UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Industrial

Diseño de una estación meteorológica basada en Arduino con propiedades de tolerancia a fallos en la memoria del microprocesador

Trabajo Fin de Grado

Grado en Ingeniería en Tecnologías Industriales

AUTOR/A: Sammut Rodríguez, Tomás Luc

Tutor/a: Gracia Morán, Joaquín

Cotutor/a: Saiz Adalid, Luis José

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

TRABAJO FIN DE GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

DISEÑO DE UNA ESTACIÓN METEOROLÓGICA BASADA EN ARDUINO CON PROPIEDADES DE TOLERANCIA A FALLOS EN LA MEMORIA DEL MICROPROCESADOR

AUTOR: TOMÁS SAMMUT RODRÍGUEZ

TUTOR: JOAQUÍN GRACIA MORÁN

COTUTOR: LUIS JOSÉ SAIZ ADALID

Curso Académico 2023/2024

AGRADECIMIENTOS

Quiero aprovechar la ocasión para agradecer el apoyo de mis padres sin el cual no hubiera podido cursar esta carrera en Valencia. Gracias por siempre apoyarme en todas mis decisiones y proyectos. Siempre han trabajado muy duro para poder ofrecerme lo mejor y se lo agradezco muchísimo.

Gracias a todos los amigos y compañeros que he hecho en Valencia con los cuales he pasado muchas horas en la biblioteca y sin los cuales no hubiera aguantado, gracias por hacer de estos años una experiencia inolvidable.

Por supuesto mencionar mis tutores Joaquín y José los cuales me han guiado y ayudado mucho a llevar a cabo este trabajo y sin los cuales hubiera sido un trabajo mucho más difícil de concretar.

RESUMEN

En este Trabajo de Fin de Grado se ha diseñado y construido una estación meteorológica formada por una placa Arduino Mega 2560, un sensor de temperatura y humedad DHT11, un LED RGB, dos LEDs, uno rojo y otro verde y una pantalla LCD. Se han implementado códigos de corrección de errores (ECC) para abordar posibles fallos en la memoria del microprocesador.

A partir de esta base se contemplan los siguientes dos objetivos:

Por un lado, la implementación de una estación meteorológica basada en Arduino que sea capaz de leer temperatura y humedad, que enseñe esos datos por la pantalla LCD y que a la vez nos enseñe, gracias al LED RGB, niveles críticos (muy altos o bajos) de temperatura y humedad. En esta parte del trabajo se ha realizado tanto la parte física de la estación, es decir, el montaje de los componentes y su conexión entre ellos, como la parte lógica, es decir, el software que permite el correcto funcionamiento de la estación.

Por otro lado, la implementación de un ECC que reduzca los errores debido a los posibles fallos de memoria del microprocesador. Para ello se utilizarán códigos Bose-Chaudhuri-Hocquenghem (BCH) y códigos de baja redundancia y sobrecarga reducida (Low Redundancy and Reduced Overhead: LRRO) los cuales nos permitirán asegurar los datos recogidos por la estación frente a los tipos de error más frecuentes (simples y dobles). Para validar los ECC se han llevado a cabo experimentos de inyección de fallos y se han analizado sus resultados.

Finalmente, se ha realizado un análisis comparativo de los resultados obtenidos con los diferentes códigos implementados, que permita determinar la mejor opción en función de los parámetros de diseño que se pretenda optimizar. En este caso, el tiempo de ejecución y el uso de memoria.

Palabras clave: Arduino Mega 2560, sensor DHT11, códigos de corrección de errores, códigos BCH, códigos de baja redundancia y sobrecarga reducida, inyección de fallos.

ABSTRACT

In this Final Degree Project we have designed and built a weather station consisting of an Arduino Mega 2560 board, a temperature and humidity sensor DHT11, an RGB LED, two LEDs, one red and one green and an LCD display, to which we have implemented error correction codes (ECC) to address possible memory failures of the microprocessor.

From this basis, the following two objectives are contemplated:

On the one hand, the implementation of a weather station based on Arduino that is able to read temperature and humidity, that shows these data through the LCD screen and that at the same time shows us thanks to the RGB LED critical levels (very high or low) of temperature and humidity. In this part of the work, both the physical part of the station, i.e. the assembly of the components and their connection to each other, and the logical part, i.e. the software that allows the station to function correctly, have been carried out.

On the other hand, the implementation of an ECC that reduces errors due to possible memory failures of the microprocessor. For this purpose, Bose-Chaudhuri-Hocquenghem (BCH) and Low Redundancy and Reduced Overhead (LRRO) codes will be used, which will allow us to secure the data collected by the station against the most frequent types of errors (single and double). To validate the ECCs, fault injection experiments have been carried out and their results have been analysed.

Finally, a comparative analysis of the results obtained with the different implemented codes has been carried out to determine the best option depending on the design parameters to be optimised. In this case, execution time and memory usage.

Keywords: Arduino Mega 2560, DHT11 sensor, error correction codes, low redundancy codes and low overhead.

RESUM

En aquest Treball de Fi de Grau s'ha dissenyat i construït una estació meteorològica formada per una placa Arduino Mega 2560, un sensor de temperatura i humitat DHT11, un LED RGB, dos Leds, un vermell i un altre verd i una pantalla LCD, a la qual se li han implementat codis de correcció d'errors (ECC) per a abordar possibles fallades de memòria del microprocessador.

A partir d'aquesta base es contemplen els següents dos objectius:

D'una banda, la implementació d'una estació meteorològica basada en Arduino que sigui capaç de llegir temperatura i humitat, que ensenyi aquestes dades per la pantalla LCD i que alhora ens ensenyi gràcies al LED RGB nivells crítics (molt alts o baixos) de temperatura i humitat. En aquesta part del treball s'ha realitzat tant la part física de l'estació, és a dir, el muntatge dels components i la seva connexió entre ells, com la part lògica, és a dir, el codi que permet el correcte funcionament de l'estació.

D'altra banda, la implementació d'un ECC que redueixi els errors a causa de les possibles fallades de memòria del microprocessador. Per a això s'utilitzaran codis Bose-Chaudhuri-Hocquenghem (BCH) i codis de baixa redundància i despesa reduïda (Low Redundancy and Reduced Overhead: LRRO) els quals ens permetran assegurar les dades recollides per l'estació enfront dels tipus d'error més freqüents (simples i dobles). Per a validar els ECC s'han dut a terme experiments d'injecció de fallades i s'han analitzat els seus resultats.

Finalment, s'ha realitzat una anàlisi comparativa dels resultats obtinguts amb els diferents codis implementats, que permeti determinar la millor opció en funció dels paràmetres de disseny que es pretengui optimitzar. En aquest cas, el temps d'execució i l'ús de memoria.

Paraules clau: Arduino Mega 2560, sensor DHT11, codis de correcció d'errors, codis de baixa redundància i despesa reduïda.

Documentos contenidos en el TFG

- Memoria
- Presupuesto
- Pliego de condiciones
- Planos
- Anejo

Índice de la memoria

Índice del presupuesto

Índice del pliego de condiciones

Índice de planos

MEMORIA

DISEÑO DE UNA ESTACIÓN METEOROLÓGICA BASADA EN ARDUINO CON PROPIEDADES DE TOLERANCIA A FALLOS EN LA MEMORIA DEL MICROPROCESADOR

AUTOR: TOMÁS SAMMUT RODRÍGUEZ

TUTOR: JOAQUÍN GRACIA MORÁN

COTUTOR: LUIS JOSÉ SAIZ ADALID

Índice de la memoria

Índice de figuras

Índice de tablas

RESUMEN EJECUTIVO

En este trabajo se aborda la problemática de la eficiencia en la corrección de errores en sistemas de comunicación y almacenamiento de datos, donde los códigos BCH (Bose-Chaudhuri-Hocquenghem) son ampliamente utilizados. Sin embargo, presentan limitaciones en términos de consumo energético y de memoria. Surge así la oportunidad de explorar y comparar los códigos BCH con una versión mejorada, los códigos LRRO (Low Redundancy and Reduced Overhead), con el objetivo de optimizar el rendimiento y la eficiencia en estos sistemas.

El diseño y la implementación de este trabajo deben cumplir con varias restricciones, incluyendo las normas y códigos de corrección de errores establecidos en la literatura, así como las necesidades y requisitos específicos del hardware utilizado (placa Arduino Mega 2560, sensor DHT11, LED RGB y pantalla LCD).

Los objetivos principales de este trabajo son:

- El diseño de una estación meteorológica formada por una placa Arduino Mega 2560, un sensor DHT11, un LED RGB y una pantalla LCD. Esta estación permitirá obtener datos de temperatura y humedad, que serán posteriormente protegidos por los códigos de corrección de errores.
- El desarrollo de un software en el IDE de Arduino, que permita el correcto funcionamiento de dicha estación. Este código permitirá la lectura de los datos del sensor y su visualización en la pantalla LCD, así como la activación del LED RGB para indicar niveles críticos de temperatura y humedad.
- La implementación de los Códigos de Corrección de Errores (ECC) en la estación meteorológica y la comparación de los códigos BCH y su versión mejorada, los códigos LRRO.

Para alcanzar estos objetivos, se hizo un análisis de las capacidades y limitaciones de los códigos BCH y LRRO. Se diseñó la estación meteorológica, considerando la integración de todos sus componentes, y se desarrolló un software que permite la correcta adquisición y visualización de los datos recogidos.

Se evaluaron los códigos BCH y LRRO para encontrar la solución más óptima. La comparación se realizó mediante la técnica de inyección de fallos, analizando el tiempo de ejecución y el uso de memoria de cada código.

Se logró diseñar y construir la estación meteorológica, desarrollar el software necesario y realizar la implementación de los códigos de corrección de errores. La comparación entre los códigos BCH y LRRO permitió confirmar las mejoras propuestas por los códigos LRRO.

Este trabajo tiene un impacto significativo en el campo de la corrección de errores, ya que demuestra que los códigos LRRO pueden ofrecer mejoras sustanciales respecto a los códigos BCH. A nivel práctico se pueden mejorar sistemas que lleven integrado códigos BCH si se sustituyen por códigos LRRO. El alcance de este estudio es amplio y se extiende a diversas aplicaciones tecnológicas, ofreciendo una base sólida para futuras investigaciones y desarrollos en esta área.

1. INTRODUCCIÓN

1.1. Objetivo del documento

El objetivo de este trabajo es comprobar el funcionamiento de los códigos de baja redundancia y sobrecarga reducida (LRRO) y compararlos de forma empírica con los códigos BCH, de los cuales están inspirados, para así confirmar o no que son una mejora en cuanto a tiempo de ejecución y uso de memoria. Para poder probar ambos códigos en una aplicación práctica también se ha diseñado una estación meteorológica basada en Arduino que nos permitirá obtener datos de temperatura y humedad, que serán los que se verán protegidos por dichos códigos de corrección de errores. Por ello, se pueden destacar tres objetivos principales que se irán desarrollando a través de esta memoria:

- El diseño de una estación meteorológica formada por una placa Arduino Mega 2560, un sensor DHT11, un LED RGB y una pantalla LCD. Esta estación permitirá obtener datos de temperatura y humedad, que serán posteriormente protegidos por los códigos de corrección de errores.
- El desarrollo de un software en el IDE de Arduino, que permita el correcto funcionamiento de dicha estación. Este código permitirá la lectura de los datos del sensor y su visualización en la pantalla LCD, así como la activación del LED RGB para indicar niveles críticos de temperatura y humedad.
- La implementación de los Códigos de Corrección de Errores (ECC) en la estación meteorológica y la comparación de los códigos BCH y su versión mejorada, los códigos LRRO.

1.2. Estructura del documento

Este trabajo está enfocado en el estudio de Códigos de Corrección de Errores y se centra concretamente en los códigos LRRO y su comparación con los clásicos códigos BCH. Para llevar a cabo esta comparativa, se implementarán ambos y se comprobará su funcionamiento mediante la técnica de inyección de fallos, tomándose nota tanto del tiempo de ejecución como del uso de memoria de cada uno de ellos. Además, también se diseñará la estación meteorológica en la que se implementarán dichos códigos, la cual estará formada por una placa Arduino Mega 2560, un sensor de temperatura y humedad DHT11, un LED RGB y una pantalla LCD.

A continuación, se detalla la estructura que tendrá el documento y cómo está organizado:

- En primer lugar, se hará una introducción al concepto de tolerancia a fallos, en la cual se explicará en detalle de qué trata para así poder entender el objetivo de este documento.
- En segundo lugar, se presentará el hardware utilizado y se dará toda la información necesaria, con el fin de conocer sus características y el papel de cada uno de los componentes en este trabajo. También se explicará detalladamente el proceso de montaje y el funcionamiento de la estación meteorológica, mostrando cómo los componentes trabajan en conjunto para medir y mostrar los datos de temperatura y humedad.

- En tercer lugar, se pasará a la presentación de los códigos de corrección de errores, empezando por una introducción general de dichos códigos para luego enfocarse en los propiamente elegidos para este trabajo.
- En cuarto lugar, se mostrará el programa desarrollado para este trabajo y se dará una explicación detallada de cada una de sus partes.
- En quinto lugar, se mostrarán las pruebas realizadas, los resultados obtenidos y una valoración final de estos.
- Por último, se cerrará el documento con las conclusiones obtenidas tras el estudio de los resultados conseguidos anteriormente.

1.3. Motivación

Los códigos BCH son ampliamente conocidos en el campo de la corrección de errores, dado que pueden corregir errores aleatorios (que pueden aparecer en cualquier bit o bits de un dato) con la mínima redundancia. Sin embargo, los códigos BCH primitivos sólo existen para algunas longitudes de datos, que no suelen ser las mismas que las que se utilizan en las memorias. Por ello, en este caso, se ha desarrollado un código BCH para corregir errores simples y dobles aleatorios en palabras de 32 bits. Aun así, presentan consumos de tiempo, de tamaño de programa, y energéticos que podrían disminuirse. Por ello, se han desarrollado los códigos LRRO, con los cuales se quiere disminuir ese gasto y, por tanto, asegurar cualquier proceso sin que su rendimiento se vea comprometido. Dado que este proyecto está enfocado en la programación y es una rama de la ingeniería que me fascina, he decidido aprovechar la oportunidad de profundizar mi aprendizaje sobre el tema, y así tener una primera experiencia que me confirme que esta rama es la que me gusta y en la que quiero formarme y crecer profesionalmente. Además, esta investigación representa una oportunidad única para evaluar mi afinidad con la rama de la ingeniería y la programación, ayudándome a confirmar mi interés en esta área y fortaleciendo mi vocación profesional, además de sumarse a los muchos otros campos ya estudiados durante los estudios de la carrera de GITI.

2. ANTECEDENTES

2.1. Introducción

Desde que se creó la primera red interconectada en 1969 por medio de una línea telefónica conmutada, el objetivo siempre ha sido intercambiar información, datos, mensajes de todo tipo a través de la red. Partiendo de esa base surge el problema de la fiabilidad de esos datos ya que, debido a diversos fallos, ya sean internos (fallos de memoria) o externos (ruidos o interferencias), la información puede ser modificada y no corresponder con lo que se tenía en un principio, lo que puede ocasionar averías en sistemas que precisan del uso de datos concretos. Además, hay que añadir que en la actualidad se ha aumentado drásticamente la capacidad de almacenamiento de los sistemas de memoria gracias a la gran reducción de tamaño de la tecnología CMOS, lo que ha tenido como efecto negativo el aumento de los fallos que se producen, ya sean simples o múltiples. Este aumento de fallos es, además, mayor cuanto más se reduce el tamaño de los microprocesadores, por lo que, con la tendencia actual, dicho número de fallos está en aumento y se ha de encontrar una solución que palie este problema.

Es por ello por lo que se desarrollaron Códigos de Corrección de Errores capaces de detectar errores y en algunos casos hasta de corregirlos, pudiendo así asegurar la fiabilidad de los datos con los que se quiere trabajar, permitiendo tener unos sistemas más robustos, que sufren menos averías y con una tasa de éxito mayor.

Aunque los códigos BCH [1] son ampliamente utilizados, presentan algunas limitaciones y son mejorables. Por un lado, los códigos BCH primitivos sólo existen para algunas longitudes de palabra, que no siempre coinciden con las utilizadas en los sistemas digitales, y por ello se tienen que adaptar para longitudes de palabra que son potencia de dos (8, 16, 32 y 64). Además, con el objetivo de agilizar el proceso de codificación y decodificación y hacerlo más rápido y eficaz, se han desarrollado los códigos LRRO [1], los cuales están inspirados en los códigos BCH previamente mencionados, pero obtenidos con una metodología totalmente diferente.

En este trabajo se estudiarán y compararán las velocidades de ejecución y el uso de memoria de los códigos BCH y LRRO de forma empírica en el caso concreto de una estación meteorológica basada en Arduino. El análisis de estas mejoras permitirá evaluar la efectividad de los códigos LRRO y su impacto en la fiabilidad de los datos recopilados por la estación meteorológica.

2.2. Introducción a la Tolerancia a Fallos

La tolerancia a fallos es un principio fundamental en el diseño y la ingeniería de sistemas, primordial para garantizar la fiabilidad y la integridad de los datos en una amplia variedad de aplicaciones [2]. La tolerancia a fallos desempeña un papel crucial para garantizar el correcto funcionamiento continuo y evitar fallos potencialmente catastróficos. En esencia, la tolerancia a fallos se refiere a la capacidad de un sistema para mantener un funcionamiento correcto incluso cuando se producen fallos en sus componentes o en el entorno circundante.

Este problema adquiere aún más relevancia en un mundo cada vez más interconectado y dependiente de la tecnología, en el que incluso pequeñas perturbaciones pueden tener consecuencias significativas. Es por ello por lo que la implantación de mecanismos robustos de tolerancia a fallos se ha convertido en una prioridad para paliar a estos requisitos.

Para garantizar el correcto funcionamiento de estos sistemas, la tolerancia a fallos implica la adopción de estrategias proactivas para detectar, aislar y corregir los efectos de los posibles errores que puedan surgir durante dicho funcionamiento. Esto implica el uso de programas o circuitos, que incorporan mecanismos de autodiagnóstico, técnicas de recuperación de errores y otras medidas preventivas y correctivas.

En resumen, la tolerancia a fallos no es solo un requisito técnico, sino también un componente crucial para garantizar la fiabilidad y la seguridad de operación de los sistemas en situaciones adversas o inesperadas.

2.3. Códigos de corrección de errores (ECCs)

En esta sección se proporcionará una introducción completa a los Códigos de Corrección de Errores (ECC). Se explicará de manera clara y concisa qué son, cuál es su propósito y cómo funcionan para detectar y corregir errores en los datos transmitidos o almacenados.

Gracias a la reducción del tamaño de la tecnología CMOS, se ha conseguido aumentar la capacidad de almacenamiento en los sistemas de almacenamiento. Sin embargo, esto también ha provocado un aumento de fallos, provocando errores simples o múltiples en la memoria.

Por ello se desarrollaron los Códigos de Corrección de Errores (ECCs), capaces de detectar y corregir uno o varios errores para una gran variedad de longitudes de palabra (en este proyecto se trabajará con palabras de 32 bits de longitud ya que es la longitud con la cual se guardan los datos de tipo float en la memoria del microprocesador).

En este TFG se han utilizado dos ECCs. En primer lugar, el código BCH (Bose-Chaudhuri-Hocquenghem) [3] [4]. Este código es capaz de corregir múltiples errores y destaca por su capacidad para manejar errores dispersos en los datos transmitidos o almacenados. Los códigos BCH son muy eficientes en términos de detección y corrección de errores. Sin embargo, uno de los desafíos principales a la hora de construir este ECC es la complejidad computacional involucrada en su implementación, especialmente para grandes tamaños de datos. El proceso de codificación y decodificación requiere operaciones matemáticas avanzadas, lo que puede resultar en un consumo considerable de recursos de memoria y tiempo de ejecución.

Por otro lado, el código de corrección de errores LRRO (Low Redundancy and Reduced Overhead) ha sido diseñado con el objetivo de superar algunas de las limitaciones de los códigos BCH. El código LRRO mantiene la capacidad de corregir errores múltiples, pero lo hace con un menor número de bits redundantes, lo que se traduce en un uso más eficiente de la memoria y un menor consumo energético. Además, los algoritmos de codificación y decodificación en LRRO están optimizados para reducir la carga computacional, lo que permite tiempos de ejecución más rápidos y una implementación más sencilla en sistemas con recursos limitados. Estas mejoras hacen que el ECC LRRO sea una opción atractiva para aplicaciones donde la eficiencia y la rapidez son cruciales, sin comprometer la capacidad de corrección de errores.

Estas diferencias en el diseño y la implementación entre los códigos BCH y LRRO son clave para evaluar cuál de los dos se adapta mejor a los requerimientos específicos de este proyecto. A través de este TFG, se analizarán y compararán empíricamente ambos ECCs, tomando en cuenta factores como el tiempo de ejecución, el uso de memoria, y la capacidad de corrección de errores.

Los ECCs se pueden diseñar atendiendo a distintos parámetros, dependiendo de la aplicación concreta que se le vaya a dar. Por ejemplo, en el caso de las memorias, el objetivo suele ser minimizar el número de bits redundantes, ya que, al agregarse a cada palabra de toda la memoria, podría aumentar considerablemente el tamaño de la información que se desea guardar. Además, el uso de dichos códigos genera latencias debido a la codificación y decodificación, y aumenta el tamaño de los programas y el consumo de energía. Con el objetivo de reducir al máximo esa latencia introducida y ese aumento del tamaño del programa y del consumo de energía, se ha buscado alternativas a los códigos BCH como pueden ser los códigos LRRO (Low Redundancy and Reduced Overhead).

Todo sistema informático está compuesto como mínimo por el hardware (parte física, aquí la placa Arduino), el software (parte lógica, código) y la información que se almacena y se procesa (datos recogidos por sensores, pulsadores, etc.). Asegurar la integridad de la información manejada es imprescindible para el correcto funcionamiento del sistema.

Los ECCs pueden usarse en diversas aplicaciones como:

- **Almacenamiento:** Registros, RAM, HDDs, SDDs, CDs/DVDs, etc.
- **Comunicación:** buses, Entrada/Salida, intercambio de información, etc.

También se pueden diferenciar dos tipos de ECCs en función de la cantidad de información que se va a transmitir:

- **Transmisiones grandes:** CDs, DVDs, comunicación de satélites, etc.
- **Microprocesadores:** con tamaños de palabra pequeños (8, 16, 32, 64).

Un concepto importante a tener en cuenta es el hecho de que la palabra codificada se obtiene codificando la palabra original y al decodificar la palabra codificada se volvería a obtener la palabra original. La codificación se puede definir como la representación de un dato siguiendo un conjunto de reglas establecidas.

Los códigos Bose-Chaudhuri-Hocquenghem (BCH) son uno de los ECC más conocidos. Fueron desarrollados en 1959 por Hocquenghem y en 1960 por Bose y Ray-Chaudhuri. Desde entonces, los códigos BCH se han empleado ampliamente en varias aplicaciones, como memorias flash en unidades de estado sólido (SSD), almacenamiento óptico como discos compactos (CD) o discos versátiles (DVD), y comunicaciones por satélite. Sus características algebraicas hacen que los códigos BCH sean muy útiles en una amplia gama de situaciones, y su cobertura de errores se consigue con una redundancia mínima.

El peso Hamming de una fila de una matriz de paridad determina el número de términos a los que se le deben aplicar una OR exclusiva (XOR) para calcular los bits de paridad y de síndrome. Por lo tanto, en la implementación de hardware, la fila más pesada define la profundidad lógica (o nivel lógico) del árbol XOR, que implementa el circuito codificador y el cálculo del síndrome en el circuito decodificador. Esto influye en la latencia introducida por el ECC. Del mismo modo, el peso Hamming de la matriz de paridad completa determina el cálculo de puertas lógicas necesarias en el circuito codificador y en el cálculo del síndrome en el circuito decodificador. Este número tiene una influencia importante tanto en el tamaño de los programas como en la energía consumida por estos circuitos o programas.

Por lo tanto, para reducir la sobrecarga introducida por los circuitos codificadores y decodificadores, debemos centrarnos en los pesos Hamming de la fila más pesada y de toda la matriz. A menudo, se pueden lograr estos objetivos aumentando el número de bits de paridad. Sin embargo, en este caso no es una buena idea ya que se trata de proteger memorias, ya que todos los bits redundantes deben ser almacenados por cada palabra en la memoria, lo que conlleva un programa de mayor tamaño y consumo de energía por parte de los circuitos de la memoria.

Luego, la única solución que queda es buscar matrices de paridad con menor peso de Hamming, como por ejemplo los ECC denominados LRRO (Low Redundancy and Reduced Overhead) [1].

Como ya se ha comentado anteriormente, este trabajo se hace sobre palabras de 32 bits y por lo tanto utilizaremos las versiones de los códigos BCH y LRRO denominadas (44,32) lo que hace referencia a que las palabras codificadas pasaran a tener 44 bits, 12 más que el dato original. Dichos bits se denominan bits de paridad o redundantes. Con esta redundancia, estos códigos son capaces de corregir errores simples y dobles aleatorios.

Los valores reales de la sobrecarga inducida por los circuitos codificadores y decodificadores de un ECC dependen de cada implementación: la tecnología, el proceso de síntesis, etc. Sin embargo, es posible hacer un estudio teórico previo. Si se comparan los dos códigos estudiados en este trabajo, es decir, el código BCH para palabras de 32 bits (cuya matriz de paridad se puede observar en la figura 1) y el código LRRO también para palabras de 32 bits (figura 1), se puede observar en la figura 2 que tanto el peso de Hamming para la fila más pesada como el peso Hamming de toda la matriz es menor para el código LRRO.

$$H_{LRRO\ 44,32} = \begin{bmatrix}
10000000000011100001100101000110101000000010\\
01000000000011011000001110100000100010000101\\
00100000000010101100011000001000011000000011\\
00010000000010010110100011001101000101000000\\
00001000000001101010110010001000000100011100\\
00000100000001010101010000100100011100101000\\
00000010000000110011001001010010000011100001\\
00000001000000000111100010001000100010010011\\
00000000100000000000111100110011000001001011\\
00000000010000000000000011110000110000101111\\
00000000001000000000000000011111000010101\\
00000000000100000000000000000000001111111110
\end{bmatrix}$$

$$H_{BCH\ 44,32} = \begin{bmatrix}
10000000000001010001101011100100100111111110110\\
01000000000001010001101011100100100111111011\\
00100000000000101000110101110010010011111101\\
00010000000010110111001101110000000110001000\\
00001000000011111000110001110001001100110010\\
00000100000011011111001111110001101001101111\\
00000010000001101111100111111000110100110111\\
00000001000000110111110011111100011010011011\\
00000000100010111000101110110111000010111011\\
00000000010001011100010111011011100001011101\\
00000000001010001101011100100100111111011000\\
00000000000101000110101110010010011111101100
\end{bmatrix}$$

Figura 1: Matrices de paridad del ECC LRRO (arriba) y del ECC BCH (abajo).
Fuente: Elaboración propia

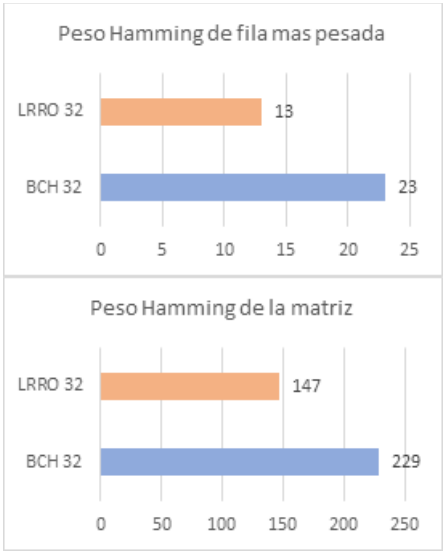

Figura 2: Peso de Hamming de los códigos BCH32 y LRRO32.
Fuente: Elaboración propia

Como ya se ha dicho, el peso Hamming de la fila más pesada y el peso Hamming de toda la matriz de paridad son parámetros importantes que influyen en la sobrecarga y la latencia introducida por los códigos de codificación y decodificación. Uno de los objetivos de este trabajo es comprobar que, efectivamente, este análisis teórico se traduce en programas más pequeños y menor tiempo de decodificación, manteniendo la misma cobertura de corrección de errores, al utilizar el código LRRO en sustitución del código BCH. Para cuantificar las mejoras, se hará una implementación práctica y se realizarán diversos ensayos.

3. HARDWARE UTILIZADO

En este apartado de la memoria se hará una breve presentación de los diferentes componentes que se van a utilizar para montar la estación meteorológica, así como una breve explicación de su funcionamiento.

En primer lugar, se justificará el porqué de la elección de la placa Arduino como núcleo de la estación meteorológica, destacando sus ventajas y su idoneidad para este proyecto. Se explorarán las características que hacen que la placa Arduino Mega 2560 [5] sea una opción interesante, especialmente para aplicaciones que requieren medición y procesamiento de datos en tiempo real.

Posteriormente, se presentarán en detalle los componentes utilizados en la estación meteorológica. Se explicará el funcionamiento del sensor DHT11 [6], que permite medir la temperatura y humedad ambiente. Asimismo, se abordará el uso del LED RGB [7], que proporcionará indicadores visuales para niveles críticos de temperatura y humedad. Además, se describirá el funcionamiento de la pantalla LCD y cómo se mostrarán los datos recopilados por el sensor.

3.1. Arduino

Arduino surge, en Italia, como un proyecto enfocado inicialmente a estudiantes del *Interaction Design Institute Ivrea* (IDII) en Ivrea, Italia, y cuyo objetivo principal era proponer microcontroladores más asequibles, ya que los que se usaban hasta entonces costaban alrededor de $100 USD, lo que dificultaba el acceso a muchos estudiantes [8]. Con el tiempo se fue expandiendo y democratizando cada vez más, abriéndose a un público mucho más amplio. Tanto es así, que en el año 2013 se estima que se habían vendido alrededor de 700.000 placas Arduino oficiales. El gran éxito de Arduino es debido a que es una plataforma electrónica de código abierto basada en hardware y software de fácil uso. Por lo tanto, por un lado, se tiene la placa Arduino que es capaz de leer entradas, ya sea por medio de un sensor o de un botón, para luego ejecutar instrucciones diversas tales como encender un LED, enseñar algo por pantalla o activar un motor. Para ello se utiliza el software Arduino (IDE), en el cual se escribirá todo código que se quiera que la placa ejecute.

La tecnología Arduino presenta varias ventajas, que se enumerarán a continuación, y es por ello por lo que se ha optado por ella:

- El software Arduino (IDE) es de fácil uso. Aprender a programa para Arduino es muy sencillo lo que agiliza el proceso de programación y evita complicaciones innecesarias.
- A nivel de Hardware, el montaje de cualquier proyecto es bastante sencillo gracias a los múltiples pines que posee la placa, y con un mínimo conocimiento en electrónica, se le pueden acoplar diversos componentes sin necesidad de soldar.
- El hecho de que no haga falta soldar nada hace que los proyectos montados con Arduino sean fácilmente montables, desmontables y modulables al gusto un gran número de veces.

- Es compatible con múltiples sistemas operativos, incluyendo Windows, Macintosh OSX y Linux, lo que amplía su versatilidad y facilidad de uso.
- Tiene un coste muy bajo, requisito indispensable ya que se trata de un TFG y el presupuesto es bastante limitado.
- Por último, hay que añadir que se trata de una tecnología de código abierto, lo que significa que se pueden encontrar infinidad de librerías que facilitan el uso del software y que permiten hacer proyectos más elaborados.

Todas estas ventajas mencionadas respaldan la elección de usar la tecnología Arduino, más concretamente la placa Arduino Mega 2560 que se presenta a continuación.

3.2. Placa Arduino Mega 2560

Arduino ofrece un amplio catálogo de placas, las cuales tienen diversos usos y propósitos. Para este proyecto se ha decantado por la placa Arduino Mega 2560 debido a sus características y funcionalidades específicas. La placa Arduino Mega 2560 se destaca por su versatilidad y capacidad de uso. Cuenta con 54 pines digitales de los cuales 15 nos permiten usar la modulación por ancho de pulsos (PWM) y 16 pines analógicos. Esta variedad de pines permite conectar y controlar una amplia gama de componentes y dispositivos, lo que resulta ideal para el montaje de la estación meteorológica. Su memoria Flash es de 256 KB (de los cuales se pueden usar 248). Por lo tanto, no habrá problema para poder almacenar tanto el código para el correcto funcionamiento de la estación meteorológica como los Códigos de Corrección de Errores, y, además, se tendrá margen para futuras mejoras o implementaciones adicionales. El microprocesador incorporado en esta placa es el ATmega2560 y opera a una frecuencia de reloj de 16 MHz, la cual es más que suficiente para ejecutar las instrucciones que se le piden en poco tiempo (se recuerda que uno de los objetivos es optimizar ese tiempo de ejecución). Además de sus características técnicas, la placa Arduino Mega 2560 destaca por su tamaño compacto (101,52x53,3 mm^2) y su peso ligero de 37 gramos, lo que la hace práctica y fácil de manejar en el montaje de la estación meteorológica.

Luego la elección de dicha placa Arduino Mega 2560 se respalda en su capacidad de proporcionar una plataforma confiable y adecuada para el desarrollo de la estación meteorológica basada en Arduino, cumpliendo con los requisitos de almacenamiento, procesamiento y flexibilidad necesarios para el proyecto.

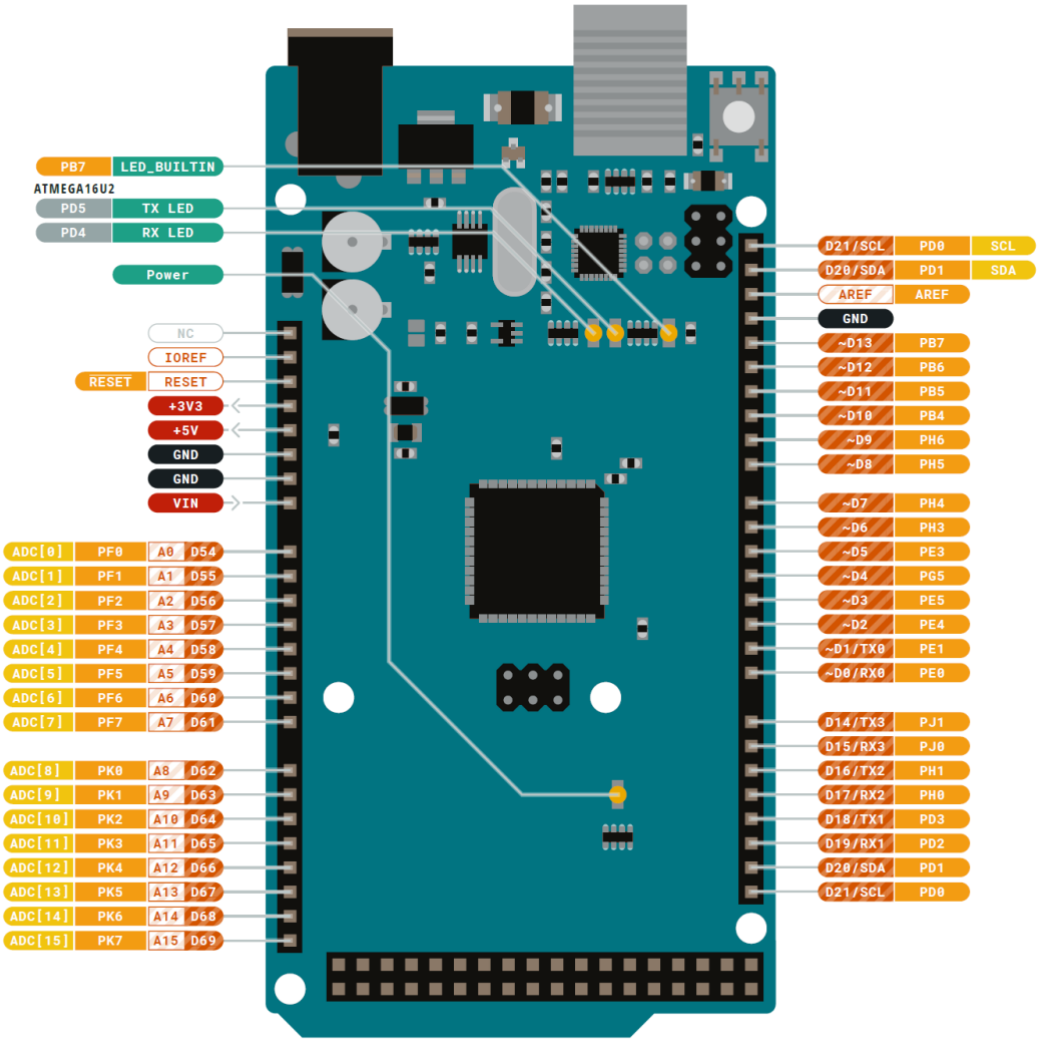

Figura 3: Placa Arduino Mega 2560.
Fuente: https://store.arduino.cc/collections/boards-modules/products/arduino-mega-2560-rev3

3.3. Sensor DHT11

El sensor DHT11 es un sensor digital para medir la temperatura y la humedad relativa del aire. Consiste en un elemento sensor de humedad capacitivo y un termistor para detectar la temperatura. Su relación calidad/precio es muy buena lo que lo convierte en una elección adecuada para este proyecto.

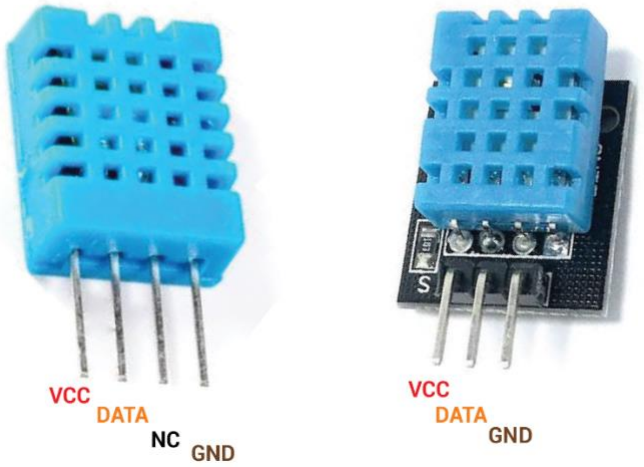

Figura 4: Sensor DHT11.
Fuente: https://www.automatizacionparatodos.com/sensor-dht11-arduino/

En cuanto a su rango de funcionamiento, es capaz de medir humedades relativas de 20% a 90% con una precisión de ±5%, y permite medir temperaturas de 0 a 50 ^{o}C con una precisión de ±2 ^{o}C. Unos rangos más que aceptables y que serán más que suficientes para los propósitos de la estación meteorológica.

Dentro del DHT11, hay un componente sensor de humedad junto con un termistor. El componente de detección de humedad tiene dos electrodos con un sustrato que retiene la humedad entre ellos. Los iones son liberados por el sustrato a medida que éste absorbe el vapor de agua, lo que a su vez aumenta la conductividad entre los electrodos. Se procesa estos valores de resistencia cambiados y los formatea en forma digital.

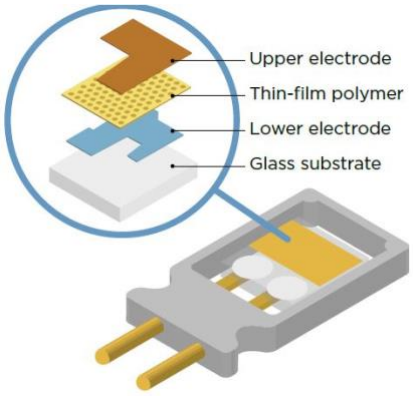

Figura 5: Estructura interna del sensor de humedad.
Fuente: https://www.vaisala.com/en/blog/2023-07/humidity-sensors-do-you-know-whats-inside

El DHT11 también contiene un sensor NTC/Termistor para medir la temperatura del aire. Un termistor es una resistencia térmica cuya resistencia cambia drásticamente con la temperatura. El término NTC significa Coeficiente Negativo de Temperatura, lo que significa que la resistencia disminuye con el aumento de la temperatura.

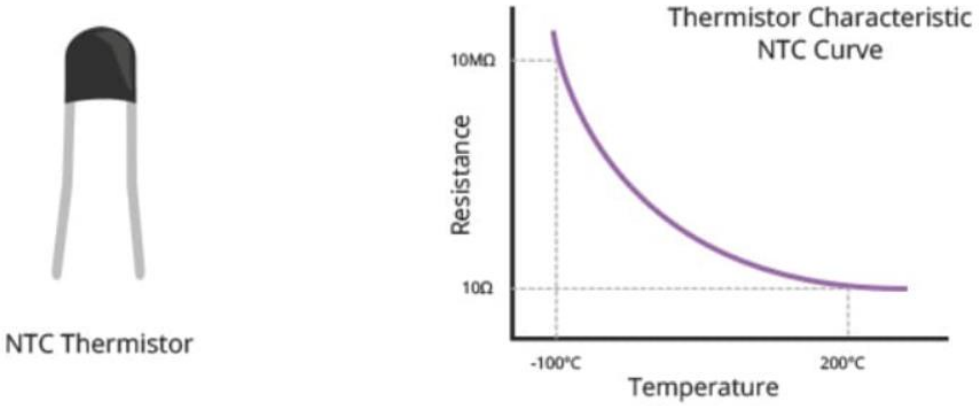

Figura 6: Sensor NTC/Termistor (izquierda) y gráfica Resistencia/Temperatura (derecha).
Fuente: https://descubrearduino.com/dht11/

En su interior, el DHT11 contiene una pequeña placa de circuito impreso (PCB) con un paquete SOIC-14 IC de 8-bit. Este circuito impreso mide y procesa la señal analógica con los coeficientes de calibración almacenados, lleva a cabo la conversión analógica a digital y devuelve una señal digital con la temperatura y la humedad.

Hay que añadir que para poder usar el sensor DHT11 se ha tenido que incluir la librería DHT en el código de la estación meteorológica, la cual permite utilizar todas las funciones necesarias para el uso del sensor.

3.4. LED RGB

Con el propósito de proporcionar una "alarma" visual rápida y sencilla, se ha incluido en el circuito un LED RGB que indicará cual es el nivel de humedad relativa y temperatura del aire en un momento dado mediante un código de colores. Un LED RGB (*Light Emitting Diode*, o Diodo Emisor de Luz) es un tipo de LED que puede emitir luz de cualquier color a partir de la combinación de sus tres colores básicos: rojo (*Red*), verde (*Green*), azul (*Blue*). En otras palabras, se trata de la unión de tres LEDs de los colores básicos (rojo, verde y azul) en un encapsulado común (hay un único cátodo y es común a los tres LEDs).

El código de colores para la temperatura es:

- Hace calor (>25 ^{o}C): led rojo encendido.
- Hace frio (<5 ^{o}C): led azul y led verde encendidos → color cian.
- Hace una temperatura agradable (>5 ^{o}C y <25 ^{o}C): led verde encendido.

El código de colores para la humedad es:

- La humedad relativa es elevada (>60%): led rojo y led azul encendidos → color magenta.
- La humedad relativa es baja (<25%): led azul encendido.
- La humedad relativa está en niveles óptimos (>25% y <60%): led verde encendido.

De esta manera, el LED RGB proporcionará una indicación visual clara del estado de temperatura y humedad en todo momento, facilitando la interpretación de los datos recolectados por la estación meteorológica.

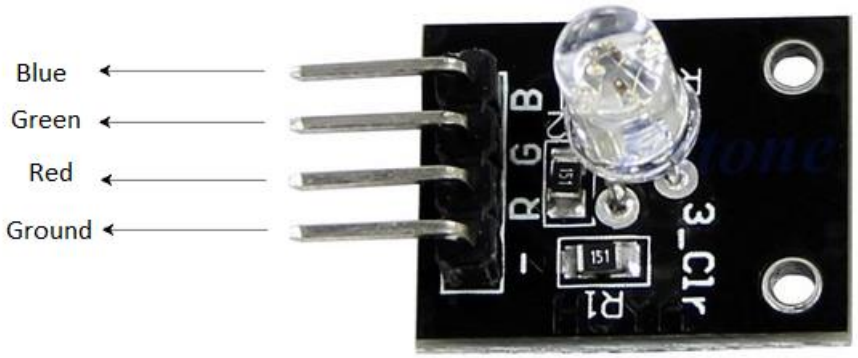

Figura 7: LED RGB con pines marcados.
Fuente: https://www.prometec.net/diodo-led-rgb-s4a/#:~:text=La%20conexi%C3%B3n%20de%20este%20tipo,puede%20conseguir%20cualquier%20otro%20color.

3.5. LED rojo y LED verde

Para añadir, de manera visual, un modo de mostrar el correcto funcionamiento o no de la estación se ha optado por añadir dos LEDs monocromos, uno de color verde que se mantendrá encendido, de manera continua, si la estación funciona correctamente y otro rojo que indicará que hay algún tipo de problema y que la estación no está operando como es debido.

Figura 8: LEDs monocromos rojo y verde
Fuente: Uxcell Led Diode Lights Plastic 5mm Diameter Round Two Color Green Red 20pcs - Walmart.com

3.6. Pantalla LCD

Con el fin de mostrar los valores exactos de temperatura y humedad medidos, se ha optado por incorporar una pantalla LCD [9] a la estación meteorológica. Esto hace que la estación sea completamente independiente, y una vez que los códigos están implementados en la placa Arduino, solo haría falta alimentarla con una pila de 5V, sin necesidad de estar conectada a un ordenador, lo cual la hace más portátil y versátil.

Para este proyecto se ha optado por una pantalla LCD 1602, que consta de dos líneas de 16 caracteres cada una, la cual será más que suficiente para mostrar los datos de temperatura y humedad o cualquier mensaje relevante (de error, por ejemplo) que se quiera mostrar.

La pantalla LCD 1602 consta de 16 pines y puede ser algo complicada de conectar con el resto de la estación meteorológica. Es por ello por lo que se ha optado por una pantalla LCD 1602 con un módulo I2C que reduce el número de pines a conectar a tan sólo 4:

- **GND:** pin negativo que irá conectado a tierra.
- **VCC:** pin positivo que irá conectado a los 5V de la placa Arduino.
- **SCL (*System Clock*):** se trata de la línea de pulsos que sincronizan el sistema.
- **SDA (*System Data*):** se trata de la línea por la que se mueven los datos entre los dispositivos (en este caso, la placa Arduino y la pantalla LCD).

Además, otra ventaja que tiene el uso de un módulo I2C es el hecho de que tiene un potenciómetro directamente incorporado, y por lo tanto, no hace falta conectar uno. Eso implica mayor facilidad de uso y hace la estación mucho más compacta. El potenciómetro se usa para regular la luminosidad de la pantalla LCD y así asegurar que se vean bien los datos.

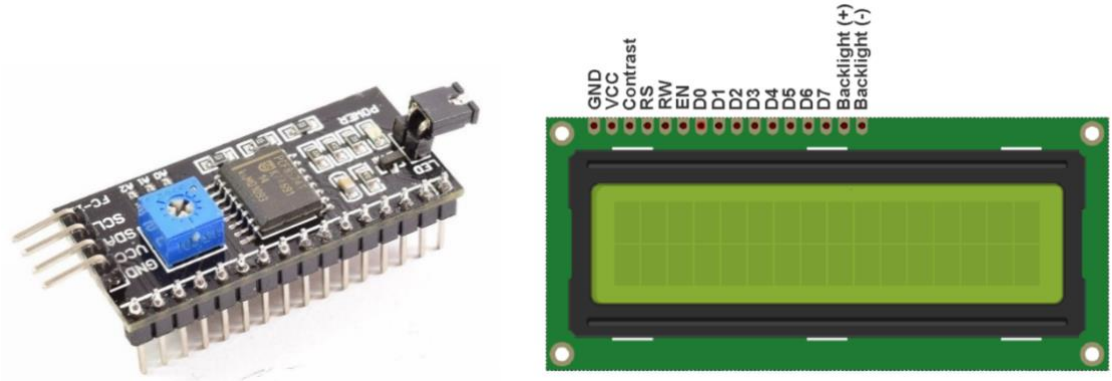

Figura 9: Módulo i2c (izquierda) y pantalla LCD 1602 (derecha).
Fuentes: https://naylampmechatronics.com/blog/35_tutorial-lcd-con-i2c-controla-un-lcd-con-solo-dos-pines.html
https://controlautomaticoeducacion.com/arduino/lcd/

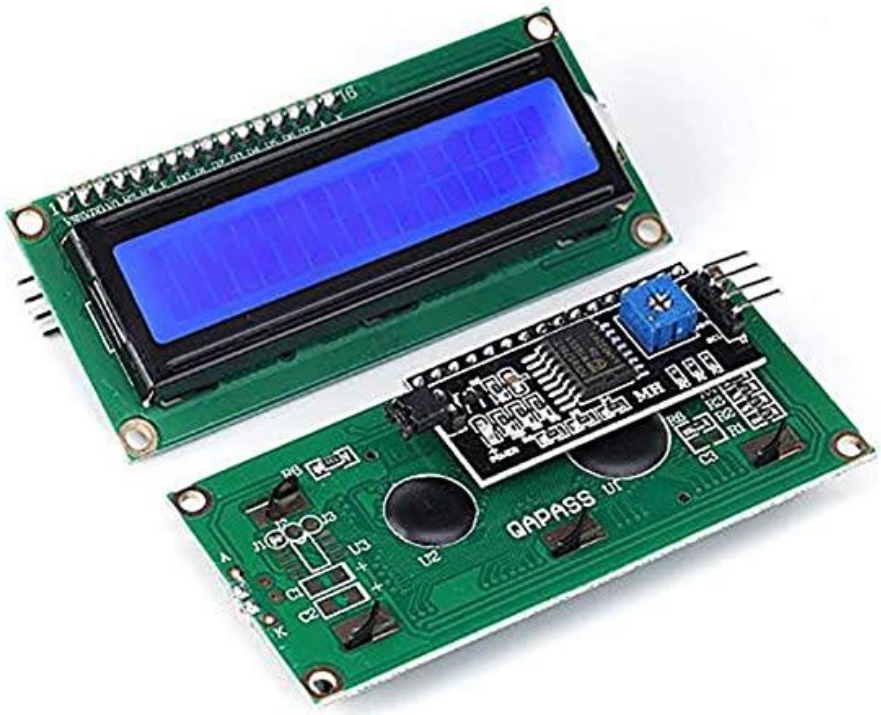

Figura 10: Pantalla LCD 1602 con módulo i2c incorporado.
Fuente: https://www.eelectronicparts.com/blogs/news/1602-lcd-with-i2c-serial-interface-adapter

En cuanto al flujo de información, en primer lugar, el sensor DHT11 recoge un valor de temperatura y de humedad, la placa Arduino lo recibe y lo almacena, y, por último, ese valor es enviado a la pantalla LCD para ser mostrado en pantalla. Este proceso se repite cada X segundos, y así se puede tener la temperatura y humedad en todo momento siempre y cuando no ocurra ningún error de lectura.

4. MONTAJE Y PROGRAMACIÓN DE LA ESTACIÓN METEOROLÓGICA

En este apartado se detallará la metodología empleada para el montaje de la estación meteorológica, así como la implementación del software tanto para el correcto funcionamiento de la estación como de los códigos de corrección de errores. Pero primero se va a hacer una breve presentación del entorno de desarrollo utilizado para la realización de este trabajo.

Como ya se ha comentado antes, para la realización e implementación del software que se va a utilizar en la estación tanto para su funcionamiento como para el uso de los ECCs, hace falta un Entorno de Desarrollo Integrado (IDE). En este caso como Arduino posee uno propio es el que se va a utilizar, al ser propio de Arduino es el más adecuado para la placa Arduino Mega 2560.

El Entorno de Desarrollo Integrado (IDE) de Arduino [10] es una aplicación multiplataforma que se utiliza para escribir y cargar programas en placas Arduino o placas compatibles con la tecnología Arduino. Admite los lenguajes C y C++, utilizando alguna que otra regla especial de estructuración de códigos.

Además, suministra una gran variedad de bibliotecas de software, también conocidas como librerías, listas para ser instaladas y utilizadas. Estas librerías informáticas son un conjunto de implementaciones, codificadas en el mismo lenguaje de programación usado (en este caso C++), que ofrece una interfaz definida para la función para la cual es invocada. La principal diferencia entre un programa ejecutable y una librería es que la librería no está concebida para ser utilizada de forma autónoma, sino que su fin es ser utilizada por otros programas. Hay que añadir que algunas librerías pueden requerir de otras para funcionar correctamente y por ello es importante saber qué hace cada librería y qué necesita para su correcto funcionamiento.

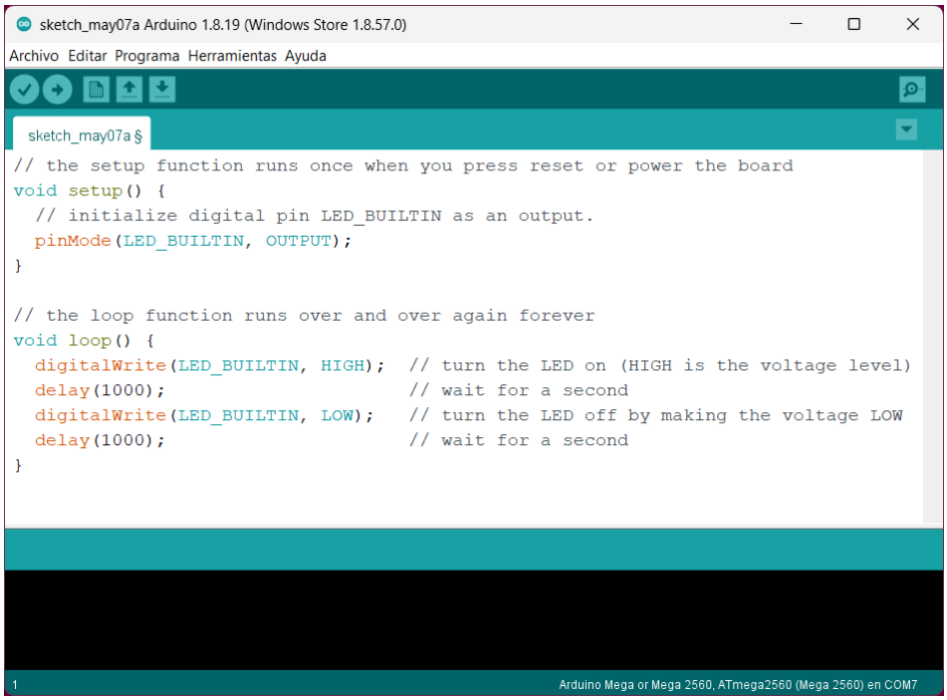

Figura 11: IDE de Arduino.
Fuente: Elaboración propia

El código escrito en el IDE de Arduino sólo requiere de dos funciones básicas para su correcto funcionamiento:

- **La función *setup*:** sólo se ejecuta una vez y sirve para la inicialización de los elementos, inicializar las variables, la comunicación con nuestro equipo o para inicializar los pines (como entrada o salida) de nuestra tarjeta de Arduino.
- **La función *loop*:** se ejecuta continuamente hasta que se apague la placa Arduino, y contiene las instrucciones que tiene que ejecutar la placa. Se usa para la lectura de las entradas, la activación de las salidas, llamadas a otras funciones auxiliares, entre otras muchas cosas.

A parte de estas dos funciones básicas e indispensables, también se pueden añadir más funciones que serán luego llamadas por las funciones *setup* o *loop* y que complementarán el código, o en su defecto también se pueden añadir librerías que contendrán ellas mismas funciones que se podrán llamar. Este último método es muy práctico, ahorra tiempo, facilita el trabajo y permite tener un código más claro y conciso.

En segundo lugar, se hablará del **montaje** de la estación meteorológica, la cual está compuesta de:

- Una protoboard para conectar todos los componentes.
- Una placa Arduino Mega 2560.
- Un sensor DHT11.
- Un LED RGB.
- Un LED Rojo.
- Un LED Verde.
- Dos resistencias de 220 Ohmios.
- Una pantalla LCD 1602 con un módulo i2c incorporado.
- Cables para conectar los componentes a la placa Arduino.
- Un cable USB-A a USB-B para conectar la placa Arduino al ordenador.

A continuación, se detallan las conexiones de los diferentes compuestos y qué pines de la placa Arduino se utilizan. Esto es muy importante tenerlo en cuenta para el posterior desarrollo del software que permitirá el correcto funcionamiento de la estación.

El sensor DHT11 tiene cuatro pines, pero solo tres conexiones:

- El primer pin se conecta a VCC, en este caso 5V.
- El segundo pin corresponde a la línea de datos, conectado al pin 7 de la placa Arduino.
- El tercer pin no se conecta a nada.
- El cuarto pin se conecta a tierra (GND).

El LED RGB tiene cuatro conexiones:

- El primer pin corresponde con el cátodo común y se conecta a tierra (GND).
- El segundo pin corresponde al LED Rojo y se conecta al pin 12 de la placa Arduino.

- El tercer pin corresponde al LED Verde y se conecta al pin 11 de la placa Arduino.
- El cuarto pin corresponde al LED Azul y se conecta al pin 10 de la placa Arduino.

Tanto el LED Rojo como el LED Verde tienen dos conexiones:

- El primer pin de cada LED (el cátodo) se conecta a tierra (GND) a través de una resistencia de 220 Ohmios.
- El segundo pin de los LED Rojo y Verde (el ánodo) se conectan respectivamente a los pines 22 y 24 de la placa Arduino.

La pantalla LCD 1602 tiene cuatro conexiones (gracias al módulo I2C):

- El primer pin se conecta a tierra (GND).
- El segundo pin se conecta a VCC (5V).
- El tercer pin corresponde a la línea de datos (SDA) y se conecta al pin 20 SDA de la placa Arduino.
- El cuarto pin corresponde a la línea de pulsos (SCL) y se conecta al pin 21 SCL de la placa Arduino.

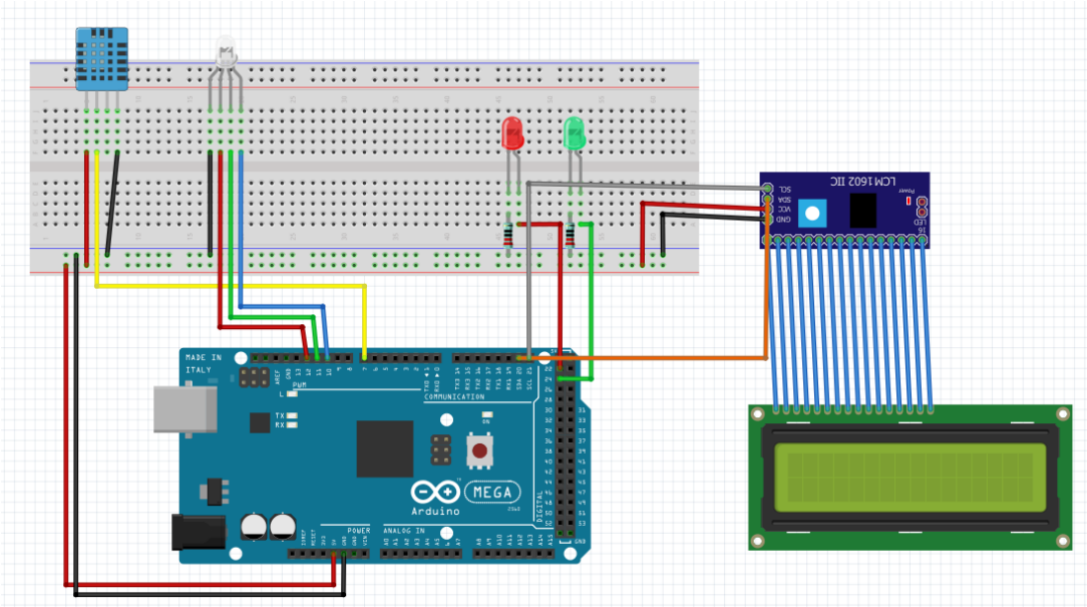

Figura 12: Esquema del montaje de la estación meteorológica.
Fuente: Elaboración propia

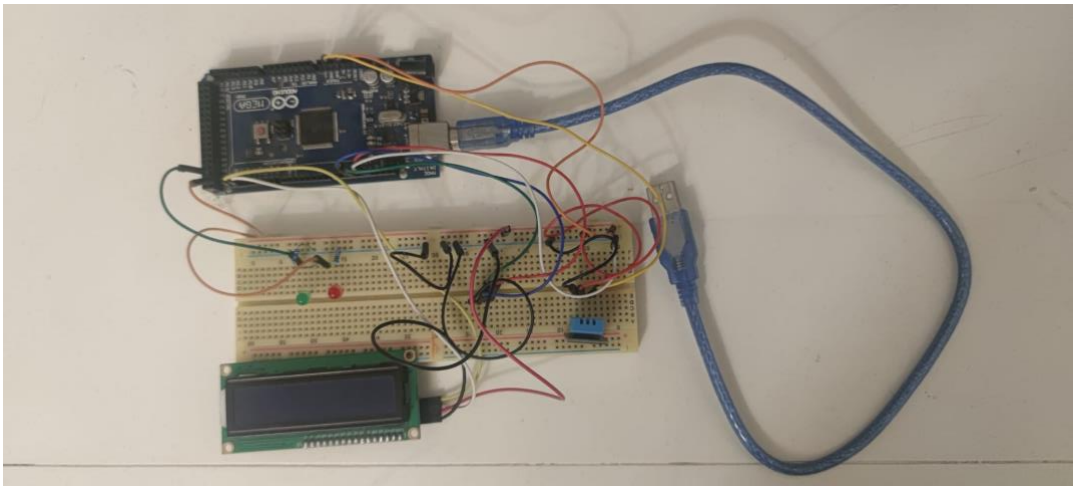

Figura 13: Estación meteorológica.
Fuente: Elaboración propia

Por último, se explican algunos aspectos del **software** necesario para el correcto **funcionamiento** de la estación meteorológica.

Para poder usar el sensor DHT11 se ha tenido que instalar la librería **DHT** directamente disponible en el IDE de Arduino. También se han tenido que instalar las librerías **Wire** y **LiquidCrystal_I2C** para poder utilizar la pantalla LCD con el módulo I2C incorporado. Estas librerías permitirán el uso de diferentes funciones para el uso del sensor DHT11 y de la pantalla LCD 1602 tales como *readHumidity, readTemperature, clear, setCursor* entre otras muchas.

En cuanto al software implementado ha requerido una labor de integración de la programación de todos los elementos (sensor, pantalla, etc.) para que trabajen coordinadamente, tal como se explica a continuación:

- En primer lugar, se incluyen las librerías y se definen las constantes necesarias.
- En segundo lugar, dentro de la función *setup*, se inicializan el sensor, la pantalla LCD y se define el modo de los pines utilizados (Output o Input).
- En tercer lugar, dentro de la función *loop*, se empieza por leer la temperatura y la humedad relativa con el sensor DHT11. Se aprovecha para verificar si la lectura se hace correctamente, en cuyo caso el LED Verde se encenderá, o si hay algún error, en cuyo caso se encenderá el LED Rojo y se enseñará un mensaje de error en la pantalla LCD.
- En cuarto y último lugar, dentro de la función *loop*, se muestran los datos de temperatura y humedad leídos por el sensor DHT11 y se enciende el LED RGB según el código de colores establecido, primero para la temperatura y luego para la humedad.

```
#include <DHT.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>

#define DHTPIN 7 // Definimos el pin digital donde se conecta el sensor
#define DHTTYPE DHT11 // Definimos tipo de sensor

DHT dht(DHTPIN, DHTTYPE); // Inicializamos el sensor DHT11
LiquidCrystal_I2C lcd(0x27,16,2);  // Crear el objeto lcd
                                   // dirección  0x27 y 16 columnas x 2 filas
const int rgb_rojo = 12;
const int rgb_verde = 11;
const int rgb_azul = 10;

const int led_rojo = 22;
const int led_verde = 24;

unsigned long tiempo;

void setup() {
  Serial.begin(9600);

  dht.begin(); // Comenzamos el sensor DHT

  lcd.init(); // Inicializar el LCD
  lcd.backlight(); //Encender la luz de fondo

  pinMode(rgb_rojo, OUTPUT);
  pinMode(rgb_verde, OUTPUT);
  pinMode(rgb_azul, OUTPUT);

  pinMode(led_rojo, OUTPUT);
  pinMode(led_verde, OUTPUT);
}
```

Figura 14: Librerías, constantes y función setup.
Fuente: Elaboración propia

El código presente en la función *loop* se repite de manera infinita dando así la temperatura y humedad en todo momento. También se ha añadido la función *micros* para calcular el tiempo de ejecución del código, característica que se comparará para los códigos BCH y LRRO.

Por último, se va a pasar a explicar la parte del **software** que corresponde con la **detección y corrección de errores**.

En este trabajo se han utilizado dos códigos de corrección de errores diferentes, un código BCH y un código LRRO. Sin embargo, su funcionamiento e implementación en el software es igual en ambos casos y se explica a continuación:

- En la función *loop*, en cuanto se obtienen los datos de temperatura y humedad se procede a la codificación de estos, llamando a una función *encoder* en la cual se ha implementado el código de codificación BCH o LRRO. Para ello se debe primero convertir el dato leído de tipo *float* en tipo entero de 32 bits.
- Luego se comprueba que los datos son correctos mediante la decodificación. Esto se hace llamando una función *decoder* en la cual se ha implementado el código de decodificación BCH o LRRO.

Se observa pues que los datos enseñados en la pantalla LCD son antes comprobados y así se asegura su integridad.

```
void loop() {

  // Leemos la humedad relativa
  float humedad = dht.readHumidity();
  uint32_t *h = (uint32_t *)&humedad;

  // Codificamos
  unsigned long temp_ECC = encoder(*h);

  //Comprobamos que el ECC es correcto
  unsigned long hum_decodificada = decoder(*h, hum_ECC);
  float *hum = (float *)&hum_decodificada;

  delay(10);

  // Leemos la temperatura en grados centígrados (por defecto)
  float temperatura = dht.readTemperature();
  uint32_t *t = (uint32_t *)&temperatura;

  // Codificamos
  unsigned long temp_ECC = encoder(*t);

  //Comprobamos que el ECC es correcto
  unsigned long temp_decodificada = decoder(*t, temp_ECC);
  float *temp = (float *)&temp_decodificada;

  delay(10);
```

Figura 15: Llamadas a las funciones encoder y decoder después de la lectura de los datos.
Fuente: Elaboración propia

Para entender un poco mejor el funcionamiento de los códigos de corrección de errores, se va a explicar qué hace cada función *encoder* y *decoder*. En la función *encoder*, lo que se hace es crear dos variables de tipo *unsigned long*, una que servirá para guardar los 32 bits del dato y otra que servirá para guardar los 12 bits de paridad. Así pues, se obtiene una palabra de 44 bits que corresponde con la palabra codificada. Por otro lado, en la función *decoder* se crean tres variables también de tipo *unsigned long*. La primera será la que corresponde con el síndrome y consta de 12 bits. Con el síndrome se obtendrá la palabra error que será de 32 bits. En la palabra error todo bit que esté a 1 indica que en esa posición se encuentra un error (error [5] = 1 → error en el bit número 5). Por último, se obtiene la palabra decodifica, de 32 bits, haciendo una OR exclusiva entre la palabra error y los bits de datos de la palabra codificada. Así es como se obtiene la palabra decodificada con los errores corregidos en el caso de que se hubiera dado alguno.

Los códigos BCH y LRRO, implementados en C++, me fueron facilitados por los tutores de este trabajo, que además son los diseñadores de los códigos LRRO. A la hora de implementarlos, se ha tenido que traducir el código original en C++ a Arduino ya que, en C++, para poder operar bit a bit, lo que se ha hecho es guardar el dato (temperatura o humedad) en un vector de bits en el

cual cada elemento representa un bit y, sin embargo, en Arduino existen funciones (*bitRead* y *bitWrite*) que permiten trabajar con bits concretos de un dato dado. Además, en C++, al usar vectores se pueden guardar todos los bits (los 32 del dato y los 12 de paridad) en un mismo vector y, sin embargo, en Arduino se han usado variables de tipo *unsigned long* que solo tienen una capacidad de 32 bits. Por lo tanto, se han tenido que utilizar dos variables diferentes, una para el dato y otra para los bits de paridad.

```
// Bit 0 (dato)
bitWrite(dat,0,bitRead(dec,0));
// Bit 1 (dato)
bitWrite(dat,1,bitRead(dec,1));
// Bit 2 (dato)
bitWrite(dat,2,bitRead(dec,2));
// Bit 3 (dato)
bitWrite(dat,3,bitRead(dec,3));
// Bit 4 (dato)
bitWrite(dat,4,bitRead(dec,4));
// Bit 5 (dato)
bitWrite(dat,5,bitRead(dec,5));
```

```
enc[12] = dec[0];
enc[13] = dec[1];
enc[14] = dec[2];
enc[15] = dec[3];
enc[16] = dec[4];
```

Figura 16: Código en formato Arduino (izquierda) y código en formato C++ (derecha). Fuente: Elaboración propia

Como puede observarse en la Figura 16, para acceder al bit número 3 de un dato **X** en Arduino se utiliza la función *bitRead(**X**,3)* y para poner el bit número 4 de dicho dato **X** a 1 *bitWrite (**X**,4,1)*. Los códigos BCH y LRRO originales (en C++) son bastante densos y realizar esos cambios puede resultar muy tedioso y sobre todo puede resultar en una gran pérdida de tiempo. Con el objetivo de ganar tiempo y facilitar esta traducción se ha pensado en automatizarla y así no tener que reescribir todo el código a mano. La solución por la que se ha optado es programar un código que sea capaz de traducir los códigos en C++ a Arduino con el formato deseado.

El lenguaje elegido para este programa ha sido Python por su fácil uso y por lo compacto que se podía hacer el programa en este lenguaje. Su funcionamiento se explica a continuación:

- Primero el texto original se introduce en una variable tipo *string*.
- Luego se guarda en un vector el texto original dividido por los corchetes.
- Luego se reemplaza el texto original por el texto nuevo en formato Arduino.
- Luego se vuelve a convertir a tipo *string*.
- Por último, se enseña por pantalla el nuevo texto, solo queda copiarlo y pegarlo en el IDE de Arduino.

De esta manera se ha ganado muchísimo tiempo, y aunque pensar y realizar el programa representa una inversión de tiempo inicial, a la larga sale mucho más rentable automatizar el proceso además de hacerlo mucho más ameno. Además, una vez hecho este tipo de programas, se puede adaptar fácilmente para traducir otros textos en un futuro de cualquier lenguaje de programación a otro.

En la Figura 17 se muestra el código que ha servido para traducir la parte del código que define el vector error a partir del vector síndrome, el cual es muy largo y sería muy tedioso de traducir a mano.

```
str=input('Texto original:  ')
print('\n')
p='sindrome'
print('\n')
f='bitRead(' + p + ', '
print('\n')
n=len(p)
texts=[]
a=str.split('[')
finaltext=''
for ai in a:
    t=ai.split(']')
    for ti in t:
        texts.append(ti)
for i,text in enumerate(texts):
    if p in text:
        text2=text.replace(p,f)
        texts[i]=text2

    elif text.isnumeric():
        text2=text+')'
        texts[i]=text2
for text in texts:
    finaltext+=text
print(finaltext)
```

Figura 17: Código de traducción de C++ a Arduino escrito en Python.
Fuente: Elaboración propia

5. PRUEBAS, RESULTADOS Y VALORACIÓN

En este apartado, se realizarán pruebas de inyección de fallos con el propósito de evaluar la capacidad del sistema para detectar, aislar y, en algunos casos, corregir errores. Estas pruebas se diseñarán para simular situaciones adversas o inesperadas que podrían afectar el funcionamiento del sistema en condiciones reales.

Posteriormente, se llevará a cabo una evaluación detallada del sistema en su conjunto, analizando su rendimiento, su fiabilidad y su capacidad para cumplir con los objetivos establecidos en este trabajo. Se examinarán los resultados obtenidos durante las pruebas y se realizará una valoración de la efectividad del sistema en términos de tolerancia a fallos.

5.1. Inyección de fallos

Una vez montada la estación e implementado todo el software, así como los ECCs, se ha de comprobar que estos funcionan y hacen su trabajo, es decir, que corrigen los posibles errores que puedan ocurrir en la memoria. Para asegurarse de que el código está efectivamente corrigiendo uno o varios errores hay que estar seguros de que dichos errores están ocurriendo y que el código no solo está codificando y decodificando los datos que ya son correctos. Es por ello por lo que se opta por inyectar fallos de manera intencionada, simulando fallos que podrían ocurrir de manera natural durante el funcionamiento normal de la estación.

La técnica de inyección de fallos [11] consiste en la introducción deliberada de fallos en el sistema para emular y evaluar su comportamiento frente a situaciones adversa. Este proceso se lleva a cabo de manera controlada, con el propósito de comprender cómo reacciona el sistema ante diferentes escenarios de fallo y qué medidas pueden implementarse para garantizar su robustez y confiabilidad.

Mediante la inyección de fallos, se pueden simular condiciones de operación anormales, como errores de hardware o software, interferencias internas o externas, entre otros, con el fin de evaluar la capacidad del sistema para detectar y corregir dichos fallos. Esta técnica proporciona información valiosa sobre la efectividad de los mecanismos de tolerancia a fallos implementados y ayuda a identificar posibles áreas de mejora en el diseño y la implementación del sistema.

Para llevar a cabo dicha inyección de fallos se ha tenido que programar una función *inyectar_fallo*. Dicha función recibe una variable y se encarga de cambiar uno o más bits (en este caso hasta un máximo de 6) a su valor opuesto, es decir, si un bit está a 1 lo cambia a 0 y viceversa. Cada cambio de bit simula un fallo en la memoria de tipo *bit-flip*. La elección de los bits a cambiar se hace de manera aleatoria gracias a la función *random* que ya viene incorporada en la librería de Arduino. Hay que tener en cuenta que, al trabajar con Arduino, como se ha explicado anteriormente, se han separado en dos variables los bits correspondientes al dato y los bits de paridad y, por lo tanto, a la hora de inyectar los fallos, habrá que contemplar ambos casos y probar tanto con los datos como con los bits de paridad.

Cuando solo se inyecta un fallo, la función *random* es suficiente para generar dicho bit a cambiar. Sin embargo, en los casos en los cuales se inyectan más de un fallo, se ha de comprobar que los bits generados aleatoriamente no se repitan, cosa que no se puede asegurar solamente

con la función *random*. Esto es muy importante ya que inyectar dos fallos en el mismo bit anula dicho fallo ya que el bit afectado cambiaría primeramente de 0 a 1 (o de 1 a 0) y luego volvería a su valor original cambiando de nuevo de 1 a 0 (o de 0 a 1) y, por lo tanto, por ejemplo, una inyección de 3 fallos en realidad solo inyectaría uno y esto falsificaría los resultados obtenidos en las pruebas.

Para asegurar pues la unicidad de los bits aleatorios seleccionados en los que se inyectará un fallo se ha añadido una función *buscar_array* la cual permite comprobar en qué bits ya se ha inyectado un fallo. El funcionamiento de dicha función es el siguiente:

- Previamente, en la función *inyectar_fallo* se ha de crear un vector vacío de longitud igual al número de fallos que se quiere inyectar.
- Luego, al llamar a la función *buscar_array* se envía dicho vector, así como el número aleatorio generado por la función *random*.
- Mediante un bucle *for* se va buscando en todos los elementos del vector si alguno coincide con el número aleatorio generado. Si se da el caso, se genera otro número aleatorio y se repite la operación. Si no, se añade dicho número al vector y se cambia el bit correspondiente del dato.

Dicha operación se repite tantas veces como fallos se quiere inyectar.

```
int buscar_array(int bits[], long busqueda, int longitud) {
    for (int x = 0; x < longitud; x++) {
        if (bits[x] == busqueda)
        return x;
    }
    return -1;
}

unsigned long inyectar_fallo(int numero_fallos, unsigned long dato) {
  int i;
  int bits[numero_fallos];
  long bit_aleatorio;

  for (i=0; i<numero_fallos; i++) {
    int buscar=32;
    while (buscar != -1) {
      bit_aleatorio=random(0,32);
      buscar=buscar_array(bits,bit_aleatorio,numero_fallos);
    }
    bits[i]=bit_aleatorio;

    bitWrite(dato,bit_aleatorio,(bitRead(dato,bit_aleatorio))^1);
  }

  return dato;
}
```

Figura 18: Funciones buscar_array e inyectar_fallo.
Fuente: Elaboración propia

A la hora de inyectar los fallos, se probarán varios casos como ya se ha comentado anteriormente:

- Fallos únicamente en los bits correspondientes al dato.
- Fallos únicamente en los bits de paridad.
- Fallos a la vez en los bits del dato y en los bits de paridad.

En todos los casos se probará con diferentes números de fallos inyectados, yendo desde 1 fallo hasta un máximo de 6 fallos. De esta manera se probará la mayor variedad posible de casos que se podrían dar durante el funcionamiento normal de la estación, y se tendrá una idea precisa y fiel de cómo reaccionan los ECCs frente a los fallos que se puedan dar.

Por último, hay que añadir que para comprobar que los ECCs funcionan, se va a controlar en qué bits se van a inyectar los fallos para posteriormente comparar bit a bit el dato codificado con el dato decodificado y observar si los bits afectados han sido corregidos o no. Se tendrá en cuenta si la corrección ha funcionado, y se calculará un porcentaje de éxito, así como el tiempo de ejecución y el espacio ocupado por el software, como ya se ha comentado en apartados anteriores. También es interesante tener en cuenta si se observan diferencias o similitudes dependiendo de los diferentes casos que se van a estudiar (fallos en los bits de datos o en los bits paridad).

5.2. Evaluación del sistema

Una vez explicados todos los fundamentos teóricos, el funcionamiento de la estación montada, el funcionamiento de los ECCs, cómo se van a realizar las pruebas y qué se va a observar, se puede pasar a la exposición de los resultados obtenidos y a su posterior análisis, el cual nos permitirá enunciar una conclusión y finalizar dicho trabajo.

Antes de realizar las pruebas que comprobarán el buen funcionamiento y la eficacia de los ECCs frente a los fallos, hay que comprobar, en un primer lugar, que la estación funciona correctamente cuando no se producen fallos y que cumple con su propósito. Es decir, que lee datos de temperatura y humedad, los muestra por una pantalla LCD y activa un LED RGB en función del valor de dicha temperatura y humedad.

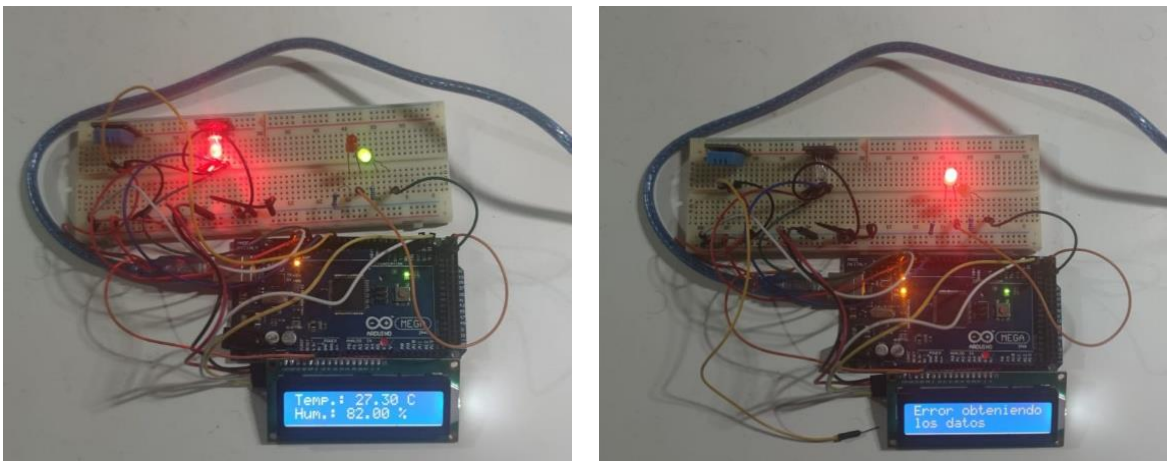

Figura 19: Comprobación del correcto funcionamiento de la estación.
Fuente: Elaboración propia

Lo primero que se va a comparar es el tamaño de cada código, es decir, cuanta memoria de la placa Arduino ocupa. No solo se han comparado los dos códigos de corrección de errores (BCH y LRRO) sino que también se ha comparado con el software de funcionamiento de la estación sin ningún código de corrección de errores añadido. Los resultados obtenidos son los siguientes:

	Espacio de almacenamiento	Memoria dinámica
Sin ECC	10740 bytes (4%)	509 bytes (6%)
Con BCH32	112986 bytes (44%)	513 bytes (6%)
Con LRRO32	86454 bytes (34%)	513 bytes (6%)

Tabla 1: Comparación del uso de memoria de los códigos BCH y LRRO.
Fuente: Elaboración propia

Notar que el espacio de almacenamiento máximo de la placa Arduino Mega 2560 es de 253952 bytes (250KB) y la memoria dinámica es de 8192 bytes (8KB). El porcentaje indicado entre paréntesis en la tabla se refiere a la proporción ocupada de esa capacidad.

Con esta primera comparación se observa que, como se había previsto, el código LRRO ocupa menos memoria que el código BCH. De hecho, dicha diferencia es bastante significativa ya que el espacio ahorrado es de 26532 bytes, lo que representa una reducción del 23,48% y un ahorro del espacio de almacenamiento del 10%, cifras para nada despreciables. Esto ya representa una gran ventaja, ya que permite utilizar el programa en microprocesadores con menos capacidad de memoria, o implementar programas más pesados y poder protegerlos con poco espacio.

```
El Sketch usa 10740 bytes (4%) del espacio de almacenamiento de programa. El máximo es 253952 bytes.
Las variables Globales usan 509 bytes (6%) de la memoria dinámica, dejando 7683 bytes para las variables locales.
```

```
El Sketch usa 112986 bytes (44%) del espacio de almacenamiento de programa. El máximo es 253952 bytes.
Las variables Globales usan 513 bytes (6%) de la memoria dinámica, dejando 7679 bytes para las variables locales.
```

```
El Sketch usa 86454 bytes (34%) del espacio de almacenamiento de programa. El máximo es 253952 bytes.
Las variables Globales usan 513 bytes (6%) de la memoria dinámica, dejando 7679 bytes para las variables locales.
```

Figura 20: Comparación del uso de memoria del código sin ECC (arriba), del código BCH (en medio) y del código LRRO (abajo).
Fuente: Elaboración propia

Lo siguiente que se va a medir es el tiempo de ejecución del programa sin ECC para tener una referencia que luego se podrá comparar con los códigos BCH y LRRO. Para los códigos BCH y LRRO se observará si el número de fallos a corregir influye en el tiempo de ejecución o si es independiente.

Para obtener el tiempo de ejecución del programa se ha optado por utilizar la función `micros()` propia de Arduino. Se toma el tiempo al principio y al final del programa y se hace la diferencia para obtener el tiempo de un ciclo del bucle. Además, se le resta el tiempo en el que el programa está parado por la función `delay()`, el cual en nuestro caso es de 4,02 segundos. Estos 4,02 segundos durante los cuales el programa está parado son necesarios para el correcto funcionamiento de este y están repartidos de la siguiente manera: en primer lugar,

se tiene el sensor DHT11 el cual proporciona una lectura cada 2 segundos con además un pequeño margen de tiempo extra de 0,02 segundos y en segundo lugar se detiene el programa a la hora de encender el LED RGB para que dé tiempo a ver los cambios de colores. Así, obtenemos el tiempo que tarda en ejecutarse el programa en cada ciclo. Los resultados obtenidos son estos:

```
69.728 milisegundos
69.752 milisegundos
69.784 milisegundos
70.788 milisegundos
69.792 milisegundos
69.816 milisegundos
70.836 milisegundos
69.800 milisegundos
69.804 milisegundos
```

Figura 21: tiempo de ejecución del programa sin ECC.
Fuente: Elaboración propia

Se procederá del mismo modo para probar los otros dos programas (con BCH y LRRO) los cuales se probarán sin fallos y posteriormente con fallos para ver si hay una correlación entre el tiempo de ejecución del programa y el número de fallos.

Después de probar con los dos ECC sin errores, se observa que el tiempo de ejecución aumenta en ambos casos, en unos 7 milisegundos para el ECC LRRO32 y en unos 10 milisegundos para el ECC BCH32, como se puede ver en la siguiente figura:

```
77.068 milisegundos     80.596 milisegundos
77.084 milisegundos     79.608 milisegundos
78.084 milisegundos     79.588 milisegundos
77.072 milisegundos     79.656 milisegundos
77.120 milisegundos     79.612 milisegundos
77.104 milisegundos     80.624 milisegundos
78.124 milisegundos     79.604 milisegundos
77.032 milisegundos     79.592 milisegundos
77.116 milisegundos     79.592 milisegundos
```

Figura 22: Tiempo de ejecución con LRRO32 (izquierda) y BCH32 (derecha) sin fallos
Fuente: Elaboración propia

También podemos observar que en el caso del ECC LRRO32, este es un poco más rápido que el BCH32, concretamente por unos 3 milisegundos, lo cual, a simple vista no parece representar una gran diferencia, pero puede resultar una gran ventaja si la cantidad de datos gestionados es muy elevada.

Antes de seguir con el estudio del tiempo de ejecución de los programas en función del número de fallos, se va a comprobar qué casos son los que los diferentes ECC (BCH32 y LRRO) son capaces de corregir.

Primero se comprobará cuantos errores en los bits del dato y en los bits de paridad (por separado) es capaz de corregir cada ECC. Para ello se lanzará el programa 500 veces para ver cuanto porcentaje de éxito tiene cada uno. Los resultados se exponen en la siguiente tabla, en la que se muestra el número de correcciones exitosas:

	Bits del dato		Bits de paridad	
	BCH32	LRRO32	BCH32	LRRO32
1 error	500 (100%)	500 (100%)	500 (100%)	500 (100%)
2 errores	500 (100%)	500 (100%)	500 (100%)	500 (100%)
3 errores	0 (0%)	0 (0%)	21 (4,2%)	29 (5,8%)
4 errores	0 (0%)	0 (0%)	0 (0%)	0 (0%)
5 errores	0 (0%)	0 (0%)	0 (0%)	0 (0%)
6 errores	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Tabla 2: Porcentaje de correcciones de éxito de los ECCs

Fuente: Elaboración propia

Como se puede observar, tanto para errores en el propio dato como para errores en los bits de paridad, el porcentaje de éxito es del 100% para uno y dos errores para ambos ECCs (BCH32 y LRRO32). En cuanto pasamos a tres o más errores, se puede ver que el porcentaje de éxito es nulo (en el caso de tres errores en el bit de paridad se observa un porcentaje de éxito en torno al 5% pero sigue siendo demasiado bajo).

El siguiente paso es estudiar el caso de errores en bits de datos y paridad a la vez. Viendo los resultados anteriores, solo se contemplarán estos casos:

- 1 error de paridad con 1 y 2 errores en el dato
- 2 errores de paridad con 1 y 2 errores en el dato

No tiene sentido estudiar más casos, ya que ya sabemos que el porcentaje de corrección será muy bajo o nulo. Después de realizar las inyecciones de fallos de dichos casos, obtenemos los resultados siguientes:

	BCH32		LRRO32	
	1 error en dato	2 errores en dato	1 error en dato	2 errores en dato
1 error en paridad	500 (100%)	41 (8,2%)	500 (100%)	30 (6%)
2 errores en paridad	0 (0%)	0 (0%)	28 (5,6%)	0 (0%)

Tabla 3: Porcentaje de correcciones de éxito con errores en dato y paridad

Fuente: Elaboración propia

De nuevo se puede observar que no influye si los errores están en el dato o en los bits de paridad. A partir de tres errores (sumando los errores en los bits de paridad con los errores en los bits de datos), tanto el BCH32 como el LRRO32 son incapaces de corregir dichos errores, con una tasa de éxito menor del 10% en estos casos.

Estos resultados son los esperados, ya que ya se sabía que tanto el BCH32 como el LRRO32 son ECCs diseñados para corregir errores simples o dobles aleatorios, tal y como hemos comprobado. También se puede añadir que no se ve una clara diferencia de efectividad entre el BCH32 y el LRRO32. Los dos códigos tienen un 100% de éxito para uno o dos errores, y una tasa cercana al 0% para tres o más errores. Por lo tanto, no será este un parámetro distintivo a la hora de valorar cuál de los dos códigos es mejor.

A continuación, se va a continuar con el estudio de los tiempos de ejecución ahora que ya se ha determinado qué casos es interesante estudiar (los que presentan una tasa de éxito del 100%). En este caso serán los siguientes:

- 1 error en el dato
- 2 errores en el dato
- 1 error en los bits de paridad
- 2 errores en los bits de paridad
- 1 error en el dato y 1 error en el bit de paridad

El objetivo es determinar si estos diferentes casos afectan en el tiempo de ejecución del programa, y también evaluar si hay una diferencia significativa entre los códigos BCH32 y los LRRO32 que favorezca el uso de uno sobre el otro.

En la siguiente tabla se pueden observar los tiempos de ejecución de los dos códigos en todos los casos mencionados anteriormente:

	BCH32	LRRO32
0 errores	80 ms	77 ms
1 error en el dato	81 ms	77 ms
2 errores en el dato	81 ms	77 ms
1 error en los bits de paridad	81 ms	77 ms
2 errores en los bits de paridad	81 ms	77 ms
1 error en el dato y 1 error en el bit de paridad	82 ms	78 ms

Tabla 4: Tiempos de ejecución de los ECCs con errores
Fuente: Elaboración propia

Como se puede ver, no parece que el hecho de que haya errores afecte al tiempo de ejecución del programa. En algunos casos, el tiempo de ejecución aumentó en 1 milisegundos, pero se puede deducir que seguramente sea debido a la ejecución de las funciones que inyectan los fallos (las cuales no estarían presentes en el uso normal de dichos programas).

Por lo tanto, el tiempo de ejecución no se ve afectado por la presencia de errores. Aun así, se sigue observando la diferencia de unos 3 milisegundos entre el BCH32 y el LRRO32 que, en el caso de tratar mucha información, podría traducirse en una diferencia de varios minutos entre ambos códigos.

Aquí acaba el estudio de los diferentes ECCs así como el análisis de los datos obtenidos mediante las diferentes pruebas realizadas. A continuación, se redactarán las conclusiones que permitirán evaluar cuales son las ventajas que ofrece cada ECC, si hay uno de los dos que destaca o si al final los dos presentan características similares.

6. CONCLUSIONES

Para finalizar el presente TFG, se presentan las conclusiones basadas en los diferentes datos obtenidos tras las pruebas realizadas anteriormente.

En primer lugar, se ha podido comprobar que la implementación de un sistema sencillo como es una estación meteorológica no es muy complicada con la ayuda de una placa Arduino. Con unos conocimientos básicos de dicha placa y del lenguaje empleado en Arduino, la implementación se puede realizar sin grandes dificultades. Obviamente, durante el proceso de integración de la programación de los distintos elementos se tuvieron que paliar algunos problemas de compilación que dificultaron un poco el avance del proyecto, pero con paciencia y trabajo se pudieron sobrellevar.

Por otro lado, se ha comprobado de manera empírica el correcto funcionamiento de los códigos de corrección de errores, tanto los BCH32 como los LRRO32, para uno o dos errores. Y, aunque su implementación en el código de la estación meteorológica ha sido un poco tediosa debido a la "traducción" que se ha tenido que llevar a cabo para pasarlo del lenguaje en C al lenguaje de Arduino, la automatización de dicho proceso ha ayudado a acelerar el avance y a reducir tareas repetitivas. Además, dicha automatización se podría extrapolar para "traducir" los ECCs a otros lenguajes si fuera necesario para otros proyectos.

Por último, y como se ha comentado anteriormente, se procede a evaluar cuál de los códigos de corrección de errores presenta mejores características y si vale más la pena usar uno frente a otro. Los diferentes parámetros que se han evaluado y que presentan diferencias son los siguientes:

- Memoria ocupada por el código.
- Tiempo de ejecución.

En cuanto a la memoria que ocupa cada código, se ha observado que el LRRO32 ocupaba 26532 bytes menos que el BCH32, lo que representa una disminución de casi una cuarta parte (23,48%). En ese sentido, es claro que el LRRO32 presenta muchas ventajas ya que permite ser implementado ocupando menos espacio y dejando por lo tanto más espacio para un código dedicado a otras tareas.

En cuanto al tiempo de ejecución, se ha observado una mejora de 3 milisegundos del LRRO32 con respecto al BCH32. Puesto que el tiempo de ejecución del programa sin ECC es de 70 milisegundos se deduce que el proceso de codificación y decodificación tarda 10 milisegundos con el código BCH y 7 milisegundos con el código LRRO. Por lo tanto, el código LRRO presenta una reducción del tiempo de ejecución del $\left(1 - \frac{7}{10}\right) * 100 = \mathbf{30}\%$ lo que es una mejora notable más aun sabiendo que se pretende ejecutar el programa un gran número de veces por lo que el tiempo ahorrado podría llegar a ser de varios minutos.

Por estas dos principales razones se estima que se ha de favorecer el uso del LRRO32 ya que tiene las mismas tasas de éxito en la corrección de errores que el BCH32, y además presenta las ventajas expuestas anteriormente, y por lo tanto se comprueba que se trata de una mejora frente a los BCH32.

Para concluir este trabajo se sugiere que un posible uso de dicha tecnología sería, por ejemplo, en una línea de producción en la que se podría tomar la temperatura de las diferentes máquinas que la componen y parar dicha producción si se sobrecalienta una. Implementar unos ECCs permitiría asegurar que dichos datos de temperatura son correctos y así evitar "falsas alarmas" y parar la producción cuando no fuera necesario. También se podría usar en el proceso de calidad del producto y así aumentar la precisión al asegurarse que los datos recogidos no tienen errores e incluso se podría repetir dicha verificación cuando se detecte un dato corrompido y así evitar el descarte de productos en buen estado o la validación de productos defectuosos.

Bibliografía

[1] Luis-J. Saiz-Adalid, Joaquín Gracia-Morán, Daniel Gil-Tomás, J.-Carlos Baraza-Calvo, Pedro-J. Gil-Vicente, «Reducing the Overhead of BCH Codes: New Double Error Correction Codes,» 11 Noviembre 2020. [En línea]. Available: https://www.mdpi.com/2079-9292/9/11/1897.

[2] Shu Lin, Daniel J. Costello, Error Control Coding, 2004.

[3] Alexis Hocquenghem, «Codes correcteurs d'erreus,» *Chiffres,* pp. 147-156, 1959.

[4] Raj Chandra Bose, Dwijendra Kumar Ray-Chaudhuri, On a Class of Error Correcting Binary Group Codes, 1960.

[5] Arduino, «Arduino Store,» [En línea]. Available: https://store.arduino.cc/products/arduino-mega-2560-rev3.

[6] Hardware Libre, «Hardware Libre,» [En línea]. Available: https://www.hwlibre.com/dht11/.

[7] Hardware Libre, «Hardware Libre,» [En línea]. Available: https://www.hwlibre.com/led-rgb/.

[8] Arduino, «Arduino,» [En línea]. Available: https://www.arduino.cc/en/about.

[9] Hardware Libre, «Hardware Libre,» [En línea]. Available: https://www.hwlibre.com/pantalla-lcd/.

[10] Arduino, «Arduino Doc,» [En línea]. Available: https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/.

[11] Alfredo Benso, Paolo Prinetto, de *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, 2003, pp. 159-176.

PORTADA DEL PRESUPUESTO

PRESUPUESTO

Presupuesto

Índice del presupuesto

Material para la estación meteorológica

Concepto	Cantidad	Coste unitario (€)	Subtotal (€)
Placa Arduino Mega 2560	1	38.50 €	38.50 €
Sensor DHT11	1	2.90 €	2.90 €
Led RGB	1	3.79 €	3.79 €
Led rojo	1	0.35 €	0.35 €
Led verde	1	0.35 €	0.35 €
Resistencia de 220 Ohmios	2	0.19 €	0.38 €
Pantalla LCD con módulo i2c integrado	1	7.49 €	7.49 €
Cableado	16	0.10 €	1.60 €
Cable USB tipo A/B	1	6.00 €	6.00 €
Coste Total			**61.36 €**

Tabla 5: Coste del equipo y del material

Equipo y software

Concepto	Amortización	Coste unitario (€)	Subtotal (€)
Portátil LG Ultra 15	8.00%	971.00 €	77.68 €
Licencia Office	41.00%	60.98 €	25.00 €
Licencia Windows	41.00%	11.60 €	4.76 €
Monitor HP M27f	4.00%	189.00 €	7.56 €
Ratón HP Travel Mini Bluetooth	4.00%	35.98 €	1.44 €
Coste Total			**116.44 €**

Tabla 6: Coste del equipo y del software.

Mano de obra

Concepto	Cantidad	Coste unitario (€)	Subtotal (€)
Ingeniero en prácticas	300	20.00 €	6,000.00 €
Ingeniero tutor UPV	20	60.00 €	1,200.00 €
Ingeniero tutor UPV	20	60.00 €	1,200.00 €
Coste Total			**8,400.00 €**

Tabla 7: Coste de la mano de obra

Presupuesto global

Concepto	Coste (€)
Material para la estación meteorológica	61.36 €
Equipo y software	116.44 €
Mano de obra	8,400.00 €
Coste Total	**8,577.80 €**
Costes Indirectos	5%
Coste Total	**9,006.69 €**
Costes IVA	21%
Coste Total	**10,898.09 €**

Tabla 8: Coste total del proyecto

PORTADA DEL PLIEGO DE CONDICIONES

PLIEGO DE CONDICIONES

Pliego de condiciones

Índice del pliego de condiciones

Material y equipo

En este apartado se diferenciarán dos partes:

- Hardware:
    - Por un lado, se necesita la placa Arduino Mega 2560 y todos los demás componentes que componen la estación meteorológica, que en este caso son: un sensor DHT11, un LED RGB, un LED rojo, un LED verde, dos resistencias de 220 ohmios, una pantalla LCD con un módulo I2C integrado y una serie de cables para poder conectarlo todo entre sí, así como al equipo informático.
    - Por otro lado, se necesita el equipo informático mencionado anteriormente para poder trabajar con la estación meteorológica, el cual tiene que contar con un puerto USB para poder conectar la placa Arduino.
- Software:
    - El programa mayormente usado ha sido el Arduino IDE aunque también se ha usado brevemente el Python IDE, concretamente a la hora de traducir los códigos de corrección de errores del lenguaje C++ al lenguaje propio del Arduino IDE.

Por último, el computador utilizado en este proyecto tiene las siguientes especificaciones:

- Tipo: Portátil
- Marca: LG Ultra 15
- Procesador: Intel(R) Core (TM) i7-10510U
- RAM: 16,0 GB (15,9 GB usable)
- Sistema operativo: Windows 11 Home
- SDD: 256GB

Cabe mencionar que los programas utilizados no necesitan de un equipo de grandes prestaciones para poder ser ejecutados y bastaría con tener un ordenador con un mínimo de 4GB de RAM, un procesador i3 y espacio suficiente en el disco duro. La decisión de usar dicho ordenador ha sido únicamente porque el alumno ya contaba con él.

Entorno de trabajo

Para este proyecto no encontramos unos requisitos específicos y por lo tanto el entorno de trabajo depende únicamente de la comodidad del alumno y de las preferencias que este pueda tener a la hora de trabajar.

Por ello, este proyecto se ha realizado en su totalidad en la casa del alumno gracias a la portabilidad del material y equipos utilizados.

PORTADA DE LOS PLANOS

PLANOS

Planos

Índice de planos

Arduino Mega 2560

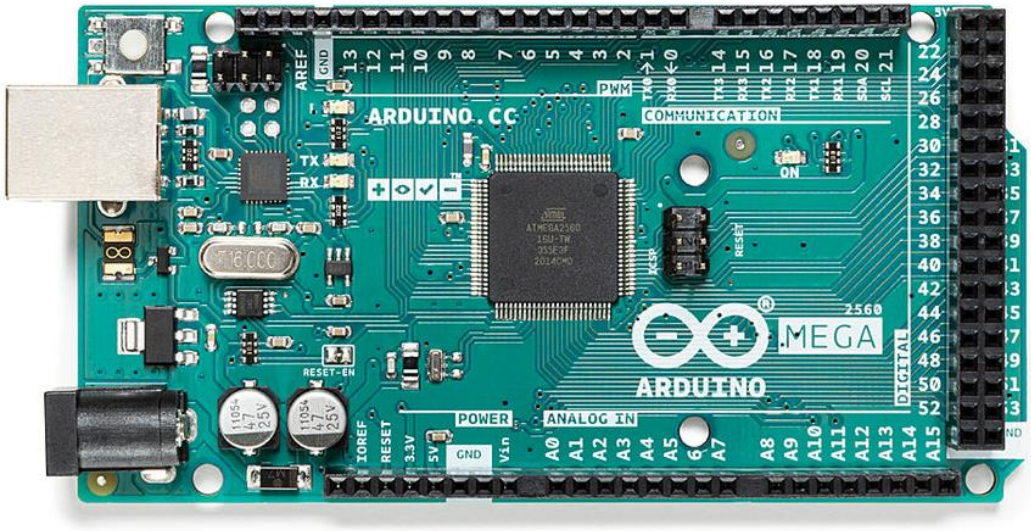

Figura 23: Placa Arduino Mega 2560

Fuente: https://store.arduino.cc/products/arduino-mega-2560-rev3

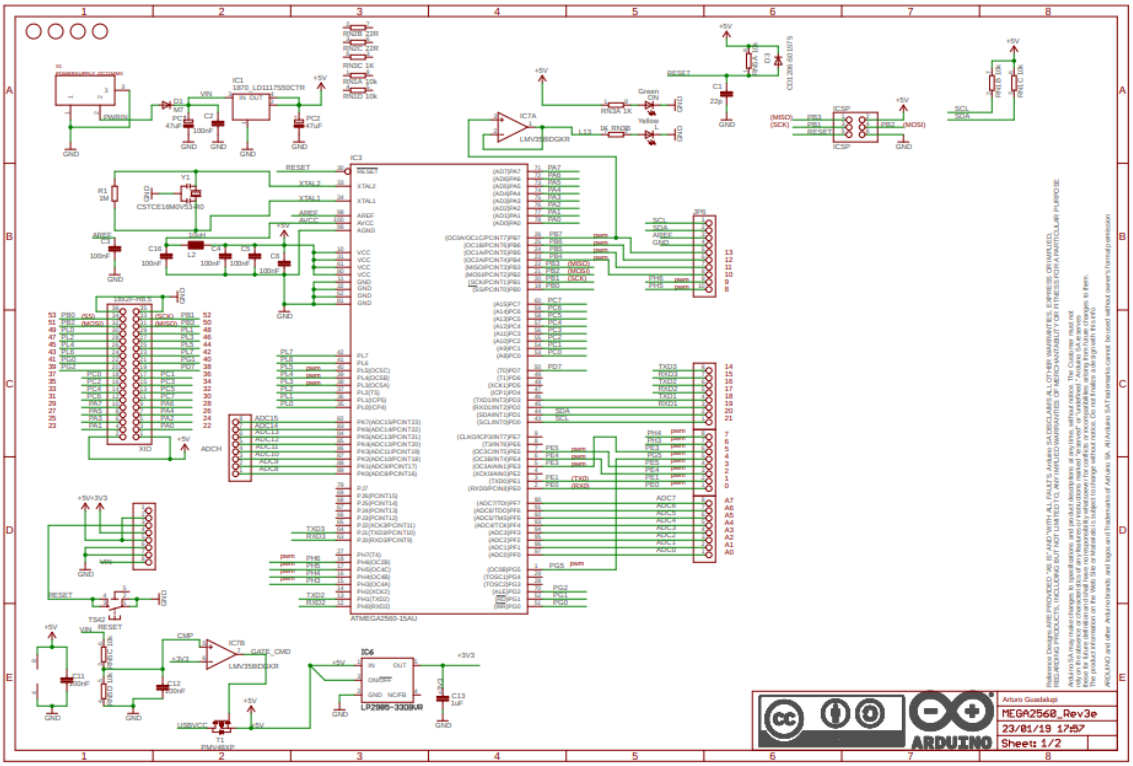

Figura 24: Esquema de referencia 1 de la placa Arduino Mega 2560

Fuente: https://store.arduino.cc/products/arduino-mega-2560-rev3

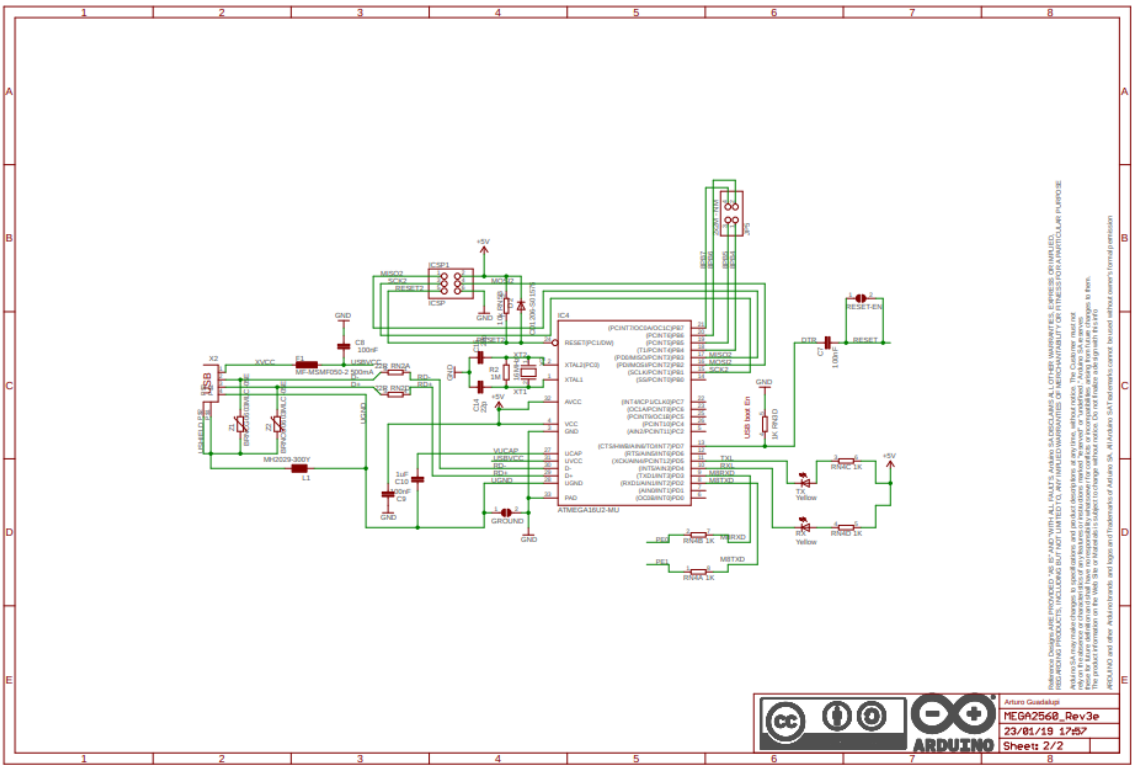

Figura 25: Esquema de referencia 2 de la placa Arduino Mega 2560
Fuente: https://store.arduino.cc/products/arduino-mega-2560-rev3

Sensor DHT11

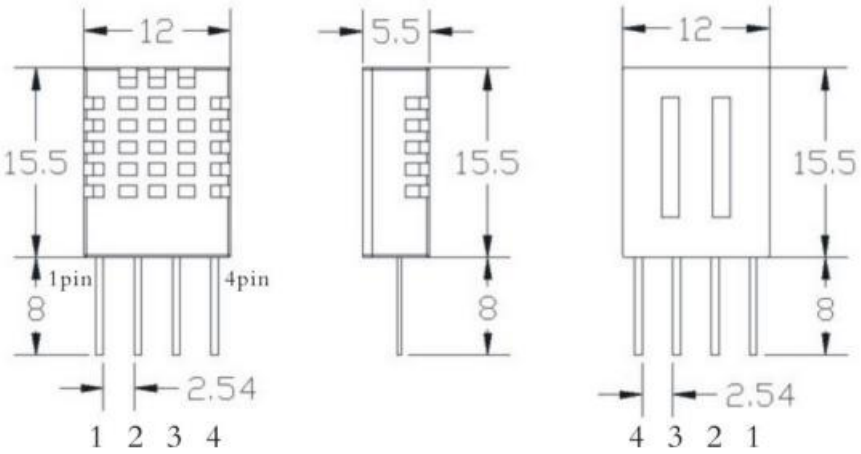

Figura 26: Plano del sensor DHT11
Fuente:
https://components101.com/sites/default/files/component_datasheet/DFR0067%20DHT11%20Datasheet.pdf

LED RGB

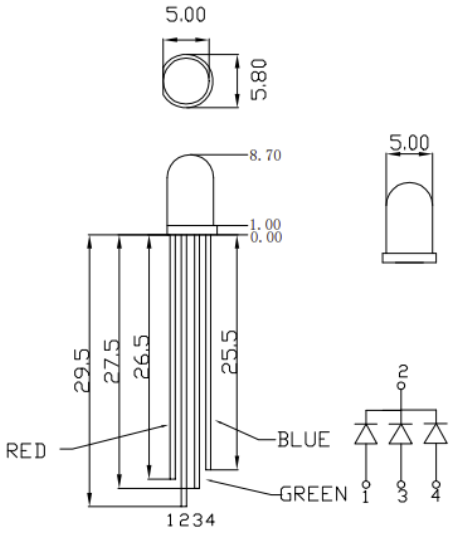

Figura 27: Plano del LED RGB

Fuente: https://components101.com/sites/default/files/component_datasheet/RGB%20LED.pdf

LED monocromo

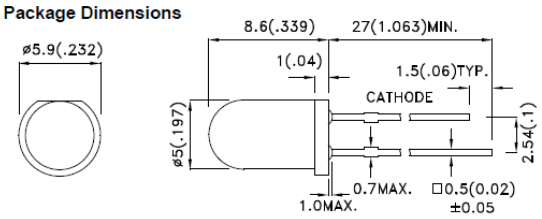

Figura 28: Plano de los LEDs rojo y verde

Fuente: https://asset.conrad.com/media10/add/160267/c1/-/en/000180144DS01/list-technickych-udaju-180144-kingbright-l-7113gc-led-s-vyvody-zelena-kulaty-5-mm-60-mcd-20-20-ma-22-v.pdf

Pantalla LCD

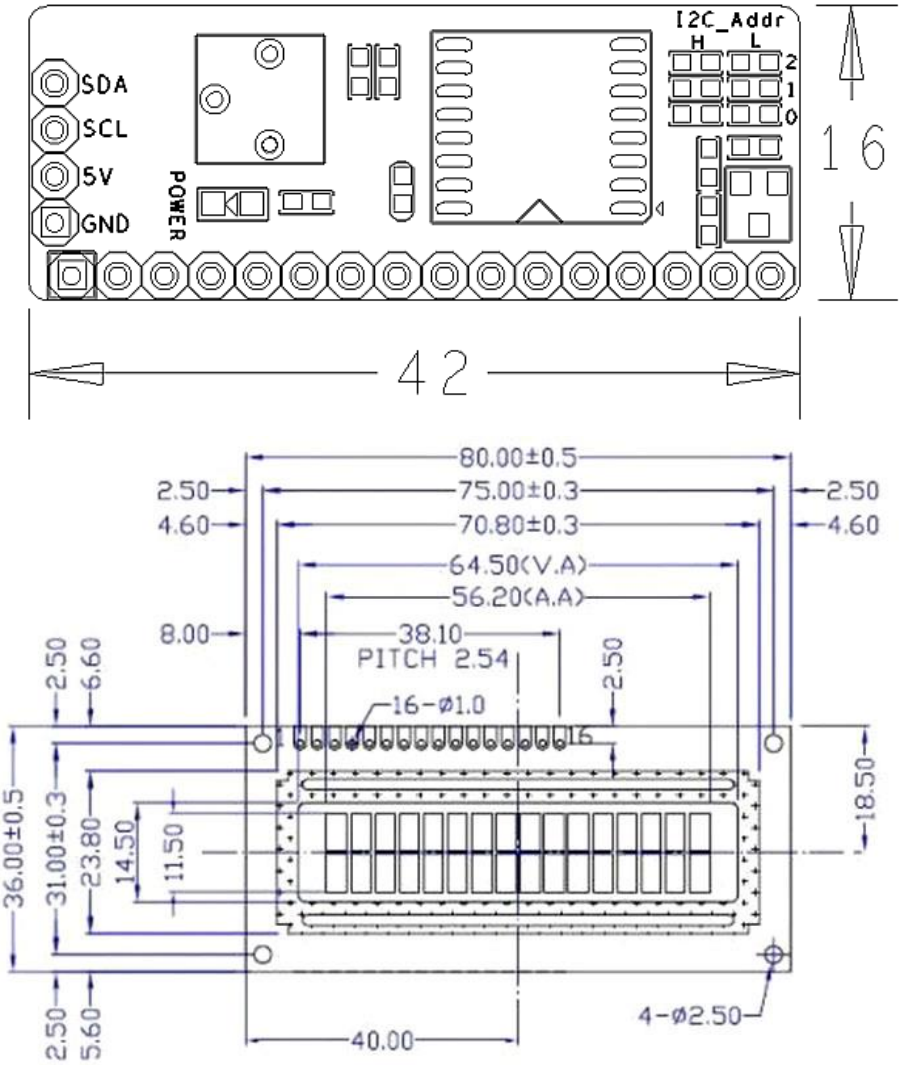

Figura 29: Plano de la pantalla LCD
Fuente: https://www.pic-control.com/lcd-display-alphanumeric-dot-matrix/

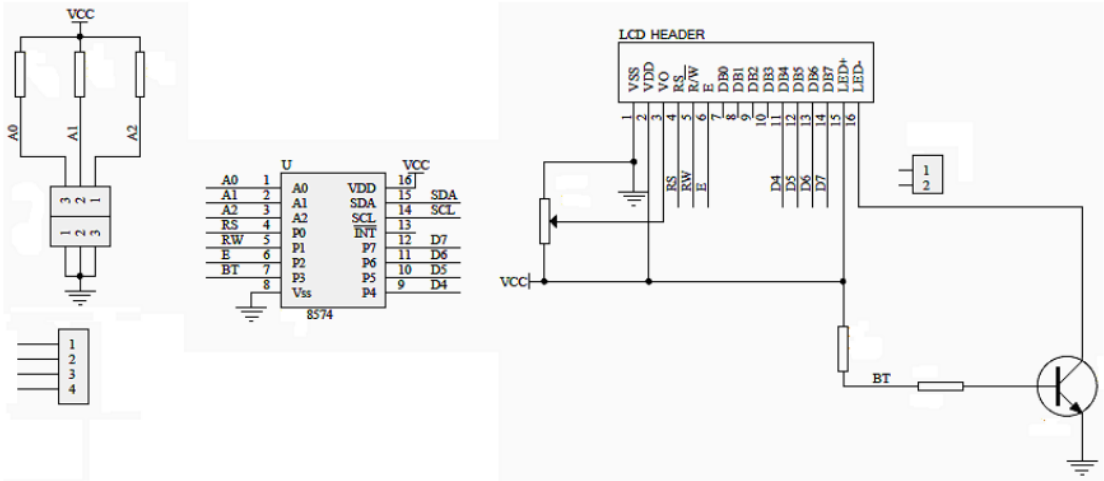

Figura 30: Esquema de referencia de la pantalla LCD
Fuente: https://www.electroschematics.com/arduino-i2c-lcd-backpack-introductory-tutorial/

PORTADA DEL ANEXO

ANEXO

Anejo

En este Anexo se adjunta un enlace al reportorio donde se pueden recuperar los tres códigos utilizados en el trabajo:

https://github.com/TomasSammut/TFG/