18     int rc;
19
20     printf("Iniciando GPS ...\n");
21
22     // Sys calls para configuracao do gpsd como GPSD_SHARED_MEMORY
23     system("sudo systemctl stop serial-getty@ttyS0.service");
24     system("sudo systemctl disable serial-getty@ttyS0.service");
25     system("sudo systemctl stop gpsd.socket");
26     system("sudo systemctl disable gpsd.socket");
27     system("sudo killall gpsd");
28     system("sudo gpsd -n /dev/ttyS0 -F /var/run/gpsd.sock");
29
30     sleep(2); // Aguarda por 2 segundos
31     buzzerTone('D',300);
32     buzzerTone('F',600);
33     buzzerTone('X',100);
34
35     if((rc = gps_open(GPSD_SHARED_MEMORY, NULL, dataGPS)) == -1){
36         printf("Erro na inicializacao do GPS na linha # %d\nRazao do erro: %s\n", __LINE__, gps_errstr(rc));
37         exit(EXIT_FAILURE);
38     }
39 } // FIM DA FUNCAO initGPS
40
41 // =====
42 // Funcao de leitura dos dados do GPS
43 // =====
44 void leituraGPS(gps_data_t* dataGPS){
45     int rc = 0;
46     if ((rc = gps_read(dataGPS)) == -1) {
47         printf("Erro ao realizar a leitura do GPS na linha # %d. Codigo do erro: %s\n", __LINE__,
48             gps_errstr(rc));
49     } else {
50         if ((dataGPS->status == STATUS_FIX) &&
51             (dataGPS->fix.mode == MODE_2D || dataGPS->fix.mode == MODE_3D) &&
52             !isnan(dataGPS->fix.latitude) &&
53             !isnan(dataGPS->fix.longitude)) {
54             printf("Sem dados do GPS disponiveis!\n");
55         }
56     } // FIM DO IF DE ESPERA ENTRE LEITURAS
57 } // FIM DA FUNCAO leituraGPS
58
59 // =====
60 // Funcao de encerramento do daemon do GPS
61 // =====
62 void killGPS(gps_data_t* dataGPS){
63     gps_stream(dataGPS, WATCH_DISABLE, NULL);
64     gps_close (dataGPS);
65 } // FIM DA FUNCAO killGPS
66
67
```

- Arquivo .h

```
1  /* =====
2  * Header da implementacao do GPS
3  * =====
4  * O presente arquivo de header tem o proposito de modularizar o codigo com
5  * as funcoes que implementam o GPS.
6  *
7  * 1_ A funcao initGPS inicializa o daemon do GPS e retorna uma struct que
8  * sera usada para guardar os dados do GPS. Por se tratar de uma variavel
9  * local, e necessario que na main haja uma declaracao explicita de uma
10 * struct igual. Outras funcoes que necessitem destes dados ou devem
11 * receber uma copia dos dados ou receber a struct completa.
12 *
13 * 2_ A funcao leituraGPS realiza a leitura dos dados adquiridos pelo GPS
14 * (armazenados em um buffer pelo daemon) e e responsavel pelo tempo de
15 * espera entre leituras. E importante lembrar que o modulo GPS esta
16 * conectado na porta UART da raspberry e, portanto, possui comportamento
17 * assincrono. Isso possibilita a realizacao de leituras do GPS a qualquer
18 * momento.
19 * =====
20 */
21
22 #ifndef _IMPLEMENTACAOGPS_H_
23 #define _IMPLEMENTACAOGPS_H_
24
25 #ifdef __cplusplus
26     extern "C" {
27 #endif
28
29 #include <gps.h>
30 #include <math.h>
31 #include <stdio.h>
32 #include <stdlib.h>
33 #include <unistd.h>
34
35 #ifndef __WIRING_PI_H__
36     #include <wiringPi.h>
37 #endif
38
39 #ifndef _BUZZER_H_
40     #include "buzzer.h"
41 #endif
42
43 #ifndef BUZZER_PIN
44     #define BUZZER_PIN 7
45 #endif
46
47 void initGPS(gps_data_t* dataGPS);
48
49 void leituraGPS(gps_data_t* dataGPS);
50
51 void killGPS(gps_data_t* dataGPS);
52
53 #ifdef __cplusplus
54     }
55 #endif
56
57 #endif // _IMPLEMENTACAOGPS_H_
58
59
```

## B. Módulo de implementação dos sensores IMU

- Arquivo .c

```
1  /* =====
2  * Pacote de implementacao das funcoes dos IMUs
3  * =====
4  * O makefile deve ser construido com cuidado. De preferencia seguir o
5  * exemplo de makefile do RTIMULibDriver no caminho:
6  * RTIMULib/Linux/RTIMULibDrive/Makefile
7  * =====
8  */
9
10 #include "implementacaoIMU.h"
11
12 // =====
13 // Funcao de inicializacao dos IMU
14 // =====
15 RTIMU* initIMU(int i){
16     // Inicializa os setting das i unidades IMU
17     RTIMUSettings* settings;
18     // Declaracao redundante do imu. Pode ser retirado para otimizacao
19     RTIMU* imu;
20
21     /* Os arquivos .ini possuem as configuracoes de inicializacao das unidades.
22     * Como a configuracao dos dois MPU6050 e igual, eles possuem o mesmo .ini
23     * com a unica diferenca sendo o endereco I2C no qual eles estao conectados.
24     * O arquivo .ini para o MPU9250 e distinto pois ele sera utilizado com o
25     * protocolo SPI.
26     */
27     switch (i) {
28     case 0:
29         // Referencia a pasta e o nome do arquivo que possui o .ini
30         settings = new RTIMUSettings("/home/pi/Embarcados/3_Trabalho/Code/finalPROG/initFiles /
MPU6050_0", "RTIMULib");
31
32         // Cria o objeto i do IMU
33         imu = RTIMU::createIMU(settings);
34
35         if((imu == NULL) || (imu->IMUType() == RTIMU_TYPE_NULL)){
36             printf("Unidade IMU [%d] nao identificado na porta I2C 0x68.\n", i);
37             exit(1);
38         }
39
40         imu->IMUInit();
41         imu->setSlerpPower(0.02);
42         imu->setGyroEnable(true);
43         imu->setAccelEnable(true);
44         imu->setCompassEnable(false);
45         buzzerTone('C', 300);
46         buzzerTone('X', 100);
47         break;
48     case 1:
49         settings = new RTIMUSettings("/home/pi/Embarcados/3_Trabalho/Code/finalPROG/initFiles /
MPU6050_1", "RTIMULib");
50
51         imu = RTIMU::createIMU(settings);
52
53         if((imu == NULL) || (imu->IMUType() == RTIMU_TYPE_NULL)){
54             printf("Unidade IMU [%d] nao identificado na porta I2C 0x69.\n", i);
55             exit(1);
56         }
57
58         imu->IMUInit();
59         imu->setSlerpPower(0.02);
60         imu->setGyroEnable(true);
61         imu->setAccelEnable(true);
62         imu->setCompassEnable(false);
63         buzzerTone('D', 300);
64         buzzerTone('X', 100);
65         break;
66     case 2:
67         settings = new RTIMUSettings("/home/pi/Embarcados/3_Trabalho/Code/finalPROG/initFiles /
MPU9250_2", "RTIMULib");
68
69         imu = RTIMU::createIMU(settings);
```



```

70
71     if((imu == NULL) || (imu->IMUType() == RTIMU_TYPE_NULL)){
72         printf("Unidade IMU [%d] nao identificado na porta SPI.\n", i);
73         exit(1);
74     }
75
76     imu->IMUInit();
77     imu->setSlerpPower(0.02);
78     imu->setGyroEnable(true);
79     imu->setAccelEnable(true);
80     imu->setCompassEnable(true);
81     buzzerTone('A', 600);
82     buzzerTone('X', 100);
83     break;
84 default:
85     printf("Nenhuma unidade IMU identificada!\n");
86     exit(1);
87     break;
88 } /* FIM DO SWITCH CASE DO BUZZER */
89 return imu;
90 } /* FIM initIMU */
91
92 // =====
93 // Funcao de leitura dos sensores
94 // =====
95 void leituraIMU(RTIMU* imu, imuDataAngulo* imu_struct){
96     // Se for necessario retornar tambem a taxa de amostragem, referir a:
97     // RTIMULib/Linux/RTIMULibDrive/RTIMULibDrive.cpp
98     RTIMU_DATA imuData;
99
100     // Realiza o polling na taxa recomendada
101     usleep(imu->IMUPollInterval() * 1000);
102
103     while(imu->IMURead()){
104         imuData = imu->getIMUData();
105         imu_struct->roll = (imuData.fusionPose.x() * RTMATH_RAD_TO_DEGREE);
106         imu_struct->pitch = (imuData.fusionPose.y() * RTMATH_RAD_TO_DEGREE);
107         imu_struct->yaw = (imuData.fusionPose.z() * RTMATH_RAD_TO_DEGREE);
108     } // FIM DO WHILE
109 } // FIM DA FUNCAO leituraIMU
110
111

```

- Arquivo .h

```

1  /* =====
2  * Header da implementacao dos IMUs
3  * =====
4  * O presente arquivo de header tem o proposito de modularizar o codigo com
5  * as funcoes que implementam o IMU.
6  *
7  * Observe que este pacote de implementacoes e o arquivo principal com a
8  * funcao main nao possuem o extern "C" uma vez que estao usando fortemente
9  * os recursos do C++.
10 *
11 * initIMU recebe um inteiro referente ao numero do IMU sendo inicializado
12 * e retorna um ponteiro para classe RTIMU. Este ponteiro sera utilizado nas
13 * outras funcoes de leitura e encerramento do IMU.
14 *
15 * =====
16 */
17
18 #ifndef _IMPLEMENTACAOIMU_H_
19 #define _IMPLEMENTACAOIMU_H_
20
21 #ifndef _BUZZER_H_
22     #include "buzzer.h"
23 #endif
24
25 #ifndef _RTIMULIB_H
26     #include "RTIMULib.h"
27 #endif
28
29 typedef struct imuDataAngulo{
30     double roll;

```

```
31     double pitch;
32     double yaw;
33 }imuDataAngulo;
34
35 RTIMU* initIMU(int i);
36
37 void leituraIMU(RTIMU* imu, imuDataAngulo* imu_struct);
38
39 #endif // _IMPLEMENTACAOIMU_H_
40
41
```

### C. Módulo de implementação do buzzer

- Arquivo .c

```
1 #include "buzzer.h"
2
3 void buzzerInit() {
4     softToneCreate(BUZZER_PIN);
5 }
6
7 void buzzerTone(char nota, int duracao) {
8     switch(nota) {
9         case 'C':
10             softToneWrite(BUZZER_PIN, 262); // C4
11             delay(duracao);
12             break;
13         case 'D':
14             softToneWrite(BUZZER_PIN, 294); // D4
15             delay(duracao);
16             break;
17         case 'E':
18             softToneWrite(BUZZER_PIN, 330); // E4
19             delay(duracao);
20             break;
21         case 'F':
22             softToneWrite(BUZZER_PIN, 349); // F4
23             delay(duracao);
24             break;
25         case 'G':
26             softToneWrite(BUZZER_PIN, 392); // G4
27             delay(duracao);
28             break;
29         case 'A':
30             softToneWrite(BUZZER_PIN, 440); // A4
31             delay(duracao);
32             break;
33         case 'B':
34             softToneWrite(BUZZER_PIN, 494); // B4
35             delay(duracao);
36             break;
37         case 'X':
38             softToneWrite(BUZZER_PIN, 0);
39             break;
40         default:
41             softToneWrite(BUZZER_PIN, 0);
42             break;
43     } /* FIM DO SWITCH CASE */
44 } /* FIM BUZZER TONE */
45
46
```

- Arquivo .h

```
1 #ifndef _BUZZER_H_
2 #define _BUZZER_H_
3
4
5 #ifndef __WIRING_PI_H__
6     #include <wiringPi.h>
7 #endif
8
9 #include <softTone.h>
10 #include <time.h>
11
12 #ifndef BUZZER_PIN
13     #define BUZZER_PIN 7
14 #endif
15
16 #ifdef __cplusplus
17     extern "C" {
18 #endif
19
20 void buzzerInit();
21 void buzzerTone(char nota, int duracao);
22
23 #ifdef __cplusplus
```

```
24     }
25 #endif
26
27 #endif // _BUZZER_H_
28
29
```



#### D. Módulo principal

```
1 /* =====
2  * SISTEMA DE AQUISICAO DE DADOS
3  * =====
4  * Universidade de Brasilia
5  * campus Gama
6  *
7  * Versao: rev 3.9
8  * Autor: Arthur Evangelista
9  * Matricula: 14/0016686
10 *
11 * =====
12 * Este codigo implementa um sistema de aquisicao de dados. Ele utiliza
13 * dois sensores MPU-6050, um em cada meia asa, um sensor MPU-9250, no
14 * centro da aeronave, um GPS tambem no centro da aeronave, um botao
15 * para inicializacao do sistema e um buzzer para sinalizar a condicao
16 * do sistema.
17 *
18 * (i) O buzzer apitara algumas vezes para sinalizar que o
19 * sistema e seus modulos foram inicializados com sucesso.
20 *
21 * (ii) Tocara uma pequena melodia para sinalizar que o
22 * sistema foi desligado. Neste momento, os dados serao salvos antes
23 * do codigo sinalizar a terminacao da presente sessao da raspberry.
24 *
25 * Para compilar este codigo, siga o esquematico, entre na pasta do
26 * projeto e digite os seguintes comandos:
27 *
28 * make -j4
29 * sudo make install
30 * sudo finalPROG
31 *
32 * =====
33 * Para rapida referencia, as structs sao enumeradas da seguinte forma:
34 *
35 * meiaAsaDireita = imu_struct[0]
36 * meiaAsaEsquerda = imu_struct[1]
37 * Aviao = imu_struct[2]
38 * =====
39 */
40
41 // =====
42 // INICIALIZACAO DAS BIBLIOTECAS EM COMUM
43 // =====
44 #include <stdio.h>
45 #include <stdlib.h>
46 #include <unistd.h>
47 #include <signal.h>
48 #include <string.h>
49 #include <pthread.h>
50 #include <sys/types.h>
51
52 #ifndef _IMPLEMENTACAOGPS_H_
53 #include "implementacaoGPS.h"
54 #endif
55
56 #ifndef __WIRING_PI_H__
57 #include <wiringPi.h>
58 #include <wiringPiI2C.h>
59 #endif
60
61 #ifndef _RTIMULIB_H
62 #include "RTIMULib.h"
63 #endif
64
65 #ifndef _IMPLEMENTACAOIMU_H_
66 #include "implementacaoIMU.h"
67 #endif
68
69 #ifndef _BUZZER_H_
70 #include "buzzer.h"
71 #endif
72
73 // =====
74 // define das constantes
```

```

75 // =====
76
77 // Pinos utilizados
78 #define CONTROL_BUTTON_PIN 20
79 #define ADDR_PIN 18
80
81 #ifndef BUZZER_PIN
82     #define BUZZER_PIN 7
83 #endif
84
85 // =====
86 // typedef das structs que armazenarao os dados
87 // =====
88 typedef struct bend{
89     double meiaAsaDireita;
90     double meiaAsaEsquerda;
91 }bend;
92
93 typedef struct torsion{
94     double meiaAsaDireita;
95     double meiaAsaEsquerda;
96 }torsion;
97
98 typedef struct aeronave{
99     // Dados aeroelasticos
100     bend anguloDeFlexao;
101     torsion anguloDeTorcao;
102     // Dados desempenho
103     float alpha;
104     float atitude;
105     float roll;
106     float pitch;
107     float yaw;
108     // Dados do GPS
109     float latitude; // graus
110     float longitude; // graus
111     float altitude; // metros
112     float velSubida; // metros/segundo
113     float velTerrestre; // metros/segundo
114     float timestamp; // segundos
115     int fixmode; // auto-explicativo
116 }aeronave;
117
118 // =====
119 // Variaveis Globais
120 // =====
121 // volatile e utilizado para evitar otimizacoes do compilador
122 // struct para TODOS os dados da aeronave
123 volatile aeronave uav;
124 // struct para uso do sinal de interrupcao
125 struct sigaction act;
126 union sigval value;
127 // Variavel global para uso do GPSD
128 gps_data_t* dataGPS;
129 // Chaves mutex para armazenamento dos dados
130 static pthread_mutex_t mutexGPS;
131 static pthread_mutex_t mutexUAV;
132 // Declaracao das threads a serem usadas
133 pthread_t pthreadGPS;
134 pthread_t processamentoDireita;
135 pthread_t processamentoEsquerda;
136
137 // =====
138 // Sub-rotina para tratamento do sinal de interrupcao
139 // =====
140 void trataSinal(int signum, siginfo_t* info, void* ptr){
141     system("clear");
142     fprintf(stderr, "Recebido o sinal de interrupcao [%d].\n", signum);
143     printf("\n%s\n", "Teste realizado com sucesso!");
144     // Encerramento dos processamentos
145     pthread_cancel(processamentoDireita);
146     pthread_cancel(processamentoEsquerda);
147     // Encerramento do GPS
148     pthread_cancel(pthreadGPS);
149     killGPS(dataGPS);

```

```

150 free(dataGPS);
151 // Destruindo chave mutex
152 pthread_mutex_destroy(&mutexGPS);
153 pthread_mutex_destroy(&mutexUAV);
154 // Toque do buzzer
155 buzzerTone('B',100);
156 buzzerTone('A',100);
157 buzzerTone('G',100);
158 buzzerTone('F',100);
159 buzzerTone('E',100);
160 buzzerTone('D',100);
161 buzzerTone('C',200);
162 buzzerTone('X',100);
163 sleep(1);
164 exit(EXIT_SUCCESS);
165 } // FIM DA SUBROTINA trataSinal
166
167 // =====
168 // thread para processamento dos dados na struct
169 // =====
170 void* procDadosDir(void* unused){
171     imuDataAngulo* imu_struct;
172     imu_struct = (imuDataAngulo*)malloc(sizeof(imuDataAngulo)*3);
173     imu_struct = (imuDataAngulo *) unused;
174
175     double pitch0 = imu_struct->pitch;
176     double roll0 = imu_struct->roll;
177
178     imu_struct++;
179     imu_struct++;
180
181     double pitch2 = imu_struct->pitch;
182     double roll2 = imu_struct->roll;
183
184     pthread_mutex_lock(&mutexUAV);
185     uav.anguloDeFlexao.meiaAsaDireita = copysign((pitch0 - std::abs(pitch2)), pitch0);
186     uav.anguloDeTorcao.meiaAsaDireita = copysign((roll0 - std::abs(roll2)), roll0);
187     pthread_mutex_unlock(&mutexUAV);
188
189     imu_struct--;
190     imu_struct--;
191
192     return NULL;
193 } // FIM DA THREAD procDadosEsqDir
194
195 void* procDadosEsq(void* unused){
196     imuDataAngulo* imu_struct;
197     imu_struct = (imuDataAngulo*)malloc(sizeof(imuDataAngulo)*3);
198     imu_struct = (imuDataAngulo *) unused;
199
200     imu_struct++;
201
202     double pitch1 = imu_struct->pitch;
203     double roll1 = imu_struct->roll;
204
205     imu_struct++;
206     double pitch2 = imu_struct->pitch;
207     double roll2 = imu_struct->roll;
208
209     pthread_mutex_lock(&mutexUAV);
210     uav.anguloDeFlexao.meiaAsaEsquerda = copysign((pitch1 - std::abs(pitch2)), pitch1);
211     uav.anguloDeTorcao.meiaAsaEsquerda = copysign((roll1 - std::abs(roll2)), roll1);
212     pthread_mutex_unlock(&mutexUAV);
213
214     imu_struct--;
215     imu_struct--;
216
217     return NULL;
218 } // FIM DA THREAD procDadosEsq
219
220 // =====
221 // thread para armazenamento dos dados
222 // =====
223 void fileHandler(){
224     /* Sub-rotina dedicada a apenas armazenar o valor enviado para thread

```

```

225 * em um arquivo. A chave MUTEX sera utilizada para que:
226 *
227 * i) Os proximos dados podem estar prontos antes que a thread tenha
228 * salvo os dados da iteracao anterior; e
229 * ii) A proxima thread pode ser chamada antes que a thread atual tenha
230 * salvo os dados.
231 *
232 * Em outras palavras, utilizaremos a chave MUTEX para evitar as condicoes
233 * de corrida critica citadas acima.
234 */
235
236 // Pra salvar o arquivo com um nome customizado toda vez que houver aquisicao
237 char buffer[100]; // Nome do arquivo
238 char scr[128]; // Para printar timestamp do GPS
239 snprintf(buffer, sizeof(char)*100, "/home/pi/Embarcados/3_Trabalho/Code/Resultados/dados_%s.txt", __TIME__);
240 FILE *fp = fopen(buffer, "a+");
241
242 if (fp == NULL){
243     // Se nao for possivel abrir o arquivo, EXIT_FAILURE
244     fprintf(stderr, "Nao foi possivel realizar a abertura do arquivo [dados_%s.txt] na linha # %d.\n",
245             __TIME__, __LINE__);
246     exit(EXIT_FAILURE);
247 }else{
248     // Caso tenha sido possivel realizar a abertura do arquivo, locka a chave
249     // MUTEX e armazena os dados no arquivo
250     pthread_mutex_lock(&mutexUAV);
251
252     // Apresentacao dos dados no terminal
253     system("clear");
254     fprintf(stderr, "Latitude: %f deg\n", uav.latitude);
255     fprintf(stderr, "Longitude: %f deg\n", uav.longitude);
256     fprintf(stderr, "Altitude: %f m\n", uav.altitude);
257     fprintf(stderr, "Velocidade Horizontal: %f m/s\n", uav.velTerrest);
258     fprintf(stderr, "Velocidade de Subida: %f m/s\n", uav.velSubida);
259     unix_to_iso8601(uav.timestamp, scr, sizeof(scr));
260     fprintf(stderr, "Tempo: %s\n\n", scr);
261     fprintf(stderr, "FIX MODE: %d\n\n", uav.fixmode);
262     fprintf(stderr, "Roll: %f deg\n", uav.roll);
263     fprintf(stderr, "Pitch: %f deg\n", uav.pitch);
264     fprintf(stderr, "Yaw: %f deg\n", uav.yaw);
265     fprintf(stderr, "Torcao meia asa Direita: %f\n", uav.anguloDeTorcao.meiaAsaDireita);
266     fprintf(stderr, "Torcao meia asa Esquerda: %f\n", uav.anguloDeTorcao.meiaAsaEsquerda);
267     fprintf(stderr, "Flexao meia asa Direita: %f\n", uav.anguloDeFlexao.meiaAsaDireita);
268     fprintf(stderr, "Flexao meia asa Esquerda: %f\n", uav.anguloDeFlexao.meiaAsaEsquerda);
269
270     fprintf(fp, "%f\t\t%f\t\t%f\t\t%f", uav.anguloDeFlexao.meiaAsaDireita, uav.anguloDeFlexao.meiaAsaEsquerda,
271             uav.anguloDeTorcao.meiaAsaDireita, uav.anguloDeTorcao.meiaAsaEsquerda);
272     fprintf(fp, "\t\t%f\t\t%f\t\t%f", uav.roll, uav.pitch, uav.yaw);
273     /* fprintf(fp, "\t\t%f\t\t%f\t\t%f\t\t%f", velocidade, posicaoX, posicaoY, posicaoZ); */
274     fprintf(fp, "\n");
275
276     pthread_mutex_unlock(&mutexUAV);
277 } // FIM DA CONDICIONAL IF-ELSE
278 fclose(fp);
279 } // FIM DA FUNCAO fileHandler
280
281 // =====
282 // Funcao de inicializacao do GPS
283 // =====
284 void* threadGPS(void* param){
285     /* Nesta thread, quando o GPS receber um 3D FIX, os dados irao ser
286     * utilizados para a Fusao dos dados com a unidade IMU no centro da
287     * aeronave. Ira ocorrer uma interrupcao que sera tratada pelo SIGUSR1
288     */
289
290     dataGPS = (gps_data_t*)malloc(sizeof(gps_data_t));
291     initGPS(dataGPS);
292
293     while (1) {
294         usleep(750000); // A cada 3/4 de segundo (1.33 Hz), verifica GPS
295         pthread_mutex_lock(&mutexGPS);
296         leituraGPS(dataGPS);
297         if ((dataGPS->status == STATUS_FIX) && (dataGPS->fix.mode == MODE_3D)){
298             // Salva os dados do dataGPS->fix.etc no uav para fusao

```

```

297 pthread_mutex_lock(&mutexUAV);
298 uav.latitude = dataGPS->fix.latitude;
299 uav.longitude = dataGPS->fix.longitude;
300 uav.altitude = dataGPS->fix.altitude;
301 uav.velTerrest = dataGPS->fix.speed;
302 uav.velSubida = dataGPS->fix.climb;
303 uav.timestamp = dataGPS->fix.time;
304 uav.fixmode = dataGPS->fix.mode;
305 pthread_mutex_unlock(&mutexUAV);
306 pthread_mutex_unlock(&mutexGPS);
307 } else {
308     pthread_mutex_unlock(&mutexGPS);
309 } // FIM DO IF-ELSE 3D FIX
310 } // FIM DO LOOP-INFINITO DA THREAD DO GPS
311
312 } // FIM DA THREAD threadGPS
313
314 // =====
315 // Funcao principal main
316 // =====
317 int main () {
318     // Ponteiro para classe imu
319     RTIMU *imu[3];
320
321     // Variavel para contar o IMU a ser lido/processado para manter a ordem
322     int contadorIMU = 0;
323
324     // imu_struct [0] e imu_struct++ [1] sao para MPU-6050
325     // imu_struct ++ ++ [2] para MPU-9250
326     imuDataAngulo* imu_struct;
327     imu_struct = (imuDataAngulo*)malloc(sizeof(imuDataAngulo)*3);
328
329     // set das flags para o sinal de interrupcao SIGINT = ctrl + c
330     act.sa_sigaction = trataSinal;
331     act.sa_flags = SA_SIGINFO; // info sobre o sinal
332
333     // Direcionamento para o devido tratamento dos sinais
334     sigaction(SIGINT, &act, NULL);
335
336     // Setup da lib wiringPi para uso do GPIO
337     wiringPiSetup();
338
339     // Setup do buzzer
340     buzzerInit();
341
342     // Botao de controle do loop infinito
343     pinMode(CONTROL_BUTTON_PIN, INPUT);
344     pinMode(ADDR_PIN, OUTPUT);
345     digitalWrite(ADDR_PIN, HIGH);
346
347     // =====
348     // INICIALIZACAO DOS SENSORES IMU
349     // =====
350     while (contadorIMU < 3) {
351         // Aqui cabe uma otimizacao fazendo initIMU(contadorIMU, imu)
352         // e alterando a implementacao do IMU
353         imu[contadorIMU] = initIMU(contadorIMU);
354         contadorIMU++;
355     }
356
357     // Zera o contador
358     contadorIMU = 0;
359
360     // =====
361     // INICIALIZACAO DO GPS
362     // =====
363     if(pthread_create(&pthreadGPS, NULL, &threadGPS, NULL) != 0){
364         fprintf(stderr, "Erro na inicializacao da thread do GPS na linha # %d\n", __LINE__);
365         exit(EXIT_FAILURE);
366     }
367
368     // Loop infinito
369     while(1){
370         // =====
371         // Laco de repeticao para leitura dos sensores

```



```

372 // =====
373 contadorIMU = 0;
374 while (contadorIMU < 3) {
375     leituraIMU(imu[contadorIMU], imu_struct);
376     if(contadorIMU == 2){
377         pthread_mutex_lock(&mutexUAV);
378         uav.roll = imu_struct->roll;
379         uav.pitch = imu_struct->pitch;
380         uav.yaw = imu_struct->yaw;
381         pthread_mutex_unlock(&mutexUAV);
382     }
383     contadorIMU++;
384     imu_struct++;
385 }
386 imu_struct--;
387 imu_struct--;
388 imu_struct--;
389
390 // =====
391 // thread para processamento dos dados na struct
392 // =====
393 if( pthread_create (&processamentoDireita, NULL, &procDadosDir, (void*) imu_struct) != 0){
394     fprintf(stderr, "Erro na inicializacao da thread procDadosDir na linha # %d\n", __LINE__);
395     exit(EXIT_FAILURE);
396 }
397 if( pthread_create (&processamentoEsquerda, NULL, &procDadosEsq, (void*) imu_struct) != 0){
398     fprintf(stderr, "Erro na inicializacao da thread procDadosEsq na linha # %d\n", __LINE__);
399     exit(EXIT_FAILURE);
400 }
401 pthread_join(processamentoDireita, NULL);
402 pthread_join(processamentoEsquerda, NULL);
403 // pthread_create (&threadKalmanFusion, NULL, &kalmanFusion, NULL);
404 // pthread_join(threadKalmanFusion, NULL);
405
406 // Chama funcao para guardar e printar os dados
407 fileHandler();
408 } // FIM DO LOOP INFINITO
409
410 // =====
411 // Pos-processamento dos dados (FFT) e plot dos graficos
412 // =====
413 // A ser implementado...
414
415 // Ao final de tudo, a propria main executa o SIGINT
416 sigqueue(getpid(), SIGINT, value);
417 return 0;
418 } // FIM DA FUNCAO MAIN
419
420

```

## E. Makefile de instalação

```
1 # =====
2 # SISTEMA DE AQUISICAO DE DADOS
3 # =====
4 # Universidade de Brasilia
5 # campus Gama
6 #
7 # Autor: Arthur Evangelista
8 # Matricula: 14/0016686
9 #
10 # Este makefile foi fortemente baseado no utilizado na
11 # RTIMULib para compilar os exemplos.
12 # =====
13 # Para executar o makefile digite no terminal:
14 #   - make -j4
15 #   - sudo make install
16 # Para iniciar o programa:
17 #   - sudo finalPROG
18 # =====
19
20 # =====
21 # Define de algumas macros
22 # =====
23
24 # Caminho das libs
25 IMUPATH = libs/RTIMULib
26 GPSPATH = libs/GPSLib
27 BUZZERPATH = libs/BUZZERLib
28
29 # Definicoes do compilador e etc
30 CC      = gcc -g
31 CXX     = g++ -g
32 DEFINES = -lpthread -lwiringPi -lm -lgps
33 CFLAGS  = -pipe -O2 -Wall -W $(DEFINES)
34 CXXFLAGS = -pipe -O2 -Wall -W $(DEFINES)
35 INCPATH = -I. -I$(IMUPATH) -I$(GPSPATH) -I$(BUZZERPATH)
36 LINK    = g++
37 LFLAGS  = -Wl,-O1
38 LIBS    = -L/usr/lib/arm-linux-gnueabi -l$(DEFINES)
39 COPY    = cp -f
40 COPY_FILE = $(COPY)
41 COPY_DIR  = $(COPY) -r
42 STRIP     = strip
43 INSTALL_FILE = install -m 644 -p
44 INSTALL_DIR  = $(COPY_DIR)
45 INSTALL_PROGRAM = install -m 755 -p
46 DEL_FILE     = rm -f
47 SYMLINK      = ln -f -s
48 DEL_DIR      = rmdir
49 MOVE        = mv -f
50 CHK_DIR_EXISTS = test -d
51 MKDIR       = mkdir -p
52
53 # Diretorio para output
54 OBJECTS_DIR = objects/
55 RESULT_DIR  = /home/pi/Resultados/
56
57 # Arquivos
58 DEPS = $(IMUPATH)/RTMath.h \
59        $(IMUPATH)/RTIMULib.h \
60        $(IMUPATH)/RTIMULibDefs.h \
61        $(IMUPATH)/RTIMUHal.h \
62        $(IMUPATH)/RTFusion.h \
63        $(IMUPATH)/RTFusionKalman4.h \
64        $(IMUPATH)/RTFusionRTQF.h \
65        $(IMUPATH)/RTIMUSettings.h \
66        $(IMUPATH)/RTIMUAccelCal.h \
67        $(IMUPATH)/RTIMUMagCal.h \
68        $(IMUPATH)/RTIMUCalDefs.h \
69        $(IMUPATH)/IMUDrivers/RTIMU.h \
70        $(IMUPATH)/IMUDrivers/RTIMUNull.h \
71        $(IMUPATH)/IMUDrivers/RTIMUMPU9150.h \
72        $(IMUPATH)/IMUDrivers/RTIMUMPU9250.h \
73        $(IMUPATH)/IMUDrivers/RTIMUGD20HM303D.h \
74        $(IMUPATH)/IMUDrivers/RTIMUGD20M303DLHC.h \
```

```

75 $(IMUPATH)/IMUDrivers/RTIMUGD20HM303DLHC.h \
76 $(IMUPATH)/IMUDrivers/RTIMULSM9DS0.h \
77 $(IMUPATH)/IMUDrivers/RTIMULSM9DS1.h \
78 $(IMUPATH)/IMUDrivers/RTIMUBMX055.h \
79 $(IMUPATH)/IMUDrivers/RTIMUBNO055.h \
80 $(IMUPATH)/IMUDrivers/RTPressure.h \
81 $(IMUPATH)/IMUDrivers/RTPressureBMP180.h \
82 $(IMUPATH)/IMUDrivers/RTPressureLPS25H.h \
83 $(IMUPATH)/IMUDrivers/RTPressureMS5611.h \
84 $(IMUPATH)/IMUDrivers/RTPressureMS5637.h \
85 $(IMUPATH)/implementacaoIMU.h \
86 $(GPSPATH)/implementacaoGPS.h \
87 $(BUZZERPATH)/buzzer.h
88
89 OBJECTS = objects/main.o \
90 objects/RTMath.o \
91 objects/RTIMUHal.o \
92 objects/RTFusion.o \
93 objects/RTFusionKalman4.o \
94 objects/RTFusionRTQF.o \
95 objects/RTIMUSettings.o \
96 objects/RTIMUAccelCal.o \
97 objects/RTIMUMagCal.o \
98 objects/RTIMU.o \
99 objects/RTIMUNull.o \
100 objects/RTIMUMPU9150.o \
101 objects/RTIMUMPU9250.o \
102 objects/RTIMUGD20HM303D.o \
103 objects/RTIMUGD20M303DLHC.o \
104 objects/RTIMUGD20HM303DLHC.o \
105 objects/RTIMULSM9DS0.o \
106 objects/RTIMULSM9DS1.o \
107 objects/RTIMUBMX055.o \
108 objects/RTIMUBNO055.o \
109 objects/RTPressure.o \
110 objects/RTPressureBMP180.o \
111 objects/RTPressureLPS25H.o \
112 objects/RTPressureMS5611.o \
113 objects/RTPressureMS5637.o \
114 objects/implementacaoIMU.o \
115 objects/implementacaoGPS.o \
116 objects/buzzer.o
117
118 MAKE_TARGET = finalPROG
119
120 STDIR = Output/
121 TARGET = Output/$(MAKE_TARGET)
122 # =====
123
124 # Regras
125 $(TARGET): $(OBJECTS)
126 @$(CHK_DIR_EXISTS) Output/ || $(MKDIR) Output/
127 @$(CHK_DIR_EXISTS) $(RESULT_DIR) || $(MKDIR) $(RESULT_DIR)
128 $(LINK) $(LFLAGS) -o $(TARGET) $(OBJECTS) $(LIBS)
129
130 clean:
131 -$(DEL_FILE) $(OBJECTS)
132 -$(DEL_FILE) *~ core *.core
133
134 # =====
135 # Compilar
136 # =====
137
138 # RTIMULib
139 $(OBJECTS_DIR)%.o : $(IMUPATH)/%.cpp $(DEPS)
140 @$(CHK_DIR_EXISTS) objects/ || $(MKDIR) objects/
141 $(CXX) -c -o $@ $< $(CFLAGS) $(INCPATH)
142
143 # Drivers dos IMUs
144 $(OBJECTS_DIR)%.o : $(IMUPATH)/IMUDrivers/%.cpp $(DEPS)
145 @$(CHK_DIR_EXISTS) objects/ || $(MKDIR) objects/
146 $(CXX) -c -o $@ $< $(CFLAGS) $(INCPATH)
147
148 # Lib do GPS
149 $(OBJECTS_DIR)%.o : $(GPSPATH)/%.cpp $(DEPS)

```

```

150 @$(CHK_DIR_EXISTS) objects/ || $(MKDIR) objects/
151 $(CXX) -c -o $$< $(CFLAGS) $(INCPATH)
152
153 # Lib do buzzer
154 $(OBJECTS_DIR)%.o : $(BUZZERPATH)/%.cpp $(DEPS)
155 @$(CHK_DIR_EXISTS) objects/ || $(MKDIR) objects/
156 $(CXX) -c -o $$< $(CFLAGS) $(INCPATH)
157
158 # Principal
159 $(OBJECTS_DIR)main.o : main.cpp $(DEPS)
160 @$(CHK_DIR_EXISTS) objects/ || $(MKDIR) objects/
161 $(CXX) -c -o $$< main.cpp $(CFLAGS) $(INCPATH)
162 # =====
163
164 # =====
165 # Instalar
166 # =====
167 install_target: FORCE
168 @$(CHK_DIR_EXISTS) $(INSTALL_ROOT)/usr/local/bin/ || $(MKDIR) $(INSTALL_ROOT)/usr/local/bin/
169 -$(INSTALL_PROGRAM) "Output/$(MAKE_TARGET)" "$(INSTALL_ROOT)/usr/local/bin/$(MAKE_TARGET)"
170 -$(STRIP) "$(INSTALL_ROOT)/usr/local/bin/$(MAKE_TARGET)"
171
172 uninstall_target: FORCE
173 -$(DEL_FILE) "$(INSTALL_ROOT)/usr/local/bin/$(MAKE_TARGET)"
174
175
176 install: install_target FORCE
177
178 uninstall: uninstall_target FORCE
179
180 FORCE:
181
182

```