# Desarrollo y Análisis de Actividades con un Kit de IoT (Documentacion)

Higinio

AUN FALTA EDITAR Y DAREL FORMATO

# INTRODUCCIÓN

El Internet de las Cosas (IoT) se ha consolidado como uno de los pilares esenciales de la transformación digital, facilitando la interconexión de dispositivos para recopilar y analizar datos en tiempo real. Este documento tiene como objetivo presentar el desarrollo y análisis de cinco actividades prácticas llevadas a cabo con un kit de IoT, diseñado específicamente para el aprendizaje y la implementación de soluciones en este ámbito.

En este trabajo, se detalla la experiencia de utilizar el Arduino MKR WiFi 1010 junto con el MKR IoT Carrier Rev2, una combinación de hardware que permite explorar desde conceptos básicos hasta aplicaciones avanzadas en el campo del IoT. Las actividades realizadas cubren una amplia gama de aspectos, desde la familiarización con los componentes básicos del kit, hasta la implementación de sistemas de monitoreo ambiental y el desarrollo de aplicaciones que emplean diversos protocolos de comunicación inalámbrica.

El propósito principal de esta documentación es ofrecer un registro detallado del proceso de aprendizaje y desarrollo con el kit de IoT, analizando tanto los logros alcanzados como los desafíos enfrentados en cada actividad. Además, este documento no solo sirve como testimonio del trabajo realizado, sino también como una guía de referencia para futuros desarrolladores interesados en adentrarse en el mundo del IoT utilizando estas herramientas.

A continuación, se presenta una descripción detallada del hardware empleado, comenzando por las especificaciones técnicas del Arduino MKR WiFi 1010 y el MKR IoT Carrier Rev2, componentes clave para la ejecución de este proyecto.

# ARDUINO MKR WIFI 1010

El Arduino® MKR WiFi 1010 es una excelente opción para quienes desean iniciarse en el desarrollo de proyectos de Internet de las Cosas (IoT). Esta placa integra un procesador Arm® Cortex®-M0 de 32 bits SAMD21, lo que le permite ofrecer un rendimiento eficiente y balanceado para diversas aplicaciones. Una de sus principales ventajas es la inclusión del módulo uBlox Nina W102, que proporciona conectividad tanto WiFi como Bluetooth BLE (Bluetooth de bajo consumo), permitiendo desde la transferencia de datos hacia dispositivos móviles hasta la creación de redes de sensores conectadas a un enrutador.

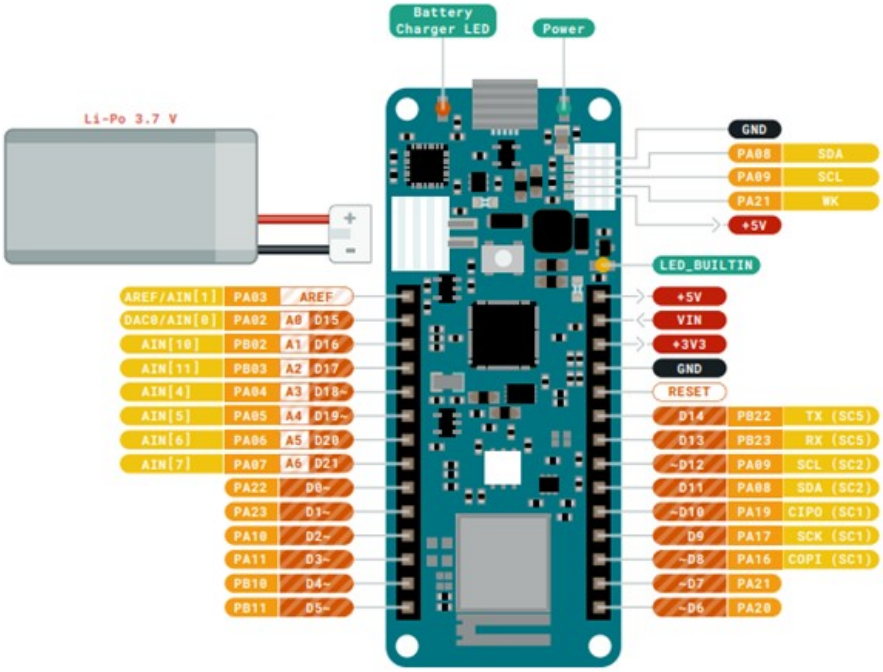


Figure 1: Arduino MKR WiFi 1010

Adicionalmente, la placa incorpora un chip criptográfico ECC508, el cual mejora la seguridad de las comunicaciones, haciendo posible el desarrollo de proyectos que requieran protección de datos. Gracias a su diseño compacto y a las múltiples características técnicas que ofrece, esta placa es ideal para una amplia gama de aplicaciones IoT, desde prototipos simples hasta proyectos más avanzados.

Entre sus especificaciones principales destacan las siguientes:

- Conector Micro USB (USB-B).
- 8 pines de entrada/salida digitales.
- 7 pines de entrada analógica con un ADC configurable de 8, 10 o 12 bits.
- 1 pin de salida analógica con DAC de 10 bits.
- 13 pines PWM (0-8, 10, 12, A3, A4).
- Interrupciones externas disponibles en pines específicos (0, 1, 4, 5, 6, 7, 8, 9, A1, A2).
- Módulo uBlox Nina W102 para conectividad WiFi y Bluetooth BLE.
- Compatibilidad con protocolos de comunicación I2C, UART y SPI.
- Operación a un voltaje de E/S de 3.3 V.
- Capacidad de corriente continua de 7 mA por pin.
- Velocidad de reloj del procesador de 48 MHz y 32.768 kHz para el RTC.

- Memoria del procesador SAMD21G18A: 256 KB de memoria Flash y 32 KB de SRAM.
- Memoria del módulo uBlox Nina W102: 448 KB de ROM, 520 KB de SRAM y 2 MB de memoria Flash.

# MKR IoT CARRIER REV 2

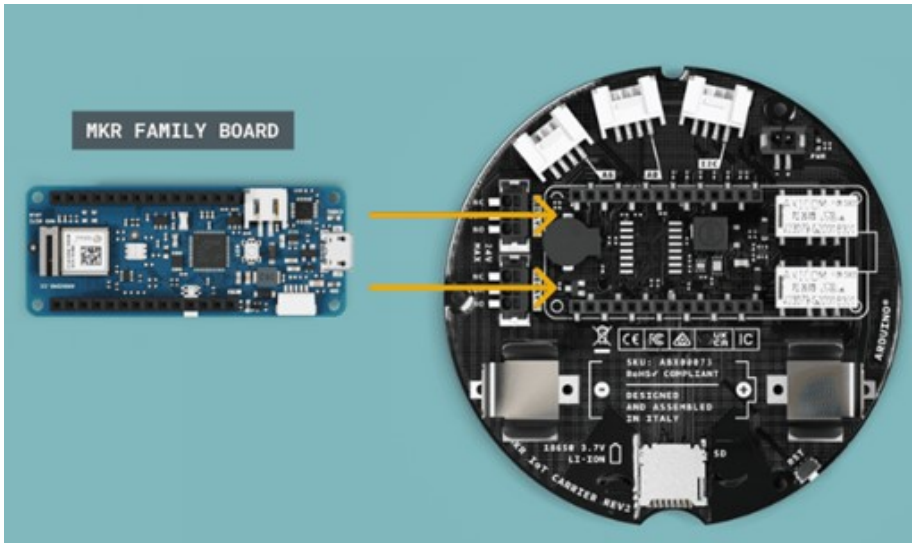


Figure 2: Carrier Rev2

El Arduino® MKR IoT Carrier Rev2 es una plataforma robusta y versátil que facilita el desarrollo de prototipos IoT. Su diseño intuitivo permite a los desarrolladores enfocarse en la programación y la lógica de sus proyectos, sin preocuparse por la complejidad de integrar múltiples componentes electrónicos. Esta herramienta ofrece una solución integral para monitoreo, automatización y conectividad en proyectos innovadores de Internet de las Cosas

Entre sus principales características destacan:

1. Relés para control de cargas eléctricas de alto voltaje.
2. Compatible con baterías de iones de litio tipo 18650.
3. Pines de alta potencia.
4. Conectores Grove para expansión con sensores.
5. Conectores JST para batería.
6. Zumbador.
7. Sensor ambiental BME688.
8. Luces de estado de los relés.
9. Sensor RGB y de gestos.

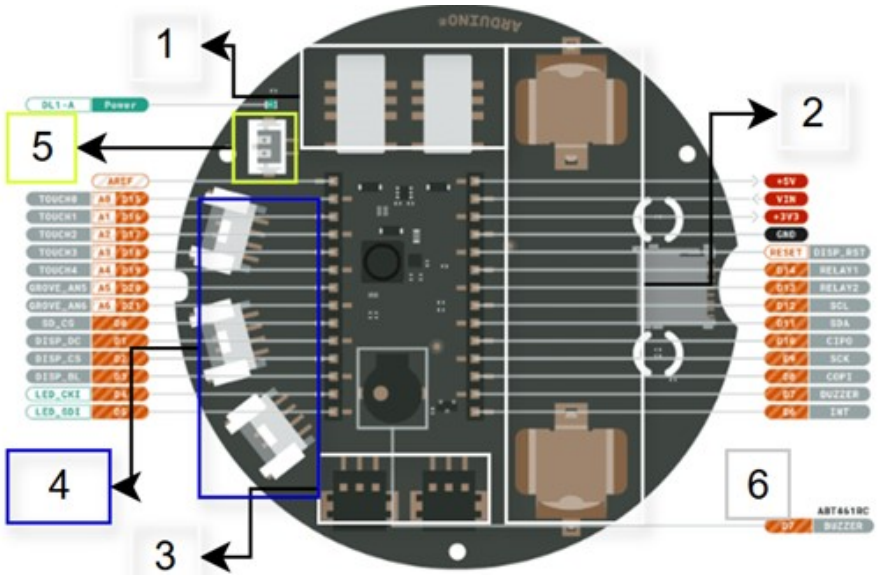


Figure 3: Vista superior de Carrier Rev 2

10. LEDs RGB programables.
11. Pantalla TFT de 1.3” a color.
12. Pads táctiles capacitivos.

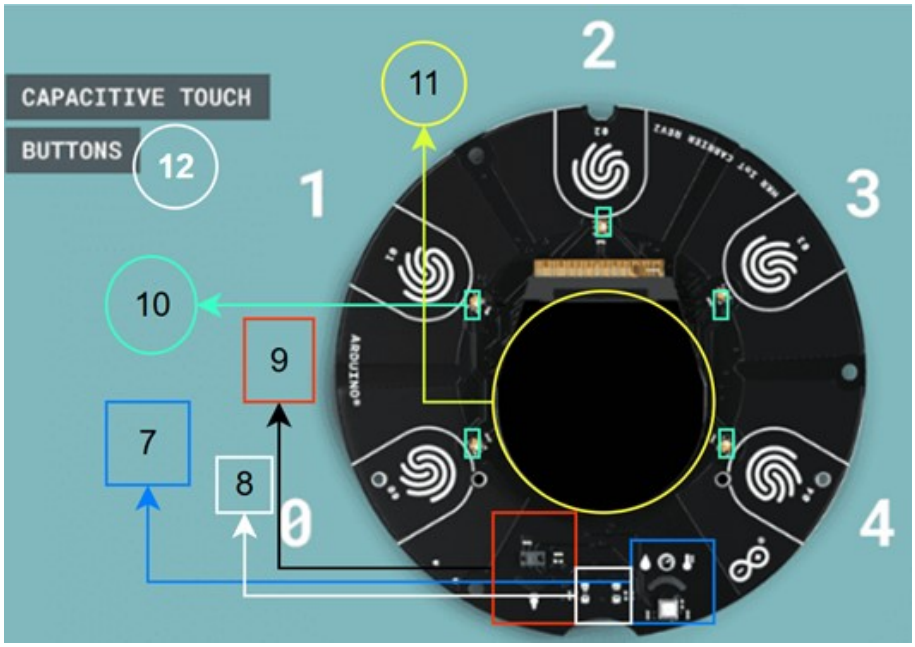


Figure 4: Vista inferior de Carrier Rev 2

El Arduino® MKR IoT Carrier Rev2 no solo cuenta con un potente conjunto de hardware, sino que también permite el desarrollo eficiente de aplicaciones IoT mediante librerías específicas que simplifican su programación:

1. WiFiNINA: Esta librería permite gestionar de forma sencilla la conectividad Wi-Fi a través del módulo u-blox NINA-W102. Proporciona funciones para

escanear redes, conectarse a puntos de acceso, enviar y recibir datos, así como implementar seguridad en las conexiones.

2. ArduinoBLE: Facilita la comunicación mediante Bluetooth Low Energy (BLE). Con esta librería se pueden desarrollar aplicaciones para conectar dispositivos IoT, compartir datos o configurar redes de sensores inalámbricos.

3. Arduino_MKRIoTCarrier: Específicamente diseñada para el Carrier Rev2, permite acceder y controlar todos sus componentes, como sensores, pantalla TFT, botones táctiles, LEDs RGB y más. Simplifica la integración de funciones avanzadas

# MATERIALES

## Hardware

- MKR IoT Carrier Rev2.
- Microcontrolador compatible (Arduino MKR WiFi 1010 u otro recomendado por el carrier).
- Cable micro-USB para la conexión.
- Sensor de humedad en el suelo Moisture 1.2v.
- Sensor RIP (detector de movimiento).

## Software

- Entorno de desarrollo Arduino IDE o compatibles.
- Librerías necesarias:
  - Arduino_MKRIoTCarrier: Para interactuar con los LEDs, botones táctiles y otros componentes del carrier.
  - WiFiNINA: Para la conectividad Wi-Fi.
  - ArduinoBLE: Para la comunicación Bluetooth Low Energy.

# DESARROLLO DE ACTIVIDADES

# Actividad 1

Se comenzó utilizando los ejemplos proporcionados por la librería `Arduino_MKRIoTCarrier.h`, específicamente el ejemplo de touchPads y otros incluidos en la documentación del Carrier Rev2, como los relacionados con el buzzer y el display. En esta etapa, se probaron de manera independiente todos los componentes del Carrier que estarían involucrados en el proyecto, verificando que las líneas de código correspondientes activaran correctamente cada dispositivo.

Una vez finalizadas estas pruebas, se inició la codificación del programa. Este proceso se realizó por etapas: primero, se implementó la funcionalidad para encender los LEDs en diferentes colores y detener su cambio al presionar los botones táctiles del touchpad. Cada botón controlaba un LED diferente. Posteriormente, se agregó una función que verificaba si los cinco LEDs estaban detenidos en el mismo color; en ese caso, se mostraba un mensaje en el display. De esta forma, el desarrollo avanzó de manera iterativa hasta completar la primera versión del juego, que cubría casi todos los requerimientos iniciales.

## Versión 1

En esta versión inicial, se implementa un juego de LEDs que cambian de color cíclicamente. El jugador gana un punto al detener todos los LEDs en el mismo color. Para ganar el juego, se deben conseguir tres puntos, evitando repetir colores ya utilizados en la misma partida. Sin embargo, esta versión presenta limitaciones, ya que el juego resulta monótono, y se requiere presionar los botones táctiles para fijar y desfijar manualmente el estado de los LEDs.

## Versión 2

La segunda versión introduce mayor aleatoriedad al juego. En lugar de cambiar los colores de forma cíclica, ahora lo hacen de manera aleatoria, lo que incrementa la imprevisibilidad y el desafío. A pesar de esta mejora, el juego sigue sin incluir una opción de apagado o reinicio automático, y la interacción sigue siendo manual, requiriendo al jugador presionar para fijar y desfijar los LEDs.

## Versión 3

La tercera versión presenta un nivel de complejidad más elevado. Los LEDs ahora parpadean de forma aleatoria con diferentes colores, incluyendo el estado apagado. Una nueva característica es que, si se presiona el botón táctil cuando un LED está apagado, este estado se fija. Sin embargo, persiste el problema de fijar y desfijar manualmente, lo que puede dificultar la experiencia de juego.

## Tabla comparativa

| **Aspecto** | **Versión 1** | **Versión 2** | **Versión 3** |
|---|---|---|---|
| Nivel de complejidad | Básico. Juego cíclico con un enfoque simple. | Intermedio. Introduce cambios aleatorios de color. | Avanzado. Agrega parpadeo aleatorio de LEDs y mayor dinamismo. |
| Interacción | Manual, requiere presionar botones. | Igual que la versión 1, pero los colores son aleatorios. | Similar, pero el estado apagado también puede fijarse. |

| | | | |
|---|---|---|---|
| Cambio de colores | Ciclos predefinidos de colores. | Colores aleatorios. | Los LEDs parpadean aleatoriamente. |
| Puntaje | Se obtiene un punto si los LEDs coinciden en color. | Igual que la versión 1. | Se mantiene la mecánica, pero es más desafiante. |
| Dinámica del juego | Monótona. | Más dinámica, pero limitada en mecánicas. | Más desafiante debido al parpadeo y fijación. |
| Estado apagado | No implementado. | No implementado. | Implementado, aumentando la dificultad. |
| Modo de apagado | No implementado. | No implementado. | No implementado. |

# Estructura del programa

1. **Configuración de hardware (inicialización de LEDs y botones)**
   El bloque de configuración se encuentra en la función `setup()`. Aquí se inicializan los componentes principales del hardware:

   - **Comunicación serial**: Se establece con `Serial.begin(9600)` para depuración.
   - **Objeto carrier**: El objeto `carrier` (instancia de `MKRIoTCarrier`), que controla LEDs, pantalla, touchpads y el buzzer, se inicializa con `carrier.begin()`.
   - **Luminosidad de LEDs**: Se configura con `carrier.leds.setBrightness(10)` para evitar un brillo excesivo.
   - **Pantalla**: Se limpia con `carrier.display.fillScreen(ST77XX_BLACK)` y se muestra el puntaje inicial llamando a `mostrarPuntaje()`.

2. **Ciclo principal (`loop`) con lógica del juego**
   El núcleo del programa se encuentra en la función `loop()`. Este bloque gestiona las acciones que se realizan durante el juego:

   - **Parpadeo de LEDs y cambio de color**: Mientras el puntaje sea menor a 3, la función `parpadeoConCambioDeColor()` controla el parpadeo de los LEDs, cambia colores aleatoriamente y permite al usuario interactuar con los botones para fijar o detener LEDs.
   - **Animación de victoria**: Si el puntaje llega a 3, se activa la animación de "victoria". Se apagan los LEDs, se muestra un mensaje de triunfo en la pantalla y se reproduce un sonido en el buzzer. Tras una pausa, se reinicia el juego con `reiniciarJuego()`.

3. **Funciones auxiliares**
   Estas funciones son piezas clave para modularizar y organizar el programa:

(a) `mostrarPuntaje()`

- Muestra el puntaje actual en la pantalla LCD. Limpia la pantalla y utiliza funciones de cursor y texto (`setCursor`, `setTextColor`, `setTextSize`) para posicionar y mostrar el puntaje.

(b) `verificarColoresYSumarPuntos()`

- Verifica si todos los LEDs están detenidos y tienen el mismo color. Si además este color no ha sido usado anteriormente, se incrementa el puntaje, se guarda el color en el historial (`coloresUsados`), y se muestra una retroalimentación visual y sonora.

(c) `reiniciarJuego()`

- Restablece las variables del juego, como el puntaje y el historial de colores usados. Además, actualiza la pantalla para reflejar el reinicio.

(d) `parpadeoConCambioDeColor(int delayTime)`

- Controla el parpadeo de los LEDs y la interacción del usuario:
  - Los LEDs cambian de color aleatoriamente con un retardo aleatorio entre 200 y 600 ms.
  - Los botones táctiles (`TOUCH0` a `TOUCH4`) permiten alternar el estado de "detenido" de cada LED.
  - Se verifica si las condiciones para sumar puntos se cumplen mediante una llamada a `verificarColoresYSumarPuntos()`.
  - El índice de colores se actualiza cíclicamente para alternar entre los colores definidos.

4. **Flujo de ejecución**

- En el `setup()`, el sistema se inicializa y muestra el puntaje inicial en la pantalla.
- En el `loop()`, el juego se gestiona mediante la interacción del usuario y el ciclo de parpadeo de LEDs.
- Si el puntaje llega a 3, se activa la animación de victoria y luego se reinicia el juego.

# Actividad 2

Para esta actividad, se utilizaron la documentación oficial y los ejemplos proporcionados para cada sensor del MKR IoT Carrier como referencia, logrando implementar una función dedicada para cada sensor. Además de aplicar los conceptos vistos en la actividad 1, se añadió una visualización específica para cada tipo de lectura en el display, lo que permitió diferenciar claramente la información de cada

sensor. Se trabajó con la librería #include Arduino_MKRIoTCarrier.h, la cual facilitó la interacción con los componentes del Carrier, y se estableció la comunicación serial para depuración y registro de datos. Se desarrolló una función principal de menú que organiza las opciones disponibles, mostrando las alternativas al usuario. Según el botón del touchpad seleccionado, esta función llama a la rutina correspondiente al sensor deseado, permitiendo un flujo claro y estructurado. Este enfoque estableció la base funcional del sistema, quedando únicamente pendientes ajustes y optimizaciones. Durante este proceso, se desarrollaron dos versiones del código, detalladas a continuación:

## Versión 1

En esta etapa inicial, el código presentaba varias deficiencias. Uno de los principales problemas era el parpadeo constante de la pantalla, lo cual generaba una experiencia visual incómoda. Además, la detección de los touchpads no era precisa, lo que requería múltiples intentos para registrar una acción. Una vez detectada la entrada, el sistema mostraba la medición del sensor correspondiente de manera muy breve, regresando rápidamente al menú sin dar tiempo al usuario para analizar los datos. Por otro lado, las mediciones eran estáticas: al seleccionar un sensor, se realizaba una única lectura que no se actualizaba de forma continua mientras se visualizaba en pantalla.

## Versión 2

Esta versión representa un avance significativo. Se solucionó el problema del parpadeo en el menú, logrando una experiencia mucho más estable y fluida. Ahora, al presionar cualquiera de los touchpads, se muestra correctamente la información del sensor seleccionado, con un diseño más claro y estético. Además, el dato permanece visible en pantalla hasta que se presiona otro botón, lo que permite al usuario visualizarlo con calma antes de regresar al menú. Sin embargo, un detalle por mejorar es que las mediciones siguen siendo estáticas: mientras se muestra la lectura de un sensor, no se actualiza de forma continua, lo que genera una sensación de "congelamiento" de la información hasta que se regresa al menú, donde las lecturas vuelven a reactivarse.

## Tabla comparativa

| **Aspecto Evaluado** | **Versión 1** | **Versión 2** |
|---|---|---|

| | | |
|---|---|---|
| Parpadeo en pantalla | La pantalla parpadea constantemente debido a una llamada recurrente a fillScreen (ST77XX_BLACK) en cada ciclo del menú principal. | Se eliminó el parpadeo al reorganizar el flujo en la función menu(), evitando que la pantalla se limpie continuamente. Ahora, solo se llama a fillScreen() cuando es necesario actualizar el contenido. |
| Detección de botones táctiles | La detección de los touchpads es imprecisa, probablemente por la falta de un mecanismo de antirrebote. | Se implementó un antirrebote con un retraso de 50 ms utilizando las variables lastDebounceTime y debounceDelay en las funciones de manejo de botones (esperarBoton y menu). Esto asegura que las acciones de los touchpads se registren de forma más confiable. |
| Transición al menú principal | Después de mostrar un sensor, la pantalla vuelve automáticamente al menú principal debido a una falta de persistencia en la visualización de datos. | Se corrigió este comportamiento al introducir un ciclo while (true) dentro de cada función de sensor (por ejemplo, temperatura(), humedad(), etc.), permitiendo que los datos permanezcan en pantalla hasta que se presione un botón diferente. |

| | | |
|---|---|---|
| Medición continua del sensor | Solo se toma una medición al seleccionar un sensor, ya que no se actualizan los valores dentro del ciclo de visualización. | Aunque las mediciones se mantienen visibles, las funciones como carrier.Env.readTemperature() solo se ejecutan una vez dentro del ciclo while (true). Es necesario agregar un bucle con actualizaciones continuas para cada lectura. |
| Estética general del menú | La presentación es básica, con pocas mejoras visuales, lo que dificulta la identificación rápida de las opciones. | Se mejoró la estética al utilizar funciones como carrier.display.setCursor() y diferentes colores (ST77XX_WHITE, ST77XX_RED, etc.) para destacar datos relevantes. Además, las opciones del menú ahora están más organizadas en secciones claras. |
| Claridad de los datos mostrados | Los datos de los sensores están desorganizados, ya que se muestran como texto plano, lo que dificulta su lectura. | La información ahora se organiza en bloques diferenciados. Sin embargo, podría implementarse un formato en tabla usando un diseño más estructurado con setCursor() para alinear valores de manera uniforme (especialmente para sensores como CO2, temperatura y humedad). |

**Detalles técnicos específicos del código que evolucionaron entre versiones**

- **Implementación del antirrebote:** Se agregaron las variables:

```
unsigned long debounceDelay = 50;  // Retardo para el antirrebote
unsigned long lastDebounceTime = 0;
```

En las funciones menu() y esperarBoton(), se verificó el tiempo transcurrido con:

```
if (millis() - lastDebounceTime > debounceDelay) {
    if (carrier.Buttons.onTouchUp(TOUCH0)) { /* Acción */ }
    lastDebounceTime = millis();
}
```

Esto previene que se registren múltiples pulsaciones de un solo toque.

- **Persistencia de datos en pantalla:** Se reorganizó la lógica de cada sensor con un ciclo while (true) para mantener los datos visibles:

```
while (true) {
    float temperature = carrier.Env.readTemperature();
    carrier.display.fillScreen(ST77XX_BLACK);
    carrier.display.setCursor(70, 40);
    carrier.display.println("TEMPERATURA:");
    // Más código de impresión
    esperarBoton(TOUCH0);
    break; // Salir del ciclo
}
```

## Estructura del programa

1. **Configuración de hardware (inicialización de componentes)** El bloque de configuración se encuentra en la función setup(). Aquí se inicializan los componentes principales del hardware:
    - **Comunicación serial:** Se establece con Serial.begin(9600) para poder depurar el programa y monitorear los valores de los sensores.
    - **Inicialización del objeto Carrier:** Se utiliza carrier.begin() para iniciar la comunicación y control de los diversos componentes del MKR IoT Carrier.
    - **No usar mayúsculas en la pantalla:** carrier.noCase() asegura que la pantalla no muestre caracteres en mayúsculas.
    - **Retraso inicial:** delay(1000) se usa para asegurar que el Carrier esté completamente inicializado antes de continuar.
    - **Impresión de cabecera en la consola:** Se imprime el encabezado de los datos que se van a mostrar en la consola: "Temperature(C)(KPa)2(ppm)".
2. **Ciclo principal (loop) con lógica del menú** El núcleo del programa está en la función loop(). Este bloque gestiona el menú y las interacciones del usuario:
    - **La función menu() se ejecuta en el ciclo principal** para presentar opciones al usuario.

- **El menú permite acceder a diferentes funciones** que muestran información relacionada con temperatura, humedad, presión, gases y giroscopio.
- **Cuando se selecciona una opción** (presionando los botones táctiles), se llama a la función correspondiente para mostrar la información del sensor.

3. **Funciones auxiliares** Estas funciones se encargan de modularizar el código y realizar tareas específicas:

   (a) `esperarBoton(int currentButton)`

   Esta función se encarga de esperar que el usuario presione un botón táctil diferente al actual, manejando el antirrebote:

   - **Antirrebote**: Se asegura de que no haya rebotes de los botones, usando un retraso controlado por la variable `debounceDelay`.
   - **Llamada al menú**: Una vez que el usuario presiona un botón, se llama a la función `menu()` para mostrar las opciones de nuevo.

   (b) `temperatura()`

   Esta función muestra la temperatura de varios tipos (Celsius, Fahrenheit y Kelvin) en la pantalla del Carrier:

   - **Lectura de temperatura**: Se lee la temperatura usando `carrier.Env.readTempera`
   - **Conversión de unidades**: Se convierte la temperatura de Celsius a Fahrenheit y Kelvin.
   - **Actualización de pantalla**: La pantalla se actualiza para mostrar los valores de temperatura en el formato adecuado.
   - **Esperar entrada del usuario**: Luego, espera a que el usuario presione el botón asociado a la opción de temperatura (`TOUCH0`) para regresar al menú.

   (c) `humedad()`

   Esta función muestra el valor de la humedad:

   - **Lectura de humedad**: Se lee la humedad utilizando `carrier.Env.readHumidity()`.
   - **Actualización de pantalla**: Se actualiza la pantalla para mostrar el valor de la humedad en porcentaje.
   - **Esperar entrada del usuario**: Luego, espera a que el usuario presione el botón asociado a la opción de humedad (`TOUCH1`) para regresar al menú.

   (d) `presion()`

   Esta función muestra la presión atmosférica:

   - **Lectura de presión**: La presión se lee utilizando `carrier.Pressure.readPressure(`
     y se multiplica por 10 para ajustar la unidad a hPa.

- **Actualización de pantalla**: La pantalla se actualiza para mostrar el valor de la presión.
- **Esperar entrada del usuario**: Luego, espera a que el usuario presione el botón correspondiente (`TOUCH2`) para regresar al menú.

(e) `gases()`

Esta función muestra información sobre los gases presentes:

- **Lectura de gases**: Se leen valores de CO2, resistencia del gas y compuestos orgánicos volátiles utilizando el objeto `carrier.AirQuality`.
- **Actualización de pantalla**: La pantalla se actualiza con los valores leídos.
- **Esperar entrada del usuario**: Luego, espera a que el usuario presione el botón correspondiente (`TOUCH3`) para regresar al menú.

(f) `giroscopio()`

Esta función muestra los valores del giroscopio (ejes X, Y, Z):

- **Lectura del giroscopio**: Se lee el valor del giroscopio usando `carrier.IMUmodule.re`
  `y, z)`.
- **Mostrar valores**: Los valores de cada eje se muestran en la pantalla.
- **Esperar entrada del usuario**: Luego, espera a que el usuario presione el botón correspondiente (`TOUCH4`) para regresar al menú.

(g) `menu()`

Esta función presenta el menú de opciones en la pantalla y maneja las entradas del usuario:

- **Mostrar menú**: Se muestra un menú con las opciones disponibles (Temperatura, Humedad, Presión, Gases, Giroscopio).
- **Ciclo de espera**: El ciclo `while` dentro de `menu()` mantiene el programa esperando la entrada del usuario.
- **Llamada a funciones**: Dependiendo del botón presionado, se llama a la función correspondiente (por ejemplo, `temperatura()`, `humedad()`, etc.).
- **Antirrebote**: La función maneja el antirrebote de los botones para evitar acciones duplicadas.

# Actividad 3

Utilizando los ejemplos de cada sensor, del display e leds, inicie este código, además de ya haber experimentado un poco con esta interfaz en actividades anteriores, comencé con crear diferentes funciones, una para cada acción, que era la de proximidad y detecccion de color, también una para mostrar las diferentes direcciones que se mostraran por un diferente gesto seleccionado.

En la función de proximidad se copio y pego el ejemplo propuesto en la documentación, esto dentro de un ciclo while para que se ejecutara de forma continua,

también se agrego la forma en que se desplegaría en el display e las condicionales para activar el buzzer y el color del led, estos varían dependiendo a la distancia detectada.

A la función de detectar blanco, de manera similar, se copio y pego el ejemplo propuesto dentro de un ciclo while, en este se implemento un display para que mostrara los valores detectados de RGB e se ñadieron dos estados, uno donde se detectaba el color blanco, en otro donde no se detectaba.

De manera similar el menú se estableció mediante una función y solo se dejo el código de la detección de gestos que igual se copio dentro del loop, para que se ejecutara de forma continua dentro de este. Se añadieron los antirebotes correspondientes y la lógica de la interfaz, sin embargo al tratar de ejecutar esto, se identifico que no funcionaba como debía, así que se probo cada función de forma individual en otro sketch para verificar que no estuviera el error en estas, de esta manera se corroboro que no era el caso.

Al final se opto por colocar todas estas funciones dentro del loop, ajustando solo un poco el código, pero prácticamente se copio y se pego. Para mostrar de forma individual cada caso se utilizaron bandera para verificar si era verdadero o falso el caso, estos estados era el de mostrar en el display la lectura de proximidad y detectar color. Además se implemento un pequeño truco utilizando el cambio de color del display y cambiando la posición dentro de este, para que la interfaz pareciera que solo cambia cuando se acciona el botón, aunque en realidad este corriendo el programa de detección de gestos todo el tiempo.

## Versión 1

El touchpad tiene una funcionalidad parcial. Si bien funciona correctamente para la detección de proximidad, en la detección de color requiere múltiples presiones para salir del modo, y solo permite realizar una medición. El código utiliza funciones específicas para el menú, la proximidad (aunque los LEDs aún no se encienden) y la detección de color. Por otro lado, la detección de gestos está implementada directamente en el loop, para que se mantenga activa en todo momento. Sin embargo, esta detección no está completamente afinada, ya que puede interpretar gestos de manera errónea.

## Versión 2

Se ha implementado el encendido de los LEDs según la distancia detectada en la función de proximidad, pero al salir de la función, los LEDs permanecen encendidos, lo que no está correctamente manejado. En la detección de color, ahora se muestran en la pantalla los valores RGB detectados, y la funcionalidad del touchpad ha mejorado notablemente. A pesar de esto, la lectura sigue sin ser continua.

## Versión 3

En esta versión, se observa una mejora significativa. Se ha eliminado el uso de funciones específicas, y el código se ejecuta casi en su totalidad dentro del loop. Esto implica que los tres sensores se ejecutan de manera continua, pero lo que se muestra en la pantalla depende de condicionales if y el uso de banderas o variables

booleanas. Esto permite activar y visualizar en el display únicamente la información necesaria en cada momento, optimizando el flujo del programa.

## Tabla comparativa

| Aspecto | Versión 1 | Versión 2 | Versión 3 (Final) |
|---|---|---|---|
| **Estructura general** | Tiene duplicación de código y funciones poco organizadas, como `loop()` y `menu()` que repiten lógica. | Mejora la organización, pero todavía incluye fragmentos redundantes. | Diseño claro y compacto, con todo el control en el `loop()` principal y funciones bien distribuidas para tareas específicas. |
| **Gestión de gestos** | Maneja los gestos de manera dispersa y poco eficiente, procesándolos en varias partes del código. | Consolida la gestión en `menu()`, pero aún tiene complejidad innecesaria. | Gestos procesados de forma centralizada con constantes (`GESTURE_LEFT`, etc.) y lógica clara, lo que facilita su extensión. |
| **Control de botones táctiles** | Procesa eventos táctiles de manera separada y no reutiliza código, complicando el mantenimiento. | Simplifica la lógica, pero sigue manejando botones de forma secundaria. | Prioriza gestos sobre botones, reduciendo la complejidad e incrementando la eficiencia del código. |
| **Funciones gráficas** | Muy detalladas, pero con densidad visual excesiva que puede confundir al usuario. | Más claras que en la Versión 1, usando colores para transmitir información útil (RGB, proximidad). | Visualización reducida y centrada en lo esencial, manteniendo claridad sin sobrecargar la interfaz. |

| Aspecto | Versión 1 | Versión 2 | Versión 3 (Final) |
|---|---|---|---|
| **Proximidad** | Activa buzzer según la distancia, pero no cambia. | Solo tiene una lectura. | Activa LEDs y buzzer según la distancia, pero no siempre apaga los componentes correctamente al finalizar. Control optimizado: las mediciones son continuas y se activa con el `TOUCH4`. |
| **Manejo de color blanco** | Lógica funcional, pero la visualización es limitada y no es intuitiva para el usuario. | Muestra valores RGB continuamente, lo que mejora la interacción. | Mediciones continuas. Gestión simplificada: detecta blanco con mensajes claros, sin sobrecargar la pantalla con datos innecesarios. |
| **Código modular** | Contiene redundancia, especialmente en lógica de gestos y botones. | Modularidad mejorada, aunque no completamente optimizada. | Código modular y minimalista, asegurando que cada función cumple un propósito claro sin redundancias. |
| **Interacción con usuario** | Interfaz visual rica pero confusa por la cantidad de elementos gráficos. | Más clara y efectiva al utilizar colores y mensajes para guiar al usuario. | Interacción directa y funcional: mensajes claros y un diseño minimalista que prioriza usabilidad sobre elementos decorativos. |

| Aspecto | Versión 1 | Versión 2 | Versión 3 (Final) |
|---|---|---|---|
| **Manejo de buzzer y LEDs** | Complejo, con lógica dispersa y solo activa el buzzer. | Complejo, con lógica dispersa y apagado de LEDs/buzzer no garantizado en todas las situaciones. | LEDs y buzzer gestionados de forma sencilla y eficiente, activados solo cuando son necesarios y desactivados correctamente. |
| **Robustez y errores** | Falta manejo adecuado de errores como el fallo de `carrier.begin()`. | Agrega manejo de errores en `setup()` para evitar bucles infinitos en caso de fallos. | Mantenimiento del manejo de errores, asegurando robustez desde el inicio. |

## Estructura de programa

1. **Configuración de hardware (inicialización de componentes)**
   El bloque de configuración se encuentra en la función setup(). Aquí se inicializan los principales componentes del hardware del MKR IoT Carrier:

   - **Comunicación serial**: Configurada con Serial.begin(9600) para depurar y monitorear los valores del programa.
   - **Inicialización del objeto Carrier**: Se usa carrier.begin() para iniciar la comunicación y los controles del hardware asociado al MKR IoT Carrier. Si falla, el programa imprime "Error" y se detiene.
   - **Configuración de pantalla**: carrier.noCase() asegura que la pantalla no muestre caracteres en mayúsculas.
   - **Variables de estado inicializadas**: Variables como r, g, b para los colores, así como los valores relacionados con la proximidad y el control de rebotes (debounceDelay).

2. **Ciclo principal (loop) con lógica central**
   El núcleo del programa se encuentra en la función loop(). Este bloque gestiona el procesamiento de gestos, la lectura de colores y proximidad, y las interacciones con los botones táctiles:

   - **Pantalla inicial**: Se muestra un mensaje en pantalla indicando que se está midiendo movimiento.
   - **Procesamiento de gestos**:
     - Si se detecta un gesto (arriba, abajo, izquierda o derecha), se imprime un mensaje en la consola y se actualiza la pantalla con un gráfico representativo del gesto detectado.
   - **Medición de colores**:

- Si los colores están disponibles, se leen con carrier.Light.readColor(r, g, b).

- **Control de botones táctiles**:
  - Botón TOUCH2: Activa o desactiva la medición de colores.
  - Botón TOUCH4: Activa o desactiva la medición de proximidad. También apaga los LEDs y borra la pantalla si se desactiva.
- **Medición de proximidad**:
  - Si está activada, lee la distancia del sensor de proximidad y actualiza la pantalla y los LEDs según el valor detectado:
    - Verde si es mayor o igual a 200.
    - Amarillo entre 120 y 199.
    - Rojo si es menor a 120.
  - También utiliza el buzzer para emitir diferentes tonos según la distancia.
- **Medición de colores**:
  - Si está activada, muestra los valores RGB en pantalla y detecta si el color predominante es blanco.

3. **Funciones auxiliares**
   Para mantener el código organizado, estas funciones realizan tareas específicas dentro del programa:

   (a) `gestionarGestos()`

   Esta función procesa los gestos detectados por el sensor de luz:
   - Detecta gestos como GESTURE_UP, GESTURE_DOWN, GESTURE_LEFT, y GESTURE_RIGHT.
   - Actualiza la pantalla con un gráfico según el gesto detectado.

   (b) `medirProximidad()`

   Realiza la lectura y procesamiento de la proximidad:
   - Actualiza la pantalla y enciende un LED según la distancia detectada.
   - Emite tonos con el buzzer según el rango de la proximidad.

   (c) `medirColor()`

   Muestra en pantalla los valores de colores (R, G, B):
   - Detecta si el color predominante es blanco e imprime un mensaje en pantalla indicando la detección.

   (d) `gestionarBotones()`

   Controla las acciones asociadas a los botones táctiles:
   - Alterna entre la medición de colores y la proximidad dependiendo del botón presionado.

# Actividad 4

A partir de los dos ejemplos proporcionados en la documentación, se implementó la comunicación por WiFi y BLE de manera independiente, para este último se utilizó la app de nRF Connect como dispositivo central. Tras confirmar que ambas funcionaban correctamente, se integraron los códigos y se desarrolló un menú para gestionar estas opciones, agregando detalles adicionales como el nombre en el monitor serial y otros ajustes estéticos. En general, no hubo complicaciones en la creación del código, ya que se partió de ejemplos funcionales.

Se diseñaron dos funciones principales para gestionar las conexiones WiFi y BLE, y otras dos para administrar el menú y permitir la selección continua. Al principio, la opción de escanear redes WiFi funcionaba correctamente, pero al seleccionar primero la opción BLE, el sistema dejaba de responder. Tras investigar, se identificaron dos posibles causas: el Arduino WiFi 1010 comparte el mismo bus para ambas comunicaciones y se requería más tiempo para estabilizar los datos. Para solucionarlo, se implementó una función de reinicio que restablecía los módulos WiFiNINA y BLE al salir de cada opción, además de pequeños retrasos para mejorar la estabilidad del sistema.

# Estructura del programa

1. **Configuración de hardware (inicialización de componentes)**
   En la función `setup()`, se realiza la inicialización de los componentes principales para la comunicación serial, el WiFi y la conexión BLE:

   - **Comunicación serial**: Se establece con `Serial.begin(9600)` para permitir la depuración y monitoreo de los valores.
   - **Inicialización del WiFi**: Se comprueba el estado del módulo WiFi con `WiFi.status()`. Si hay un fallo en la comunicación, el sistema se detiene.
   - **Verificación de versión de firmware WiFi**: Se compara la versión del firmware del WiFi con la versión más reciente, y se informa si es necesario actualizarlo.
   - **Menú inicial**: Se imprime un menú de opciones para interactuar con el usuario, permitiendo elegir entre BLE, WiFi o salir del sistema.

2. **Ciclo principal (`loop`) con lógica del menú**
   El bloque principal del programa se encuentra en la función `loop()`, que gestiona la interacción del usuario:

   - **Manejo de entradas del usuario**: Se espera que el usuario ingrese una opción válida (`'1'`, `'2'` o `'s'`). Si la opción es válida, se procesa la acción correspondiente (conectar por BLE, conectar a WiFi o salir).
   - **Repetir el menú**: Después de procesar una opción, se vuelve a mostrar el menú para permitir al usuario seleccionar otra opción.

3. **Funciones auxiliares**
   Estas funciones realizan tareas específicas y ayudan a modularizar el código:

(a) `listNetworks()`

- Escanear redes Wi-Fi: Esta función escanea las redes Wi-Fi disponibles y muestra la lista en la consola. Si no se puede obtener una conexión Wi-Fi, se informa al usuario.

(b) `connectToNetwork()`

- Conexión a una red Wi-Fi: Intenta conectar el dispositivo a la red Wi-Fi seleccionada, utilizando el SSID y la contraseña proporcionados por el usuario. Si la conexión es exitosa, se muestra la dirección IP y se espera que el usuario desconecte.

(c) `printWiFiStatus()`

- Mostrar estado Wi-Fi: Imprime la dirección IP obtenida después de una conexión exitosa a la red Wi-Fi.

(d) `printMacAddress(byte mac[])`

- Mostrar dirección MAC: Imprime la dirección MAC del dispositivo Wi-Fi.

(e) `initializeBLE()`

- Inicialización de BLE: Inicializa el módulo BLE, configura los servicios y las características, y comienza a publicitar el dispositivo BLE. Si el dispositivo se conecta o desconecta, se gestionan los eventos correspondientes mediante las funciones `onBLEConnected()` y `onBLEDisconnected()`. Durante el proceso, el programa espera la entrada del usuario para desconectar manualmente BLE y volver al menú.

(f) `resetModule()`

- Reinicio de módulos: Desactiva tanto el Wi-Fi como el BLE, con un pequeño retraso para estabilizar el hardware antes de continuar.

(g) `wifi()`

- Interfaz de conexión Wi-Fi: Permite al usuario escanear redes Wi-Fi, seleccionar una red, introducir la contraseña y conectar. Si se selecciona una opción inválida, se permite al usuario intentar de nuevo o regresar al menú.

(h) `showMenu()`

- Mostrar menú: Muestra el menú de opciones disponibles (conectar por BLE, conectar Wi-Fi o salir del sistema).

(i) `handleMenuOption(char option)`

- Procesar opción del menú: Según la opción seleccionada por el usuario (`'1'`, `'2'` o `'s'`), se invoca la función correspondiente: `initializeBLE()` para BLE, `wifi()` para Wi-Fi, o salir del sistema.

4. **Flujo de ejecución**

- En el `setup()`, el sistema se inicializa y muestra el menú.
- En el `loop()`, se maneja la entrada del usuario para elegir entre las opciones disponibles.
- Dependiendo de la opción seleccionada, el flujo pasa a conectar por BLE, conectar a Wi-Fi o salir.

# Actividad 5

Con el propósito de realizar la actividad, se inició el proceso con el ejemplo de la documentación del Carrier, en el cual se usan las dos entradas (`A0` y `A6`), previamente ya dispuestas y mediante conexiones keying. Se conectaron los dos sensores deseados: el sensor de humedad del suelo (pin analógico `A0`) y el sensor de movimiento PIR (pin digital `A6`). Con esto se corrió el programa y se verificó el funcionamiento correcto.

Posterior a ello, se comenzó con la calibración de los sensores, de tal manera que sea óptima para realizar la actividad deseada. En el caso del PIR, resultó un poco tedioso y más difícil, puesto que se tenía que calibrar tanto la sensibilidad como el retardo en el tiempo, esto usando un destornillador para poder ajustar sus dos potenciómetros integrados en este circuito.

En el caso del sensor de humedad fue más fácil, puesto que solo se tuvieron que realizar diferentes pruebas en distintas condiciones, de manera iterativa, observando cómo variaban los valores obtenidos. Una vez vista la variación aproximada entre el estado seco y el estado completamente mojado, estos valores se ingresaron en las variables correspondientes para que, mediante un mapeo, se realizara la conversión de valores de entrada a porcentaje de humedad.

Habiendo hecho esto, el siguiente paso fue crear dos funciones, una por sensor, en las cuales se ejecutaba de manera continua dentro de un bucle `while` la lectura del sensor y se mostraba tanto en el display como en el monitor serial. Posteriormente, se estableció un menú dentro del `loop`, seguido de un ciclo `while` donde se encapsulaban la selección de las opciones, ya sea mediante el ingreso de números con comunicación serial o la interrupción de touchpads en el Carrier.

Se incluyeron, en cada uso de los touchpads, un antirrebote, además de incluir funciones relacionadas con `millis()`, para sustituir el uso de `delay()` y darle más fluidez al programa. A continuación se presentan las dos versiones realizadas durante esta actividad:

## Versión 1

El código implementa un menú con tres opciones: dos para seleccionar el sensor a monitorear (humedad del suelo o proximidad) y una más para salir del sistema. La comunicación y el control se realizan a través de la interfaz serial, donde el usuario

debe ingresar `1`, `2` o `3` para seleccionar una opción, mientras que la tecla `r` permite regresar al menú principal.

Los datos de los sensores también se muestran en el display del Carrier. Sin embargo, al regresar al menú, el tiempo de respuesta varía según el sensor en uso. En el caso del sensor de humedad del suelo, existe un retardo de 10 segundos debido al uso de `delay()`, mientras que el sensor de proximidad solo introduce un retardo de 2 segundos. Además, el sistema incluye la conversión de las lecturas del sensor de humedad a porcentajes, y se realizó una calibración del sensor PIR mediante sus potenciómetros.

En resumen, esta versión cumple con su funcionalidad básica, aunque presenta algunos puntos a mejorar: el uso de retardos bloqueantes afecta la eficiencia y la dependencia exclusiva de la interfaz serial limita su flexibilidad.

## Versión 2

Esta versión mantiene la estructura y funcionalidades del código original, pero introduce diversas mejoras significativas:

1. **Calibración del sensor de humedad:** Se realizaron pruebas bajo diferentes condiciones, lo que permitió reducir el margen de error en las lecturas y mejorar la precisión.

2. **Control mediante touchpads:** Se habilitaron los touchpads del Carrier (`TOUCH1`, `TOUCH2` y `TOUCH3`). El tercer touchpad permite regresar al menú de forma más intuitiva.

3. **Optimización del tiempo de respuesta:** Se reemplazó el uso de `delay()` por funciones basadas en `millis()`, lo que permite un manejo no bloqueante del tiempo. Gracias a esto, la opción de regresar al menú ahora es casi instantánea, sin necesidad de esperar a que termine el ciclo del sensor.

En esta versión se logró una mejor experiencia de usuario, con controles más eficientes y tiempos de respuesta significativamente reducidos.

## Tabla comparativa

| **Aspecto** | **Versión 1** | **Versión 2** |
|---|---|---|
| Arquitectura del Menú | Interfaz de menú básico mediante comunicación serial con tres opciones: sensor de humedad, sensor PIR y salida. | Menú interactivo con soporte para botones táctiles del Carrier además de la comunicación serial. Mejora la experiencia del usuario. |

| | | |
|---|---|---|
| Retorno al Menú | Uso de `delay()` para controlar el tiempo entre lecturas: 2 segundos para el PIR y 10 segundos para el sensor de humedad. La salida requiere esperar la finalización del ciclo del `delay()`. | Implementación de temporización no bloqueante con `millis()`, lo que permite capturar eventos como salida sin esperar la ejecución completa del ciclo de lecturas, mejorando la eficiencia del sistema. |
| Medición de Humedad | Valores iniciales del sensor definidos como 0 para seco y 1023 para húmedo. | Valores ajustados (924 seco y 320 húmedo) con calibración basada en pruebas empíricas, mejorando la precisión de la lectura. |
| Visualización | Información mostrada en el display del Carrier y por comunicación serial. | Se mantiene la visualización en el display del Carrier y serial, pero con mensajes más claros y precisos. |
| Control del Sensor PIR | Captura y visualización del estado del sensor de movimiento con opción de salida mediante serial (`r`). | Captura continua del estado del PIR usando `millis()`. Admite salida con serial (`r`) o mediante botón táctil (`TOUCH3`). |
| Interacción Táctil | No implementada. | Soporte para tres botones táctiles del Carrier: `TOUCH1` para humedad, `TOUCH2` para movimiento y `TOUCH3` para salir. |
| Antirrebote | No implementado. | Implementación básica de antirrebote con `debounceDelay`. |
| Eficiencia del Código | Uso de `delay()` bloqueante que interfiere con la capacidad de respuesta. | Mejora significativa mediante temporización no bloqueante con `millis()`. |

# Estructura del programa

1. **Configuración del hardware (`setup()`)**
   - `Serial.begin(9600)`: Habilita la comunicación serial para monitoreo de datos.
   - Configuración de pines:
     - `pinMode(pir, INPUT)` para el sensor PIR.
     - `pinMode(sensorPin, INPUT)` para el sensor de humedad.

- Inicialización del Carrier (`carrier.begin()`): Permite manejar su pantalla y botones.
- Retardo inicial (`delay(1000)`): Asegura la correcta inicialización del Carrier.

2. **Lógica del menú (`loop()`)**
   El menú presenta tres opciones:

   - Activar el sensor de humedad del suelo.
   - Activar el sensor de movimiento.
   - Salir del programa.

   Control de entrada:

   - Botones táctiles del Carrier (`TOUCH1`, `TOUCH2`, `TOUCH3`).
   - Control por consola serial: Permite seleccionar opciones usando entradas numéricas.

3. **Funciones auxiliares**

   (a) `senPir()`
   - Muestra el estado del sensor PIR en la pantalla.
   - Si detecta movimiento, imprime `"SI"` y muestra `"Moviéndose..."` en pantalla; si no, muestra `"Tranquilo..."`.
   - Usa antirrebote mediante `debounceDelay`.
   - Permite salir del modo con botón táctil o comando por Serial.

   (b) `senHumedad()`
   - Lee el sensor de humedad, convierte su valor en porcentaje (`map`) y lo ajusta a 0-100% (`constrain`).
   - Muestra el porcentaje de humedad en la pantalla del Carrier.
   - Permite salir del modo mediante botones táctiles o comandos por Serial.

   (c) `menu()` integrado en `loop()`
   - Presenta las opciones en pantalla y mantiene el control mediante botones táctiles.
   - Imprime el menú en Serial, lo cual es útil para monitoreo remoto.

# ANÁLISIS TÉCNICO GLOBAL

## Limitaciones y Problemas Identificados

### Limitaciones de Hardware

1. **Sensibilidad de los touchpads:** La variabilidad en la detección de movimientos debido a la presencia o ausencia de la carcasa afecta la precisión de las interacciones.

2. **Display parcialmente dañado:** La configuración manual de coordenadas (`x`, `y`) para el display resulta tediosa, especialmente con cambios de tamaño de fuente.

3. **Daño del bootloader:** Durante las pruebas, el microcontrolador Arduino WiFi 1010 dejó de ser reconocido por la laptop, mostrando un error de "dispositivo no identificado".

4. **Calibración de sensores:** El proceso de calibración, especialmente para el sensor PIR, es repetitivo y complicado debido a su alta sensibilidad.

5. **Comportamiento inverso del sensor de humedad:** Los valores registrados por el sensor de humedad son inversos a lo esperado (menor valor indica mayor humedad y viceversa).

### Limitaciones de Software

1. **Fijación y desfijación manual de LEDs:** La necesidad de realizar estas acciones manualmente en las tres versiones del código hace que el juego sea repetitivo y poco intuitivo.

2. **Falta de apagado automático:** No existe un mecanismo para apagar o reiniciar el juego automáticamente después de un tiempo o al completarlo.

3. **Monotonía en la interacción:** Aunque la versión 3 añade complejidad, el núcleo del juego sigue siendo similar, lo que limita su rejugabilidad.

4. **Lecturas en tiempo real:** En las actividades 2 y 3, las lecturas de los sensores no se actualizan continuamente cuando se utilizan funciones, lo que dificulta la visualización de datos en tiempo real.

5. **Proceso bloqueante con `delay()`:** El uso de la función `delay()` en la actividad 5 impide la salida inmediata del ciclo, generando una experiencia de usuario poco fluida.

6. **Congelamiento de WiFi:** Al cambiar de BLE a WiFi, el sistema se congela debido a un manejo inadecuado de las operaciones de comunicación.

## Soluciones Implementadas

### Soluciones de Hardware

1. **Reinicio de la placa:** El problema del bootloader se resolvió presionando dos veces el botón de reinicio inmediatamente después de conectar el USB al computador.

2. **Mejora en la detección de gestos:** Se ajustó la dirección de los gestos detectados mediante un proceso iterativo de verificación y corrección.

### Soluciones de Software

1. **Automatización de la fijación/desfijación de LEDs:** Se implementó un sistema que permite fijar automáticamente los LEDs al presionar un botón, eliminando la necesidad de desfijarlos manualmente.

2. **Apagado de LEDs:** Se incluyó la función `carrier.leds.show()` después de limpiar los LEDs para asegurar su apagado correcto.

3. **Lectura continua de sensores:** En la actividad 5, se reemplazó el uso de `delay()` con funciones basadas en `millis()` para permitir lecturas continuas y evitar bloqueos.

4. **Reinicio de comunicaciones:** Se implementó una función de reinicio que cierra correctamente las operaciones de comunicación antes de cambiar entre BLE y WiFi, resolviendo el congelamiento del sistema.

5. **Corrección del sensor de humedad:** Se ajustó la lógica del código para interpretar correctamente los valores del sensor de humedad.

6. **Menú funcional en el `loop`:** En la Actividad 5, se implementó un menú dentro del `loop()` que permite la lectura continua de sensores, resolviendo problemas de bloqueo y mejorando la experiencia del usuario (Leer Nota extra).

## Propuestas de Mejora

### Mejoras Potenciales de Hardware

1. **Cambio o optimización del display:** Sustituir el display actual o desarrollar una función que automatice los saltos de línea para mejorar la visualización.

2. **Mejora en la sensibilidad de los touchpads:** Implementar un sistema de antirrebote en el código para reducir las activaciones no deseadas.

### Mejoras Potenciales de Software

1. **Incremento de la dificultad progresiva:** Introducir niveles de dificultad en el juego, como parpadeos más rápidos o una mayor variedad de colores.

2. **Retroalimentación visual y sonora:** Añadir animaciones en la pantalla y sonidos al interactuar con los botones o alcanzar metas.

3. **Actualización continua de mediciones:** Implementar un sistema que permita la lectura continua de los sensores sin necesidad de salir y volver al menú (Leer Nota extra).

4. **Modularidad del código:** Separar la lógica del menú y la lectura de botones en funciones independientes para mejorar la organización y mantenibilidad del código.

5. **Base de datos para registros:** Implementar una base de datos para almacenar y visualizar los datos de los sensores en tiempo real, facilitando su análisis.

6. **Validación de entradas:** Añadir validaciones para evitar valores fuera de rango en las entradas del usuario.

**Optimizaciones Propuestas**

1. **Uso de bibliotecas especializadas:** Emplear bibliotecas como `Bounce2` para manejar el antirrebote de botones de manera más eficiente.

2. **Optimización del uso de recursos:** Reducir el consumo de recursos al evitar que los sensores operen continuamente cuando no es necesario.

3. **Pruebas iterativas:** Realizar pruebas continuas para identificar y corregir problemas en la detección de gestos y otras funcionalidades.

# Notas Extra de la Actividad 5

En la Actividad 5, se logró un avance notable al implementar un menú funcional que permite la lectura continua y en tiempo real de los sensores. Este éxito se atribuye a las siguientes condiciones y decisiones técnicas:

1. **Menú dentro del `loop`:** A diferencia de actividades anteriores donde el menú se implementaba como una función separada, en esta actividad el menú se desarrolló directamente dentro del `loop()`. Esto permitió una mayor fluidez en la ejecución y evitó problemas de bloqueo o interrupciones en la lectura de los sensores.

2. **Uso de sensores externos conectados directamente a pines del Arduino:** Los sensores utilizados en esta actividad estaban conectados directamente a los pines del Arduino, lo que facilitó su acceso y lectura sin dependencias adicionales de librerías complejas o intermediarios. Esto contribuyó a una mayor eficiencia en la adquisición de datos.

3. **Lógica simple y bien estructurada:** Se implementó una lógica más sencilla y elaborada, lo que permitió una mejor organización del código y una ejecución más eficiente. Esto contrasta con versiones anteriores donde la complejidad del código generaba problemas de rendimiento o bloqueos.

4. **Pruebas iterativas:** Se realizaron pruebas continuas para verificar el funcionamiento del menú y la lectura de los sensores. Este enfoque iterativo permitió identificar y corregir errores de manera rápida, asegurando que el sistema operara de manera óptima.

# RESULTADOS

Como parte de los resultados de la documentación, se logró capturar el funcionamiento de cada una de las 5 actividades mediante videos demostrativos, los cuales permiten

visualizar de manera práctica el comportamiento de la placa y su interacción con los diferentes componentes. Además, se creó un repositorio estructurado que alberga tanto los códigos fuente como las versiones iterativas desarrolladas para cada actividad. Este repositorio está organizado en carpetas numeradas, correspondientes a cada uno de los programas implementados, lo que facilita la consulta y el acceso a las especificaciones técnicas de cada proyecto.

Como valor añadido, se incluyó en el repositorio una versión comentada del código final de cada mini proyecto. Estos comentarios detallan el funcionamiento paso a paso, explicando la lógica implementada y las decisiones técnicas tomadas durante el desarrollo. Esto no solo enriquece la documentación, sino que también sirve como recurso educativo para quienes deseen comprender o replicar las actividades.

En resumen, los resultados obtenidos incluyen:

1. Videos demostrativos del funcionamiento de cada actividad.
2. Un repositorio organizado con los códigos fuente y sus versiones.
3. Códigos comentados que explican de manera detallada el funcionamiento de cada proyecto.
4. Especificaciones técnicas accesibles dentro de las carpetas correspondientes en el repositorio.

Estos recursos están disponibles para su consulta y representan una guía completa para quienes deseen explorar o profundizar en el desarrollo de proyectos con la placa `Carrier Rev 2` y el `Arduino WiFi 1010`.

- **Este es el enlace al** repositorio en GitHub donde se encuentran las actividades desarrolladas para la MKR Carrier Rev2.
- **Este es el enlace directo**: `https://github.com/Higinio2277/MKR_Carrier_Rev2_Actividades_USM`.

# CONCLUSIÓN Y RECOMENDACIONES

La experiencia adquirida durante el trabajo con la placa de desarrollo IoT, específicamente el Arduino WiFi 1010 y el Carrier Rev 2, ha sido sumamente enriquecedora y reveladora. A través de las diversas actividades realizadas, no solo logramos comprender en profundidad las capacidades de esta plataforma, sino que también identificamos aspectos importantes para su optimización.

Particularmente, observamos que existen ciertas limitaciones al trabajar con las funciones relacionadas con los sensores integrados en el Carrier, un aspecto que merece un estudio más detallado para determinar si estas restricciones provienen del hardware, software o de nuestra implementación específica. Las pruebas prácticas realizadas fueron fundamentales para identificar estos desafíos, siendo la lectura en tiempo real uno de los aspectos más destacados que requiere atención especial.

A pesar de estos retos, la placa demostró ser una herramienta versátil y potente, ideal para el desarrollo de proyectos IoT más avanzados. La experiencia nos ha permitido establecer mejores prácticas, como la importancia de realizar pruebas iterativas y mantener una documentación detallada de los problemas encontrados y sus soluciones.

## Recomendaciones

1. **Optimización de lecturas en tiempo real:** Se recomienda implementar el uso de `millis()` en lugar de `delay()`, así como investigar la implementación de interrupciones para mejorar la eficiencia en la adquisición de datos. También sería beneficioso explorar librerías especializadas o desarrollar funciones personalizadas para el manejo de sensores.

2. **Estudio de limitaciones:** Es importante profundizar en el estudio de las limitaciones identificadas con los sensores integrados, realizando pruebas específicas y documentadas que permitan determinar el origen exacto de las restricciones encontradas. Esto facilitará el desarrollo de soluciones más efectivas en futuros proyectos.

3. **Expansión de capacidades:** Para expandir las capacidades del sistema, se sugiere explorar la integración con tecnologías complementarias como la comunicación BLE, el manejo de actuadores más complejos y la implementación de sistemas de monitoreo remoto a través de plataformas en la nube. Estas adiciones permitirían crear aplicaciones más robustas y versátiles.

4. **Enfoque en capacitación y colaboración:** Mirando hacia el futuro, será crucial mantener un enfoque en la capacitación continua y el intercambio de experiencias con otros desarrolladores. La colaboración y el aprendizaje constante nos permitirán no solo superar los desafíos actuales sino también anticipar y resolver los que puedan surgir en proyectos más ambiciosos.

# REFERENCIAS


[1] PerfecXX. (s. f.). GitHub - PerfecXX/Arduino_ExploreIoTKit: Example usage of Arduino IoT Explore Kit. GitHub. Recuperado de `https://github.com/PerfecXX/Arduino_ExploreIoTKit`

[2] mArduino. (n.d.). MKRIoT Carrier Rev 2. Documentación oficial del MKRIoT Carrier Rev 2. Recuperado el 24 de enero de 2025, de `https://docs.arduino.cc/hardware/mkr-iot-carrier-rev2/`

[3] Arduino. (n.d.). Arduino_MKRIoTCarrier: Información sobre las funciones disponibles para interactuar con el hardware. Recuperado el 24 de enero de 2025, de `https://docs.arduino.cc/libraries/arduino_mkriotcarrier/`