EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

Instituto Tecnológico de Chihuahua II

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

FRAMEWORK PARA LA CONSTRUCCIÓN DE SISTEMAS EMBEBIDOS IOT

TESIS

PARA OBTENER EL GRADO DE

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA

ING. OSCAR BELTRÁN GÓMEZ

DIRECTOR DE TESIS

M.C. ARTURO LEGARDA SÁENZ

CODIRECTOR DE TESIS

DR. RAFAEL SANDOVAL RODRIGUEZ

CHIHUAHUA, CHIH., DICIEMBRE 2023

Dictamen

Chihuahua, Chihuahua, 08 de febrero 2024

M.C. MARIA ELENA MARTINEZ CASTELLANOS

COORDINADORA DE POSGRADO E INVESTIGACION.

PRESENTE

Por medio de este conducto el comité tutorial revisor de la tesis para obtención de grado de Maestro en Sistemas Computacionales, que lleva el nombre de:

"FRAMEWORK PARA LA CONSTRUCCIÓN DE SISTEMAS EMBEBIDOS IOT", que presenta el C. OSCAR BELTRÁN GOMEZ., hace de su conocimiento que después de ser revisado ha dictaminado la APROBACIÓN de la misma.

Sin otro particular de momento queda de usted.

Atentamente

La Comisión de Revisión de Tesis.

M.C. ARTURO LEGARDA SÁENZ

Director de tesis

DR. RAFAEL SANDOVAL RODRIGUEZ

Co –Director

DR. HERNÁN DE LA GARZA GUTIÉRREZ

Revisor

Leonardo Nevárez Chávez

M.C. LEONARDO NEVAREZ CHAVÉZ

Revisor

Resumen

En los últimos años, el Internet de las cosas (IoT) se ha consolidado en una de las tecnologías más ampliamente utilizadas. En la actualidad, existe un extenso ecosistema dedicado al desarrollo de tecnologías IoT, en el que incluso los sectores de la manufactura y la industria participan activamente.

Esta tendencia ha generado una creciente demanda de herramientas de software destinadas a simplificar y acelerar el desarrollo de soluciones IoT. Es en este contexto los frameworks y lenguajes de alto nivel se han convertido en la elección más idónea para dicho propósito.

A partir de esto, es de esperar que surjan diversas dificultades, por ejemplo, la resistencia al uso de características avanzadas debido a su complejidad y el riesgo de introducir errores en el código (Li, 2018; Günter, 2018). Sin embargo, los marcos de trabajo bien establecidos ayudan a mitigar este problema; y aunque la creación de un framework para acelerar el desarrollo de soluciones IoT difiere en cierta medida de la filosofía de los enfoques convencionales, sigue siendo parte de las soluciones IoT y por lo tanto su desarrollo debe ser evaluado con las métricas y metodologías de ingeniería de software sin que su modelo de trabajo interfira en ello.

Esta tesis adopta una metodología iterativa que divide el proyecto en ciclos para abordar los requisitos del desarrollo de un “Framework o Micro-Framework para la Construcción de Sistemas Embebidos IoT”.

El framework que aquí se presenta esta diseñado siguiendo el “método de recolección” propuesto por **Martin Fowler** en su obra '**Harvested Platform**' (Fowler, Harvested Platform, 2003). Además, se destaca la pertinencia de utilizar **Arduino** como base del desarrollo software en combinación con la plataforma **PlatformIO** y la placa **ESP32** en el lenguaje **C++**, con el fin de facilitar la adopción del paradigma de Programación Orientada a Objetos (POO) y del Modelo de Actores.

Como se ve más adelante, uno de los objetivos es integrar el paradigma de la Programación Orientada a Objetos (POO) con el conocido framework “FreeRTOS” para la gestión de tareas concurrentes y paralelas.

Es importante destacar que este trabajo no busca mejorar el rendimiento de los sistemas IoT. En cambio, introduce su propia curva de aprendizaje. La propuesta de este marco de trabajo se enfoca en agilizar el tiempo de desarrollo, facilitar la interacción con otros lenguajes y simplificar la implementación de tareas concurrentes y paralelas.

Este trabajo respalda la idea de que la creación, adopción y uso de nuevos frameworks que cumplan con los estándares actuales resulta en una mejora significativa en la integración de IoT con otras tecnologías.

Abstract

In recent years, the Internet of Things (IoT) has established itself as one of the most widely used technologies. Currently, there is an extensive ecosystem dedicated to the development of IoT technologies, in which even the manufacturing and industry sectors actively participate.

This growth has generated a growing demand for software tools aimed at simplifying and accelerating the development of IoT solutions. It is in this context where high-level frameworks and languages have become the most suitable choice for this purpose.

In this sense, various difficulties are expected to arise, such as resistance to the use of advanced features due to their complexity and the risk of introducing errors into the code (Li, 2018; Günter, 2018). However, well-established frameworks help mitigate this problem. Although the creation of a framework to accelerate the development of IoT solutions differs to some extent from the philosophy of conventional approaches, it is still a valid application. Therefore, your proposal must be evaluated with current metrics and methodologies.

This thesis proposes an iterative methodology that divides the project into cycles to address the requirements of the development of a 'FRAMEWORK or MICRO-FRAMEWORK FOR THE CONSTRUCTION OF EMBEDDED IOT SYSTEMS'.

The framework presented in this context has been designed following the 'harvesting method' mentioned by Martin Fowler in his work 'Harvested Platform' (Fowler, Harvested Platform, 2003). Furthermore, the relevance of using Arduino as an integrated development environment (IDE) and hardware platform is highlighted, in combination with the PlatformIO platform and the ESP32 board that runs the C++ language, in order to facilitate the adoption of the Programming Oriented Paradigm. Objects (OOP) and the Actor Model.

As we will see later, one of our objectives is to integrate the paradigm of Object-Oriented Programming (OOP) with the well-known FreeRTOS framework for the management of concurrent and parallel tasks.

It is important to note that this work does not seek to improve the performance of IoT systems. Instead, it introduces its own learning curve. Our proposed framework focuses on speeding up development time, facilitating interaction with other languages, and simplifying the implementation of concurrent and parallel tasks.

This work supports the idea that the creation, adoption and use of new frameworks that comply with current standards results in a significant improvement in the integration of IoT with other technologies.

ÍNDICE

ÍNDICE DE TABLAS Y CÓDIGOS

ÍNDICE DE FIGURAS

I INTRODUCCIÓN

1.1 Introducción

El internet de las cosas (IoT) se ha convertido en una de las tecnologías más empleadas en los últimos años. Sin embargo, también se ha vuelto cada vez más compleja debido a la gran cantidad de dispositivos y tecnologías que intervienen en su proceso. Actualmente hay un gran ecosistema para el desarrollo de tecnologías IoT en donde incluso las áreas de la manufactura y la industria son partícipes.

A partir de esto, la demanda de herramientas de software, para simplificar y acelerar el desarrollo de soluciones IoT, va en aumento; es aquí en donde los frameworks y lenguajes de alto nivel se han convertido en la opción más apropiada para tal propósito. Por ende, cada vez es más importante abordar e innovar en sus métodos de desarrollo, ya que los elementos IoT van desde aplicaciones embebidas, pasando por servicios en la nube y brokers de mensajería (**AWS**, **Google Cloud** y **Azure**, por ejemplo), hasta los denominados endpoints o puntos finales que pueden ser aplicaciones residentes en computadoras, dispositivos móviles e incluso microcontroladores.

En este sentido, también es de esperar una gran cantidad de adversidades, una de las más importantes es el uso y adopción de lenguajes y framework de alto nivel, ya que, aunque ofrecen una abstracción que facilita la programación de los sistemas IoT también incluyen retos como la renuencia en el uso de las características avanzadas debido a su complejidad y a la posibilidad de errores en el código (Weimin Li, 2018) y (Günter, 2018). La creación de un framework dista un poco de la filosofía de los desarrollos convencionales; su funcionalidad depende de solventar las necesidades de otros sistemas (Fowler, Foundation Platform, 2003) y su especialidad también influye su comportamiento. Sin embargo, no deja de ser una aplicación que apoya a las tecnologías IoT en donde su análisis y dominio llevan a la elección y adaptación de la metodología de desarrollo más adecuada.

En esta tesis se plantea una metodología iterativa, la ventaja de esta metodología es que divide el proyecto en ciclos en donde cada uno tiene el propósito de agregar diferentes funcionalidades para cubrir los requerimientos; a continuación, se listan las iteraciones (tabla I.1) con respecto a los temas a desarrollar.

Tabla I.1.- Tabla de Iteraciones.

Tema/Subtema	No. Iteración
1- Programación Orientada a Objetos	
Integración de Clases	1
Herencia y polimorfismo	2
Lambdas	3
2- El modelo de Actores	
Colas o Queues	4
Sobrecarga de operadores	5
3- Protocolos IoT	
HTTP	6
MQTT	7
4 - Protocolos IIoT	
Modbus	8
5 – DSL o Lenguaje Específico de Dominio	
Encadenamiento de Métodos	9

La propuesta de esta tesis es crear un framework o micro-framework que integre características avanzadas de programación para lograr, en medida de lo posible, agilidad en la implementación de sistemas IoT.

El framework que aquí se presenta ha sido diseñado a partir del "método de recolección" mencionado por **Martin Fowler** en "**Harvested Framework**" (Fowler, Harvested Platform, 2003) de forma iterativa. Partiendo de un prototipo funcional "base" al que le han aplicado diferentes patrones y paradigmas de programación para obtener el conjunto de estructuras y librerías. Como es de imaginar, este proceso es recurrente y termina hasta alcanzar los requerimientos y/o métricas establecidas.

Se destaca la pertinencia de utilizar **Arduino** como base del desarrollo de software en combinación con la plataforma **PlatformIO** y la placa **ESP32** en el lenguaje **C++**, con el fin de facilitar la adopción del paradigma de Programación Orientada a Objetos (POO) y del Modelo de Actores.

Como se ve más adelante, uno de los objetivos consiste en incluir el paradigma POO con el ya conocido framework **FreeRTOS** para el manejo de tareas concurrentes y paralelas. Este conjunto de plataformas y librerías genera la posibilidad de usar una vasta cantidad de ejemplos en combinación con los mecanismos de concurrencia y paralelismo (en hardware y software), que ofrece la placa **ESP32** con **FreeRTOS** en **Arduino**.

Este trabajo no pretende mejorar el rendimiento de los sistemas IoT y, por el contrario, trae consigo su propia curva de aprendizaje, la propuesta de este marco de trabajo se centra en la idea de mejorar el tiempo de desarrollo y la interacción con otros lenguajes además de simplificar el uso en tareas concurrentes y paralelas.

Además, como se mencionó anteriormente, la propuesta de crear un framework para la integración de características avanzadas también conduciría a una mayor consistencia en el desarrollo. Esto se logra al alinear los mecanismos propuestos con los utilizados en contrapartes complementarías como: la nube, dispositivos móviles y entornos de escritorio.

Es tangible que este tipo de herramientas producen resultados positivos no solo en términos de la codificación, sino también en aspectos más amplios, como la comprensión del problema, la colaboración y la validación del sistema en sí mismo.

1.2 Planteamiento de Problema

Es evidente que, además del manejo de hardware, los dispositivos IoT generalmente abarcan dos aspectos fundamentales: la conectividad a Internet y su funcionalidad operativa (Guerrero-Ulloa, 2023). Estos aspectos, normalmente vistos como uno solo, deben de coordinarse y ejecutarse en el mismo segmento de tiempo (Fortino, 2017). Esta ejecución, ya sea concurrente o paralela, resulta crucial para la interacción entre las tareas de conectividad y las operativas además de facilitar la propagación y reacción a los cambios en su entorno.

Sin embargo, y a pesar de que el núcleo del Internet de las Cosas es el cómputo en la nube (por su capacidad de procesamiento superior en comparación a los dispositivos IoT) (Soman, 2017), todavía existe una brecha notable entre la implementación de este tipo de servicios y la programación de sistemas IoT.

Por ejemplo, los servicios en la nube incorporan técnicas y lenguajes de programación avanzados respaldados por metodologías y paradigmas igualmente avanzados; mientras que los sistemas embebidos que se utilizan en el desarrollo de sistemas IoT se centran mayormente en el rendimiento, dejando un poco más rezagadas estas técnicas y metodologías (Pacheco, 2021) (Guerrero-Ulloa, 2023).

Si bien se reconocen las diferencias naturales entre tecnologías, es importante destacar que los conceptos e ideas que implementan son transversales; por lo tanto, **“aún se necesitan frameworks y utilerías para IoT que reduzcan estas diferencias al proporcionar abstracciones de alto nivel como: programación orientada a objetos, concurrencia y paralelismo”**.

Se espera que, la creación y adopción de nuevos frameworks bajo los estándares vigentes disminuya significativamente la disparidad entre IoT y otras tecnologías.

1.3 Alcances y Limitaciones

1.3.1 Alcances

- **Mejorar el tiempo de desarrollo**. - Este punto se refiere a que el framework está enfocado a mejorar el tiempo de desarrollo usando y aplicando herramientas, patrones y paradigmas de programación modernos.

- **Mejorar la interacción con otros lenguajes y tecnologías**. - Las tecnologías IoT por su naturaleza y ejecución sobre internet interactúan a menudo con varios sistemas desarrollados en variados lenguajes de programación, es de esperar que la aplicación de los patrones y paradigmas mencionados en el alcance anterior, mejore también la interacción con otros lenguajes y/o tecnologías.

- **Simplicidad de uso en tareas concurrentes y paralelas**. - Este framework pretende, con elementos básicos pero potentes, cubrir el manejo de tareas concurrentes y paralelas.

- **Integración de protocolos IoT**. - Integra protocolos como **HTTP** y **MQTT;** estos últimos son importantes ya que agregan mecanismos de comunicación con los servicios en la nube.

- **Integración de protocolo Modbus**. - Se valida el uso del protocolo Modbus que, al igual que los protocolos anteriores, agrega los mecanismos de comunicación dispositivos y sistemas de carácter industrial.

1.3.2 Limitaciones

- **No pretende mejorar el rendimiento**. - Es un framework enfocado solo a las mejoras en la parte del manejo e integración software, deja fuera el rendimiento del hardware e incluso el ahorro de memoria, sin embargo, como ya se mencionará en las justificaciones se debe a la firme creencia de que el hardware avanzará a tal medida que solventará tales déficits.

- **Es una nueva adopción**. - A pesar de las ventajas que pueda ofertar la gran cantidad de frameworks disponibles también representa la adopción de diferentes herramientas y modos de uso, esto es una limitante si los usuarios del framework no comparten las ideologías o técnicas desarrolladas en el framework en cuestión.

- **Exceso de código para elementos simples**. - Este framework es fuertemente tipado y sus ejemplos más simples involucran exceso de código, si bien es solventable o de esperar en proyectos de gran tamaño, los códigos que integran tareas simples pueden llegar a representar mucho trabajo de codificación.

1.4 Justificación

Con el avance de las **TIC's** en cómputo en la nube, big data, aplicaciones móviles e inteligencia artificial (**IA)** los elementos físicos ahora pueden recopilar y propagar información entre sí y otros servicios sobre internet con más facilidad y pertinencia (C4IR.CO, 2021, pág. 13).

Así mismo, es cada vez más fácil suponer que el software tomará un mayor alcance e importancia en los sistemas IoT. Se puede respaldar tal idea al ver tantas y variadas empresas de servicios en internet apoyando y ofreciendo tecnologías IoT modernas (AWS IoT, 2023), (Intel IoT, 2023), (Huawei Cloud, 2023), (Rockwell Automation, 2023) sin depender de una plataforma de hardware en específico.

Sin embargo **Bernd Gunter**, de Thoughtworks (Günter, 2018), advierte que

"Hoy, en la ingeniería, a menudo se ve la fricción cuando el ***desarrollo de software ágil moderno*** *se encuentra con los buenos fabricantes de hardware" . . .*

. . . "Mientras tanto, los fabricantes de máquinas deben comprender que el software planificado no se puede pre-diseñar en papel tan fácilmente".

Reducir tal fricción, en medida de lo posible, mediante la implementación de un *"FRAMEWORK PARA LA CONSTRUCCIÓN DE SISTEMAS EMBEBIDOS IOT"* es una prioridad.

A partir de lo dicho por **Gunter** se entiende que toda mejora en la interacción entre los desarrollos de software moderno y el desarrollo de sistemas embebidos e IoT es importante, también se puede deducir que aún hay trabajo por hacer en la adopción de tales formas de trabajo, sin embargo, se considera posible.

Otro justificante es la asimilación tecnológica del IoT, del modelo de actores y de los aspectos como la concurrencia y paralelismo en los sistemas embebidos. Ya que todos estos conceptos y modelos convergen para formar plataformas con un mayor rendimiento; entonces, su integración, su uso y su estudio también importante.

El entrenamiento en estas tecnologías ya es un estándar en algunos países, además de que la mayoría de las empresas que ofertan estos servicios también ofrecen alguna especie de certificación (AWS, 2023) (Learn Microsoft, 2023), (IBM Training, 2023).

Tal vez el mayor justificante será la demanda y el abastecimiento del desarrollo tecnológico de sistemas embebidos en IoT; a pesar de la incertidumbre por la escasez de microcontroladores del 2021 para este 2026, se espera que el mercado de IoT alcance los **$650,5 billones de dólares 2026** (Markets and Markets, 2022), un excelente incremento en comparación a los **300 billones del 2021**; esta proyección da una buena idea del gran protagonismo de IoT en los siguientes años.

Por lo tanto, cualquier esfuerzo por crear o mejorar las herramientas, componentes y algoritmos de IoT es fundamental, ya que de ello dependen avances emergentes en rubros como la Industria 4.0, domótica y ciudades inteligentes (C4IR.CO, 2021, pág. 44).

1.5 Objetivo

1.5.1 General

El objetivo general es implementar un framework o micro-framework IoT para la placa **ESP32** con **FreeRTOS** (FreeRTOS™, 2023), como una interfaz para mejorar la integración y el uso de otros framework y librerías. Se enfoca en el desarrollo de aplicaciones de software embebidas de IoT sobre el paradigma de Programación Orientada a Objetos, esto como una serie de pequeñas librerías o segmentos de código comunicados a través de mensaje o eventos integrando servicios de Internet.

El framework pretende disponer de clases, objetos y segmentos de código que tengan la posibilidad de ser ejecutado de forma separada en su propio contexto (ya sea en su propio microprocesador o núcleo), comunicándose entre sí a través de eventos propagados localmente o a través de Internet. La idea es que estos códigos pueden evolucionar implementando mecanismos de concurrencia y paralelismo.

1.5.2 Específicos

Los objetivos específicos son:

- Integrar el paradigma de Programación Orientada a Objetos (POO) al framework para una mayor reutilización del software generado.
- Implementar el modelo de **Actores,** por medio del framework de **FreeRTOS,** para reproducir un comportamiento concurrente y paralelo del propio del modelo.
- Agregar soporte a los protocolos **HTTP** y **MQTT** para cubrir la comunicación sobre internet propio de las tecnologías IoT.
- Agregar soporte al protocolo **Modbus** como prueba de concepto y para cubrir la comunicación **IIoT**, elemento fundamental de las tecnologías **I4.0**.
- Agregar lenguaje especifico de dominio para guiar al usuario del framework en su uso.

II ESTADO DEL ARTE

2.1 Introduccón

El Internet de las Cosas (IoT) se ha convertido en una de las tecnologías más empleadas en los últimos años. Su avance ha sido tal que incluso del IoT, emergen arquitecturas de referencia, paradigmas computacionales especialidades y modelos como: Cloud, Fog, Edge Computing y la I4.0 (Mazon-Olivo, 2021), (Tai-hoon Kim, 2017).

La necesidad de conectividad de dispositivos inteligentes ha llevado al desarrollo de sistemas embebidos que sean capaces de procesar y comunicar información de manera rápida y eficiente a través de Internet.

Para el desarrollo de estos sistemas, se han utilizado frameworks y lenguajes que van desde **lenguaje ensamblador** (**ASM**), hasta de alto nivel como **C++, Python y Rust**, que ofrecen una gran variedad de herramientas de programación para dispositivos IoT.

A partir de esto y debido a la gran cantidad de dispositivos y tecnologías que intervienen en su proceso el desarrollo de sistemas IoT se ha vuelto cada vez más complejo, Por lo tanto, es necesario contar con herramientas de software que permitan simplificar y acelerar el desarrollo de estas soluciones. En este contexto, los frameworks y lenguajes de alto nivel se han convertido en la opción más apropiada para tal desarrollo.

2.2 Frameworks y Lenguajes IoT

En particular, los frameworks proporcionan un conjunto de herramientas y bibliotecas que simplifican el proceso de desarrollo permitiendo a los desarrolladores centrarse en la funcionalidad del sistema. En este sentido, existen varias opciones de IDE's y plataformas que también complementan el desarrollo de sistemas IoT; entre los más destacados se puede mencionar a: **Arduino IDE** (Arduino, 2023)**, Expressif** (Expressif, 2023)**, PlatformIO**

(PlatformIO, 2023)**, Eclipse IoT** (Eclipse Foundation, 2023) **y MPLAB X IDE** (Microchip, 2023).

Además, lenguajes como **C++**, ofrecen una abstracción que facilita la programación de los sistemas IoT, especialmente en términos de gestión de memoria y seguridad. No obstante, algunos desarrolladores pueden mostrar renuencia en el uso de las características avanzadas debido a su complejidad y a la posibilidad de errores en el código (Chatzigeorgiou A., 2002).

Otra característica importante en el desarrollo de sistemas IoT es la capacidad de gestionar tareas en tiempo real. Para ello, se utilizan sistemas operativos de tiempo real o **RTOS**, estos **SO** permiten ejecutar múltiples tareas de forma simultánea para garantizar la respuesta en tiempo real del sistema (Barry, 2018, pág. 2); en esta área **FreeRTOS** se destaca por ser ampliamente utilizado (Marketing, 2019) ya que además es la base de otros frameworks de desarrollo **RTOS**.

A partir de esto actualmente hay todo un ecosistema para el desarrollo de tecnologías IoT; plataformas que van desde desarrollo de aplicaciones embebidas pasando por servicios en la nube y brokers de mensajería (AWS, Google Cloud y Azure por ejemplo), hasta los puntos finales que pueden ser aplicaciones residentes en computadoras, dispositivos móviles o incluso microcontroladores, esto último en conjunto con la gran variedad de plataformas de hardware como Arduino, ESP8266, ESP32, STM32, Raspberry Pi, Beaglebone, Orange Pi, Odroid, Jetson Nano y Coral Google.

2.3 La Industria, la Estandarización y sus Retos

Por otro lado, la industria también ha adoptado las tecnologías IoT, por ejemplo, el artículo “A Resource Service Model in the Industrial IoT System Based on Transparent Computing” (Weimin Li, 2018) menciona que el concepto de IoT también puede proporcionar servicios inteligentes en sistemas y aplicaciones industriales, pero como toda tecnología su adopción depende de aspectos clave como la estandarización y el procesamiento de datos.

Weimin Li dice: *"la mayoría de los investigadores de IoT consideran que la falta de estándares es un gran problema. La estandarización de protocolos, arquitecturas e interfaces de programación de aplicaciones (API), que proporciona interconexión, interoperabilidad y compatibilidad entre objetos inteligentes heterogéneos, juega un papel fundamental en la solución de problemas*" (Weimin Li, 2018)

Esto aunado a que los sistemas IoT generan datos de forma masiva, dispara la necesidad de mejores y continuas implementaciones basadas en los estándares vigentes.

Por ejemplo, **Myers** afirma que la integración de sensores y el uso de herramientas de software como base de datos de series temporales permitirían la recopilación de grandes cantidades de datos que pueden ser utilizados para optimizar la producción y mejorar la calidad (Jason Myers Technical Writer, 2021) además de generar la capacidad de integrarse con soluciones modernas de TI. Las tecnologías IoT incluyen elementos esenciales para la industria, tales como: la automatización de procesos, recopilación y analizar datos en tiempo real, mantenimiento predictivo y más seguridad, en comunión con el contexto computacional que ofrecen las tecnologías IoT.

Sin embargo, aunque el Cloud Computing en IoT mitiga las limitaciones de IoT, creando incluso nuevos paradigmas como "**Thing as a Service**" (Zaslavsky, 2012) (B. Christophe, 2011), también trae consigo sus propios retos; la interoperabilidad, la seguridad y la privacidad son, hoy, los más importantes (Manuel Díaz, 2016).

2.4 El IoT y sus Aspectos Economico

En el entorno económico el IoT también tiene mucha presencia; por ejemplo aunque este año (2023) aún se ven lo estragos económicos de la pasada pandemia y **Leo Kelion**, Editor de Tecnología de la **BBC** (Leo Kelion, 2021), cuenta que desde el **2020** se tienen problemas en la producción de microchips ocasionado limitaciones en varias empresas tecnológicas, también se ve que la industria del IoT ha salido a flote; el reporte de **Markets and Markets** visualiza un incremento en el mercado de 350 billones de dólares para este 2026, más del

100%. Esto no es para menos si ve que cada vez más empresas, líderes en su ramo, apuestan por esta tecnología. A continuación, se listan algunas empresas que implementan y comercializan tecnologías y servicios para IoT.

- **AWS IoT** proporciona los servicios en la nube que conectan los dispositivos IoT a otros dispositivos y servicios en la nube de **AWS**, proporciona también software para dispositivos que puede ayudar a integrar los dispositivos IoT en soluciones basadas en **AWS IoT**.

- **Microsoft**: Ofrece una plataforma llamada **Azure IoT** que incluye servicios de análisis, conectividad y seguridad.

- **Google**: Tiene una plataforma denominada **Google Cloud IoT** que permite la conexión y gestión de dispositivos de IoT en la nube.

- **IBM**: Ofrece una plataforma de IoT llamada **IBM Watson IoT** que permite la conexión, gestión y análisis de datos de dispositivos de IoT.

- **Cisco**: La plataforma **Cisco Kinetic** permite la conexión y gestión de dispositivos de IoT en entornos industriales.

- **Intel**: Ofrece una plataforma nombrada **Intel IoT** que incluye herramientas de desarrollo, conectividad y seguridad.

- **Huawei**: Cuenta con **Huawei IoT** que permite la conexión y gestión de dispositivos de IoT en entornos industriales y empresariales.

Por lo antes escrito se espera que las tecnologías IoT continúen evolucionando y creciendo en los próximos años, impulsadas seguramente por la creciente demanda de soluciones de conectividad, una mayor adopción de tecnologías de inteligencia artificial y la expansión de la Industria 4.0 (Groombridge, 2022)**,** (Torres, 2020).

III MARCO TEÓRICO

En este documento, se ha presentado la esencia del IoT. Ahora es momento de explorar y estudiar en detalle el desarrollo que aquí se plantea. En este marco teórico, se analizan los conceptos y teorías que condujeron a la creación de este framework IoT.

El desarrollo de un framework difiere un poco de los desarrollos convencionales. En primer lugar, su funcionalidad radica en satisfacer las necesidades de otras aplicaciones como si fueran propias (Fowler, Foundation Platform, 2003). Sin embargo, en la construcción de un framework también se pueden emplear las mismas metodologías de desarrollo que se utilizarían en la creación de una aplicación estándar. **Martin Fowler** señala que, a medida que se desarrollan múltiples sistemas y se adquiere un dominio común, se pueden descubrir las señales un marco de trabajo (Fowler, Harvested Platform, 2003).

El framework que aquí se presenta ha sido diseñado a partir del **"método de recolección"** mencionado por **Fowler** en **"Harvested Framework"** (Fowler, Harvested Platform, 2003).

Para el desarrollo del Framework IoT, planteado en este documento, es necesario tomar algunas consideraciones o requerimientos; los siguientes temas describen tales necesidades.

3.1 Requerimientos Previos

Es necesario realizar un análisis previo con el fin de anticipar situaciones y necesidades. Esto implica investigar aspectos como: "ambientes de desarrollo", elección del "hardware o placa de desarrollo" a utilizar y las "librerías y plataformas de desarrollo" que podrían servir como base. Estos elementos son fundamentales para dar inicio al proceso de creación del framework.

A continuación, se describen tales requerimientos.

3.1.1 Ambiente de Desarrollo

Como ya se ha mencionado existen varios ambientes y plataformas de desarrollo de microcontroladores y de sistemas embebidos (Arduino, PlatformIO, MPLab, etc.), sin embargo, la elección del mismo depende muchas veces de la tarjeta de desarrollo y del tipo de lenguaje que se use para su programación. Cabe mencionar que en este trabajo se han seleccionado los ambientes de desarrollo de Arduino y PlatformIO.

Aquí es importante mencionar que, aunque es verdad que los requerimientos de las plataformas y software asociado se deben de asignar de forma minuciosa y discreta hoy se tiene la tecnología par que tal proceso se haga de forma automática o semiautomática. **PlatformIO** cuenta con los repositorios para gestionar la descarga e instalación de la paquetería necesaria según la elección del microcontrolador a usar (Boards PlatformIO, 2023).

3.1.2 Hardware de Desarrollo

Como en el caso anterior existen varias y excelentes alternativas de desarrollo, (Boards PlatformIO, 2023), (STM32 Nucleo Boards - Products, 2023), (Arduino Boards, 2023), la elección de las mismas depende de algunos factores como: el tipo de desarrollo, las capacidades de la placa, la aceptación de la misma en la empresa o industria o por el recurso humano disponible y en menor medida el precio.

La placa ESP32 (figura III.1) cumple con las expectativas para tal desarrollo, una breve descripción técnica se anexa en [Anexo 1]; pero una descripción breve muestra que cuenta con: 2 núcleos Xtensa® Dual-Core 32-bit LX6, una memoria de 520 KB de SRAM y 448 KB de ROM 4 Mg con conectividad Wifi y Bluetooth. Características por demás deseables para el desarrollo de sistemas IoT.

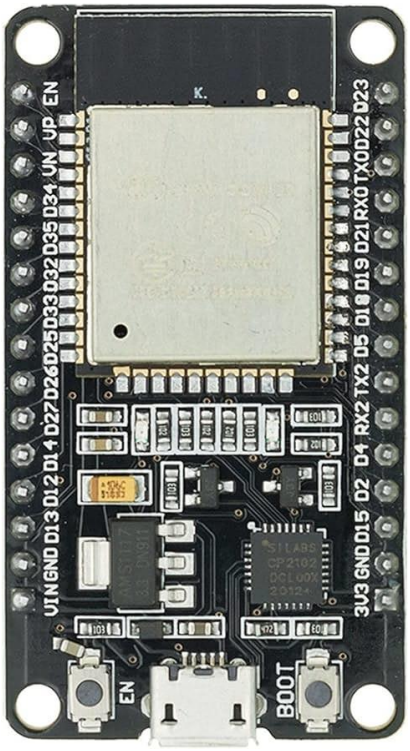


Figura III.1.- ESP32 de 30 Pins.

Sin embargo, para hacer un seguimiento más formal para este y cualquier desarrollo codificado hace falta un depurador de código o “Debugger”; Un depurador es una plataforma de software, hardware o ambas que se utiliza para detectar e identificar los errores en la codificación.

A partir de esto, el término ”depurador” o “debugger” se refiere a el proceso de detección de fallos de software conocido también como depuración.

La depuración de hardware y o sistemas embebidos trae consigo sus propias técnicas y estándares, en esta tesis se lleva a cabo la depuración con un “ESP32 ESP Prog” (figura III.2) bajo el estándar de **JTAG** (Joint Test Action Group), que es una especificación de hardware y protocolo de comunicación que se utiliza en la industria electrónica para llevar a cabo pruebas de hardware, depuración de circuitos integrados y programación de dispositivos embebidos.

Figura III.2.- ESP32 Prog JTAG.

Por otra parte, el ESP32 Prog es una herramienta de desarrollo y depuración desarrollada por Lexin, con funciones como la descarga automática de firmware, la comunicación en serie y la depuración en línea JTAG, se anexa un apartado [Anexo 2] la puesta en marcha del ESP32 Prog JTAG.

3.1.3 Plataformas y Librerías Básicas

En esta sección **Arduino** funge como plataforma, no como **IDE** ni como placa base de desarrollo, mientras que las librerías de **FreeRTOS** se usan para integrar las características de concurrencia y paralelismo en el framework.

3.2 Implementación y Pruebas

En esta etapa, se plantea el uso de una metodología iterativa. Esta metodología divide el proyecto en ciclos o iteraciones; cada ciclo va agregando funcionalidades para cubrir los requerimientos. Después de cada iteración, se revisan los resultados para ajustar y refinar el plan y el diseño antes de comenzar la siguiente iteración. La intención es que al término de este proceso se obtenga el conjunto de estructuras y librerías que conforman el framework.

La metodología seleccionada guía la creación del framework de forma sistemática, ya que, cada ciclo agrega nuevas mejoras al desarrollo para avanzar de forma más segura en el cumplimiento de las especificaciones. Normalmente la tabla o lista de iteraciones muestra el orden y los temas a desarrollar.

Se han considerado los siguientes temas y características en el desarrollo del framework.

- Orientación a Objetos.
- El modelo de Actores.
- Protocolos IoT e IIoT.
- DSL o lenguaje especifico de dominio.

A continuación, se da una breve introducción a los temas antes mencionados.

3.2.1 Orientación a Objetos

Este paradigma está muy difundido, hoy lo soportan e implementan los lenguajes de programación más usados como: **Java**, **Python** y **C++** (TIOBE Index Programming Language, 2023), además estos últimos también son muy usados en plataformas y servicios de IoT (AWS IoT Core Developer Guide, 2023).

En esta tesis, se usa **C++** como lenguaje base para la creación y desarrollo del ya mencionado framework IoT, C++ se basa en el lenguaje C e incluye todas las ventajas del mismo además de admitir conceptos orientados a objetos, tipos genéricos o plantillas, así como una amplia biblioteca de llamadas a función (C++ Reference, 2023).

Para desarrollar la parte de Programación Orientada a Objetos se propone los siguientes subtemas: Integración y manejo de clases, Herencia y Polimorfismo y uso de Lambdas.

3.2.2 El Modelo de Actores

Con múltiples tareas ejecutándose de forma concurrente o paralela la comunicación entre ellas se vuelve imprescindible (FreeRTOS Queues, 2023), (Digikey FreeRTOS Queue Example, 2023). **FreeRTOS** proporciona varios mecanismos de comunicación (FreeRTOS Inter-task Communication, 2023), las "colas de mensajes" son el más fácil de implementar, sin embargo, el uso de colas en **FreeRTOS** está muy abierto al uso e implementación del usuario final, si bien esto proporciona libertad creativa, cuando se habla de concurrencia es mejor tener un modelo. El modelo de **Actores** es la guía que se ha adoptado para este fin.

*"El término actor fue introducido por **Carl Hewitt** en el **MIT** a principios de la década de **1970** para describir el concepto de agentes razonadores.*
Se ha refinado a lo largo de los años en un modelo de concurrencia"
(Agha G. , 2004, pág. 13).

El modelo de Actores es considerado como un modelo matemático de computación simultánea que trata a los "actores" como los primitivos universales de la computación concurrente (Hewitt, 2017).

Un sistema definido por actores es una colección de actores (en pares) comunicándose entre sí por medio de paso de mensajes. Este modelo remplaza la concepción de los hilos y los problemas que conllevan, la coordinación de hilos se suplanta con la coordinación de actores, pero de una forma más eficiente ya que los actores solo pueden procesar los mensajes que pertenecen a su propia interfaz que entran a un buffer (cola) para esperar su turno. Este reemplazo de datos y estados compartidos por el paso de mensajes hace que el manejo de hilos desaparezca del diseño dando lugar a un conjunto unificado de comportamientos.

Un sistema de actores está formado por tres componentes básicos (Agha G. , 2004, pág. 43):

1. Los mensajes o estructuras de datos que se envían, normalmente, de forma "asíncrona" con información para su procesamiento.
2. Los buzones o recipientes que se comportan como colas para almacenar la información recibida para su posterior procesamiento.
3. El actor mismo**,** un proceso que funciona como unidad de proceso para reaccionar con respecto al mensaje recibido.

En este modelo, es factible crear actores como instancias de clase; este objeto, contaría con una cola de mensajes o buzón y un comportamiento reactivo.

Los mensajes también pueden ser instancias y representan parte del sistema de comunicación; los mensajes son intercambiados y almacenados en buzones en donde cada recepción esperaría su turno con el comportamiento del actor. Los actores pueden, también, enviar o reenviar mensajes a otros actores, instanciar o eliminar a otros actores y presentar un nuevo comportamiento.

La versatilidad de asumir comunicación síncrona o asíncrona genera un alto rendimiento al asumir un bloque o no de su ejecución. Otro aspecto importante es la independencia en el contexto, los actores de forma natural, no camparte su estado y solo ofrecen algunos métodos para su interacción (Hewitt, 2017, pág. 46), esto evita problemas como “deadlock’s” y problemas de integridad.

Una aplicación en tiempo real que utiliza un RTOS se puede estructurar como un conjunto de tareas independientes. Cada tarea se ejecuta dentro de su propio contexto sin dependencia coincidente de otras tareas dentro del sistema o del propio programador RTOS. Solo se puede ejecutar una tarea dentro de la aplicación en cualquier momento y el programador RTOS en tiempo real es responsable de decidir qué tarea debe ser. Por lo tanto, el programador RTOS puede iniciar y detener repetidamente cada tarea (intercambiar cada tarea dentro y fuera) a medida que se ejecuta la aplicación.

Como una tarea no tiene conocimiento de la actividad del programador de RTOS, es responsabilidad del programador de RTOS en tiempo real garantizar que el contexto del procesador (valores de registro, contenido de la pila, etc.) cuando se intercambia una tarea es exactamente el mismo que cuando la misma tarea fue intercambiada. Para lograr esto, cada tarea cuenta con su propia pila (FreeRTOS RTOS Implementation, 2023), para obtener más información (FreeRTOS Tasks and Co-routines, 2023).

En POO es común y práctico modelar objetos con métodos o comportamientos de entrada y de salida que definen un protocolo simple y semántico de comunicación, este es el enfoque adoptado en el modelo de actores (Agha G. , 2004). Los actores, pueden ser modelados con objetos que envuelvan el comportamiento de las tareas de FreeRTOS, y funcionarían como componentes autónomos, interactivos e independientes de un sistema informático que se comunican mediante el paso de mensajes asincrónicos” (Agha G. , 1990, pág. 128).

Las acciones o primitivas básicas del comportamiento de un actor son: “create”, “sent_to” y “become” y el cambio de estado que se especifica usando “comportamientos de reemplazo” expuestas comúnmente con la implementación de métodos y con el uso de funciones

"lambda". Este comportamiento, aunque básico, da pie a abstracciones y paradigmas de alto nivel en programación concurrentes (Agha G. , 1990, pág. 128).

En este modelo, cada vez que un actor procesa un elemento de su buzón, también calcula su comportamiento en respuesta al próximo envío que pueda procesar (Agha G. , 1990, pág. 128). Este comportamiento se puede fácilmente representar como un diagrama de arribo de eventos en donde la recepción o envío de un mensaje representa los eventos ocurridos entre actores.

Una función "Lambda" es una función anónima que puede tomar el carácter de objeto y que puede ser implementada en línea; es esta última característica la que permite el cambio de "comportamientos de reemplazo", además que las lambdas capturan el estado del contexto al momento de ser implementadas (C++ Reference Lambda Expressions, 2023).

3.2.3 Modelo de Clases y Objetos

Es más que apreciable la noción presentada por Fowler de lo fundamental de un objeto, su descripción menciona que un objeto:

> *"es la combinación de datos y comportamientos con un contexto de datos común al escribir un conjunto de funciones relacionadas.*
>
> *También proporciona una interfaz para manipular los datos que permite al objeto controlar el acceso a esos datos, lo que facilita el soporte de datos derivados y evita modificaciones no válidas de los datos"* (Fowler, Function As Object, 2023)

Como se deduce de las palabras de Fowler, las clases definen abstracciones de la realidad bajo un dominio o contexto, en este caso se usa esa noción para representar las tareas de FreeRTOS bajo el paradigma de la programación orientada a objetos. El uso de clases que representen a las tareas de FreeRTOS como objetos agrega ventajas a la implementación; la

primera es, sin duda, la conceptualización de las tareas como objetos con un comportamiento definido.

Por ejemplo, a una tarea de FreeRTOS referenciada como objeto se le podría asignar comportamientos como: una prioridad, de nombre, o incluso afectar su ciclo de vida, todo desde una misma "referencia" con la ventaja de poder definir métodos nuevos que agreguen comportamientos a la clase.

3.2.4 Concurrencia y Paralelismo

La concurrencia y el paralelismo son conceptos sobresalientes para los sistemas embebidos y se refieren a la capacidad de ejecutar múltiples tareas o procesos de forma "simultánea".

La concurrencia se refiere a la capacidad de ejecutar varias tareas o procesos dividiéndolos en segmentos ejecutables que pudieran tener acceso a un lapso del tiempo del procesador tan pequeño que al ser combinado con los otros segmentos ejecutables de las demás tareas crea la emulación de ejecución simultanea de las tareas involucradas. La figura III.3 muestra el esquema de ejecución concurrente de FreeRTOS.

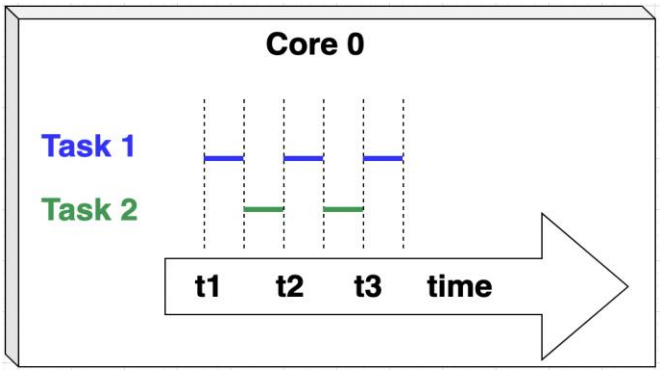


Figura III.3.- Esquema de ejecución concurrente de FreeRTOS.

La concurrencia, entonces, se logra mediante la ejecución intercalada de los segmentos resultado de la división de cada una de las tareas a ejecutar en forma concurrente.

Por otro lado, el paralelismo implica la “ejecución simultánea real” de múltiples tareas o procesos, donde cada uno se ejecuta en un hilo de ejecución o núcleo de procesador separado. En otras palabras, las tareas se ejecutan al mismo tiempo, aprovechando los recursos de hardware disponibles.

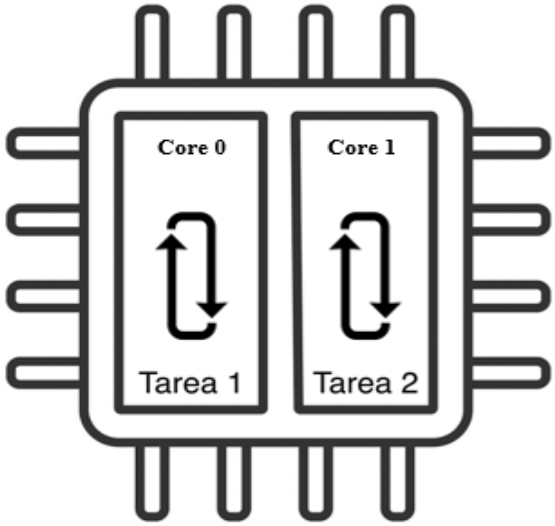


Figura III.4.- Esquema de múltiples núcleos.

La combinación de FreeRTOS y ESP32, logra las dos implementaciones, concurrencias y paralelismo, esta última mediante la asignación de diferentes tareas a los núcleos independientes del ESP32. La figura III.4 muestra a groso modo el esquema de múltiples núcleos y como se pueden asignar diferentes tareas a los mismos.

A partir de este punto cada funcionalidad y necesidad como: la integración de Programación Orientada a Objetos (POO), el manejo de mensajes como mecanismo de comunicación, protocolos de internet y protocolos industriales además de una semántica estandarizada se implementan una a una en diferentes iteraciones para dar forma y esencia al marco de trabajo.

3.3 Protocolos IoT

Con tantas opciones se podría suponer que la elección de un protocolo de comunicación debería ser una tarea sencilla, pero no es así; la elección adecuada es un tema sensible incluso para la tecnología IoT; los protocolos pueden variar el comportamiento de las aplicaciones bajo diferentes estados de red (M. O. Al Enany, 2021), entonces su análisis es parte fundamental del diseño.

En los siguientes párrafos se habla de la esencia de los protocolos HTTP y MQTT, ya que son considerados los protocolos de comunicación base para el desarrollo de la mayoría de las aplicaciones IoT.

3.3.1 El protocolo HTTP

HTTP, de sus siglas en inglés: "Hypertext Transfer Protocol", es el protocolo cliente-servidor más básico en la web para realizar una petición de datos y/o recursos; estos recursos van desde archivos HTML hasta imágenes, sonidos o videos. Funciona con una petición que es iniciada por el cliente, normalmente un navegador Web, y termina con una respuesta del servidor con el recurso solicitado (Developer Mozilla).

Aunque los mecanismos de solicitud-respuesta del HTTP pueden resultar provechosos para las tecnólogas web actuales, para algunos dispositivos con bajos recursos, como los IoT; un protocolo sin estado que necesita enviar un mensaje de solicitud varias veces hasta recibir la confirmación del remitente (server) puede llegar a representar un sobre trabajo (Cao Vien Phung, 2021). A pesar de todo, el uso del HTTP en IoT no es cuestionado, los beneficios que proporciona con la interoperabilidad entre la Web y el IoT son mayores, esto es porque desde el principio las implementaciones IoT estaban orientadas hacia la Web (Navid Shariatzadeh, 2016).

El HTTP es el soporte y la base para otros mecanismo y arquitecturas en donde la más usual en IoT es la arquitectura REST o API-REST. La mayoría de los dispositivos IoT no están

directamente relacionados entre sí, por el contrario, están conectados a través de API's (interfaz de programación de aplicaciones) o servicios que proporcionan interfaces a otras aplicaciones o incluso usuarios. Estas interfaces establecen un contrato entre dos aplicaciones y cómo serán las acciones entre sí.

Es normal usar las API REST como "middleware's" o intermediarios entre dispositivos IoT, ya que exponen de forma segura los recursos contenidos garantizando la transferencia de datos a través de las regulaciones de la red. El trasfondo de las API's REST es utilizar el protocolo HTTP para realizar operaciones, como: leer, crear, modificar y eliminar mediante los verbos "get", "post", "put" y "delete" descritos en la propia implementación HTTP (HTTP Working Group, 1999, pág. 35).

Una API REST representa adecuadamente muchos tipos de aplicaciones, incluidas las IoT. Para las IoT las API's pueden propagar y conservar los datos del contexto, por lo que la disponibilidad de la información está en función de las llamas a cada una de las API's. (Richa Maurya, 2021)

3.3.2 El protocolo MQTT

MQTT.org describe al protocolo como:

> *"Un protocolo de mensajería estándar de **OASIS** (uno de los organismos de estándares sin fines de lucro más reconocidos del mundo) para Internet de las cosas (IoT). Diseñado como un transporte de mensajería de publicación/suscripción extremadamente liviana que es ideal para conectar dispositivos remotos con un espacio de código pequeño y un ancho de banda de red mínimo. Hoy en día se utiliza en una amplia variedad de industrias, como la automotriz, la manufactura, las telecomunicaciones, el petróleo y el gas"* (Mqtt.org, 2023)].

Mientras que la página principal de **OASISI** lo secunda mencionando que es un

"protocolo de transporte de mensajería confiable de publicación/suscripción liviana adecuado para la comunicación en contextos M2M/IoT donde se requiere una huella de código pequeña y/o el ancho de banda de la red es primordial."
(Richard Coppen, 2020)

Como se puede apreciar, el MQTT es uno de los protocolos más utilizado para el IoT e IIoT; las reglas del protocolo definen que las aplicaciones y dispositivos pueden publicar y suscribirse a los servicios MQTT. Como es de suponer, la base del protocolo es el patrón de diseño publicación/suscripción (Pub/Sub); aquí el remitente (Publicador) y el receptor (Suscriptor) se comunican a través del envió de mansajes etiquetados por temas o tópicos.

Este esquema evita un alto acoplamiento ya que las conexiones entre los dispositivos se manejan por un intermediario o server MQTT que filtra todos los mensajes entrantes y los distribuye a los suscriptores mediante los ya mencionados tópicos (Hivemq, 2023), generando un esquema en que rara vez las aplicaciones tengan que ver directamente con las demás aplicaciones o módulos del sistema.

Los elementos más básicos de la especificación MQTT son:

Conexión de red. - Una construcción proporcionada por el protocolo para conectar a los clientes con el servidor que proporciona los medios para enviar un flujo de bytes ordenado y sin pérdidas en ambas direcciones.

Mensajes de la aplicación. - Especifica los datos y metadatos transportados por el protocolo a través de la red, Los metadatos incluyen, por supuesto, la calidad del servicio, el tema o tópico de subscripción, así como algunas propiedades.

Cliente. - Un programa o dispositivo que utiliza MQTT en el siguiente orden:

- abre la conexión de red al servidor
- publica mensajes de la aplicación que podrían interesar a otros clientes.

- se suscribe para solicitar mensajes de aplicación que le interese recibir.
- cancela la suscripción para eliminar una solicitud de mensajes de la aplicación.
- cierra la conexión de red al servidor.

Servidor. - Un programa o dispositivo que actúa como intermediario entre los clientes que publican mensajes de aplicación y clientes que han realizado suscripciones, un servidor puede:

- aceptar conexiones de red de clientes.
- aceptar mensajes de aplicación publicados por clientes.
- procesar las solicitudes de suscripción y cancelación de suscripción de los clientes.
- reenviar mensajes de aplicación que coincidan con las suscripciones de clientes.
- cierra la conexión de red de los clientes conectados.

Se puede ver el capítulo 1 de la especificación en el anexo 3 (OASIS Standard, 2019, pág. 11)

Como se abordará en la implementación, a través de este protocolo el intercambio de información puede tomar la esencia de otros conceptos más abstractos como: avisos, eventos, estímulos y alertas.

3.4 Protocolos IIoT

La incorporación de tecnologías IoT en los procesos de manufactura es fundamental para la evolución hacia la Industria 4.0 (M. Wollschlaeger, 2017).

Aunque tradicionalmente, los sistemas industriales han progresado de forma independiente de las Tecnologías de la Información y Comunicación, centrándose en demandas como el procesamiento en tiempo real y la disponibilidad, la colaboración con las TIC se ha convertido en un aspecto crítico y cada vez más importante.

Los dispositivos que se conectan a Internet con fines industriales generalmente se engloban en la categoría del Internet Industrial de las Cosas (IIoT). Con la fusión en curso de las TIC y los esquemas de producción industrial, se cada vez más interoperabilidad con protocolos industriales como: Modbus, Profinet y OPC UA.

Por ejemplo, el protocolo Modbus agrega robustez y simplicidad, permitiendo a la comunicación efectiva entre dispositivos en entornos industriales (NATIONAL INSTRUMENTS CORP, 2022). Profinet, por otro lado, ofrece capacidades en tiempo real y es ampliamente adoptado para la automatización de procesos complejos debido a su alta velocidad y eficiencia (Profibus, 2023). OPC UA (Open Platform Communications Unified Architecture) (OPC Foundation, 2023) complementa estos protocolos al proporcionar un marco de comunicación seguro y flexible que es fundamental para el intercambio de datos en la Industria 4.0.

La implementación de protocolos industriales en sistemas embebidos para la comunicación con redes de computadoras es importante porque sus estándares aseguran una mayor interoperabilidad y eficiencia incorporando elementos como: seguridad, supervisión y escalabilidad bajo normativas y regulaciones.

3.5 DSL o Lenguajes Específicos de Dominio

Podría decirse que un Lenguaje Especifico de Dominio o DSL es la formulación, recopilación y creación de modelos de un dominio en particular utilizando un lenguaje focalizado y especializado sobre el mismo dominio que permite a su vez especificar la soluciones usando directamente conceptos del dominio del problema generando estas especificaciones de alto nivel (Claudia Pons, 2010, pág. 37). Sin embargo, en un sentido más práctico, también se percibe como un conjunto de construcciones e instrucciones para un dominio particular.

Los DSL’s, en comparación con los lenguajes de propósito general (LPG) suelen ser más reducidos, pero también sujetos a menos errores, aunque, la verdadera ventaja que brindan los DSL en comparación con los LPG es plantear la solución del sistema en los términos del

mismo dominio (Mariano Luzza, 2012). Otra ventaja es que al integrar un DSL en el framework se agrega a su vez una guía de uso, semántica e incluso una metodología de desarrollo.

El ejemplo más emblemático de un DSL es el ya conocido SQL o Lenguaje Estructurado de Consultas, en donde el dominio de este lenguaje son los datos y la manipulación de los mismos; por ejemplo, la sentencia "select * from table where id = '34'" es una clara expresión de la interacción que se tiene con el dominio y el paradigma. Traduciendo la sentencia del idioma ingles al español, más el significado de su simbología, se obtiene la siguiente frase: "selecciona todos los registros de la tabla en donde su id sea igual a 34", como se puede ver la sentencia habla por sí misma, pero también se nota que hay terminología especializada, esta última es una propuesta del autor y pude llegar a carecer de generalidad, sin embargo, las buenas abstracciones de DSL's llegan a convertirse en verdaderos estándares.

Los DSL's se pueden clasificar en internos y externos. El DSL interno se implementa dentro de un lenguaje anfitrión y generalmente, dependiendo del paradigma, se desarrollan estructuras funciones y métodos para integrar el dominio a un lenguaje de propósito general.

Al otro extremo, los DSL's externos, se implementan como cualquier otro lenguaje con la salvedad de que desde el principio se desarrollan e integran las representaciones y conceptos del dominio, estos DSL's además suelen tener su propio compilador o interprete.

Como se obeservara más a detalle en el capítulo de "Desarrollo", la implementación de in DSL interno es más sencilla de lo que parece. Los métodos y/o patrones de diseño más comunes para generar un DSL interno son: el encadenamiento de métodos (Riti, 2018, pág. 46), fábricas de objetos, proxis y el singleton, pero la creatividad y los paradigmas de los lenguajes en convergencia con el dominio marcaran la pauta.

Se recomienda que el proceso de creación de un DSL interno comience por revisar la semántica, seguida de la sintaxis y, finalmente, la parte léxica. Es importante destacar que este orden difiere del enfoque tradicional utilizado en el análisis de un lenguaje, intérprete o

compilador, pero existe una razón válida para ello. La inclusión de un DSL interno en un framework generalmente tiene como objetivo agilizar flujos de trabajo, automatizar tareas o satisfacer necesidades específicas del dominio. Sin importar la razón, estas situaciones ya poseen una semántica propia gracias al conocimiento del dominio.

La adaptación del código y API's con respecto al DSL va encaminada justo a eso, a apatarse al dominio y no al contrario.

IV DESARROLLO

4.1 Análisis

Ya se ha visto la pertinencia de Arduino y la placa ESP32, también se conoce por experiencia previa que el IDE Arduino y la plataforma de PlatformIO puede usar el lenguaje de programación C++ y que por ende se puede usar el paradigma Orientado a Objetos (PlatformIO, 2022).

También es de conocimiento público que Expressif, la empresa que desarrolla los ESP32, usa FreeRTOS en algunos de sus frameworks como SDK base (Espressif, 2023), (Espressif, 2023), (FreeRTOS, 2023) y que, a su vez, entre todas las plataformas que soporta FreeRTOS incluida la de Arduino (Arduino, 2023).

Esto último genera la posibilidad de usar una vasta cantidad de ejemplos y librerías en combinación con los mecanismos de concurrencia y paralelismo (en hardware y software), que ofrece la placa ESP32 con FreeRTOS en Arduino.

A partir de lo antes escrito y de la ya mencionada necesidad de elaborar un framework IoT con soporte a eventos que incluya los trabajos previos para combinarlos con los nuevos soportes de concurrencia y paralelismo se asume que se tienen los elementos necesarios para realizar tal tarea.

4.2 Metodología de Desarrollo

En esta etapa, y como ya se ha mencionado antes, se plantea una metodología iterativa. Se parte de un prototipo funcional "base" al que se le aplican diferentes patrones y paradigmas de programación para obtener el conjunto de estructuras y librerías que conformen el framework. Este proceso es recurrente y termina hasta alcanzar los requerimientos y/o métricas establecidas.

Como prototipo principal, el cual fungió de base para el desarrollo del framework se toman varios ejemplos, uno de los más destacados es de archivo ejemplo *"***Blink_AnalogRead.ino"** que viene con la instalación de la plataforma FreeRTOS en el IDE de Arduino y que se anexan en este documento [Anexo 4].

A partir de ello se inicia con el desarrollo de los temas y subtemas con las iteraciones presentadas en la tabla I.1 dedicada a cada implementación.

4.3 Tema 1, Programación Orientada a Objetos

Para desarrollar los subtemas de Programación Orientada a Objetos listados en la tabla de Iteraciones se emplearon las iteraciones 1, 2 y 3 correspondientes a: **Integración de Clases, Herencia, Polimorfismo y Lambdas**.

4.3.1 Iteración 1, Integración de Clases

El objetivo de la primera iteración es proporcionar al framework la capacidad de emplear **POO** básica con la declaración de clases y el manejo de objetos.

Esta iteración desarrolla las siguientes actividades:

- Validar el empleo de clases.
- Desarrollo de clases de envoltura para las tareas principales de **FreeRTOS**.
- Integrar el uso de directivas del preprocesador que se utilizan comúnmente en la gestión de inclusiones de librerías y en la prevención de inclusiones múltiples en programas de C y C++.

Validar el Uso de POO

Una clase es un mecanismo para crear tipos de datos definidos por el usuario y su uso es necesario para desarrollar las diferentes librerías que componen la propuesta que aquí se presenta de un "Framework IoT" (IBM, 2023), (CPP Reference, 2023).

El uso de clases y objetos, o dicho en forma más precisa, el empleo de las clases a partir de las cuales se crean los objetos, es en esencia el uso de componentes reutilizables de software (Deitel, 2012).
Los enunciados anteriores describen dos partes fundamentales para cualquier Framework, Tipos de Datos Definidos por el Usuario y Componentes Reutilizables. Para incluir el uso de POO, y por ende

de clases, se validó el uso de las mismas como se muestra en la tabla IV.1, para ello se adaptó el código a la estructura de programación de Arduino el código expuesto en (CPP Reference, 2023), esto con el fin de tener acceso al compilador de C++ 11 soportado por la misma plataforma de Arduino y para validar el uso de POO y de clases.

Tabla IV.1.- Validación del uso de POO y de clases

```
#include <Arduino.h>

class Foo;  // forward declaration of a class

class Bar { // definition of a class
 public:
   Bar(int i) : m_i(i) {}
   int getM_i()
   {
     return m_i;
   }
 private:
   int m_i;
};

template <class T> // template argument
void qux() {
   T t;
}

enum class Pub {   // scoped enum, since C++11
   b, d, p, q
};

Bar Bar1(1);

class Bar Bar2(2); // elaborated type

void setup() {

    Serial.begin(115200);
}

void loop() {

  Serial.println(Bar1.getM_i());
  Serial.println(Bar2.getM_i());

 while(1);

}
```

La validación se aprueba a partir de la compilación y ejecución del código anterior, así mismo en (CPP Reference, 2023), se muestra diferentes formas de declarar una "Clase" en C++, en este punto las diferentes formas de declarar clases son importantes porque impactan elementos relevantes del framework como: la incorporación de patrones de diseño y el uso de la sobrecarga de operadores.

***Nota**: Las clases enumeradas (**enum class**), están soportadas desde C++11.*

Clases Envoltura para Tareas de FreeRTOS

Esta actividad pretende generar las clases correspondientes al manejo de las tareas de FreeRTOS, es decir, la intención es desarrollar clases para que las tareas de FreeRTOS puedan ser manipulados como objetos.

Para generar las clases que envuelvan las tareas de FreeRTOS, hace falta ver el contexto de ejecución de cada tarea. La sentencia "xTaskCreate" (FreeRTOS, 2023)**,** tabla IV.2, da una buena introducción, esta función recibe "6" parámetros que describen su contexto de ejecución.

Tabla IV.2.- Cabecera de la función **xTaskCreate**.

```
BaseType_t  xTaskCreate(  TaskFunction_t pvTaskCode,
                          const char * const pcName,
                          configSTACK_DEPTH_TYPE
usStackDepth,
                          void *pvParameters,
                          UBaseType_t uxPriority,
                          TaskHandle_t *pxCreatedTask );
```

- El primer parámetro, tal vez el más importante, es la función que va a desempeñar, es decir, el código de ejecución de la tarea misma.
- El siguiente parámetro es el nombre, este parámetro es informativo y útil, ya que da la posibilidad de generar un seguimiento más personal.
- El tercero es la memoria asignada a la tarea.

- El cuarto recuerda que se esta pasando como parámetro una función y se refiere a los parámetros de entrada de la tarea.
- El quinto parámetro es la prioridad de la tarea; ya que las tareas compiten por un tiempo de procesador es necesario asignar prioridades para un buen manejo de este recurso.
- El último parámetro es el "TaskHandle_t", y es la referencia que usa FreeRTOS para la tarea creada.

Aunque cada parámetro es importante solo los parámetros de entrada y el "TaskHandle_t" pueden pasarse como nulos. A continuación, se presenta y describe la clase propuesta como envoltura para la función "xTaskCreate" de FreeRTOS:

Tabla IV.3.- Clase envoltura.

```
1   #include <Arduino.h>
2
3
4   class xTask
5   {
6      public:
7           int16_t         memSize;
8           int8_t          priority;
9           void            (*task)(void *pvParameter);
10          char*           name;
11          TaskHandle_t    taskHandle;
12          void*           params;
13
14          xTask( char* n, int16_t mS, int8_t p ) {
15
16              name        = n;
17              memSize     = mS;
18              priority    = p;
19              taskHandle  = NULL;
20              params      = NULL;
21          }
22
23
24          void run()
25          {
26              xTaskCreate(
27                              task
28                          ,   name
29                          ,   memSize
30                          ,   params
```

```
31                          ,  priority
32                          ,  &taskHandle
33                      );
34          }
35
36
37          xTask* setT( void (*tsk)(void *pvParameter) )
38          {
39              task = tsk;
40              return this;
41          }
42  };
43
```

La clase que se muestra en la tabla IV.3 se divide en 3 secciones: declaración de atributos, declaración de un constructor parametrizado y la declaración de métodos.

La declaración de atributos, básicamente incorpora cada uno de los parámetros de la función "xTaskCreate", es esencial que la clase "xTask" obtenga las mismas variables ya que podrían ser utilizadas en algún momento para la ejecución de la tarea.

La sección del constructor asegura que la clase solo pueda ser instanciada con el paso de parámetros correspondiente, es decir, que al momento de la creación del objeto se cuente con los elementos básicos para su posterior ejecución.
La última sección declara el método "run", como su nombre en inglés lo indica será el encargado de correr o ejecutar la tarea.

Ejemplo de Uso

El uso de la clase "xTask" (visto en la tabla IV.3), es similar a la función "xTaskCreate" (expuesto en la tabla IV.2), incluso se podría decir que solo se han intercambiado algunas secciones de código, pero el trasfondo es que ahora se conceptualiza la tarea como un objeto.

En la tabla IV.4 se pode ver la declaración del objeto tarea en línea 7, la asignación del código a ejecutar está escrita en la línea 10, y el llamado a "run" en la línea 11.

Tabla IV.4.- Declaración del objeto tarea.

```
1  #include <Arduino.h>
2  #include "xTask.cpp"
3
4  int LED_BUILTIN = 2;
5  void TaskBlink(void *pvParameters);
6
7  xTask x_task("Blink", 512, 1);
8
9  void setup() {
10  x_task.setT(TaskBlink);
11  x_task.run();
12 }
13
14 void loop() {
15 }
16
17 // This is a task.
18 void TaskBlink(void *pvParameters)
19 {
20  pinMode(LED_BUILTIN, OUTPUT);
21  for (;;)
22  {
23    // turn the LED on (HIGH is the voltage level)
24    digitalWrite(LED_BUILTIN, HIGH);
25    // wait for one second
26    vTaskDelay( 3000 / portTICK_PERIOD_MS );
27    // turn the LED off by making the voltage LOW
28    digitalWrite(LED_BUILTIN, LOW);
29    // wait for one second
30    vTaskDelay( 3000 / portTICK_PERIOD_MS );
31  }
32 }
```

La línea 10 merece un desglose más detallado: la sentencia en esa posición recibe el puntero de la función que se ejecutará; esta función a su vez está dividida en su prototipo en la línea 5 y en su implementación en la línea 18.

El paso de puntero a función como parámetro es una característica de C++ que ayudará al framework a ser más dinámico.

Uso de las directivas del preprocesador

El preprocesador ofrece un mecanismo por el que una porción de código de un programa se puede ocultar o considerar dependiendo del valor de alguno de los símbolos definidos con la directiva **#define**.

En este caso se usa la directiva **#ifndef** para cargar una única vez la clase "xTask", la macro que se presenta a continuación asegura que solo una vez de se incluya la clase "xTask" (ver tabla IV.5).

Tabla IV.5.- La directiva "#ifndef".

```
#include <Arduino.h>

#ifndef xT_H
#define xT_H
    class xTask ...
#endif
```

4.3.2 Iteración 2, Herencia y Polimorfismo

La iteración "2" emplea "Herencia" y "Polimorfismo**"** para especializar objetos que tienen una base en común. A partir del uso de "Herencia**"** y "Polimorfismo**"** se pueden generar una clase base para las tareas de FreeRTOS con sus correspondientes clases derivadas de especialidad.

El uso de varios métodos con el mismo nombre, pero con diferente paso de parámetros proporciona semántica al Framework IoT, un elemento importante en la adopción de cualquier lenguaje librería o framework.

Esta iteración desarrolla las siguientes actividades:

- Validar el empleo de herencia.
- Desarrollo de clases base y clases derivadas para las tareas previas de FreeRTOS.
- Desarrollo de métodos polimórficos, en un primer acercamiento a la semántica final del Framework IoT.

A continuación, se presenta y se describe la clase que implementa la parte de “Herencia”.

Tabla IV.6.- Clases base y derivadas.

```
1   #include <Arduino.h>
2   #ifndef xT_H
3   #define xT_H
4
5   class xTaskBase //Clase base o raiz
6   {
7      public:
8          int16_t         memSize;
9          int8_t          priority;
10         void            (*task)(void *pvParameter);
11         char*           name;
12         TaskHandle_t    taskHandle  = NULL;
13         void*           params      = NULL;
14
15         xTaskBase( char* n, int16_t mS, int8_t p ):
16                         memSize(mS), priority(p), name(n)
17         {  }
18
19         virtual void run()
20         {  }
21
22         xTaskBase* setT( void (*tsk)(void *pvParameter) )
23         {
24             task = tsk;
25             return this;
26         }
27  };
28
29
30  class xTask : public xTaskBase //Clase derivada de xTaskBase
31  {
32     public:
33
34         xTask( char* n, int16_t mS, int8_t p ):
35                                      xTaskBase(n, mS, p)
36         {  }
37
```

```
38          virtual void run()
39          {
40              xTaskCreate(
41                              task
42                          ,   name
43                          ,   memSize
44                          ,   params
45                          ,   priority
46                          ,   &taskHandle
47                      );
48          }
49  };
50
51  class dxTask : public xTask //Clase derivada de xTaskBase y
52  {                           // de dxTask
53     public:
54       int8_t xnCore;
55
56       dxTask(char* n, int16_t mS, int8_t p, int8_t xnC):
57                               xTask(n, mS, p), xnCore(xnC)
58          {  }
59
60          void run()
61          {
62              xTaskCreatePinnedToCore(
63                              task
64                          ,   name
65                          ,   memSize
66                          ,   params
67                          ,   priority
68                          ,   &taskHandle
69                          ,   xnCore
70                      );
71          }
72  };
73  #endif
```

Como se puede percibir en la tabla IV.6 ahora hay 3 clases, la clase “xTaskBase” en la línea 5 que funge como clase “base”, la clase “xTask” en la línea 30 que es una clase derivada y la clase “dxTask” en la línea 51, que es una segunda derivación, pero por parte de “xTask”.

A raíz de esto la clase “xTask” sufre un par de cambios y se describe como primera clase especializada y funcional para ejecutar una tarea, la clase expone el comportamiento más básico de todos al solo contar con el paso de parámetros indispensable. La segunda clase derivada, la clase “dxTask”, incorpora el parámetro “xnC” que seguido de la sobreescritura

del método “run” con la sentencia “xTaskCreatePinnedToCore” designa el núcleo en donde se ejecutará la tarea. Con esto se deja abierta la posibilidad de expandir las clases que manejan la creación y ejecución de las tareas. El siguiente ejemplo, tabla IV.7, muestra cómo son las declaraciones e invocaciones de las clases antes mencionadas.

Tabla IV.7.- Ejemplo de las declaraciones e invocaciones de clases.

```
1  #include <Arduino.h>
2  #include "xTask.cpp"
3
4  int8_t LED_BUILTIN = 2;
5  int8_t Pin32       = 32;
6
7  void TaskBlink      (void *pvParameters);
8  void TaskBlinkPin32 (void *pvParameters);
9
10 dxTask *blink = new dxTask("Blink", 1024, 1, 1);
11
12 void setup() {
13
14  blink->setT(TaskBlink);
15  blink->run();
16
17  xTask *blink32 = new xTask("Blink32", 1024, 1);
18  blink32->setT(TaskBlinkPin32);
19  blink32->run();
20 }
21
22 void loop() { }
23
24 void TaskBlink(void *pvParameters)
25 {
26  pinMode(LED_BUILTIN, OUTPUT);
27
28  for (;;)
29  {
30    digitalWrite(LED_BUILTIN, HIGH);
31    vTaskDelay( 1000 / portTICK_PERIOD_MS );
32    digitalWrite(LED_BUILTIN, LOW);
33    vTaskDelay( 1000 / portTICK_PERIOD_MS );
34  }
35 }
36
37 void TaskBlinkPin32(void *pvParameters)
38 {
39  pinMode(Pin32, OUTPUT);
40
41  for (;;)
```

```
42   {
43     digitalWrite(Pin32, HIGH);
44     vTaskDelay( 1000 / portTICK_PERIOD_MS );
45     digitalWrite(Pin32, LOW);
46     vTaskDelay( 1000 / portTICK_PERIOD_MS );
47   }
48 }
```

El código anterior representa el uso básico de las funciones y clases que integran el "Framework IoT" en Arduino después de la implementación de la “herencia”; se observa, también, que es suficiente con incluir el archivo "xTask.cpp" (línea 2 de la tabla IV.7) para acceder a las definiciones de las clases “xTask” y “dxTask”.

Despues, en las líneas 7 y 8, tabla IV.7, se presenta el prototipo de las funciones que llevan a cabo la lógica de cada tarea.

En la línea 10, tabla IV.7, se introduce la instancia “blink” derivada de "dxTask". La generación de esta instancia implica el uso de un constructor diseñado para recibir no solo los parámetros de la clase base ("xTask"), sino también el número que indica en qué núcleo se desea que la instancia ejecute su función.

Usualmente, el método "setup()" se emplea para iniciar o cargar configuraciones. En este caso, el "setup()" configura la tarea "blink" al invocar el método "setT" y pasarle su respectiva tarea (“TaskBlink”) para, posteriormente, iniciarla con el método "run", línea 14 y 15 respectivamente.

La ejecución anterior además de confirmar la operatividad de la clase "dxTask", también presenta la instanciación, asignación y activación de la tarea "blink32". Esta última operación, aunque pueda llegar a parecer redundante en este ejemplo, es importante porque destaca la integración funcional y jerárquica de las clases "xTask" y "dxTask" a través del concepto de "Herencia".

4.3.3 Iteración 3, Lambdas

La iteración "**3**" aborda el uso de Lambdas que dan un modo de uso al "Framework IoT".

Las "Lambdas" proporcionan una forma de terminar la implementación, es decir, agrega los mecanismos para asignar funciones anónimas con código escrito por el usuario final (programador) para así terminar su propia implementación.

Esta iteración desarrolla las siguientes actividades:

- Validar el uso de Lambdas.
- Declaración de tipos mediante la directiva "using".

El archivo "xTask.cpp", tabla IV.8, implementa algunos cambios, se agrega la definición de tipo tTask en la línea 6 y cambia el nivel de acceso a "protected" las variables de: "memSize", "priority", "name", "taskHandle" y "params":

Tabla IV.8.- Archivo "xTask.cpp".

```
1   #include <Arduino.h>
2
3   #ifndef xT_H
4   #define xT_H
5
6   using tTask   = void (*)(void *pvParameter);
7
8   class xTaskBase
9   {   protected:
10          int16_t         memSize;
11          int8_t          priority;
12          char*           name;
13          TaskHandle_t    taskHandle  = NULL;
14          void*           params      = NULL;
15
16      public:
17          tTask task;
18
19          xTaskBase( char* n, int16_t mS, int8_t p ):
20                          memSize(mS), priority(p), name(n)
```

```
21          {  }
22
23          virtual void run()
24          {  }
25
26          xTaskBase* setT( void (*t)(void *pvParameter) )
27          {
28              task = t;
29              return this;
30          }
32  };
33
34  class xTask : public xTaskBase
35  {   . . .
36  };
37
38  class dxTask : public xTask
39  {  . . .
40  };
41
42  #endif
```

Aunque el cambio en el archivo anterior no es amplio, el cambio en el uso de framework si lo es.

Tabla IV.9.- Declaración de la variable “myTask”.

```
1   #include <Arduino.h>
2   #include "xTask.cpp"
3
4   int8_t LED_BUILTIN = 2;
5   int8_t Pin32       = 32;
6   int8_t Pin33       = 33;
7
8   void TaskBlink      (void *pvParameters);
9   void TaskBlinkPin32 (void *pvParameters);
10
11  xTask *blink   = new dxTask("Blink"  , 1024, 1, 1);
12  xTask *blink32 = new xTask( "Blink32", 1024, 1);
13
14  tTask myTask = [] (void *pvParameter) -> void {
15
16     pinMode(Pin33, OUTPUT);
17
18     for (;;)
19     {
20       digitalWrite(Pin33, HIGH);
```

```
21      vTaskDelay( 500 / portTICK_PERIOD_MS );
22      digitalWrite(Pin33, LOW);
23      vTaskDelay( 500 / portTICK_PERIOD_MS );
24    }
25  };
26
27  void setup() {
28    blink->task = [] (void *pvParameter) -> void {
29
30     pinMode(LED_BUILTIN, OUTPUT);
32
33     for (;;)
34     {
35       digitalWrite(LED_BUILTIN, HIGH);
36       vTaskDelay( 500 / portTICK_PERIOD_MS );
37       digitalWrite(LED_BUILTIN, LOW);
38       vTaskDelay( 500 / portTICK_PERIOD_MS );
39     }
40  };
41  blink->run();
42
43  blink32->setT(
44         [] (void *pvParameter) -> void
45         {
46           pinMode(Pin32, OUTPUT);
47
48           for (;;)
49           {
50             digitalWrite(Pin32, HIGH);
51             vTaskDelay( 1000 / portTICK_PERIOD_MS );
52             digitalWrite(Pin32, LOW);
53             vTaskDelay( 1000 / portTICK_PERIOD_MS );
54           }
55         }
56        )
57        ->run();
58
59  xTask( "Blink33", 1024, 1).setT(myTask)->run();
60  }
61
62  void loop()
63  {
64  }
```

En la línea 14 de la tabla IV.9, se declara la variable “myTask” de tipo “tTask” y se le asigna una función “lambda”, poco después en la línea 59 se instancia un objeto anónimo y desde

su constructor se invoca el método “setT” para pasarle la variable “myTask” y lanzar su método “run”, todo esto en una sola línea.

Otro modo de uso es el expuesto en la línea 28, en donde a la variable “task” del objeto “blink” se le asigna una lambda para su posterior ejecución.

La declaración del tipo “tTask” crea la posibilidad de declarar en otros archivos las tareas para invocarlas desde el archivo principal.

4.4 Tema 2, EL Modelo de Actores

En la programación orientada a objetos es habitual y funcional representar objetos mediante métodos o acciones de entrada y salida, los cuales establecen un protocolo claro y significativo de interacción. Este enfoque es el que se utiliza en el modelo de actores (Agha G. , 1990).

Los actores, al igual que las tareas vistas en esta tesis, son componentes autónomos, interactivos e independientes de un sistema informático que se comunican mediante el paso de mensajes asincrónicos (Agha G. , 2004).

Las iteraciones 4 y 5 implementan los mecanismos necesarios para generar una aproximación al modelo de actores en este Framework.

4.4.1 Iteración 4, Colas de Mensajes

Las “colas” o “queues” de mensajes son uno de los mecanismos más usados para la comunicación entre tareas de FreeRTOS, pero también se describen en el Modelo de Actores como la estructura base para la comunicación (Agha G. , 1990, págs. 36-37).

El uso de las colas bajo este modelo brindará un sistema de comunicación entre objetos y tareas.
Esta iteración desarrolla las siguientes actividades:

- Adición y/o reescritura como evolución de clases obtenidas en la iteración 2 que le proporcionen el uso de colas para la comunicación entre tareas por medio de paso de mensajes.
- Uso de plantillas o elementos genéricos como tipos de intercambio entre colas de mensajes.

Como se muestra a continuación, las actividades se completan al agregar la clase “xQueue” y al modificar la clase “xTask”.

Tabla IV.10.- Clase xQueue.

```
1   template <class T>
2   class xQueue {
3      public:
4          xQueueHandle xQ;
5
6      xQueue()
7      {
8          int8_t msgSize = sizeof(T);
9          xQ = xQueueCreate(10, msgSize);
10     }
11
12     xQueue(int8_t size)
13     {
14         int msgSize = sizeof(T);
15         xQ = xQueueCreate(size, msgSize);
16     }
17
18     T receive()
19     {
20       . . .
21     }
22  };
```

La clase “xQueue”, tabla IV.10, representa la estructura que usarán las tareas como medio de comunicación. En la práctica esta clase envuelve el tipo de dato “xQueueHandle”, línea 4, que es usado por FreeRTOS como uno de sus medios de comunicación.

Los elementos restantes de la clase como los constructores y el método “receive” son las funcionalidades con las que otros objetos pueden interactuar con las instancias de esta clase.

Tabla IV.11.- Reescritura de la clase xTask.

```
1   template <class T>
2   class xTask : public xbTask
3   {
4      private:
5          xQueue<T>   queue;
6
7          .
```

```
8           .
9           .
10
11           void send (const T element)
12           {
13               if ( xQueueSendToBack(queue.xQ, &element,
14                        2000/portTICK_RATE_MS) != pdTRUE )
15               {
16                   Serial.println("error");
17               }
18           }
19  };
```

La reescritura de la clase “xTask”, en la tabla IV.11, muestra la adición de una instancia de la clase xQueue (línea 5) y su posterior uso con el método “send” (línea 13). Además, se puede ver los códigos “template <class T>” y “<T>” (línea 1 y 5 respectivamente) que define el tipo de dato “*genérico*” con el que va a trabajar la instancia queue.

Los genéricos son tipos parametrizados, es decir, como muchas estructuras de datos y algoritmos tienen el mismo aspecto y comportamiento independiente del tipo que operan, es posible pasar el tipo de datos con el cual van a operar. Con la palabra reservada “template”, plantilla en español, se agrega tal posibilidad.

Una plantilla es una construcción que genera un tipo o función normal en tiempo de compilación en función del o los tipos de datos enviados en forma de argumentos (Microsoft, 2023) (Microsoft, 2023). Gracias a las plantillas las instancias de la clase xQueue dependen del tipo de dato que se especifica al momento de su creación.

El agregar una instancia de la clase xQueue tiene buenas repercusiones, la más importante es que da la posibilidad de tratar a las instancias de “xTask” como “Actores”. Esto último se debe a las nuevas características que la instancia xQueue le proporciona a la clase “xTask”.

Las características expuestas por Agha en (Agha G. , 1990) y (Agha G. , 2004) para la definición de Actores describen que:

- Puede procesar sólo aquellas tareas cuyo destino corresponda a su dirección.
- Cuando se acepta una comunicación, puede crear nuevos actores o tareas.
- Debe calcular un comportamiento de reemplazo o el término de su ejecución

Por ejemplo, en las líneas 1 y 13 de la tabla IV.12, se observa la posibilidad de generar un comportamiento de reemplazo al asignar y reasigna la función que desempeñara la tarea misma.

Tabla IV.12.- Ejemplo de un comportamiento de reemplazo.

```
1   worker->task = [] (void *pvParams) -> void
2    {
3      while(true)
4      {
5        objReceive = worker->queue.receive();
6        // Se procesa el objeto objReceive
7      }
8    };
9    worker->run();
10     .
11     .
12     .
13  worker2->task = [] (void *pvParams) -> void
14   {
15     while(true)
16     {
17       if( . . . ){
18         // Se crea una nueva tarea
19         xTask( "Blink33", 1024, 1).setT(myTask)->run();
20       }
21     }
22   };
23   worker->run();
24
```

Por último, se observa que la invocación del método “recive”, línea 5, permite a la clase “xTask”, procesar los objetos recibidos; en este punto cabe resaltar que el objeto devuelto por la clase “xQueue” con el método “recive” es del mismo tipo usado para instanciar la clase “xTask”. El diseño general del Modelo de Actores permite seguir una línea segura para que los sistemas concurrentes se comporten de manera predecible (Agha G. , 1990, págs. 4-5)

4.4.2 Iteración 5, Sobrecarga de Operadores

Se podría decir que la sobrecarga de operadores es una licencia semántica en C++ que mejora la lectura del código. Más aún, ya que los operadores tienen un significado propio también proporciona un modo de uso explícito.

La ventaja que proporciona la sobrecarga de operadores es que al conocer como un objeto interactúa con él, es fácilmente deducible el comportamiento que expondrá con los demás.

Esta iteración desarrolla las siguientes actividades:

- Listar los símbolos, clases y métodos sugeridos para el uso de sobrecarga de operadores.
- Sobreescritura de los métodos listados en la actividad por operadores sobrecargados.

A continuación, y como ejemplo, se presenta la lista de símbolos (tabla IV.13) con sus respectivos métodos sugeridos para ser sustituidos por la sobrecarga de un operador.

Tabla IV.13.- Lista de símbolos para la sobrecarga de operadores.

Operador	Clase y Método
>>	xTask<T>, **to_xCore** (int8_t c);
>>	xTask<T>, **send** (const T element);

El siguiente código presenta la sobrecarga de los operadores antes listados.

Tabla IV.14.- Código de la sobrecarga de los operadores.

```
1  template <typename T>
2  void operator >> (xTask<T>* task, x_Core xCore)
3  {
4     task->to_xCore(xCore.number);
5  };
6
7
```

```
8   template <typename T>
9   void operator >> (const T element, xTask<T>* task)
10  {
11     task->send(element);
12  };
```

La sobrecarga del operador **“>>”,** (tabla IV.14, línea 9) que agrega un poco de azúcar sintáctica establece un flujo de trabajo. Sin embargo, la sobrecarga es solo una abreviación del uso del método “send”.

4.5 Tema 3, Protocolos IoT

El uso de protocolos de comunicación sobre internet es parte esencial de la tecnología IoT siendo los protocolos **HTTP** y **MQTT** los más comunes (AWS, 2023) (Microsoft, 2023).

El protocolo **HTTP** es el usado por los servidores y clientes web para la publicación y consumo de páginas. Su uso en IoT proporciona el mismo mecanismo de comunicación en donde un dispositivo puede solicitar o recibir datos para obtener o proporcionar una respuesta en internet.

Por otra parte, como ya se vio en el marco teórico, el **MQTT** se usa para conexiones a un servidor central de mensajería denominado bróker.

El objetivo de las siguientes 2 iteraciones es incluir el uso de estos protocolos en el set de objetos y mecanismos del framework.

4.5.1 Iteración 6, El protocolo HTTP

El protocolo **HTTP** en el framework expone los mecanismos de comunicación a elementos fuera del microcontrolador como lo son Servidores o Clientes Web, este esquema extiende el borde del sistema.

Ya sea fungiendo como cliente o servidor su uso describe que la frontera del sistema existe también fuera del microcontrolador.

Esta iteración desarrolla las siguientes actividades:

- Validar el uso del protocolo **HTTP** en su vertiente de Server con un ejemplo que interactúe con las clases y elementos generados en las iteraciones pasadas.

- Agregar los métodos “GET” y “POST” como una adecuación semántica para su coincidencia con las librerías y framework que se usan comúnmente para desarrollar servicios de internet fuera de un contexto embebido.

A continuación, se presenta el código que valida el uso del protocolo **HTTP** y su interacción con el trabajo previo. El código siguiente es ejecutable y muestra cómo las tareas concurrentes de FreeRTOS pueden coexistir con el “WebServer”.

Tabla IV.15.- Uso del protocolo HTTP.

```
1  #include <WebServer.h>
2
3  void setup() {
4
5   server.on("/lambda", [] () -> void {
6         server.send(200, "text/html", "...");
7   });
8
9   xsTask(512,  1, "blick", 1).setT(
10                        [&] (void *pvParameter) -> void
11  {
12    for (;;)
13    {
14      digitalWrite(led, LOW);
15      vTaskDelay( 1000 / portTICK_PERIOD_MS );
16
17      digitalWrite(led, HIGH);
18      vTaskDelay( 1000 / portTICK_PERIOD_MS );
19    }
20
21  })->run();
22 }
23
24 void loop() {
25
26   server.handleClient();
27 }
```

Otra adecuación importante es que si el “WebServer” se configura con una función anónima o Lambda (tabla IV.15, línea 5), puede ver como local las variables tipo “xeTask” para usar la ya mencionada sobrecarga de operadores. El siguiente código muestra tal ejemplo.

La segunda actividad se finaliza al cambiar el nombre del método “on” (tabla IV.16, línea 3) por “get” (tabla IV.17, línea 3).

Tabla IV.16.- Código original con el método “on”.

```
1  Pin pin2;
2
3   server.on("/xTask", [] () -> void {
4         .
5         .
6         .
7         pin2 >> my_xeTask;
8         server.send(200, "text/html", "...");
9   });
```

Tabla IV.17.- Cambio de nombre del método “get” a “on”.

```
1  Pin pin2;
2
3   get("/xTask", [] () -> void {
4         .
5         .
6         .
7         pin2 >> my_xeTask;
8         server.send(200, "text/html", "...");
9   });
```

Como se mencionó antes, el cambio de nombre obedece al estándar de las API’s REST implementadas por otros lenguajes como Java, C# y Python (Spring Guides, 2023) (Microsoft, 2023) (FastAPI, 2023), para que el desarrollo embebido se apegue lo más posible a ellos.

4.5.2 Iteración 7, El Protocolo MQTT

El protocolo **MQTT** incorpora atención a eventos específicos desde la red; en el marco teórico, se ve el concepto de tópico como metadato para la discriminación de eventos.

Cualquier aplicación, embebida o no, tiene la capacidad de publicar o subscribirse a los eventos de un sistema más grande y las aplicaciones que desarrollan o implementan este protocolo suelen ser módulos intercambiables.

Esta iteración desarrolla las siguientes actividades:

- Validar el uso del protocolo **MQTT** con un ejemplo compilado.
- Generar un ejemplo de uso con las clases y librerías generadas por las iteraciones anteriores.

De la misma forma que la iteración anterior, se presenta el código que valida el uso del protocolo **MQTT** y su interacción con las demás tareas.

Tabla IV.18.- Envío de mensajes a tareas desde un callback.

```
1  #include <PubSubClient.h>
2
3  WiFiClient espClient;
4  PubSubClient client(espClient);
5
6  xeTask my_xeTask( . . . );
7  Pin pin2;
8
9  void callback(char* topic, byte* message, unsigned int
10 length)
11 {
12  Serial.print("Message arrived on topic: ");
13
14  pin2 >> my_xeTask;
15 }
16
17 void setup() {
```

```
18   .
19   .
20   .
21   my_xeTask.setT( [&] (void *pvParameter) -> void
22   {
23     for (;;)
24     {
25         . . .
26     }
27   })->run();
28
29 }
30
31 void loop() {
32
33   if (!client.connected())
34   {
35    reconnect();
36   }
37
38   client.loop();
39 }
```

El código de la tabla IV.18, es un ejemplo práctico de un cliente MQTT que ejecuta de forma concurrente una tarea, además se observa que la función callback (función que es llamada cada vez que llega un mensaje del bróker MQTT), puede también llamar al operador “>>” (línea 33) para enviar un tipo de dato predeterminado a la tarea “my_xeTask”.

Estos últimos códigos y comentarios validan las dos actividades de este tema.

4.6 Tema 4 Protocolos IIoT

Como se menciona en el marco teórico la utilización de tecnologías de IoT dentro de los sistemas de producción es un factor clave dentro de la transformación hacia la “Industria 4.0” (M. Wollschlaeger, 2017).

Aunque los sistemas industriales han avanzado de forma independiente de las “TIC’s**”**, debido a que abordan necesidades explícitas como: procesamiento en tiempo real y alta disponibilidad, cada vez es más frecuente y necesaria la interacción entre estas dos tecnologías.

Por lo general a cualquier equipo que aprovecha la conexión a Internet con fines industriales se los considera “IIoT”. A partir de esto y atendiendo la reciente integración de las “TIC’s” y los modelos de producción industrial, en este framework se agregan las pruebas necesarias para validar que pueda interactuar con otras librerías y framework que den soporte a protocolos de carácter industrial como Modbus y Profinet.

4.6.1 Iteración 8, El protocolo Modbus

Modbus/TCP es un protocolo de comunicación en red para equipos industriales como: Controladores Lógicos Programables (PLCs), computadoras y drivers de motores entre otros (NATIONAL INSTRUMENTS CORP, 2022).

Es escalable y fácil de administrar, ya que para un dispositivo que tiene un propósito simple solo se necesita implementar uno o dos tipos de mensaje sin ningún tipo de herramienta especial (Peinado Royero, 2005, pág. 33).

Ya sea fungiendo como maestro o esclavo el protocolo Modbus amplía el uso del framework a aplicaciones y redes industriales.

Esta iteración desarrolla las siguientes actividades:

- Validar cualquiera de sus vertientes del protocolo maestro/esclavo con el framework aquí propuesto.
- Probar el uso de mensajes entre los objetos tareas originados por algún cambio de los registros “Modbus”.
- Validar tal implementación con algún dispositivo o herramienta de carácter industrial.

A continuación, se desarrollan dichas actividades.

La herramienta de carácter industrial seleccionada es “CODESYS” ya que tiene un amplio soporte, su IDE es de uso libre se apega al estándar internacional “IEC 61131” (CEI, 2003) (Codesys, 2023), (Codesys, 2020) que proporciona una colección completa de estándares sobre controladores programables y sus periféricos asociados (ver figura IV.1).

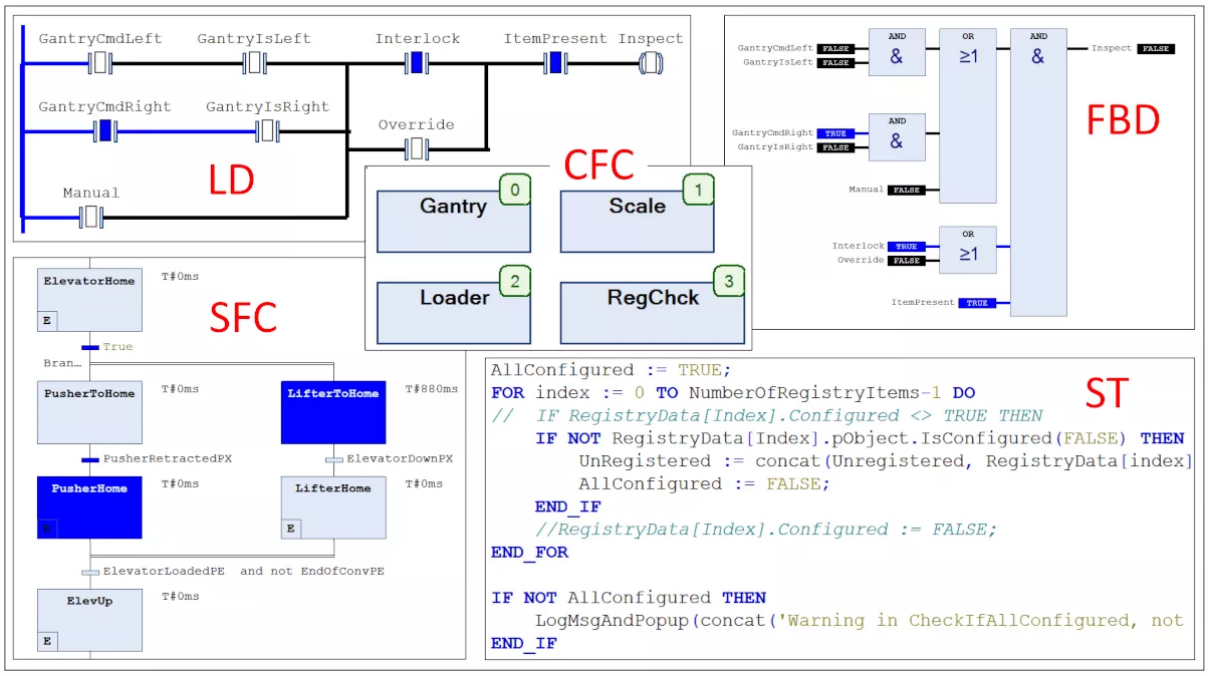


Figura IV.1.- Lenguajes soportados por el estándar IEC 61131.

Otro punto importante es el soporte a POO (Codesys, 2019); la "CODESYS Group" en su artículo "**LEVERAGE OBJECT-ORIENTED INDUSTRIAL PROGRAMMING**" menciona que:

*"Las plantas y los equipos se ensamblan a partir de objetos, por lo que la arquitectura de controles también debería serlo. Las nuevas herramientas ayudan a los programadores industriales a ofrecer la productividad de la programación orientada a objetos (**POO**) sin complejidad".*

Tabla IV.19.- Código Modbus para el envío de mensajes a tareas "xeTask".

```
#include <WiFi.h>
#include <Modbus.h>
#include <ModbusIP_ESP8266.h>

#include "xTask.cpp"

//ModbusIP object
ModbusIP mb;

xeTask<Led> xeTLed(1024,  1, "LED_32");

mb.addCoil(LED_COIL0);
.
.
.
Led led {
    .val = LED_COIL0
 };

void loop() {
  //Call once inside loop()
  mb.task();

  //Read LED_COIL0 register
  digitalWrite(LED1pin, mb.Coil(LED_COIL0));

  led.setVal(mb.Coil(LED_COIL0));
  //Send l to task xeTLed
  led >> xeTLed;
}
```

El código anterior, tabla IV.19, verifica que:

- El uso del protocolo Modbus con el framework no interfiere con las tareas de FreeRTOS envueltas en las clases “xsTask” y “xeTask”.
- Intercambio de información proveniente del protocolo Modbus hacia la tarea “xeTLed” (ver línea 53).

***Nota**: Hay que recordar que las tareas xsTask y xeTask en este contexto se están ejecutando de forma concurrente.*

A continuación, se muestra parte del código hecho en “CODESYS” para validar el uso del protocolo Modbus.

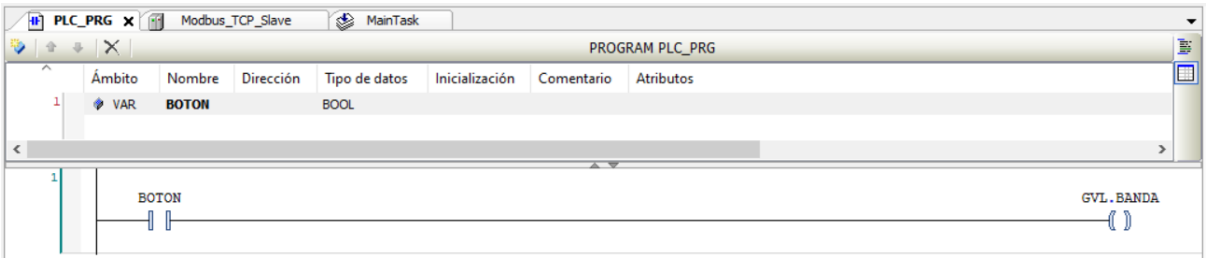


Figura IV.2.- Código escalera para la prueba del protocolo Modbus.

El código escalera de la figura IV.2, implementa la activación de la bobina Modbus etiquetada como “GVL_BANDA”; se puede observar que la activación de la variable “GVL_BANDA” a su vez depende del contacto normalmente abierto denominado BOTON por estar en el mismo escalón.

Así mismo, se tiene la conexión de la variable “GVL_BANDA” con la bobina Modbus “0”, se observa como un mapeo dado de alta en la pestaña “ModbusTCPSlave Asignación E/S” de “Modbus_TCP_Slave” con la función “Write Single Coil” (figura IV.3).

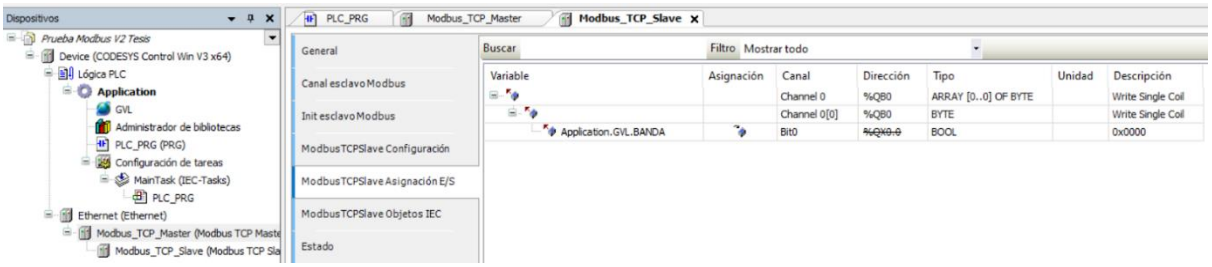


Figura IV.3.- Adición de la función “Write Single Coil”.

La ejecución completa, del código hecho en CODESYS y el código Modbus del framework, muestran que el estado propagado por CODESYS de la variable GVL_BANDA es censado y enviado a la variable “led” y a la tarea “xeTLed” (tabla IV.19, línea 51 y 53) respectivamente.

4.7 Tema 5, DSL

Integrar un “DSL” o Lenguajes Específicos de Dominio en el framework agrega a su vez una guía de uso semántica.

Con clases y objetos muy concretos dentro del framework su implementación se torna igual de “concreta”, aquí es donde el “DSL” es una guía. A estos lenguajes pequeños y limitados en frameworks se les conoce también como “DSL internos” y la claridad del enfoque que proporcionan hace que valga la pena su implementación (Fowler, Defining Domain-Specific Languages, 2010, pág. 27).

Aunque hay muchas formas de implementar un “DSL Interno” en un framework, el patrón de diseño “Encadenamiento de Métodos” es una de las primeras formas de lograrlo (Fowler, Defining Domain-Specific Languages, 2010, pág. 62).

4.7.1 Iteración 9, Encaminamiento de Métodos

Como se ve en el código de esta iteración, el patrón de “Encadenamiento de Métodos“ en combinación con interfaces integra el “DSL interno” antes mencionado.

La idea de tras el “Encadenamiento de Métodos” es que cada método altera el estado, pero devuelve un objeto para continuar la cadena (Fowler, Defining Domain-Specific Languages, 2010, pág. 62).

Esta iteración desarrolla las siguientes actividades:

- Lista los métodos que, por su nombre o valor de retorno, puedan generar una línea de código como idea semántica que indique un comportamiento definido o deseable.
- A partir de la lista anterior codificar el cambio en los métodos (valores de retorno y nombres) para permitir la invocación de métodos “*encadenados"* en pro de la semántica del uso framework.

- Evaluar el uso de interfaces que puedan enmascarar los valores de retorno para exponer solo ciertos comportamientos o métodos.

La siguiente lista (tabla IV.20) muestra los métodos que pueden generar una cadena de invocaciones para definir un comportamiento deseable acarreando un estado previo.

Tabla IV.20.- Lista de métodos que puedan generar una línea de métodos encadenados.

Nombre	Descripción
`xTask* setT( void (*tsk)(void *pvParameter) )`	Agregar una función a la tarea
`to_xCore (int8_t c)`	Reinicia la tarea en el núcleo que se pasa como parámetro.
`send (const T element)`	Envía un elemento procesable a la tarea en cuestión.

La tabla IV.21, muestra la sentencia “return this” ya codificada en los tres métodos; se usa para retornar el apuntador de su propia instancia y generar la oportunidad de seguir invocando otro método del mismo objeto.

Tabla IV.21.- Código actualizado para el encadenamiento de métodos.

Funciones que retornan la instancia “this”

```
xTask* setT( void (*tsk)(void *pvParameter) )
{
   task = tsk;
   return this;
}
```

```
xTask* to_xCore(int8_t c)
{
    deleteTask();
    xTaskCreatePinnedToCore (t, "n", mSize, prms, pty, &tH, c);
    return this;
}
```

```
xTask* send (const T element)
{
  if ( xQueueSendToBack(queue.xQ, &element,
                   2000/portTICK_RATE_MS) != pdTRUE )
  {
    Serial.println("error");
  }
  return this;
}
```

Se podría definir que el costo de implementación del patrón de métodos encadenados es la imposición de retornar, con el fin de seguir la cadena, la instancia “this”; en donde el valor de respuesta es la instancia misma.

Esta ultima adecuación, aunque sencilla, genera un proceso mas amigable al momento de crear o recrear una lista de métodos para ejecutar un sistema determinado.

4.8 Cierre

A continuación, se revisan algunos diagramas para representar y visualizar los diferentes aspectos de un sistema. Estos diagramas proporcionan información clave sobre la estructura, el comportamiento y las interacciones del desarrollo.

4.8.1 Modelo de la Arquitectura Escalonada

La figura IV.4 proporcionada ilustra la arquitectura escalonada del software que se genera al integrar el “Framework IoT” en una aplicación para sistemas embebidos.

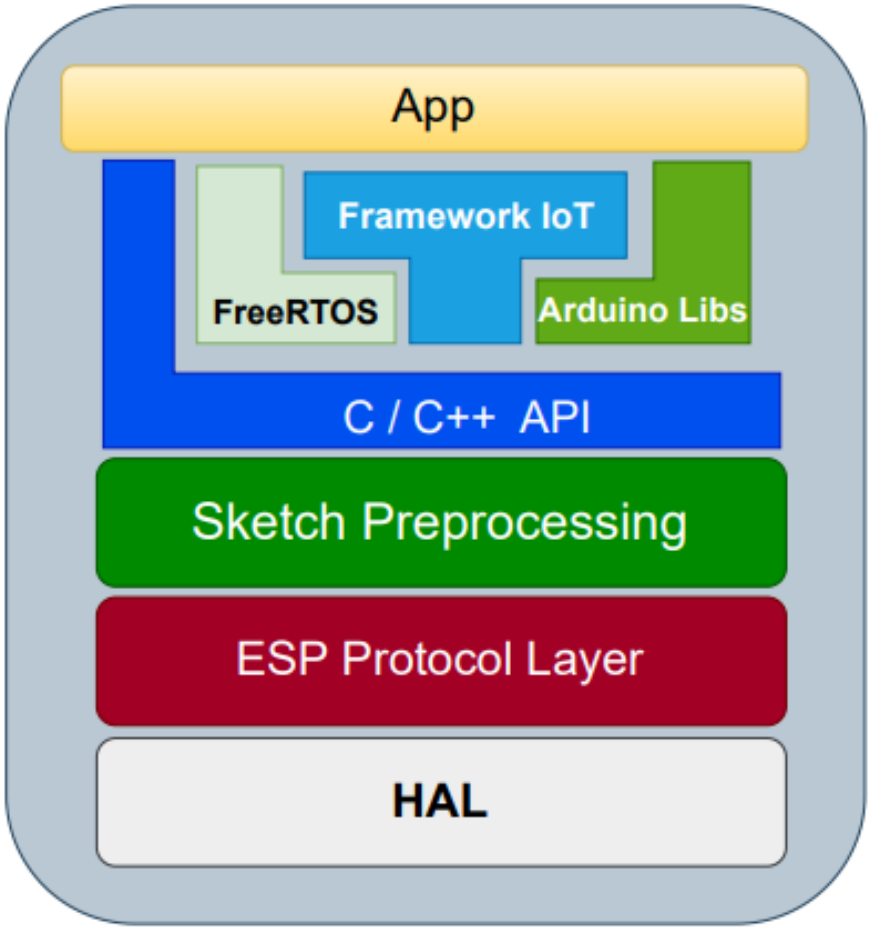


Figura IV.4.- Arquitectura Escalonada.

En un principio, en el nivel más alto se encuentra la “App”, que es la aplicación en sí y la que contiene la lógica de negocio específica del proyecto.

El segundo nivel consta de múltiples capas que incluyen la capa de lenguaje C/C++, FreeRTOS, “Framework IoT” y las librerías de Arduino. Estas capas son importantes, ya que ofrecen servicios y recursos necesarios para el funcionamiento de la aplicación.

La figura muestra, también, cómo el “Framework IoT” actúa como un intermediario que simplifica la interacción entre la aplicación y las otras capas, como las librerías de Arduino y FreeRTOS conservando el acceso directo a estas tecnologías para una mayor flexibilidad y control.

La capa inferior “Sketch Preprocessing” actúa como un enlace transformando el “Sketch” de Arduino (escrito en el lenguaje de programación “Processing” o incluso C/C++) a un código C/C++ más puro o especifico a la plataforma. El empleo de “Processing” se justifica por sus simplificaciones y extensiones que lo hacen más accesible y fácil de usar.

Este proceso de transformación permite que el código se compile con herramientas estándar de C++, garantizando la compatibilidad con una amplia gama de hardware y sistemas, incluyendo las placas ESP32.

Entonces la capa “ESP Protocol Layer” se encarga de la compilación utilizando un compilador específico de C/C++ para ESP32; este compilador suele estar basado en GCC e incluye, de ser necesario, las bibliotecas que son usadas en el sketch.

Finalmente, se encuentra la “Hardware Abstraction Layer” (HAL). La “HAL” es una capa de abstracción que simplifica la tarea de compilar código para sistemas embebidos. Esta capa asegura la portabilidad y la integración del hardware y software, optimizando el rendimiento abstrayendo la complejidad del hardware.

En la figura IV.5, se aprecia un análisis más detallado de esta arquitectura, se observa que el bloque etiquetado como “Framework IoT” podría actuar como un intermediario entre la capa de la aplicación (“App”) y las demás capas. Este nivel de indirección introduce una capa adicional de abstracción en la comunicación (en particular con FreeRTOS). El objetivo principal de esta abstracción es proporcionar métodos de integración más sencillos dentro de la arquitectura, permitiendo al mismo tiempo el uso directo de las otras capas cuando sea necesario.

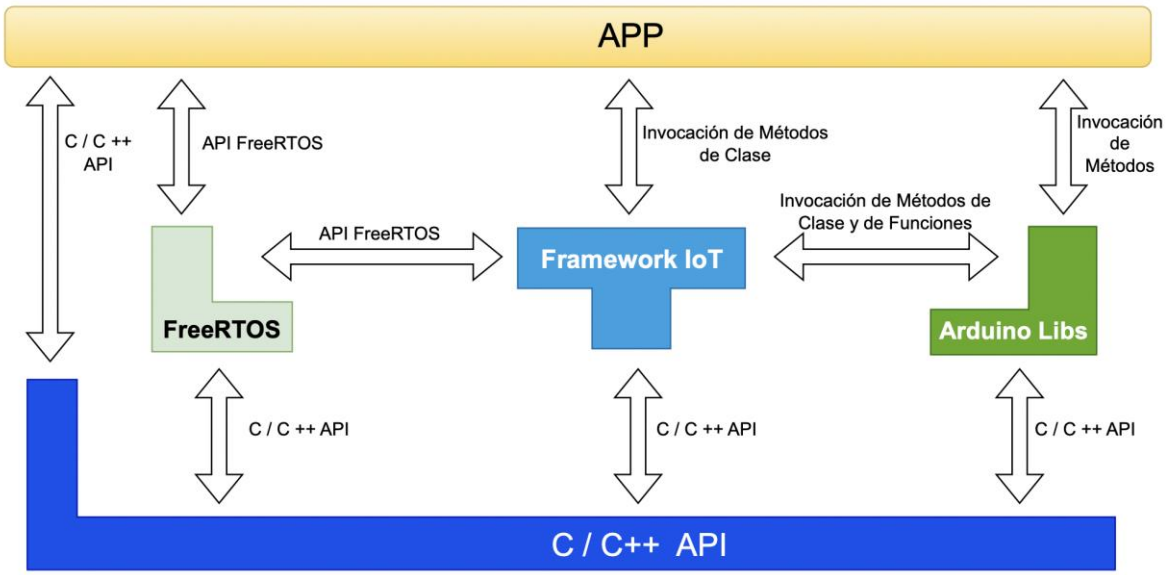


Figura IV.5.- Interacción entre bloques.

La interacción del “Framework IoT” también está representada en el diagrama mediante flechas, por ejemplo, la interacción del framework con FreeRTOS se realiza a través de la “API de FreeRTOS”, mientras que la interacción del framework con las librerías de Arduino se lleva a cabo mediante la invocación de métodos de clase y/o funciones específicas.

Como ya se ha indicado, la capa de aplicación tiene la capacidad de comunicarse directamente con los demás bloques. Sin embargo, para abordar de forma mas adecuada los conceptos como el “Modelo de Actores”, la “Concurrencia” y el “Paralelismo”, es recomendable seguir ciertas directrices y patrones para tal desarrollo.

4.8.2 Diagramas UML

El diagrama que a continuación se presenta es el diagrama de clases, el diagrama de clases muestra las clases del framework, sus relaciones y las entidades que están involucradas con sus atributos y métodos; este diagrama también incluye las etiquetas que identifican el tipo de dato con el que trabaja la clase.

4.8.2.1 Diagrama de Clases

Figura IV.6.- Diagrama de Clases.

El diagrama de Clases presentado en la figura IV.6 muestra elementos clave de la programación orientada a objetos, como la herencia y polimorfismo aplicados a la estructura compuesta del sistema.

En sí, el diagrama muestra la implementación de herencia a través de la clase “xTaskBase”,

fungiendo como la clase padre de las demás clases. Entonces las clases “xbTask” y “xTaskS” se derivan de “xTaskBase”, compartiendo funcionalidades comunes establecidas en ella, pero también introduciendo características más particulares.

El diagrama también resalta cómo las clases derivadas utilizan plantillas para lograr un comportamiento “genérico”. Esto significa que operan de manera consistente independientemente del tipo de dato que manejan.

Un claro ejemplo de esto se encuentra en las estructuras "Pilas" y "Colas". Estas estructuras siguen los principios de "último en entrar, primero en salir" y "primero en entrar, primero en salir", respectivamente. Las estructuras de "Pilas" y "Colas" implementan un comportamiento genérico, ya que el tipo de dato manejado (sea un objeto tipo “led” o “sensor” por ejemplo) no afecta la forma operativa de estas estructuras.

Por último, el diagrama muestra una relación de composición entre las clases “xTask” y “xTaskS” y las instancias “xQueue” y “Sender”. En esta relación de composición, una clase (denominada "compuesta") incorpora o se compone de instancias de otra clase (la "componente"). Esta relación conceptualiza en gran medida a la clase compuesta, en la que la clase componente aporta significado al contenedor.

La composición es una relación fuerte, donde el ciclo de vida de los componentes está intrínsecamente vinculado al del contenedor: si se destruye el contenedor, también se destruyen sus componentes.

4.8.2.2 Diagrama de Secuencia

Un diagrama representa cómo se comunican los objetos en diferentes secuencias de interacciones a lo largo del tiempo. Describe cómo se realiza un proceso o una función específica por medio de un flujo de mensajes entre los objetos involucrados.

La figura IV.7 describe la secuencia interactiva del “Framework IoT” como intermediario entre la aplicación y el Framework de FreeRTOS.

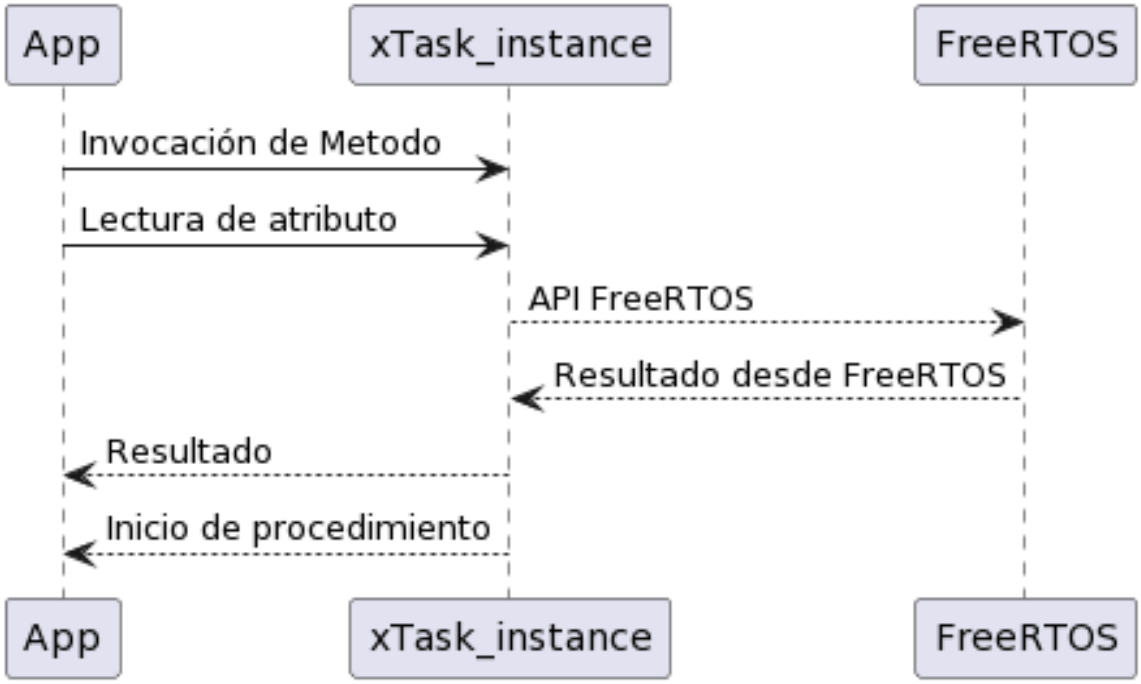


Figura IV.7.- Diagrama de Secuencia.

Se puede apreciar que la aplicación (App), interactúa con las instancias del “Framework IoT” (denominadas “xTask_instance”) mediante la invocación de sus métodos de clase y la consulta de sus atributos. Además, es evidente que la clase “xTask_instance” tiene un papel de intermediario al con la “App” al utiliza el framework de FreeRTOS a través de la API que el mismo FreeRTOS proporciona.

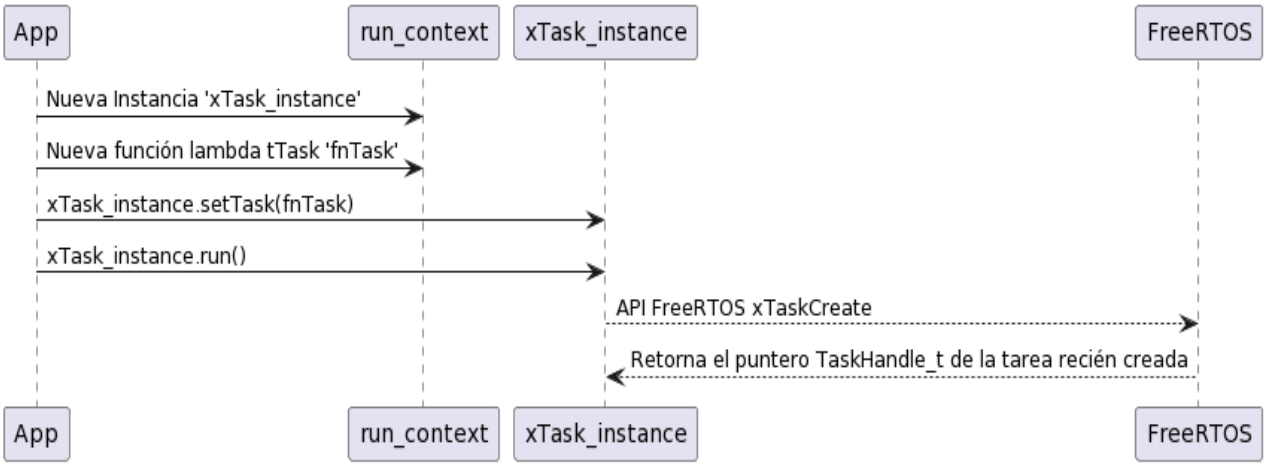


Figura IV.8.- Diagrama de Secuencia de la Creación de una Tarea.

En un proceso más detallado, como es la instanciación de una tarea dentro del "Framework IoT" (figura IV.8), se observa la siguiente secuencia:

- Se necesita una instancia de clase que sirva como intermediario.
- Hay que generar una función de tipo "tTask" que será la tarea a ejecutar.
- Hay que invocar el método "run()" para invocar la creación de la tarea desde FreeRTOS.

Acto seguido la instancia "xTask_instance" llama a la función "xTaskCreate" de la API de FreeRTOS para recibir el puntero "TaskHandle_t" como referencia de la tarea recién creada en FreeARTOS; hay que mencionar que este puntero es conservado por la clase "xTask_instance" como encapsulamiento del ya mencionado puntero "TaskHandle_t".

4.8.3 Comparación del Framework IoT con otras plataformas y frameworks.

El ESP32 de Espressif es una plataforma de hardware popular para proyectos del Internet de las Cosas (IoT), entonces, es de esperar que hoy se cuente con diferentes plataformas, frameworks y bibliotecas disponibles que facilitan su integración en diversos proyectos.

Algunos de las plataformas y frameworks más conocidas son:

- Arduino: Arduino es un entorno de desarrollo integrado (IDE) que facilita su uso con una gran cantidad de bibliotecas que simplifican el trabajo con diferentes sensores y actuadores.

- ESP-IDF (Espressif IoT Development Framework): Es el framework oficial de Espressif para el ESP32. Ofrece un control detallado sobre el hardware y es más adecuado para aplicaciones de bajo nivel (Espressif, 2023).

- Micropython: Es el lenguaje de programación Python adaptado para microcontroladores. Micropython en ESP32 es una opción para abstracciones de alto nivel en donde el acceso a las características de bajo nivel no es una prioridad (MicroPython, 2023).

- NodeMCU (Lua): NodeMCU utiliza el lenguaje de programación Lua, ofreciendo un equilibrio entre facilidad de uso y acceso a características avanzadas. No es tan utilizado como Arduino o Micropython, pero es una buena alternativa a estos (Nodemcu, 2023).

- PlatformIO: Es una plataforma que integra varios frameworks, como Arduino y ESP-IDF. Ofrece un entorno de desarrollo sofisticado para trabajar en proyectos complejos que necesitan gestionar múltiples ambientes y frameworks (PlatformIO, 2022).

La comparación que más interesa en este trabajo es la de Arduino y ESP-IDF, esto principalmente porque el "Framework IoT" toma como base para su ejecución al framework de Arduino y el ESP-IDF es el framework oficial de Espressif para la placa ESP32.

A continuación, se presenta una tabla comparativa de características cualitativas entre los ya mencionados frameworks.

Tabla IV.22.- Tabla comparativa de los frameworks Arduino y ESP-IDF.

Característica a Evaluar	**Arduino**	**ESP-IDF**
Facilidad de Uso y Accesibilidad	Arduino está diseñado para ser fácil de usar con solo conocimientos básicos de programación.	Tiene una curva de aprendizaje pronunciada y requiere un conocimiento más profundo de C/C++ así como del hardware ESP32.
Comunidad y Soporte	Tiene una gran comunidad de usuarios, así como un amplio desarrollo de librerías disponibles.	Cuenta con una comunidad mediana, pero activa, especializada y en crecimiento. El soporte proviene principalmente de la empresa Espressif y de los usuarios avanzados.
Eficiencia y Rendimiento	Adecuado para aplicaciones menos complejas y menos exigentes en términos de rendimiento.	Recomendado en el desarrollo de aplicaciones complejas y proyectos de IoT que requieren características avanzadas como el uso de Bluetooth, Wi-Fi, y tareas en tiempo real.

4.8.4 C/C++ y FreeRTOS como base del desarrollo del Framework IoT

Para el trabajo desarrollado en esta tesis es importante destacar el uso de los lenguajes C/C++ y FreeRTOS con respecto a Arduino y ESP-IDF, ya que son una de las bases para la construcción del "Framework IoT".

El empleo de FreeRTOS y de POO en C/C++ genera robustez en los desarrollos de ESP-IDF. El ESP-IDF integra a FreeRTOS como un componente, ofreciendo diferentes implementaciones para soportar el procesamiento simétrico en múltiples núcleos (SMP) de los chips ESP. El ESP-IDF FreeRTOS se basa en el FreeRTOS Vanilla v10.5.1 pero incorporando modificaciones significativas para soportar SMP, optimizado para un máximo de dos núcleos.

El ESP-IDF soporta características avanzadas de C++ (C++23 con extensiones GNU) como el manejo de excepciones, multithreading y RTTI (Runtime Type Information). Soporta, también, hilos de C++ que se implementan sobre "pthreads", que a su vez envuelven las tareas de FreeRTOS. Esto significa que puedes aprovechar las capacidades de multitarea de FreeRTOS en un entorno de programación orientado a objetos con C++.

El uso de C/C++ y FreeRTOS en Arduino también presenta varias ventajas, por ejemplo:

- Compatibilidad: Aunque un poco más reducida va versión de C/C++ de Arduino es totalmente funcional, se puede aprovechar perfectamente la programación orientada a objetos con una amplia gama de bibliotecas (entre ellas FreeRTOS 11.0.1-4) (Arduino Rreference Libraries Freertos, 2023) y ejemplos disponibles para Arduino, lo cual facilita la implementación de una variedad de funciones y características.

- Integración con PlatformIO: PlatformIO ofrece una integración fluida con el ecosistema de Arduino, permitiendo el uso de herramientas y versiones de C/C++ más recientes en un entorno más avanzado y con características adicionales.

- Soporte Multiplataforma: PlatformIO es compatible con varios sistemas operativos con sus propias versiones de C/C++, lo que facilita el desarrollo en diferentes entornos.

Sin embargo, al incorporar FreeRTOS en Arduino, se observa que la claridad de la sintaxis y el sistema de tipado están un poco alejados de la fluidez que proporciona la semántica de la programación orientada a objetos. Dado que es habitual que algunos frameworks sirvan como base para otros, uno de los objetivos principales del 'Framework IoT' es agregar una semántica más conceptualizada tanto en el tipado como la sintaxis. Esto se logra mediante el uso de varios elementos clave de C/C++, como la Programación Orientada a Objetos (POO), las expresiones Lambda y las plantillas (templates) para el soporte de sistemas genéricos.

A partir de esto es perceptible que las diferencias entre los dos radican principalmente en:

- La complejidad del desarrollo y de la mejora del rendimiento, en donde ESP-IDF es más minucioso con componentes y versiones específicas.

- Desarrollo acelerado del prototipado utilizando bloques de librerías y funciones con un soporte más amplio pero genérico.

Y como ya se ha mencionado estas diferencias se ven reflejadas en su “tipado”, sintaxis y semántica; las siguientes tablas de código muestran el mismo ejercicio de programación para una comparación más efectiva.

Tabla IV.23.- Ejercicio Blink en ESP-IDF.

ESP-IDF

```
1  #include <stdio.h>
2  #include "freertos/FreeRTOS.h"
3  #include "freertos/task.h"
4  #include "driver/gpio.h"
5  #include "esp_log.h"
6  #include "led_strip.h"
7  #include "sdkconfig.h"
8
9  static const char *TAG = "example";
10
11 /*
12 Use project configuration menu (idf.py menuconfig) to choose the GPIO to blink,
13    or you can edit the following line and set a number here.
14 */
15 #define BLINK_GPIO CONFIG_BLINK_GPIO
16
17 static uint8_t s_led_state = 0;
18
19 #ifdef CONFIG_BLINK_LED_STRIP
20
21 static led_strip_handle_t led_strip;
22
23 static void blink_led(void)
24 {
25    /* If the addressable LED is enabled */
26    if (s_led_state) {
27       /* Set the LED pixel using RGB from 0 (0%) to 255 (100%) for each color */
```

```
28          led_strip_set_pixel(led_strip, 0, 16, 16, 16);
29          /* Refresh the strip to send data */
30          led_strip_refresh(led_strip);
31      } else {
32          /* Set all LED off to clear all pixels */
33          led_strip_clear(led_strip);
34      }
35  }
36
37  static void configure_led(void)
38  {
39      ESP_LOGI(TAG, "Example configured to blink addressable LED!");
40      /* LED strip initialization with the GPIO and pixels number*/
41      led_strip_config_t strip_config = {
42          .strip_gpio_num = BLINK_GPIO,
43          .max_leds = 1, // at least one LED on board
44      };
45  #if CONFIG_BLINK_LED_STRIP_BACKEND_RMT
46      led_strip_rmt_config_t rmt_config = {
47          .resolution_hz = 10 * 1000 * 1000, // 10MHz
48          .flags.with_dma = false,
49      };
50      ESP_ERROR_CHECK(led_strip_new_rmt_device(&strip_config, &rmt_config, &led_strip));
51  #elif CONFIG_BLINK_LED_STRIP_BACKEND_SPI
52      led_strip_spi_config_t spi_config = {
53          .spi_bus = SPI2_HOST,
54          .flags.with_dma = true,
55      };
56      ESP_ERROR_CHECK(led_strip_new_spi_device(&strip_config, &spi_config, &led_strip));
```

```
57 #else
58 #error "unsupported LED strip backend"
59 #endif
60     /* Set all LED off to clear all pixels */
61     led_strip_clear(led_strip);
62 }
63 
64 #elif CONFIG_BLINK_LED_GPIO
65 
66 static void blink_led(void)
67 {
68     /* Set the GPIO level according to the state (LOW or HIGH)*/
69     gpio_set_level(BLINK_GPIO, s_led_state);
70 }
71 
72 static void configure_led(void)
73 {
74     ESP_LOGI(TAG, "Example configured to blink GPIO LED!");
75     gpio_reset_pin(BLINK_GPIO);
76     /* Set the GPIO as a push/pull output */
77     gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
78 }
79 
80 #else
81 #error "unsupported LED type"
82 #endif
83 
84 void app_main(void)
85 {
```

```
86
87    /* Configure the peripheral according to the LED type */
88    configure_led();
89
90    while (1) {
91        ESP_LOGI(TAG, "Turning the LED %s!", s_led_state == true ? "ON" : "OFF");
92        blink_led();
93        /* Toggle the LED state */
94        s_led_state = !s_led_state;
95        vTaskDelay(CONFIG_BLINK_PERIOD / portTICK_PERIOD_MS);
96    }
97 }
```

Tabla IV.24.- Ejercicio Blink en Arduino.

Arduino

```
1  #include <Arduino_FreeRTOS.h>
2
3  // define two tasks for Blink & AnalogRead
4  void TaskBlink( void *pvParameters );
5
6  // the setup function runs once when you press reset or power the board
7  void setup() {
8
9    // initialize serial communication at 9600 bits per second:
10   Serial.begin(9600);
11
12   while (!Serial) {
```

```
13    ; // wait for serial port to connect. Needed for native USB, on LEONARDO, MICRO, YUN,
14  and other 32u4 based boards.
15   }
16
17   // Now set up two tasks to run independently.
18   xTaskCreate(
19     TaskBlink
20     , (const portCHAR *)"Blink"   // A name just for humans
21     , 128  // This stack size can be checked & adjusted by reading the Stack Highwater
22     , NULL
23     , 2  // Priority, with 3 (configMAX_PRIORITIES - 1) being the highest, and 0 being the
24  lowest.
25     , NULL );
26
27   // Now the task scheduler, which takes over control of scheduling individual tasks, is
28  automatically started.
29  }
30
31  void loop()
32  {
33   // Empty. Things are done in Tasks.
34  }
35
36  /*--------------------------------------------*/
37  /*---------------------- Tasks ---------------------*/
38  /*--------------------------------------------*/
39
40  void TaskBlink(void *pvParameters)  // This is a task.
41  {
    (void) pvParameters;
```

```
42
43   // initialize digital LED_BUILTIN on pin 13 as an output.
44   pinMode(LED_BUILTIN, OUTPUT);
45
46   for (;;) // A Task shall never return or exit.
47   {
48     digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
49     vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one second
50     digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
51     vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one second
52   }
}
```

Tabla IV.25.- Ejercicio Blink en el "Framework IoT"

Framework IoT

```
1   #include <Arduino.h>
2   #include "xTask.cpp"
3
4   int LED_BUILTIN = 2;
5   void TaskBlink(void *pvParameters);
6
7   xTask x_task("Blink", 512, 1);
8
9   void setup() {
10   x_task.setT(TaskBlink);
11   x_task.run();
12  }
```

```
13
14  void loop() {
15  }
16
17  void TaskBlink(void *pvParameters)
18  {
19   pinMode(LED_BUILTIN, OUTPUT);
20   for (;;)
21   {
22     // turn the LED on (HIGH is the voltage level)
23     digitalWrite(LED_BUILTIN, HIGH);
24     // wait for one second
25     vTaskDelay( 3000 / portTICK_PERIOD_MS );
26     // turn the LED off by making the voltage LOW
27     digitalWrite(LED_BUILTIN, LOW);
28     // wait for one second
29     vTaskDelay( 3000 / portTICK_PERIOD_MS );
30   }
31  }
```

En los códigos anteriores observa, más que la disminución de código, la semántica y concepción de la idea del ejemplo.

Aunque las diferencias son palpables entre los frameworks, también se distingue los esfuerzos de la plataforma PlatformIO por generar una convergencia entre ellos. La compatibilidad con PlatformIO es importante por ser una de las plataformas base para este desarrollo. La compatibilidad es un aporte que permite la coexistencia de ambos frameworks, lo que significa que se pueden compilar ambos desde la fuente y agregarlos al firmware. Esto ofrece una gran flexibilidad para las ventajas de ambos frameworks en un solo proyecto.

V RESULTADOS

A continuación, se agrega una tabla con la descripción de los resultados obtenidos.

Tabla V.1.-Listado y descripción de los resultados obtenidos

Nombre	Descirpción
Framework	Código fuente anexado en esta tesis y adjunto en una memoria USB.
Lista de herramientas de desarrollo	Lista de herramientas de desarrollo, así como las librerías usadas en esta tesis para la creación del framework. ***Nota**: Adjuntas en la memoria USB.*
Dos artículos en revistas indexadas.	1er Artículo [Anexo 5] Título: “El patrón de eventos Handler en Arduino para la implementación de un protocolo simple de comunicación basado en eventos” Revista/Congreso: IEEE, RVP-AI/ROC&C 2022 ISBN: 978-607-95630-8-0 2do Artículo [Anexo 6] Título: “Desarrollo de una Librería Concurente en FreeRTOS en POO” Revista/Congreso: ELECTRO 2023 ISSN: 1405-2172

Validación externa del Framework	Carta membretada de la empresa Advantage Tecnología [Anexo 7], validando el uso del framework en un proyecto propio de la empresa.
Ejemplos codificados del uso del framework.	Diez ejemplos codificados del uso del framework resultado de las iteraciones empleadas para la implementación del mismo. ***Nota**: También adjuntos en la memoria USB.*
Talleres	Dos talleres de 30 hrs. a nombre del TecNM. Sede: Instituto Tecnológico de Chihuahua II con título y folio, respectivamente. Desarrollo de Aplicaciones Single Web Applications – TNM-028-15-2021/I01. Se anexan constancias [Anexo 8] Introducción a los Sistemas IoT con Placas ESP – TNM-028-15-2021/I01. Se anexan constancias [Anexo 9]

VI CONCLUSIONES

A partir del trabajo realizado, se listan una serie de conclusiones y recomendaciones técnicas.

El Internet de las cosas (IoT) es una de las tecnologías más ampliamente utilizadas en la actualidad, lo que hace que el desarrollo de herramientas de software para simplificar y acelerar su implementación sea de gran importancia. En este contexto, los frameworks y lenguajes de alto nivel se han convertido en la opción más adecuada para abordar este desafío. Sin embargo, a pesar de que esto conlleva sus propios retos, contribuir a través de la implementación de un 'FRAMEWORK PARA LA CONSTRUCCIÓN DE SISTEMAS EMBEBIDOS IOT' es de suma importancia en todos los aspectos.

Debido a que los dispositivos IoT abarcan la conectividad a Internet y su funcionalidad operativa (Guerrero-Ulloa, 2023), es importante concebirlos con procesos concurrentes y/o paralelos ya que resulta crucial para la interacción entre las tareas de conectividad y las operativas.

Aunque el núcleo del Internet de las Cosas es el Cómputo en la Nube y se reconocen las diferencias naturales entre tecnologías, es importante destacar que los conceptos e ideas que implementan son transversales. Por lo tanto, "*aún se necesitan frameworks y utilerías para IoT que reduzcan estas diferencias al proporcionar abstracciones de alto nivel como: programación orientada a objetos, concurrencia y paralelismo*".

El núcleo de framework aquí expuesto implementa técnicas y elementos tales como Programación Orientada a Objetos (POO), Lambdas y Patrones de Diseño con FreeRTOS; tal desarrollo proporciona versatilidad en el manejo de tareas concurrentes y paralelas.

El uso de POO genera componentes de software reutilizables que ahorran tiempo al proporcionar acceso al código que realiza una tarea de programación (AWS, 2023) que representa una abstracción de algún artefacto o concepto.

Esta última idea no interfiere con la representación de esas tareas como sucesos ocurridos en el mismo umbral de tiempo durante su ejecución, es decir, de forma concurrentemente, potencialmente paralela y como partes de un cálculo o proceso sin importar el orden particular de ejecución.

El uso de colas en FreeRTOS está abierto a la implementación del usuario final, si bien esto proporciona libertad creativa, cuando se habla de concurrencia es mejor tener un modelo o guía de referencia.

El Modelo de Actores (MA) es la guía que aquí se ha adoptado para este fin. La clase "xTask" junto con la clase "xQueue" pueden llegar a representar la implementación de un Actor y generar la aproximación al Modelo de Actores en este framework.

Son perceptible las condiciones descritas por Nelson Rodríguez al comentar que "El advenimiento de la concurrencia masiva a través de la computación en la nube y las arquitecturas de computadora multi núcleo ha estimulado el interés en el Modelo de Actores" (Rodríguez, 2019), sin embargo, se cree que aún está por venir un mayor auge para este modelo.

La elección de un protocolo de comunicación IoT no es una tarea sencilla; los protocolos pueden variar el comportamiento de las aplicaciones (M. O. Al Enany, 2021), la comprobación de su uso dentro del framework fue una de las actividades más importantes, la combinación con el framework IoT genera un mecanismo para recibir mensajes y/o elementos desde internet para gestionarlos en tareas de FreeRTOS.

La necesidad de desarrollos no es ajena a la I4.0, así que la validación del framework con respecto al protocolo Modbus proporciona una certeza en el uso del mismo en ambientes industriales.

Como se menciona en el marco teórico el conocimiento del dominio es apreciable cuando se pretende mejorar la semántica o agregar un DSL al framework. Los DSL's se destacan por

su capacidad para representar y resolver problemas de un dominio específico de manera más clara y acertada. Estas características han generado una mejora en la adopción y uso de framework aquí presentado.

El patrón de diseño empleado para generar un pseudo DSL es el encaminamiento de métodos ya que de forma natural pude incluir las reglas internas del dominio, los métodos encadenados, construyen expresiones y operaciones de manera más concisa y expresiva con una sintaxis fluida (Fowler, Defining Domain-Specific Languages, 2010, pág. 62) encapsulando lógica compleja, abstracta y de alto nivel. Por ejemplo, la llamada al método "run" de la clase "xTask" puede ser invocado justo después del método "setTask" lo que genera la semántica o lenguaje natural de "*ejecuta la tarea recién agregada*".

Es acertada la apreciación expuesta en Thoughtworks al mencionar que

"Todos los indicadores apuntan hacia un futuro de IoT que depende de la investigación rápida implementada a través de actualizaciones modulares de hardware y software para modificar ese hardware.
Dada la tendencia del desarrollo de software hacia todas las cosas ágiles, parece lógico que el desarrollo ágil de hardware, a través de una mayor modularidad y otras innovaciones, se convierta en el camino a seguir en nuestro futuro en red." (McCarthy, 2015)

Es concluyente también que para explotar y mejorar la tasa de éxito el potencial de IoT, se requieren metodologías de desarrollo bien definidas.

Entonces todo depende de la investigación rápida e implementada a través de actualizaciones modulares de hardware y software para modificar el comportamiento de ese hardware (McCarthy, 2015).

En consecuencia, para el desarrollo de un IoT, es importante analizar la tecnología existente y disponible, e incluso desarrollar los dispositivos y mecanismo de software necesarios (Gleiston Guerrero-Ulloa, 2023).

Además, cabe señalar que los sistemas IoT no solo se componen de aplicaciones de software, hardware (sensores/actuadores) y mecanismos de comunicación. Además, también se debe considerar la interacción entre cosas y personas a través de interfaces bien definidas (McCarthy, 2015).

Con respecto al “Framework IoT”, el que aquí se desarrolla, se puede concluir que:

Sí, es posible emplear la programación orientada a objetos (POO) para conceptualizar y contextualizar las tareas de FreeRTOS mediante el uso de instancias y objetos.

Con el C++ de Arduino, el que se usa en esta implementación, se pueden crear clases que representen distintos tipos de tareas, encapsulando la lógica y la API de FreeRTOS para proporcionar una interfaz (constructor, métodos y atributos de la clase) que facilite la configuración y ejecución de dicha tarea.

Los métodos de clase actúan como intermediarios entre FreeRTOS y la aplicación, encapsulando la lógica de ejecución de FreeRTOS y proporcionando una interfaz más fácil de emplear. Incluso se podría llegar al punto de dejar de pensar en términos de FreeRTOS y utilizar solo las definiciones de la realidad del objeto mismo.

Las instancias generadas a través del “Framework IoT” representan conceptos específicos; por ejemplo, pueden ser un sensor y/o un actuador con su propio contexto de ejecución basado en una tarea de FreeRTOS.

Del "Framework IoT," se obtiene la instanciación de definiciones de tareas específicas de FreeRTOS. Esto implica que, aunque una clase base puede definir tareas con características y comportamientos compartidos, las clases derivadas tienen la capacidad de añadir funcionalidades específicas sin tener que duplicar toda la estructura de la clase base. Este caso se ejemplifica con las clases "xbTask" y "xTaskS", las cuales se derivan de la clase "xTaskBase".

A través de una jerarquía de clases el “Framework IoT” puede modelar las relaciones entre objetos y tareas para obtener una representación más compacta y un uso más eficiente de la memoria; una característica apreciable tomando en cuenta la variedad de similitues entre diferentes tipos de objetos que coexisten dentro del framework.

Las funciones lambda en C++ proporcionan flexibilidad y código conciso al "Framework IoT.". Este tipo de comportamiento de adaptabilidad permite definir funciones anónimas de forma concisa y directa en el lugar donde se necesiten, evitando la declaración de funciones adicionales fuera del contexto principal.

Al evitar la necesidad de definir funciones separadas, las lambdas contribuyen a reducir la huella y las fugas de memoria en el framework. Son una forma elegante de integrar nuevas funcionalidades de manera localizada, mejorando la expresividad del código con la definición de funciones in situ.

Es importante equilibrar la simplicidad de las lambdas con el mantenimiento del código, ya que, en algunos casos, pueden introducir cierta complejidad más que proporcionar una mejora significativa al desarrollo.

El uso de lambdas es por demás recomendado, sin embargo, es importante recordar que el uso de ellas conlleva el empleo de C++ en sistemas embebidos, que a su vez está sujeto a restricciones y consideraciones específicas del hardware y del compilador.

Por último, el "Framework IoT" que se presenta integra y expone diversas características importantes en el manejo de POO, concurrencia y paralelismo; no obstante, es importante destacar que sigue siendo una prueba de concepto que requiere más trabajo e implementación. Se espera que esfuerzos similares se estén llevando a cabo en otras universidades e institutos para que, en un futuro cercano, puedan ser incorporados nativamente a las versiones futuras de FreeRTOS como parte de su conjunto de instrucciones o de su filosofía de desarrollo.

VII BIBLIOGRAFÍA

Agha, G. (Septiembre de 1990). Concurrent object-oriented programming. *Communications of the ACMVolume 33Issue 9*, págs. 125-141.

Agha, G. (2004). *ACTORS: A Model of Concurrent Computation in Distributed Systems.* Cambridge, Massachusetts: MIT Artificial Intelligence Laboratry.

Arduino Boards. (1 de 6 de 2023). *Arduino Boards*. Obtenido de https://store-usa.arduino.cc/collections/boards?selectedStore=us

Arduino. (1 de 6 de 2023). *Arduino.cc*. Obtenido de https://www.arduino.cc/en/software

Arduino. (1 de 6 de 2023). *FreeRTOS - Arduino Reference*. Obtenido de FreeRTOS: https://www.arduino.cc/reference/en/libraries/freertos/

AWS. (01 de 06 de 2023). *AWS FreeRTOS. Un sistema operativo de tiempo real para dispositivos limitados por los recursos.* Obtenido de https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html

AWS. (1 de 6 de 2023). *AWS Training and Certification Blog*. Obtenido de Digital courses on Coursera and edX: Identity and Access Management (IAM), Internet of Things (IoT), and Data Lakes: https://aws.amazon.com/es/blogs/training-and-certification/new-digital-courses-identity-and-access-management-and-internet-of-things/

AWS. (15 de 01 de 2023). *Características de AWS IoT Core*. Obtenido de AWS IoT Core: https://aws.amazon.com/es/iot-core/features/

AWS IoT. (1 de 6 de 2023). *AWS IoT*. Obtenido de https://aws.amazon.com/es/iot/

AWS IoT Core Developer Guide. (1 de 6 de 2023). *AWS IoT Core Developer Guide*. Obtenido de https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html

B. Christophe, M. B. (Junio de 2011). The Web of things vision: Things as a service and interaction patterns,. *Bell Labs Technical Journal, vol. 16, no. 1*, págs. 55-61.

Barry, R. (7 de 2018). *Mastering the FreeRTOS™ Real Time Kernel.* Obtenido de 161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

Boards PlatformIO. (1 de 6 de 2023). *Docs*. Obtenido de PlatformIO: https://docs.platformio.org/en/latest/boards/index.html

C++ Reference. (1 de 6 de 2023). *C++ Reference*. Obtenido de https://en.cppreference.com

C++ Reference Lambda Expressions. (1 de 6 de 2023). *C++ Reference Lambda Expressions.* Obtenido de https://en.cppreference.com/w/cpp/language/lambda

C4IR.CO. (2021). *Informe de inteligencia global, INTERNET DE LAS COSAS (IoT).* Medellín, Colombia: C4IR.CO.

Cao Vien Phung, J. D. (23 de Febrero de 2021). *Arxiv.* Obtenido de An Experimental Study of Network Coded REST: https://arxiv.org/pdf/2003.00245.pdf

CEI. (2003). *INTERNATIONAL STANDARD, IEC 61131-1.* Ginebra, Suiza: COMMISSION ELECTROTECHNIQUE INTERNATIONALE - CEI.

Chatzigeorgiou A., S. G. (17 de Junio de 2002). Evaluating Performance and Power of Object-Oriented Vs. Procedural Programming in Embedded Processors. *Ada-Europe*, págs. 65-75.

CLAUDIA PONS, R. G. (2010). *DESARROLLO DE SOFTWARE DIRIGIDO POR MODELOS.* Buenos Aires - Argentina: Editorial de la Universidad Nacional de La Plata.

Codesys. (30 de 09 de 2019). *LEVERAGE OBJECT-ORIENTED INDUSTRIAL PROGRAMMING*. Obtenido de https://www.codesys.com/news-events/publications/article/leverage-object-oriented-industrial-programming.html

Codesys. (14 de 10 de 2020). *WHICH IEC 61131-3 PROGRAMMING LANGUAGE IS BEST? PART 1*. Obtenido de https://www.codesys.com/news-events/press-releases/article/which-iec-61131-3-programming-language-is-best-part-1.html

Codesys. (15 de 01 de 2023). *Codesys The System*. Obtenido de THE COMPREHENSIVE SOFTWARE SUITE FOR AUTOMATION TECHNOLOGY: https://www.codesys.com/the-system.html

CPP Reference. (18 de 7 de 2023). *Classes*. Obtenido de https://en.cppreference.com/w/cpp/language/classes

CPP Reference. (17 de 7 de 2023). *Keyword Class*. Obtenido de https://en.cppreference.com/w/cpp/keyword/class

Deitel, P. &. (2012). *Cómo programar Java (9a. ed. --.).* México: Pearson.

Developer Mozilla. (s.f.). *MDN.* Obtenido de Generalidades del protocolo HTTP: https://developer.mozilla.org/es/docs/Web/HTTP/Overview

Digikey FreeRTOS Queue Example. (1 de 6 de 2023). *Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example)*. Obtenido de https://www.digikey.com.mx/en/maker/projects/introduction-to-rtos-solution-to-part-5-freertos-queue-example/72d2b361f7b94e0691d947c7c29a03c9

Eclipse Foundation. (1 de 6 de 2023). *Eclipse IoT*. Obtenido de Leading open source community for IoT innovation: https://iot.eclipse.org/

Espressif. (27 de 8 de 2023). *FreeRTOS (ESP-IDF)*. Obtenido de docs.espressif.com: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos_idf.html

Espressif. (20 de 9 de 2023). *FreeRTOS Overview*. Obtenido de https://docs.espressif.com/: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html

Expressif. (1 de 6 de 2023). *Expressif*. Obtenido de Systems Software: https://www.espressif.com/en/products/software

FastAPI. (15 de 01 de 2023). *Path Parameters*. Obtenido de Learn Fastapi: https://fastapi.tiangolo.com/tutorial/path-params/

Fortino, G. &. (1 de Enero de 2017). Modeling Opportunistic IoT Services in Open IoT Ecosystems. *CEUR Workshop Proceedings*, págs. 90-95.

Fowler, M. (3 de Junio de 2003). *Foundation Platform.* Obtenido de https://martinfowler.com/: https://martinfowler.com/bliki/FoundationPlatform.html

Fowler, M. (3 de Junio de 2003). *Harvested Platform.* Obtenido de https://martinfowler.com: https://martinfowler.com/bliki/HarvestedFramework.html

Fowler, M. (2010). *Defining Domain-Specific Languages.* Addison-Wesley Professional.

Fowler, M. (15 de 01 de 2023). *Function As Object*. Obtenido de martinFowler.com: https://martinfowler.com/bliki/FunctionAsObject.html

FreeRTOS. (15 de 4 de 2023). *Featured FreeRTOS IoT Integration*. Obtenido de https://www.freertos.org/featured-freertos-iot-integration-targeting-an-espressif-esp32-c3-risc-v-mcu/

FreeRTOS. (1 de 6 de 2023). *xTaskCreate()*. Obtenido de This page describes the RTOS xTaskCreate() FreeRTOS API function which is part of the RTOS task control API. : https://www.freertos.org/a00125.html

FreeRTOS Inter-task Communication. (1 de 6 de 2023). *FreeRTOS Inter-task Communication*. Obtenido de https://www.freertos.org/Inter-Task-Communication.html

FreeRTOS Queues. (1 de 6 de 2023). *FreeRTOS Queues*. Obtenido de https://www.freertos.org/Embedded-RTOS-Queues.html

FreeRTOS RTOS Implementation. (1 de 6 de 2023). *FreeRTOS RTOS Implementation.* Obtenido de https://www.freertos.org/implementation/main.html

FreeRTOS Tasks and Co-routines. (1 de 6 de 2023). *FreeRTOS Tasks and Co-routines.* Obtenido de https://www.freertos.org/taskandcr.html

FreeRTOS™. (1 de 6 de 2023). *FreeRTOS*. Obtenido de https://freertos.org/

Gleiston Guerrero-Ulloa, C. R.-D. (10 de 01 de 2023). Agile Methodologies Applied to the Development of Internet of Things (IoT)-Based Systems: A Review. *Sensors 2023 Vol. 2*, pág. 790. Obtenido de https://www.mdpi.com/1424-8220/23/2/790

Groombridge, D. (17 de Octubre de 2022). *Gartner.* Obtenido de Las 10 principales tendencias tecnológicas estratégicas de Gartner para 2023: https://www.gartner.mx/es/articulos/las-10-principales-tendencias-tecnologicas-estrategicas-de-gartner-2023

Guerrero-Ulloa, G. R.-D. (2023). Agile Methodologies Applied to the Development of Internet of Things (IoT)-Based Systems: A Review. *Sensors*, 23(2), 790.

Günter, B. (1 de Agosto de 2018). *Thoughtworks.* Obtenido de IoT: Smart Ecosystems are door openers for new business models: https://www.thoughtworks.com/insights/blog/iot-smart-ecosystems-are-door-openers-new-business-models

Hewitt, C. (Marzo de 2017). Actor Model of Computation for Scalable Robust Information Systems. *One IoT is No IoT1. Symposium on Logic and Collaboration for Intelligent Applications*. Obtenido de [https://hal.science/hal-01163534/document#:~:text=The%20Actor%20Model%20is%20a%20mathematical%20theory%20of%20computation%20that,Steiger%201973%3B%20Hewitt%201977%5D

Hivemq. (s.f.). *Hivemq*. Obtenido de MQTT Essentials: https://www.hivemq.com/mqtt-essentials/

HTTP Working Group. (1999). *www.rfc-editor.org.* Obtenido de Hypertext Transfer Protocol -- HTTP/1.1, Method: https://www.rfc-editor.org/rfc/rfc2616.txt

Huawei Cloud. (1 de 6 de 2023). *Infraestructura IoT en la nube*. Obtenido de https://www.huaweicloud.com/intl/es-us/solution/iot/

IBM. (11 de 4 de 2023). *IBM Documentation*. Obtenido de Classes (C++ only): https://www.ibm.com/docs/en/i/7.4?topic=reference-classes-c-only

IBM. (1 de 6 de 2023). *Soluciones IoT*. Obtenido de https://www.ibm.com/mx-es/cloud/internet-of-things

IBM Training. (1 de 6 de 2023). *IBM Certified Solution Architect*. Obtenido de Watson IOT Continuous Engineering V1: https://www.ibm.com/training/certification/C0002300

Intel IoT. (1 de 6 de 2023). *Intel, Inteligentes con Internet de Cosas (IoT)*. Obtenido de https://www.intel.la/content/www/xl/es/internet-of-things/overview.html

Jason Myers Technical Writer, I. (23 de Noviembre de 2021). *DZone Refcard #367.* Obtenido de Data Management for Industrial IoT: https://dzone.com/refcardz/data-management-for-industrial-iot

Learn Microsoft. (1 de 6 de 2023). *Browse Certifications* . Obtenido de Exam AZ-220: Microsoft Azure IoT Developer: https://learn.microsoft.com/en-us/certifications/exams/az-220/

Leo Kelion, E. d. (10 de Febrero de 2021). *BBC NEWS MUNDO*. Obtenido de Escasez de microchips: por qué hay una crisis de semiconductores y cómo puede afectarte: https://www.bbc.com/mundo/noticias-55955119

M. O. Al Enany, H. M. (3 de Julio de 2021). A Comparative analysis of MQTT and IoT application protocols. *International Conference on Electronic Engineering (ICEEM), Menouf, Egypt, 2021*, págs. 1-6.

M. Wollschlaeger, T. S. (2017). The Future of Industrial Communication: Automation Networks in the Era of the Internet of Things and Industry 4.0. *IEEE Industrial Electronics Magazine, vol. 11*, 17-27.

Manuel Díaz, C. M. (Mayo de 2016). State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. *Journal of Network and Computer Applications, Volume 67*, págs. 99-117.

Mariano Luzza, M. B. (2012). Diseño y Construcción de Lenguajes Específicos del Dominio. *Diseño y Construcción de Lenguajes Específicos del Dominio* (págs. 542-546). San Juan, Argentina: WICC, Workshop de Investigadores en Ciencias de la Computación.

Marketing, I. F. (14 de Noviembre de 2019). *Lynx.* Obtenido de WHAT ARE THE MOST POPULAR REAL-TIME OPERATING SYSTEMS?: https://www.lynx.com/embedded-systems-learning-center/most-popular-real-time-operating-systems-rtos

Markets and Markets. (Febrero de 2022). *IoT Market.* Obtenido de Internet of Things (IoT) Market Analysis: https://www.marketsandmarkets.com/Market-Reports/internet-of-things-market-573.html

Mazon-Olivo, B. &. (27 de Mayo de 2021). Internet of Things: State-of-the-art, Computing Paradigms and Reference Architectures. *IEEE Latin America Transactions, 20(1)*, págs. 49-63.

McCarthy, T. (08 de 04 de 2015). *New Technologies, New Methods: Closing the Feedback Loop in the Internet of Things.* . Obtenido de Thoughtworks UX in the Era of IoT: https://www.thoughtworks.com/insights/blog/ux-in-the-era-of-iot

Microchip. (1 de 6 de 2023). *MPLAB® X IDE*. Obtenido de MPLAB® X Integrated Development Environment (IDE): https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide

Microsoft. (16 de 06 de 2023). *Clases genéricas (C++/CLI)*. Obtenido de learn.microsoft.com: https://learn.microsoft.com/es-es/cpp/extensions/generic-classes-cpp-cli?view=msvc-170

Microsoft. (12 de 06 de 2023). *Elección de un protocolo de comunicación de dispositivos*. Obtenido de Learn Microsoft: https://learn.microsoft.com/es-es/azure/iot-hub/iot-hub-devguide-protocols

Microsoft. (16 de 06 de 2023). *Introducción a los genéricos en C++/CLI*. Obtenido de learn.microsoft.com: https://learn.microsoft.com/es-es/cpp/extensions/overview-of-generics-in-visual-cpp?view=msvc-170

Microsoft. (23 de 07 de 2023). *Llamada a una API web desde un cliente .NET (C#)*. Obtenido de Learn Microsoft: https://learn.microsoft.com/es-es/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client

Mqtt.org. (s.f.). *mqtt.org/*. Obtenido de https://mqtt.org/

NATIONAL INSTRUMENTS CORP. (20 de 12 de 2022). *Introducción a Modbus*. Obtenido de https://www.ni.com/es/shop/labview/introduction-to-modbus-using-labview.html

Navid Shariatzadeh, T. L. (2016). Integration of Digital Factory with Smart Factory Based on Internet of Things. *Procedia CIRP*, págs. 512-517.

OASIS Standard. (7 de Marzo de 2019). *OASIS Standard.* Obtenido de MQTT Version 5.0: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf

OPC Foundation. (15 de 01 de 2023). *OPC Technologies, Unified Architecture*. Obtenido de https://opcfoundation.org/about/opc-technologies/opc-ua/

Pacheco, A. E. (2021). USO DE PATRONES DE DISEÑO Y METAPROGRAMACIÓN. *ELECTRO Vol. 45*, 50-55.

Peinado Royero, K. P. (2005). *INTRANET: APLICACIONES EN INSTRUMENTACIÓN INDUSTRIAL.* Cartagena- Colombia: Universidad Tecnológica de Bolívar .

PlatformIO. (8 de 7 de 2022). *Documentation*. Obtenido de docs.platformio.org: https://docs.platformio.org/en/latest/faq/ino-to-cpp.html

PlatformIO. (1 de 6 de 2023). *PlatformIO*. Obtenido de Professional collaborative platform for embedded development: https://platformio.org/

Profibus. (02 de 02 de 2023). *PROFINET: ¿Qué es y cómo funciona?* Obtenido de https://profibus.com.ar/profinet-que-es-y-como-funciona/

Richa Maurya, K. A. (11 de Febrero de 2021). Application of Restful APIs in IOT: A Review. *International Journal for Research in Applied Science & Engineering Technology (IJRASET)*, págs. 145-151.

Richard Coppen, I. C. (2020). *OASIS-OPEN*. Obtenido de OASIS Message Queuing Telemetry Transport (MQTT) TC: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt

Riti, P. (2018). *Practical Scala DSLs.* Apress.

Rockwell Automation. (1 de 6 de 2023). *MES, analítica y IIoT*. Obtenido de https://www.rockwellautomation.com/es-mx/products/software/factorytalk/innovationsuite.html

Rodríguez, N. R. (2019). El modelo de programación de actor aplicado a Edge Computing utilizando Calvin. *SEDICI UNLP*, 970-979.

Soman, S. (2 de Mayo de 2017). *Thoughtworks.* Obtenido de Thoughtworks: https://www.thoughtworks.com/insights/blog/two-opposing-iot-revolutions-play

Spring Guides. (15 de 01 de 2023). *Building REST services with Spring*. Obtenido de Spring Guides: https://spring.io/guides/tutorials/rest/

STM32 Nucleo Boards - Products. (1 de 6 de 2023). *Nucleo Boards - Products*. Obtenido de STM32: https://www.st.com/en/evaluation-tools/stm32-nucleo-boards/products.html

Tai-hoon Kim, C. R. (Noviembre de 2017). Smart City and IoT. *Future Generation Computer Systems, Volume 76*, págs. 159-162.

Thoughtworks. (29 de 5 de 2023). Obtenido de Thoughtworks Technology Radar: www.thoughtworks.com/radar

TIOBE Index Programming Language. (10 de 2 de 2023). *TIOBE*. Obtenido de TIOBE Index Programming Language: https://www.tiobe.com/tiobe-index/

Torres, O. (2020). La internet industrial de las cosas & la inteligencia artificial, el impacto en la economía y las oportunidades de nuevos negocios. *CIES*, 127-139.

Weimin Li, B. W. (2018). A Resource Service Model in the Industrial IoT System Based on Transparent Computing. *Sensors*, 18(4), 981.

Zaslavsky, A. &. (Julio de 2012). Sensing as a Service and Big Data. *Proceedings of the International Conference on Advances in Cloud Computing (ACC)*.

VIII ANEXOS

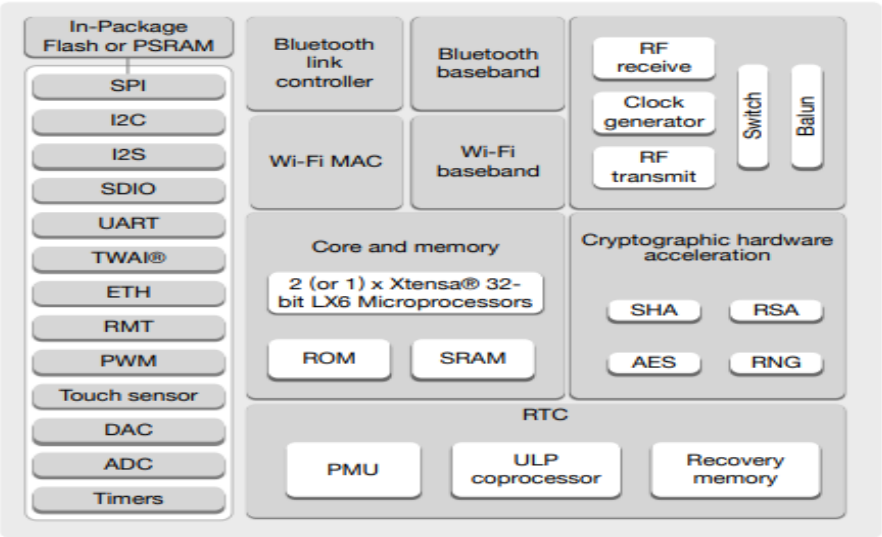
Anexo 1.- Descripción técnica del ESP32

Product Overview

ESP32 is a single 2.4 GHz Wi-Fi-and-Bluetooth combo chip designed with the TSMC low-power 40 nm technology. It is designed to achieve the best power and RF performance, showing robustness, versatility and reliability in a wide variety of applications and power scenarios. The ESP32 series of chips includes ESP32-D0WD-V3, ESP32-D0WDR2-V3, ESP32-U4WDH, ESP32-S0WD (NRND), ESP32-D0WDQ6-V3 (NRND), ESP32-D0WD (NRND), and ESP32-D0WDQ6 (NRND), among which,

- ESP32-S0WD (NRND), ESP32-D0WD (NRND), and ESP32-D0WDQ6 (NRND) are based on chip revision v1 or chip revision v1.1.
- ESP32-D0WD-V3, ESP32-D0WDR2-V3, ESP32-U4WDH, and ESP32-D0WDQ6-V3 (NRND) are based on chip revision v3.0 or chip revision v3.1.

For details on part numbers and ordering information, please refer to Section 1 ESP32 Series Comparison. For details on chip revisions, please refer to ESP32 Chip Revision v3.0 User Guide and ESP32 Series SoC Errata. The functional block diagram of the SoC is shown below



ESP32 Functional Block Diagram

Features

Wi-Fi

- 802.11b/g/n
- 802.11n (2.4 GHz), up to 150 Mbps
- WMM • TX/RX A-MPDU, RX A-MSDU
- Immediate Block ACK
- Defragmentation
- Automatic Beacon monitoring (hardware TSF)
- 4 × virtual Wi-Fi interfaces
- Simultaneous support for Infrastructure Station, SoftAP, and Promiscuous modes Note that when ESP32 is in Station mode, performing a scan, the SoftAP channel will be changed.
- Antenna diversity

Bluetooth®

• Compliant with Bluetooth v4.2 BR/EDR and Bluetooth LE specifications

• Class-1, class-2 and class-3 transmitter without external power amplifier

• Enhanced Power Control

• +9 dBm transmitting power

• NZIF receiver with –94 dBm Bluetooth LE sensitivity

• Adaptive Frequency Hopping (AFH)

• Standard HCI based on SDIO/SPI/UART

• High-speed UART HCI, up to 4 Mbps

• Bluetooth 4.2 BR/EDR and Bluetooth LE dual mode controller

• Synchronous Connection-Oriented/Extended (SCO/eSCO)

• CVSD and SBC for audio codec

• Bluetooth Piconet and Scatternet

• Multi-connections in Classic Bluetooth and Bluetooth LE

• Simultaneous advertising and scanning

CPU and Memory

• Xtensa® single-/dual-core 32-bit LX6 microprocessor(s)

• CoreMark® score: – 1 core at 240 MHz: 504.85 CoreMark; 2.10 CoreMark/MHz Espressif Systems 3 Submit Documentation Feedback ESP32 Series Datasheet v4.3 – 2 cores at 240 MHz: 994.26 CoreMark; 4.14 CoreMark/MHz

• 448 KB ROM • 520 KB SRAM

• 16 KB SRAM in RTC

• QSPI supports multiple flash/SRAM chips

Clocks and Timers

• Internal 8 MHz oscillator with calibration

• Internal RC oscillator with calibration

• External 2 MHz ~ 60 MHz crystal oscillator (40 MHz only for Wi-Fi/Bluetooth functionality)

• External 32 kHz crystal oscillator for RTC with calibration

• Two timer groups, including 2 × 64-bit timers and 1 × main watchdog in each group

• One RTC timer

• RTC watchdog

Advanced Peripheral Interfaces

• 34 × programmable GPIOs

– 5 strapping GPIOs

– 6 input-only GPIOs

– 6 GPIOs needed for in-package flash/PSRAM (ESP32-D0WDR2-V3, ESP32-U4WDH)

• 12-bit SAR ADC up to 18 channels

• 2 × 8-bit DAC • 10 × touch sensors

• 4 × SPI • 2 × I2S • 2 × I2C • 3 × UART • 1 host (SD/eMMC/SDIO)

• 1 slave (SDIO/SPI)

• Ethernet MAC interface with dedicated DMA and IEEE 1588 support

• TWAI®, compatible with ISO 11898-1 (CAN Specification 2.0)

• RMT (TX/RX)

• Motor PWM

• LED PWM up to 16 channels

Power Management

• Fine-resolution power control through a selection of clock frequency, duty cycle, Wi-Fi operating modes, and individual power control of internal components

• Five power modes designed for typical scenarios: Active, Modem-sleep, Light-sleep, Deep-sleep, Hibernation • Power consumption in Deep-sleep mode is 10 µA

• Ultra-Low-Power (ULP) coprocessors • RTC memory remains powered on in Deep-sleep mode

Security

• Secure boot

• Flash encryption • 1024-bit OTP, up to 768-bit for customers

• Cryptographic hardware acceleration: – AES – Hash (SHA-2) – RSA – ECC – Random Number Generator (RNG)

Applications

With low power consumption, ESP32 is an ideal choice for IoT devices in the following areas:

• Smart Home
• Industrial Automation
• Health Care
• Consumer Electronics
• Smart Agriculture
• POS machines
• Service robot
• Audio Devices • Generic Low-power IoT Sensor Hubs
• Generic Low-power IoT Data Loggers
• Cameras for Video Streaming
• Speech Recognition
• Image Recognition
• SDIO Wi-Fi + Bluetooth Networking Card
• Touch and Proximity Sensing

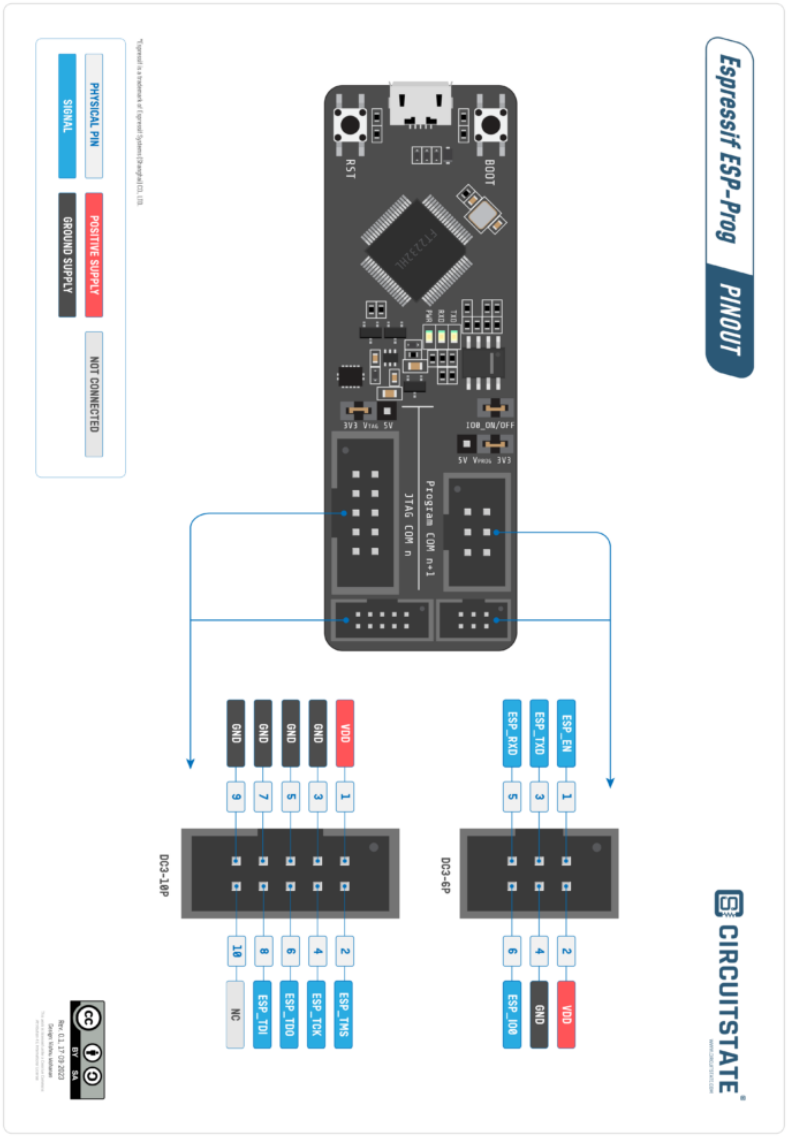
Referencias:

https://www.espressif.com/sites/default/files/documentation/esp32_hardware_design_guidelines_en.pdf

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

Anexo 2.- Puesta en marcha del ESP32 Prog JTAG

Introduction to the ESP-Prog Board



1. Overview

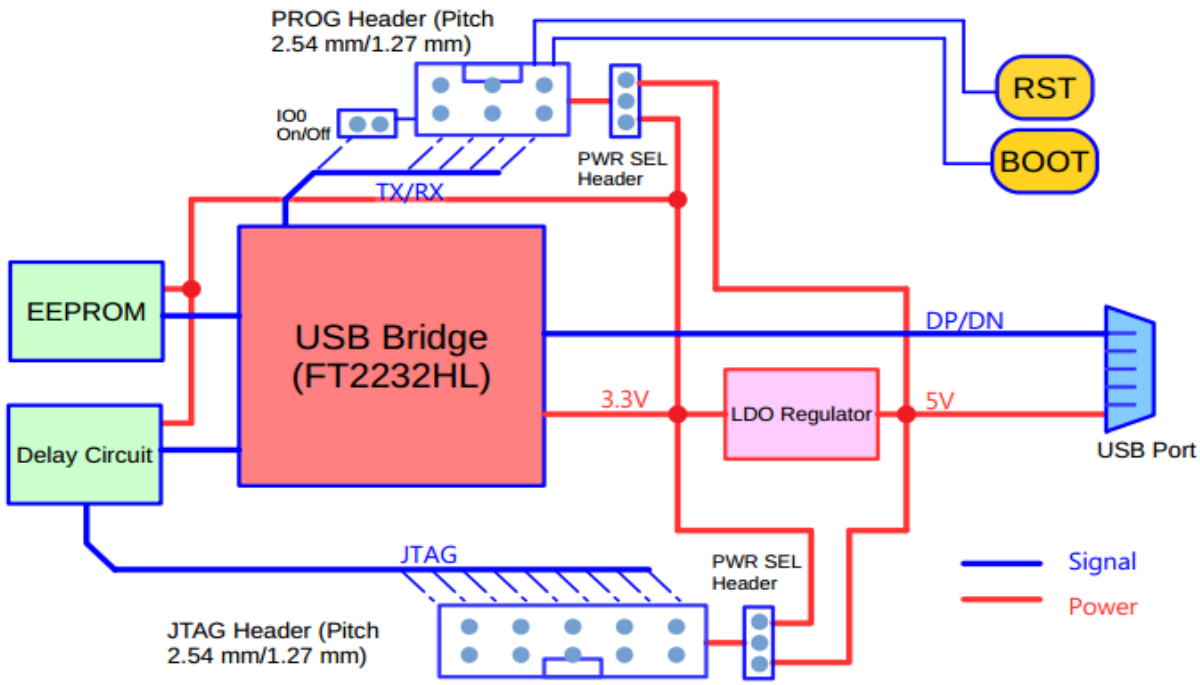
ESP-Prog is one of Espressif's development and debugging tools, with functions including automatic firmware downloading, serial communication, and JTAG online debugging. ESP-Prog's automatic firmware downloading and serial communication functions are supported on both the ESP8266 and ESP32 platforms, while the JTAG online debugging is supported only on the ESP32 platform.

ESP-Prog can easily connect to a PC with the use of only one USB cable. Then, the PC can identify the board's downloading and JTAG interfaces (functions) by their port numbers.

Given that the power supply voltage may vary on different user boards, either of the ESP-Prog interfaces can provide the 5V or the 3.3V power supply through pin headers, in order to ensure power compatibility.

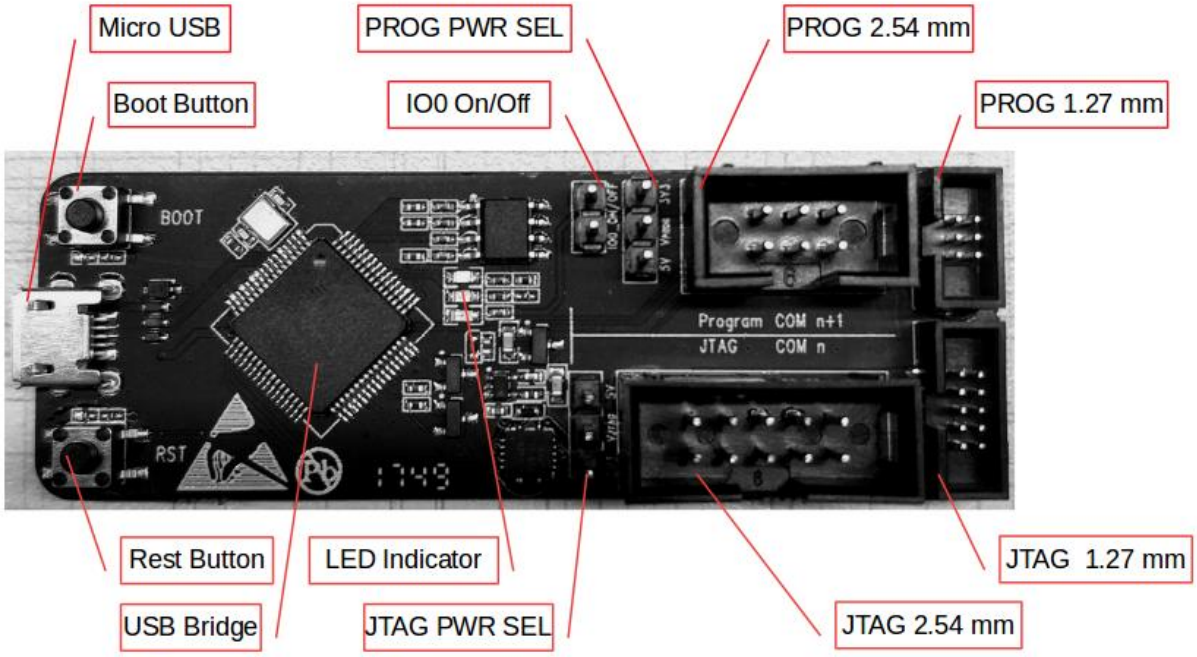
2. System Diagram

ESP-Prog's overall functional block diagram is displayed below:



3. Hardware Introduction

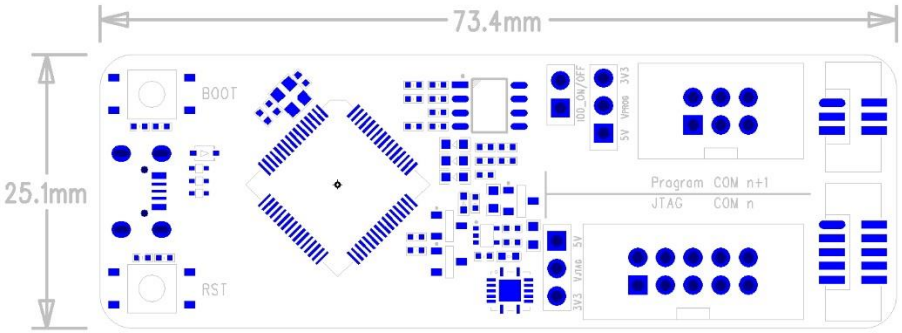
The figure below describes the areas of each function on the ESP-Prog board.



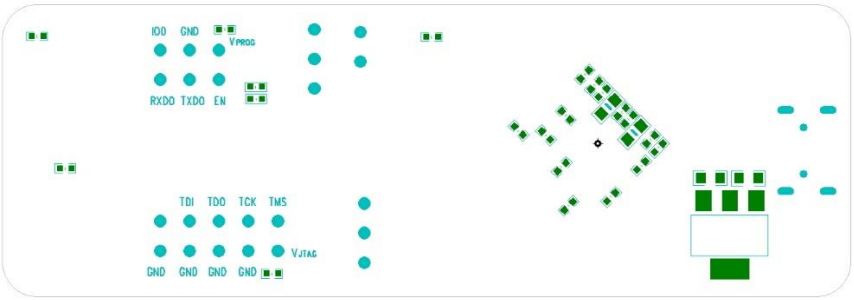
3.1. PCB Layout and Dimensions

ESP-Prog's PCB layout is displayed below, showing the board's dimensions and its interface markings. Please refer to Espressif website for the hardware resources including schematics, PCB reference design, BOM and other files.

- Top side



- Bottom side



3.2. Introduction to Functions

3.2.1. The Working Mode of USB Bridge

ESP-Prog uses FT2232HL, which is provided by FTDI, as its USB Bridge Controller chip. The board can be configured to convert the USB 2.0 interface to serial and parallel interfaces that support a wide range of industry standards. ESP-Prog uses FT2232HL's default dual-asynchronous serial interface mode, allowing users to use the board easily by installing the FT2232HLdriver on their PCs.

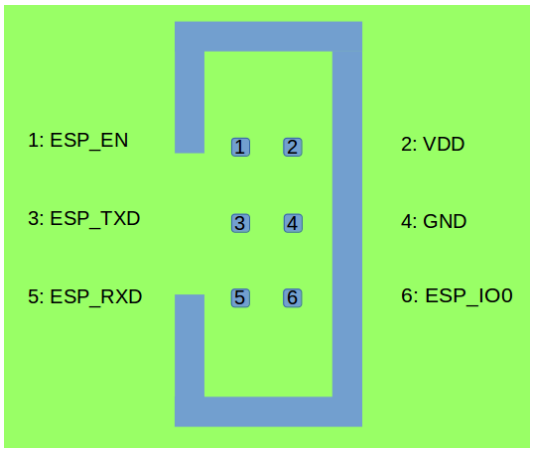
Note

The PC is able to identify the ESP-Prog's two ports by their port numbers. The bigger port number represents the Program interface, while the other one represents the JTAG interface.

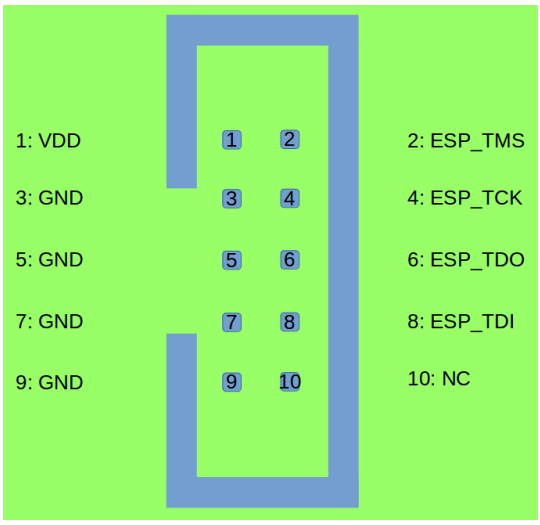
3.2.2. Communication Interface

ESP-Prog can connect to ESP32 user boards, using both the Program interface and the JTAG interface. Users should connect the user board interfaces in a way that corresponds to the ESP-Prog board.

- **Program Interface** The Program interface has six pins, including the UART interface (TXD, RXD), boot mode selection pin (ESPIO0) and reset pin (ESPEN). The design for the Program interface on the user board should follow the reference provided in the figure below.



- **JTAG Interface** The design for the JTAG interface on the user board should follow the reference provided in the figure below.



- **Fool-proof Design** The ESP-Prog board uses header connectors (DC3-6P / DC3-10P) which support reverse-current circuitry protection. In such cases, it is recommended that users also use header connectors on their user boards, such as `FTSH-105-01-S-DV-*` or `DC3-*P`.

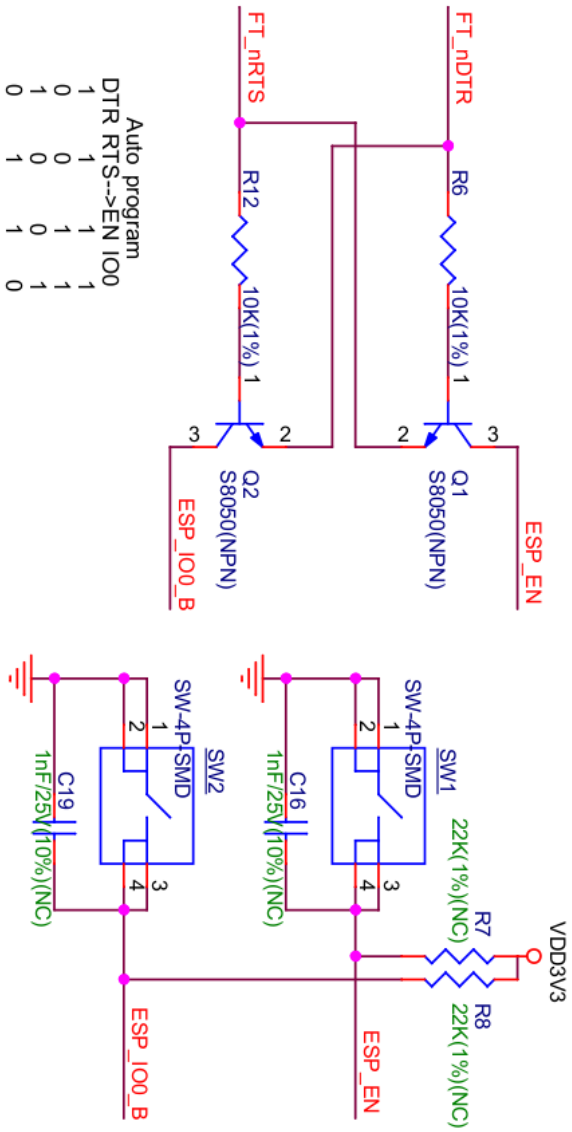
Note

The flat cables used here are directional. Please use the cables provided by Espressif.

3.2.3. Automatic Downloading Function

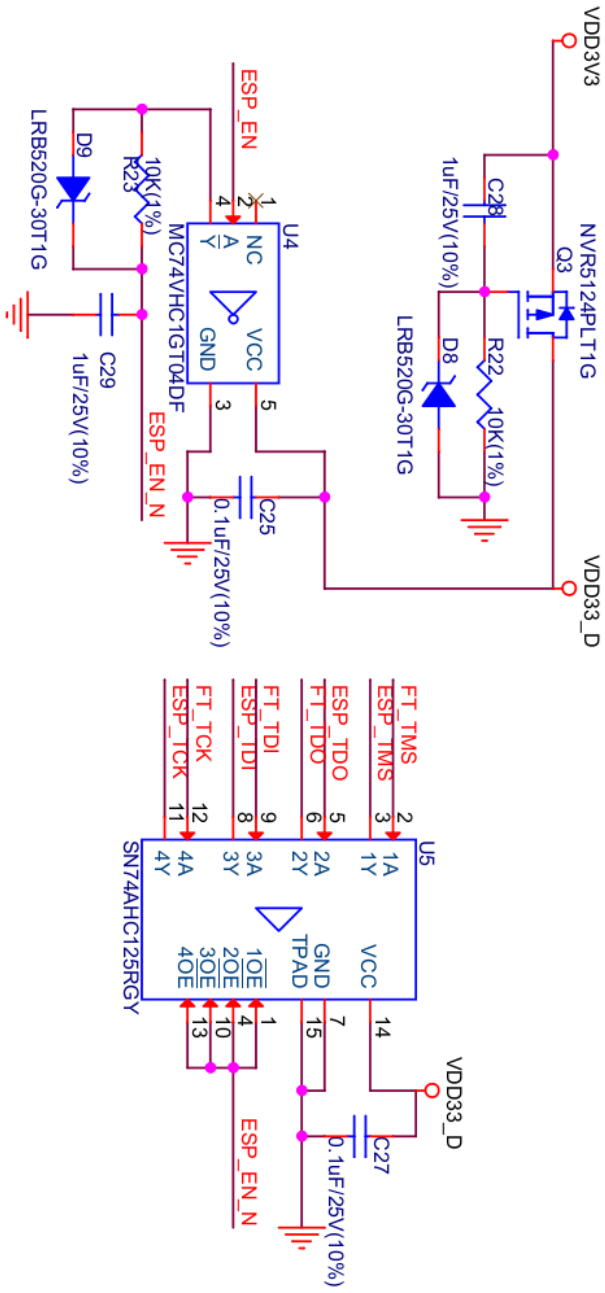
ESP-Prog supports automatic downloading. After connecting the Program interface of ESP-Prog to the user board, the downloading program can download data or run programs automatically by controlling the states of the start-mode selection pin (ESPIO0) and reset pin (ESPEN), which spares the users from manually restarting the device and selecting the downloading modes. The two buttons on the ESP-Prog board enable users to reset and control the boot mode of the device manually.

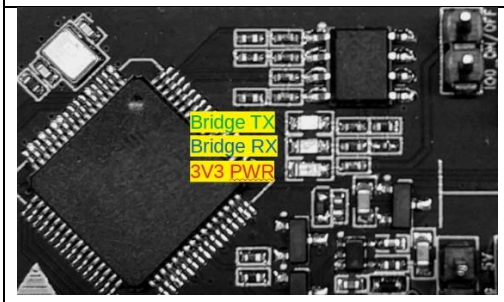
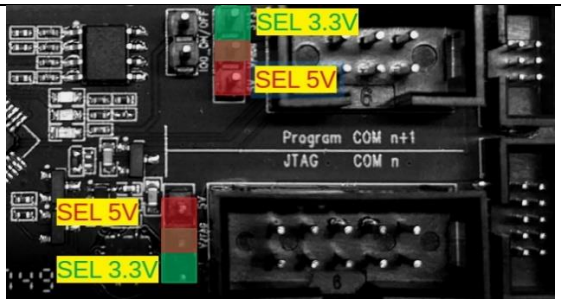
The schematics of the automatic downloading circuit is displayed below.



3.2.4. Delay Circuit

The delay circuit of ESP-Prog includes the bus buffer, inverter, MOS tube, first-order RC circuit, and other components. This delay circuit ensures that the ESP32 chip can power up or reset itself, before connecting with the JTAG signal, thus protecting the chip from the influence of JTAG on power-up or reset.



3.2.5. LED Status Indication	3.2.6. Pin Headers¶
• The red LED lights up when the system is connected to the 3.3V power; • The green LED lights up when ESP-Prog is downloading data to ESP32; • The blue LED lights up when ESP-Prog is receiving data from ESP32.	Users can choose either the 3.3V or 5V power supply for the Program and JTAG interfaces, using the pin headers shown in the figure below. • **Pin header to select power supply** The pin header in the middle is the power input pin for each interface. When this pin is connected to 5V, the power output of the interface is 5V. When this pin is connected to 3.3V, the power output of the interface is 3.3V. • **IO0 On/Off Pin** Pin IO0 can be set to select ESP8266's and ESP32's boot modes. This pin can be used as a common GPIO, after the chip is powered on. Users can then disconnect Pin IO0 manually to protect the operation of the user board from the influence of ESP-Prog's automatic downloading circuit.
	

4. Step by Step Instruction

1. Connect the ESP-Prog board and the PC USB port via a USB cable.
2. Install the FT2232HL chip driver on your PC. The PC then detects the two ports of ESP-Prog, indicating that the driver has been installed successfully.
3. Select the output power voltage for the Program / JTAG interfaces, using pin headers.
4. Connect the ESP-Prog and ESP user boards with the gray flat cables provided by Espressif.
5. Start automatic downloading or JTAG debugging, using the official software tools or scripts provided by Espressif.

5. Useful Links

- Espressif's Official Website
- **How to buy**: espressif_systems (WeChat Account), Purchase consulting
- ESP-Prog schematics, PCB reference design, BOM
- Introduction to the ESP32 JTAG Debugging
- Flash Download Tools (ESP8266 & ESP32)
- FT2232HL Chip Driver

Anexo 3.- OASIS Standard, 2019

Ref: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf

Introduction

1.0 Intellectual property rights policy

This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis7 open.org/committees/mqtt/ipr.php).

1.1 Organization of the MQTT specification

The specification is split into seven chapters:

- Chapter 1 - Introduction
- Chapter 2 - MQTT Control Packet format
- Chapter 3 - MQTT Control Packets
- Chapter 4 - Operational behavior
- Chapter 5 - Security
- Chapter 6 - Using WebSocket as a network transport
- Chapter 7 - Conformance Targets

1.2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD

NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as

described in IETF RFC 2119 [RFC2119], except where they appear in text that is marked as non22 normative.

Network Connection:

A construct provided by the underlying transport protocol that is being used by MQTT.

• It connects the Client to the Server.

• It provides the means to send an ordered, lossless, stream of bytes in both directions.

Refer to section 4.2 Network Connection for non-normative examples.

Application Message:

The data carried by the MQTT protocol across the network for the application. When an Application Message is transported by MQTT it contains payload data, a Quality of Service (QoS), a collection of Properties, and a Topic Name.

Client:

A program or device that uses MQTT. A Client:

• opens the Network Connection to the Server

• publishes Application Messages that other Clients might be interested in.

• subscribes to request Application Messages that it is interested in receiving.

• unsubscribes to remove a request for Application Messages.

• closes the Network Connection to the Server.

Server:

A program or device that acts as an intermediary between Clients which publish Application Messages and Clients which have made Subscriptions. A Server:

- accepts Network Connections from Clients.
- accepts Application Messages published by Clients.
- processes Subscribe and Unsubscribe requests from Clients.
- forwards Application Messages that match Client Subscriptions.
- closes the Network Connection from the Client.

Session:

A stateful interaction between a Client and a Server. Some Sessions last only as long as the Network

Connection, others can span multiple consecutive Network Connections between a Client and a Server.

Subscription:

A Subscription comprises a Topic Filter and a maximum QoS. A Subscription is associated with a single

Session. A Session can contain more than one Subscription. Each Subscription within a Session has a different Topic Filter.

Shared Subscription:

A Shared Subscription comprises a Topic Filter and a maximum QoS. A Shared Subscription can be associated with more than one Session to allow a wider range of message exchange patterns. An Application Message that matches a Shared Subscription is only sent to the Client associated with one of these Sessions. A Session can subscribe to more than one Shared Subscription and can contain both Shared Subscriptions and Subscriptions which are not shared.

Wildcard Subscription:

A Wildcard Subscription is a Subscription with a Topic Filter containing one or more wildcard characters.

This allows the subscription to match more than one Topic Name. Refer to section 4.7 for a description of wildcard characters in a Topic Filter.

Topic Name:

The label attached to an Application Message which is matched against the Subscriptions known to the Server.

Topic Filter:

An expression contained in a Subscription to indicate an interest in one or more topics. A Topic Filter can include wildcard characters.

MQTT Control Packet:

A packet of information that is sent across the Network Connection. The MQTT specification defines fifteen different types of MQTT Control Packet, for example the PUBLISH packet is used to convey Application Messages.

Malformed Packet:

A control packet that cannot be parsed according to this specification. Refer to section 4.13 for information about error handling.

Protocol Error:

An error that is detected after the packet has been parsed and found to contain data that is not allowed by the protocol or is inconsistent with the state of the Client or Server. Refer to section 4.13 for information about error handling.

Will Message:

An Application Message which is published by the Server after the Network Connection is closed in cases where the Network

Connection is not closed normally. Refer to section 3.1.2.5 for information about Will Messages.

Disallowed Unicode code point:

The set of Unicode Control Codes and Unicode Noncharacters which should not be included in a UTF-8 Encoded String. Refer to section 1.5.4 for more information about the Disallowed Unicode code points.

1.3 Normative references

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI

10.17487/RFC2119, March 1997,

http://www.rfc-editor.org/info/rfc2119

[RFC3629]

Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI

10.17487/RFC3629, November 2003,

http://www.rfc-editor.org/info/rfc3629

[RFC6455]

Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December

2011,

http://www.rfc-editor.org/info/rfc6455

[Unicode]

The Unicode Consortium. The Unicode Standard,

http://www.unicode.org/versions/latest/

1.4 Non-normative references

[RFC0793]

Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981,

http://www.rfc-editor.org/info/rfc793

[RFC5246]

Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI

10.17487/RFC5246, August 2008,

http://www.rfc-editor.org/info/rfc5246

[AES]

Advanced Encryption Standard (AES) (FIPS PUB 197).

https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf

[CHACHA20]

ChaCha20 and Poly1305 for IETF Protocols

https://tools.ietf.org/html/rfc7539

[FIPS1402]

Security Requirements for Cryptographic Modules (FIPS PUB 140-2)

https://csrc.nist.gov/csrc/media/publications/fips/140/2/final/documents/fips1402.pdf

[IEEE 802.1AR]

IEEE Standard for Local and metropolitan area networks - Secure Device Identity

http://standards.ieee.org/findstds/standard/802.1AR-2009.html

[ISO29192]

ISO/IEC 29192-1:2012 Information technology -- Security techniques -- Lightweight cryptography -- Part

1: General

https://www.iso.org/standard/56425.html

[MQTT NIST]

MQTT supplemental publication, MQTT and the NIST Framework for Improving Critical Infrastructure

Cybersecurity

http://docs.oasis-open.org/mqtt/mqtt-nist-cybersecurity/v1.0/mqtt-nist-cybersecurity-v1.0.html

[MQTTV311]

MQTT V3.1.1 Protocol Specification

http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[ISO20922]

MQTT V3.1.1 ISO Standard (ISO/IEC 20922:2016)

https://www.iso.org/standard/69466.html

[NISTCSF]

Improving Critical Infrastructure Cybersecurity Executive Order 13636

https://www.nist.gov/sites/default/files/documents/itl/preliminary-cybersecurity-framework.pdf

[NIST7628]

NISTIR 7628 Guidelines for Smart Grid Cyber Security Catalogue

https://www.nist.gov/sites/default/files/documents/smartgrid/nistir-7628_total.pdf

[NSAB]

NSA Suite B Cryptography

http://www.nsa.gov/ia/programs/suiteb_cryptography/

[PCIDSS]

PCI-DSS Payment Card Industry Data Security Standard

https://www.pcisecuritystandards.org/pci_security/

[RFC1928]

Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D., and L. Jones, "SOCKS Protocol Version 5", RFC

1928, DOI 10.17487/RFC1928, March 1996,

http://www.rfc-editor.org/info/rfc1928

[RFC4511]

Sermersheim, J., Ed., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, DOI

10.17487/RFC4511, June 2006,

http://www.rfc-editor.org/info/rfc4511

[RFC5280]

Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key

Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI

10.17487/RFC5280, May 2008,

http://www.rfc-editor.org/info/rfc5280

[RFC6066]

Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI

10.17487/RFC6066, January 2011,

http://www.rfc-editor.org/info/rfc6066

[RFC6749]

Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October

2012,

http://www.rfc-editor.org/info/rfc6749

[RFC6960]

Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and C. Adams, "X.509 Internet Public

Key Infrastructure Online Certificate Status Protocol - OCSP", RFC 6960, DOI 10.17487/RFC6960, June

2013,

http://www.rfc-editor.org/info/rfc6960

[SARBANES]

Sarbanes-Oxley Act of 2002.

http://www.gpo.gov/fdsys/pkg/PLAW-107publ204/html/PLAW-107publ204.htm

[USEUPRIVSH]

U.S.-EU Privacy Shield Framework

https://www.privacyshield.gov

[RFC3986]

Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD

66, RFC 3986, DOI 10.17487/RFC3986, January 2005,

http://www.rfc-editor.org/info/rfc3986

[RFC1035]

Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI

10.17487/RFC1035, November 1987,

http://www.rfc-editor.org/info/rfc1035

[RFC2782]

Gulbrandsen, A., Vixie, P., and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)",

RFC 2782, DOI 10.17487/RFC2782, February 2000,

http://www.rfc-editor.org/info/rfc2782

1.5 Data representation

1.5.1 Bits

Bits in a byte are labelled 7 to 0. Bit number 7 is the most significant bit, the least significant bit is assigned bit number 0.

1.5.2 Two Byte Integer

Two Byte Integer data values are 16-bit unsigned integers in big-endian order: the high order byte precedes the lower order byte. This means that a 16-bit word is presented on the network as Most Significant Byte (MSB), followed by Least Significant Byte (LSB).

1.5.3 Four Byte Integer

Four Byte Integer data values are 32-bit unsigned integers in big-endian order: the high order byte precedes the successively lower order bytes. This means that a 32-bit word is presented on the network as Most Significant Byte (MSB), followed by the next most Significant Byte (MSB), followed by the next most Significant Byte (MSB), followed by Least Significant Byte (LSB).

1.5.4 UTF-8 Encoded String

Text fields within the MQTT Control Packets described later are encoded as UTF-8 strings. UTF-8 [**RFC3629**] is an efficient encoding of Unicode [Unicode] characters that optimizes the encoding of ASCII characters in support of text-based communications.

Each of these strings is prefixed with a Two Byte Integer length field that gives the number of bytes in a UTF-8 encoded string itself, as illustrated in Figure 1.1 Structure of UTF-8 Encoded Strings below. Consequently, the maximum size of a UTF-8 Encoded String is 65,535 bytes.

Unless stated otherwise all UTF-8 encoded strings can have any length in the range 0 to 65,535 bytes.

Figure 1-1 Structure of UTF-8 Encoded Strings

Bit	7	6	5	4	3	2	1	0
Byte 1	String length MSB							
Byte 2	String length LSB							
Byte 3 …	UTF-8 encoded character data, if length > 0.							

The character data in a UTF-8 Encoded String MUST be well-formed UTF-8 as defined by the Unicode specification [Unicode] and restated in RFC 3629 [RFC3629]. In particular, the character data MUST NOT include encodings of code points between U+D800 and U+DFFF [MQTT-1.5.4-1]. If the Client or Server receives an MQTT Control Packet containing ill-formed UTF-8 it is a Malformed Packet. Refer to section 4.13 for information about handling errors.

A UTF-8 Encoded String MUST NOT include an encoding of the null character U+0000. [MQTT-1.5.4-2].If a receiver (Server or Client) receives an MQTT Control Packet containing U+0000 it is a Malformed Packet. Refer to section 4.13 for information about handling errors.

The data SHOULD NOT include encodings of the Unicode [Unicode] code points listed below. If areceiver (Server or Client) receives an MQTT Control Packet containing any of them it MAY treat it as a Malformed Packet. These are the Disallowed Unicode code points. Refer to section 5.4.9 for more information about handling Disallowed Unicode code points.

- U+0001..U+001F control characters
- U+007F..U+009F control characters
- Code points defined in the Unicode specification [Unicode] to be non-characters (for example U+0FFFF)

A UTF-8 encoded sequence 0xEF 0xBB 0xBF is always interpreted as U+FEFF ("ZERO WIDTH NO289 BREAK SPACE") wherever it appears in a string and MUST NOT be skipped over or stripped off by a packet receiver [MQTT-1.5.4-3].

Non-normative example

For example, the string A鱳 which is LATIN CAPITAL Letter A followed by the code point U+2A6D4(which represents a CJK IDEOGRAPH EXTENSION B character) is encoded as follows:

Figure 1-2 UTF-8 Encoded String non-normative example

byte 2	String Length LSB (0x05)							
	0	0	0	0	0	1	0	1
byte 3	'A' (0x41)							
	0	1	0	0	0	0	0	1
byte 4	(0xF0)							
	1	1	1	1	0	0	0	0
byte 5	(0xAA)							
	1	0	1	0	1	0	1	0
byte 6	(0x9B)							
	1	0	0	1	1	0	1	1
byte 7	(0x94)							
	1	0	0	1	0	1	0	0

1.5.5 Variable Byte Integer

The Variable Byte Integer is encoded using an encoding scheme which uses a single byte for values up to 127. Larger values are handled as follows. The least significant seven bits of each byte encode the data, and the most significant bit is used to indicate whether there are bytes following in the representation. Thus, each byte encodes 128 values and a "continuation bit". The maximum number of bytes in the Variable Byte Integer field is four. The encoded value MUST use the

minimum number of bytes necessary to represent the value [MQTT-1.5.5-1]. This is shown in Table 1-1 Size of Variable Byte Integer.

Table 1-1 Size of Variable Byte Integer

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16,383 (0xFF, 0x7F)
3	16,384 (0x80, 0x80, 0x01)	2,097,151 (0xFF, 0xFF, 0x7F)
4	2,097,152 (0x80, 0x80, 0x80, 0x01)	268,435,455 (0xFF, 0xFF, 0xFF, 0x7F)

Non-normative comment

The algorithm for encoding a non-negative integer (X) into the Variable Byte Integer encoding

scheme is as follows:

```
do
encodedByte = X MOD 128
X = X DIV 128
// if there are more data to encode, set the top bit of this byte
if (X > 0)
encodedByte = encodedByte OR 128
endif
'output' encodedByte
while (X > 0)
```

Where MOD is the modulo operator (% in C), DIV is integer division (/ in C), and OR is bit-wise or (| in C).

Non-normative comment

The algorithm for decoding a Variable Byte Integer type is as follows:

```
multiplier = 1
value = 0
do
encodedByte = 'next byte from stream'
value += (encodedByte AND 127) * multiplier
if (multiplier > 128*128*128)
throw Error(Malformed Variable Byte Integer)
multiplier *= 128
while ((encodedByte AND 128) != 0)
```

where AND is the bit-wise and operator (& in C).

When this algorithm terminates, value contains the Variable Byte Integer value.

1.5.6 Binary Data

Binary Data is represented by a Two Byte Integer length which indicates the number of data bytes, followed by that number of bytes. Thus, the length of Binary Data is limited to the range of 0 to 65,535 Bytes.

1.5.7 UTF-8 String Pair

A UTF-8 String Pair consists of two UTF-8 Encoded Strings. This data type is used to hold name-value pairs. The first string serves as the name, and the second string contains the value.

Both strings MUST comply with the requirements for UTF-8 Encoded Strings [MQTT-1.5.7-1]. If a receiver

(Client or Server) receives a string pair which does not meet these requirements it is a Malformed Packet.

Refer to section 4.13 for information about handling errors.

1.6 Security

MQTT Client and Server implementations SHOULD offer Authentication, Authorization and secure communication options, such as those discussed in Chapter 5. Applications concerned with critical infrastructure, personally identifiable information, or other personal or sensitive information are strongly advised to use these security capabilities.

1.7 Editing convention

Text highlighted in Yellow within this specification identifies conformance statements. Each conformance statement has been assigned a reference in the format [MQTT-x.x.x-y] where x.x.x is the section number and y is a statement counter within the section.

1.8 Change history

1.8.1 MQTT v3.1.1

MQTT v3.1.1 was the first OASIS standard version of MQTT [MQTTV311].

MQTT v3.1.1 is also standardized as ISO/IEC 20922:2016 [ISO20922].

1.8.2 MQTT v5.0

MQTT v5.0 adds a significant number of new features to MQTT while keeping much of the core in place.

The major functional objectives are:

• Enhancements for scalability and large scale systems

- Improved error reporting
- Formalize common patterns including capability discovery and request response
- Extensibility mechanisms including user properties
- Performance improvements and support for small clients

Refer to Appendix C for a summary of changes in MQTT v5.0.

Anexo 4.- Ejemplo “Blink_AnalogRead.ino” de FreeRTOS en el IDE de Arduino

```
#include <Arduino_FreeRTOS.h>

// define two tasks for Blink & AnalogRead
void TaskBlink( void *pvParameters );
void TaskAnalogRead( void *pvParameters );

// the setup function runs once when you press reset or power
the board
void setup() {

  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);

  while (!Serial) {
    ; // wait for serial port to connect. Needed for native
USB, on LEONARDO, MICRO, YUN, and other 32u4 based boards.
  }

  // Now set up two tasks to run independently.
  xTaskCreate(
    TaskBlink
    ,  "Blink"   // A name just for humans
    ,  128  // This stack size can be checked & adjusted by
reading the Stack Highwater
    ,  NULL
    ,  2  // Priority, with 3 (configMAX_PRIORITIES - 1) being
the highest, and 0 being the lowest.
    ,  NULL );

  xTaskCreate(
    TaskAnalogRead
    ,  "AnalogRead"
    ,  128  // Stack size
    ,  NULL
    ,  1  // Priority
    ,  NULL );

  // Now the task scheduler, which takes over control of
scheduling individual tasks, is automatically started.
}

void loop()
{
  // Empty. Things are done in Tasks.
}

/*--------------------------------------------------*/
```

```
46 /*---------------------- Tasks ---------------------*/
47 /*--------------------------------------------------*/
48
49 void TaskBlink(void *pvParameters)  // This is a task.
50 {
51   (void) pvParameters;
52
53 /*
54   Blink
55   Turns on an LED on for one second, then off for one second,
56 repeatedly.
57
58   Most Arduinos have an on-board LED you can control. On the
59 UNO, LEONARDO, MEGA, and ZERO
60   it is attached to digital pin 13, on MKR1000 on pin 6.
61 LED_BUILTIN takes care
62   of use the correct LED pin whatever is the board used.
63
64   The MICRO does not have a LED_BUILTIN available. For the
65 MICRO board please substitute
66   the LED_BUILTIN definition with either LED_BUILTIN_RX or
67 LED_BUILTIN_TX.
68   e.g. pinMode(LED_BUILTIN_RX, OUTPUT); etc.
69
70   If you want to know what pin the on-board LED is connected to
71 on your Arduino model, check
72   the Technical Specs of your board  at
73 https://www.arduino.cc/en/Main/Products
74
75   This example code is in the public domain.
76
77   modified 8 May 2014
78   by Scott Fitzgerald
79
80   modified 2 Sep 2016
81   by Arturo Guadalupi
82 */
83
84   // initialize digital LED_BUILTIN on pin 13 as an output.
85   pinMode(LED_BUILTIN, OUTPUT);
86
87   for (;;) // A Task shall never return or exit.
88   {
89     digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH
90 is the voltage level)
91     vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one
92 second
93     digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by
94 making the voltage LOW
95     vTaskDelay( 1000 / portTICK_PERIOD_MS ); // wait for one
96 second
   }
}
```

```
 97
 98 void TaskAnalogRead(void *pvParameters)  // This is a task.
 09 {
101   (void) pvParameters;
102
103 /*
104   AnalogReadSerial
105   Reads an analog input on pin 0, prints the result to the
106 serial monitor.
107   Graphical representation is available using serial plotter
108 (Tools > Serial Plotter menu)
109   Attach the center pin of a potentiometer to pin A0, and the
110 outside pins to +5V and ground.
112
113   This example code is in the public domain.
114 */
115
116   for (;;)
117   {
118     // read the input on analog pin 0:
119     int sensorValue = analogRead(A0);
120     // print out the value you read:
121     Serial.println(sensorValue);
122     vTaskDelay(1);  // one tick delay (15ms) in between reads
for stability
   }
}
```

Anexo 5.- Artículo “El patrón de eventos “Handler" en Arduino para la implementación de un protocolo simple de comunicación basado en eventos”

Revista/Congreso: IEEE, RVP-AI/ROC&C 2022, ISBN: 978-607-95630-8-0

El patrón de eventos “Handler" en Arduino para la implementación de un protocolo simple de comunicación basado en eventos

Ing. Oscar Beltrán Gómez

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. de las Industrias 11101, Complejo Industrial Chihuahua, 31130 Chihuahua, Chih., Mex.
Tels. (614) 442-50-00
oscar.bego@chihuahua2.tecnm.mx

Dr. Rafael Sandoval Rodríguez

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. Tecnologico 2909, Tecnológico, 31200 Chihuahua, Chih. Mex.
Tels. (614) 201-20-00
rafael.saro@chihuahua2.tecnm.mx

M.I. Arturo Legarda Sáenz

Tecnológico Nacional de México/I.T. Chihuahua 2, DEPI.
Av. de las Industrias 11101, Complejo Industrial Chihuahua, 31130 Chihuahua, Chih., Mex.
Tels. (614) 442-50-00
arturo.ls@chihuahua2.tecnm.mx

Resumen

Cada vez es más frecuente ver sistemas que interactúan con algún aspecto físico y disponer de la información que estos generan o requieren puede representar un reto por la gran variedad de tecnologías tanto en “computación física” como en la “computación tradicional”.
La programación basada en eventos puede representar un punto de convergencia al despreocuparse de cómo los componentes se comportan de manera individual para lograr un comportamiento propagado por eventos.

Este artículo trata del patrón de eventos “Handler” en el desarrollo de una librería ejecutable en la placa Arduino para la implementación de un protocolo simple de comunicación basado en eventos.

1. Introducción

Cada vez es más frecuente ver sistemas que interactúan con algún aspecto físico y aunque contamos con conceptos como el “Internet de las cosas”, disponer de la información que estos elementos físicos generan o proporcionar las que estos requieren puede representar un verdadero reto para los desarrolladores.

La dificultad radica en la gran variedad de dialectos, paradigmas y tecnologías que soportan una mayor cantidad de plataformas, es decir, la implementación de sistemas que se puedan adaptar a tal cantidad de variantes es una tarea compleja; también influye que la información rara vez queda aislada, es decir, normalmente ocupamos la información “distribuida” en: bases de datos, aplicaciones de escritorio y móviles,

páginas web y más recientemente en aplicaciones implementadas en lo que se conoce como computación física o IoT.

"La computación física implica el diseño de objetos interactivos que se pueden comunicar con las personas usando sensores y actuadores controlados por un comportamiento implementado en software que se ejecuta dentro de un microcontrolador" [1].

Por otra parte, y para efectos de este artículo, la "computación tradicional", la que de forma indirecta interactúa con el hardware que la contiene, normalmente a través de algún intermediario o sistema operativo; este "computo más tradicional" ha evolucionado para el acceso intensivo de la información, mientras que "la computación física" aún está en ese camino.

En consecuencia, los desarrolladores que implementan aplicaciones de cómputo físico necesitan adentrarse en áreas como redes de computadoras y electrónica para atender o propagar la información que se presenta en forma de suceso de la contraparte física del sistema y lograr una arquitectura homogénea.

La programación basada en eventos o sucesos puede representar un punto de convergencia, para el desarrollo de sistemas que interactúen con aspectos físicos y la ya mencionada programación tradicional (CT), al proveer de:

"un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema o que ellos mismos provoquen" [2].

La propuesta de una arquitectura basada en eventos distribuidos es despreocuparse de cómo los componentes, físicos e intangibles, de un sistema se comportan de manera individual para lograr un comportamiento propagado por eventos.

2. Objetivo

En este artículo se aborda el patrón "Handler" para la creación de una librería para la placa de desarrollo Arduino [3] en su variante "Handler sin Cabeza" para la implementación de un protocolo simple de comunicación basado en eventos entre ya mencionadas aplicaciones de CT y las desarrolladas a su vez en "computación física".

Esta arquitectura considera como puntos fundamentales:

- · Un desarrollo dirigido por eventos. · Bajo acoplamiento.
- · Un enfoque de propagación de eventos en red (sockets).

3. Tecnologías utilizadas para la demostración

En la demostración e implantación de la arquitectura se utilizan las siguientes tecnologías:

- · Arduino UNO R3 [4].
- · Lenguaje de programación C++ [5].

4. El Patrón "Handler" para el desarrollo dirigido por eventos

El principal patrón en el desarrollo dirigido por eventos es el llamado "Handler" [6]. El patrón "Handler" (ver figura. 1), define un flujo de eventos, un despachador y un conjunto de manejadores.

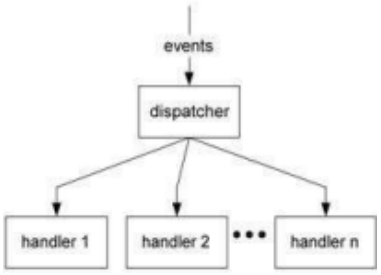


Figura 1. Patrón Handler.

Podríamos decir que el despachador es quien rutea los eventos a uno o varios manejadores (Handler's), los cuales realizan alguna función de acuerdo al tipo de evento; esta lógica de negocio incluye un ciclo para seguir atendiendo los eventos entrantes.

De las diferentes variantes del patrón "Handler" son: "Handler Extendido", "Handler con Colas de Eventos" y el "Handler sin Cabeza" [7], figura 2; este último se ajusta más al modelo que en este artículo se describe.

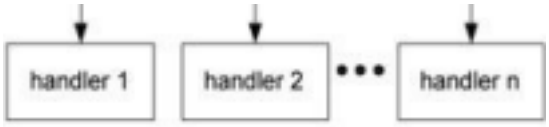


Figura 2. Patrón Handler sin Cabeza.

El "Handler sin Cabeza", ó en inglés "Headless Handlers Pattern" [7], oculta al despachador, dejando únicamente visible la lista de manejadores.

Los sistemas basados en cómputo físico pueden tratar flujos de eventos infinitos, sin embargo, en la mayoría de los sistemas basados en eventos el flujo de eventos es finito. La lógica de la aplicación debe de contar con la capacidad de salir del ciclo de eventos ó bien ingresar al mismo.

```
1:  int socketClientEvent::on (char
2:  dato, void (*callb)())
3:  {
4:    Elemento *nuevo_elemento;
5:
6:    if ( (nuevo_elemento =
7:   (Elemento *)
8:   malloc (sizeof (Elemento))) ==
9:  NULL
10:  )
11:   return -1;
12:
13:   nuevo_elemento->dato = dato;
14:   nuevo_elemento->callb =
15:  callb;
16:
17:   nuevo_elemento->siguiente =
18:  lista->inicio;
19:   lista->inicio =
20:  nuevo_elemento;
21:
22:   return 0;
     }
```

Figura 3. Método "on".

5. Implementación de la librería "socketClientEvent"

La librería "*socketClientEvent*" [8] consta de tres mecanismos principales: el primero es el método llamado "*on*", fig. 3, este método asigna una correspondencia de un evento a una función específica o manejador definiendo una ruta. Este método, de forma interna, agrega a una "*Lista Enlazada*" el par evento-manejador.

```
1:  void
2:  socketClientEvent::disparar
3:   (char c) {
4:
5:  Elemento *actual;
6:  actual = lista->inicio;
7:
8:   while (actual != NULL) {
9:   if(actual->dato == c )
10:  actual->callb();
11:
12:  actual = actual-
13  >siguiente;  }
     }
```

Figura 4. Método "disparar".

El evento está definido, en este caso y por facilidad, como un carácter o

"char", mientras que el manejador está representado por un apuntador a función.

El segundo mecanismo, fig. 4, es el disparador. Cuando es llamado este método ocupa el paso de un parámetro tipo "*char"*, correspondiente al evento, para buscarlo en la "*Lista Enlazada"* y posterior mente llamar a la función que apunta.

```
1:  void
2:
3:  socketClientEvent::listener()
4:
5:  {  _cliente = _server-
6:
7:  >available();  if (_cliente) {
8:
9:   while    (_cliente.connected())
10:
11: {        char       data       =
12:
13: _cliente.read();   disparar(data
14:
15: );
16:
17:  switch(data) {
18:
19:  case 'x':
20:  _cliente.print("adiós :)  ");
21:  _cliente.stop();
22:  break;
23:
24:  default:
25:  break;
26:  }
27:  }
     }

     _cliente.stop();
    }
```

Figura 5. Método "listener".

El tercer mecanismo, fig. 5, representa al despachador; este método, que continuamente está en espera de clientes (clientes sockets) rutea sus peticiones a algún método a disparar con los datos que recibe de la misma petición.

En código de la figura 6, líneas 9, 10 y 11, muestra como de forma desacoplada se definen los manejadores. Este tipo de ensamblaje es apropiado ya que el mismo evento puede estar ligado a más de un manejador, líneas 25 y 28.

Hay que recordar que por la forma en que está programada la *"Lista Enlazada"* los eventos desencadenarán los manejadores en el orden en que fueron ingresados con el método "*on*". Lo anterior en las líneas de la 25 a la 28.

```
1:   #include <SPI.h>
2:   #include <Ethernet.h>
3:   #include <socketClientEvent.h>
4:
5:  byte mac[] = { 0xDE, 0xED,
6:  0xBA, 0xFE, 0xFE, 0xED };
7:  byte ip[] = { 192, 168, 2, 2 };
8:
9:  void f() {
10:  Serial.println("se llamó a
11: f");  }
12: void f1() {
13:  Serial.println("se llamó a
14: f1");  }
15: void f2() {
16:  Serial.println("se llamó a
17: f2");  }
18:
19: EthernetServer server(23);
20: socketClientEvent sc =
21: socketClientEvent(server);
22:
23: void setup() {
24:
25:  Serial.begin(9600);
26:  Ethernet.begin(mac, ip);
27:
28:  sc.on('a', f);
29:  sc.on('1', f1);
30:  sc.on('2', f2);
31:  sc.on('c', f);
32:
33:  //podemos hacer algo antes
34: de   //cerrar
35:  sc.on('x', f1);
36: }
37:
38: void loop() {
39:
40:  sc.listener();
41: }
```

Figura 6. Código para la placa Arduino.

La librería se ajusta al patrón "*Handler sin Cabeza*" ya que de forma natural oculta la figura del despachador dejando muy a la vista la colección de manejadores.

6. Conclusiones

La arquitectura "*Handler*" de eventos es claramente una forma de la implementación del patrón cliente-servidor ya que el servidor funciona como un despachador de eventos que espera a la escucha de una o varias peticiones de clientes; estas solicitudes son tratadas como eventos que son reenviados a uno o varios manejadores para posteriormente devolver un resultado.

Aunque los sistemas basados en eventos suelen ser multihilos, "para atender eventos en paralelo", "los sistemas como el Arduino solo cuentan con un procesador, esta es una consideración a tomar en cuenta".

La una arquitectura basada en eventos podría tomar encueta no solo los eventos que se provengan de la red, sino según el caso, el despachador de eventos podría rutear más de un solo tipo de sucesos, por ejemplo, sucesos como la presión de un botón o el cambio de estado de un sensor.

Algunos patrones de diseño de comportamiento como el "*Observer*" [9], el "*Visitor*"[10] y "*Reactor*" [11] roporcionan, también, una pauta en la mayoría de los proyectos basados en eventos.

Ahora el mercado informático ofrece y cuenta con muchas herramientas enfocadas a la integración de aplicaciones a partir del paso de mensajes, que habitualmente suelen representar eventos [12, 13, 14, 15, 16, 17].
La librería denominada "*socketClientEvent*" es solo una demostración del patrón "*Handler sin Cabeza*" en la placa Arduino; se recomienda su uso solo para fines didácticos y demostrativos.

7. Referencias

[1] Introducción a Arduino, Massimo Banzi, Mayo 2012, O'REILLY/ANAYA, I.S.B.N.: 978- 84-415-3177-2.
[2]Sarmiento J, Díaz de León J, Chimal J. TESIS: UN PARADIGMA PROACTIVO ORIENTADO A OBJETOS, INSTITUTO POLITÉCNICO NACIONAL, CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN, México D.F. a Junio 2009,

en:
http://www.cic.ipn.mx/sitioCIC/images/sources/ci c/tesis/A050821.pdf.
[3] Arduino - Libraries, Home Page: http://arduino.cc/en/pmwiki.php?n=Reference/Lib raries
[4] Arduino - ArduinoBoardUno, Home Page:
http://arduino.cc/en/Main/arduinoBoardUno [5] Cómo programar en C++ - Harvey M. Deitel, Paul J. Deitel Pearson Educación, 2003. [6] Programación en timepo real y bases de datos: un enfoque práctico, Josefina López Herrera, Universitat Politécnica de Cataluña, pag. 36.
[7] Event-Driven Programming: Introduction, Tutorial, History. Home Page: http://www.cnblogs.com/alex tech/archive/2011/10/27/2227058.html.
[8] SocketClientEvent - Library, Repository:
https://github.com/oscarbego/socketClientEvent. [9] E. Gamma, et al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley 1995, ISBN 0-201-63361-2. [10] The essence of the Visitor pattern, Computer Software and Applications Conference, 1998.

COMPSAC '98. Proceedings. The Twenty Second Annual International, Aug 1998, pp 9 - 15. [11] Douglas C. Schmidt, Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching, 1995. [12] Apache Camel, Home Page: http://camel.apache.org/, Julio 2012.
[13] FuseSource Integration Everywhere, Home Page: http://fusesource.com/, Julio 2012. [14] RabbitMQ: Spring Source, Home Page: http://www.rabbitmq.com/, Julio 2012. [15] MuleSoft, Home Page: http://www.mulesoft.org/, Julio 2012.
[16] Apache ServiceMix, Home Page: http://servicemix.apache.org/index.html, Julio 2012.
[17] Spring Integration, Home Page: http://www.springsource.org/spring-integration/, Julio 2012.

Anexo 6.- Artículo “DESARROLLO DE UNA LIBRERÍA CONCURRENTE EN FREERTOS EN POO”

Revista/Congreso: ELECTRO 2023, ISSN 1405-2172

DESARROLLO DE UNA LIBRERÍA CONCURRENTE EN FREERTOS EN POO

Beltrán-Gómez Oscar, Márquez-Gutiérrez Pedro-Rafael, Ruiz-Varela Oscar-Ramsés, Sandoval-Rodríguez Rafael, Trujillo-Preciado Edgar

Tecnológico Nacional de México / Instituto Tecnológico de Chihuahua

Ave. Tecnológico #2909, Chihuahua, Chih. México. C.P. 31310 Tel.. +52 (614) 2-01-2000

oscar.bg@chihuahua.tecnm.mx, pedro.mg@chihuahua.tecnm.mx, oscar.rv@chihuahua.tecnm.mx, rafael.s@chihuahua.tecnm.mx, edgar.tp@chihuahua.tecnm.mx

RESUMEN.

La percepción generalizada de las tecnologías de información ha generado una búsqueda de respuestas instantáneas independientes del dispositivo o de su tecnología. Esta solicitud exhaustiva de recursos computacionales ha obligado a la pronta evolución de las tecnologías y sistemas embebidos. Para solventar la demanda del alto nivel computacional fue necesario el desarrollo del Multiprocessor System-on-Chips (MPSoC). Este avance, incluye sus propios retos como la brecha en el uso de lenguajes y técnicas modernas de programación.Sin embargo, la luz al final del camino es la aparición y el uso de implementaciones modernas como FreeRTOS para la implementa concurrencia y paralelismo en varias marcas y modelos de MPSoC’s y SoC’s, pero aunque FreeRTOS es un excelente framework aún se nota la separación entre las implementaciones de alto nivel como las que proporciona la Programación Orientada a Objetos. En este artículo se desarrolla una librería que incorpora un nivel de abstracción al Framework FreeRTOS haciendo uso de la POO para el desarrollo de sistemas concurrentes y paralelos en ambientes embebidos.

Palabras Clave: Actores, FreeRTOS, librería, POO, C++.

ABSTRACT.

The widespread perception of information technologies has generated a search for instant answers independent of the device or its technology. This exhaustive request for computational resources has forced the rapid evolution of embedded technologies and systems. To meet the demand for a high computational level, it was necessary to develop the Multiprocessor System-on-Chips (MPSoC). This advance includes its own challenges such as the gap in the use of modern programming languages and techniques. However, the light at the end of the road is the emergence and use of modern implementations such as FreeRTOS for implementing concurrency and parallelism in various brands and models of MPSoC's and SoC's, but although FreeRTOS is an excellent framework, the separation between the high-level implementations such as those provided by Object Oriented Programming.

Keywords: Actors, FreeRTOS, library, OOP, C++.

1. INTRODUCCIÓN.

Las fronteras tenues entre las tecnologías de información suponen una percepción

genérica de las mismas; hoy se solicita la ejecución de varias tareas y se solicita la respuesta al instante sin importar el tipo de dispositivo su tamaño o capacidad.

Esta solicitud exhaustiva de recursos computacionales ha obligado a la pronta evolución de tecnologías en donde los sistemas embebidos juegan un papel importante.

Usualmente los sistemas embebidos se basan en **SoC** ó **S**ystem **o**n a **C**hip, por sus siglas en inglés. Sin embargo, para solventar la demanda del alto nivel computacional actual era necesario la combinación de varios **SoC's** que compartían a su vez el control y procesamiento [1]. La solución natural fue el desarrollo del **M**ultiprocessor **S**ystem-**o**n-**C**hips (**MPSoC**), un **SoC** con múltiples memorias y unidades de procesamiento (muchas veces heterogéneas), conectadas y comunicadas entre sí; a partir de esto, entonces cada unidad de procesamiento podría satisfacer alguna tarea en específico. [2]

Este avance, como todos, incluye sus propios retos; tal vez el más desapercibido es la brecha que aún se tiene en el uso de lenguajes y técnicas modernas de programación en los ya mencionados **SoC** y **MPSoC.** Por ejemplo, aunque **Bjarne Stroustrup**, el creador del lenguaje **C++**, menciona que uno de los objetivos principales de **C++ 11** fue *"Mejorar el rendimiento y la capacidad de trabajar directamente con el Hardware"* [3], lo cierto es que aún existe bastante renuencia en el uso del mismo en **SoC's** y **MPSoC's**, este desapego incrementa incluso si hablamos del uso de técnicas y paradigmas de programación modernas. También, aquí, se ve el caso de la **P**rogramación **O**rientada a **O**bjetos (**POO**) que a pesar de su amplio éxito no han sido tan aceptadas en el desarrollo de software para sistemas embebidos [4].

Esta apatía toma como base la reducción del desempeño o incluso el incremento del consumo de memoria en el desarrollo de sistemas embebidos, pero se tendría que evaluar si tal precio es incosteable al cubrir la necesidad de abstracción, modularidad y una base sólida para la computación concurrente en multiprocesadores [5].

La luz al final del camino es que, a pesar del desánimo en el uso de implementaciones modernas, se han creado frameworks y librerías para trabajar sobre ello. **FreeRTOS** es uno de los más aceptados [6], este framework implementa concurrencia y paralelismo en varias marcas y modelos de **MPSoC's** y **SoC's**. Sin embargo y aun cuando **FreeRTOS** es un excelente framework aún se hace notar la separación que tiene con las implementaciones de alto nivel como las que proporciona la **P**rogramación **O**rientada a **O**bjetos.

En este artículo se desarrolla una librería que incorpora un nivel de abstracción al Framework **FreeRTOS** haciendo uso de la **POO** para el desarrollo de sistemas concurrentes y paralelos en ambientes embebidos.

2. Bases para la Implementaciones de la Librería

Una biblioteca es un componente de software reutilizable que ahorra tiempo al proporcionar acceso al código que realiza una tarea de programación [7] que podría llegar a representar una abstracción de algún artefacto o concepto.

Esta última idea no interfiere con la representación de esas tareas como sucesos ocurridos en el mismo umbral de tiempo durante la ejecución, es decir, de forma concurrentemente, potencialmente paralela, como partes de un cálculo o proceso sin importar el orden particular de ejecución [5].

Para que eso se lleve a cabo se deben de implementar diferentes características dentro de tal librería como son:

- **POO**.- **Clases**, **Herencia** y **Polimorfismo** para especializar objetos que tienen una base en común. Esto genera la posibilidad de crear diferentes tipos de tareas.
- **Lambdas**.- Las **lambdas** con la sentencia **USING** proporcionan una forma de terminar la implementación, es decir, agrega los mecanismos para asignar funciones anónimas con tipos definidos a la ejecución de cada tarea.
- **Colas**.- Las **colas** de mensajes o **queues** son uno de los mecanismos más usados para la comunicación entre tareas de **FreeRTOS**, el uso de colas bajo este modelo brinda un sistema de comunicación alternativo y concurrente entre objetos.

La lista previamente mencionada proporciona una base para el desarrollo de las clases e implementación de una librería para tareas genéricas para su ejecución concurrente.

3. Abstracción de las Tareas de FreeRTOS en Clases

Envolver las tareas de **FreeRTOS** en clases y objetos agrega varias ventajas, la primera sin duda es la conceptualización de las tareas como objetos con un comportamiento definido.

El paradigma de **POO** aplicado a las tareas de **FreeRTOS** genera implementaciones de las mismas como objetos, lo cual permite asignarle una prioridad, un nombre y/o incluso acceso a comportamientos como detener su ejecución, todo esto desde una única "*referencia*". Esta ventaja abre la puerta a la creación de métodos adicionales que enriquezcan el comportamiento de las tareas.

Para generar las clases que "*envuelven*" las tareas de **FreeRTOS**, hace falta ver el contexto de ejecución de cada tarea. La sentencia **xTaskCreate** (código 1), ya definida en **FreeRTOS**, proporciona una buena introducción.

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    configSTACK_DEPTH_TYPE usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask
                    );
```

Código 1 - Cabecera de la función xTaskCreate [8].

Como se puede ver, esta función recibe 6 parámetros que describen su marco de ejecución.

- El primer parámetro, es la función que va a desempeñar, es decir, el código de ejecución de la misma tarea.
- El siguiente parámetro es el nombre, este parámetro es sólo informativo y da la posibilidad de generar un seguimiento más personal.

- El tercero es la memoria asignada a la tarea.
- El cuarto representa los parámetros de la función descrita en el primer parámetro.
- El quinto parámetro es la prioridad de la tarea; ya que las tareas compiten por un tiempo de procesador es necesario asignar prioridades para un buen manejo de este recurso.
- El último es el ***pxCreatedTask**, y es la referencia que usa **FreeRTOS** para apuntar a la tarea recién creada.

Se considera que cada parámetro es importante pero solo los parámetros de entrada (***pvParameters**) y el ***pxCreatedTask** pueden pasarse como nulos.

4. Implementación de las Clases

Podemos ver, en la línea 1, código 2, la declaración de una clase base. Como se ha comentado anteriormente es la encargada de instancias los objetos que fungirán como tareas. En la clase **xbTask**, expone la clase más básica de la librería y se aprecia que desde la línea 3 a la línea 8 se describen los parámetros de entrada, así mismo en la línea 11 se puede ver el apuntador de la función que va a ejecutar.

La línea 13 describe el constructor, este trae preasignado el núcleo en el cual se ejecutará la tarea, esta asignación por defecto se usa para omitir tal parámetro en futuras implementaciones en placas que solo cuentan con un solo procesador.

El método **run**, línea 23, envuelve la creación de la tarea con los parámetros antes mencionados, se etiqueta como `virtual` para su posterior reescritura en clases derivadas.

Por último, el método **setTask**, recibe la función que ejecutará tal tarea; la firma de tal función es

```
void (*)(void *pvParams);
```

y es representada por el parámetro `TaskFunction_t pvTaskCode` (**imagen 1**) del método `xTaskCreate`.

En la línea 1, de la código 3, se observa el uso de la sentencia **using** para generar un **alias** que define el tipo de la función anterior. El empleo de la sentencia proporciona semántica, un aspecto fundamental para la aceptación de la librería.

La clase **xTask** presentada en la código 3, línea 4, es una clase derivada de la **xbTask**, se observa que su dimensión es mínima en comparación a su clase padre, sin embargo su funcionalidad es basta, primero se presenta como una plantilla; este mecanismo integra la posibilidad de trabajar con un tipo de dato específico y designado al momento de crear cualquier instancia de la clase. Esta funcionalidad es apreciable porque concede un comportamiento genérico independiente del tipo de dato. El segundo punto a considerar es que la especialidad de la clase **xTask** radica en que cuenta con una instancia de **xQueue<T>**, línea 6 del código 3.

La clase **xQueue**, también etiquetada como plantilla (línea 1, código 4), implementa el sistema de comunicación, su función es incorporar una estructura de datos tipo cola para que funja como mecanismo de comunicación; el concepto detrás del uso de **colas** es contar con un segmento de memoria en que las tareas puedan publicar un elemento procesable para que otras a su vez puedan evaluar dichos elementos.

```
1  class xbTask {
2   protected:
3   int16_t      memSize;
4   int8_t       priority;
5   char*        name;
6   TaskHandle_t tHandle = NULL;
7   void*        params  = NULL;
8   int8_t       xnCore;
9
10    public:
11     tTask task;
12
13     xbTask(  char* n,
14              int16_t mS,
15              int8_t p,
16              int8_t xnC = 0
17              ):
18              name(n),
19              memSize(mS),
20              priority(p),
21              xnCore(xnC) {  }
22
23      virtual void run() {
24       tHandle =
25 xTaskCreatePinnedToCore (
26                   task,
27                   name,
28                   memSize,
29                   params,
30                   priority,
31                   &taskHandle,
32                   xnCore
33           );
34       }
35
36   xbTask* setTask( tTask t ) {
37             task = t;
38             return this;
39         }
40 };
```

Código 2 - Clase base, envoltura.

Para complementar la semántica de la librería se opta por agregar la sobrecarga del operador "**>>**", que como se verá más adelante esto proporciona un flujo de trabajo. La sobrecarga solo es una abreviación del uso del método **send**, línea 8 del código 3.

```
1  using tTask = void (*)(void *pvParams);
2
3  template <class T>
4  class xTask : public xbTask {
5     public:
6          xQueue<T>   queue;
7
8          void send (const T elem) {
9              queue.send(elem);
10         }
11 };
12
13 template <typename T>
14 void operator >> (const T elem, xTask<T>*
   task)
15 {
16    task->send(elem);
17 };
18
```

Código 3 - Clase derivada de **xbTask**.

5. Caso de Uso para Concurrencia.

Se plantea la lectura del puerto serie al tiempo que un "led" se prende y se apaga con un retardo definido, el efecto parpadeo es

implementado usualmente para comprobar el buen funcionamiento del sistema.

El problema clásico del ejemplo anterior es la ejecución secuencial; la lectura del puerto serie depende estrictamente del tiempo que le tome al microcontrolador encender y a apagar el led para generar el parpadeo; además, el reinicio del parpadeo depende del tiempo que le toma al microcontrolador leer y procesar la información que arriba desde el puerto serie.

La librería que aquí se desarrolla resuelve el problema anterior con la clase **`xbTask`**. El código de la **imagen 6**, implementa la respuesta al caso de uso.

Se puede observar en las primeras dos líneas las instancias de las tareas" serial" y "***parpadeo***", de los parámetros cabe destacar el parámetro que se refiere al núcleo, en este caso el parámetro está preasignado a cero, así que las dos tareas se ejecutarán en ese núcleo.

```
1   template <class T>
2   class xQueue {
3      public:
4          xQueueHandle xQ;
5
6      xQueue(int8_t size = 10) {
7          int msgSize = sizeof(T);
8          xQ = xQueueCreate(size, msgSize);
9      }
10
11     void send (T _elemt) {
```

```
12         if ( xQueueSendToBack( xQ,
&_elemt,
13
                  2000/portTICK_RATE_MS) !=
14
                  pdTRUE )
15
             Serial.println("error");
16
     }
17
};
```

Imagen 4 - Clase **xQueue<T>.**

Otro parámetro es la prioridad de la tarea, en este caso la tarea "***serial***" tiene una prioridad más alta que la de "***parpadeo***"; aquí recordamos que las clases de la librería son envoltorios de las funciones de **FreeRTOS** y el manejo de la prioridad depende de **FreeRTOS** y no de la librería.

En la línea 5 y 20, código 5, se observa cómo se pasa la función en forma de **lambda** que representa el código de la tarea a ejecutar, lo más destacable es el alcance de la misma [9], aunque esto depende de la versión de **C++** que estemos usando por lo general se puede trabajar con un alcance más amplio y por referencia. Por último, los métodos **`run`**, líneas 18 y 32 del código 5, desencadenan la ejecución de las tareas; la **figura 1** muestra como **FreeRTOS** implementa el patrón de ejecución.

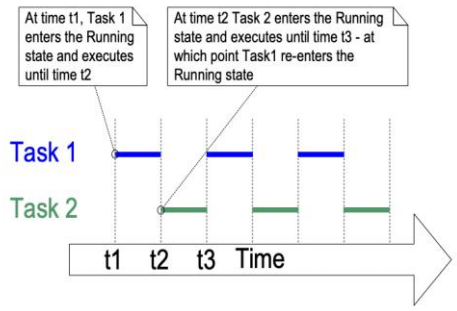


Figura 1 - El diagrama de ejecución de las tareas "***serial***" y "***parpadeo***" [10].

```
1  xbTask serial    = xbTask("Serial",
   512, 3);
2
3  xbTask parpadeo  = xbTask("Parpadeo",
   512, 2);
4
5
6  tTask tSerial = [] (void
   *pvParameter)-> void
7  {
8     for (;;)
9        {
10         if(Serial.available())
11         {
12           // proceso de la entrada
   serial
13         }
14       }
15 };
16
17
18 void setup() {
19
20  serial.setTask(tSerial)->run();
21
22  parpadeo.setTask([] (void
   *pvParameter)-> void
23     {
24       pinMode(led, OUTPUT);
25       for (;;)
26       {
27         digitalWrite(13, 1);
28         delay(1000);
29         digitalWrite(13, 0);
30         delay(1000);
31       }
32     }
33  )->run();
34 }
35
36 void loop()
   { }
```

Código 5 - Implementación de respuesta para el caso de uso de concurrencia.

7. Caso de Uso de Paralelismo

No obstante, en esta situación, se requiere disponer de una placa que cuente con dos procesadores o **MPSoC**. Para lograr una ejecución en paralelo, es esencial que cada tarea se realice en una unidad de procesamiento separada. La figura 2 ilustra de manera gráfica el esquema de ejecución.

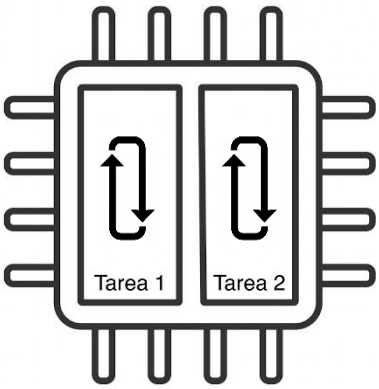


Figura 2 - El patrón de ejecución de las dos tareas de forma paralela.

```
1  xTask serial   = xTask("Serial",
   512, 3, 0);
2
3  xTask parpadeo = xTask("Parpadeo",
   512, 3, 1);
4
5
6  void setup() {
7
8   serial.setTask([] (void *pvParameter)
   -> void {
9
10        //código para leer el puerto
   serie
11
12      }
13
14   )->run();
15
16
17   parpadeo.setTask(
18
19      [] (void *pvParameter) -> void
20
21      {
22
23         //código para el parpadeo
24
25      }
26
27   )->run();
28  }
29
30
31  void loop()
32  { }
```

Código 6 - Implementación de respuesta para el caso de uso paralelo.

El código del código 6, es esencialmente el mismo del caso de uso para la concurrencia, sin embargo, la línea 1 y 2 incorporan el parámetro que indica el núcleo que ejecutará la tarea.

Es importante mencionar que bajo este esquema cada núcleo puede e implementa concurrencia, es decir, cada tarea asignada a un núcleo en particular competirá con las demás por él tiempo de ejecución del mismo, figura 3.

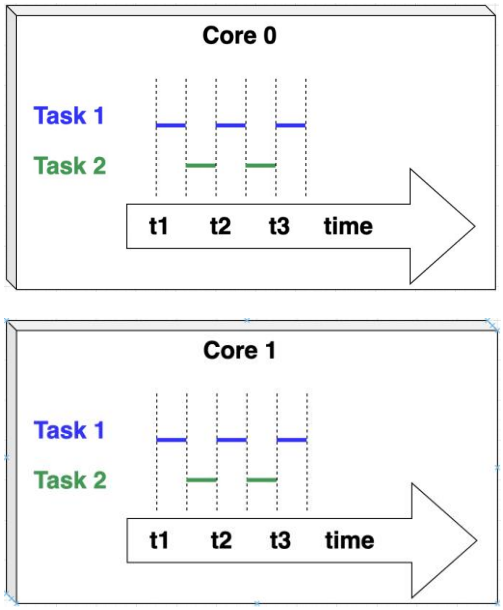


Figura 3 - Ejemplificación de tareas paralelas bajo un esquema concurrente.

8. El Modelo de Actores

En la programación orientada a objetos es habitual y funcional representar objetos mediante métodos o acciones de entrada y salida, los cuales establecen un protocolo claro y significativo de interacción. Este enfoque es el que se utiliza en el modelo de actores [11].

Los actores, al igual que las tareas vistas en este artículo, son componentes autónomos, interactivos e independientes de un sistema informático que se comunican mediante el paso de mensajes asincrónicos [12].

*"El **término actor** fue introducido por **Carl Hewitt** en el MIT a principios de la década de*

1970 *para describir el concepto de agentes razonadores. Se ha refinado a lo largo de los años en un modelo de concurrencia"* [5].

Las acciones ó primitivas básicas del comportamiento de un actor son: **create**, **sent to** y **become**.

create (**crear**): Se refiere a crear un actor con una funcionalidad bajo su propio contexto de ejecución; esto se hace al pasarle la función **tTask** a la clase **xTask**.

send to (**enviar a**): envía un elemento procesable de forma asíncrona que se coloca en el buzón (en la **xQueue** del objeto) del actor como una forma de comunicación con él. En caso de esta librería la dirección de cada Actor es la referencia de la instancia misma en la que podemos usar el método send o el operador **">>"** para tal envío;

become (**convertirse en**): describe que un actor adapta su ejecución; como cada instancia maneja su propio contexto el estado del mismo depende de sus propias variables, en esta librería incluso se podría cambiar la función que se ejecuta por otra a partir del método **setTask**.

En el modelo de actores, el cambio de estado se especifica usando comportamientos de reemplazo **setTask**.

Estas primitivas, aunque básicas, dan pie a abstracciones y paradigmas de alto nivel en programación concurrentes [13].

A partir de este momento, cada vez que un actor procesa un elemento de su buzón, también calcula su comportamiento en respuesta al próximo envío que pueda procesar [14].

Los elementos que llegan al buzón de cada actor en esencia pueden ser denominados eventos y representan la forma en que los actores pueden comunicarse. La información contenida en el evento, es variada, sin embargo, basta el agregar la referencia del actor que lo envía (en remitente) para generar un ciclo de trabajo.

Aunque la **OTP** u Open Telecom Platform, por sus siglas en inglés, en Erlang y Elixir proporcionan ya un conjunto de bibliotecas y principios de diseño estamos seguros que su adopción en plataformas y lenguajes orientados a software embebido será más amplia. Prueba de ello es **Nerves** que es una plataforma de código abierto que combina la máquina virtual **BEAM** (la máquina virtual de Erlang) y el lenguaje Elixir para construir sistemas embebidos [15].

9. CONCLUSIONES.

El uso de colas en **FreeRTOS** está abierto a la implementación del usuario final, si bien esto proporciona libertad creativa, cuando se habla de concurrencia es mejor tener un modelo o guía de referencia.

El **Modelo de Actores** (**MA**) es la guía que aquí se ha adoptado para este fin. La clase **xTask** junto con la clase **xQueue** pueden llegar a representar la implementación de un **Actor** y generar la aproximación al **MA** en esta librería.

Son perceptible las condiciones descritas por **Nelson Rodríguez** al comentar que "*El advenimiento de la concurrencia masiva a través de la computación en la nube y las arquitecturas de computadora multi núcleo ha estimulado el interés en el Modelo de Actores*" [16], sin embargo, se cree que aún está por venir un mayor auge para este modelo.

10. Referencias.

[1] Chicaiza, W. M., & Verdesoto, D. G. (2013). Diseño e Implementación de un Multiprocessor Systems-on-Chip (MPSoC) Interconectado por una Networks-on-Chip (NoC). MASKAY, 3(1), 40–48. https://doi.org/10.24133/maskay.v3i1.129 https://journal.espe.edu.ec/ojs/index.php/maskay/article/view/129/pdf

[2]

[3] Schranzhofer, Andreas & Chen, Jian-Jia & Thiele, Lothar. (2010). Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. Industrial Informatics, IEEE Transactions on. 6. 692 - 707. 10.1109/TII.2010.2062192.

[4] Bjarne Stroustrup“Mejorar el rendimiento y la capacidad de trabajar directamente con el Hardware” https://www.stroustrup.com/C++11FAQ.html

[5] puedo citar Alberto Pacheco, http://electro.itchihuahua.edu.mx/revista/2021/C-Sub32.pdf

[6] Gul Agha. 1990. Concurrent Object-Oriented Programming. Commun. ACM 33, 9 (sep 1990), page 126.

[7] AWS FreeRTOS. (1 de 6 de 2023). Un sistema operativo de tiempo real para dispositivos limitados por los recursos. Obtenido de https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html.

[8] Colebourne, S. & Java Magazine Written by the Java community for Java and JVM developers [Java Magazine]. (2020, January 20). *Designing and Implementing a Library*. Java Magazine. Retrieved May 8, 2023, from https://blogs.oracle.com/javamagazine/post/designing-and-implementing-a-library

[9] FreeRTOS Tasks and Co-routines. (1 de 6 de 2023). FreeRTOS Tasks and Co-routines. Obtenido de https://www.freertos.org/taskandcr.html

[10] Lambda Expressions. (1 de 6 de 2023). C++ Reference Lambda Expressions. Obtenido de https://en.cppreference.com/w/cpp/language/lambda

[11] Barry, R. & FreeRTOS. (2016). Mastering the FreeRTOSTM Real Time Kernel, A Hands-On Tutorial Guide (Pre-release 161204 Edition.). © Real Time Engineers Ltd. 2016. https://www.freertos.org

[12] L. Eunsang. Developing a low cost microcontroller based model for teaching and learning. European Journal of education research. 2020 pages 921-934.

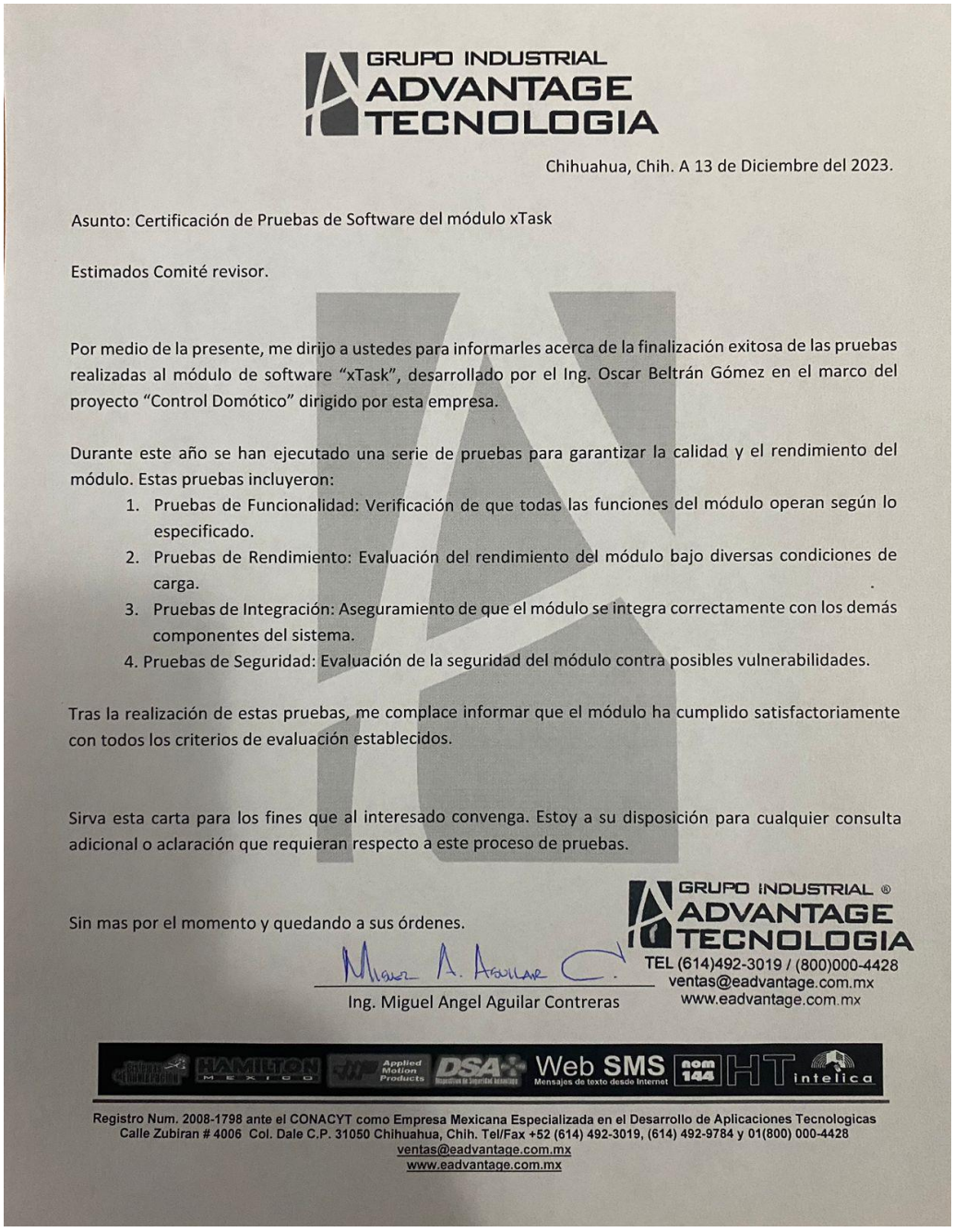
[13] Agha, G. (2004). ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, Massachusetts: MIT Artificial Intelligence Laboratory, pages 141.

[14] Gul Agha. 1990. Concurrent Object-Oriented Programming. Commun. ACM 33, 9 (sep 1990), page 128.

[15] Nerves Project. (n.d.). © The Nerves Project Authors 2023. Retrieved July 24, 2023, from https://nerves-project.org/

[16] Rodríguez, N. R., Murazzo, M. A., & Runco, T. (2019). El modelo de programación de actor aplicado a Edge Computing utilizando Calvin. Web

Anexo 7.- Carta de la empresa Advantage Tecnología validando el uso del framework en un proyecto propio



GRUPO INDUSTRIAL ADVANTAGE TECNOLOGIA

Chihuahua, Chih. A 13 de Diciembre del 2023.

Asunto: Certificación de Pruebas de Software del módulo xTask

Estimados Comité revisor.

Por medio de la presente, me dirijo a ustedes para informarles acerca de la finalización exitosa de las pruebas realizadas al módulo de software "xTask", desarrollado por el Ing. Oscar Beltrán Gómez en el marco del proyecto "Control Domótico" dirigido por esta empresa.

Durante este año se han ejecutado una serie de pruebas para garantizar la calidad y el rendimiento del módulo. Estas pruebas incluyeron:

1. Pruebas de Funcionalidad: Verificación de que todas las funciones del módulo operan según lo especificado.
2. Pruebas de Rendimiento: Evaluación del rendimiento del módulo bajo diversas condiciones de carga.
3. Pruebas de Integración: Aseguramiento de que el módulo se integra correctamente con los demás componentes del sistema.
4. Pruebas de Seguridad: Evaluación de la seguridad del módulo contra posibles vulnerabilidades.

Tras la realización de estas pruebas, me complace informar que el módulo ha cumplido satisfactoriamente con todos los criterios de evaluación establecidos.

Sirva esta carta para los fines que al interesado convenga. Estoy a su disposición para cualquier consulta adicional o aclaración que requieran respecto a este proceso de pruebas.

Sin mas por el momento y quedando a sus órdenes.

Miguel A. Aguilar C.

Ing. Miguel Angel Aguilar Contreras

GRUPO INDUSTRIAL ® ADVANTAGE TECNOLOGIA
TEL (614)492-3019 / (800)000-4428
ventas@eadvantage.com.mx
www.eadvantage.com.mx

Registro Num. 2008-1798 ante el CONACYT como Empresa Mexicana Especializada en el Desarrollo de Aplicaciones Tecnologicas
Calle Zubiran # 4006 Col. Dale C.P. 31050 Chihuahua, Chih. Tel/Fax +52 (614) 492-3019, (614) 492-9784 y 01(800) 000-4428
ventas@eadvantage.com.mx
www.eadvantage.com.mx

Anexo 8.- Constancia del taller “Desarrollo de Aplicaciones Single Web Applications”.

Folio: TNM-028-15-2021/I01.

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

EL TECNOLÓGICO NACIONAL DE MÉXICO
A TRAVÉS DEL INSTITUTO TECNOLÓGICO DE CHIHUAHUA II

OTORGA EL PRESENTE

RECONOCIMIENTO

A

OSCAR BELTRÁN GÓMEZ

POR SU DESTACADA PARTICIPACIÓN COMO INSTRUCTOR DEL CURSO "INTRODUCCIÓN A LOS SISTEMAS IoT CON PLACAS ESP" REALIZADO DEL 6 AL 10 DE SEPTIEMBRE DE 2021 CON DURACIÓN DE 30 HORAS.

CHIHUAHUA, CHIH. A 23 DE FEBRERO DE 2022.

DRA. LUISA YOLANDA QUIÑONES MONTENEGRO
DIRECTORA

Dirección General de educación Superior Tecnológica
Instituto Tecnológico d Chihuahua II

TNM-028-23-2021/I-01

Anexo 9.- Constancia del taller “Introducción a los Sistemas IoT con Placas ESP”.

Folio: TNM-028-15-2021/I01.

EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO

EL TECNOLÓGICO NACIONAL DE MÉXICO
A TRAVÉS DEL INSTITUTO TECNOLÓGICO DE CHIHUAHUA II

OTORGA EL PRESENTE

RECONOCIMIENTO

A

OSCAR BELTRÁN GÓMEZ

POR SU DESTACADA PARTICIPACIÓN COMO INSTRUCTOR DEL CURSO "INTRODUCCIÓN A LOS SISTEMAS IoT CON PLACAS ESP" REALIZADO DEL 6 AL 10 DE SEPTIEMBRE DE 2021 CON DURACIÓN DE 30 HORAS.

CHIHUAHUA, CHIH. A 23 DE FEBRERO DE 2022.

DRA. LUISA YOLANDA QUIÑONES MONTENEGRO
DIRECTORA

Dirección General de educación Superior Tecnológica
Instituto Tecnológico de Chihuahua II

TNM-028-23-2021/I-01

GLOSARIO

Este glosario proporciona definiciones básicas de los términos y abreviaciones relacionadas con sistemas computacionales, desarrollo de sistemas embebidos y sistemas IoT.

1. Lambdas: Funciones anónimas en programación que se pueden utilizar como argumentos o valores de retorno.

2. Colas o Queues: Estructuras de datos que siguen el principio FIFO (primero en entrar, primero en salir) para la gestión de elementos.

3. HTTP: Protocolo de transferencia de hipertexto, utilizado para la comunicación en la World Wide Web.

4. MQTT: Protocolo de mensajería ligero y de suscripción utilizado en aplicaciones de IoT y comunicación de máquina a máquina.

5. Modbus: Protocolo de comunicación utilizado en sistemas de automatización industrial para el intercambio de datos entre dispositivos electrónicos.

6. DSL: Lenguaje de dominio específico (Domain-Specific Language), un lenguaje de programación diseñado para abordar un dominio o problema específico.

7. Encadenamiento de Métodos: Patrón de diseño en el que se invocan varios métodos de un objeto de forma consecutiva.

8. Programación Orientada a Objetos (POO): Paradigma de programación que se basa en la creación y manipulación de objetos que interactúan entre sí.

9. FreeRTOS: Sistema operativo en tiempo real de código abierto diseñado para sistemas embebidos y aplicaciones IoT.

10. Arduino: Plataforma de hardware y software de código abierto utilizada para crear proyectos interactivos y sistemas embebidos.

11. Framework: Conjunto de herramientas, bibliotecas y normas que proporcionan una estructura para el desarrollo de software.

12. IA (Inteligencia Artificial): Campo de estudio y desarrollo de sistemas y algoritmos que exhiben características de inteligencia humana.

13. TIC's: Tecnologías de la Información y la Comunicación, término que engloba los dispositivos y sistemas relacionados con la informática y las comunicaciones.

14. Big Data: Conjunto de técnicas y herramientas utilizadas para analizar y procesar grandes volúmenes de datos.

15. I4.0 (Industria 4.0): Concepto de la cuarta revolución industrial, que se centra en la interconexión de máquinas y sistemas inteligentes en la industria.

16. IIoT (Industrial Internet of Things): Integración de dispositivos y sistemas de IoT en entornos industriales.

17. Domótica: Automatización de viviendas y edificios a través de la integración de tecnologías de IoT.

18. Desarrollo de Software Ágil: Enfoque de desarrollo de software basado en la colaboración, la adaptabilidad y la entrega incremental.

19. Sistemas Embebidos: Sistemas informáticos diseñados para realizar tareas específicas y generalmente integrados en otros dispositivos.

20. BBC: British Broadcasting Corporation, una corporación de medios y radiodifusión del Reino Unido.

21. Librerías: Conjunto de funciones y rutinas predefinidas que se pueden utilizar en el desarrollo de software.

22. Core: Núcleo o parte central de un sistema o plataforma.

23. Núcleo: Parte central de un sistema operativo que gestiona los recursos y brinda servicios a las aplicaciones.

24. Modelo: Representación abstracta de un sistema o entidad en el contexto del desarrollo de software.

25. Ciudades Inteligentes: Entornos urbanos que utilizan tecnología y datos para mejorar la calidad de vida

26. C++: Lenguaje de programación de propósito general que amplía el lenguaje C con características de programación orientada a objetos.

27. PlatformIO: Plataforma de desarrollo y gestión de software embebido, compatible con diferentes plataformas y frameworks.

28. Expressif: Empresa de tecnología que desarrolla microcontroladores y módulos utilizados en aplicaciones de IoT, como los chips ESP8266 y ESP32.

29. MPLab: Entorno de desarrollo integrado (IDE) utilizado para programar microcontroladores de la familia PIC.

30. RTOS: Sistema operativo en tiempo real (Real-Time Operating System) diseñado para aplicaciones que requieren respuestas rápidas y predecibles.

31. Google Cloud: Plataforma de servicios en la nube ofrecida por Google, que proporciona almacenamiento, computación y otros servicios para aplicaciones y datos.

32. Azure: Plataforma de servicios en la nube de Microsoft, que ofrece una amplia gama de servicios y herramientas para el desarrollo y despliegue de aplicaciones.

33. ESP8266: Chip de bajo costo y bajo consumo de energía utilizado en aplicaciones de IoT y Wi-Fi.

34. ESP32: Microcontrolador de bajo costo y bajo consumo de energía con capacidades avanzadas de conectividad, utilizado en aplicaciones de IoT.

35. STM32: Familia de microcontroladores de 32 bits desarrollados por STMicroelectronics, ampliamente utilizados en aplicaciones embebidas.

36. Raspberry Pi: Computadora de placa única (SBC) de bajo costo y tamaño reducido, utilizada para proyectos de electrónica y desarrollo de sistemas embebidos.

37. Beaglebone: Plataforma de desarrollo de hardware de una sola placa que combina un microprocesador ARM con capacidades de E/S.

38. Orange Pi: Serie de computadoras de placa única basadas en el procesador ARM, utilizadas para aplicaciones de IoT y sistemas embebidos.

39. Odroid: Familia de computadoras de placa única de alto rendimiento y bajo consumo de energía, utilizadas en una variedad de aplicaciones.

40. Jetson Nano: Plataforma de computación de inteligencia artificial (AI) desarrollada por NVIDIA para aplicaciones de visión por computadora y aprendizaje automático.

41. Coral Google: Plataforma de inteligencia artificial de Google que incluye hardware y software optimizados para aplicaciones de aprendizaje automático en dispositivos Edge.

42. MarketsandMarkets: Empresa de investigación de mercado que proporciona informes y análisis en diversas industrias y tecnologías.

43. AWS IoT: Servicio de Amazon Web Services que permite conectar y administrar dispositivos de IoT de manera segura y escalable.

44. IBM: International Business Machines Corporation, una empresa multinacional de tecnología que ofrece servicios y soluciones en diversas áreas.

45. IBM IoT: Soluciones y servicios de Internet de las cosas ofrecidos por IBM para ayudar a las empresas a conectar, gestionar y analizar dispositivos de IoT.

46. Cisco: Empresa de tecnología que ofrece soluciones de redes, comunicaciones y servicios relacionados.

47. Cisco Kinetic: Plataforma de software de IoT de Cisco que facilita la administración y análisis de datos de sensores y dispositivos.

48. Intel: Empresa de tecnología que desarrolla y fabrica procesadores y otros componentes electrónicos.

49. Huawei: Empresa de tecnología china que ofrece productos y soluciones en el campo de las telecomunic

50. Huawei IoT: Soluciones y plataformas de Internet de las cosas desarrolladas por Huawei para aplicaciones de conectividad y gestión de dispositivos IoT.

51. Harvested Framework: Marco de desarrollo de software utilizado para facilitar la implementación de aplicaciones IoT y sistemas embebidos.

52. Xtensa® Dual-Core 32-bit LX6: Arquitectura de procesador de baja potencia desarrollada por Tensilica, utilizada en microcontroladores y sistemas embebidos.

53. SRAM: Memoria de acceso aleatorio estática, un tipo de memoria volátil utilizada para almacenar datos en sistemas embebidos.

54. ROM: Memoria de solo lectura, un tipo de memoria no volátil utilizada para almacenar datos permanentes o programas que no requieren modificaciones.

55. Java: Lenguaje de programación de propósito general que se enfoca en la portabilidad y la orientación a objetos.

56. Python: Lenguaje de programación de alto nivel conocido por su simplicidad y legibilidad, utilizado en una amplia gama de aplicaciones.

57. Lenguaje C: Lenguaje de programación de bajo nivel ampliamente utilizado para el desarrollo de sistemas embebidos y aplicaciones de tiempo real.

58. Clases: Componentes fundamentales de la programación orientada a objetos que definen objetos con propiedades y comportamientos.

59. Integración: Proceso de combinar diferentes componentes o sistemas para que funcionen juntos de manera coherente.

60. Herencia: Concepto de programación orientada a objetos en el que una clase puede heredar propiedades y comportamientos de otra clase.

61. Polimorfismo: Capacidad de un objeto para tomar diferentes formas y comportarse de diferentes maneras según el contexto.

62. MIT: Instituto Tecnológico de Massachusetts, una de las principales instituciones de educación superior y de investigación en ciencia y tecnología.

63. Actores: Entidades independientes y concurrentes en sistemas distribuidos o paralelos que interactúan entre sí enviando y recibiendo mensajes.

64. Concurrencia: La capacidad de ejecutar múltiples tareas en paralelo o de forma intercalada en un sistema.

65. Paralelismo: La capacidad de realizar múltiples tareas simultáneamente en un sistema con múltiples recursos de procesamiento.

66. Buzón o buffer: Estructura de datos utilizada para almacenar temporalmente datos en tránsito entre diferentes componentes o procesos.

67. Deadlock: Situación en la que dos o más procesos quedan bloqueados esperando recursos que nunca se liberan, impidiendo que avancen.

68. Referencia: En programación, una referencia es una dirección que apunta a un objeto o ubicación de memoria.

69. Task: Una tarea en sistemas operativos o sistemas embebidos, que representa una unidad de trabajo o proceso a realizar.

70. Real Time: Tiempo real, en el contexto de sistemas embebidos, se refiere a la capacidad de responder a eventos y ejecutar tareas dentro de límites de tiempo determinados.

71. SDK: Kit de desarrollo de software, un conjunto de herramientas y recursos que facilitan el desarrollo de aplicaciones para una plataforma o sistema.

72. Metodología Iterativa: Enfoque de desarrollo de software en el que el proceso de desarrollo se divide en ciclos repetitivos de planificación, desarrollo y revisión.

73. Blink.ino: Es un archivo de código fuente utilizado en el entorno de programación Arduino, es probablemente el primer ejemplo expuesto por la plataforma para probar el buen funcionamiento de la misma.

de desarrollo de Arduino para crear un programa que hace parpadear un LED.

74. Linkeo: Proceso de vinculación de diferentes módulos de código durante la compilación para crear un ejecutable final.

75. Handle: Un identificador o referencia a un recurso o entidad en un sistema, utilizado para manipular o acceder a ese recurso.

76. Run: Término utilizado para iniciar la ejecución de un programa o proceso.

77. Directiva: Instrucción o comando utilizado en el código fuente para indicar al compilador o al sistema cómo procesar o configurar ciertas características.

78. Parámetros: Valores o variables proporcionadas a una función o método para que sean utilizados durante su ejecución.

79. Template: Plantilla o patrón genérico utilizado para crear objetos, funciones o clases parametrizadas.

80. Genéricos: En programación, se refiere a elementos que pueden trabajar con múltiples tipos de datos sin requerir especificación.

81. Operadores: Símbolos o palabras clave utilizados para realizar operaciones matemáticas, lógicas o de manipulación de datos en un programa.

82. Método GET: Método utilizado en protocolos como HTTP para solicitar y recibir recursos o información desde un servidor.

83. Método POST: Método utilizado en protocolos como HTTP para enviar datos al servidor para su procesamiento o almacenamiento.

84. WebServer: Servidor de aplicaciones que responde a solicitudes HTTP y proporciona recursos web.

85. Broker: En sistemas de mensajería y comunicación de IoT, un broker es un intermediario que recibe y distribuye mensajes entre diferentes dispositivos o clientes.

86. Bobina: En el contexto de sistemas embebidos o electrónica, una bobina es un componente que almacena energía en forma de campo magnético.

87. Maestro: En sistemas de comunicación o automatización, el maestro es el dispositivo o componente que controla y coordina a otros dispositivos o componentes.

88. Esclavo: En sistemas de comunicación o automatización, el esclavo es el dispositivo o componente controlado o coordinado por un maestro.

89. PROFINET: Protocolo de comunicación industrial utilizado en redes de automatización para intercambiar datos en tiempo real entre dispositivos.