**RELATÓRIO TÉCNICO**

**TÓPICOS AVANÇADOS 1 – EC020**

Henrique de Paiva Ribeiro

Lucas Reis

Matheus Campos

Samuel Pereira

Prof.º Evandro Luis Brandão Gomes

Santa Rita do Sapucaí, 18 de junho de 2018

1. **Objetivo do projeto**

Este projeto tem como objetivo monitorar algumas informações importantes a respeito dos aviões durante as viagens, tais como a temperatura e a inclinação, para garantir o máximo de segurança durante o voo.

1. **Descrição geral e funções do projeto**

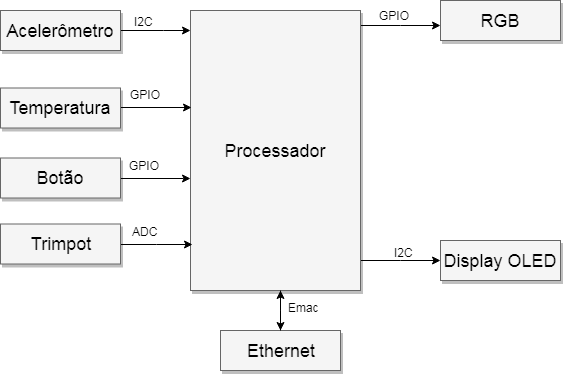
Os aviões são transportes de suma importância na sociedade atual.Visto isso, é necessário garantir o máximo de segurança possível durante os voos, pois qualquer erro pode levar a consequências catastróficas. Diante disso, foi proposto um sistema de monitoramento de informações importantes para o decorrer de uma viagem.

O projeto consiste em coletar informações de inclinação e temperatura do avião. Essas informações são exibidas em um display OLED local e também enviada para um cliente HTTP para que possa ocorrer exibição remota com uma interface amigável.

A temperatura do avião será constantemente monitorada por um sensor de temperatura, que será mostrada no display, como já dito. Além disso, a imagem do céu, ao fundo da figura de um avião, mudará de cor em uma página web que poderá ser acessada remotamente.

O ângulo de inclinação do avião será monitorado pelo acelerômetro, que além de mostrar as informações no display, irá alterar o ângulo da figura do avião na página web. Além disso, um LED RGB indicador poderá ser configurado por meio de um trimpot para mudar de cor de acordo com a inclinação do avião. Essa função do LED poderá ser ativada ou desativada por meio de um botão. Ele fica azul quando a ponta do avião está ligeiramente apontada para cima e vermelho quando a mesma está ligeiramente inclinada para baixo.

1. **Diagrama de blocos do hardware**



1. **Descrição dos sensores**

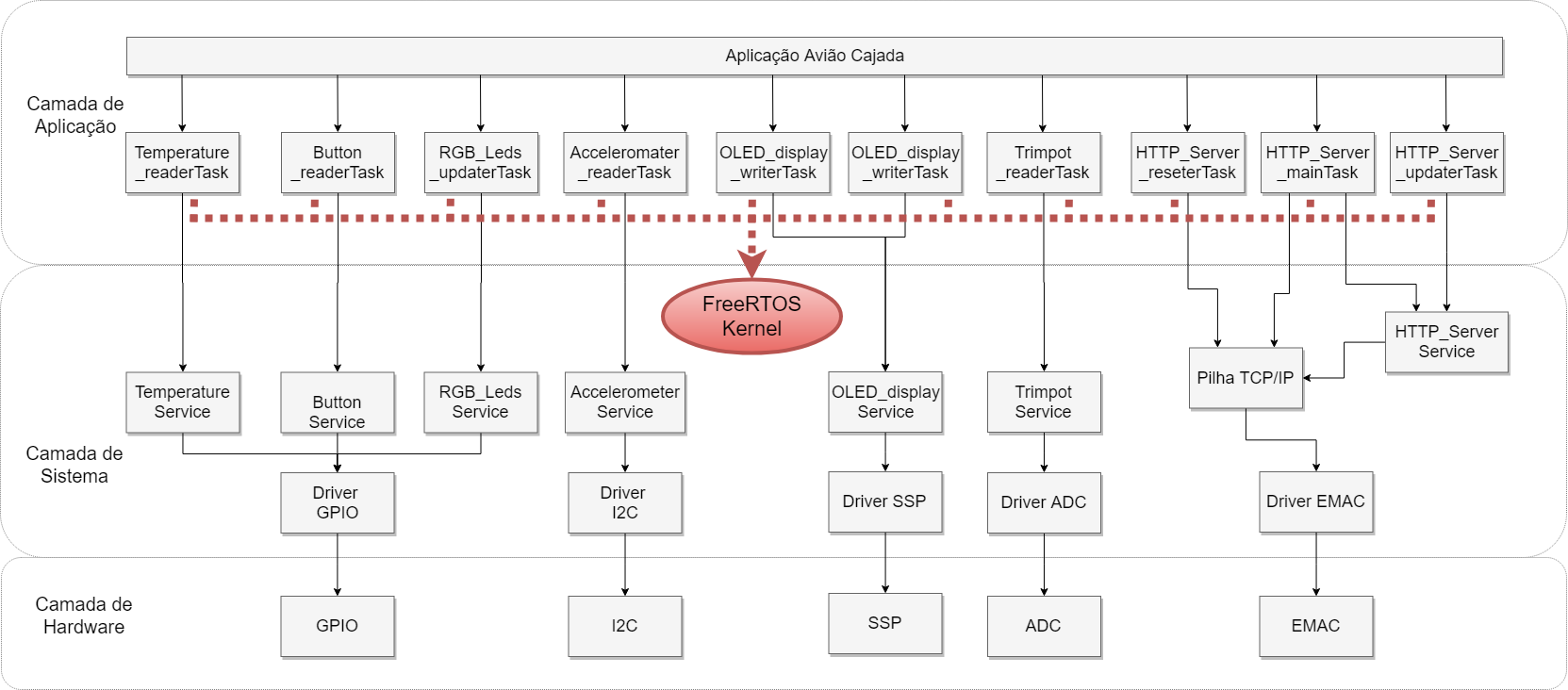
Sensor de temperatura: Possui uma saída PWM, logo sua leitura é feita através de um timer (Systick, nesse caso). Coleta informações a respeito da temperatura e comunica com o controlador por GPIO.

Acelerômetro: Capta informações da inclinação do avião e comunica com o controlador por I2C.

Trimpot: Regula o limite para operação dos leds RGB. Seu valor é lido através do conversor AD, cuja resolução de 14 bits.

Botão: Ativa e desativa o funcionamento do led RGB. Seu valor é acessado através de uma leitura GPIO.

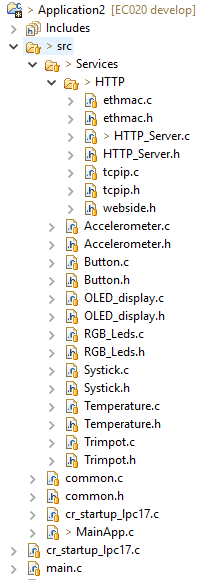
1. **Diagrama de arquitetura do software**



1. **Software implementado**

Todo o código do projeto pode ser acessado através do link <https://github.com/samuelpereira7/EC020>.

O projeto está organizado da seguinte maneira no workspace do ambiente de desenvolvimento:



Para o desenvolvimento do firmware, foram utilizadas as bibliotecas CMSISv2p00, Lib\_MCU e Lib\_EaBaseboard. Elas foram incluídas como projetos separados.

Segue abaixo o código da aplicação principal do firmware:

/\*

\* MainApp.c

\*

\* Created on: 01 de abr de 2018

\* Author: samuelpereira

\*/

/\* FreeRTOS.org includes. \*/

**#include** "FreeRTOS.h"

**#include** "task.h"

/\* Demo includes. \*/

**#include** "basic\_io.h"

**#include** "stdlib.h"

**#include** "stdio.h"

**#include** "string.h"

**#include** <Systick.h>

**#include** "Temperature.h"

**#include** "Button.h"

**#include** "Accelerometer.h"

**#include** "Trimpot.h"

**#include** "RGB\_Leds.h"

**#include** "OLED\_display.h"

**#include** "Services/HTTP/tcpip.h"

**#include** <HTTP\_Server.h>

**#include** "common.h"

**void** **acc\_callback**(message\_t msg);

**void** **trim\_callback**(message\_t msg);

**void** **but\_callback**(message\_t msg);

**void** **temp\_callback**(message\_t msg);

**void** **RGB\_callback**(message\_t msg);

**int** **main**( **void** )

{

/\* Init the semi-hosting. \*/

**printf**( "\n" );

Accelerometer\_init();

Accelerometer\_setCallback( acc\_callback );

Trimpot\_init();

Trimpot\_setCallback( trim\_callback );

Button\_init();

Button\_setCallback( but\_callback );

Temperature\_init( &getTicks );

Temperature\_setCallback( temp\_callback );

RGB\_Leds\_init();

RGB\_Leds\_setCallback( RGB\_callback );

OLED\_display\_init();

HTTP\_Server\_init();

/\* Start the scheduler so our tasks start executing. \*/

vTaskStartScheduler();

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

**for**( ;; );

**return** 0;

}

**void** **acc\_callback**(message\_t msg)

{

**if**(msg.source == *ACC*)

{

/\* Send message to HTTP Server \*/

**if** ( xQueueSendToBack( HTTP\_Server\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail acc");

}

/\* Send message to RGB Leds \*/

**if** ( xQueueSendToBack( RGB\_Leds\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail acc");

}

/\* Send message to OLED display \*/

**if** ( xQueueSendToBack( OLED\_display\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail acc");

}

}

}

**void** **trim\_callback**(message\_t msg)

{

**if**(msg.source == *TRIM*)

{

/\* Send message to RGB Leds \*/

**if** ( xQueueSendToBack( RGB\_Leds\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail trim");

}

}

}

**void** **but\_callback**(message\_t msg)

{

**if**(msg.source == *BUT*)

{

/\* Send message to RGB Leds \*/

**if** ( xQueueSendToBack( RGB\_Leds\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail button");

}

}

}

**void** **temp\_callback**(message\_t msg)

{

**if**(msg.source == *TEMP*)

{

/\* Send message to HTTP Server \*/

**if** ( xQueueSendToBack( HTTP\_Server\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail temp\n");

}

/\* Send message to OLED display \*/

**if** ( xQueueSendToBack( OLED\_display\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail temp\n");

}

}

}

**void** **RGB\_callback**(message\_t msg)

{

**if**(msg.source == *RGB*)

{

/\* Send message to OLED display \*/

**if** ( xQueueSendToBack( OLED\_display\_queue, &msg, 15) != pdTRUE )

{

vPrintString("fail RGB\n");

}

}

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationMallocFailedHook**( **void** )

{

/\* This function will only be called if an API call to create a task, queue

or semaphore fails because there is too little heap RAM remaining. \*/

**for**( ;; );

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationStackOverflowHook**( xTaskHandle \*pxTask, **signed** **char** \*pcTaskName )

{

/\* This function will only be called if a task overflows its stack. Note

that stack overflow checking does slow down the context switch

implementation. \*/

**for**( ;; );

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationIdleHook**( **void** )

{

/\* This example does not use the idle hook to perform any processing. \*/

}

/\*-----------------------------------------------------------\*/

**void** **vApplicationTickHook**( **void** )

{

/\* This example does not use the tick hook to perform any processing. \*/

}

Para troca de mensagens entre as tarefas, foi estabelecido um tipo padrão de mensagem, que se encontra no arquivo “common.h”.

**typedef** **enum** enuSource

{

*ACC* = 0,

*TRIM*,

*BUT*,

*TEMP*,

*RGB*

}source\_t;

**typedef** **struct** message

{

source\_t source;

int16\_t payload[3];

}message\_t;

**typedef** **void** (\*callback\_t)(message\_t msg);

Cada um dos serviços de sensores (Button, Trimpot, Accelerometer e Temperature) utilizam um callback para enviar dados para os módulos consumidores das informações geradas por esses serviços. Esse callback é configurado logo após a criação do serviço (função “init” de cada serviço) pela função “setCallback”. Para isso, deve ser definido um ponteiro de função em cada serviço. Segue abaixo a implementação desse ponteiro de callback e da função “setCallback” do acelerômetro.  
  
**static** callback\_t tx\_callback;

**void** **Accelerometer\_setCallback**(callback\_t c)

{

**if** (c != NULL)

{

tx\_callback = c;

}

}

O firmware atual possui 10 tarefas, sendo que a tarefa principal do servidor HTTP (“HTTP\_Server\_mainTask”) é a única que possui prioridade 2 (mais alta). Todas as demais possuem prioridade 1 (mais baixa). Segue abaixo uma breve descrição das tarefas:

* Accelerometer\_readerTask: responsável por fazer a leitura do acelerômetro a cada 300 milissegundos e enviar esses dados aos serviços interessados;
* Button\_readerTask: responsável por fazer a leitura do botão a cada 400 milissegundos e enviar esses dados aos serviços interessados;
* OLED\_display\_writerTask: responsável por atualizar o display OLED com os dados recebidos dos serviços inferiores a cada 1 segundo;
* OLED\_display\_updaterTask: responsável por atualizar os valores usados para escrita no display;
* RGB\_Leds\_updaterTask: responsável por ativar o LED necessário de acordo com os valores enviados pelo acelerômetro, botão e trimpot. Também envia uma mensagem ao display OLED referente ao estado seu estado (ativado ou desativado, que é definido pelo botão);
* Temperature\_readerTask: responsável por fazer a leitura da temperatura a cada 500 milissegundos e enviar esses dados aos serviços interessados;
* Trimpot\_readerTask: responsável por fazer a leitura do trimpot a cada 800 milissegundos e enviar esses dados aos serviços interessados;
* HTTP\_Server\_mainTask: responsável por verificar as requisições HTTP recebidas e tratá-las;
* HTTP\_Server\_reseterTask: responsável por reiniciar os parâmetros TCP a cada 20 segundos;
* HTTP\_Server\_updaterTask: responsável por atualizar os dados locais dos sensores.

Na função de inicialização de cada serviço (função “init”), as tarefas necessárias são criadas. Segue abaixo o código da função de inicialização do serviço de temperatura:

**void** **Temperature\_init** (uint32\_t (\*getMsTick)(**void**))

{

temp\_init(getMsTick);

xTaskCreate( Temperature\_readerTask, "Temp", 192, NULL, 1, NULL );

}

A seguir está o código da tarefa de leitura do trimpot como exemplo de geração de mensagens. As tarefas do acelerômetro, do sensor de temperatura e do botão seguem o mesmo princípio.

**void** **Trimpot\_readerTask**( **void** \*pvParameters )

{

message\_t msg;

**while**(1)

{

**memset**(&msg, 0x00, **sizeof**(msg));

msg.source = *TRIM*;

vPortEnterCritical();

msg.payload[0] = (int16\_t)Trimpot\_read();

vPortExitCritical();

**if** (tx\_callback != NULL)

{

tx\_callback(msg);

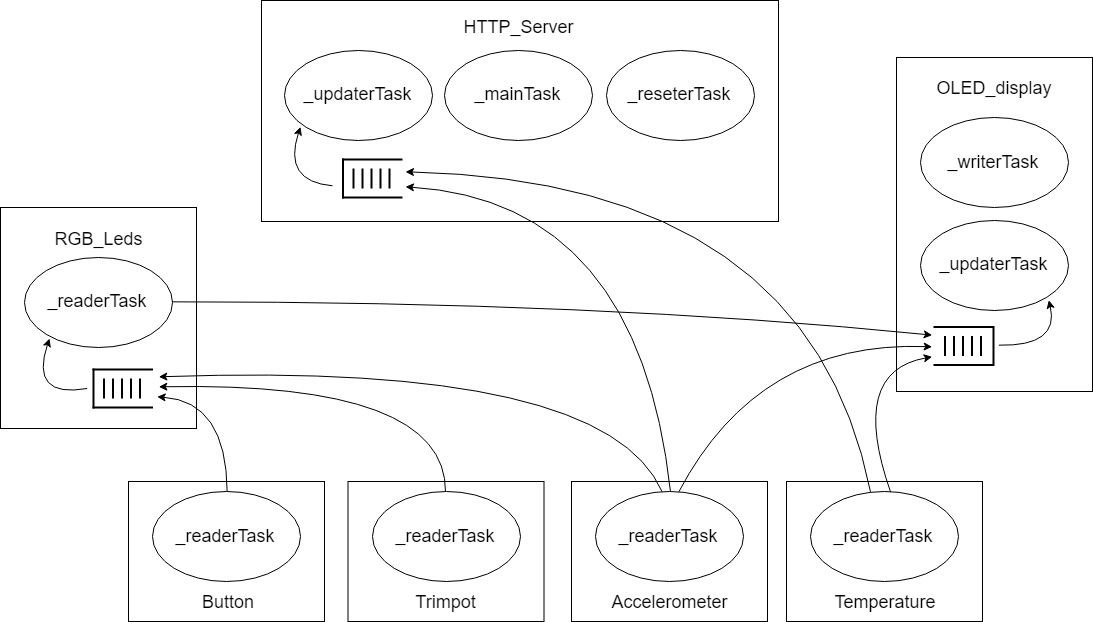
}

vTaskDelay( 800 / portTICK\_RATE\_MS );

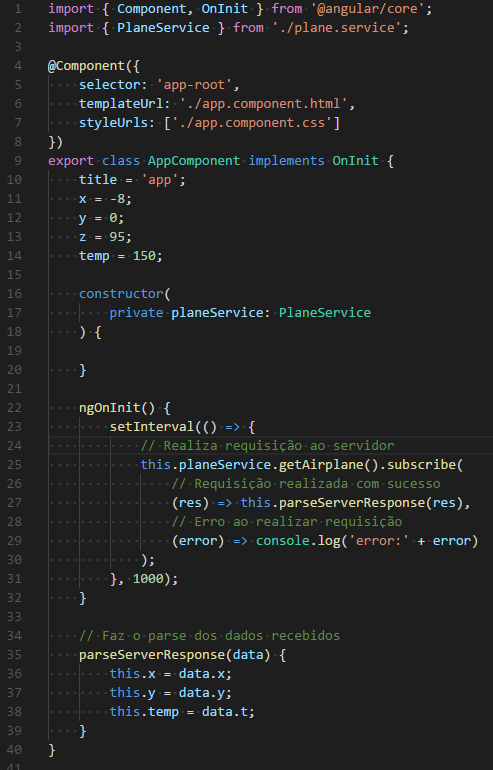
}

}

A relação entre as tarefas e serviços e o fluxo de mensagens entre essas tarefas podem ser visualizados na figura abaixo. Ainda, é possível verificar as quais as tarefas que publicam mensagens em cada uma das filas e quais tarefas consomem essas mensagens.



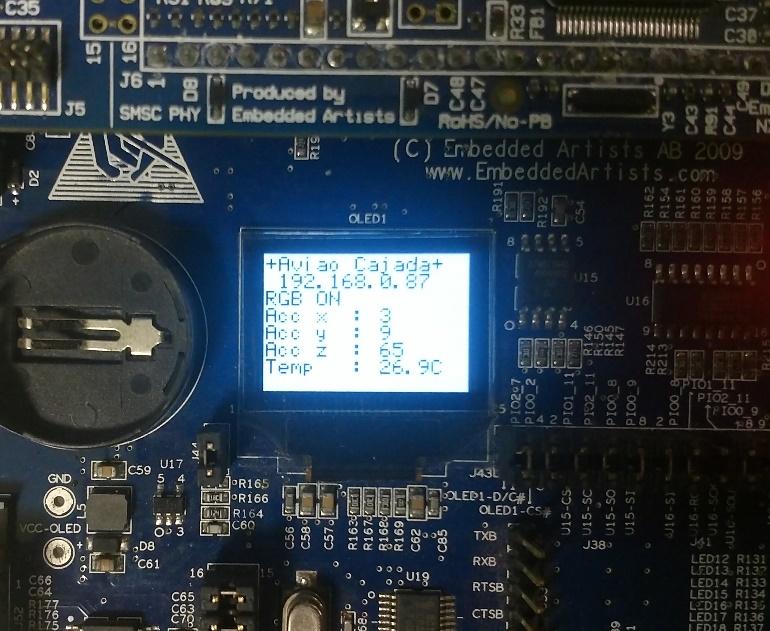
Para criação do cliente, foi utilizado o framework web Angular, segue abaixo código da aplicação principal do front-end, em que são realizadas as requisições para o servidor.



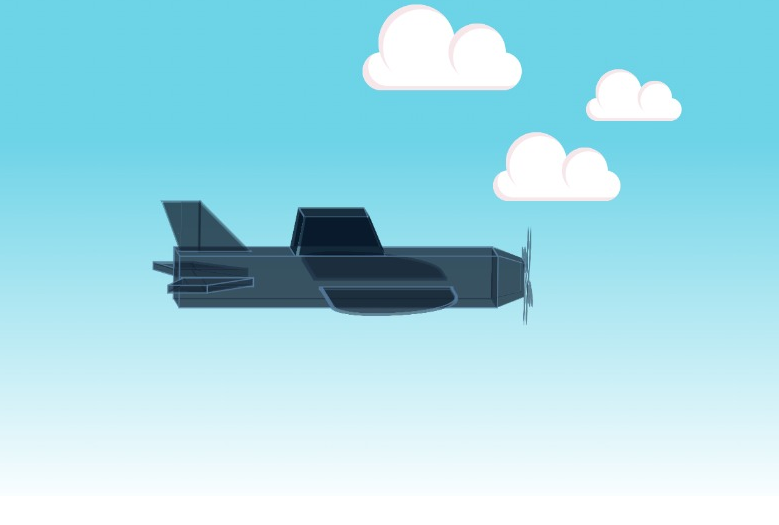
1. **Resultados obtidos**

Os resultados puderam ser analisados na própria baseboard e na aplicação que consome os dados enviados.

Segue abaixo uma fotografia do display OLED. É possível verificar as informações exibidas.

****

Agora, segue um screenshot do website.



1. **Conclusões**

Esse projeto nos proporcionou maior aprendizado sobre arquitetura ARM em geral e também sobre arquiteturas cliente/servidor. Também aprendemos a manipular bibliotecas e como os arquivos de um projeto de firmware se relacionam.

Ainda, tivemos a oportunidade de utilizar o FreeRTOS na prática. Isso proporcionou um grande aprendizado sobre sistemas operacionais de tempo real para sistemas embarcados.

Sobre o escopo do projeto, pode-se concluir que o controle de informações de aviões em tempo real é de extrema importância. Por isso, o projeto é viável não só de ser implementado, como também ser complementado, fazendo análise de ainda mais informações.