Informe Descriptivo del Proyecto SKYLF: Simulador de Vuelo Realista

Título:

Simulador de Vuelo Realista con Control de Variadores de Frecuencia y Panel de Instrumentos Físicos

Nombre del Proyecto y Año:

SKYFLY - 7° 2° Aviónica

Link a la Web:

https://estebanlautaro.github.io/Paginaweb-SKYFLY/

Link a Trello:

https://trello.com/b/u9Dn0bU8/kanban

Link a GitHub:

https://github.com/brizuu750/SKYLF

Link a Redes Sociales:

- Instagram: https://www.instagram.com/skylf.proyect/
- Tiktok: https://www.tiktok.com/@skyfly826?_t=8qnCGQhRKsA&_r=1

Presentación del Equipo:

Lauaro Eseban:

Mail: Lautyesteban14@gmail.com

Linkedin: Lautaro Sebastian Esteban

Instagram: @lauty.esteban

Santiago Rubio:

Mail: Santy201205@gmail.com

Linkedin: Santiago Rubio

Instagram: @santy_201205

Lucas Meabrio:

Mail: Meabriolucas@gmail.com

Linkedin: Lucas Meabrio

Instagram: @lucass_meab

Leandro Flores:

Mail: leandro200flores@gmail.com

Linkedin: Leandro Flores

Instagram: @lean.floresss

Agustin Brizuela:

Mail: brizuu750@gmail.com

Linkedin: Agustin Lionel Brizuela

Instagram: @agustiin.brizuela

Santiago Leiva:

Mail: Santiagoleiva745@gmail.com

Linkedin: Santiago Leiva

Instagram: @leivva.s

Emiliano Romocordoba:

Mail: Romoemiliano324@gmail.com

Linkedin: Emiliano Romo Cordoba

Jofiel Godoy Baldovino:

Mail: Markbaldj@gmail.com

Linkedin: Marco Baldovino

Instagram:@jofiel_godoy

Foto de cada integrante:

Santiago Rubio

Lucas Meabrio

Marco Godoy Baldovino

Lautaro Esteban:

Leandro Flores:

Agustin Brizuela:

Emiliano Romocordoba

Santiago Leiva

Foto grupal:

Horas dedicadas por cada integrante:

Participaron un total de 8 personas en un promedio de 14 hs semanales durante 28 semanas aproximadamente

Índice General

1. Introducción

Este proyecto de **Simulador de Vuelo Realista**, basado en la cabina del **Cessna 152**, replica las maniobras y controles de la aeronave mediante la integración de un panel físico de control y un sistema de visualización interactuando con **Flight Simulator 2020**. Se implementa un circuito de comunicación mediante un **Arduino UNO** y un **ESP32** que permiten gestionar diversos sistemas y enviar comandos a través de variables de comunicación específicas, sincronizando los instrumentos físicos con el simulador virtual. Además, el simulador utiliza un sistema de control de motores con variadores de frecuencia y sensor de movimiento **MPU6050** para añadir realismo en el control de vuelo.

2. Objetivo

Objetivo General:

Construir un simulador de vuelo con una interfaz de usuario realista, que permita experimentar el control de una aeronave tipo Cessna 152 mediante un panel físico conectado a Flight Simulator 2020.

Objetivo Específico:

Integrar un sistema de variadores de frecuencia, Arduino UNO y ESP32, junto con un panel de control que permita al usuario activar sistemas de vuelo, como el control de luces, bomba de combustible y buses aviónicos, replicando el entorno real de un avión en el simulador virtual.

3. Descripción de la Solución

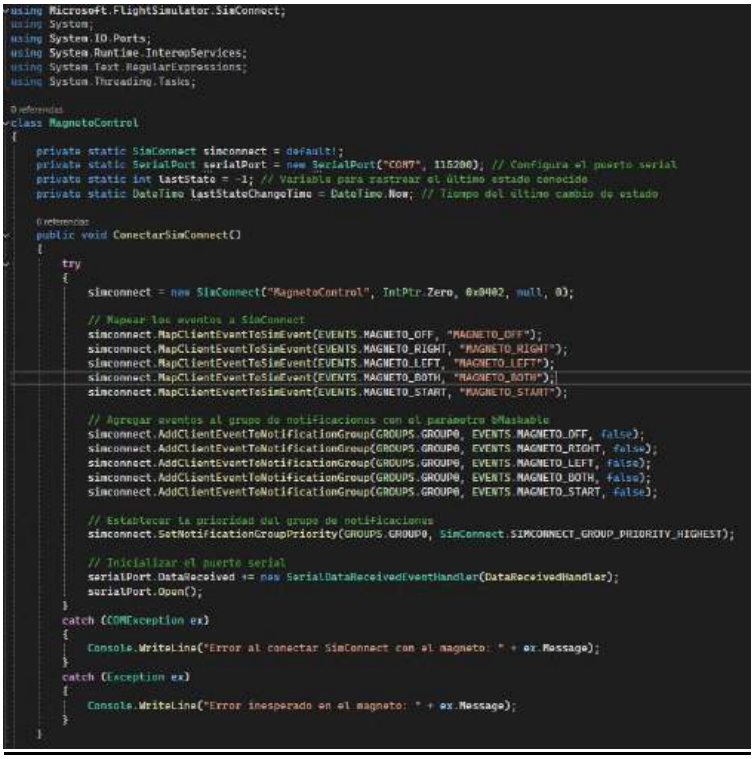
El simulador incluye dos sistemas principales: un **Panel de Control** y un **Panel de Visualización de Instrumentos**, con un esquema de comunicación que integra varios componentes para emular los controles de un avión real.

Panel de Control

El **Panel de Control** cuenta con 9 interruptores de palanca, 3 pulsadores, y una llave selectora rotativa de 6 estados. Este panel físico permite al usuario activar sistemas esenciales de la aeronave en el entorno virtual de **Flight Simulator 2020**, mediante la integración de **MobiFlight** y **Arduino UNO**. Algunos de los sistemas principales controlados por el panel incluyen:

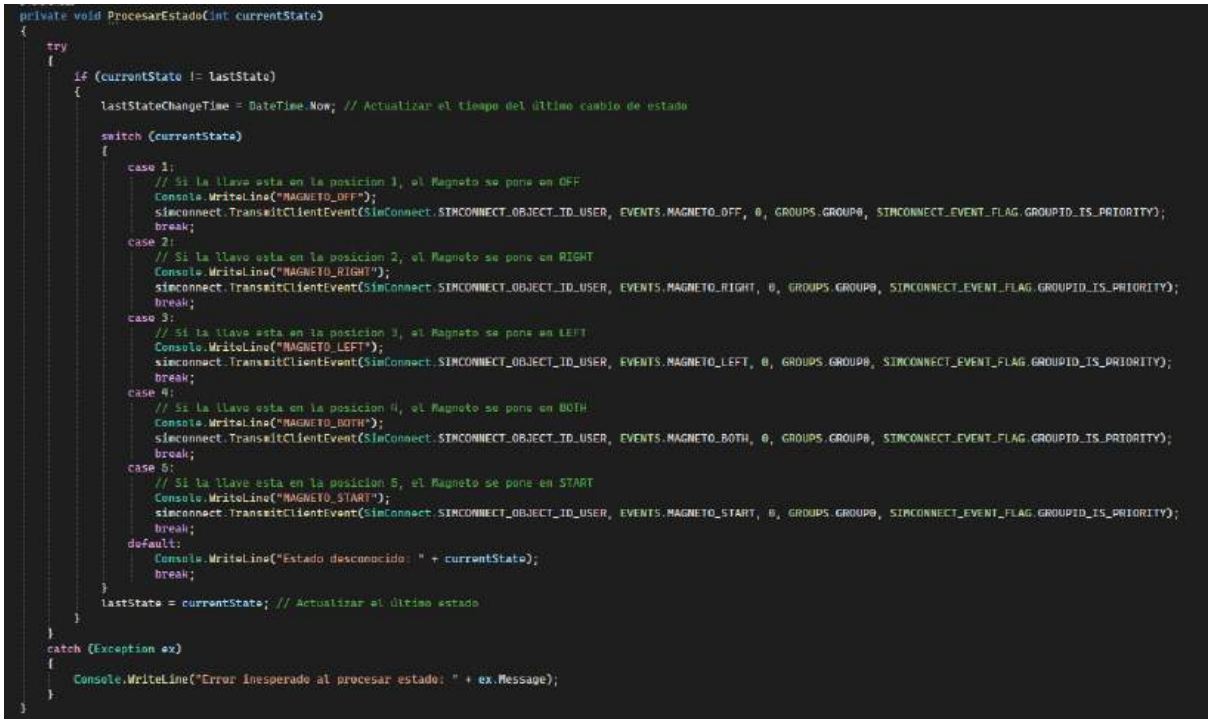
- **Bomba de Combustible**: Activa una bomba de combustible que permite el cebado inicial del motor.

- **Luces (BCN, LAND, TAXI, NAV, STRB)**: Controla las luces exteriores del avión para el despegue, aterrizaje y condiciones de vuelo específicas.
- **Interruptores de Bus Aviónico**: Activa los sistemas de navegación y comunicación de la aeronave.
- **Tubo Pitot**: Controla el calentador del tubo Pitot para evitar el congelamiento.
- **Primer**: Se encarga de inyectar combustible a los cilindros del motor antes de arrancarlo, facilitando el arranque en condiciones especialmente frías.
- **Magnetos**: Distribuyen corriente eléctrica a las bujías, gestionadas por una llave rotativa de 6 estados. Para comunicar la llave selectora tuvimos que hacer un código que se encargue de mandar los “Events id” para las posiciones del magneto dependiendo de la actual posición de la llave selectora ya que al usar Mobiflight, algunas posiciones son erróneas (se utilizan como una forma de interactuar entre diferentes sistemas del simulador), pero en el código las posiciones andan bien. Las partes más importantes del código son:



```
using Microsoft.FlightSimulator.SimConnect;
using System;
using System.IO.Ports;
using System.Runtime.InteropServices;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

0 referencias
class MagnetoControl
{
    private static SimConnect simconnect = default!;
    private static SerialPort serialPort = new SerialPort("COM7", 115200); // Configura el puerto serial
    private static int lastState = -1; // Variable para rastrear el último estado conocido
    private static DateTime lastStateChangeTime = DateTime.Now; // Tiempo del último cambio de estado

    0 referencias
    public void ConectarSimConnect()
    {
        try
        {
            simconnect = new SimConnect("MagnetoControl", IntPtr.Zero, 0x0402, null, 0);

            // Mapear los eventos a SimConnect
            simconnect.MapClientEventToSimEvent(EVENTS.MAGNETO_OFF, "MAGNETO_OFF");
            simconnect.MapClientEventToSimEvent(EVENTS.MAGNETO_RIGHT, "MAGNETO_RIGHT");
            simconnect.MapClientEventToSimEvent(EVENTS.MAGNETO_LEFT, "MAGNETO_LEFT");
            simconnect.MapClientEventToSimEvent(EVENTS.MAGNETO_BOTH, "MAGNETO_BOTH");
            simconnect.MapClientEventToSimEvent(EVENTS.MAGNETO_START, "MAGNETO_START");

            // Agregar eventos al grupo de notificaciones con el parámetro bMaskable
            simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP0, EVENTS.MAGNETO_OFF, false);
            simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP0, EVENTS.MAGNETO_RIGHT, false);
            simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP0, EVENTS.MAGNETO_LEFT, false);
            simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP0, EVENTS.MAGNETO_BOTH, false);
            simconnect.AddClientEventToNotificationGroup(GROUPS.GROUP0, EVENTS.MAGNETO_START, false);

            // Establecer la prioridad del grupo de notificaciones
            simconnect.SetNotificationGroupPriority(GROUPS.GROUP0, SimConnect.SIMCONNECT_GROUP_PRIORITY_HIGHEST);

            // Inicializar el puerto serial
            serialPort.DataReceived += new SerialDataReceivedEventHandler(DataReceivedHandler);
            serialPort.Open();
        }
        catch (COMException ex)
        {
            Console.WriteLine("Error al conectar SimConnect con el magneto: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error inesperado en el magneto: " + ex.Message);
        }
    }
```

Se ingresan las librerías necesarias y los events id que se van a usar:



```
private void ProcesarEstado(int currentState)
{
    try
    {
        if (currentState != lastState)
        {
            lastStateChangeTime = DateTime.Now; // Actualizar el tiempo del último cambio de estado

            switch (currentState)
            {
                case 1:
                    // Si la llave esta en la posicion 1, el Magneto se pone en OFF
                    Console.WriteLine("MAGNETO_OFF");
                    simconnect.TransmitClientEvent(SimConnect.SIMCONNECT_OBJECT_ID_USER, EVENTS.MAGNETO_OFF, 0, GROUPS.GROUP0, SIMCONNECT_EVENT_FLAG.GROUPID_IS_PRIORITY);
                    break;
                case 2:
                    // Si la llave esta en la posicion 2, el Magneto se pone en RIGHT
                    Console.WriteLine("MAGNETO_RIGHT");
                    simconnect.TransmitClientEvent(SimConnect.SIMCONNECT_OBJECT_ID_USER, EVENTS.MAGNETO_RIGHT, 0, GROUPS.GROUP0, SIMCONNECT_EVENT_FLAG.GROUPID_IS_PRIORITY);
                    break;
                case 3:
                    // Si la llave esta en la posicion 3, el Magneto se pone en LEFT
                    Console.WriteLine("MAGNETO_LEFT");
                    simconnect.TransmitClientEvent(SimConnect.SIMCONNECT_OBJECT_ID_USER, EVENTS.MAGNETO_LEFT, 0, GROUPS.GROUP0, SIMCONNECT_EVENT_FLAG.GROUPID_IS_PRIORITY);
                    break;
                case 4:
                    // Si la llave esta en la posicion 4, el Magneto se pone en BOTH
                    Console.WriteLine("MAGNETO_BOTH");
                    simconnect.TransmitClientEvent(SimConnect.SIMCONNECT_OBJECT_ID_USER, EVENTS.MAGNETO_BOTH, 0, GROUPS.GROUP0, SIMCONNECT_EVENT_FLAG.GROUPID_IS_PRIORITY);
                    break;
                case 5:
                    // Si la llave esta en la posicion 5, el Magneto se pone en START
                    Console.WriteLine("MAGNETO_START");
                    simconnect.TransmitClientEvent(SimConnect.SIMCONNECT_OBJECT_ID_USER, EVENTS.MAGNETO_START, 0, GROUPS.GROUP0, SIMCONNECT_EVENT_FLAG.GROUPID_IS_PRIORITY);
                    break;
                default:
                    Console.WriteLine("Estado desconocido: " + currentState);
                    break;
            }
            lastState = currentState; // Actualizar al último estado
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error inesperado al procesar estado: " + ex.Message);
    }
}
```

El código enviá estos events id al propio Flight Simulator dependiendo de la posición de la llave selectora

El **Arduino UNO** es el microcontrolador principal del panel de control y se configura mediante **MobiFlight** para ejecutar comandos específicos en Flight Simulator 2020 según los interruptores activados. Esto asegura que cada acción en el panel tenga un reflejo preciso en el simulador virtual, mejorando la inmersión en el entorno de vuelo.

Paso a Paso de configuración de palancas para vincularlas con MobiFlight y el FS2020:

1. Conectamos el Arduino UNO al MobiFlight y el FSUIPC7

2. Asignamos un Pin del 2 al 13 del Arduino UNO a uno de los terminales de la palanca (En nuestro caso lo pusimos al pin 8 pero puede ser cualquiera)

3. Seleccionar en MobiFlight la pestaña de "MobiFlight Modules"

4. Para no tener inconvenientes actualizar el firmware así el MobiFlight detecte correctamente el la Arduino UNO

5. En la barra inferior ir a la pestaña de "Add Device" para agregar un dispositivo que en nuestro caso es una palanca.

6. Luego le asignamos a la palanca el nombre con el que queramos que se identifique, siendo en este caso "Tubo Pitot" para la función de la palanca que es sobre el Calentador del Tubo Pitot

7. Poner en la "Configuración de Pin" el pin que le asignamos a la palanca (en nuestro caso el Pin 8 de la Arduino UNO)

8. En el Menú principal crear una configuración de entrada en el "Input Configs"

9. Para editar la configuración selecciona a la derecha de la fila en "Edit"

10. Luego seleccionar los dispositivos conectados al MobiFlight

11. Seleccionar en el Action Type el "FSUIPC - Offset" para luego asignarle la acción al dispositivo que hayamos conectado (en nuestro caso la palanca)

12. Dentro de la configuración de entrada se encuentran las configuraciones para cuando se activa la palanca "On Press" y cuando está del lado contrario "On Release"

13. Para la configuración del Offset nos fijamos en el manual de Offset del FSUIPC que está en internet para buscar la acción de la palanca, y nos dice cuál código de Offset tiene que ir.

14. En nuestro caso buscamos el del calentador del tubo pitot. Por lo que ponemos el que nos dice "029C"

15. Para que el Calentador del tubo pitot del Cessna 152 se active en simultáneo a la palanca que conectamos al FS2020, le asignamos "1" como valor de seteo.

16. En "On Release" de igual forma poner el mismo código de Offset pero que cuando se baje la palanca que el calentador se apague ponemos "0" como valor de seteo

17. Luego de esto al probar la palanca que vinculamos a dispositivo que sea (en nuestro caso el calentador del tubo pitot), se tendría que activar simultáneamente

Comunicación con el Variador de Frecuencia y Sistema de Control de Movimiento

Para lograr una experiencia de vuelo más realista, el simulador utiliza un **variador de frecuencia GK500-2T2.2B** controlado por un ESP32 y un potenciómetro digital para gestionar el movimiento de los motores, lo que permite simular inclinaciones en los ejes de cabeceo y alabeo. La configuración del variador de frecuencia y el sistema de comunicación están organizados de la siguiente forma:

- **ESP32 y Potenciometro digital**: El ESP32 envía comandos al potenciómetro digital a través de los pines de control (UP, DOWN, CS) para ajustar su resistencia.El potenciómetro digital modifica los valores eléctricos que afectan directamente el comportamiento de los variadores de frecuencia, lo que impacta en el movimiento de los motores trifásicos y, en consecuencia, en la inclinación de la cabina del simulador.

- **Configuración de Pines**:

    - Conexión del MPU6050:

- VCC del MPU6050 a 3.3V del ESP32.
- GND del MPU6050 a GND del ESP32.
- SDA del MPU6050 a GPIO21 del ESP32.
- SCL del MPU6050 a GPIO22 del ESP32.

- Conexión de los Potenciómetros Digitales

a. Para el primer potenciómetro:
- CS_PIN a GPIO15 del ESP32.
- INC_PIN a GPIO14 del ESP32.
- UP_DOWN_PIN a GPIO13 del ESP32.
- VW va al GND del variador 1
- VH y VL van a dos pines del variador 1
- VCC del potenciómetro a la entrada analógica del variador 1 (AI1).
- GND del potenciómetro a GND común del variador 1

b. Para el segundo potenciómetro:
- CS_PIN a GPIO32 del ESP32.
- INC_PIN a GPIO33 del ESP32.
- UP_DOWN_PIN a GPIO25 del ESP32.
- VW va al GND del variador 2
- VH y VL van a dos pines del variador 2
- VCC del potenciómetro a la entrada analógica del variador 2 (AI1).
- GND del potenciómetro a GND común del variador 2.

- Conexión de los optoacopladores y resistencias:

a. Para el primer optoacoplador:
- Anodo del led va a la primera resistencia de 330 Ω y luego al GPIO27 del ESP32
- Cátodo del led va a GND del esp32
- Colector va conectado a el puerto x1 del variador 1 y también conectado a la primera resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optoacoplador va a COM del variador 1 y también va a GND de la fuente

b. Para el segundo optoacoplador:
- Anodo del led va a la segunda resistencia de 330 Ω y luego al GPIO12 del ESP32
- Cátodo del led va a GND del esp32
- Colector va conectado a el puerto x2 del variador 1 y también está conectado a la segunda resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optoacoplador va a COM del variador 1 y también va a GND de la fuente

c. Para el tercer optoacoplador:

- Anodo del led va la tercera resistencia de 330 Ω y luego al GPIO18 del ESP32
- Cátodo del led va a GND del esp32
- Colector va conectado a el puerto x1 del variador 2 y también está conectado a la tercera resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optoacoplador va a COM del variador 2 y también va a GND de la fuente

d. Para el cuarto optoacoplador:

- Anodo del led va la cuarta resistencia de 330 Ω y luego al GPIO19 del ESP32
- Cátodo del led va a GND del esp32
- Colector va conectado a el puerto x2 del variador 2 y también está conectado a la cuarta resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optoacoplador va a COM del variador 2 y también va a GND de la fuente

3. **Sensor de Movimiento MPU6050**: Integrado al ESP32, el MPU6050 detecta la inclinación y aceleración en tiempo real, lo que permite ajustar los movimientos del simulador para reflejar las maniobras de vuelo de manera más precisa. Este sensor es crucial para sincronizar el sistema físico con los cambios en el entorno virtual del Flight Simulator 2020.

Panel de Visualización de Instrumentos

Este panel de visualización está diseñado para mostrar los datos de vuelo en una pantalla de 18 pulgadas, reflejando los instrumentos de la aeronave. La comunicación de los datos de vuelo se realizaría mediante **Air Manager o Pop Out Panel Manager**, que gestionan las variables de Flight Simulator 2020. Este diseño permite una representación en tiempo real de los parámetros del simulador, ajustando el diseño para que se asemeje al de un Cessna 152.

Flight Simulator 2020

Utilizamos Microsoft Flight Simulator - 40th Anniversary Edition, compatible con herramientas externas de comunicación que extraen variables en tiempo real, como Simvar Watcher.

Simvar Watcher y Extracción de Variables

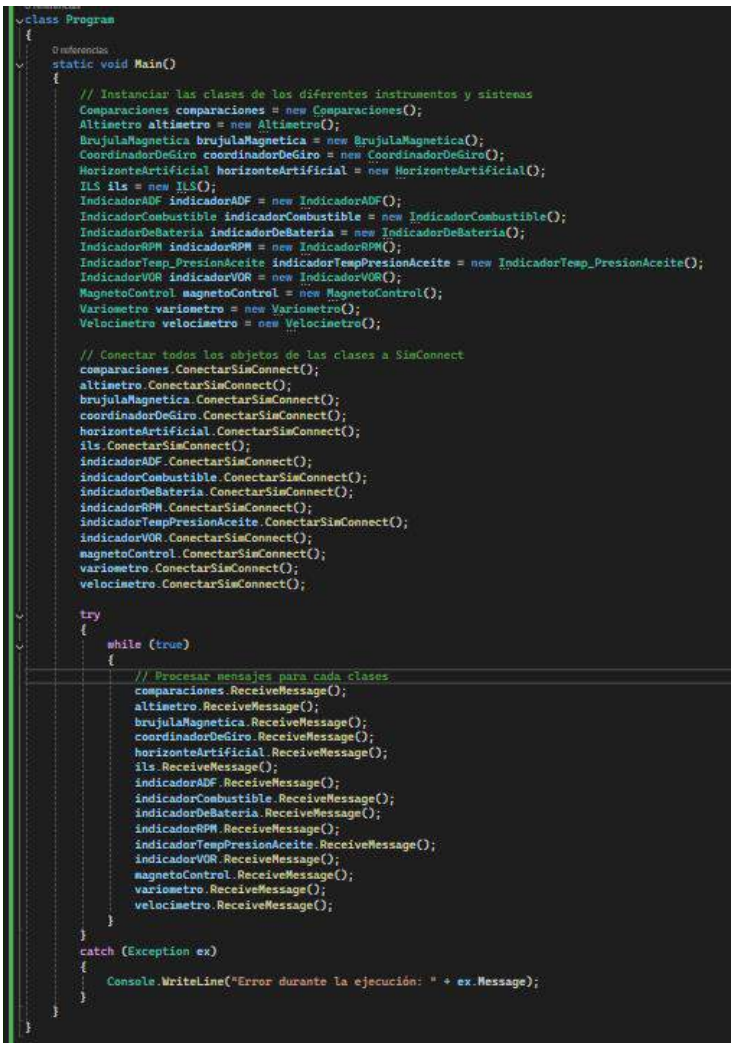
El Simvar Watcher facilita la lectura de variables seleccionadas del simulador, configurado con las librerías SimConnect y otros archivos DLL y CFG, permitiendo acceder a las lecturas en tiempo real. Las variables se agrupan para la lectura de datos de instrumentos (altímetro, velocímetro, etc.) y control de movimiento.

Para la integración de Simvar Watcher con Flight Simulator, utilizamos códigos en C# en Visual Studio 2022. Estos códigos gestionan la conexión y extracción de

variables, que se dividen según los instrumentos que controlan (altímetro, brújula, velocímetro, etc) y un código para los movimientos y aceleraciones del avión.

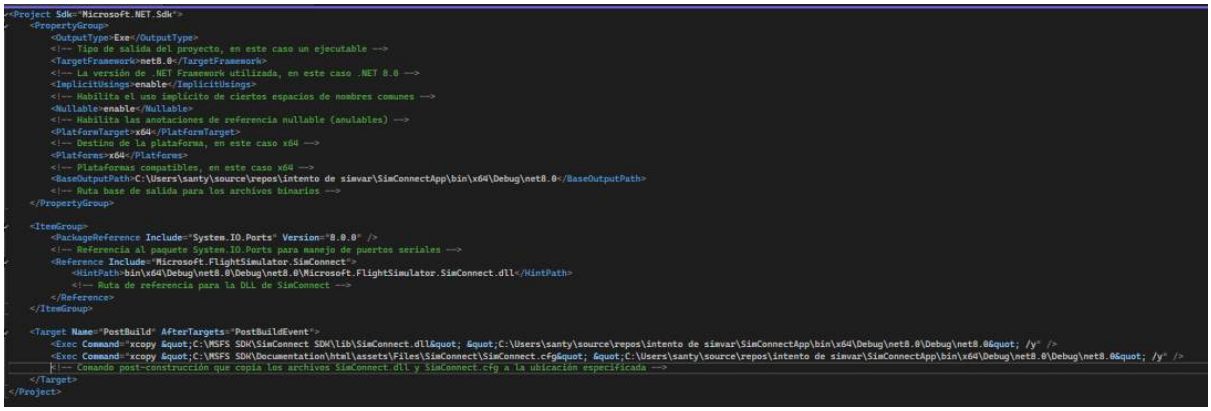
Códigos Generales:

- **Program.cs:** Este código configura y gestiona la comunicación entre el simulador de vuelo y múltiples sistemas de instrumentos y controles, permitiendo que cada uno reciba y procese datos en tiempo real.



```
class Program
{
    0 referencias
    static void Main()
    {
        // Instanciar las clases de los diferentes instrumentos y sistemas
        Comparaciones comparaciones = new Comparaciones();
        Altimetro altimetro = new Altimetro();
        BrujulaMagnetica brujulaMagnetica = new BrujulaMagnetica();
        CoordinadorDeGiro coordinadorDeGiro = new CoordinadorDeGiro();
        HorizonteArtificial horizonteArtificial = new HorizonteArtificial();
        ILS ils = new ILS();
        IndicadorADF indicadorADF = new IndicadorADF();
        IndicadorCombustible indicadorCombustible = new IndicadorCombustible();
        IndicadorDeBateria indicadorDeBateria = new IndicadorDeBateria();
        IndicadorRPM indicadorRPM = new IndicadorRPM();
        IndicadorTemp_PresionAceite indicadorTempPresionAceite = new IndicadorTemp_PresionAceite();
        IndicadorVOR indicadorVOR = new IndicadorVOR();
        MagnetoControl magnetoControl = new MagnetoControl();
        Variometro variometro = new Variometro();
        Velocimetro velocimetro = new Velocimetro();

        // Conectar todos los objetos de las clases a SimConnect
        comparaciones.ConectarSimConnect();
        altimetro.ConectarSimConnect();
        brujulaMagnetica.ConectarSimConnect();
        coordinadorDeGiro.ConectarSimConnect();
        horizonteArtificial.ConectarSimConnect();
        ils.ConectarSimConnect();
        indicadorADF.ConectarSimConnect();
        indicadorCombustible.ConectarSimConnect();
        indicadorDeBateria.ConectarSimConnect();
        indicadorRPM.ConectarSimConnect();
        indicadorTempPresionAceite.ConectarSimConnect();
        indicadorVOR.ConectarSimConnect();
        magnetoControl.ConectarSimConnect();
        variometro.ConectarSimConnect();
        velocimetro.ConectarSimConnect();

        try
        {
            while (true)
            {
                // Procesar mensajes para cada clases
                comparaciones.ReceiveMessage();
                altimetro.ReceiveMessage();
                brujulaMagnetica.ReceiveMessage();
                coordinadorDeGiro.ReceiveMessage();
                horizonteArtificial.ReceiveMessage();
                ils.ReceiveMessage();
                indicadorADF.ReceiveMessage();
                indicadorCombustible.ReceiveMessage();
                indicadorDeBateria.ReceiveMessage();
                indicadorRPM.ReceiveMessage();
                indicadorTempPresionAceite.ReceiveMessage();
                indicadorVOR.ReceiveMessage();
                magnetoControl.ReceiveMessage();
                variometro.ReceiveMessage();
                velocimetro.ReceiveMessage();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error durante la ejecución: " + ex.Message);
        }
    }
}
```

- **SimConnectApp.csproj:** Este código configura un proyecto de .NET 8.0 que genera un ejecutable (Exe) para la plataforma x64. Define referencias y dependencias necesarias, incluyendo `System.IO.Ports` para manejar la comunicación por puertos seriales y `SimConnect` para interactuar con Microsoft Flight Simulator. También, especifica la ruta de salida para los binarios compilados y define un objetivo post-construcción que copia los archivos `SimConnect.dll` y `SimConnect.cfg` necesarios a la ubicación de salida del proyecto.



Códigos de Instrumentos:

Cada instrumento está asignado a un código específico que se comunica con el simulador. Los siguientes están hechos para ser comunicados con una aplicación externa, como el Air Manager en caso de los instrumentos analógicos, y en caso de que se desee añadir alguno digital, se pueden encontrar variantes a este como el pop out manager. Además, son los encargados de la sincronización con mobilight, por lo que permite que funcionen tanto el panel de instrumentos como el de control. Por ejemplo:

Altímetro.cs: Extrae la altura en tiempo real.

Variómetro.cs: Controla la velocidad vertical del avión.

Velocímetro.cs: Muestra la velocidad de vuelo.

BrujulaMagnetica.cs: Muestra la dirección cardinal (N, S, E, O) en la que se encuentra el avión.

CoordinadorDeGiro.cs: Indica la tasa de giro del avión y ayuda a mantener un giro coordinado.

HorizonteArtificial.cs: Representa la orientación del avión respecto al horizonte real.

ILS.cs: Muestra las indicaciones del sistema de aterrizaje por instrumentos (Instrument Landing System).

IndicadorADF.cs: Indica la dirección hacia una estación de radio NDB (Non-Directional Beacon).

IndicadorCombustible.cs: Monitorea el nivel de combustible del avión.

IndicadorDeBateria.cs: Mide y muestra el estado de la batería del avión.

IndicadorRPM.cs: Mide y muestra las revoluciones por minuto del motor.

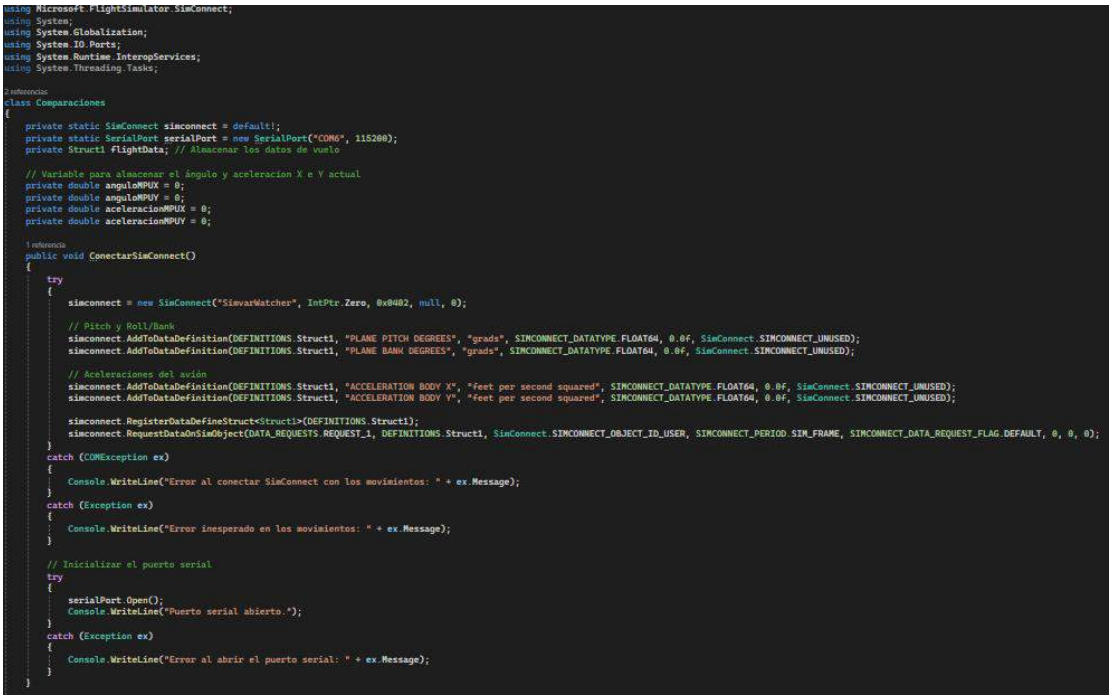
IndicadorTemp_PresionAceite.cs: Monitorea la temperatura y presión del aceite del motor.

IndicadorVOR.cs: Indica la dirección hacia una estación VOR (VHF Omnidirectional Range)

Código para movimientos y aceleraciones del avión (Comparaciones.cs):

El código se encargará de comunicarse con ciertas variables para el pitch, bank y las aceleraciones (excepto el eje Z, ya que no hay motor que se mueva en ese eje). También se encarga de recibir los datos del mpu6050 y de transformar el giro X e Y en el ángulo donde estaría el mpu en cada momento, y de transformar las aceleraciones de fuerza G a ft/s^2. Luego el codigo compara de 4 formas diferentes los valores de las variables con los valores del mpu6050 y, dependiendo de la diferencia entre los valores, el código se encarga de mandar diferentes comandos (como MOVER_IZQUIERDA y MOVER_ABAJO) al ESP32 para que luego este se los envíe a los variadores de frecuencia.

Las partes más importantes del código son:



```
using Microsoft.FlightSimulator.SimConnect;
using System;
using System.Globalization;
using System.IO.Ports;
using System.Runtime.InteropServices;
using System.Threading.Tasks;

2 referencias
class Comparaciones
{
    private static SimConnect simconnect = default!;
    private static SerialPort serialPort = new SerialPort("COM6", 115200);
    private Struct1 flightData; // Almacenar los datos de vuelo

    // Variable para almacenar el ángulo y aceleracion X e Y actual
    private double anguloMPUX = 0;
    private double anguloMPUY = 0;
    private double aceleracionMPUX = 0;
    private double aceleracionMPUY = 0;

    1 referencia
    public void ConectarSimConnect()
    {
        try
        {
            simconnect = new SimConnect("SimvarWatcher", IntPtr.Zero, 0x0402, null, 0);

            // Pitch y Roll/Bank
            simconnect.AddToDataDefinition(DEFINITIONS.Struct1, "PLANE PITCH DEGREES", "grads", SIMCONNECT_DATATYPE.FLOAT64, 0.0f, SimConnect.SIMCONNECT_UNUSED);
            simconnect.AddToDataDefinition(DEFINITIONS.Struct1, "PLANE BANK DEGREES", "grads", SIMCONNECT_DATATYPE.FLOAT64, 0.0f, SimConnect.SIMCONNECT_UNUSED);

            // Aceleraciones del avión
            simconnect.AddToDataDefinition(DEFINITIONS.Struct1, "ACCELERATION BODY X", "feet per second squared", SIMCONNECT_DATATYPE.FLOAT64, 0.0f, SimConnect.SIMCONNECT_UNUSED);
            simconnect.AddToDataDefinition(DEFINITIONS.Struct1, "ACCELERATION BODY Y", "feet per second squared", SIMCONNECT_DATATYPE.FLOAT64, 0.0f, SimConnect.SIMCONNECT_UNUSED);

            simconnect.RegisterDataDefineStruct<Struct1>(DEFINITIONS.Struct1);
            simconnect.RequestDataOnSimObject(DATA_REQUESTS.REQUEST_1, DEFINITIONS.Struct1, SimConnect.SIMCONNECT_OBJECT_ID_USER, SIMCONNECT_PERIOD.SIM_FRAME, SIMCONNECT_DATA_REQUEST_FLAG.DEFAULT, 0, 0, 0);
        }
        catch (COMException ex)
        {
            Console.WriteLine("Error al conectar SimConnect con los movimientos: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error inesperado en los movimientos: " + ex.Message);
        }

        // Inicializar el puerto serial
        try
        {
            serialPort.Open();
            Console.WriteLine("Puerto serial abierto.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error al abrir el puerto serial: " + ex.Message);
        }
    }
```

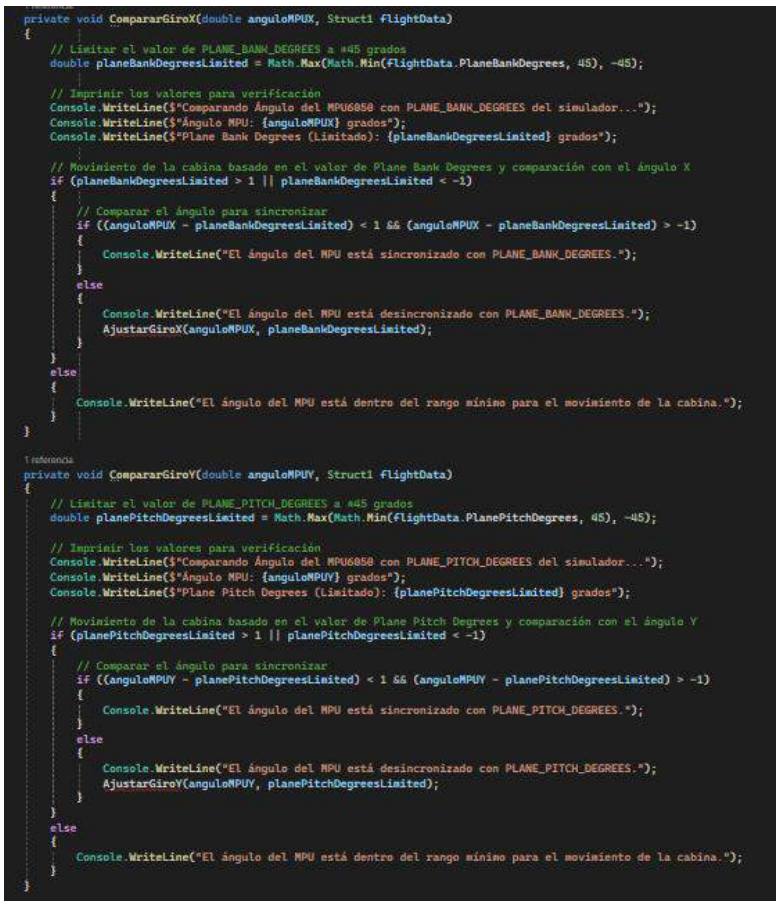
Se incluye las librerías necesarias y se extrae las variables del pitch, bank y las aceleraciones

```
// Método para actualizar el ángulo basado en la lectura del giroscopio
1 referencia
private void ActualizarAnguloX(double giroX, double deltaTime)
{
    // Convertir la velocidad angular de radianes a grados
    double giroXGrados = giroX * (180.0 / Math.PI);

    // Integrar la velocidad angular para obtener el ángulo X en grados
    anguloMPUX += giroXGrados * deltaTime;
}

1 referencia
private void ActualizarAnguloY(double giroY, double deltaTime)
{
    // Convertir la velocidad angular de radianes a grados
    double giroYGrados = giroY * (180.0 / Math.PI);

    // Integrar la velocidad angular para obtener el ángulo Y en grados
    anguloMPUY += giroYGrados * deltaTime;
}

1 referencia
private void ActualizarAceleracionX(double aceleracionX)
{
    // Convertir de g a ft/s^2
    aceleracionMPUX = aceleracionX * 9.81 * 3.28084;
}

1 referencia
private void ActualizarAceleracionY(double aceleracionY)
{
    // Convertir de g a ft/s^2
    aceleracionMPUY = aceleracionY * 9.81 * 3.28084;
}
```

El giro X e Y se transforma en ángulos y las aceleraciones en ft/s^2

```
public void ReadFromMPU()
{
    try
    {
        if (serialPort.IsOpen)
        {
            try
            {
                string dataFromArduino = serialPort.ReadLine();
                string[] dataParts = dataFromArduino.Split(',');

                if (dataParts.Length == 4)
                {
                    double aceleracionX = double.Parse(dataParts[0].Split(':')[1], CultureInfo.InvariantCulture);
                    double aceleracionY = double.Parse(dataParts[1].Split(':')[1], CultureInfo.InvariantCulture);
                    double giroX = double.Parse(dataParts[2].Split(':')[1], CultureInfo.InvariantCulture);
                    double giroY = double.Parse(dataParts[3].Split(':')[1], CultureInfo.InvariantCulture);

                    // Definir deltaTime constante
                    double deltaTime = 0.02;

                    // Actualizar datos del MPU
                    ActualizarAnguloX(giroX, deltaTime);
                    ActualizarAnguloY(giroY, deltaTime);
                    ActualizarAceleracionX(aceleracionX);
                    ActualizarAceleracionY(aceleracionY);

                    // Imprimir el ángulo actual del MPU en grados
                    Console.WriteLine($"Ángulo en el eje X del MPU (grads): {anguloMPUX}°");
                    Console.WriteLine($"Ángulo en el eje Y del MPU (grads): {anguloMPUY}°");
                    Console.WriteLine($"Aceleracion en el eje X del MPU (ft/s2): {aceleracionMPUX}°");
                    Console.WriteLine($"Aceleracion en el eje Y del MPU (ft/s2): {aceleracionMPUY}°");

                    // Llamar a los métodos de comparación (debes pasar flightData como parámetro si es necesario)
                    CompararGiroX(anguloMPUX, flightData);
                    CompararGiroY(anguloMPUY, flightData);
                    CompararAcelX(aceleracionMPUX, flightData);
                    CompararAcelY(aceleracionMPUY, flightData);
                }
            }
            catch (TimeoutException)
            {
                Console.WriteLine("Error: La lectura del puerto serial ha excedido el tiempo de espera.");
            }
            catch (IOException ioEx)
            {
                Console.WriteLine("Error de E/S: " + ioEx.Message);
            }
            catch (Exception ex)
            {
                Console.WriteLine("Error inesperado: " + ex.Message);
            }
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("Error al leer datos del MPU6050: " + ex.Message);
    }
}
```

Esta función se encarga de recibir los datos del mpu6050 y llamar a los distintos metodos de comparacion



```
private void CompararGiroX(double anguloMPUX, Struct1 flightData)
{
    // Limitar el valor de PLANE_BANK_DEGREES a ±45 grados
    double planeBankDegreesLimited = Math.Max(Math.Min(flightData.PlaneBankDegrees, 45), -45);

    // Imprimir los valores para verificación
    Console.WriteLine($"Comparando Ángulo del MPU6050 con PLANE_BANK_DEGREES del simulador...");
    Console.WriteLine($"Ángulo MPU: {anguloMPUX} grados");
    Console.WriteLine($"Plane Bank Degrees (Limitado): {planeBankDegreesLimited} grados");

    // Movimiento de la cabina basado en el valor de Plane Bank Degrees y comparación con el ángulo X
    if (planeBankDegreesLimited > 1 || planeBankDegreesLimited < -1)
    {
        // Comparar el ángulo para sincronizar
        if ((anguloMPUX - planeBankDegreesLimited) < 1 && (anguloMPUX - planeBankDegreesLimited) > -1)
        {
            Console.WriteLine("El ángulo del MPU está sincronizado con PLANE_BANK_DEGREES.");
        }
        else
        {
            Console.WriteLine("El ángulo del MPU está desincronizado con PLANE_BANK_DEGREES.");
            AjustarGiroX(anguloMPUX, planeBankDegreesLimited);
        }
    }
    else
    {
        Console.WriteLine("El ángulo del MPU está dentro del rango mínimo para el movimiento de la cabina.");
    }
}

1 referencia
private void CompararGiroY(double anguloMPUY, Struct1 flightData)
{
    // Limitar el valor de PLANE_PITCH_DEGREES a ±45 grados
    double planePitchDegreesLimited = Math.Max(Math.Min(flightData.PlanePitchDegrees, 45), -45);

    // Imprimir los valores para verificación
    Console.WriteLine($"Comparando Ángulo del MPU6050 con PLANE_PITCH_DEGREES del simulador...");
    Console.WriteLine($"Ángulo MPU: {anguloMPUY} grados");
    Console.WriteLine($"Plane Pitch Degrees (Limitado): {planePitchDegreesLimited} grados");

    // Movimiento de la cabina basado en el valor de Plane Pitch Degrees y comparación con el ángulo Y
    if (planePitchDegreesLimited > 1 || planePitchDegreesLimited < -1)
    {
        // Comparar el ángulo para sincronizar
        if ((anguloMPUY - planePitchDegreesLimited) < 1 && (anguloMPUY - planePitchDegreesLimited) > -1)
        {
            Console.WriteLine("El ángulo del MPU está sincronizado con PLANE_PITCH_DEGREES.");
        }
        else
        {
            Console.WriteLine("El ángulo del MPU está desincronizado con PLANE_PITCH_DEGREES.");
            AjustarGiroY(anguloMPUY, planePitchDegreesLimited);
        }
    }
    else
    {
        Console.WriteLine("El ángulo del MPU está dentro del rango mínimo para el movimiento de la cabina.");
    }
}
```

En estas funciones se compara los valores del ángulo X e Y del mpu y de las variables del pitch y bank para obtener la diferencia entre ellos.

```
private void CompararAcelX(double aceleracionMPUX, Struct1 flightData)
{
    // Limitar el valor de ACCELERATION_BODY_X a ±10 ft/s²
    double accelerationBodyXLimited = Math.Max(Math.Min(flightData.AccelerationBodyX, 10), -10);

    // Imprimir los valores para verificación
    Console.WriteLine($"Comparando Aceleración del MPU6050 con ACCELERATION_BODY_X del simulador...");
    Console.WriteLine($"Aceleración MPU: {aceleracionMPUX} ft/s²");
    Console.WriteLine($"Acceleration Body X (Limitado): {accelerationBodyXLimited} ft/s²");

    // Movimiento de la cabina basado en el valor de Acceleration Body X y comparación con la aceleración X
    if (accelerationBodyXLimited > 0.5 || accelerationBodyXLimited < -0.5)
    {
        // Comparar la aceleración para sincronizar
        if ((aceleracionMPUX - accelerationBodyXLimited) < 0.5 && (aceleracionMPUX - accelerationBodyXLimited) > -0.5)
        {
            Console.WriteLine("La aceleración del MPU está sincronizada con ACCELERATION_BODY_X.");
        }
        else
        {
            Console.WriteLine("La aceleración del MPU está desincronizada con ACCELERATION_BODY_X.");
            AjustarAceleracionX(aceleracionMPUX, accelerationBodyXLimited);
        }
    }
    else
    {
        Console.WriteLine("La aceleración del MPU está dentro del rango mínimo para el movimiento de la cabina.");
    }
}

1 referencia
private void CompararAcelY(double aceleracionMPUY, Struct1 flightData)
{
// Limitar el valor de ACCELERATION_BODY_Y a ±10 ft/s²
double accelerationBodyYLimited = Math.Max(Math.Min(flightData.AccelerationBodyY, 10), -10);

// Imprimir los valores para verificación
Console.WriteLine($"Comparando Aceleración del MPU6050 con ACCELERATION_BODY_Y del simulador...");
Console.WriteLine($"Aceleración MPU: {aceleracionMPUY} ft/s²");
Console.WriteLine($"Acceleration Body Y (Limitado): {accelerationBodyYLimited} ft/s²");

// Movimiento de la cabina basado en el valor de Acceleration Body Y y comparación con la aceleración Y
if (accelerationBodyYLimited > 0.5 || accelerationBodyYLimited < -0.5)
{
    // Comparar la aceleración para sincronizar
    if ((aceleracionMPUY - accelerationBodyYLimited) < 0.5 && (aceleracionMPUY - accelerationBodyYLimited) > -0.5)
    {
        Console.WriteLine("La aceleración del MPU está sincronizada con ACCELERATION_BODY_Y.");
    }
    else
    {
        Console.WriteLine("La aceleración del MPU está desincronizada con ACCELERATION_BODY_Y.");
        AjustarAceleracionY(aceleracionMPUY, accelerationBodyYLimited);
    }
}
else
{
    Console.WriteLine("La aceleración del MPU está dentro del rango mínimo para el movimiento de la cabina.");
}
```

En estas funciones se compara los valores de la aceleración X e Y del mpu y de las variables de las aceleraciones X e Y del avión para obtener la diferencia entre ellos.

```
void AjustarGiroX(double anguloMPUX, double planeBankDegreesLimited)
{
    // Calcular la diferencia
    double diferenciaGiroX = planeBankDegreesLimited - anguloMPUX;
    Console.WriteLine($"Ajustando cabina con diferencia: {diferenciaGiroX} grados");

    // Determinar la dirección y magnitud del ajuste
    if (diferenciaGiroX > 1)
    {
        // Enviar comando al ESP32 para mover la cabina a la derecha
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina a la derecha.");
        serialPort.WriteLine("MOVER_DERECHA"); // Comando adecuado para mover la cabina a la derecha
    }
    else if (diferenciaGiroX < -1)
    {
        // Enviar comando al ESP32 para mover la cabina a la izquierda
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina a la izquierda.");
        serialPort.WriteLine("MOVER_IZQUIERDA"); // Comando adecuado para mover la cabina a la izquierda
    }
}

void AjustarGiroY(double anguloMPUY, double planePitchDegreesLimited)
{
    // Calcular la diferencia
    double diferenciaGiroY = planePitchDegreesLimited - anguloMPUY;
    Console.WriteLine($"Ajustando cabina con diferencia: {diferenciaGiroY} grados");

    // Determinar la dirección y magnitud del ajuste
    if (diferenciaGiroY > 1)
    {
        // Enviar comando al ESP32 para mover la cabina hacia abajo
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina hacia abajo.");
        serialPort.WriteLine("MOVER_ABAJO"); // Comando adecuado para mover la cabina hacia abajo
    }
    else if (diferenciaGiroY < -1)
    {
        // Enviar comando al ESP32 para mover la cabina hacia arriba
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina hacia arriba.");
        serialPort.WriteLine("MOVER_ARRIBA"); // Comando adecuado para mover la cabina hacia arriba
    }
}
```

```
void AjustarAceleracionX(double aceleracionMPUX, double accelerationBodyXLimited)
{
    // Calcular la diferencia
    double diferenciaAcelX = accelerationBodyXLimited - aceleracionMPUX;
    Console.WriteLine($"Ajustando cabina con diferencia de aceleración: {diferenciaAcelX} ft/s²");

    // Determinar la dirección y magnitud del ajuste
    if (diferenciaAcelX > 0.5)
    {
        // Enviar comando al ESP32 para mover la cabina a la izquierda
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina a la izquierda.");
        serialPort.WriteLine("MOVER_IZQUIERDA"); // Comando adecuado para mover la cabina a la izquierda
    }
    else if (diferenciaAcelX < -0.5)
    {
        // Enviar comando al ESP32 para mover la cabina a la derecha
        Console.WriteLine("Enviando comando al ESP32 para mover la cabina a la derecha.");
        serialPort.WriteLine("MOVER_DERECHA"); // Comando adecuado para mover la cabina a la derecha
    }
    else if (diferenciaAcelX > 0 && diferenciaAcelX <= 0.5)
    {
        // Enviar comando al ESP32 para disminuir la aceleración hacia la izquierda
        Console.WriteLine("Enviando comando al ESP32 para disminuir la aceleración hacia la izquierda.");
        serialPort.WriteLine("DISMINUIR_ACELERACION_IZQUIERDA");
    }
    else if (diferenciaAcelX < 0 && diferenciaAcelX >= -0.5)
    {
        // Enviar comando al ESP32 para disminuir la aceleración hacia la derecha
        Console.WriteLine("Enviando comando al ESP32 para disminuir la aceleración hacia la derecha.");
        serialPort.WriteLine("DISMINUIR_ACELERACION_DERECHA");
    }
}

void AjustarAceleracionY(double aceleracionMPUY, double accelerationBodyYLimited)
{
    // Calcular la diferencia
    double diferenciaAcelY = accelerationBodyYLimited - aceleracionMPUY;
    Console.WriteLine($"Ajustando cabina con diferencia de aceleración: {diferenciaAcelY} ft/s²");

    // Determinar la dirección y magnitud del ajuste
    if (diferenciaAcelY > 0.5)
    {
        // Enviar comando al ESP32 para aumentar la aceleración hacia arriba
        Console.WriteLine("Enviando comando al ESP32 para aumentar la aceleración hacia arriba.");
        serialPort.WriteLine("AUMENTAR_ACELERACION_ARRIBA");
    }
    else if (diferenciaAcelY < -0.5)
    {
        // Enviar comando al ESP32 para aumentar la aceleración hacia abajo
        Console.WriteLine("Enviando comando al ESP32 para aumentar la aceleración hacia abajo.");
        serialPort.WriteLine("AUMENTAR_ACELERACION_ABAJO");
    }
    else if (diferenciaAcelY > 0 && diferenciaAcelY <= 0.5)
    {
        // Enviar comando al ESP32 para disminuir la aceleración hacia arriba
        Console.WriteLine("Enviando comando al ESP32 para disminuir la aceleración hacia arriba.");
        serialPort.WriteLine("DISMINUIR_ACELERACION_ARRIBA");
    }
    else if (diferenciaAcelY < 0 && diferenciaAcelY >= -0.5)
    {
        // Enviar comando al ESP32 para disminuir la aceleración hacia abajo
        Console.WriteLine("Enviando comando al ESP32 para disminuir la aceleración hacia abajo.");
        serialPort.WriteLine("DISMINUIR_ACELERACION_ABAJO");
    }
}
```

Estas funciones leen la diferencia todo momento y mandan comandos dependiendo del valor de la diferencia

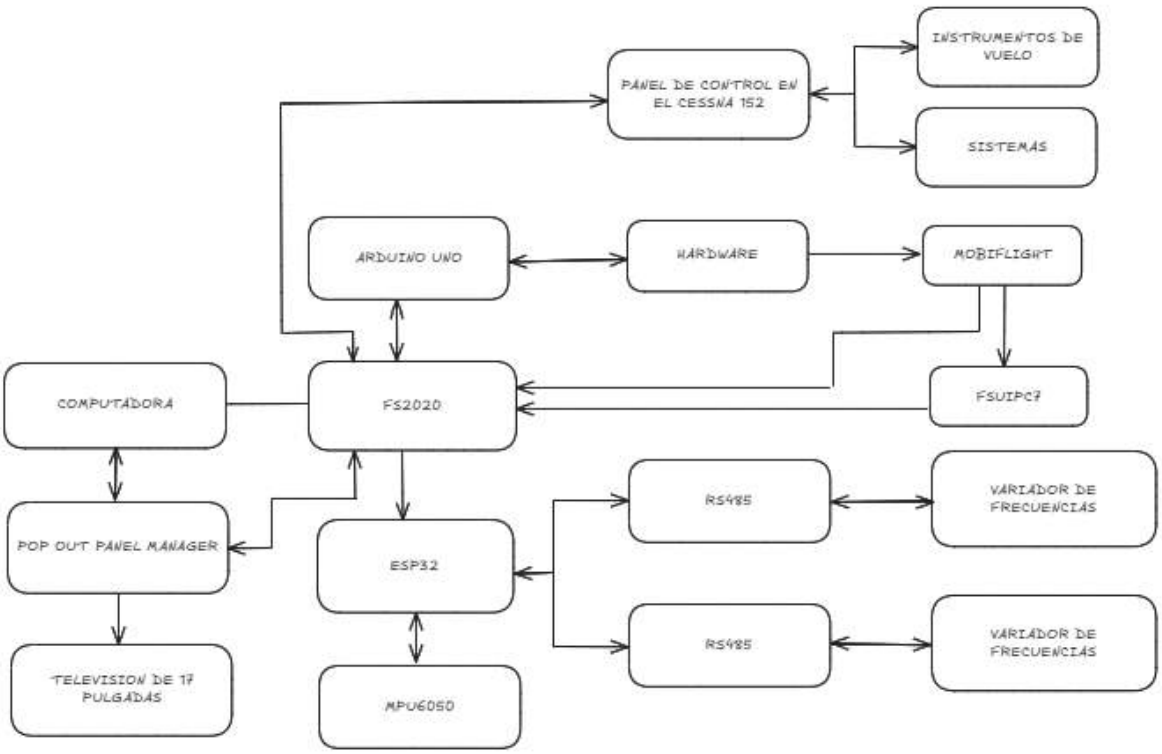
4. Segmento Destino y Alcance

Este simulador está destinado a estudiantes de ingeniería, entusiastas de la aviación y centros de enseñanza técnica que busquen herramientas de formación práctica en sistemas de control, mecánica de vuelo y electrónica aplicada. El proyecto tiene el potencial de ser una plataforma educativa completa en simulación de vuelo.

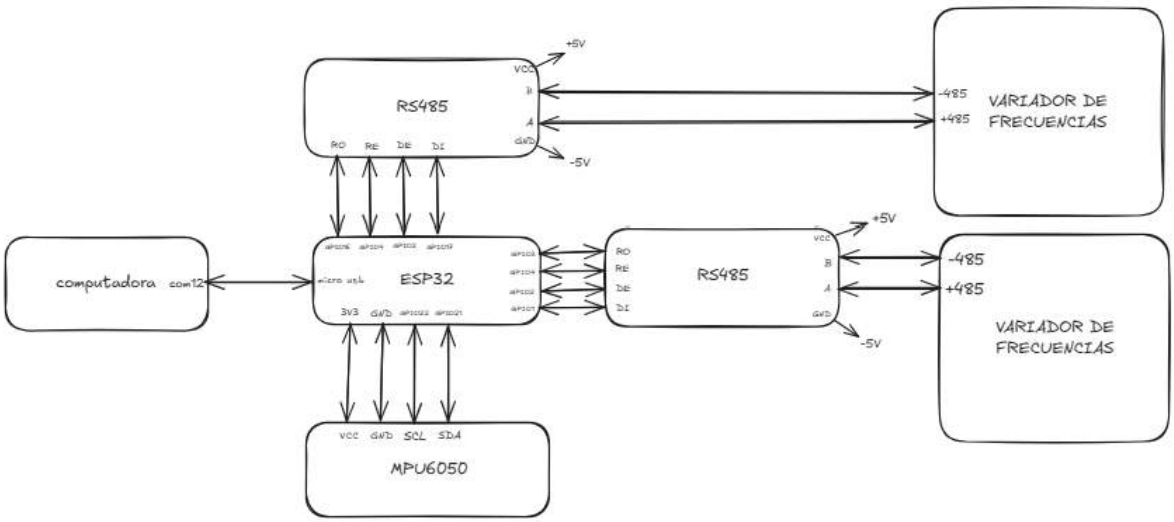
5. Captura Representativa del Proyecto

6. Diagrama en Bloques del Proyecto

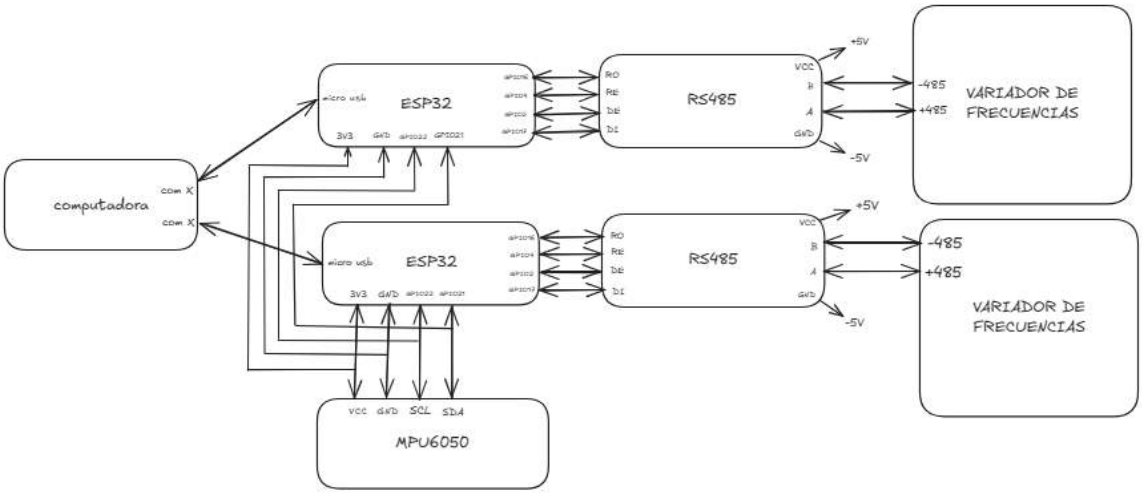
Diagrama general:



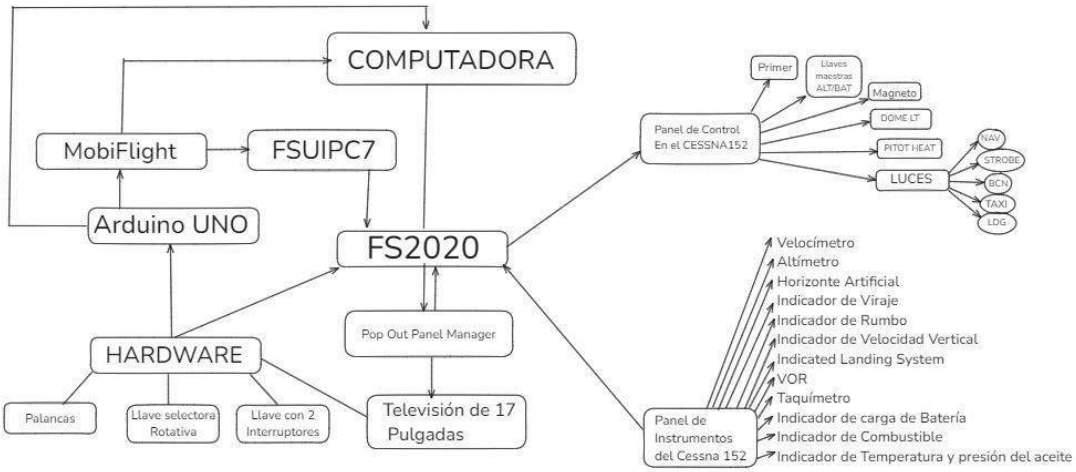
Variador de Frecuencia:



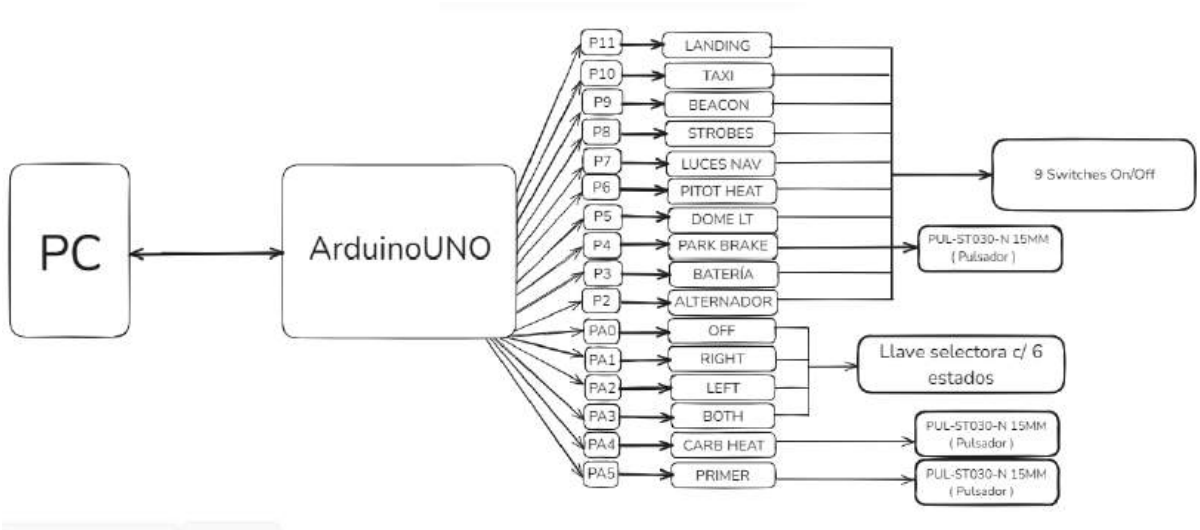
Cabe una posibilidad de que para evitar fallas en la transmision y recepcion de datos se usen 2 ESP32 y asi quedaría el diagrama:



Panel de control y visualización:



Conexiones del ArduinoUNO al Panel de control:



7. Resultados Conseguidos

Se trabaja en la implementación de la conexión y funcionamiento del panel de control con Flight Simulator 2020. Las pruebas iniciales de los motores todavía están en proceso. El sistema continúa en fase de pruebas y optimización, principalmente en el uso de variables de comunicación para mejorar la precisión de las lecturas y comandos en tiempo real, y también la comunicación con el variador de frecuencias

8. Software

Lenguajes Utilizados:

- **C/C++**: Programación del ESP32 y Arduino UNO para la comunicación con los variadores de frecuencia.

- **C/C++**: Programación del Arduino UNO para gestionar los interruptores.

Capturas de Códigos Significativos:

Código de prueba entre el ESP32 y el Potenciómetro digital:

```
// Pines de conexión para el AR-X9C103S
const int CS_PIN = 15;         // Chip Select
const int INC_PIN = 14;        // Incremento
const int UP_DOWN_PIN = 13;    // Dirección (incremento o decremento)

int targetFrequency = 0;       // Frecuencia deseada introducida por el usuario

void setup() {
  // Configuración de los pines de control
  pinMode(CS_PIN, OUTPUT);
  pinMode(INC_PIN, OUTPUT);
```

```
  pinMode(UP_DOWN_PIN, OUTPUT);

  // Inicializar los pines en estado alto
  digitalWrite(CS_PIN, HIGH);
  digitalWrite(INC_PIN, HIGH);
  digitalWrite(UP_DOWN_PIN, HIGH);

  Serial.begin(115200);
  Serial.println("Introduce la frecuencia deseada en el Monitor Serie:");
}

void loop() {
  // Comprobar si hay datos en el Monitor Serie
  if (Serial.available() > 0) {
    targetFrequency = Serial.parseInt(); // Leer la frecuencia ingresada por el usuario
    Serial.print("Frecuencia recibida: ");
    Serial.println(targetFrequency);

    // Ajustar la resistencia en función de la frecuencia
    adjustResistance(targetFrequency);
  }
  delay(500); // Pausa para evitar lecturas rápidas consecutivas
}

void adjustResistance(int frequency) {
  int steps = map(frequency, 0, 100, 0, 100); // Mapear frecuencia a pasos (ajustar según tu rango)

  // Incrementar la resistencia del potenciómetro al máximo
  Serial.println("Ajustando resistencia...");
  digitalWrite(UP_DOWN_PIN, HIGH);  // Configurar para incrementar
  digitalWrite(CS_PIN, LOW);        // Activar el potenciómetro

  for (int i = 0; i < steps; i++) {
    pulseIncrement();
    delay(50);  // Pausa entre pasos para observar el cambio
  }

  digitalWrite(CS_PIN, HIGH);       // Desactivar el potenciómetro
}

void pulseIncrement() {
  digitalWrite(INC_PIN, LOW);
  delayMicroseconds(1); // Pulso corto para cambiar el valor
  digitalWrite(INC_PIN, HIGH);
  delayMicroseconds(1);
}
```

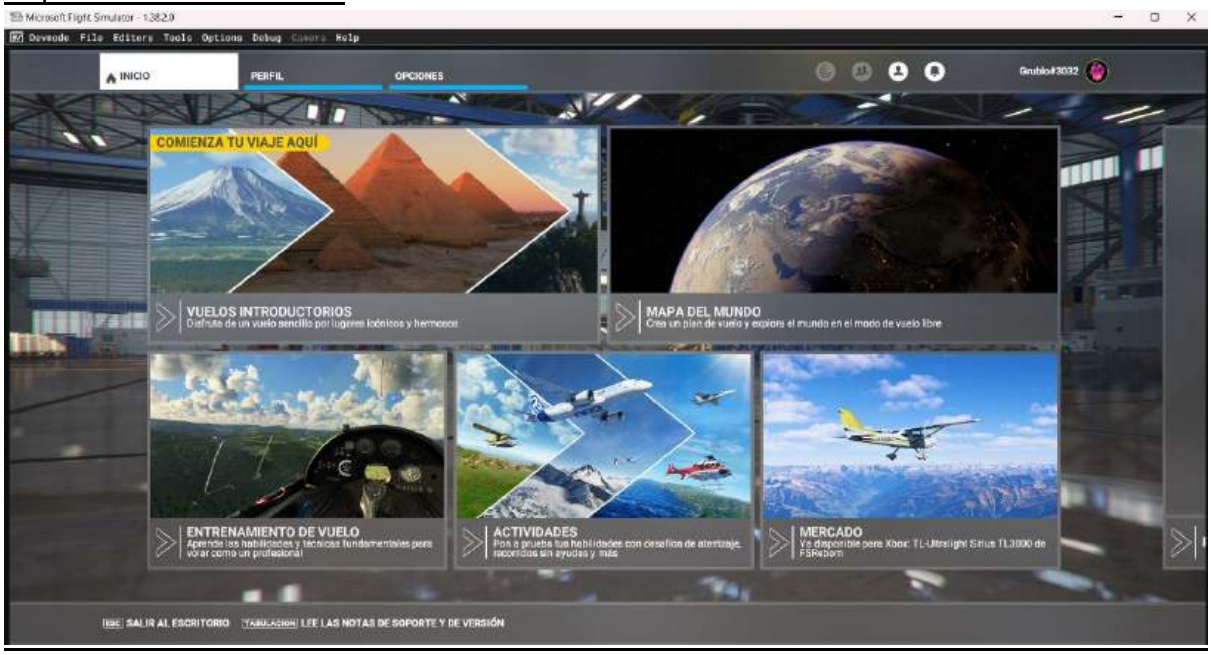
Código de prueba entre el ESP32 y el Potenciómetro digital con el MPU6050:

```
#include <Wire.h>
#include <MPU6050.h>

MPU6050 mpu;

// Pines de conexión para el AR-X9C103S
const int CS_PIN = 15;         // Chip Select
const int INC_PIN = 14;        // Incremento
```

```
const int UP_DOWN_PIN = 13;    // Dirección (incremento o decremento)

// Rango de ángulo para mapear a la resistencia del potenciómetro
const int ANGLE_MIN = -90;
const int ANGLE_MAX = 90;

// Variables para el control de resistencia
int lastSteps = -1;  // Valor previo de pasos del potenciómetro
const float DEADZONE_MIN = -1.0;
const float DEADZONE_MAX = 1.0;

void setup() {
  Serial.begin(115200);
  Serial.println("Iniciando el MPU6050 y el potenciómetro digital.");

  // Configuración de los pines de control
  pinMode(CS_PIN, OUTPUT);
  pinMode(INC_PIN, OUTPUT);
  pinMode(UP_DOWN_PIN, OUTPUT);

  // Inicializar los pines en estado alto
  digitalWrite(CS_PIN, HIGH);
  digitalWrite(INC_PIN, HIGH);
  digitalWrite(UP_DOWN_PIN, HIGH);

  // Inicializar la comunicación I2C con el MPU6050
  Wire.begin();
  mpu.initialize();

  // Comprobar conexión con el MPU6050
  if (mpu.testConnection()) {
    Serial.println("MPU6050 conectado correctamente.");
  } else {
    Serial.println("Error al conectar el MPU6050. Verifica las conexiones.");
  }
}

void loop() {
  int16_t ax, ay, az;
  mpu.getAcceleration(&ax, &ay, &az);

  // Calcular el ángulo aproximado en el eje X
  float angleX = atan2(ay, az) * 180 / PI;

  // Aplicar punto muerto
  if (angleX > DEADZONE_MIN && angleX < DEADZONE_MAX) {
    // Si el ángulo está en el rango del punto muerto, no hacer ajustes
    return;
  }

  // Mapear el ángulo al rango de pasos del potenciómetro (0 a 99)
  int steps = map(angleX, ANGLE_MIN, ANGLE_MAX, 0, 99);

  // Solo ajustar si el valor de pasos ha cambiado
  if (steps != lastSteps) {
    Serial.print("Ángulo X: ");
    Serial.print(angleX);
    Serial.print(" | Ajustando a pasos del potenciómetro: ");
```

```
    Serial.println(steps);

    adjustResistanceSmoothly(steps);
    lastSteps = steps;
  }

  // Reducir el retraso para actualizaciones más continuas
  delay(10);
}

void adjustResistanceSmoothly(int targetSteps) {
  // Determina la dirección según el cambio necesario
  bool increment = targetSteps > lastSteps;
  digitalWrite(UP_DOWN_PIN, increment ? HIGH : LOW);
  digitalWrite(CS_PIN, LOW); // Activar el potenciómetro

  // Cambia un paso a la vez para hacer el ajuste suave
  int stepDifference = abs(targetSteps - lastSteps);
  for (int i = 0; i < stepDifference; i++) {
    pulseIncrement();
    delay(5); // Pausa corta para cada pulso, ajustable para mayor fluidez
  }

  digitalWrite(CS_PIN, HIGH); // Desactivar el potenciómetro
  Serial.println("Resistencia ajustada suavemente.");
}

void pulseIncrement() {
  digitalWrite(INC_PIN, LOW);
  delayMicroseconds(1); // Pulso corto para cambiar el valor
  digitalWrite(INC_PIN, HIGH);
  delayMicroseconds(1);
}
```

Captura de interfaz visual:



Estructuras de Datos:

Las librerías principales usadas en el variador de frecuencias son:

- **MPU6050_light:** Esta librería está diseñada para facilitar el uso del sensor **MPU6050**, que combina un acelerómetro y un giroscopio en un solo módulo. Esta librería es una versión optimizada y ligera para Arduino.

Lista de variables del Flight Simulator 2020

Variables de instrumentos:

- Altímetro:

INDICATED ALTITUDE	The indicated altitude.	Feet	✓
KOHLSMAN SETTING HG:index	The value for the given altimeter index in inches of mercury. ***IMPORTANT!*** *In the system.cfg file, altimeters are indexed from 0, but the SimVar indexes from 1. So, altimeter 0 in that file is accessed using* `KOHLSMAN SETTING HG:1`*, 1 by* `KOHLSMAN SETTING HG:2`*, etc...*	Inches of Mercury, *inHg*	✗

- Brújula Magnética:

MAGNETIC COMPASS	Compass reading.	Degrees	✗

- Coordinador de giro:

DELTA HEADING RATE	Rate of turn of heading indicator.	Radians per second	✓
TURN COORDINATOR BALL	Turn coordinator ball position.	Position 128 (-127 to 127)	✗
TURN INDICATOR RATE	Turn indicator reading. ***NOTE:*** *This is available in multiplayer to all* ***near*** *aircraft. See here for more information: Note On SimVars In Multiplayer.*	Radians per second	✗

- Horizonte artificial:

ATTITUDE INDICATOR BANK DEGREES	AI bank indication	Radians	✗
ATTITUDE INDICATOR PITCH DEGREES	AI pitch indication	Radians	✗

- Variómetro:

VERTICAL SPEED	The current indicated vertical speed for the aircraft.	Feet (*ft*) per second	✓

- Velocímetro:

AIRSPEED INDICATED	Indicated airspeed.	*Knot*s	✓

- ADF:

ADF ACTIVE FREQUENCY:index	*ADF* frequency. Index of 1 or 2.	Frequency ADF BCD32	✗
ADF RADIAL MAG:index	Returns the magnetic bearing to the currently tuned ADF transmitter.	Degrees	✗

- VOR:

NAV VOR LATLONALT:index	Returns the VOR station latitude, longitude and altitude.	SIMCONNECT_DATA_LATLONALT structure	✗
NAV OBS	OBS setting. Index of 1 or 2.	Degrees	✗
NAV TOFROM	Returns whether the Nav is going to or from the current radial (or is off).	Enum: 0 = Off 1 = TO 2 = FROM	✗
NAV RADIAL	Radial that aircraft is on.	Degrees	✗

- ILS:

NAV LOCALIZER	Localizer course heading.	Degrees	✗
NAV LOC AIRPORT IDENT	The airport ICAO ident for the localizer that is currently tuned on the nav radio (like 'EGLL' or 'KJFK')	String	✗

NAV LOC RUNWAY NUMBER	NAV LOC RUNWAY NUMBER - The number portion of the runway that the currently tuned localizer is tuned to (so if the runway was 15L, this would be 15).	String '1' - '36' 'N' 'NE' 'E' 'SE' 'S' 'SW' 'W' 'NW'	✗
NAV RAW GLIDE SLOPE	The glide slope angle.	Degrees	✗
NAV GS FLAG	Glideslope flag.	Bool	✗
NAV GS LATLONALT:index	Returns the glide slope.	SIMCONNECT_DATA_LATLONALT structure	✗
INNER MARKER	Inner marker state.	Bool	✓
INNER MARKER LATLONALT	Returns the latitude, longitude and altitude of the inner marker of an approach to a runway, if the aircraft is within the required proximity, otherwise it will return zeros.	SIMCONNECT_DATA_LATLONALT structure	✗
MIDDLE MARKER	Middle marker state.	Bool	✓
MIDDLE MARKER LATLONALT	Returns the latitude, longitude and altitude of the middle marker.	SIMCONNECT_DATA_LATLONALT structure	✗
OUTER MARKER	Outer marker state.	Bool	✓
OUTER MARKER LATLONALT	Returns the latitude, longitude and altitude of the outer marker.	SIMCONNECT_DATA_LATLONALT structure	✗

- Indicador de temperatura y presión del aceite:

GENERAL ENG OIL PRESSURE:index	The indexed engine (see note) oil pressure.	Psf	✓
GENERAL ENG OIL TEMPERATURE:index	The indexed engine (see note) oil temperature.	*Rankine*	✓

- Indicador de carga de batería:

ELECTRICAL MAIN BUS AMPS	Main bus current	Amperes	✓
ELECTRICAL MAIN BUS VOLTAGE	The main bus voltage. Use a bus index when referencing.	Volts	✓
ELECTRICAL BATTERY LOAD	The load handled by the battery (negative values mean the battery is *receiving* current). Use a battery index when referencing.	Amperes	✓

- Indicador de RPM:

GENERAL ENG RPM:index	The RPM for an indexed engine (see note). ***NOTE**: This is available in multiplayer to all **far** aircraft. See here for more information: Note On SimVars In Multiplayer.*	*RPM*	✓
GENERAL ENG PCT MAX RPM:index	Percent of max rated rpm for the indexed engine (see note). ***NOTE**: This is available in multiplayer to all **far** aircraft. See here for more information: Note On SimVars In Multiplayer.*	Percent	✓
MAX RATED ENGINE RPM	Maximum rated rpm for the indexed engine (see note).	*RPM*	✗

- Indicador de combustible:

FUEL TOTAL CAPACITY	Total fuel capacity of the aircraft for *all* tanks.	*Gallons*	✖
FUEL TOTAL QUANTITY	Current total quantity of fuel in volume for *all* tanks of the aircraft.	*Gallons*	✖
FUEL LEFT QUANTITY	Current quantity in volume of *all* the tanks on the left side of the aircraft.	*Gallons*	✖
FUEL RIGHT QUANTITY	Current quantity in volume of *all* the tanks on the right side of the aircraft.	*Gallons*	✖

Variables utilizadas en Comparaciones.cs:

PLANE PITCH DEGREES	Pitch angle, although the name mentions degrees the units used are radians.	Radians	✔
PLANE BANK DEGREES	Bank angle, although the name mentions degrees the units used are radians.	Radians	✔
ACCELERATION BODY X	Acceleration relative to aircraft X axis, in east/west direction.	Feet (*ft*) per second squared	✖
ACCELERATION BODY Y	Acceleration relative to aircraft Y axis, in vertical direction.	Feet (*ft*) per second squared	✖

9. Sistema Embebido

Componentes:

- **ESP32**: Controlador central para los variadores de frecuencia, MPU6050 y las variables del MSFS2020.

- **Arduino UNO**: Controla el panel de control, configurado con MobiFlight para activar los instrumentos.

- **Potenciómetro digital:** Modifica los valores eléctricos que afectan directamente el comportamiento de los variadores de frecuencia.

Protocolo y Conexiones:

- **I2C**: Utilizado por el **MPU6050** para la detección de movimiento y aceleración.

Configuración de Pines:

- Conexión del MPU6050:
- VCC del MPU6050 a 3.3V del ESP32.
- GND del MPU6050 a GND del ESP32.
- SDA del MPU6050 a GPIO21 del ESP32.
- SCL del MPU6050 a GPIO22 del ESP32.

- Conexión de los Potenciómetros Digitales

a. Para el primer potenciómetro:
- CS_PIN a GPIO15 del ESP32.
- INC_PIN a GPIO14 del ESP32.
- UP_DOWN_PIN a GPIO13 del ESP32.
- VW va al GND del variador 1
- VH y VL van a dos pines del variador 1
- VCC del potenciómetro a la entrada analógica del variador 1 (AI1).
- GND del potenciómetro a GND común del variador 1

b. Para el segundo potenciómetro:
- CS_PIN a GPIO32 del ESP32.
- INC_PIN a GPIO33 del ESP32.
- UP_DOWN_PIN a GPIO25 del ESP32.
- VW va al GND del variador 2
- VH y VL van a dos pines del variador 2
- VCC del potenciómetro a la entrada analógica del variador 2 (AI1).
- GND del potenciómetro a GND común del variador 2.

- Conexión de los optoacopladores y resistencias:

a. Para el primer optoacoplador:
- Anodo del led va a la primera resistencia de 330 Ω y luego al GPIO27 del ESP32
- Catodo del led va a GND del esp32
- Colector va conectado a el puerto x1 del variador 1 y también conectado a la primera resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optocoplador va a COM del variador 1 y también va a GND de la fuente

b. Para el segundo optoacoplador:

- Anodo del led va a la segunda resistencia de 330 Ω y luego al GPIO12 del ESP32
- Catodo del led va a GND del esp32
- Colector va conectado a el puerto x2 del variador 1 y también está conectado a la segunda resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optocoplador va a COM del variador 1 y también va a GND de la fuente

c. Para el tercer optoacoplador:

- Anodo del led va la tercera resistencia de 330 Ω y luego al GPIO18 del ESP32
- Catodo del led va a GND del esp32
- Colector va conectado a el puerto x1 del variador 2 y también está conectado a la tercera resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optocoplador va a COM del variador 2 y también va a GND de la fuente

d. Para el cuarto optoacoplador:

- Anodo del led va la cuarta resistencia de 330 Ω y luego al GPIO19 del ESP32
- Catodo del led va a GND del esp32
- Colector va conectado a el puerto x2 del variador 2 y también está conectado a la cuarta resistencia de 10 KΩ y luego va al VCC de la fuente
- Emisor del optocoplador va a COM del variador 2 y también va a GND de la fuente

10. Electrónica

Componentes y Configuraciones:

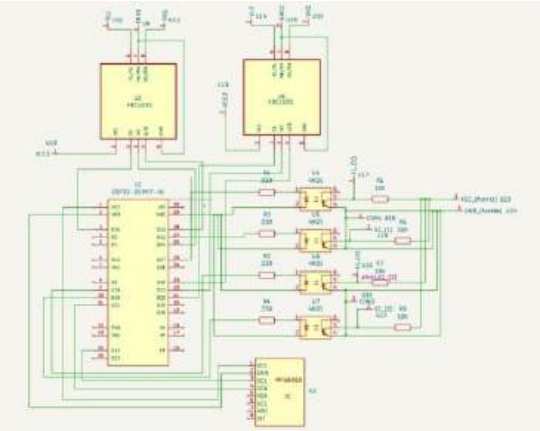
- **Variador de Frecuencia GK500-2T2.2B**: Utilizado para controlar los motores a través del protocolo RS485, con una tasa de transferencia de 9600 baudios.

- **Arduino UNO**: Control del panel de instrumentos.

- **MPU6050**: Detección de movimiento y orientación para simular los ejes de vuelo.

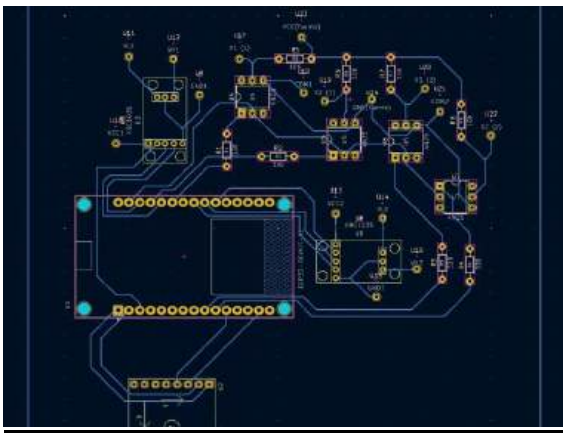
- **Potenciómetro digital:** Modifica los valores eléctricos que afectan directamente el comportamiento de los variadores de frecuencia.

Esquemático de Conexiones y Fuente de Alimentación:



- Fuente de alimentación separada para el ESP32 y variador de frecuencia, asegurando estabilidad y seguridad en la operación.

PCB:



11. Estructura

La estructura de Skylf Flight Simulator, pasó por cambios desde su versión 2019(Avis) hasta la versión que hoy en día nosotros nos hemos dedicado a modificar en la versión 2024(Skylf).
Estos cambios se produjeron por razones, como la implementación de nuevos sistemas de manipulación motora, renovaciones de infraestructura que antes eran inseguras e implementaciones adicionales para distinguir a nuestra versión Skylf.
Anteriormente la cabina contaba con un sistema a 4 soportes por sistema roscado para el soporte de los instrumentos de vuelo, hoy en dia ya es una base fija, soldada directamente a la televisión superior y a la cara posterior del simulador.
El joystick de movimiento, contaba con una base de metal colocada en diagonal al usuario. Actualmente nuestra manera de mover el simulador es mediante un yoke, puesto sobre una base metálica, roscada a la base inferior y soldada al pilar donde se haya el motor dónde predomina el alabeo, dónde se encontraba el anterior stick de control, está colocado (de manera vertical) las palancas del pedestal.

3 de los 4 soportes del anterior panel, fueron removidos completamente, el cuarto soporte fué recortado a la mitad, y se le hizo una estructura a medida para nuestro nuevo panel de control.
Las partes oxidadas fueron removidas y renovadas con pintura negra antioxidante.

12. Anexo

Resultados de Investigaciones Hechas:

Variadores de Frecuencia: Los variadores de frecuencia, como el GK500-2T2.2B, permiten controlar la velocidad y es para motores de aplicaciones eléctricas. A través de la comunicación Modbus, se pueden ajustar parámetros en tiempo real, facilitando la implementación de sistemas de control más precisos.

Sensor MPU6050: Este sensor combina un acelerómetro y un giroscopio en un solo chip, permitiendo la captura de datos sobre orientación y movimiento. La utilización de este sensor en el simulador de vuelo ha mostrado ser efectiva para la estabilización y el control dinámico del simulador, proporcionando datos esenciales para la experiencia de vuelo realista.

Fuentes:

https://github.com/brizuu750/SKYLF/tree/main/variadores%20de%20frecuencia/manuales