PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

Diseño de una Metodología para la realización de Pruebas Unitarias en Sistemas Embebidos que emplean el framework de Arduino

Tesis para obtener el título profesional de Ingeniero Electrónico

AUTOR:

Joel Alberto Lozano Tapia

ASESOR:

Jorge Benavides Aspiazu

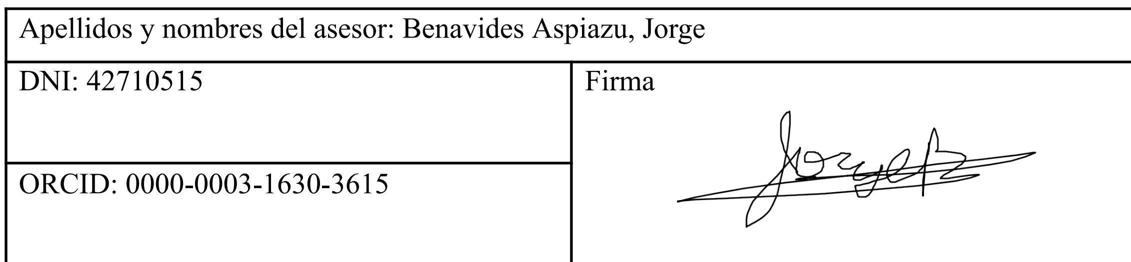
Lima, Septiembre, 2025

Informe de Similitud

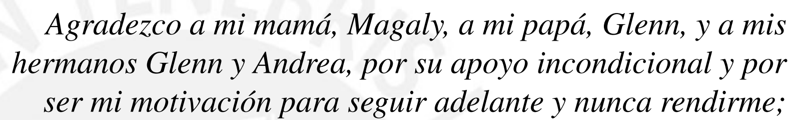
Yo, **Jorge Benavides Aspiazu**, docente de la **Facultad de Ciencias e Ingeniería** de la Pontificia Universidad Católica del Perú, asesor de la tesis titulada: **Diseño de una metodología para la realización de pruebas unitarias empleando el framework de Arduino**, del autor **Joel Alberto Lozano Tapia** dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 13%. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 28/09/2025.
- He revisado con detalle dicho reporte y la Tesis o Trabajo de Suficiencia Profesional, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

San Miguel, Lima, lunes 29 de septiembre,



Apellidos y nombres del asesor: Benavides Aspiazu, Jorge	
DNI: 42710515	Firma
ORCID: 0000-0003-1630-3615	



Agradezco a mi mamá, Magaly, a mi papá, Glenn, y a mis hermanos Glenn y Andrea, por su apoyo incondicional y por ser mi motivación para seguir adelante y nunca rendirme;

a mi asesor de tesis, Jorge Benavides, por su guía y apoyo en mis últimos pasos para convertirme en profesional;

a todos mis compañeros y amigos que he podido conocer a lo largo de la carrera, por todos los buenos momentos compartidos;

y a todos mis amigos y profesores del mundo de la marinera norteña, por impulsarme a lograr mis objetivos y regalarme sonrisas cuando más lo necesitaba.

Resumen

El desarrollo de sistemas embebidos cada vez se hace más complejo debido a la cantidad de tecnologías existentes y el sinfín de aplicaciones que presentan. Existen áreas críticas para la seguridad como la aeronáutica, espacial y médica que hacen uso de sistemas embebidos, los cuales su funcionamiento debe ser verificado correctamente, ya que una falla del mismo podría atentar con la vida de una o varias personas. Es por ello que es importante tener en cuenta procesos de pruebas y validación de funcionamiento como parte del desarrollo de un sistema embebido. Actualmente, existen estándares y normas como la IEC-61508 y la ISO-26262 para el desarrollo de sistemas embebidos, los cuales se rigen por el modelo de desarrollo V, un modelo que en la etapa de pruebas señala 3 etapas: pruebas unitarias, pruebas de integración y pruebas de sistema, siendo las pruebas unitarias la primera etapa de pruebas. Sin embargo, no existe una guía o metodología que permita conocer el proceso, consideraciones y herramientas necesarias para poder realizar pruebas unitarias en un sistema embebido, ya que este tipo de pruebas suele emplearse comúnmente en únicamente software. El presente trabajo de tesis presenta el diseño de una metodología para realizar pruebas unitarias en sistemas embebidos, con el fin de dar a conocer a los desarrolladores, las herramientas, consideraciones y metodologías ágiles necesarias para su uso en este tipo de sistemas, centrándose en software embebido desarrollado con el framework de Arduino.

Índice

Introducción **1**

I. Marco problemático **3**
1.1. Descripción del problema . . . 3
1.2. Estado del arte . . . 4
1.2.1. SmartUnit . . . 5
1.2.2. TECSUnit . . . 7
1.2.3. Automated Compilation Test System . . . 8
1.3. Justificación . . . 11
1.4. Objetivos . . . 12
1.4.1. Objetivo general . . . 12
1.4.2. Objetivos específicos . . . 12

II. Marco teórico **13**
2.1. Fundamentos de los Sistemas Embebidos . . . 13
2.1.1. Definición y características de los sistemas embebidos . . . 13
2.1.2. Aplicaciones y desafíos en el desarrollo de sistemas embebidos . . . 15
2.1.3. Complejidad y problemas de diseño de sistemas embebidos . . . 15
2.1.4. Arquitectura y componentes de los sistemas embebidos . . . 16
2.2. Fundamentos de las Pruebas Unitarias . . . 18
2.2.1. Definición de las pruebas unitarias . . . 18
2.2.2. Características de las pruebas unitarias . . . 19
2.2.3. Aspectos para tomar en cuenta en la implementación de pruebas unitarias 21
2.2.4. Clasificación de las pruebas unitarias . . . 22
2.3. Modelo V de desarrollo de software . . . 22
2.4. Metodologías de desarrollo de software embebido para realizar pruebas unitarias 24
2.4.1. Ejecución de pruebas simbólicas . . . 24
2.4.2. Test Driven Development . . . 25
2.4.3. Behavior Driven Development . . . 26
2.4.4. Component Based Software Development . . . 26
2.4.5. Model Based Development . . . 27
2.4.6. Equivalence Partitioning . . . 28
2.5. Herramientas y tecnologías para realizar pruebas unitarias en sistemas embebidos 29
2.5.1. Entornos de desarrollo de software embebido . . . 29

2.5.2. Frameworks de desarrollo de software embebido 31
2.5.3. Frameworks para pruebas unitarias, simuladores y emuladores 32
2.5.4. Herramientas de reporte de cobertura 36

III. Diseño de la metodología propuesta **37**
3.1. Definición de requerimientos del sistema 39
3.2. Análisis de diseño del sistema 39
3.3. Planificación y Diseño de las pruebas unitarias 40
3.3.1. Consideraciones de diseño de pruebas unitarias según el Estándar IEEE 1008 para pruebas unitarias 42
3.3.2. Características principales de las pruebas unitarias usadas en la metodología propuesta 43
3.4. Desarrollo de las pruebas unitarias 43
3.4.1. Estructura para la declaración de funciones de prueba 47
3.4.2. Procedimiento para la declaración de funciones que simulan la interacción entre hardware y software 48
3.5. Ejecución de las pruebas unitarias 49
3.6. Documentación de resultados de ejecución de pruebas 51

IV. Resultados de aplicación de la metodología **54**
4.1. Desarrollo de la metodología en un sistema embebido desarrollado 54
4.1.1. Definición de requerimientos del sistema 55
4.1.2. Análisis de diseño de sistema 56
4.1.3. Planificación y Diseño de las Pruebas Unitarias 59
4.1.4. Desarrollo de las Pruebas Unitarias 61
4.1.5. Ejecución de las Pruebas Unitarias 62
4.1.6. Documentación de resultado de ejecución de pruebas 63
4.2. Verificación de funcionamiento del sistema embebido desarrollado 66
4.3. Desarrollo de la metodología en un sistema embebido en etapa de desarrollo . . 68
4.4. Verificación de funcionamiento del sistema en etapa de desarrollo 70

Conclusiones **72**

Bibliografía **73**

Anexos **80**

Índice de figuras

1.1. La arquitectura de SmartUnit 5
1.2. Proceso de la ejecución simbólica dinámica 6
1.3. Diagrama de componentes de TECSUnit 8
1.4. Diagrama de estados de la metodología usada en Automated Compilation Test System for Embedded Systems 9
1.5. Diagrama de flujo de ACTS 10

2.1. Ejemplo de conexión entre microcomputadora y componentes físicos en un sistema embebido 14
2.2. Ejemplo de aplicación de sistemas embebidos en el campo de la róbotica 16
2.3. Ejemplo de arquitectura de sistema embebido 17
2.4. Ejemplo de hardware incluído en un sistema embebido 18
2.5. Flujo de datos entre las fases de pruebas unitarias de software 19
2.6. Actividades de las pruebas unitarias 20
2.7. Ejemplo de uso simuladores en ambiente de pruebas en un sistema embebido . 22
2.8. Proceso de desarrollo de software llamado Modelo V 23
2.9. Flujo de trabajo de la metodología Test Driven Development 25
2.10. Flujo de trabajo basado en la metodología Behavior Driven Development . . . 26
2.11. Selección de componentes en desarrollo de software basado en la metodología Component Based Development 27
2.12. Flujo de trabajo de la metodología Model Based Development 28
2.13. Implementación de software embebido en el IDE PlatformIO 31
2.14. Implementación de pruebas unitarias con el framework Unity 33
2.15. Implementación de pruebas unitarias empleando objetos simulados mediante la herramienta CMock 35

3.1. Diagrama de flujo de desarrollo de un sistema embebido 38
3.2. Metodología Propuesta 38
3.3. Diagrama de bloques del sistema usado para el diseño de la metodología 40
3.4. Ambiente de desarrollo declarado para el sistema embebido 44
3.5. Declaración de ambiente de desarrollo nativo para el sistema embebido 45
3.6. Estructura básica de un archivo de pruebas 47
3.7. Adaptación de función para poder realizar pruebas en un ambiente nativo . . . 50
3.8. Selección de ambiente de desarrollo en PlatformIO 51
3.9. Herramienta "Testing"de PlatformIO 51

3.10. Ejemplo de reporte de resultado de pruebas en PlatformIO 52
3.11. Ejemplo de reporte de cobertura de código usando la herramienta gcovr 53

4.1. Fotografía de sistema embebido usado como caso de estudio 55
4.2. Distribución de pines en el módulo WSSFM11R2DAT 57
4.3. Distribución de pines en el módulo MPU6050 57
4.4. Distribución de pines en el módulo GPS BN-880 58
4.5. Especificaciones de hardware en el microcontrolador ESP32-WROOM-32 . . . 59
4.6. Librería auxiliar para la simulación de funciones que interactúan con el hardware 61
4.7. Diagramas de flujo de las pruebas realizadas 62
4.8. Configuraciones de pruebas a ignorar en el entorno embebido 63
4.9. Configuraciones de pruebas a ignorar en el entorno nativo 64
4.10. Prueba fallida en entorno embebido 64
4.11. Muestreo de datos de aceleración en el equipo 67
4.12. Detección de vibración en el equipo 67
4.13. Detección de movimiento en el equipo 68
4.14. Verificación de envío de mensajes mediante el backend de Sigfox 68
4.15. Prototipo de sistema embebido con aplicación en jardines 69
4.16. Verificación de envío y recibimiento de mensajes en el prototipo 70
4.17. Verificación de envío de mensajes cada 30 minutos con retardo en el prototipo . 71
4.18. Verificación de envío de mensajes mediante el backend de Sigfox en el sistema en desarrollo 71

Índice de tablas

4.1. Tabla de resultado de pruebas en ambiente nativo 63
4.2. Tabla de resultado de pruebas en ambiente embebido 64
4.3. Tabla de Cobertura de Pruebas 65
4.4. Tabla comparativa de tiempos de ejecución en ambos ambientes 65
4.5. Tabla de números de pruebas consistentes en ambos ambientes 66
4.6. Tabla de resultado de pruebas en ambiente nativo para el sistema en desarrollo . 69

Introducción

Entre las etapas de desarrollo de un sistema embebido, la etapa de pruebas es la que suele tener mayores tiempos de implementación. En el caso del software embebido de estos sistemas, mayormente es validado en funcionamiento cuando el sistema ya ha sido desarrollado a nivel de hardware, resultando así que la etapa de pruebas de software sea dependiente del hardware del mismo. Las pruebas unitarias permiten segmentar las funcionalidades del software y realizar pruebas independientemente de las demás funcionalidades del sistema, cabe resaltar que estas se suelen emplear en únicamente software. En el caso del software embebido, al incluir funcionalidades que dependen del hardware, hace que el desarrollo de este tipo de pruebas sea más complejo. Sin embargo, el desarrollo de pruebas unitarias en software embebido, debido a sus características propias, permitiría validar el funcionamiento del sistema a la par con el desarrollo del software y sin dependencia del hardware, promoviendo también el desarrollo colaborativo del mismo. Por ello, la presente tesis tiene como objetivo diseñar una metodología que permita conocer las consideraciones necesarias, herramientas y metodologías para realizar pruebas unitarias en sistemas embebidos, específicamente en el software embebido del mismo. El capítulo 1 presenta la descripción de la problemática, revisión de trabajos académicos similares en el estado del arte, objetivos y justificación de la tesis. El capítulo 2 incluye los conceptos teóricos de los sistemas embebidos, pruebas unitarias, metodologías existentes de desarrollo de software embebido y herramientas para la realización de pruebas unitarias en software embebido. El capítulo 3 detalla los pasos que incluye la metodología propuesta, indicando las herramientas escogidas a partir de las identificadas en el capítulo 2 y las consideraciones necesarias para la realización de pruebas unitarias en software embebido. Finalmente, el capítulo 4 presenta los resultados de aplicación de la metodología propuesta tanto en un sistema embebido desarrollado a nivel de hardware y en un sistema embebido en etapa de desarrollo, añadiendo también

una comparativa entre las ejecuciones de pruebas en un ambiente de desarrollo embebido y en uno nativo.

Capítulo I

Marco problemático

En el presente capítulo, se desarrollará el marco problemático mediante cuatro subsecciones: En primer lugar, se describirá el problema de la falta de metodologías de desarrollo claras que detallen el uso de pruebas unitarias en sistemas embebidos. En segundo lugar, en el estado del arte, se mostrarán tres metodologías propuestas para la implementación de pruebas en sistemas embebidos, las cuales incluyen en el desarrollo, el uso y definición de pruebas unitarias en este tipo de sistemas. En tercer lugar, se explicarán los objetivos generales y específicos del trabajo de investigación. Finalmente, se abordará la justificación de la presente tesis.

1.1. Descripción del problema

El desarrollo del software de los sistemas embebidos cada vez se vuelve más complejo, esto implica un mayor costo de implementación y tiempo de diseño, así como de validación (Morisaki, Shirata, Oyama, y Azumi, 2020). Realizar pruebas de software en sistemas embebidos implica un mayor reto que pruebas de únicamente software (Hodel, Silva, Yoshioka, Justo, y Santos, 2022), esto se debe a la relación directa entre el hardware y software que existe en estos sistemas. Esta relación implica que las pruebas funcionales sean un reto para el desarrollador, realizar su validación difiere de los métodos de validación de software y hardware que

se usan (Alam, Yang, Chen, Armour, y Ray, 2022). La validación del funcionamiento de un sistema embebido mejora su confiabilidad y seguridad; sin embargo, el procedimiento para realizarlo no está claramente definido. A pesar de que existen ciertas normas y estándares como IEC-61508, ISO-26262, DO-178B/C que indican el proceso de prueba para la validación de sistemas (C. Zhang y cols., 2018), la cual empieza por las pruebas unitarias, no brindan detalles de implementación. Las pruebas unitarias en software son ampliamente usadas para su validación; sin embargo, su uso en software embebido implica otros aspectos característicos del sistema embebido. No usar pruebas unitarias en el desarrollo del software embebido puede afectar en la verificación del comportamiento de cada función (Shin y Lim, 2018). También puede hacer que los errores en el desarrollo se detecten tardíamente, generando así un mayor tiempo de implementación.

1.2. Estado del arte

Los sistemas embebidos se utilizan a menudo en actividades cotidianas como en los electrodomésticos, la comunicación, el transporte, la róbotica, el espacio, etc. (Sinha, Goyal, y Mall, 2021). Por esta razón, su evaluación debe estar sujeta a diversas restricciones, que también pueden ser estrictas (Stoico, 2021). Las pruebas unitarias se han consolidado como una importante actividad de ingeniería para garantizar la calidad del software en la industria, especialmente para los fabricantes de sistemas críticos para la seguridad, como las empresas aeroespaciales y de control de señales ferroviarias (Ishak, Hwan, Jiashen, y Isa, 2019). Por ende, su aplicación en sistemas embebidos, aumenta su seguridad y confiabilidad de funcionamiento. Sin embargo, las pruebas en software embebido son mucho más críticas que las generales, esto se debe a la complejidad de los productos que se desarrollan (Shin y Lim, 2018). Otro aspecto que hace que las pruebas en software embebido sean complejas, es que las pruebas en este tipo de sistemas consisten en comprobar la integración del software y el hardware. Además, cada vez es más difícil evaluar la funcionalidad de cada módulo en poco tiempo debido al creciente número de pruebas necesarias que pueden existir (Ishak y cols., 2019). En este contexto, el estado del arte

se centrará en la revisión de las metodologías y herramientas existentes para la realización de pruebas unitarias en sistemas embebidos. Esta revisión aportará en el desarrollo del diseño de una metodología de accesible entendimiento y aplicación.

1.2.1. SmartUnit

En el artículo científico de Zhang et al. (2018), “SmartUnit: Empirical Evaluations for Automated Unit Testing of Embedded Software in Industry", se detalla el estudio de evaluación, implementación y uso de la herramienta SmartUnit. SmartUnit es una herramienta de pruebas unitarias basada en la cobertura para ahorrar la mano de obra en gran medida. Uno de sus principales objetivos es generar el conjunto de pruebas para conseguir una alta cobertura del código. Esta herramienta sigue el principio de la ejecución simbólica dinámica para la ejecución de las pruebas.

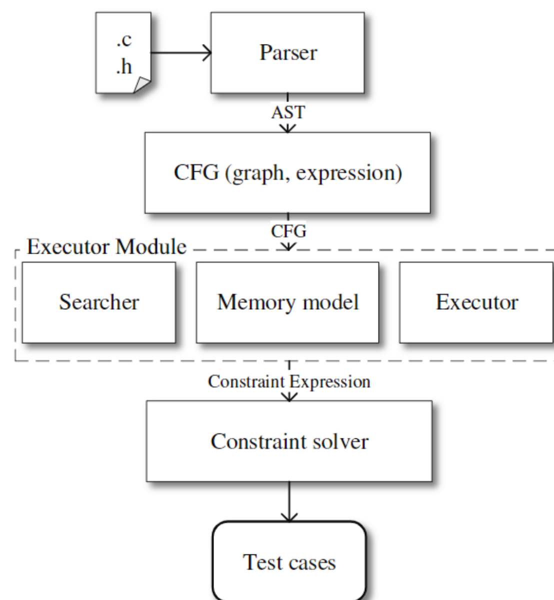


Figura 1.1: La arquitectura de SmartUnit (C. Zhang y cols., 2018)

Para su funcionamiento, primero la herramienta recibe los archivos .c y .h correspondientes al software embebido. SmartUnit interpreta esos archivos mediante un preprocesador, de ma-

nera que pueda trabajar con macros o incluir variables externas al archivo fuente, el cual será procesado posteriormente. Este preprocesador también tiene la funcionalidad de analizar el archivo procesado y generar un árbol de sintaxis abstracta, esto con el fin de generar un modelo de gráfico de control de flujo al software embebido.

Después de haber establecido el gráfico de control de flujo, este es analizado mediante un módulo ejecutor. Este se encarga de evaluar bloques de código, los cuales clasifica dependiendo de sus declaraciones o restricciones de ruta. Posterior a esa clasificación, almacena los bloques con su respectiva información en la herramienta SmartUnit. Con el uso de un buscador integrado, se pueden ejecutar entre diferentes bloques en el código.

Siguiendo el principio de la ejecución simbólica dinámica, SmartUnit calcula el camino correcto y más corto para completar el caso de prueba en cada uno de los bloques de código; para realizarlo, se complementa con un solucionador de restricciones que incluye.

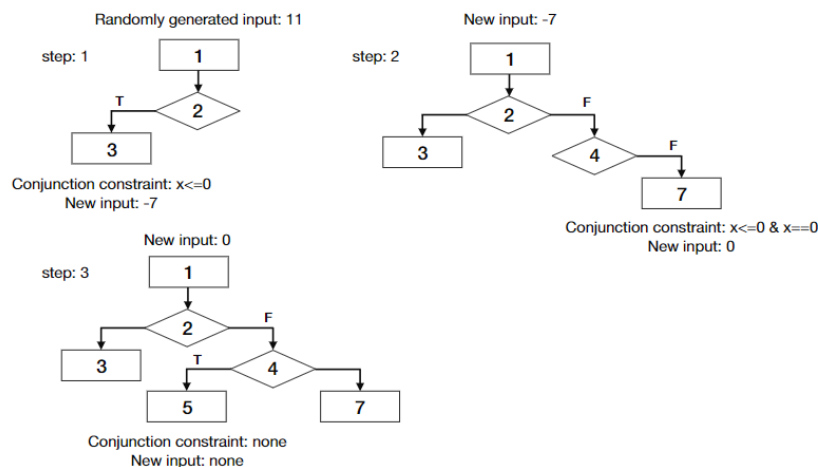


Figura 1.2: Proceso de la ejecución simbólica dinámica (C. Zhang y cols., 2018)

Finalmente, con ayuda de una plataforma en la nube, SmartUnit ayuda a realizar el reporte de cálculos de cobertura por función. Si determinada función cumple con el porcentaje mínimo esperado según el tipo de cobertura, la evaluación de la función ha sido correcta y se ha verificado su correcto funcionamiento.

1.2.2. TECSUnit

En el artículo científico de Morisaki et al. (2020), “Unit Testing Framework for Embedded Component Systems", se menciona a TECSUnit, una propuesta de un marco de trabajo para la realización de pruebas unitarias en sistemas embebidos, el cual se rige a partir del desarrollo basado en componentes.

Lo que caracteriza a los sistemas que siguen este tipo de desarrollo, es que son flexibles y fáciles de reconfigurar, así como de añadir o eliminar componentes. La división del sistema por componentes permite la realización de pruebas unitarias, la cual también permite evaluar el comportamiento de cada componente.

TECSUnit permite el uso de pruebas unitarias a partir de la creación de un código que incluye la descripción de datos de casos de prueba desde varios puntos de vista, señalado en el artículo científico como código JSON. Este código incluye la especificación de los componentes de destino, condiciones previas y posteriores para la prueba, argumentos y valores esperados en el código de prueba. TECSUnit va creando las pruebas unitarias a la par que va leyendo el código JSON, esto evita que los desarrolladores tengan que reconstruir todo el código, aun cuando se actualicen los casos de prueba. Cabe resaltar que se puede acceder a la información de los casos de prueba mediante el marco de trabajo TECSInfo; por lo tanto, se reduce la información a escribir en el código JSON.

Ambos marcos de trabajo mencionados han sido desarrollados con el uso de un sistema desarrollado en base a componentes llamado TOPPERS Embedded Component Systems (TECS) (Azumi y cols., 2007).

Figura 1.3: Diagrama de componentes de TECSUnit (Morisaki y cols., 2020)

1.2.3. Automated Compilation Test System

En el artículo científico de Ishak et al. (2019), “Automated Compilation System for Embedded Systems", se menciona una metodología para realizar pruebas unitarias a partir de un sistema de automatización de compilación de pruebas basado en pruebas de caja negra y pruebas de caja blanca. Este sistema permite ahorrar tiempos de ejecución de pruebas y mejorar la calidad de los resultados obtenidos.

El diseño de la metodología propuesta en esta investigación empieza con el análisis de diseño del sistema, luego con la especificación de los requerimientos, posteriormente con la compilación y ejecución de las pruebas usando la combinación de diferentes métodos de prueba, después con la priorización por casos de prueba y finalmente con la revisión de la secuencia de pruebas.

Para empezar con el análisis del diseño del sistema, se hace una investigación de las terminologías de la comunicación entre hardware y software. Esto ayuda a tener un dominio del desarrollo del software implicado en los sistemas embebidos. En la fase de la especificación de los requerimientos, se describen y analizan en detalle los distintos entornos de la plataforma de desarrollo de código de software.

En la fase de compilación y ejecución de pruebas, se usan dos métodos para la compilación

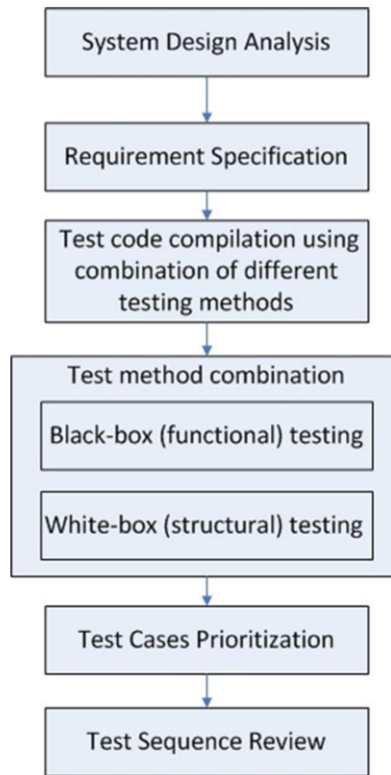


Figura 1.4: Diagrama de estados de la metodología usada en Automated Compilation Test System for Embedded Systems (Ishak y cols., 2019)

de las pruebas, las pruebas por caja negra y las pruebas por caja blanca. En el caso de pruebas por caja negra, se toman en cuenta todas las posibles combinaciones de configuración para cada archivo del proyecto, estas configuraciones incluyen tipo de plataforma usada, solución por depuración o liberación y modo de configuración por simulación o hardware. Para las pruebas por caja blanca, se usa un enfoque de pruebas de cobertura de decisiones, el cual permite examinar la estructura interna de un archivo de proyecto. Para la ejecución de las pruebas, en la metodología propuesta se usa un sistema de compilación de pruebas automático (ACTS). Al iniciarse este sistema, se usa una herramienta de compilación, posteriormente se especifica una ruta de ubicación y se exploran todos los archivos de proyecto de código de prueba disponibles, las cuales se enumeran en una tabla de cuadro de lista. Se seleccionan todos los archivos de proyecto de la tabla y se realiza automáticamente la compilación de cada código de prueba.

Finalizada la compilación, todos los archivos de ejecución creados se ejecutan automáticamente sin interacción del usuario. Por último, se analizan conjuntamente los resultados de las pruebas de compilación y ejecución.

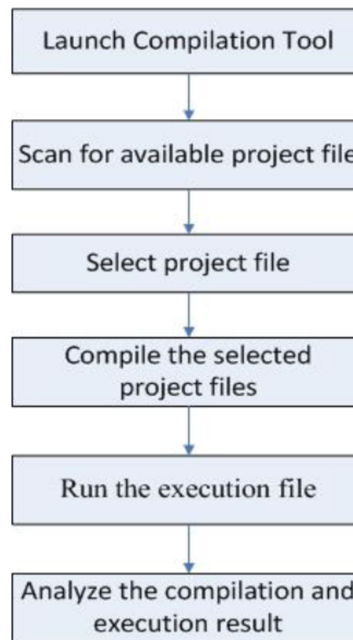


Figura 1.5: Diagrama de flujo de ACTS (Ishak y cols., 2019)

En la fase de priorización de casos de prueba, los resultados son analizados usando tablas y gráficos. Para definir la priorización, se evalúa el rendimiento tanto de la compilación como la ejecución de las pruebas en función del tiempo total consumido. Para finalizar, en la etapa de revisión de secuencia de pruebas, se revisan las mismas para determinar un flujo de pruebas más eficaz para la prueba de regresión. Esta revisión permite maximizar la cobertura de las pruebas con un tiempo de ejecución mínimo.

1.3. Justificación

Hacer pruebas funcionales a un sistema, implica un costo de tiempo de validación mayor o igual que el de implementación (Mizoguchi, Iida, y Irie, 2020). Esto también podría derivarse en mayores costos económicos para identificar y arreglar fallas. La aplicación de pruebas unitarias, al ser autónomas, facilitan la detección de errores en el software embebido, así como su corrección temprana. Su uso agrega una capa adicional a la documentación del software, generando así una mayor trazabilidad al funcionamiento y a la confiabilidad del sistema. Sin embargo, su uso en sistemas embebidos, por sus características propias, hacen que sea complejo. El diverso uso de tecnologías en hardware y software en el sistema embebido, la falta de conocimiento en pruebas en este tipo de sistemas (Garousi, Felderer, Çağri Murat Karapiçak, y Yilmaz, 2018) y su documentación no detallada son algunos de los desafíos que implica. El desarrollo colaborativo de software embebido, a través de herramientas de control de versiones, está limitado a que sus colaboradores tengan acceso al hardware. A pesar de ello, el uso de pruebas unitarias promueve este tipo de desarrollo a través de su independencia de ambiente de desarrollo, permitiendo así validar el código realizado haciendo uso de herramientas de simulación. Por lo tanto, es necesario el diseño de una metodología que describa el uso de pruebas unitarias en sistemas embebidos, de manera que se aborden todos los aspectos que implica su uso y los desafíos que se presentan en su desarrollo.

En la presente tesis se desarrollará el diseño de una metodología de pruebas unitarias en sistemas embebidos, la cual permitirá aumentar su confiabilidad y seguridad y permitirá el uso de desarrollo colaborativo a nivel de software en este tipo de sistemas.

1.4. Objetivos

1.4.1. Objetivo general

Diseñar una metodología para implementar pruebas unitarias en sistemas embebidos con el fin de que sirva como guía para sus desarrolladores a nivel de software.

1.4.2. Objetivos específicos

- Revisar las metodologías existentes para realizar pruebas unitarias en sistemas embebidos.
- Implementar pruebas unitarias en sistemas embebidos para identificar las limitaciones y desafíos que implica.
- Identificar las herramientas de simulación de hardware existentes para su uso en pruebas unitarias.
- Evaluar y validar la metodología propuesta en diferentes softwares embebidos.

Capítulo II

Marco teórico

En este capítulo se definirán los fundamentos necesarios para el diseño de la metodología. El objetivo de este capítulo es examinar y revisar los conceptos que se deben tener en cuenta a la hora de realizar pruebas unitarias en sistemas embebidos. Se verán los diversos retos que esta implica, limitaciones y técnicas y enfoques que son actualmente usados para asegurar la fiabilidad de un sistema embebido. Primero se verán fundamentos de los sistemas embebidos, seguido de definiciones de pruebas unitarias y su relación con otro tipo de pruebas, así como las herramientas más usadas recientemente por los desarrolladores de sistemas embebidos.

2.1. Fundamentos de los Sistemas Embebidos

2.1.1. Definición y características de los sistemas embebidos

Para poder definir a un sistema embebido, primero es importante tener en cuenta la definición de lo que es un sistema. Esta se puede definir como un conjunto de componentes que se combinan e interrelacionan para realizar de forma coordinada una tarea compleja (Valvano, 2014). En el campo de la ingeniería electrónica, estos componentes se pueden entender como el software, el hardware, los circuitos analógicos, suministros de energía, sensores, etc. Los sistemas en general tienen una estructura, comportamiento e interconectividad que funcionan

en un marco de trabajo sujeto a sus requerimientos, normas o reglamentos (Valvano, 2014). Actualmente no existe una definición formal para los sistemas embebidos, estos tienen diversas definiciones elaboradas por diferentes desarrolladores. Por ejemplo, Marwedel define a los sistemas embebidos con la siguiente cita (Marwedel, 2021): "Los sistemas embebidos son sistemas de procesamiento de la información integrados en productos". En base a esta definición y a la definición de sistema, podemos definir a un sistema embebido como un sistema que cumple ciertas funciones específicas dentro un sistema de mayor complejidad, ya sean dispositivos o máquinas. Este tipo de sistemas se caracteriza por su interacción con procesos físicos, por ello se les relaciona con el concepto de sistemas ciber físicos (Marwedel, 2021). Algo que también caracteriza a los sistemas embebidos es que cuentan con un sistema informático integrado, el cual incluye un microordenador con dispositivos mecánicos, químicos y eléctricos conectados, programados de tal manera que pueda cumplir con su función determinada (Valvano, 2014). En el contexto de los sistemas embebidos, a este tipo de software informático se le llama firmware o software embebido, esto con el fin de diferenciarlos del software de aplicación de alto nivel. Entre los lenguajes de programación más usados para el desarrollo de firmware están Assembly, C, C++ y Java (Gu, 2016). Cabe resaltar que los sistemas embebidos tienen limitaciones de tiempo necesarias que se deben tomar en cuenta en el desarrollo para realizar una tarea o actividad (Sangiovanni-Vincentelli, Zeng, Di, y Marwedel, 2014), esto hace que podamos definir a los sistemas embebidos también como sistemas de tiempo real.

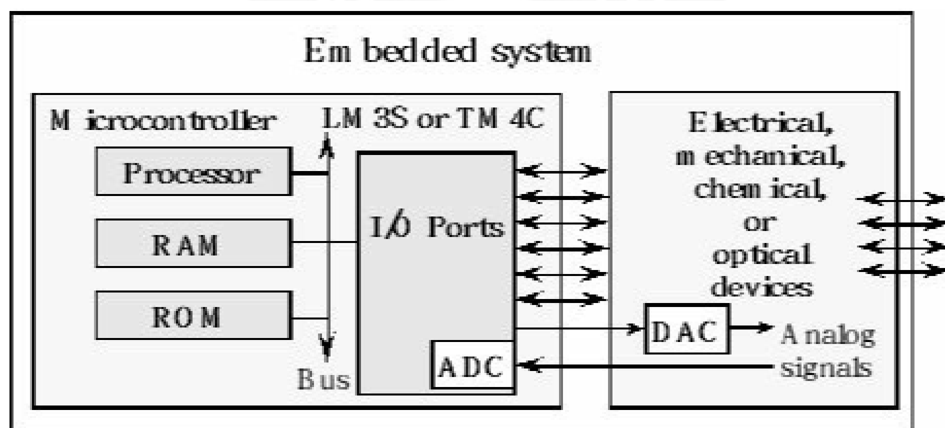


Figura 2.1: Ejemplo de conexión entre microcomputadora y componentes físicos en un sistema embebido (Valvano, 2014)

En las siguientes subsecciones se definirán más conceptos que se deben tomar en cuenta para el desarrollo de sistemas embebidos, así como de su software integrado.

2.1.2. Aplicaciones y desafíos en el desarrollo de sistemas embebidos

Hoy en día, los sistemas embebidos tienen un sinfín de aplicaciones, entre ellas en coches, trenes, aviones, dispositivos electrónicos de uso cotidiano, equipos de telecomunicaciones, entre otros. Recientemente se han previsto aplicaciones de sistemas en conjunto con el concepto de Internet de las Cosas (IoT), teniendo así oportunidades de aplicación en diversas áreas, tales como en la ingeniería mecánica, en robótica, en ingeniería de potencia, en ingeniería civil, en recuperación de desastres, en edificaciones inteligentes, en ingeniería agrícola, en el sector médico e ingeniería médica, en experimentos científicos, en seguridad pública, en aplicaciones militares, en telecomunicaciones y en electrónica de consumo (Marwedel, 2021). Uno de los principales desafíos en el desarrollo de los sistemas embebidos, es la interacción que estos sistemas deben tener con procesos físicos, específicamente la interacción entre el software y el hardware. Sin embargo, últimamente en el desarrollo de sistemas embebidos y su integración con sistemas IoT, los desarrolladores se han encontrado con más desafíos, entre ellos aspectos de seguridad, confidencialidad, fiabilidad, reparabilidad y disponibilidad (Marwedel, 2021).

2.1.3. Complejidad y problemas de diseño de sistemas embebidos

Los sistemas embebidos cada vez son más complejos, esto debido a que tienen más funciones o que, en algunos casos, deben realizar tareas críticas para la seguridad (Sangiovanni-Vincentelli y cols., 2014). Esta complejidad se debe, en su mayoría, a las consideraciones que se deben tener en cuenta en el desarrollo del software embebido del sistema. Su interacción con el entorno físico hace que su sistema informático se diferencie con respecto a otros sistemas informáticos que están orientados únicamente al procesamiento de datos y al software (Sangiovanni-Vincentelli y cols., 2014). Aunque no existe un procedimiento estandarizado para

validar la interacción entre el software y el hardware de los sistemas embebidos, existen metodologías de pruebas y de desarrollo de software embebido que permiten la detección temprana de fallas en la interacción.

Figura 2.2: Ejemplo de aplicación de sistemas embebidos en el campo de la róbotica (Marwedel, 2021)

2.1.4. Arquitectura y componentes de los sistemas embebidos

Los sistemas embebidos tienen al menos una capa arquitectónica que incluya software y hardware, estas tienen como mínimo 4 módulos: Procesamiento de la interfaz de usuario, Procesamiento de la entrada del sistema, Procesamiento de la salida del sistema y Unidades principales de proceso y control (Mirtalebi, 2017).

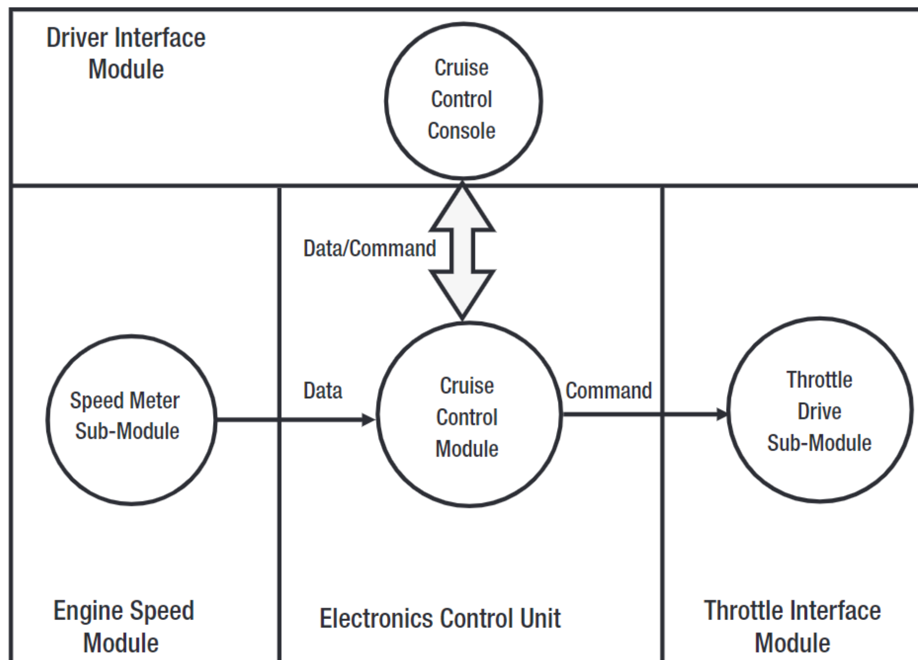


Figura 2.3: Ejemplo de arquitectura de sistema embebido (Mirtalebi, 2017)

Estos módulos principalmente pueden ser componentes de software y hardware, los cuales se pueden agrupar también para formar módulos. Los sistemas embebidos tienen dos funcionalidades principales: funcionalidades basadas en controles y funcionalidades basadas en datos, una arquitectura eficiente separa ambas funcionalidades por capas de componentes o módulos (Mirtalebi, 2017). La mayoría de componentes electrónicos de un sistemas embebido se pueden dividir en las siguientes categorías (Gu, 2016):

- Controlador (una CPU, una FPGA o un híbrido entre las dos)
- Memoria (incluyendo volátil y no volátil)
- Periféricos
- Reloj/reinicio y unidad de gestión de energía

Para transportar información a través de buses de datos y control, es necesario el uso de interfaces. Entre los estándares de bus de datos más populares tenemos: PCI, USB, RS-232, I2C y SPI (Gu, 2016).

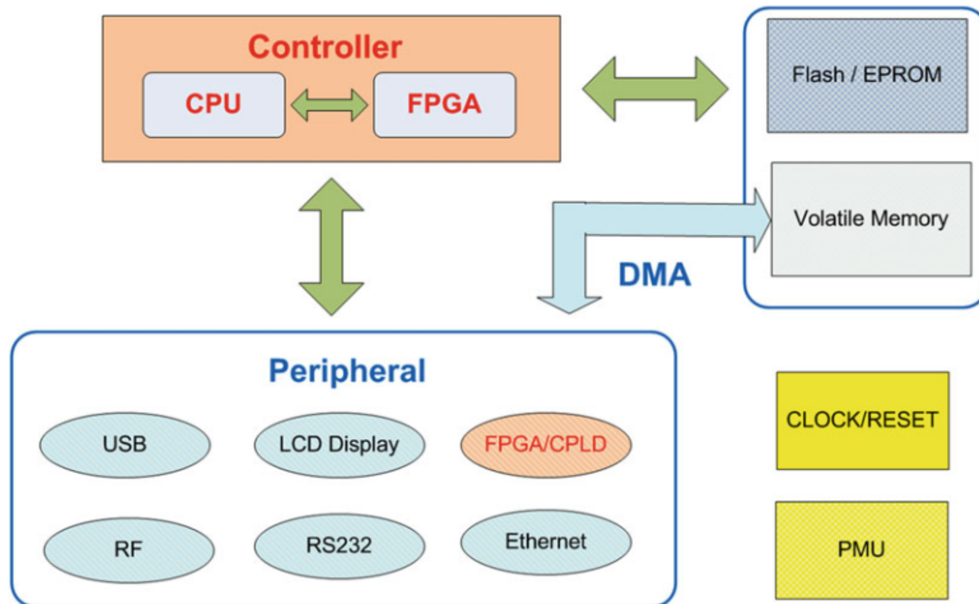


Figura 2.4: Ejemplo de hardware incluído en un sistema embebido (Gu, 2016)

2.2. Fundamentos de las Pruebas Unitarias

2.2.1. Definición de las pruebas unitarias

Para el desarrollo de una metodología para realizar pruebas unitarias en sistemas embebidos, primero se debe definir qué son las pruebas unitarias. Estas según la norma IEEE Standard for Software Unit Testing ("IEEE Standard for Software Unit Testing", 1986), se definen como un proceso compuesto por una jerarquía de principalmente 3 fases, cada una incluyendo sus actividades y tareas correspondientes:

1. Realización de la planificación de las pruebas

    a) Planificar el enfoque general, los recursos y el calendario

    b) Determinar las características que se van a probar

    c) Perfeccionar el plan general

2. Adquisición del conjunto de pruebas

    a) Diseñar el conjunto de pruebas

b) Aplicar el plan y el diseño perfeccionados

3. Medición de las unidades de prueba con respecto a sus requisitos

   a) Ejecutar los procedimientos de prueba

   b) Comprobación de la terminación

   c) Evaluar el esfuerzo y la unidad de prueba

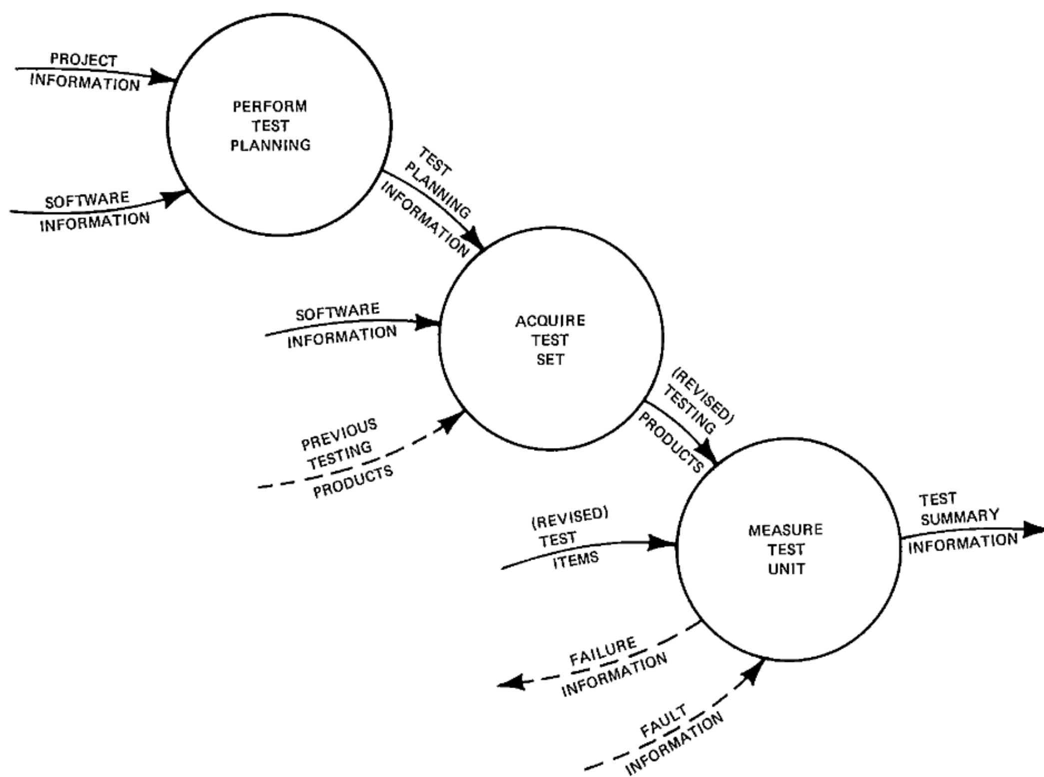


Figura 2.5: Flujo de datos entre las fases de pruebas unitarias de software ("IEEE Standard for Software Unit Testing", 1986)

2.2.2. Características de las pruebas unitarias

Cada una de las fases en el proceso de pruebas implica ciertos aspectos para asegurar la fiabilidad del desarrollo del software. En la etapa de medición, se evalúa que cada unidad cumpla

con cada uno de los requisitos documentados para la misma a partir de los datos de muestra obtenidos de cada prueba (“IEEE Standard for Software Unit Testing”, 1986). Cabe resaltar que las unidades se pueden definir como piezas pequeñas de código (Torkar, Mankefors, Hansson, y Jonsson, 2003) o como métodos, clases o funciones individuales (Dooley, 2017). El proceso de desarrollo de pruebas unitarias no solo termina en la etapa de medición, sino que las muestras de prueba de cada unidad ayudan a ejercitar la unidad de modo que cumpla con su comportamiento requerido. Cabe resaltar que la información de requisitos de cada una de las unidades son diseñados tomando en cuenta la futura integración de las mismas. Otra forma de separar el proceso de las pruebas unitarias, es a través de actividades, las cuales cada una de ellas deben ser llevadas a cabo para su verificación (“IEEE Standard for Software Unit Testing”, 1986). Cabe resaltar que aplicar pruebas unitarias al software, no necesariamente asegura la calidad del código, pero sí permite seguir una metodología ágil de organización del software (Gren y Antinyan, 2017).

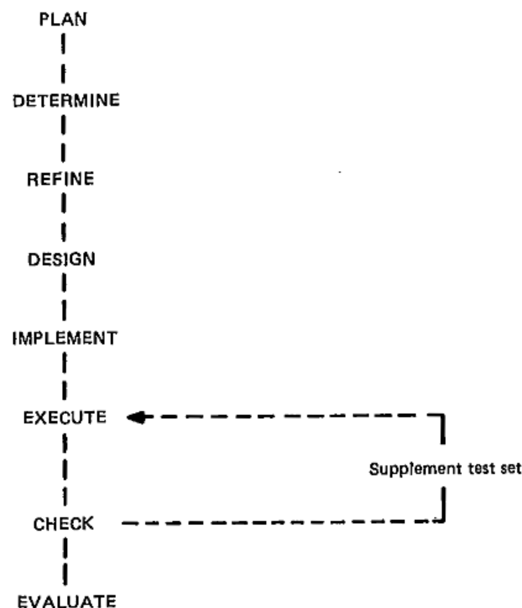


Figura 2.6: Actividades de las pruebas unitarias (“IEEE Standard for Software Unit Testing”, 1986)

2.2.3. Aspectos para tomar en cuenta en la implementación de pruebas unitarias

Con respecto a la implementación de las pruebas unitarias, estas se pueden implementar en cualquier software informático digital o firmware, la norma IEEE no especifica ninguna clase de software o firmware en específico para someter las pruebas. También, la norma de realización de pruebas unitarias es aplicable si el probador de unidades es el desarrollador del software o no ("IEEE Standard for Software Unit Testing", 1986). Con respecto a la programación de las pruebas, estas normalmente se escriben en el mismo lenguaje de programación que se utiliza para crear el software (Torkar y cols., 2003), a pesar de que esto no está especificado en alguna norma. En el caso de que las pruebas unitarias las implemente el desarrollador, este ya tiene un entendimiento claro del funcionamiento del software, ya que conoce las clases, métodos, variables y sus tipos de retorno. A este enfoque de pruebas se le llama pruebas de caja blanca (Dooley, 2017). Las pruebas unitarias también pueden ser parametrizadas, de modo que se instancian las pruebas con determinados conjuntos de argumentos. Estas permiten separar dos tareas de prueba: la especificación de comportamientos externos de caja negra por parte de los desarrolladores y la generación y selección de entradas de prueba internas de caja blanca por parte de las herramientas (Xie, Tillmann, y Lakshman, 2016).

La implementación de pruebas unitarias en software embebido no necesariamente tiene que ser implementada en el sistema embebido. Actualmente existen herramientas que permiten ejecutar las pruebas en simuladores y emuladores de componentes en el caso de que no se tenga acceso directo al hardware. Adicionalmente, el uso de simuladores y emuladores reduce el tiempo de ejecución de pruebas. Es por ello que es importante que al momento de realizar la ejecución de pruebas se defina el ambiente de pruebas. Estas pueden ser, en un ambiente local como en la computadora que se desarrolla el software embebido o en el ambiente del sistema embebido. Finalizando el proceso de las pruebas unitarias, la norma ANSI/IEEE Std 829-1983 requiere la preparación de dos documentos, la Especificación de Diseño de Pruebas y el Informe de Resumen de Pruebas ("IEEE Standard for Software Unit Testing", 1986).

2.2.4. Clasificación de las pruebas unitarias

Las pruebas unitarias pueden clasificarse en pruebas de análisis estático y pruebas de análisis dinámico. El análisis estático suele comprobar los errores sintácticos y semánticos, mientras que el análisis dinámico comprueba el comportamiento de entrada y salida y la lógica interna del subsistema u objeto (N. Zhang, Bao, y Ding, 2009). En el caso del software embebido, es preferible usar simuladores en el análisis dinámico, ya que no siempre se tiene acceso al hardware que interactúa con el mismo.

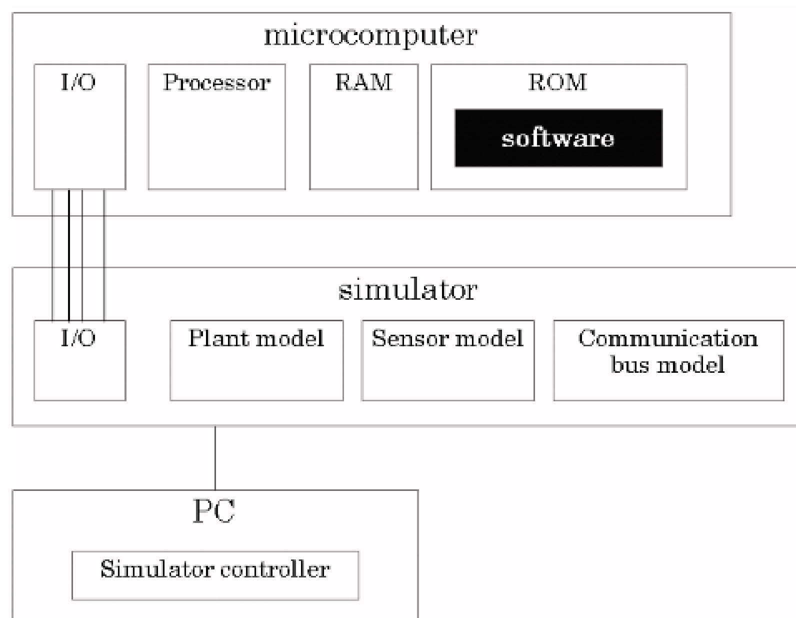


Figura 2.7: Ejemplo de uso simuladores en ambiente de pruebas en un sistema embebido (Mizoguchi y cols., 2020)

2.3. Modelo V de desarrollo de software

Las pruebas funcionales desempeñan un rol importante en el desarrollo de software, ya que ayudan a garantizar su correcto funcionamiento. En el contexto de los sistemas embebidos, las pruebas ayudan a verificar que los componentes a nivel de software y hardware cumplan con sus requerimientos especificados. Las pruebas funcionales de un sistema embebido tienen un enfoque de caja negra, ya que solo se evalúan salidas a partir de entradas sin tener un conocimiento

del funcionamiento interno del sistema (Bajer, Szlagor, y Wrzesniak, 2015). La realización de pruebas funcionales permiten identificar fallas o errores de funcionamiento en un sistema embebido; sin embargo, realizar modificaciones al software a partir de los resultados de las pruebas suele ser complejo. La mayor parte del software embebido se desarrolla siguiendo el proceso denominado modelo V (Mizoguchi y cols., 2020). Las pruebas constan de pruebas unitarias, pruebas de integración y pruebas de sistema. En este modelo de desarrollo los probadores de software participan desde la especificación de los requisitos (Bajer y cols., 2015).

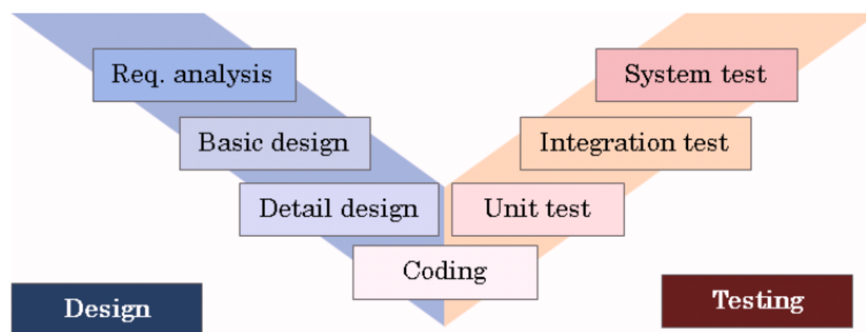


Figura 2.8: Proceso de desarrollo de software llamado Modelo V (Mizoguchi y cols., 2020)

Ya habiendo definido a las pruebas unitarias en la anterior sección, se explicará su relación con las pruebas de integración. Este tipo de pruebas se centra en la evaluación de las interacciones entre las versiones integradas de las unidades (Brar y Kaur, 2015). Cabe resaltar que en el caso de los sistemas embebidos, las pruebas de integración no solo sirven para verificar la integración de componentes de software, sino que también las interfaces software-hardware (Mizoguchi y cols., 2020). A partir de una correcta validación de las pruebas de integración, se pueden realizar las pruebas de sistema para validar el correcto funcionamiento del sistema embebido.

2.4. Metodologías de desarrollo de software embebido para realizar pruebas unitarias

Como se ha evidenciado en el estado del arte, existen diferentes metodologías de desarrollo de software que permiten implementar pruebas unitarias en software embebido. Algunas metodologías son más empleadas que otras, dependiendo del desarrollador y la complejidad del sistema embebido. Lo que tienen en común estas metodologías es que permiten separar al software embebido en unidades para poder realizar las pruebas de cada una y consecuentemente integrarlas. A continuación se mostrarán algunas metodologías de desarrollo de software más usadas en los últimos años:

2.4.1. Ejecución de pruebas simbólicas

La ejecución simbólica fue una metodología propuesta por primera vez en el año 1976 por James C. King. En ese entonces aún no era una técnica práctica debido a la limitación de recursos informáticos. Actualmente, investigadores y desarrolladores la usan para la generación automática de casos de prueba (C. Zhang y cols., 2018). El objetivo de la ejecución simbólica es generar una condición de trayectoria que especifique valores de entrada concretos para los que la ejecución real del programa tenga la misma trayectoria. Esta condición de ruta, la cual trabaja con símbolos abstractos y no valores concretos, se refina cada vez que la ejecución simbólica alcanza una rama en el flujo de control del programa que depende del estado simbólico, permitiendo así acumular condiciones de trayectoria de manera que el flujo del programa se realice de forma satisfactoria (de Boer y Bonsangue, 2019). Esta metodología permite hacer que las pruebas unitarias sean satisfactorias a partir de la condición de ruta definida en la ejecución simbólica.

2.4.2. Test Driven Development

El desarrollo basado en pruebas o mejor conocido como TDD por sus siglas en inglés, es una de las metodologías de desarrollo software que permite realizar un desarrollo ágil. Esta metodología se introdujo desde la década de 1960 con el Proyecto Mercury de la NASA, y redescubierto y recibido su nombre actual después de ser introducido como una práctica en eXtreme Programming (XP) (Demeyer, Verhaeghe, Etien, Anquetil, y Ducasse, 2018). El desarrollo basado en pruebas consiste en desarrollar y ejecutar pruebas antes del desarrollo real del software. En esta metodología, se desarrollan los posibles casos de prueba para cada pequeña funcionalidad del sistema, de manera que estos casos de prueba guíen el desarrollo del código y su futura validación (Abushama, Alassam, y Elhaj, 2021). Cabe resaltar que en este enfoque de desarrollo, cuando se definen nuevas pruebas, no necesariamente estas han sido cubiertas en el desarrollo del software en base a las pruebas anteriores (Demeyer y cols., 2018).

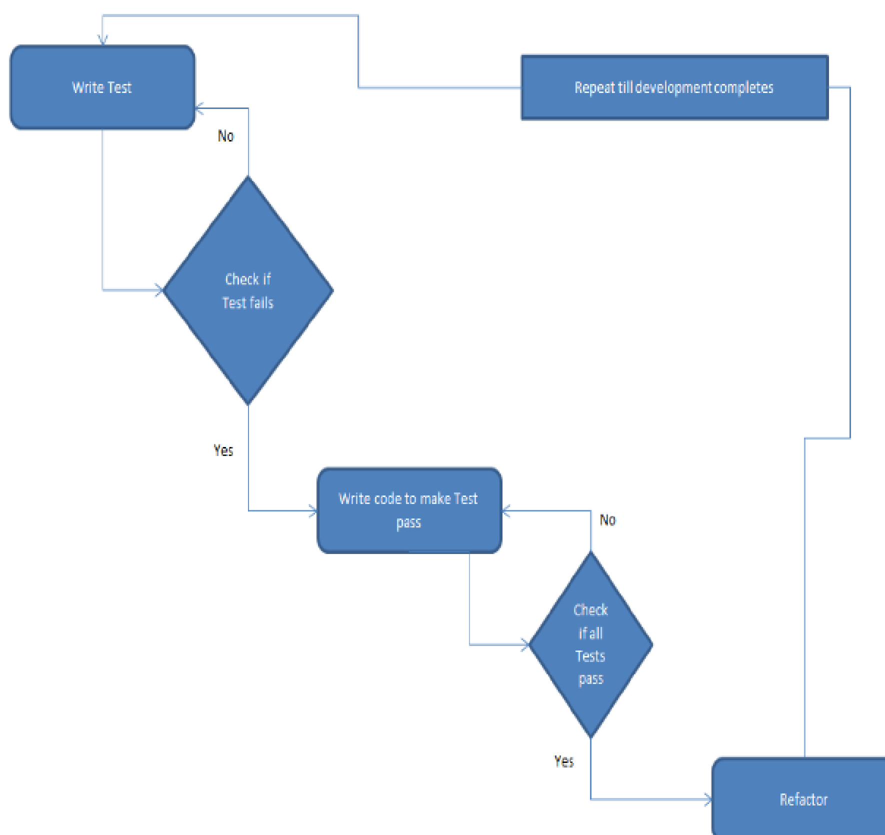


Figura 2.9: Flujo de trabajo de la metodología Test Driven Development (Lenka y cols., 2018)

2.4.3. Behavior Driven Development

El desarrollo basado en descripción de comportamiento o BDD por sus siglas en inglés, es una extensión de la metodología basada en pruebas (TDD) que se denomina como una metodología de desarrollo de software en la que la especificación y diseño de una aplicación se realizan mediante la descripción del comportamiento que un observador externo debe observar (Lenka y cols., 2018). Esta metodología trabaja en torno a las expectativas y escenarios del usuario, las cuales son reflejadas en forma de escenarios de pruebas. Estos escenarios deben obtener el consentimiento de las partes interesadas en el desarrollo del software, de manera que después se automaticen para facilitar el proceso de prueba. Esta metodología obliga a un equipo ágil de desarrollo de software a mirar la aplicación del programa desde la perspectiva del usuario final (Abushama y cols., 2021).

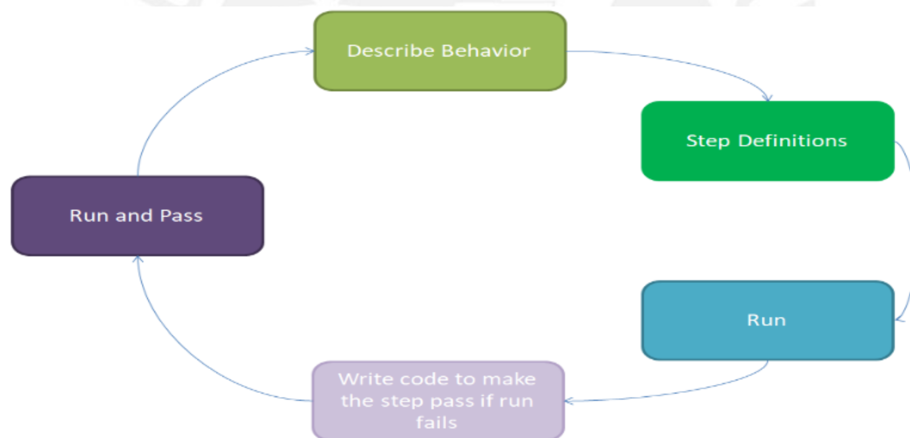


Figura 2.10: Flujo de trabajo basado en la metodología Behavior Driven Development (Lenka y cols., 2018)

2.4.4. Component Based Software Development

El desarrollo de software basado en componentes o mejor conocido como CBSD por sus siglas en inglés, se caracteriza por una de las metodologías ágiles de desarrollo de software debido a que no requiere la necesidad de desarrollar software desde cero, ahorra tiempo y es fácilmente utilizable (Mohan y Jha, 2019). Esta metodología de desarrollo se centra en la reutilización de

componentes ya existentes en función de los requisitos definidos por el usuario. Estos componentes se personalizan y se integran sin codificación explícita para formar una aplicación. El enfoque individual en los componentes y el uso de componentes probados existentes mejorará la calidad y fiabilidad general del producto final (Singh, Kumar Tiwari, y Kumar, 2020). En el método tradicional de desarrollo de software, el software es construido desde cero, en la metodología CBSD se puede utilizar un componente ya construido y ensamblarlo en el software de desarrollo, ahorrando así tiempo de desarrollo, aumentando reutilización, dando flexibilidad al código, entre otras ventajas (Mohan y Jha, 2019).

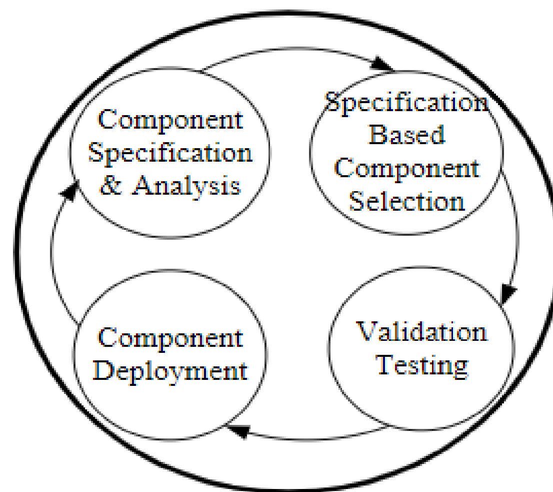


Figura 2.11: Selección de componentes en desarrollo de software basado en la metodología Component Based Development (Singh y cols., 2020)

2.4.5. Model Based Development

El desarrollo basado en modelos o MBD por sus siglas en inglés, es una metodología de desarrollo utilizada ampliamente en desarrollo de software aeronáutico por las ventajas de tiempo y coste que proporciona a las empresas del rubro. En esta metodología, los problemas relacionados con el modelo pueden identificarse al principio de la fase de desarrollo, ya que el modelo se verifica en una fase de definición de requisitos. También pueden construir, verificar y analizar sus diseños antes de escribir ningún código (Saraç, 2019). La combinación de componentes

y parámetros necesarios para satisfacer los requisitos del modelo se suelen verificar mediante simulaciones utilizando bloques de simulación (Wakitani y Yamamoto, 2017). El desarrollo tradicional de software suele tener tres niveles de requisitos: requisitos del sistema asignados al software, requisitos de alto nivel del software y requisitos de bajo nivel del software. En este tipo de desarrollo tradicional suelen haber redundancias e incoherencias entre estos niveles; en la metodología MBD, el uso de herramientas de modelado cualificadas tiene potencial de reducir estas redundancias e incoherencias (Saraç, 2019).

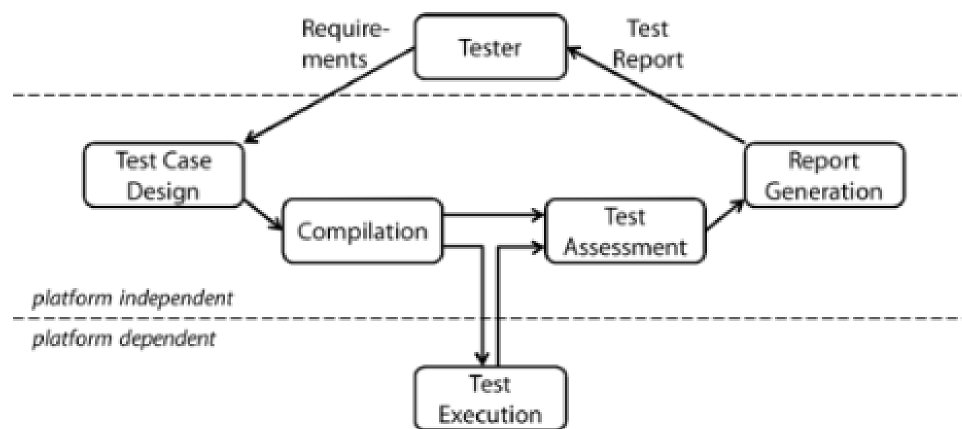


Figura 2.12: Flujo de trabajo de la metodología Model Based Development (Shaout y Pattela, 2021)

2.4.6. Equivalence Partitioning

El particionamiento de equivalencias es una de las técnicas de prueba de caja negra que permiten crear varias condiciones basadas en casos de prueba obtenidos (Siahaan, Kusumawardani, y Hidayah, 2022). Esta metodología se basa en que, si un programa específico funciona adecuadamente para un determinado valor, entonces aparentemente funcionará correctamente para todos los demás valores de esa clase. La idea fundamental de esta metodología es que funciona correctamente para todos los valores o para ninguno (Bhat y Quadri, 2015). Los desarrolladores de software que emplean particionamiento de equivalencias organizan escenarios de

prueba para cada característica definida en los requerimientos creando particiones para objeto, clase o función que se va a probar (Siahaan y cols., 2022).

2.5. Herramientas y tecnologías para realizar pruebas unitarias en sistemas embebidos

2.5.1. Entornos de desarrollo de software embebido

Para definir las funcionalidades que debe seguir un sistema embebido, es necesario el desarrollo de su software embebido a partir de los requerimientos establecidos para el sistema. En este desarrollo se definen las interaciones entre el software y hardware que va a tener el sistema embebido; sin embargo, estas configuraciones en algunos casos pueden traer ciertos problemas o errores en el funcionamiento general del sistema. Existen herramientas que permiten la elaboración de software embebido mediante editores de texto, creación de proyectos y analizadores de sintáxsis de código. A continuación, se presentarán entornos de desarrollo mayormente usados por desarrolladores de sistemas embebidos:

Code Composer Studio

Code Composer Studio es un entorno de desarrollo integrado para microcontroladores y procesadores de Texas Instruments. Incluye un conjunto de herramientas para el desarrollo y depuración de aplicaciones integradas. Este entorno de desarrollo incluye un compilador de C/C++ optimizado, editor de código fuente, entorno de creación de proyectos, un depurador, entre muchas otras funciones (*CCSTUDIO IDE, configuration, compiler or debugger | TI.com*, s.f.).

Keil µVision

Keil µVision es una plataforma de desarrollo de software basada en ventanas. Este entorno de desarrollo integra todas las herramientas necesarias para el desarrollo de software embebido, incluido un compilador de C/C++, un ensamblador de macros, un enlazador, localizador y un generador de archivos HEX. µVision también ofrece un modo de compilación para crear aplicaciones y un modo de depuración. Cabe resaltar que las aplicaciones pueden depurarse en un simulador µVision integrado o directamente en el hardware a través de algún adaptador (*µVision User's Guide*, s.f.).

PlatformIO IDE for Visual Studio Code

PlatformIO es un entorno de desarrollo integrado fácil de usar y ampliable con un conjunto de instrumentos de desarrollo profesionales que ofrece funciones modernas y potentes para acelerar y simplificar la creación y entrega de productos embebidos. Esta herramienta ofrece un editor de código fuente multiplataforma, completaciones de código inteligentes basadas en tipos de variables, definiciones de funciones y dependencias de bibliotecas, asistente de proyecto intuitivo, monitor serial, analizador de código estático, debugger, entre otros beneficios. Cabe resaltar que esta herramienta es una extensión descargable para el editor de código Visual Studio Code, por lo que PlatformIO tiene compatibilidad con todas las extensiones que esta incluye (*A professional collaborative platform for embedded development · PlatformIO*, s.f.).

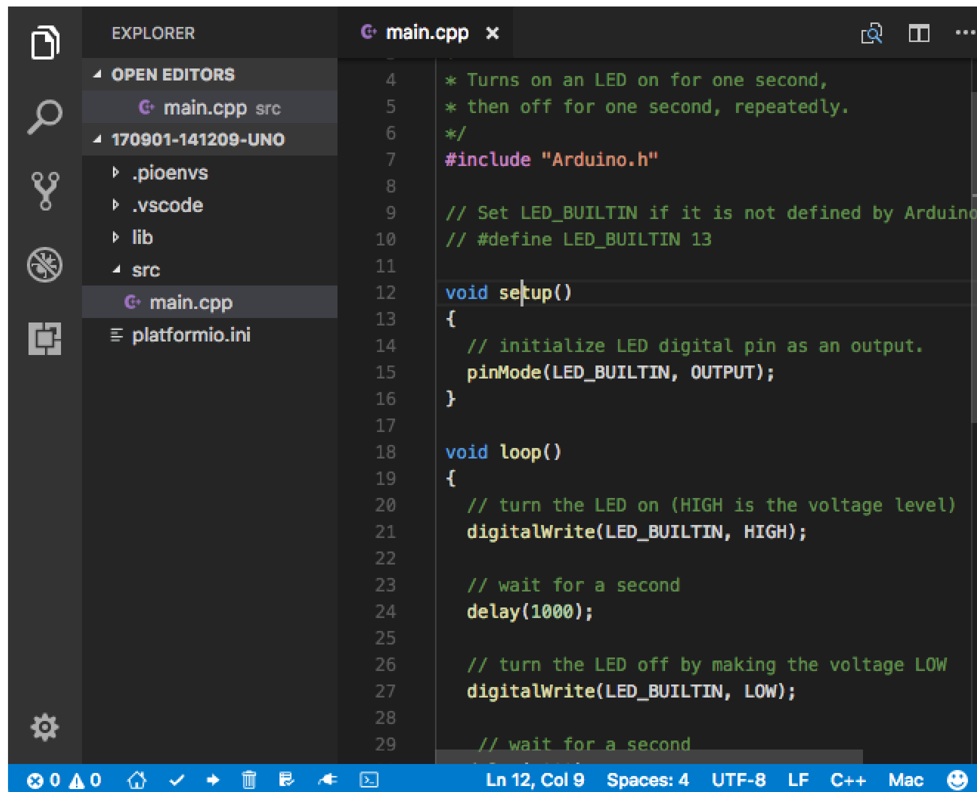


Figura 2.13: Implementación de software embebido en el IDE PlatformIO (*A professional collaborative platform for embedded development · PlatformIO*, s.f.)

2.5.2. Frameworks de desarrollo de software embebido

En programación embebida, una de las herramientas fundamentales para el desarrollo es el uso de frameworks, los cuales permiten realizar operaciones de configuración del microcontrolador de una manera más accesible y comprensible. Estos permiten que la programación embebida se trabaje en un nivel más alto, ya que abstrae la complejidad de trabajar con la interacción directa con el hardware y simplifica la configuración de registros internos.

Actualmente existen diferentes tipos de frameworks diseñados para distintos microcontroladores de diversas compañías de desarrollo; sin embargo, uno de los más utilizados es el framework de Arduino. En el ámbito educativo, Arduino ha tenido un impacto exponencial debido a su versatilidad en diseño y experimentación (García-Tudela y Marín-Marín, 2023). Actualmente existen un sinfín de librerías y ejemplos de implementaciones de programación embebida utilizando Arduino para diferentes módulos electrónicos, facilitando así el desarrollo. Además de

ser una herramienta fudndamental en el diseño de circuitos electrónicos (Maity y cols., 2023), el framework de Arduino es compatible con una gran variedad de microcontroladores de diferentes fabricantes. Por ello, este trabajo de investigación se centrará en la realización de pruebas unitarias a software embebido empleando el framework de Arduino.

2.5.3. Frameworks para pruebas unitarias, simuladores y emuladores

Para poder implementar pruebas unitarias en sistemas embebidos, existen diferentes herramientas desarrolladas para diferentes lenguajes de programación y aplicaciones. Estas herramientas suelen ser denominadas como marcos de trabajo o framewoks, las cuales incluyen funciones que nos permiten diseñar y ejecutar las pruebas en el software embebido. Cabe resaltar que no solo existen herramientas que ayudan a la implementación de pruebas unitarias, sino que existen herramientas adicionales que permiten ejecutar las pruebas sin hacer uso explícito del sistema embebido, mediante simuladores y emuladores que se pueden programar según el comportamiento de los componentes de los cuales no tenemos acceso. A continuación se presentarán algunas herramientas que facilitan el uso de pruebas unitarias en software embebido:

Unity

Unity es un framework que permite realizar pruebas unitarias en sistemas embebidos. Este está escrito en código C, sigue los estándares ANSI y soporta la mayoría de peculiaridades de los compiladores de software embebido. Unity cuenta con un gran conjunto de aserciones para implementar las pruebas según los requerimientos establecidos. Se puede ejecutar en un ejecutable nativo o un simulador (*Unity — Throw The Switch*, s.f.).

CppUTest

CppUTest es un framework de pruebas xUnit basado en C/C++ para pruebas unitarias y testear código. Se suele usar en sistemas embebidos pero también funciona para cualquier pro-

yecto C/C++. Esta herramienta está construido para desarrolladores de software que siguen la metodología de desarrollo basado en pruebas (TDD) (*Cpputest*, s.f.).

Google Test

Google Test es un marco de pruebas desarrollado por el equipo de Testing Technology teniendo en cuenta los requisitos y limitaciones específicos de Google. Esta herramienta admite cualquier tipo de pruebas, no sólo pruebas unitarias. Google Test funciona en diferentes sistemas operativos y con diferentes compiladores, por lo que pueden funcionar con una gran variedad de configuraciones (*GoogleTest Primer | GoogleTest*, s.f.).

```
#include "unity.h"
#include "DumbExample.h"

void test_AverageThreeBytes_should_AverageMidRangeValues(void)
{
TEST_ASSERT_EQUAL_HEX8(40, AverageThreeBytes(30, 40, 50));
TEST_ASSERT_EQUAL_HEX8(40, AverageThreeBytes(10, 70, 40));
TEST_ASSERT_EQUAL_HEX8(33, AverageThreeBytes(33, 33, 33));
}

void test_AverageThreeBytes_should_AverageHighValues(void)
{
TEST_ASSERT_EQUAL_HEX8(80, AverageThreeBytes(70, 80, 90));
TEST_ASSERT_EQUAL_HEX8(127, AverageThreeBytes(127, 127, 127));
TEST_ASSERT_EQUAL_HEX8(84, AverageThreeBytes(0, 126, 126));
}

int main(void)
{
UNITY_BEGIN();
RUN_TEST(test_AverageThreeBytes_should_AverageMidRangeValues);
RUN_TEST(test_AverageThreeBytes_should_AverageHighValues);
return UNITY_END();
}
```

Figura 2.14: Implementación de pruebas unitarias con el framework Unity (*Unity — Throw The Switch*, s.f.)

El uso de simuladores y emuladores nos permite ejecutar pruebas cuando tenemos limitaciones de acceso directo al hardware del sistema embebido. Estas herrramientas también ayudan a detectar posibles fallas, ya que los simuladores y emuladores se pueden programar de tal forma que fallen en la ejecución para asegurarnos de que el software embebido que estemos elaborando sepa actuar ante estos casos. En el contexto de desarrollo de software, los componentes simulados suelen ser llamados Mocks, y su práctica en ejecución de pruebas se le denomina Mocking. A las componentes que suelen ser utilizados temporalmente en la ejecución de las pruebas, también en el contexto de desarrollo de software, se les denomina Stubs. La diferencia entre los Mocks y los Stubs es que los Mocks simulan el comportamiento de componentes u objetos a los que no tenemos acceso, mientras que los Stubs simulan salidas predefinidas para los objetos, los cuales simplifican la futura integración de las pruebas. A continuación se presentarán herramientas que permiten la implementación de estos componentes u objetos simulados y emulados para la ejecución de las pruebas unitarias:

CMock

CMock es una herramienta que genera código C puro a partir de cabeceras C normales. Esta ayuda a especificar qué funciones se esperan que sean llamadas en las pruebas y bajo qué argumentos. CMock permite que las funciones no sean llamadas para algunas pruebas si no es necesario, hace el llamado de funciones personalizadas y puede manejar cualquier tipo de datos. La herramienta CMock genera sus módulos mock a partir de simples cabeceras, ahorrándole la tediosa creación de sus propios stubs (*CMock — Throw The Switch*, s.f.).

FakeIt

FakeIt es un framework que permite el uso de mocking en C++ y C. Tiene soporte con GCC, Clang y MS Visual C++. Su uso es sencillo y simple debido a las herrramientas que incluye. Desde la declaración del mock, hasta su simulación de comportamiento según una entrada específica a la función. (*eranpeer/FakeIt: C++ mocking made easy. A simple yet very expressive, headers only library for c++ mocking.*, s.f.).

ArduinoFake

ArduinoFake es un framework basado en FakeIt que facilita el desarrollo de mocking para las funciones incluidas en el framework de Arduino. Su uso permite que el código desarrollado para un sistema embebido basado en el framework de Arduino, se ejecute de forma nativa sin problemas.(*FabioBatSilva/ArduinoFake: Arduino mocking made easy*, s.f.).

```
1   void test_MyFunc_should_ParseStuffAndCallTheHandlerForNeatFeatures(void)
2   {
3     int retval;
4
5     //We start by saying what our expectations are, and what we want to return
6     NEAT_FEATURES_T ExpectedFeatures = { 1, "NeatStuff" };
7     ParseStuff_ExpectAndReturn("NeatStuff", 1);
8     HandleNeatFeatures_Expect(ExpectedFeatures);
9
10    //Run Actual Function Under Test
11    retval = MyFunc("NeatStuff");
12
13    //We can still verify whatever things we normally would after
14    TEST_ASSERT_EQUAL(1, retval);
15  }
```

Figura 2.15: Implementación de pruebas unitarias empleando objetos simulados mediante la herramienta CMock (*CMock — Throw The Switch*, s.f.)

2.5.4. Herramientas de reporte de cobertura

Para agregar una etapa adicional a la documentación después de la ejecución de las pruebas, se puede emplear el uso de reportes de cobertura. Estas herramientas permiten tener en análisis de las línea de código del programa que fueron ejecutadas en la prueba y cuales no. A partir del porcentaje final obtenido en este reporte, se evalúa la inclusión de más pruebas para llegar a obtener un porcentaje de cobertura cercano al 100 %. A continuación se presentarán herramientas que permiten su uso:

Gcovr

Gcvor es una herramienta utilizada para reporte de resultados de cobertura de código desarrollado con el compilador GCC. Esta herramienta permite la generación del reporte como salida de texto, salida HTML, salida en archivo JSON o CSV .(*gcovr — gcovr 6.0 documentation*, s.f.).

Lcov

Lcov es una herramienta que fue desarrollada a partir de GCOV que provee información sobre las partes del programa con son ejecutadas mientras se corre una prueba en particular. Esta herramienta fue inicialmente diseñada para mediciones de cobertura en el Kernel de Linux; sin embargo, funciona bien para mediciones de cobertura en código desarrolado por el usuario.(*linux-test-project/lcov: LCOV*, s.f.).

Capítulo III

Diseño de la metodología propuesta

Este capítulo se centrará en el diseño de la metodología para realizar pruebas unitarias en sistemas embebidos. A partir de los conceptos definidos en el marco teórico, se utilizarán estrategias de desarrollo de software basadas en metodologías ágiles para asegurar el correcto funcionamiento y validación de los sistemas embebidos. Se usarán diferentes técnicas utilizadas en las diversas metodologías de desarrollo de pruebas unitarias en base a las herramientas que facilitan el desarrollo e implementación de unidades de prueba. El diseño de la metodología para realizar pruebas unitarias en sistemas embebidos se basará en una de las etapas que implica el desarrollo de sistemas embebidos, que es la etapa de pruebas. Esta etapa de pruebas se divide principalmente en 3 etapas: pruebas unitarias, pruebas de integración y pruebas de sistema. En el capítulo anterior, correspondiente al marco teórico, se han analizado diversas metodologías para realizar pruebas unitarias en sistemas embebidos.

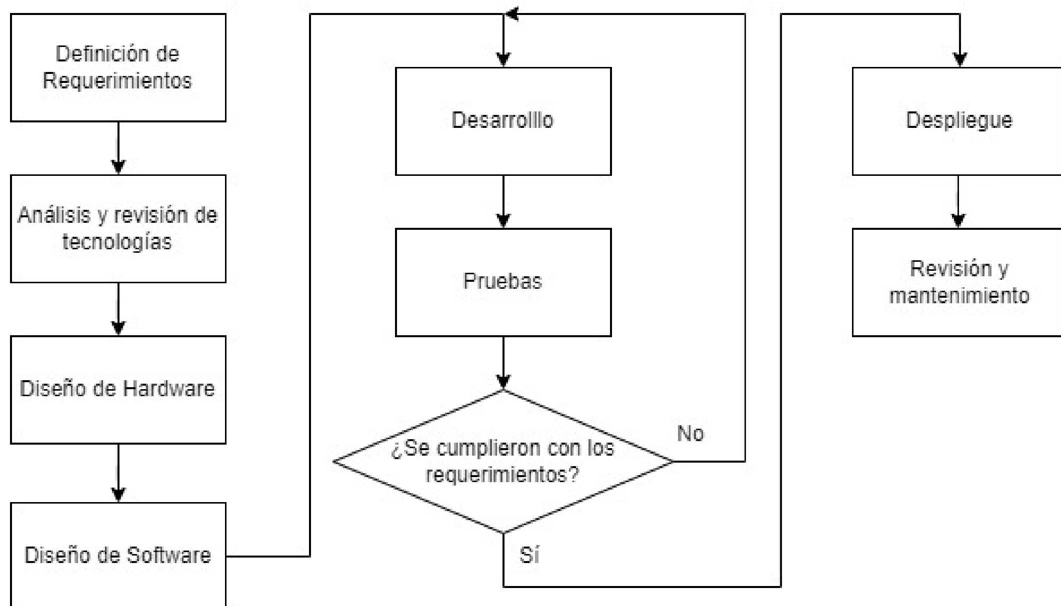


Figura 3.1: Diagrama de flujo de desarrollo de un sistema embebido

El diseño de la metodología se realizará en base al desarrollo de un sistema embebido con aplicaciones en internet de las cosas (IoT) en el área de seguridad como caso de estudio. El objetivo del desarrollo del diseño de la metodología de pruebas unitarias es entender cómo aplicarlas en el concepto de desarrollo de un sistema embebido para así asegurar la confiabilidad del sistema y promover el desarrollo colaborativo. La estructura de la metodología propuesta para el desarrollo de pruebas unitarias en software embebido se presenta en la figura 3.2. Cada una de sus etapas incluye las consideraciones a tomar en cuenta y configuraciones necesarias a nivel de software para su correcta implementación. A lo largo del desarrollo de este capítulo se brindarán más detalles de cada una de estas etapas.

Figura 3.2: Metodología Propuesta

3.1. Definición de requerimientos del sistema

Algo característico de un proyecto que involucre el diseño de un sistema embebido, es que parte de una definición de requerimientos, como se puede apreciar en el diagrama anterior, esta suele ser la primera etapa de desarrollo. Esta etapa de desarrollo involucra una comunicación entre todas las áreas de desarrollo de un proyecto. Para la metodología propuesta de desarrollo de pruebas unitarias en software embebido, de igual manera se deben partir de esos requerimientos. Esto debido a que los requerimientos incluyen la documentación de funcionamiento principal del sistema, así como sus funciones principales, secundarias y otros aspectos a tomar en cuenta. En el caso de que se implemente la metodología propuesta en un sistema que sus requerimientos ya fueron definidos, es necesario un análisis de las especificaciones del mismo.

Tener un claro entendimiento de los requerimientos del sistema nos permite identificar las necesidades del mismo y saber qué funciones son necesarias para realizar pruebas.

3.2. Análisis de diseño del sistema

Como segunda etapa de desarrollo de la metodología propuesta, se tiene la etapa de análisis de diseño del sistema. Esta etapa tiene relación también con la segunda etapa de desarrollo de un sistema embebido, el cual corresponde a la de análisis y revisión de tecnologías. En el caso del desarrollo de un sistema embebido, esta etapa involucra la definición de herramientas a usar para el sistema a nivel de hardware y también una parte de software. Para la metodología propuesta, se parte de un proyecto ya desarrollado previamente a nivel de hardware y software para realizar las pruebas unitarias al sistema; sin embargo, también podría desarrollarse la metodología propuesta en un proyecto que se encuentra en una etapa inicial de desarrollo. Cabe resaltar que para ello, se debe considerar una revisión previa de componentes de hardware que permitan cumplir con los requerimientos establecidos para el sistema y a nivel de software se puede seguir la metodología TDD de desarrollo, ya que esta se basa en pruebas. Para seguir con el diseño de la metodología en esta segunda etapa, considerando que se aplicará en un sistema embebido ya desarrollado a nivel de hardware y software, es necesario tener un entendimiento

de cada uno de los módulos del sistema a nivel de hardware y cómo se comunican entre sí. Esto nos permitirá conocer las interacciones directas entre hardware y software en el sistema embebido para posteriormente identificarlas cuando se vayan a realizar las pruebas a nivel de software. En las siguientes etapas de la metodología propuesta se presentará la relevancia de este análisis de sistema en el desarrollo de las pruebas unitarias. El desarrollo de un diagrama de bloques brinda una visión general de las interacciones entre hardware y software que tendrá como mínimo el proyecto, por lo que darle una revisión es conveniente en esta etapa.

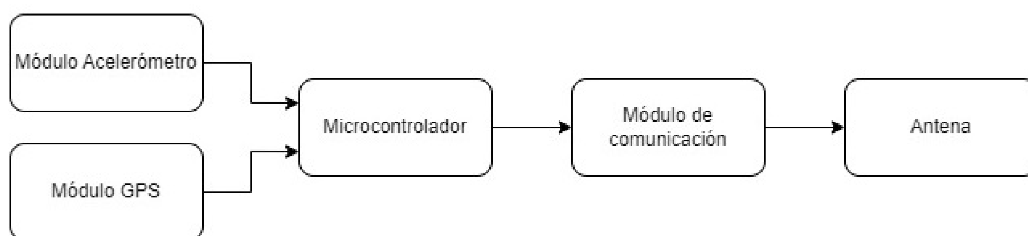


Figura 3.3: Diagrama de bloques del sistema usado para el diseño de la metodología

3.3. Planificación y Diseño de las pruebas unitarias

Esta etapa de definición de pruebas corresponde a la tercera de la metodología propuesta. Se centra en las consideraciones y herramientas a utilizar para la definición de las pruebas unitarias. Se incluyen uso de metodologías de desarrollo de software, tipo de enfoque de pruebas, ambiente de pruebas y consideraciones a partir del estándar IEEE para las pruebas unitarias de software.

Para el presente trabajo y diseño de la metodología, como fue mencionado anteriormente, se desarrollará esta etapa en base a un proyecto de software ya desarrollado previamente. En el caso de que se desarrolle en un proyecto en su etapa inicial, se pueden usar las mismas herramientas y metodologías descritas en esta etapa.

El enfoque de pruebas que usará esta metodología será el enfoque de pruebas por caja blanca, ya que permite tener un entendimiento de la lógica interna de los componentes del sistema al momento de realizar las pruebas unitarias. Al usar este enfoque de pruebas, también se tendrá

conocimiento de las funciones y métodos utilizados para el proyecto a nivel de software embebido. Estas funciones y métodos principalmente suelen incluir las interacciones directas entre el hardware y software, envío y recepción de datos seriales, cálculos matemáticos a partir de los datos leídos, entre otras funcionalidades características del propio sistema.

Como metodologías de desarrollo de software embebido, la metodología propuesta hace uso de la metodología TDD (Desarrollo basado en pruebas) y la metodología CBD (Desarrollo basado en componentes). Al usar la metodología CBD se hará provecho de la etapa de segmentación de software que esta incluye, además de la reutilización de componentes de software previamente desarrollados y probados. Esta división por componentes de software permite definir de una forma más eficiente las pruebas unitarias, ya que se elimina una capa de dependencia a nivel de software entre los componentes del sistema. Para la definición de las unidades de prueba se parte de cada una de las funciones o métodos que incluye cada componente del sistema. Separar archivos de código por cada componente del sistema, permite realizar la segmentación característica de la metodología CBD. Cabe resaltar que estos componentes de software no necesariamente están relacionados directamente a cada uno de los componentes a nivel de hardware del sistema, se pueden definir como componentes de software también componentes de procesamiento de datos, componentes lógicos del proyecto, componentes de codificación y decodificación, entre otros. Por otro lado, al usar la metodología TDD, se facilita el desarrollo de las pruebas debido a que las pruebas y el software se desarrollan a la par. En el caso de que se implemente en proyectos que están en una etapa inicial de desarrollo de software embebido, es de las más apropiadas debido a sus características. Emplear esta metodología en el desarrollo de las pruebas, permite que se modifique el software de la unidad de prueba de manera que pase satisfactoriamente la prueba, con esta metodología se pueden identificar errores de análisis estático en el software embebido.

Antes de definir las pruebas unitarias, es necesario tomar en cuenta las consideraciones para el diseño de pruebas unitarias según el estándar IEEE 1008 para pruebas unitarias de software ("IEEE Standard for Software Unit Testing", 1986):

3.3.1. Consideraciones de diseño de pruebas unitarias según el Estándar IEEE 1008 para pruebas unitarias

1. Se deben definir los requerimientos de cada unidad de prueba en base al análisis de requerimientos del sistema. Esto facilitará el diseño de los casos de prueba para la unidad.

2. Se deben seleccionar los elementos que se incluirán en la prueba de cada unidad definida. Es necesario definir los procedimientos, estados, transiciones de estado y características a probar para realizar el diseño de prueba de cada unidad.

3. Se debe analizar la información de planificación de pruebas unitarias, las cuales son desarrolladas en base al entendimiento de la lógica interna de la unidad y los requerimientos definidos para la misma.

4. Se debe tener en cuenta los aspectos de diseño de la unidad, ya que con su entendimiento de la lógica interna, se podrán detectar errores fácilmente.

5. Se debe de definir la arquitectura del conjunto de pruebas basado en las características a probar. Esta arquitectura debe seguir el diseño de un conjunto jerárquicamente descompuesto de objetivos de prueba, de modo que cada objetivo de un nivel más bajo se pueda probar directamente mediante algunos casos de prueba.

6. Se debe definir un procedimiento de pruebas para cada unidad. Esta debe incluir principalmente una combinación entre la información de la planificación de pruebas, la documentación de los requisitos de la unidad (ambas determinadas en base al análisis de requerimientos del sistema) y las especificaciones de los casos de prueba.

7. Se deben definir las especificaciones de los casos de prueba para la unidad. Cabe resaltar que también se podrían definir nuevos casos de prueba a partir de estas especificaciones.

8. Se debe evaluar el aumento del conjunto de especificaciones de los casos de prueba basados en la información del diseño de la unidad de prueba. Esto se debe a las características internas de la unidad, que podrían ser el flujo de control de datos, estructuras de datos internos, algoritmos utilizados, entre otros.

En base al desarrollo de los puntos mencionados anteriormente se deben obtener las especificaciones de diseño de las pruebas unitarias, especificaciones de procedimientos de pruebas separadas, especificaciones de casos de pruebas separadas y solicitudes de mejora de diseño de las unidades definidas. En base a estas especificaciones y solicitudes obtenidas se podrán diseñar las pruebas que permitan validar el correcto funcionamiento del sistema.

Otro aspecto a tomar en cuenta en el diseño de las pruebas unitarias, es que estas deben seguir principalmente las siguiente características ("IEEE Standard for Software Unit Testing", 1986):

3.3.2. Características principales de las pruebas unitarias usadas en la metodología propuesta

1. Las pruebas unitarias deben ser autónomas e independientes.
2. Las pruebas unitarias deben ser simples y entendibles.
3. La ejecución de las pruebas deben realizarse independientemente de la ejecución de otras pruebas.
4. Las pruebas unitarias deben seguir una etapa de documentación de resultados de ejecución.

3.4. Desarrollo de las pruebas unitarias

La siguiente etapa de la metodología propuesta, corresponde al desarrollo de las pruebas unitarias, las cuales se realizó su planificación en base al análisis de diseño de sistema y revisión de requerimientos en las etapas anteriores. Como paso previo al desarrollo de las pruebas unitarias, es importante definir las herramientas a usar para el desarrollo de software. Para la metodología, se propone el uso de la extensión de PlatformIO de Visual Studio Code para el desarrollo del proyecto a nivel de software del sistema. Su uso se debe a que incluye todas las

configuraciones necesarias para el desarrollo de software en la mayoría de microcontroladores compatibles con el framework de Arduino. Además, Visual Studio Code cuenta con una variedad de extensiones que permiten identificar errores en el software a nivel de análisis estático. Cabe resaltar que la extensión de PlatformIO, tiene compatibilidad con el framework de Unity para su uso en el desarrollo y ejecución de las pruebas unitarias del sistema. Un aspecto importante que se debe considerar para el desarrollo de las pruebas unitarias, es el ambiente de desarrollo de pruebas. Como ambiente de desarrollo de pruebas se tienen un ambiente de desarrollo embebido, involucrando el desarrollo directamente en el hardware, y un ambiente de desarrollo local o nativo, el cual hace uso de los recursos de la computadora donde se desarrolle el software embebido. Como fue mencionado en los capítulos anteriores, realizar pruebas unitarias directamente en el sistema, no es la mejor opción para su implementación. Esto debido a que la ejecución de las pruebas es más lenta que en un ambiente nativo y que esta no promueve el desarrollo colaborativo al depender directamente del acceso al hardware del sistema. Por esta razón, se usará un ambiente de prueba nativo o local para el desarrollo de la metodología propuesta.

En el caso de PlatformIO, cuando se crea un proyecto, se crea un archivo platformio.ini que incluye las configuraciones del ambiente de desarrollo del software embebido. Para el proyecto desarrollado del sistema usado para el diseño de la metodología, se tienen las siguientes configuraciones:

```
[env:d1_mini_pro]
platform = espressif32
framework = arduino
board = lolin_d32
lib_deps =
    electroniccats/MPU6050@^0.3.0
    knolleary/PubSubClient@^2.8
    mikalhart/TinyGPSPlus@^1.0.2
```

Figura 3.4: Ambiente de desarrollo declarado para el sistema embebido

Para incluir un ambiente nativo como segundo ambiente de desarrollo en un proyecto de PlatformIO, es necesario declarar las siguientes configuraciones en el archivo platformio.ini:

```
[env:native]
platform = native
build_flags = -std=gnu++17
              -D TEST
              -lgcov
              --coverage
lib_deps =
    ArduinoFake
```

Figura 3.5: Declaración de ambiente de desarrollo nativo para el sistema embebido

Para la declaración del ambiente de prueba, solo basta con incluir en el archivo platformio.ini la primera y segunda línea de la imagen presentada anteriormente; sin embargo, en la imagen se incluyen configuraciones adicionales que se detallarán en las siguientes etapas del desarrollo de la metodología propuesta.

Para el presente trabajo se usará el framework de Unity para la ejecución de las pruebas. Para ello es necesario incluir la línea test_framework = unity para definir el marco de trabajo usado para ejecutar las pruebas. Al agregar esto en el proyecto, la primera vez que se ejecutan las pruebas se instala la librería correspondiente para usar unity previo a la ejecución de las mismas.

Habiendo definido un ambiente de desarrollo nativo, es necesario adaptar el software para su ejecución de forma nativa. El software del proyecto ha sido previamente diseñado para su ejecución en un entorno de desarrollo embebido; por ello, incluye configuraciones, funciones y métodos que pueden ejecutarse únicamente en un ambiente embebido. Para ejecutar el software embebido de forma nativa, es necesario simular las funciones y métodos que involucran directamente interacciones entre hardware y software.

Como herramientas de simulación, se hará uso principalmente de dos frameworks, ArduinoFake y FakeIt. Ambos frameworks tienen compatibilidad con PlatformIO y principalmente se les dará su uso para simular las interacciones entre hardware y software a través de mocks. El framework ArduinoFake pemitirá simular todas las funciones del framework de Arduino incluídas en el proyecto y FakeIt facilitará el uso de mocks para simular cualquier función del proyecto que involucre interacción con el hardware del sistema. Para su uso, es necesario decla-

rar en el archivo platformio.ini, la línea lib_deps incluyendo la librería a usar de ArduinoFake. Como esta librería está basada en el framework de FakeIt, no es necesario añadirlo directamente en las configuraciones del proyecto.

Siguiendo con el desarrollo de las pruebas, es necesario incluir un directorio para guardar los archivos de pruebas del proyecto de software embebido. En el caso de PlatformIO, cuando se crea el proyecto, también se crea una carpeta correspondiente para el almacenamiento de pruebas de nombre test. Para que el comando de ejecución de pruebas de PlatformIO detecte los archivos de prueba, es necesario crear subcarpetas que empiecen por el nombre test_ y en ellas incluir los archivos .c de pruebas. De esta manera se podrán dividir los archivos de prueba por componentes del sistema a nivel de hardware y realizar su ejecución correspondiente.

Para incluir las funciones y métodos en los archivos de prueba, es necesario organizar los archivos de proyecto en subcarpetas incluídas en la carpeta de proyecto lib, donde cada una incluya la declaración de un archivo .cpp con su .h correspondiente. Si no se realiza esta organización de archivos, se presentarán errores de ejecución de las pruebas por dependencia de archivos.

Para el desarrollo de los archivos de prueba, como se mencionó anteriormente, se hará uso del framework de Unity. La declaración del archivo de prueba de Unity debe incluir como mínimo 3 funciones: setUp(), tearDown() y el main(). Esta última función depende del ambiente de desarrollo de pruebas, si las pruebas se ejecutan un ambiente embebido para el framework de Arduino, se deben incluir las funciones setup() y loop(). Sin estas funciones como mínimo, en la ejecución se mostrarán errores de declaración de funciones.

```
void setUp(void) {
    // set stuff up here
}

void tearDown(void) {
    // clean stuff up here
}

void test_function(void) {
    // include unit to test
}

int main(){

    UNITY_BEGIN(); // start unit testing

    RUN_TEST(test_function);

    UNITY_END(); // stop unit testing

}
```

Figura 3.6: Estructura básica de un archivo de pruebas

Para la declaración de los módulos de prueba, la metodología sigue la estructura propuesta:

3.4.1. Estructura para la declaración de funciones de prueba

1. Inclusión de bibliotecas de pruebas: Se deben añadir los archivos de cabecera que permitan invocar las funciones declaradas en proyecto del sistema. Así mismo, se debe incluir la librería unity.h para hacer su uso en la declaración de las pruebas.

2. Definición de las función de prueba: Se define el módulo de prueba para invocarlo a futuro en la función main().

3. Configuración de entorno de prueba: Se debe declarar dentro del módulo de prueba, las condiciones iniciales según los casos de prueba definidos para la unidad. Cabe resaltar que también se puede añadir la creación e inicialización de variables y objetos que faciliten la ejecución de la prueba unitaria. La declaración de la función setUp(), propia del framework de Unity, facilita esta configuración, ya que esta función se ejecuta antes de la ejecución del módulo de prueba.

4. Inclusión de unidad de prueba: Se debe invocar a la función o método para el cual se diseñó la prueba.

5. Evaluación de resultado: Se debe incluir una etapa de confirmación para verificar si la unidad ha pasado satisfactoriamente la prueba o no. Esto, en el framework de Unity, se suele incluir según la lista de funciones de aserción que incluye.

6. Limpieza de entorno de prueba: Después de la ejecución de prueba, es necesario revertir los cambios realizados debido a la ejecución según el caso de prueba. Si no se realiza este paso, la ejecución de la prueba podría afectar el funcionamiento de las siguientes pruebas, a pesar de que estas sean independientes entre sí. La función tearDown del framework de Unity, se ejecuta automáticamente después de la ejecución de las pruebas, permitiendo esta limpieza de ambiente de prueba.

Habiendo definido el módulo de prueba siguiendo los pasos mostrados previamente, si la unidad a probar incluye interacciones directas entre el hardware y software, es necesario la inclusión de simuladores o mocks que permitan su ejecución. Previo a la ejecución de las pruebas, es necesario adaptar los archivos de proyecto para su ejecución en un entorno nativo. Para poder ejecutar las pruebas en este ambiente local, se debe seguir el siguiente procedimiento:

3.4.2. Procedimiento para la declaración de funciones que simulan la interacción entre hardware y software

1. Identificación de funciones que interactúen directamente con el hardware: Se debe hacer un análisis de las funciones que incluyan esta interacción software y hardware para futuramente desarrollar su declaración de mock en cada uno de los archivos declarados para el proyecto. Cabe resaltar que para la declaración de los mocks se hará uso de la librería FakeIt.h

2. Creación de archivo de simulación de interacción: Se debe declarar un archivo que incluya todas las funciones a simular correspondientes al archivo de proyecto. Se deben crear sus archivos .cpp y .h correspondientes.

3. Creación de librería propia de simulación: Después de haber definido las funciones de simulación de interacción, estas se pueden incluir en un un archivo único para su uso directo en las pruebas. En este archivo se incluyen todas las funciones que tengan interacción directa entre hardware y software de cada uno de los componentes del sistema.

Habiendo definido las funciones de simulación, ya se pueden adaptar los archivos de código de proyecto para su futura ejecución en un ambiente de desarrollo nativo. Para diferenciar el uso de las funciones de simulación de las funciones que interactúan con el hardware cuando se ejecute el código en diferentes ambientes de desarrollo, PlatformIO cuenta con banderas que se declaran cuando el código termina de ser construído en un ambiente de desarrollo en específico. Esta declaración de banderas se incluye en la configuración del ambiente en el achivo platformio.ini. Su ejemplo de uso se evidencia en la línea correspondiente a build_flags en la definición -D TEST. Esta configuración quiere decir que cuando se ejecuten los archivos de proyecto en el ambiente nativo de desarrollo, se creará la bandera TEST.

Haciendo uso de esta bandera, se podrán diferenciar las funciones en diferentes entornos evaluando si la bandera ha sido declarada o no. Cabe resaltar que para este caso, la bandera solo será declarada cuando se vayan a realizar las pruebas. Esta se puede incluir en los archivos de prueba de la siguiente manera:

En la imagen anterior, se puede apreciar que las funciones como Serial2.Available() y gps.location.lat(), ambas desarrolladas para un ambiente de desarrollo embebido, no son usadas directamente en un ambiente de desarrollo nativo si la bandera TEST ha sido declarada. Como reemplazo, se usan funciones que simulan la interacción directa entre hardware y software a través de mocks. Para el uso de este tipo de funciones, se debe incluir la librería desarrollada previamente que incluye todas las funciones y métodos que simulen dicha interacción.

3.5. Ejecución de las pruebas unitarias

La siguiente etapa en la metodología propuesta, corresponde a la ejecución de las pruebas unitarias. En las etapas anteriores ya se definieron aspectos a tomar en cuenta para el desarrollo de las pruebas unitarias en software embebido, tales como el ambiente de desarrollo de pruebas,

```
bool gpsGetCoordinates(float *lat, float *lng)
{
    #ifndef TEST
    if(!Serial2.available())
        return false;
    while(Serial2.available())
    {
            int c = Serial2.read();
            gps.encode(c);

            if(gps.location.isUpdated())
    #else
    if(!Serial.available())
        return false;
    while(Serial.available())
    {
        int c = Serial.read();
        gps.encode(c);
        if(gps.isUpdated())
    #endif
        {
            double tempLat, tempLng;

        #ifndef TEST
            tempLat = gps.location.lat();
            tempLng = gps.location.lng();
        #else
            tempLat = gps.lat();
            tempLng = gps.lng();
        #endif
```

Figura 3.7: Adaptación de función para poder realizar pruebas en un ambiente nativo

tipo de enfoque de pruebas, simulación de interacciones entre hardware y software, etc. En el caso de PlatformIO, este cuenta con un comando para la ejecución de las pruebas en su línea de comandos. También cuenta con un atajo a la sección de pruebas llamada “Testing”, donde se incluyen todos los archivos de pruebas existentes identificados en las carpetas de proyecto de PlatformIO, específicamente en la carpeta “test”.

Antes de realizar la ejecución de las pruebas, se debe configurar el ambiente de proyecto al ambiente nativo donde se desarrollaron las pruebas, para ello se hace la configuración en la opción de proyecto “Switch PlatformIO Project Environment”

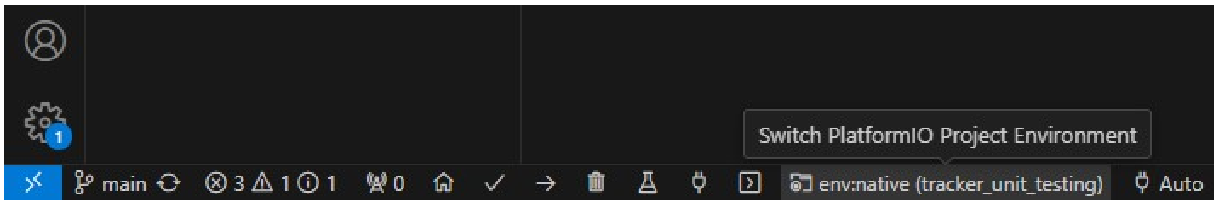


Figura 3.8: Selección de ambiente de desarrollo en PlatformIO

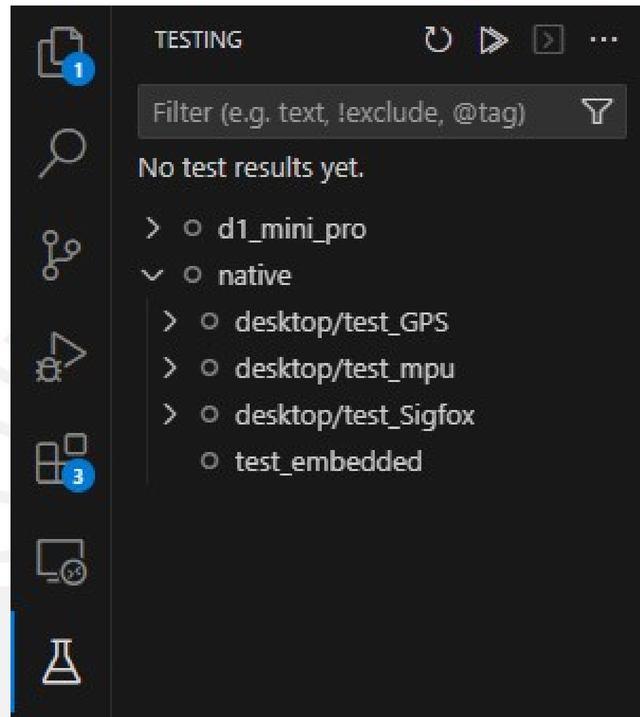


Figura 3.9: Herramienta "Testing"de PlatformIO

3.6. Documentación de resultados de ejecución de pruebas

Realizada la ejecución de las pruebas, la siguiente etapa en la metodología propuesta, corresponde a la de documentación de resultados de ejecución. En esta etapa se hace una revisión de las unidades que pasaron satisfactoriamente la prueba y las que no. La documentación de la ejecución de las pruebas permite visualizar el estado de ejecución de unidad de prueba, el tiempo que tomó su ejecución total y un resumen de ejecución de pruebas. Esta etapa de documentación, genera una mayor trazabilidad al desarrollo del proyecto a nivel de software embebido.

```
Processing desktop/test_GPS in native environment
--------------------------------------------------------------------------------
Building...
Testing...
test\desktop\test_GPS\test_GPSModule.cpp:37: test_gpsGetCoordinates_receive_DataCorrectly       [PASSED]
-------------- native:desktop/test_GPS [PASSED] Took 0.74 seconds --------------

=================================== SUMMARY ===================================
Environment    Test              Status    Duration
-------------  ----------------  --------  ------------
native         desktop/test_GPS  PASSED    00:00:00.736
================= 1 test cases: 1 succeeded in 00:00:00.736 =================
```

Figura 3.10: Reporte de resultado de pruebas en PlatformIO

La herramienta de ejecución de pruebas de PlatformIO ya incluye una posterior documentación; sin embargo, para el análisis de la confiabilidad del software es recomendable considerar un análisis de cobertura de las pruebas. Esto permitirá identificar las funciones o métodos que no han sido cubiertos en la ejecución de las pruebas para una futura implementación de pruebas. En la metodología propuesta, se hará uso de la herramienta gcvor para este propósito. Para la creación de la documentación de cobertura de prueba, se debe ingresar en el terminal de proyecto el comando gcvor seguido de las configuraciones de exclusión de archivos dependiendo de la prueba desarrollada. Antes de ingresar el comando, es necesario incluir banderas en la configuración del proyecto que permita obtener los archivos relacionados a los resultados de ejecución de las pruebas para que la herramienta gcvor pueda identificarlos para su análisis. Estas se puede apreciar en la figura 3.4 en las línesa -gcov y –coverage. Cabe resaltar que también se deben ejecutar las pruebas en el terminal para que se pueda crear el reporte de cobertura, esto se puede realizar dándole click a la opción Test de PlatformIO en la barra de herramientas de proyecto.

Cabe resaltar que este comando se debe ejecutar en la misma carpeta donde se ubique el archivo de proyecto y posterior a la ejecución de la prueba. Ingresando el comando, se obtiene un reporte de cobertura de pruebas que detalla por cada archivo, la cantidad de líneas, la cantidad de líneas ejecutadas, la cantidad de líneas ignoradas en las pruebas y el porcentaje de cobertura. En la parte final del reporte se incluye el porcentaje total de cobertura de pruebas.

```
PS C:\Users\User\Documents\GitHub\tracker_unit_testing> gcovr -e .pio/libdeps
------------------------------------------------------------------------------
                           GCC Code Coverage Report
Directory: .
------------------------------------------------------------------------------
File                                       Lines    Exec  Cover   Missing
------------------------------------------------------------------------------
.pio/build/native/unity_config/unity_config.c
                                               8       4    50%   17-20
lib/TrackerFake/TrackerFake.cpp                5       5   100%
lib/TrackerFake/TrackerFake.h                 14      13    92%   59
lib/TrackerFake/gpsFake.cpp                    5       5   100%
lib/TrackerFake/i2cFake.cpp                    5       0     0%   12,14,16,18,20
lib/gps/gps.cpp                               38      15    39%   24,29,31-33,48,80-81,88,91,93-94,96-97,99,101-102,106-111
test/desktop/test_GPS/test_GPSModule.cpp      21      21   100%
------------------------------------------------------------------------------
TOTAL                                         96      63    65%
------------------------------------------------------------------------------
```

Figura 3.11: Ejemplo de reporte de cobertura de código usando la herramienta gcovr

Capítulo IV

Resultados de aplicación de la metodología

En este capítulo se presentará la aplicación de la metodología propuesta en un sistema embebido ya implementado a nivel de hardware y en uno aún en etapa de desarrollo. Se presentará la implementación de la metodología propuesta siguiendo cada paso a detalle de la misma en el sistema embebido ya desarrollado a nivel de hardware. Se realizarán las pruebas unitarias para verificar cada uno de los requisitos planteados para cada sistema embebido en particular y posteriormente se realizará un análisis de los resultados obtenidos y comparaciones entre el desarrollo de las pruebas unitarias en un ambiente embebido y un ambiente nativo.

4.1. Desarrollo de la metodología en un sistema embebido desarrollado

A continuación se presentan los resultados de implementación de la metodología para un sistema embebido que ya está desarrollado completamente a nivel de hardware; sin embargo se realizarán y ejecutarán las pruebas unitarias para un ambiente embebido y uno nativo. El sistema al cual se aplicarán las pruebas unitarias, es el mismo sistema que se usó como caso de estudio para realizar el diseño de la metodología. En líneas generales, este equipo tiene la funcionalidad de enviar mensajes de ubicación GPS cuando este entre en movimiento.

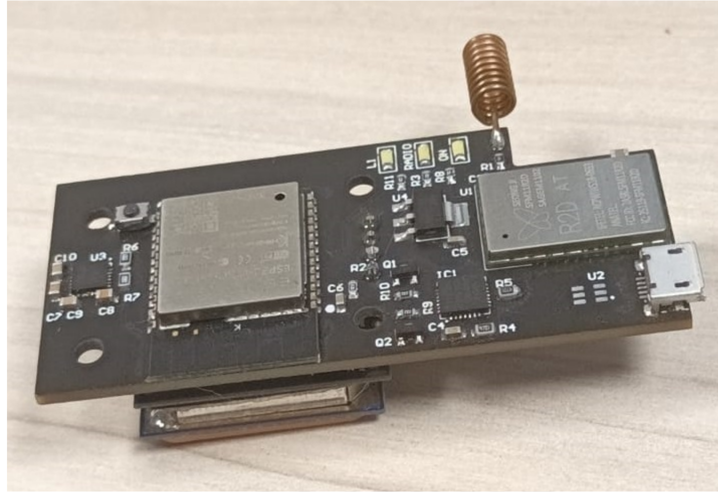


Figura 4.1: Fotografía de sistema embebido usado como caso de estudio

A continuación, se presentará el desarrollo de cada paso de la metodología propuesta:

4.1.1. Definición de requerimientos del sistema

Para dar inicio al desarrollo de la metodología es necesario una clara definición de requerimientos definidos entre el desarrollador y el cliente. El sistema embebido real en este caso de aplicación, cuenta con la siguiente lista de requerimientos de sistema.

Lista de requerimientos del sistema

Funciones principales:

- El equipo debe enviar mensajes a través de una red de comunicación.
- El equipo debe leer datos de ubicación y temperatura.
- El equipo debe detectar movimiento.
- El equipo debe diferenciar entre movimiento y vibración.
- El equipo debe enviar datos de ubicación y temperatura cuando se detecte un movimiento del mismo, no cuando se detecte vibración.

Funciones secundarias:

- El equipo debe entrar a modo de bajo consumo cuando no se detecte movimiento.
- El equipo deberá despertarse del modo bajo consumo cuando se detecte movimiento para enviar el mensaje.

En base a esta lista de requerimientos, se podrán definir las pruebas a realizar para cada una de las funcionalidades principales del sistema.

4.1.2. Análisis de diseño de sistema

A continuación, se presenta la lista de componentes de hardware del sistema que permitirán cumplir con los requerimientos del mismo:

- Módulo de comunicación WSSFM11R2DAT LPWAN Sigfox
- Módulo acelerómetro MPU6050
- Módulo GPS BN-880
- Microcontrolador ESP32-WROOM-32

Para cada uno de los componentes del sistema, se realizará una revisión de especificaciones técnicas para identificar únicamente las interacciones entre hardware y software, ya que serán necesarias identificarlas para el desarrollo a nivel de software:

Módulo de comunicación WSSFM11R2DAT LPWAN Sigfox

Según la hoja de datos, las últimas siglas del número de parte corresponden al tipo de aplicación, este puede ser por comandos AT o por API. En este caso, este módulo trabaja con comandos AT y para ello necesita conectarse por UART a un microcontrolador. Para ello, el módulo cuenta dos pines UARTTX y UARTRX para la recepción de comandos AT seriales. (WISOL, 2016).

1	GND	9	GPIO5	17	TXLED/ DBG_CLK	25	GPIO2
2	GND	10	GPIO4	18	NC4/DBG_EN	26	GPIO3
3	GND	11	CPU_LED	19	RST_N	27	GND
4	GND	12	RADIO_LED	20	GND	28	GND
5	NC3/ SYSCLK	13	GPIO9	21	VDD_IO	29	GND
6	GPIO8	14	UARTTX	22	GND	30	RF_IO
7	GPIO7	15	UARTRX	23	GPIO0	31	GND
8	GPIO6	16	RXLED/ DBG_DATA	24	GPIO1		

Figura 4.2: Distribución de pines en el módulo WSSFM11R2DAT

Módulo acelerómetro MPU6050

El módulo MPU6050 trabaja únicamente con interfaz I2C para transferencia de datos. Para ello incluye los pines SCL (Pin 23), SDA (Pin 24) y AD0 (Pin 9). Por estos pines se transmiten los datos de aceleración en los 3 ejes, así como también datos de temperatura. (InvenSense, 2013).

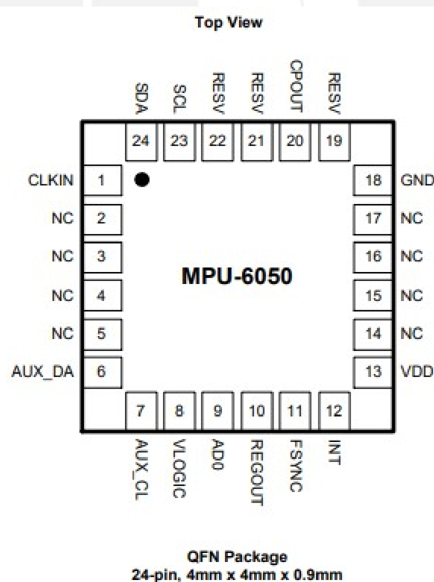


Figura 4.3: Distribución de pines en el módulo MPU6050 (InvenSense, 2013)

Módulo GPS BN-880

El módulo GPS BN-880 trabaja con interfaz I2C y UART; sin embargo, para la transmisión de datos de ubicación, latitud y longitud, se usa la interfaz de comunicación UART. Para ello el módulo cuenta con pines UART TX y UART RX. (*Beitian compass QMC5883 HMC5883 AMP2.6/PIX4/PIXHAWK GNSS GPS GLONASS D – Beitian store*, s.f.).

PIN	Name	I/O	Description
1	SDA	O	Compass SDA
2	GND	G	Ground
3	TX	O	Serial Data Output.
4	RX	I	Serial Data input.
5	VCC	I	DC 3.6V~5.5V supply input,Typical: 5.0V
6	SCL	I	Compass SCL

Figura 4.4: Distribución de pines en el módulo GPS BN-880 (*Beitian compass QMC5883 HMC5883 AMP2.6/PIX4/PIXHAWK GNSS GPS GLONASS D – Beitian store*, s.f.)

Microcontrolador ESP32-WROOM-32

Este microcontrolador cuenta con 38 puertos GPIO, 3 puertos de comunicación UART, 2 para I2C y 2 para SPI. Cuenta con una capacidad de memoria de 4MB y una memoria RAM de 520 KB, además se puede programar con el framework de Arduino. Para su aplicación en el sistema embebido, este incluye la cantidad mínima de puertos para comunicación con los demás módulos del sistema, ya que se requieren 2 puertos UART y un puerto I2C. (Espressif-Systems, s.f.).

Hardware	Module interfaces	SD card, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S, IR, pulse counter, GPIO, capacitive touch sensor, ADC, DAC, Two-Wire Automotive Interface (TWAI®), compatible with ISO11898-1 (CAN Specification 2.0)
	Integrated crystal	40 MHz crystal
	Integrated SPI flash	4 MB
	Operating voltage/Power supply	3.0 V ~ 3.6 V
	Operating current	Average: 80 mA
	Minimum current delivered by power supply	500 mA
	Recommended operating ambient temperature range	–40 °C ~ +85 °C
	Package size	18 mm × 25.5 mm × 3.10 mm
	Moisture sensitivity level (MSL)	Level 3

Figura 4.5: Especificaciones de hardware en el microcontrolador ESP32-WROOM-32 (Espressif-Systems, s.f.)

Cabe resaltar que para la facilidad en el desarrollo del software, los módulos incluyen librerías compatibles con el framework de Arduino, las cuales, algunas de ellas, se incluyen en los archivos de código del proyecto.

4.1.3. Planificación y Diseño de las Pruebas Unitarias

Ya teniendo definidos los requerimientos del sistema, se puede dar paso a la declaración de pruebas unitarias para el sistema embebido. Cabe resaltar que la verificación de algún requerimiento también puede resultar en el desarrollo de varias pruebas unitarias en vez de una sola.

Lista de pruebas unitarias a realizar en el sistema embebido

1. **Prueba de correcta lectura de datos de ubicación a través del módulo GPS BN-880**: Esta prueba consiste en la verificación de que se pueden recibir datos por comunicación serial. La lectura de datos es correcta si las variables donde se guardan los valores de latitud y longitud no están vacías después de ejecutar la función.

2. **Prueba de correcta detección de datos de aceleración en los 3 ejes y temperatura a través del módulo MPU6050**:

Esta prueba consiste en la verificación de que se pueden recibir datos por comunicación SPI. Se verifica que el arreglo definido para almacenar los valores de aceleración en una ventana de tiempo, no esté vacía; así como la variable encargada de almacenar el dato de temperatura.

3. **Prueba de correcta diferenciación entre movimiento y vibración a través del Módulo MPU6050**:
   Esta prueba evalúa el arreglo obtenido del cálculo del módulo de las lecturas de aceleración en los 3 ejes para determinar si el equipo está en movimiento o en vibración. Para ello, se calcula y evalúa la desviación estándar del arreglo y se compara con un valor umbral definido en base a pruebas realizadas con el acelerómetro. Si la desviación estándar calculada supera a este valor, se detecta una vibración en el equipo; caso contrario, se detecta movimiento. Cabe resaltar que el valor umbral considerado para esta prueba es de **0.1**, calculado en base a pruebas experimentales realizadas con el equipo.

4. **Prueba de correcto empaquetamiento de mensaje que incluya los datos de latitud, longitud y temperatura del equipo**:
   Esta prueba evalúa si se genera el mensaje empaquetado correctamente, previo a su transmisión. La estructura que el mensaje debe seguir es la siguiente: AT$SF=<latitud><longitud> <temperatura>, donde cada variable debe ser convertida a su equivalente en 4 bytes. Esta prueba valida que la conversión y el orden de las variables sean correctos.

5. **Prueba de correcto envío de mensajes mediante el módulo Wisol WSSFM11R2DAT LPWAN Sigfox**
   Esta prueba evalúa que la cadena con el mensaje empaquetado se envíe correctamente con las configuraciones previas necesarias para el módulo wisol. Se verifica que el pin de habilitación para el módulo sea configurado correctamente; así como también se verifica que el comando de configuración de región sea el permitido. Posterior a ello, se realiza la verificación de que se pueda transmitir el comando de envío de mensaje por comunicación serial.

4.1.4. Desarrollo de las Pruebas Unitarias

En total se realizaron 7 pruebas para el sistema, cada una para la verificación de una función en específico a ciertas condiciones de prueba.

Antes del desarrollo de las pruebas, es necesario adaptar el software embebido del sistema para su ejecución en un entorno nativo. Para ello se realizó una generación de archivos de simulación que incluyan cada una de las funciones que interactúan directamente con el hardware del sistema embebido, para ello se usó el framework de FakeIt (*FabioBatSilva/ArduinoFake: Arduino mocking made easy*, s.f.).

Para el proyecto realizado, se creó una librería auxiliar llamada TrackerFake.h. En esta librería se incluyeron archivos que incluían funciones que simulaban las funciones del módulo GPS y las interfaces de comunicación I2C para el módulo MPU6050.

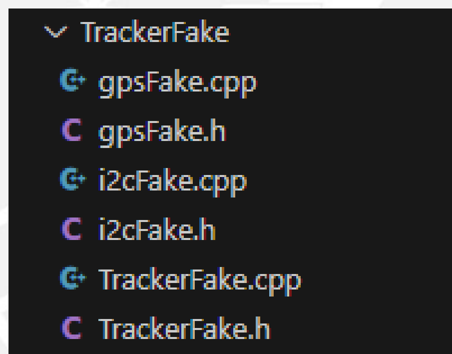
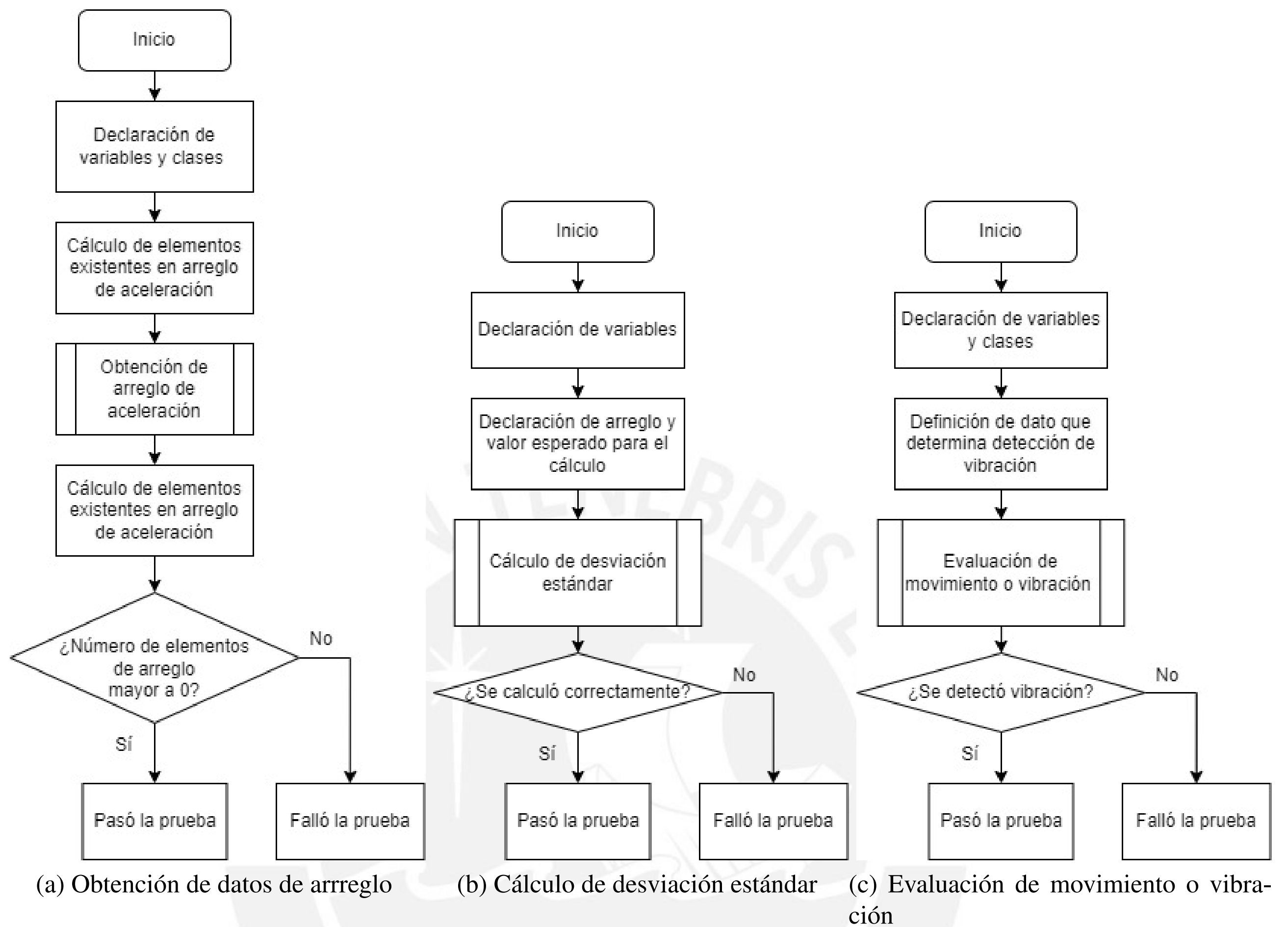


Figura 4.6: Librería auxiliar para la simulación de funciones que interactúan con el hardware

Haciendo uso de las build flags, se puede adaptar el código para decidir usar las funciones de simulación o las funciones propias del sistema embebido dependiendo del entorno de ejecución.

Ya habiendo definido una librería auxiliar para la ejecución del software en un entorno nativo, se desarrollará la estructura de las pruebas a ejecutar para validar el funcionamiento del sistema.

A continuación se presentan los diagramas de flujo para algunas de las pruebas realizadas. Estas pruebas servirán para determinar si el equipo detecta una vibración o un movimiento, para ello ha sido necesario realizar las pruebas unitarias para tres funciones que permiten la evaluación del mismo, acorde a los requerimientos.



(a) Obtención de datos de arrreglo (b) Cálculo de desviación estándar (c) Evaluación de movimiento o vibración

Figura 4.7: Diagramas de flujo de las pruebas realizadas

4.1.5. Ejecución de las Pruebas Unitarias

Para la ejecución de las pruebas, se realizaron archivos de pruebas para diferentes entornos. Cabe resaltar que la configuración entre ambos entornos es diferente, ya que algunas requieren ciertas inicializaciones. Se segmentaron también los archivos de prueba para su ejecución en cada entorno en específico.

Para cada una de las configuraciones, se agregó la ruta de archivos de prueba a ignorar debido a que ambas estan diseñadas para diferentes ambientes de ejecución.

```
[env:d1_mini_pro]
platform = espressif32
framework = arduino
board = lolin_d32
monitor_speed = 115200
test_ignore = desktop/test_GPS
              desktop/test_mpu
              desktop/test_sigfox
lib_ignore = TrackerFake
build_flags = -D NO_SEND_MSG
lib_deps =
    electroniccats/MPU6050@^0.3.0
    knolleary/PubSubClient@^2.8
    mikalhart/TinyGPSPlus@^1.0.2
```

Figura 4.8: Configuraciones de pruebas a ignorar en el entorno embebido

4.1.6. Documentación de resultado de ejecución de pruebas

Como siguiente y última etapa de la metodología, se realizaron las ejecuciones de las pruebas unitarias en un entorno nativo y embebido. En las tablas 4.1 y 4.2 se muestran los resultados de ejecución de pruebas.

A continuación, se muestra la tabla de pruebas ejecutadas en un ambiente nativo.

Pruebas en Ambiente Nativo				
Evaluación	test_GPS	test_MPU	test_Sigfox	Total
Pasaron	1	4	2	7
Fallaron	0	0	0	0
Número total de pruebas				7

Tabla 4.1: Tabla de resultado de pruebas en ambiente nativo

En el entorno nativo, de las 7 pruebas realizadas se puede apreciar que todas las pruebas pasan satisfactoriamente para cada uno de los módulos del sistema, usando simulaciones.

A continuación, se muestra la tabla de pruebas ejecutadas en un ambiente embebido.

En el entorno embebido, se puede apreciar que una de las pruebas no pasa la prueba satisfactoriamente cuando se ejecuta en un ambiente embebido; sin embargo, sí pasa la prueba satisfactoriamente en un entorno nativo.

```
[env:native]
platform = native
test_ignore = embedded/test_GPS
              embedded/test_mpu
              embedded/test_sigfox
build_flags = -std=gnu++17
              -D TEST
              -lgcov
              --coverage
lib_deps =
    ArduinoFake
```

Figura 4.9: Configuraciones de pruebas a ignorar en el entorno nativo

Pruebas en Ambiente Embebido				
Evaluación	test_GPS	test_MPU	test_Sigfox	Total
Pasaron	0	4	2	6
Fallaron	1	0	0	1
Número total de pruebas				7

Tabla 4.2: Tabla de resultado de pruebas en ambiente embebido

A partir de la ejecución de las pruebas realizadas, se halla el reporte de cobertura de pruebas en el entorno nativo con la herramienta gcvor (*gcovr — gcovr 6.0 documentation*, s.f.). Los datos obtenidos del reporte, se incluyen en la siguiente tabla:

Del reporte se puede apreciar que las pruebas tienen una cobertura del 50 %, el cual es un indicador de que se pueden implementar aún más pruebas que verifiquen el funcionamiento de las demás funciones que incluye el software embebido.

Se realizó una prueba de rendimiento a la ejecución de las pruebas en ambos ambientes

```
Processing embedded/test_GPS in d1_mini_pro environment
--------------------------------------------------------------------------------
Building & Uploading...
Testing...
If you don't see any output for the first 10 secs, please reset board (press reset button)

test/embedded/test_GPS/test_GPSModule.cpp:36: test_gpsGetCoordinates_shoulReceive_DataCorrectly: No se recibe informacion del GPS por Serial    [FAILED]
-------------------------------------------------- d1_mini_pro:embedded/test_GPS [FAILED] Took 12.99 seconds ---
```

Figura 4.10: Prueba fallida en entorno embebido

Cobertura de Pruebas			
Archivo	Líneas	Ejecutadas	Cobertura
lib/TrackerFake/TrackerFake.cpp	5	5	100 %
lib/TrackerFake/TrackerFake.h	14	13	92 %
lib/TrackerFake/gpsFake.cpp	5	5	100 %
lib/TrackerFake/i2cFake.cpp	5	1	20 %
lib/gps/gps.cpp	36	14	38 %
lib/integer/integer.cpp	22	9	40 %
lib/mpu/mpu.cpp	161	58	36 %
lib/sigfox/sigfox.cpp	110	29	26 %
test/desktop/test_GPS/test_GPSModule.cpp	21	21	100 %
test/desktop/test_GPS/test_SigfoxModule.cpp	32	32	100 %
test/desktop/test_GPS/test_MPUModule.cpp	43	43	100 %
Total	462	234	50 %

Tabla 4.3: Tabla de Cobertura de Pruebas

tomando en cuenta diez ejecuciones a las pruebas:

Tiempos de Ejecución (s)		
# Ejecución	Entorno Nativo	Entorno Embebido
1	17.873	55.156
2	6.126	41.301
3	6.140	43.269
4	6.216	41.425
5	6.145	42.986
6	6.144	40.746
7	6.140	40.938
8	6.218	48.299
9	6.096	43.786
10	6.242	43.336

Tabla 4.4: Tabla comparativa de tiempos de ejecución en ambos ambientes

De la tabla 4.4 se puede apreciar que el tiempo de ejecución en un entorno embebido es mayor que en el entorno nativo. La primera vez que se ejecutan las pruebas suele demorar un tiempo mayor debido a que en la ejecución del código se detectaron algunos warnings para la implementación en ambos ambientes de prueba.

Se hizo también una prueba de consistencia de las pruebas para la verificación de que las pruebas siguen respondiendo igual para diferentes ejecuciones y no generar falsos resultados de prueba.

Número de pruebas consistentes		
# Ejecución	Entorno Nativo	Entorno Embebido
1	7	7
2	7	7
3	7	7
4	7	7
5	7	7
6	7	7
7	7	7
8	7	7
9	7	7
10	7	7

Tabla 4.5: Tabla de números de pruebas consistentes en ambos ambientes

De la tabla 4.5, se puede apreciar que los resultados de las pruebas no varían en su resultado de ejecución en ambos ambientes de ejecución.

4.2. Verificación de funcionamiento del sistema embebido desarrollado

En base a los resultados de las pruebas unitarias, se verificará el correcto funcionamiento de las funciones probadas del sistema acorde a los requerimientos.

Subiendo el proyecto del software embebido, cuando el equipo está operando, se observa lo siguiente:

En la imagen 4.11 se puede observar el muestreo de las aceleraciones en los 3 ejes obtenido del acelerómetro MPU6050 para evaluarlos constantemente y determinar si el equipo se mueve o detecta una vibración. Cuando se genera una vibración al equipo, se puede verificar que lo identifica correctamente.

```
New calibrated Values for Aceleration.
 Ax: 0.15 Ay: -1.00 Az: 0.21 Modulo vector: 0.01 T: 29.75
 Ax: 0.16 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.85
 Ax: 0.15 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.89
 Ax: 0.15 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.85
 Ax: 0.15 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.80
 Ax: 0.15 Ay: -1.00 Az: 0.19 Modulo vector: 0.00 T: 29.89
 Ax: 0.16 Ay: -0.99 Az: 0.19 Modulo vector: 0.01 T: 29.89
 Ax: 0.15 Ay: -0.99 Az: 0.20 Modulo vector: 0.01 T: 29.99
New calibrated Values for Aceleration.
 Ax: 0.15 Ay: -1.00 Az: 0.21 Modulo vector: 0.01 T: 29.89
 Ax: 0.15 Ay: -0.99 Az: 0.20 Modulo vector: 0.01 T: 29.94
 Ax: 0.15 Ay: -1.00 Az: 0.21 Modulo vector: 0.01 T: 29.89
 Ax: 0.14 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.89
 Ax: 0.14 Ay: -0.99 Az: 0.20 Modulo vector: 0.02 T: 29.94
 Ax: 0.15 Ay: -0.99 Az: 0.20 Modulo vector: 0.01 T: 29.99
 Ax: 0.15 Ay: -1.00 Az: 0.20 Modulo vector: 0.01 T: 29.94
 Ax: 0.14 Ay: -0.99 Az: 0.21 Modulo vector: 0.01 T: 29.94
```

Figura 4.11: Muestreo de datos de aceleración en el equipo

```
 Ax: 0.03 Ay: -0.04 Az: 0.97 Modulo vector: 0.01 T: 31.45
 Ax: 0.03 Ay: -0.04 Az: 0.99 Modulo vector: 0.01 T: 31.45
 Ax: 0.03 Ay: -0.04 Az: 0.98 Modulo vector: 0.00 T: 31.45
 Ax: 0.03 Ay: -0.04 Az: 0.97 Modulo vector: 0.01 T: 31.49
 Ax: 0.02 Ay: -0.05 Az: 0.97 Modulo vector: 0.01 T: 31.49
 Ax: 0.03 Ay: -0.03 Az: 0.99 Modulo vector: 0.02 T: 31.49
 Ax: 0.03 Ay: -0.05 Az: 1.00 Modulo vector: 0.02 T: 31.59
 Ax: 0.03 Ay: -0.05 Az: 0.98 Modulo vector: 0.01 T: 31.49
 Ax: 0.02 Ay: -0.04 Az: 0.97 Modulo vector: 0.02 T: 31.40
MPU is detecting an unexpected variation in its measurements!.
Evaluation started.
False Alarm
 Ax: 0.02 Ay: -0.05 Az: 0.99 Modulo vector: 0.01 T: 31.68
 Ax: 0.03 Ay: -0.05 Az: 0.98 Modulo vector: 0.01 T: 31.64
```

Figura 4.12: Detección de vibración en el equipo

Cuando el equipo detecta vibración, el código responde como una falsa alarma de movimiento y no procede a enviar algún mensaje,

En el caso de cuando se mueve el equipo, el sistema identifica movimiento mediante el acelerómetro y lee los datos de latitud y longitud con el módulo GPS.

Posterior a la lectura y la detección de movimiento, se puede apreciar que se envía un mensaje mediante la red de sigfox, la cual se verifica correctamente su envío mediante el backend de Sigfox (*Home - Sigfox 0G Technology*, s.f.).

Cabe mencionar que a pesar de que específicamente la prueba de lectura correcta de los datos del GPS de forma serial no haya pasado la prueba cuando se ejecutó en un ambiente embebido, sí funcionó correctamente en la ejecución del código principal del proyecto, ya que

```
MPU is detecting an unexpected variation in its measurements!.
Evaluation started.
MPU values that have generated changes in the FSM: Idle -> Location:
Changing to location state. Before changing state we send current location.
[!!] Values read from GPS -> lat: -12.01212550 lng: -77.00335433
[*] Coordinates received from GPS and MPU -> lat: -12.01, lng: -77.00, T: 29.52
[*] Sigfox message: AT$SF=ab3140c1b8019ac25925ec41
[!] Sigfox message to send: AT$SF=ab3140c1b8019ac25925ec41
AT$SF=ab3140c1b8019ac25925ec41
```

Figura 4.13: Detección de movimiento en el equipo

				Base station reception attributes					
Time	Delay (s)	Seq Num	Data / Decoding	Station	RSSI (dBm)	Freq (MHz)	LQI	Callbacks	Location
2023-10-29 10:31:01	< 1	573	953140c1ab019ac27fe50842	95BE 97E7 97E6	-122.00 -103.00 -122.00	920.8536 920.8537 920.8323			

Figura 4.14: Verificación de envío de mensajes mediante el backend de Sigfox

podemos ver que los datos de latitud y longitud se leyeron correctamente.

4.3. Desarrollo de la metodología en un sistema embebido en etapa de desarrollo

Como segundo sistema para implementar la metodología propuesta, se optará por su aplicación en un proyecto de software embebido para un equipo que mide parámetros de suelo y ambiente en jardines.

En este caso, el equipo no está listo a nivel de hardware, sin embargo se pueden realizar las pruebas unitarias para verificar la lógica del programa desarrollado en un entorno de desarrollo nativo. De forma práctica, no se darán detalles de implementación de cada uno de los pasos de la metodología tal como la aplicación de la metodología en el sistema de la anterior sección; sin embargo, se hará un análisis de resultados de pruebas y validación de funcionamiento de algunas funciones del sistema a nivel de hardware.

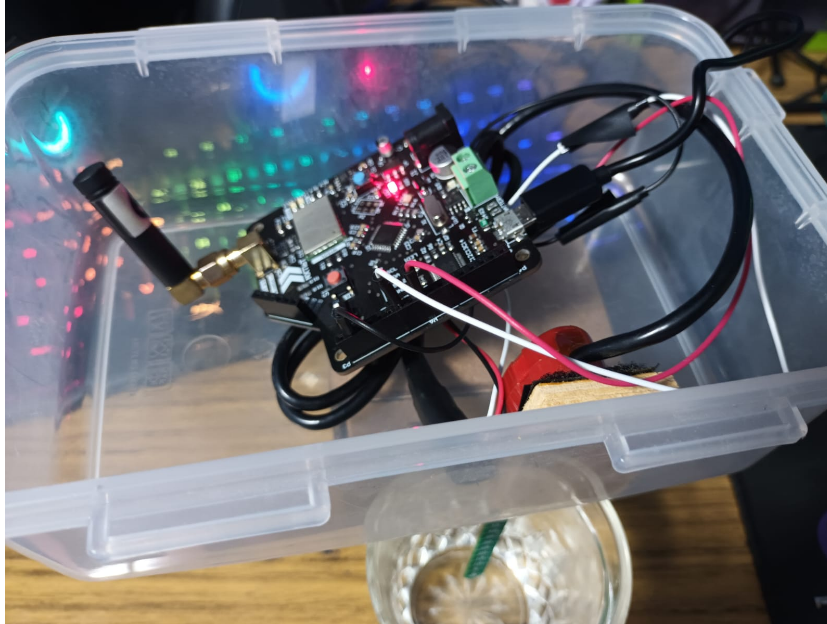


Figura 4.15: Prototipo de sistema embebido con aplicación en jardines

Se desarrollaron pruebas unitarias para los siguientes requerimientos:

- El equipo debe leer datos de humedad.
- El equipo debe recibir dato de hora en timestamp a través de la red Sigfox.
- El equipo debe enviar mensajes cada minuto múltiplo de 30 con un retardo aleatorio.

Se separaron las pruebas unitarias por componentes del sistema, y se ejecutaron las pruebas en un ambiente nativo, dando el siguiente resultado.

Pruebas en Ambiente Nativo				
Evaluación	test_humidity	test_rtc	test_sigfox	Total
Pasaron	1	2	2	5
Fallaron	0	0	0	0
Número total de pruebas				5

Tabla 4.6: Tabla de resultado de pruebas en ambiente nativo para el sistema en desarrollo

4.4. Verificación de funcionamiento del sistema en etapa de desarrollo

Al no contarse con el hardware completo a disposición, se verificó el funcionamiento de las funciones ya desarrolladas y probadas por la metodología en una tarjeta de desarrollo NXT IoT 2.0 (*Devkit 2.0 | Sigfox Partner Network | The IoT solution book*, s.f.) conectado al sensor de humedad de suelo VH400 (*Soil Moisture Sensor - VH400*, s.f.). Cabe resaltar que para estas pruebas de verificación de funcionamiento, no se tomó en cuenta la ejecución de las pruebas en un ambiente embebido.

A continuación, se presentan resultados de funcionamiento del sistema en operación normal, considerando que no aún no se tiene acceso al hardware final del sistema:

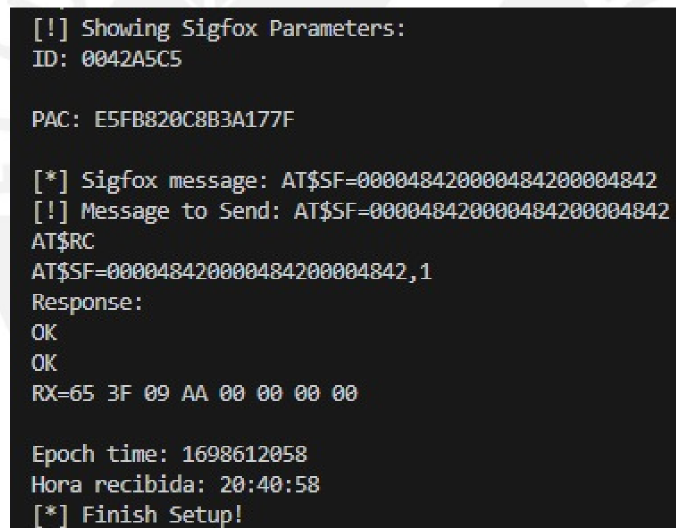


Figura 4.16: Verificación de envío y recibimiento de mensajes en el prototipo

En la figura 4.16 se puede apreciar que el mensaje de hora en formato timestamp se recibe y convierte correctamente. También se puede observar que el equipo se configura correctamente a esa hora obtenida mediante el RTC implementado a nivel de hardware. Cabe resaltar que para que el equipo reciba un mensaje, debe haber al menos podido enviar alguno, por lo que también se verifica de que el dispositivo puede enviar mensajes.

A continuación, se verificará si el equipo envía mensajes cada 30 minutos, añadiendo un retardo aleatorio, tal como se verificó en la prueba ejecutada en un entorno nativo.

```
[*] Current Time -> 20:59:55
[*] Current Time -> 20:59:56
[*] Current Time -> 20:59:57
[*] Calculating Parameters:
[*] Humidity: 98.00
[*] Temperature: 0.00
[*] Current Time -> 20:59:58
[*] Current Time -> 20:59:59
[*] Random Value for minutes: 4
[*] Random Value for seconds: 57
[*] SE ENVIARA A LAS -> 21:4:57
[*] Current Time -> 21:0:0
[*] Current Time -> 21:0:1
```

Figura 4.17: Verificación de envío de mensajes cada 30 minutos con retardo en el prototipo

En la figura 4.17 se puede verificar que cuando el equipo detecta un minuto múltiplo de 30, detecta que debe enviar un mensaje y le agrega un delay para enviarlo específicamente a esa hora. Tambiém se aprecia en la figura que el dato de humedad se lee correctamente, ya que el equipo fue probado cuando se colocó el sensor de humedad en agua, lo que da un valor cercano al 100 % en humedad.

Al igual que en el sistema de la sección anterior, se verificará el correcto envío y recibo de mensajes vía el backend de Sigfox (*Home - Sigfox 0G Technology*, s.f.).

Time	Seq Num	Data / Decoding	LQI	Callbacks		Location
2023-10-29 21:05:19	733	59c8c4420000000000004842 vwc: 98.391304				
2023-10-29 20:40:26	731	000048420000484200004842 vwc: 50.0				

Figura 4.18: Verificación de envío de mensajes mediante el backend de Sigfox en el sistema en desarrollo

De los resultados, se puede comprobar de que las funciones desarrolladas para el sistema pueden ser comprobadas en funcionamiento sin necesidad de acceso al hardware del sistema.

Conclusiones

- Se diseñó una metodología para implementar pruebas unitarias en sistemas embebidos a nivel de software. Asímismo, se corroboró que se puede implementar la metodología propuesta en sistemas embebidos reales.
- Se implementaron pruebas unitarias en sistemas embebidos para identificar las limitaciones y desafíos que implica. Esto permitió identificar las herramientas necesarias para poder diseñar la metodología.
- Se identificaron las herramientas de simulación de hardware existentes para su uso en pruebas unitarias. Se comprobó que es posible la implementación de las pruebas unitarias en sofware embebido sin necesidad del hardware del mismo.
- Se evaluó y validó la metodología propuesta en diferentes softwares embebidos, correspondientes a diferentes sistemas.
- Se comprobó que el desarrollo y ejecución de las pruebas unitarias en un ambiente nativo exige un menor tiempo de ejecución que en un ambiente embebido. Esto hace preferir la ejecución de pruebas en un ambiente nativo en comparación a un ambiente embebido debido a que el mismo resulta en una ejecución más optimizada y automatizada, sin dependencia del hardware para realizar el proceso de pruebas de software embebido.

Bibliografía

Abushama, H. M., Alassam, H. A., y Elhaj, F. A. (2021). The effect of test-driven development and behavior-driven development on project success factors: A systematic literature review based study. En *2020 international conference on computer, control, electrical, and electronics engineering (iccceee)* (p. 1-9). doi: 10.1109/ICCCEEE49695.2021.9429593

Alam, T., Yang, Z., Chen, B., Armour, N., y Ray, S. (2022). Firver: Concolic testing for systematic validation of firmware binaries. *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, *2022-Janua*, 352-357. doi: 10.1109/ASP-DAC52403.2022.9712594

Azumi, T., Yamamoto, M., Kominami, Y., Takagi, N., Oyama, H., y Takada, H. (2007). A new specification of software components for embedded systems. En *10th ieee international symposium on object and component-oriented real-time distributed computing (isorc'07)* (p. 46-50). doi: 10.1109/ISORC.2007.7

Bajer, M., Szlagor, M., y Wrzesniak, M. (2015, 8). Embedded software testing in research environment. a practical guide for non-experts. En (p. 100-105). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/MECO.2015.7181877

Beitian compass qmc5883 hmc5883 amp2.6/pix4/pixhawk gnss gps glonass d – beitian store. (s.f.). `https://store.beitian.com/products/beitian-compass-qmc5883l-amp2-6-pix4-pixha` $1_sid = 9096ae4e1_ss = rvariant = 44696120295711.((Accessedon10/14/2023))$

Bhat, A., y Quadri, S. M. K. (2015). Equivalence class partitioning and boundary value analysis - a review. En *2015 2nd international conference on computing for sustainable global*

development (indiacom) (p. 1557-1562).

Brar, H. K., y Kaur, P. J. (2015). Differentiating integration testing and unit testing. En *2015 2nd international conference on computing for sustainable global development (indiacom)* (p. 796-798).

Ccstudio ide, configuration, compiler or debugger | ti.com. (s.f.). `https://www.ti.com/tool/CCSTUDIO`. ((Accessed on 06/16/2023))

Cmock — throw the switch. (s.f.). `https://www.throwtheswitch.org/cmock`. ((Accessed on 06/16/2023))

Cpputest. (s.f.). `https://cpputest.github.io/`. ((Accessed on 06/16/2023))

de Boer, F. S., y Bonsangue, M. M. (2019). On the nature of symbolic execution. En *2019 21st international symposium on symbolic and numeric algorithms for scientific computing (synasc)* (p. 4-5). doi: 10.1109/SYNASC49474.2019.00009

Demeyer, S., Verhaeghe, B., Etien, A., Anquetil, N., y Ducasse, S. (2018). Evaluating the efficiency of continuous testing during test-driven development. En *2018 ieee workshop on validation, analysis and evolution of software tests (vst)* (p. 21-25). doi: 10.1109/VST.2018.8327152

Devkit 2.0 | sigfox partner network | the iot solution book. (s.f.). `https://partners.sigfox.com/products/devkit-2.0`. ((Accessed on 11/19/2023))

Dooley, J. F. (2017). *Software development, design and coding: With patterns, debugging, unit testing, and refactoring second edition*. Apress Media LLC. doi: 10.1007/978-1-4842-3153-1

eranpeer/fakeit: C++ mocking made easy. a simple yet very expressive, headers only library for c++ mocking. (s.f.). `https://github.com/eranpeer/FakeIt`. ((Accessed on 10/14/2023))

Espressif-Systems. (s.f.). *esp32-wroom-32_datasheet_en.pdf.*

https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32$_{d}$*atasheet*

Fabiobatsilva/arduinofake: Arduino mocking made easy. (s.f.). https://github.com/FabioBatSilva/ArduinoFake. ((Accessed on 10/14/2023))

García-Tudela, P. A., y Marín-Marín, J.-A. (2023). Use of arduino in primary education: A systematic review. *Education Sciences*. Descargado de https://api.semanticscholar.org/CorpusID:256451918

Garousi, V., Felderer, M., Çağri Murat Karapiçak, y Yilmaz, U. (2018, 7). What we know about testing embedded software. *IEEE Software*, *35*, 62-69. doi: 10.1109/MS.2018.2801541

gcovr — gcovr 6.0 documentation. (s.f.). https://gcovr.com/en/stable/. ((Accessed on 10/14/2023))

Googletest primer | googletest. (s.f.). https://google.github.io/googletest/primer.html. ((Accessed on 06/16/2023))

Gren, L., y Antinyan, V. (2017). On the relation between unit testing and code quality. En *2017 43rd euromicro conference on software engineering and advanced applications (seaa)* (p. 52-56). doi: 10.1109/SEAA.2017.36

Gu, C. (2016). *Building embedded systems: Programmable hardware*. Apress Media LLC. doi: 10.1007/978-1-4842-1919-5

Hodel, K. N., Silva, J. R. D., Yoshioka, L. R., Justo, J. F., y Santos, M. M. D. (2022). Fat-aes: Systematic methodology of functional testing for automotive embedded software. *IEEE Access*, *10*, 74259-74279. doi: 10.1109/ACCESS.2021.3128431

Home - sigfox 0g technology. (s.f.). https://www.sigfox.com/. ((Accessed on 11/19/2023))

Ieee standard for software unit testing. (1986). *ANSI/IEEE Std 1008-1987*, 1-28. doi: 10.1109/IEEESTD.1986.81001

InvenSense. (2013). *Mpu-6000 and mpu-6050 product specification revision 3.4 mpu-6000/mpu-6050 product specification.*

Ishak, M. K., Hwan, O. J., Jiashen, T., y Isa, N. A. M. (2019, 1). Automated compila-

tion test system for embedded system. *Makara Journal of Technology*, *22*, 115. doi: 10.7454/mst.v22i3.3515

Lenka, R. K., Kumar, S., y Mamgain, S. (2018). Behavior driven development: Tools and challenges. En *2018 international conference on advances in computing, communication control and networking (icacccn)* (p. 1032-1037). doi: 10.1109/ICACCCN.2018.8748595

linux-test-project/lcov: Lcov. (s.f.). `https://github.com/linux-test-project/lcov`. ((Accessed on 10/14/2023))

Maity, I., Bhattacharya, S., Bhattacharjee, A., Bhaumik, S., Shit, A., Paul, S., y Gangopadhyay, M. (2023, 12). Exploring arduino based electronic circuit design: A study of innovative applications and prototypes. En (p. 1-5). doi: 10.1109/IEMENTech60402.2023.10423427

Marwedel, P. (2021). *Embedded system design embedded systems foundations offcyber-physical systems, and theeinternet offthings fourth edition.* Descargado de `http://www.springer.com/series/8563`

Mirtalebi, M. (2017). *Embedded systems architecture for agile development: A layers-based model*. Apress Media LLC. doi: 10.1007/978-1-4842-3051-0

Mizoguchi, M., Iida, T., y Irie, T. (2020). Optimization of automated executions based on integration test configurations of embedded software. *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, 358-363. doi: 10.1109/ICSTW50294.2020.00064

Mohan, A., y Jha, S. K. (2019). Predicting and accessing reliability of components in component based software development. En *2019 international conference on intelligent computing and control systems (iccs)* (p. 1110-1114). doi: 10.1109/ICCS45141.2019.9065290

Morisaki, S., Shirata, S., Oyama, H., y Azumi, T. (2020, 12). Unit testing framework for

embedded component systems. En (p. 41-48). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/EUC50751.2020.00013

A professional collaborative platform for embedded development · platformio. (s.f.). `https://platformio.org/`. ((Accessed on 06/16/2023))

Sangiovanni-Vincentelli, A., Zeng, H., Di, M., y Marwedel, N. P. (2014). *Embedded systems development from functional models to implementations.* Descargado de `http://www.springer.com/series/8563`

Saraç, T. (2019). Certification aspects of model based development for airborne software. En *2019 ieee 2nd international conference on information and computer technologies (icict)* (p. 285-291). doi: 10.1109/INFOCT.2019.8711129

Shaout, A., y Pattela, S. (2021). Model based approach for automotive embedded systems. En *2021 22nd international arab conference on information technology (acit)* (p. 1-7). doi: 10.1109/ACIT53391.2021.9677298

Shin, K.-W., y Lim, D.-J. (2018). Model-based automatic test case generation for automotive embedded software testing. *International Journal of Automotive Technology*, *19*, 107-119. doi: 10.1007/s1223901800116

Siahaan, R. D., Kusumawardani, S. S., y Hidayah, I. (2022). E-learning evaluation of del superior high school based on black box testing with equivalence partitioning and graph-based testing. En *2022 8th international conference on science and technology (icst)* (Vol. 1, p. 1-6). doi: 10.1109/ICST56971.2022.10136292

Singh, G., Kumar Tiwari, U., y Kumar, S. (2020). R-r model: Reliable and reusable component-based development process model. En *2020 2nd international conference on advances in computing, communication control and networking (icacccn)* (p. 946-951). doi: 10.1109/ICACCCN51052.2020.9362777

Sinha, S., Goyal, N. K., y Mall, R. (2021, 4). Reliability and availability prediction of embedded systems based on environment modeling and simulation. *Simulation Modelling Practice*

and Theory, *108*. doi: 10.1016/j.simpat.2020.102246

Soil moisture sensor - vh400. (s.f.). https://vegetronix.com/soil-moisture-sensor. ((Accessed on 11/19/2023))

Stoico, V. (2021). A model-driven approach for early verification and validation of embedded systems. En (p. 684-688). Institute of Electrical and Electronics Engineers Inc. doi: 10.1109/MODELS-C53483.2021.00110

Torkar, R., Mankefors, S., Hansson, K., y Jonsson, A. (2003). An exploratory study of component reliability using unit testing. En *14th international symposium on software reliability engineering, 2003. issre 2003.* (p. 227-233). doi: 10.1109/ISSRE.2003.1251045

Unity — throw the switch. (s.f.). https://www.throwtheswitch.org/unity. ((Accessed on 06/16/2023))

Valvano, J. W. (2014). *Embedded systems : introduction to arm® cortex(tm)-m microcontrollers. volume 1.*

Wakitani, S., y Yamamoto, T. (2017). Practice of model-based development for automotive engineers. En *2017 ieee frontiers in education conference (fie)* (p. 1-4). doi: 10.1109/FIE.2017.8190678

WISOL. (2016). *Pba rf module wssfm11r2dat.*

Xie, T., Tillmann, N., y Lakshman, P. (2016). Advances in unit testing: Theory and practice. En *2016 ieee/acm 38th international conference on software engineering companion (icse-c)* (p. 904-905).

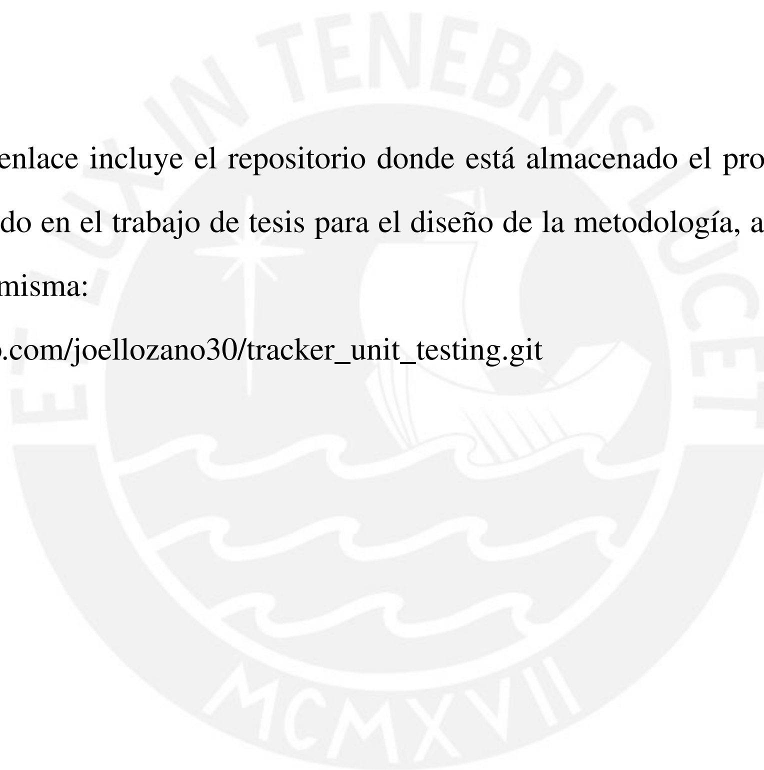
Zhang, C., Yan, Y., Zhou, H., Yao, Y., Wu, K., Su, T., ... Pu, G. (2018, 5). Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. En (p. 296-305). IEEE Computer Society. doi: 10.1145/3183519.3183554

Zhang, N., Bao, X., y Ding, Z. (2009). Unit testing: Static analysis and dynamic analysis. En *2009 fourth international conference on computer sciences and convergence information technology* (p. 232-237). doi: 10.1109/ICCIT.2009.106

μvision user's guide. (s.f.). https://developer.arm.com/documentation/101407/0538/About-uVision
((Accessed on 06/16/2023))

Anexos

Anexo A



El siguiente enlace incluye el repositorio donde está almacenado el proyecto de software embebido utilizado en el trabajo de tesis para el diseño de la metodología, así como su implementación de la misma:

https://github.com/joellozano30/tracker_unit_testing.git