UNIVERSIDAD AUTÓNOMA
DE SAN LUIS POTOSÍ

CENTRO DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO

MAESTRÍA EN INGENIERÍA DE LA COMPUTACIÓN

TESIS:

PLATAFORMA DE HARDWARE Y SOFTWARE BASADA EN COMPONENTES VIRTUALES PARA EL DESARROLLO ÁGIL DE REDES DE SENSORES INALÁMBRICAS DE BAJO COSTO

Para obtener el grado de maestría en ingeniería de la computación

PRESENTA:

LESD Luis Octavio Ortega Gutiérrez

UNIVERSIDAD AUTÓNOMA
DE SAN LUIS POTOSÍ

CENTRO DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO

MAESTRIA EN INGENIERÍA DE LA COMPUTACIÓN

TESIS:

PLATAFORMA DE HARDWARE Y SOFTWARE BASADA EN COMPONENTES VIRTUALES PARA EL DESARROLLO ÁGIL DE REDES DE SENSORES INALÁMBRICAS DE BAJO COSTO

Para obtener el grado de maestría en ingeniería de la computación

PRESENTA:

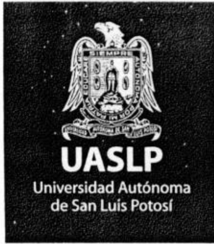
LESD Luis Octavio Ortega Gutiérrez

ASESOR:

Dr. Carlos Soubervielle Montalvo

CO-ASESOR:

Dr. Pedro David Arjona Villicaña



FACULTAD DE
INGENIERÍA

19 de mayo de 2022

ING. LUIS OCTAVIO ORTEGA GUTIÉRREZ
P R E S E N T E.

En atención a su solicitud de Temario, presentada por los **Dres. Carlos Soubervielle Montalvo y Pedro David Arjona Villicaña,** Asesor y Coasesor de la Tesis que desarrollará Usted, con el objeto de obtener el Grado de **Maestro en Ingeniería de la Computación**, me es grato comunicarle que en la sesión del H. Consejo Técnico Consultivo celebrada el día 19 de mayo del presente, fue aprobado el Temario propuesto:

TEMARIO:

"Plataforma de Hardware y Software Basada en Componentes Virtuales para el Desarrollo Ágil de Redes de Sensores Inalámbricas de Bajo Costo"

Introducción.
1. Marco teórico de los campos de computación relevantes.
2. Metodología y materiales para el desarrollo de la plataforma propuesta.
3. Diseño de la plataforma de hardware y software para el desarrollo ágil de redes de sensores inalámbricas.
4. Pruebas y validación de la plataforma desarrollada.

Conclusiones.
Referencias.

"MODOS ET CUNCTARUM RERUM MENSURAS AUDEBO"

A T E N T A M E N T E

DR. EMILIO JORGE GONZÁLEZ GALVÁN
DIRECTOR.

UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI
FACULTAD DE INGENIERIA
DIRECCION

www.uaslp.mx

Copia. Archivo.
*etn.

Av. Manuel Nava 8
Zona Universitaria • CP 78290
San Luis Potosí, S.L.P.
tel. (444) 826 2330 al39
fax (444) 826 2336

Mi Familia

Agradecimientos

Agradezco a mi Asesor, el Dr. Carlos Soubervielle Montalvo y mi Co-Asesor, el Dr. Pedro David Arjona Villicaña, por su paciencia y apoyo en la realización de esta tesis.

Al Dr. Oscar Ernesto Pérez Cham por su apoyo.

Doy gracias en especial:

A Dios, por permitirme vivir uno de los momentos más importantes de mi vida.

A mi Familia por su apoyo y comprensión.

Al Dr. Hugo Iván Medellín Castillo*, Jefe de Investigación y Posgrados de la Facultad de Ingeniería de la UASLP*, por las facilidades otorgadas en el uso de las instalaciones.

A los Profesores que me ayudaron a forjar.

A Gerardo Gabriel López Rocha por su apoyo como técnico académico.

A las demás personas que contribuyeron de alguna manera en la elaboración del presente trabajo de investigación.

Índice

Contenido **Página**

Glosario iii
Introducción 1
Problemáticas en redes inalámbricas de sensores para monitorización ambiental 2
Trabajo previo 2
Revisión de la literatura 3
Planteamiento del problema 4
Pregunta de Investigación 4
Hipótesis 4
Objetivos general y específico 4
Objetivo general 4
Objetivos específicos 4
Organización de la tesis 5
Capítulo 1: Marco teórico de los campos de la computación relevantes 7
1.1 Redes Inalámbricas de Sensores 7
1.1.1 Protocolos de Redes Inalámbricas de bajo consumo 8
Topologías físicas para BLE 9
Topologías físicas para ZigBee 10
1.2 Diseño de Sistemas Embebidos 11
1.2.1 Metodologías para el diseño de sistemas embebidos 12
1.2.2 Ingeniería de SW embebido 16
Capítulo 2: Metodología y materiales para el desarrollo de la plataforma propuesta 22
2.1 Flujo de Trabajo de la plataforma basada en Co-*design* HW/SW 22
2.1.1 Requerimientos del sistema 23
2.1.2 Fraccionamiento del HW y SW 24

2.1.3 Selección de herramientas de HW, SW e Interfaces …... 25
2.1.4 Selección específica de herramientas de HW, SW e Interfaces ….. 28

Capítulo 3: Diseño de la plataforma de Hardware y Software para el desarrollo ágil de redes de sensores inalámbricas …….………………………………………………...….. 38

3.1 Implementación del HW: configuración del HW ……. 38
3.1.1 Configuración de herramientas de HW ….... 38
3.1.2 Correcciones a la PCB del Mote empleado en la tesis de licenciatura …………………………….......... 40
3.2 Implementación del SW: especificación del FW ……. 41
3.3 Compilación del SW embebido ……………………. 45
3.3.1 Desarrollo de los componentes virtuales de HW…. 45
3.3.2 Desarrollo de GUI para los diferentes roles de la red …….………………………………………... 48
3.4 Descripción de pruebas necesarias de las herramientas disponibles ……………………………………………………….. 49
3.4.1 Descripción de pruebas de HW…………………... 50
3.4.2 Crear ejecutable del SW …………….......... 52

Capítulo 4: Pruebas y validación de la plataforma desarrollada.. 53

4.1 Pruebas de funcionalidad de la plataforma …………….. 53
4.1.1 Conectividad………………….……………….. 53
4.1.2 Componentes virtuales de HW ….….......... 54
4.1.3 Espacio en memoria de los recursos de HW……... 55
4.1.4 Medición de los sensores de la plataforma ….... 56
4.1.5 Consumo de energía de los Motes ….…........... 58
4.1.6 Establecer la WSN ….……………….......... 62
4.2 Usabilidad de la plataforma ………………………......... 63
4.3 Comparación con otras plataformas ………….………… 64

Conclusiones ……..…………………………………………… 65

Trabajo a Futuro ………………………………………… 66

Anexo I ……..………………………………………………… 67

Referencias ……..………………………………………………… 110

Glosario

API *Application Programming Interface* (Interfaz para la Programación de Aplicaciones). Son conjuntos de funciones con gran nivel de abstracción, creadas para ofrecer servicios a otras aplicaciones.

BLE *Bluetooth Low Energy* (*Bluetooth* de Baja Energía). Es la versión de *Bluetooth* de bajo consumo energético y que se utiliza para implementar WSN. Un concepto importante de las WSN es la implementación de redes de malla, por lo que, a partir de BLE 5.1, se pueden implementar redes de esta topología.

Build Crear ejecutable. Dentro del IDE CWDS v10.2, ocurren dos procesos al momento de crear un ejecutable: compilación y enlace (*linker*). En inglés, a ambos procesos, se le denomina *build*.

CAD *Computer Aided Design* (Diseño Asistido por Computadora). Es una técnica utilizada en la creación de bosquejos en el diseño, mediante el uso y apoyo de una computadora.

CSS *Cascading Style Sheet* (Hoja de Estilo en Cascada). En el contexto de XML, le da un formato amigable a un DTD o XML *Schema*. Es menos eficiente que XSL pero sencillo de utilizar.

Core Basado en la definición de *Oxford advanced learner´s dictionary*, es la parte central de un objeto.

CWDS v10.2 *Codewarrior Development Studio* versión 10.2. Es el IDE de Freescale/NXP para los módulos XBee basado en el IDE de código libre Eclipse. Cuenta con un ensamblador, un compilador para C, C++ y un intérprete/compilador para Java.

DTD *Document Type Definition* (Definición de Tipo de Documento). En el contexto de XML, es un documento le da sentido a una estructura de *tags*. Es menos eficiente que XML *Schema* pero sencillo de utilizar.

FW *Firmware*. Programa que se guarda dentro de una memoria y que permite la comunicación a nivel básico entre los diferentes módulos de hardware de un sistema además de la configuración de parámetros esenciales del mismo.

GATT *Generic Attribute Profile* (Perfil de Atributo Genérico). Perfil para que dispositivos BLE de diferentes fabricantes se puedan unir a una red BLE.

GUI *Graphic User Interface* (Interfaz Gráfica de Usuario). Permite una interacción amigable entre un usuario y una computadora, por medio de elementos gráficos en pantalla.

HAL *Hardware Abstraction Layer* (Capa de Abstracción de Hardware). Son funciones que debido a su nivel de abstracción, permiten la comunicación por SW entre una aplicación y el HW de un dispositivo.

IDE *Integrated Development Environment* (Ambiente Integrado de Desarrollo). Sistema de SW que contiene bibliotecas, APIs y otras funciones que agilizan el desarrollo de aplicaciones.

ISM *Industrial, Scientific, & Medical* (Industrial, Científico y Médico). Banda de frecuencias libres del espectro electromagnético. Consta de diferentes rangos de frecuencias, pero los más conocidos son el de 2.4 y 5 GHz.

IoT *Internet of things* (Internet de las Cosas). Nuevo paradigma para la interconexión de dispositivos a una red de datos.

JTAG *Joint Test Action Group* (Grupo de Acción de Prueba de Uniones). Interfaz de hardware que se utiliza mayormente para las depuraciones del software o pruebas de chips.

LoRWPAN *Low Rate Wireless Personal Area Network* (Redes Inalámbricas Personales de Baja Tasa de Transferencia). Estándar de la IEEE que trata de redes de soluciones específicas, cuyo objetivo principal es la baja tasa de transferencia y el ahorro de energía.

Mote Ver definición de nodo sensor.

Nodo sensor Sistema embebido inalámbrico autónomo de recursos limitados. En la literatura les llaman sensor y a los de tamaño pequeño, se les denomina Mote.

PCB *Printed Circuit Board* (Tarjeta de Circuito Impreso). En el diseño de sistemas electrónicos, es una placa de material conductor sobre una superficie no conductora que define un sistema. Está compuesta por diferentes componentes electrónicos como chips, bus de datos, resistencias, etc.

Plug-in Funcionalidades adicionales que se añaden a un sistema de SW o aplicación para aumentar su productividad.

SDK *Software Development Kit* (Caja de Herramientas para el Desarrollo de Software). Es un *plug-in* para un IDE específico de alguna plataforma de hardware.

SoC *System on Chip* (Sistema dentro de un chip). Conjunto de módulos de hardware que realizan tareas específicas, encapsulados dentro de un solo chip. Uno de los propósitos principales de los módulos es el ahorro de energía.

WSN *Wireless Sensor Network* (Red Inalámbrica de Sensores). Consta de la distribución espacial de nodos sensores, a lo largo de un área amplia de terreno con el propósito de monitorizar. Una de sus características importantes es su implementación en topología física de malla.

XML *eXtensible Markup Language* (Lenguaje de Marcado Extensible). Es un metalenguaje basado en una estructura de etiquetas o *tags*, mayormente utilizado para dar formato o extraer datos almacenados o para intercambiar datos entre plataformas incompatibles.

XML *Schema* En el contexto de XML, es un documento que le da sentido a la estructura de *tags*. Es más eficiente que DTD pero difícil de utilizar.

XSL *eXtensible Stylesheet Language* (Lenguaje Extendido para Hojas de Estilo). En el contexto de XML, le da un formato amigable a un DTD o XML *Schema*. Es más eficiente que CSS pero difícil de utilizar.

ZB Acrónimo de la palabra ZigBee que se refiere al protocolo de comunicación inalámbrica que cumple con el estándar IEEE 802.15.4 referente a las LoRWPAN.

Introducción

Desde hace algunos años, se ha vuelto relevante la necesidad de automatizar el monitoreo de entornos internos o externos para control de procesos y toma de decisiones, esto con el fin de atender las problemáticas de interés mundial en las que se requieren soluciones urgentes, tales como: prevención de incendios forestales, reducción de gases de efecto invernadero, infraestructura social como monitoreo de puentes, cuidado de la salud, agricultura como los cultivos hidropónicos, entre otros. Dicha automatización es posible gracias a sistemas embebidos encargados de generar, procesar y comunicar datos mediante diversas tecnologías relacionadas a los campos de las Redes de Sensores Inalámbricas (WSN, *Wireless Sensor Networks*) y el Internet de las Cosas (IoT, *Internet of Things*) [1].

Una probable solución para el monitoreo ambiental, podría ser la implementación de una WSN. Una WSN consta de la distribución espacial de sistemas inalámbricos autónomos de aplicaciones específicas y recursos limitados denominados nodos sensores o Motes. Cada uno de estos Motes tiene la capacidad de sensar distintas variables meteorológicas de forma independiente lo que permite una mayor resolución espacial, además pueden ser ubicados a distancias mayores de 100 metros dependiendo del protocolo de comunicación inalámbrico empleado. El destino de dicha información puede ser otro Mote o una computadora personal (PC, *Personal Computer*) conectada al Internet para el análisis de la información o para la integración de la WSN al IoT.

Actualmente, la implementación de WSN se realiza por medio de *Software* (SW) embebido personalizado para los diferentes nodos que las conforman, pero sin seguir una metodología de desarrollo de manera clara y concisa. Así mismo, comúnmente se emplean dos o más Microcontroladores (MCU, *Microcontroller*) para el desarrollo de cada nodo sensor o Mote. Un MCU se utiliza como radio (dedicado al protocolo inalámbrico), los demás MCU se usan para la adquisición y acondicionamiento de las señales obtenidas por los sensores. Lo anterior, incrementa el costo de las WSN, su consumo de energía y tiempos de desarrollo, causando problemas de mantenimiento y escalabilidad. Lo que se propone es una plataforma de *Hardware*/*Software* (HW/SW) genérica que permita la implementación ágil de una WSN de bajo costo y consumo energético para monitoreo ambiental que siga una metodología que le ayude al programador de aplicaciones, pero también al usuario con poca experiencia en programación, aunque con la capacidad necesaria para poder implementar una WSN. Lo anterior, sería una aportación al conocimiento debido a que en el estado del arte no se encontraron plataformas de HW/SW genéricas en donde se indique claramente el seguimiento de una metodología, además de que no cuentan con la suficiente documentación, como en el caso de *Sprouts* y *Cookie Nodes* [2, 3], que son los prototipos parecidos al desarrollo propuesto en esta tesis, pero que no cumplen con todos los requerimientos de esta tesis como se mencionan más adelante. Para lograr la agilidad deseada al momento de la implementación, la plataforma facilita la configuración de módulos de HW del *Core* XBee por medio de componentes virtuales de HW ya que estos están relacionados directamente por SW, además de generar el SW embebido para los diferentes Motes según el rol que cumplan dentro de la WSN. Los componentes virtuales de HW serán parte de una Interfaz Gráfica de Usuario (GUI, *Graphic User Interface*) que se define mediante el Lenguaje de Marcado Extensible (XML, *eXtended Markup Language*) que conforman parte del Paquete de Herramientas para el Desarrollo de SW (SDK, *Software Development Kit*) específico del *Core* de la plataforma conocido como XBee SDK, el cual es un complemento para el Ambiente de Desarrollo Integrado (IDE, *Integrated Development Environment*) *Codewarrior Development Studio* versión 10.2 (que se le referirá como CWDS v10.2) que se utilizará como herramienta de programación. Es decir, dicha propuesta está basada en componentes virtuales de HW que se crean con el uso de archivos XML en la que varios de ellos en conjunto, definen una GUI que

permitirá realizar las configuraciones del HW de una manera sencilla, generar código embebido para la plataforma e integrar Interfaces de Programación de Aplicaciones, (API, *Application Programming Interface*) puestas a disposición por el fabricante del *Core*.

La comparación en cuanto a costo, consumo energético, mantenibilidad y escalabilidad con respecto a otras plataformas del estado del arte, deberá confirmar que la propuesta es una mejor opción para implementar WSN de bajo costo y consumo energético, además, la metodología utilizada, deberá reducir los tiempos de desarrollo.

Todo lo anterior ayudará a subsanar los inconvenientes presentados en trabajos anteriores, agregar nuevas funcionalidades y facilitar el uso del dispositivo. Adicionalmente, otros proyectos deberán verse beneficiados debido a su versatilidad.

Problemática en redes inalámbricas de sensores para monitorización ambiental.

Existen trabajos de investigación [4, 5] en los cuales se ha propuesto el diseño de Motes para monitorizar variables meteorológicas a largo plazo, sin embargo, dichos desarrollos tecnológicos no cuentan con soporte para poder escalarse, en aspectos como el tamaño de las WSN y la posibilidad de agregar otros sensores al Mote. Además, tales propuestas no permiten el desarrollo de nuevas WSN de una forma ágil, versátil y eficiente, que es lo que se pretende lograr con esta propuesta de una plataforma HW/SW para desarrollo ágil de WSN mediante componentes virtuales.

Trabajo previo.

Como antecedente al presente trabajo de tesis, se toman en cuenta dos trabajos de tesis de licenciatura de la facultad de ingeniería de la UASLP [6, 7] donde se desarrolló el Firmware (FW) y el primer diseño de una Tarjeta Electrónica (PCB, *Printed Circuit Board*) para el Mote. Dentro de los trabajos de tesis mencionados se usó el *Core* Digi XBee PRO S2B (se le referirá como *Core* XBee) así como el protocolo de comunicación inalámbrica ZigBee (se le referirá como ZB), además de que se experimentó con una pequeña WSN ZB. Los avances derivados de dichos trabajos son los siguientes:

- En ambos trabajos se propuso como una opción para medir los efectos del cambio climático el uso de una WSN, para lograr esto se desarrolló un primer PCB del Mote que permitió hacer la prueba de concepto, logrando implementar una WSN ZB de hasta dos Motes *end-device*.
- Con respecto al HW, se seleccionó el *Core*, considerando sus características de bajo consumo, posibilidad de ser programado para tareas específicas y tamaño reducido.
- Se realizó una prueba de concepto con el primer FW desarrollado, con el cuál se implementó una pequeña WSN ZB (considerando máximo dos Motes *end-device*). Se comprobó la viabilidad del FW comparando las lecturas con instrumentos comerciales.
- Se diseñó e implementó la PCB para el Mote usado en la prueba de concepto.

Producto del análisis del prototipo, se detectaron algunos inconvenientes como son:

- Se revisó que para medir los efectos del cambio climático se requieren periodos muy largos de tiempo, además de que se tienen que sensar extensiones muy grandes de terreno. Por lo anterior, las WSN no se consideran como una solución viable para evaluar los efectos del cambio climático; para este tipo de problemáticas se está considerando combinar la tecnología de imágenes satelitales, el uso de drones con sensores para realizar

mediciones de alta precisión y variables atmosféricas en extensiones grandes de terreno junto con una WSN, como se menciona en [8, 9].

- El FW desarrollado y su respectivo código empleados en la prueba de concepto de las tesis de licenciatura [6, 7], necesitaban ser depurados. Se debe realizar una reestructuración que permita aprovechar las diferentes bibliotecas de interfaz, el funcionamiento de los sensores, transferencia y procesamiento de datos, así como brindar la posibilidad de agilizar la corrección de fallas, ajustes y mejoras.
- No se realizaron pruebas de la WSN ZB con más de dos Motes *end-device*, además de que se tenía que configurar de manera explícita la Dirección Física (MAC *Address*) del dispositivo destino de la información con el que los Motes se deben comunicar.
- El FW desarrollado para los diferentes roles de los Motes en la prueba de concepto de los trabajos [6, 7] no cuenta con las siguientes características que permiten la correcta funcionalidad de una WSN ZB: el coordinador no recibía la información conglomerada de la red, el *Router* no permite sensar las variables meteorológicas, no se pueden activar o desactivar los diferentes sensores de los Motes, los sensores analógicos no se les podía configurar la resolución de la precisión, no era posible identificar a los Motes en la red mediante un nombre descriptivo, la exportación e importación del proyecto era complicada.
- Poca eficiencia en el uso de los recursos de cómputo del *Core* XBee, ya que no se permite deshabilitar módulos no deseados, lo que reduciría el consumo de energía.

Derivado de dicho análisis, se concluyó que es posible corregir los inconvenientes encontrados, agregar mayor funcionalidad y facilitar el uso del Mote, por lo que como trabajo de tesis se propone desarrollar una plataforma genérica de HW/SW utilizando el *Core* XBee basada en componentes virtuales para el desarrollo ágil de WSN de bajo costo y consumo de energía, siguiendo los pasos de una metodología de desarrollo como *Co-design* HW/SW y con ello, facilitar el desarrollo e implementación de WSN para monitoreo ambiental proveyendo la documentación correspondiente. Dicha plataforma tendrá sensores en ubicaciones específicas, se le podrán conectar sensores diferentes a los propuestos, aunque habrá que agregar la programación correspondiente mediante una biblioteca que permita incrementar el catálogo de sensores y deberán cumplir con las características de comunicación, similares a los utilizados en el proyecto. Todo lo anterior, permitirá subsanar los inconvenientes encontrados, agilizar la programación y uso de la plataforma HW/SW propuesta, además de ahorrar tiempo, dinero y energía. Adicionalmente, se pretende beneficiar a otros proyectos que deseen aprovechar la plataforma propuesta.

Revisión de la literatura.

En la consulta del estado del arte destacan aquellos trabajos como en [2] donde proponen una plataforma de HW/SW modular, escalar y versátil llamada SPROUTS con conectividad al IoT, implementan WSN *Bluetooth* versión 4.0 (se le referirá como WSN BLE) en ambientes controlados, utilizan dos *Cores* lo que aumenta el consumo de energía además de utilizar una antena de tipo PCB que en [10] indican que introducen ruido a la señal. Como herramienta de SW, se utiliza un middleware para el ahorro de energía, es una plataforma hermética al momento de querer implementarla en otras soluciones. Utilizan HW extra para realizar las mediciones en el consumo de energía. Se pueden implementar WSN ZB, pero no realizaron pruebas de ningún tipo además de que no se indica que metodología de diseño y desarrollo siguen.

En [3], se presenta un Mote que solo se puede utilizar en la realización de pruebas, es decir, el autor lo descarta para implementación de WSN en tiempo real. Está compuesta por cuatro PCB apiladas y unidas por un conector especial, lo que la hace una plataforma poco ágil, versátil y

consumidora de energía además de un alto costo por Mote debido al número de tarjetas y *Cores* que se ocupan.

Planteamiento del problema.

Debido a que la implementación de una WSN no es una tarea sencilla además de que las propuestas encontradas en el estado del arte utilizan soluciones personalizadas y su desarrollo es lento, se propone desarrollar una plataforma de HW/SW genérica de bajo costo y consumo energético que contribuya en la implementación de una WSN ZB para monitorización de fenómenos relacionados con el cuidado del medio ambiente siguiendo los pasos de una metodología que agilice el desarrollo e implementación de la WSN y le facilite el trabajo no solo al desarrollador de aplicaciones si no también al usuario con poca experiencia en programación.

Pregunta de investigación.

Con base a la problemática expuesta con anterioridad es que surgió la pregunta de investigación, ¿existe una plataforma de HW/SW que permita al programador o diseñador de Motes para WSN desarrollar WSN genéricas que cumplan con las características de escalabilidad, portabilidad, mantenibilidad, y que además, permita un desarrollo ágil, versátil y eficiente?

Hipótesis.

Derivada de la pregunta de investigación, se formuló la hipótesis de que, mediante la consideración y seguimiento de una metodología como *Co-design* y el uso de herramientas de SW como un SDK, una API o archivos XML, y módulos de HW es posible desarrollar una plataforma de HW/SW para implementar WSN que cumplan con las características de ser escalables, portables, mantenibles y eficientes.

Objetivos general y específicos.

Objetivo general.

En el presente trabajo de tesis se propone que, mediante el uso de la metodología *Co-design* HW/SW, se pueda desarrollar paso a paso una plataforma genérica de HW/SW con *Core* XBee basada en componentes virtuales de HW, lo que permitirá una implementación ágil y eficiente en el montaje de una WSN ZB con topología de estrella, además que ayude a bajar costos de producción y tiempos de implementación.

Objetivos específicos.

Derivado del punto anterior, surgen objetivos específicos como:

- Uso de la metodología *Co-design* HW/SW para el desarrollo de la plataforma propuesta.
- Desarrollo del código del FW del Mote con el fin de aprovechar las diferentes bibliotecas creadas por el fabricante del *Core* XBee, seleccionar e insertar únicamente el código de los sensores deseados, agilizar la corrección de errores, ajustes y mejoras al código principal de los dispositivos de la WSN ZB con roles de *Router* o *end-device*.
- Análisis del XBee *Project Smart Editor* (se le referirá como *Smart Editor*) que se encuentra dentro del XBee SDK de Digi, para comprender como crear componentes de HW virtuales que se puedan configurar mediante una GUI además de poder insertar eventos dentro del código principal. Los componentes pueden tener diferente funcionalidad y en conjunto, permitirán la integración de una API.
- Con base en lo anterior, ensamblar la plataforma de HW/SW que ayude no solo al programador de aplicaciones si no también al usuario con pocas habilidades en programación en el desarrollo de WSN de forma genérica, ágil y eficiente.

- Pruebas y evaluación de la plataforma propuesta.
- Desarrollo de una WSN ZB mediante la plataforma propuesta.

Organización de la tesis.

El presente trabajo de tesis está estructurado de la siguiente manera:
En el Capítulo 1, se describen los temas relevantes que hubo que analizar para la resolución del problema como aspectos importantes sobre las WSN, otras metodologías para el diseño y desarrollo de sistemas embebidos que se podrían utilizar para el desarrollo del Mote prototipo en lugar de la metodología propuesta, y algunos aspectos importantes de la ingeniería de SW como lo que es una API, un SDK y XML. Después de conocer definiciones y conocimientos previos, en el Capítulo 2 se describe a detalle cómo hacer uso de la metodología *Co-desing* HW/SW y se documentan las herramientas de HW/SW para el desarrollo de la plataforma genérica además del IDE, SDK empleados dentro del trabajo de investigación, dando una pequeña introducción al *Smart Editor* contenido dentro del XBee SDK para el IDE CWDS v10.2, esto con el fin de poder crear componentes de HW virtuales que facilitarán las configuraciones de la plataforma propuesta, inserción de eventos al código principal que facilitarán la agilización y versatilidad en la implementación de los Motes para una WSN ZB. En el Capítulo 3, se abordarán las funciones en lenguaje C necesarias para la mejora y restructuración del código en la construcción de un FW de Mote mucho más confiable y sencillo de modificar que le permita al usuario final una personalización, como cargar el FW del radio del *Core* XBee, el FW del Mote según su rol dentro de la WSN ZB así como realizar pruebas individuales del HW y bloques de SW para asegurar su acoplamiento. Una vez conocidas las herramientas necesarias para el desarrollo de la plataforma propuesta, en el Capítulo 4 se determinan los atributos de la plataforma y se experimenta con una WSN ZB con topología de estrella extendida de un coordinador, un Mote *Router* y cuatro Motes *end-device*. Se realizan las pruebas que validarán el cumplimiento de los requisitos planteados. Se mostrarán los resultados obtenidos en pruebas a campo abierto y en interiores. Por último, se comentan las conclusiones finales sobre el trabajo y se discute el trabajo a futuro.

Capítulo 1

Marco teórico de los campos de computación relevantes

En este Capítulo, se habla de los temas más relevantes que ayudaron a comprender como alcanzar el objetivo del trabajo de investigación. En la Subsección 1.1, veremos de manera general, aspectos importantes de las WSN como la generación de datos sin la intervención del ser humano, como distribuir dispositivos que proveerán dichos datos, una pequeña comparación entre los protocolos de comunicación inalámbrica que cumplen con las requisitos esenciales para las Redes Inalámbricas Personales de Baja Tasa de Transferencia (LoRWPAN, *Low Rate Wireless Personal Area Network*) como son la baja tasa de transferencia de datos y el ahorro de energía; en la Subsección 1.2, se habla de los aspectos importantes en la realización de un proyecto, en el diseño de sistemas embebidos, temas relevantes en el desarrollo de SW embebido como las interfaces entre HW y SW, interfaces que ayuden a las aplicaciones de diferentes tipos a comunicarse entre sí, como lograr la versatilidad del dispositivo, como logar la agilidad en su programación creando GUI además de un FW más amigable para el MCU programable del Mote, según su rol dentro de la red.

1.1 Redes inalámbricas de sensores.

Hoy en día, se requiere conectividad entre computadoras para formar redes de datos con el fin de poder realizar intercambios de información entre ellas. Para la implementación de redes de datos existen medios físicos como los cableados de cobre y fibra óptica, pero el aire también se puede emplear como medio para la transferencia de información. Las redes que utilizan al aire como medio de transporte son las redes inalámbricas y permiten la movilidad de los dispositivos que pertenecen a ella, lo que lo convierte en una ventaja de implementación.

Las WSN son aquellas en que se distribuyen nodos sensores a lo largo de un área extensa de terreno por lo que son cada vez más empleadas para dar soluciones urgentes que provean datos relevantes que ayuden a tomar decisiones importantes como en aquellas problemáticas relacionadas con el medio ambiente.

Debido a su naturaleza inalámbrica, las WSN ofrecen una gran flexibilidad de implementación lo que nos permite seleccionarlas al momento de requerir un análisis de datos de áreas de difícil acceso como bosques o desiertos, con el beneficio de ahorro en infraestructura porque no se requerirá cableado o una gran infraestructura, porque los Motes son pequeños y fáciles de distribuir. En la Figura 1.1, se pude ver un ejemplo de una WSN para la prevención de incendios forestales, donde se tienen *Router* (color verde) y *end-devices* (color rojo). Los *end-devices* son los que están directamente con la problemática, gracias a los sensores que se le conectan, generan la información que transmiten inalámbricamente hacia los *Routers* y estos a su vez hacia un Gateway que permitirá la conexión de la WSN con el IoT.

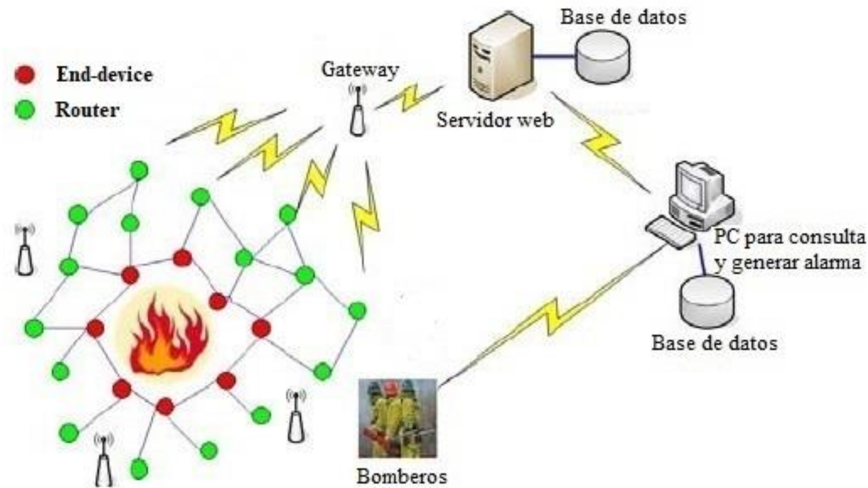


Figura 1.1: Ejemplo de WSN para prevención de incendios forestales [5].

En las siguientes secciones se abordarán de manera sintetizada, los temas relacionados con las WSN que nos permitirán comprender el tema de investigación.

1.1.1 Protocolos de redes inalámbricas de bajo consumo.

La transmisión de información a través del aire es posible gracias a factores como la utilización de protocolos de comunicación inalámbrica (en adelante se utilizará el acrónimo procomi para simplificar su anunciamiento), tipos de modulación, una determinada banda de frecuencia, entre otros.

En lo que respecta a las bandas de frecuencia, existen aquellas que son reguladas con acuerdos internacionales y por los gobiernos de cada país, por lo que se tienen que pedir permisos y pagar por ser utilizadas, como las señales de radio y televisión; también existen bandas de frecuencia de uso libre y no comercial como aquellas que operan dentro de la llamada banda ISM (*Industrial, Scientific & Medical*, en español Industrial, Científico y Médico) en donde las más utilizadas operan dentro de los rangos de 902-928 MHz, 2.4-2.4835 GHz o 5.725-5.85 GHz. Ejemplos de procomi que operan dentro de las bandas ISM son WiFi, *Bluetooth*, ZigBee, entre otros.

Algunas de las problemáticas con las que se enfrentan las redes inalámbricas son las interferencias y seguridad de los datos. En el presente trabajo de tesis, nos concierne el tema de las interferencias. Una interferencia ocurre debido a que una señal inalámbrica se encuentra cerca o dentro de la misma frecuencia que otras señales del mismo tipo, provocando pérdida de datos por lo que se deben de implementar métodos o técnicas que permitan evitar este tipo de situaciones, como aquella proveniente de los celulares donde se controla la energía en el dispositivo transmisor como se señala en [11]. La existencia de las bandas ISM implica una gran probabilidad de interferencia debido a la naturaleza libre de las frecuencias por lo que cada procomi debe implementar técnicas que traten de amortiguar dichas interferencias.

Por otro lado, en el contexto de la WSN, los procomi que se deben de analizar como posible implementación son ZB o BLE (también llamado *Bluetooth Smart*) o posteriores, ya que cumplen con los requisitos del estándar IEEE 802.15.4 que trata de LoRWPAN, por lo que

deben cumplir con un bajo consumo de energía. Para ayudar a tomar una decisión sobre el procomi que se adecue a las necesidades de un proyecto, en la TABLA 1.1 [12], se presentan algunas de las características más importantes, que hay que tomar en consideración de los procomi WiFi, ZB y BLE:

TABLA 1.1: Comparativa de características a considerar de algunos de los procomi.

	PROTOCOLOS DE COMUNICACIÓN INALÁMBRICA				
Característica	**WiFi 5**	**ZigBee**	**BLE 4.0**	**BLE 5.0**	**BLE 5.2**
Banda ISM	5 GHz	2.402-2.4835 GHz	2.402-2.4835 GHz	2.402-2.4835 GHz	2.402-2.4835 GHz
Cobertura en interiores (máx.)	20 m	60 m	10 m	40 m	40 m
Cobertura en exteriores (máx.)	150 m	1.5 km	50 m	200 m	200 m
Tasa de transferencia de datos	6.9 Gbps	250 Kbps	1 Mbps	2 Mbps	2 Mbps
Payload del paquete de datos (en Bytes)	Desconocido	255	20	20	20
Redes de Malla	SI	SI	NO	NO	SI
Total de nodos que se pueden conectar	Depende del Ancho de Banda	255	7	7	22

De la TABLA 1.1, se puede observar de una manera abstracta, las ventajas y desventajas entre los diferentes procomi y debido a la problemática que se aborda en el presente trabajo, esto justifica el porqué en [6] se seleccionó como procomi a ZB el cual se ha utilizado para trabajos posteriores como el presente trabajo de tesis.

Como ya se mencionó, el presente trabajo de tesis está basado en trabajos previos, por lo que en [6] se definió como plataforma de HW al *Core* XBee y como procomi a ZB. En [7], se definieron los sensores, se realizó una prueba de concepto que proporciona una solución específica, y se experimentó con una WSN ZB de dos Motes.

En las siguientes subsecciones, se abordará el concepto de las topologías de red de los procomi BLE y ZB, por ser los más adecuados en la implementación de WSN que cumplen con el estándar IEEE 802.15.4. Las topologías de red definen la manera en que serán distribuidos los nodos, ya sea física o lógicamente, para un mejor desempeño y rendimiento de la red. Con respecto a BLE, solo se mencionan los aspectos más relevantes por ser el competidor más cercano al protocolo seleccionado en el presente trabajo de tesis.

Topologías físicas para BLE.

Originalmente, *Bluetooth* fue diseñado para operar a distancias cortas pero debido a la competitividad y exigencias actuales de las redes personales en las que se requieren dispositivos que puedan ser distribuidos a lo largo de grandes extensiones de terreno, que consuman poca energía, costos de fabricación y mantenimiento bajos, es que hubo la necesidad de que *Bluetooth* evolucionara hacia estas nuevas tendencias. Es por ello que, en 2009, surge BLE v4.0 con el fin de aumentar la distancia y reducir el consumo de energía, aunque no soluciona la distribución de los dispositivos en grandes extensiones de terreno debido a que no cuenta con el soporte para topologías físicas de malla, un concepto esencial en la implementación de WSN para exteriores.

Las topologías físicas soportadas tanto por *Bluetooth* como por BLE [13, 14] son: punto a punto, piconet, scatternet y malla.

Sin adentrarse mucho en la explicación de cada una de ellas, solo se toma la más importante para nuestro trabajo de tesis como lo es la topología física de malla ya que este tipo de topología permite la escalabilidad que se requiere en una WSN para exteriores, como en

situaciones donde podría haber peligro, por ejemplo, en la monitorización de incendios forestales. Las versiones anteriores a BLE 5.1 no cuentan con el soporte completo para este tipo de topología por lo que para poder implementar una WSN BLE con topología de malla, se tienen que utilizar radios BLE v5.1 o posteriores (la última versión de BLE al 10/dic/2021 es la 5.3).

Por lo anterior, aun cuando BLE v5.1 o posterior soluciona el problema de la distribución de nodos a lo largo de un área extensa de terreno, pero no soluciona dos aspectos esenciales para una WSN que cumple con el estándar IEEE 802.15.4, como son:

- Distancia entre nodos: es de máximo 200 m que, comparándolo con ZB, donde cada nodo puede estar separado a una distancia máxima de 1500 metros, este último sería el protocolo a seleccionar cuando se trata de distancia largas entre nodos que pertenecen a la red.
- Mayor cantidad de nodos: con las versiones actuales de BLE, como la 5.2, se pueden unir un máximo de 22 nodos por red y con ZB se pueden unir hasta 255.

Topologías físicas para ZigBee.

En toda WSN ZB siempre habrá 3 roles principales para los diferentes dispositivos que la conformen: el de mayor jerarquía que es el coordinador, el de segunda mayor jerarquía que es el *Router* y el de menor jerarquía el *end-device*. A cada radio del dispositivo que contenga al *Core* XBee se le deberá transferir un FW según el rol que cumplirá dentro de la WSN ZB, esto se logra con la herramienta XCTU provista por el fabricante del *Core*.

A continuación, se hablará de las topologías físicas soportadas por el procomi ZB [15].

- Par: Es la más simple de las 4 topologías soportadas por este protocolo; solo se pueden conectar 2 dispositivos entre sí y uno de ellos tendrá el rol de coordinador mientras que el otro, podría ser *Router* o *end-device*. Físicamente se representa como en la Figura 1.2.

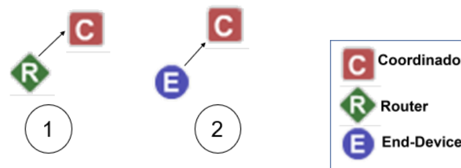


Figura 1.2: Topología de red par del procomi ZB.

De la Figura 1.2, la primera opción es cuando uno de los dispositivos se configura como *Router* y en la segunda opción, como *end-device*.

- Estrella: En esta topología, se pueden conectar varios dispositivos *end-devices* o *Routers* con un punto central como un coordinador o *Router*. Los *end-devices* solo se pueden comunicar con su dispositivo padre (coordinador o *Router*). Físicamente se representa como en la Figura 1.3.

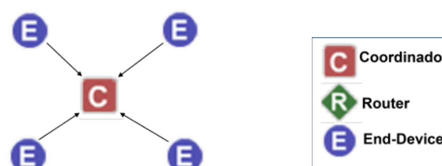


Figura 1.3: Topología de red estrella del procomi ZB.

- Malla: Consta de un coordinador, varios *Routers* conectados entre sí para seleccionar la mejor ruta para los datos, tanto el coordinador como los *Routers* serán los dispositivos padre de los *end-device*, el primero controlando a un grupo específico de *end-devices* y los segundos, a otros grupos; esto último es posible ya que aunque se considera al coordinador como el dispositivo de más alta jerarquía dentro de la WSN ZB, actúa como un *Router* la mayor parte del tiempo. Físicamente se representa como en la Figura 1.4.

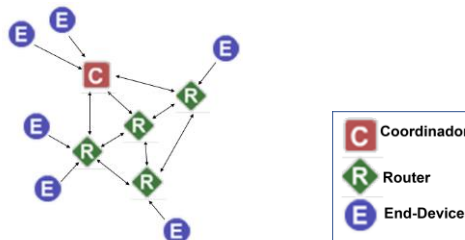


Figura 1.4: Topología de red malla del procomi ZB.

- Árbol de grupos de *Routers*: Esta topología es muy similar a la red de malla con la diferencia que los *Routers* ya no se envían información entre ellos a través de un enlace directo si no que tienen que enviar la información a través del coordinador. Además de lo anterior, cada *Router* puede manejar un máximo de 20 *end-devices* para formar un grupo único (del inglés, *cluster*). Físicamente se representa como en la Figura 1.5.

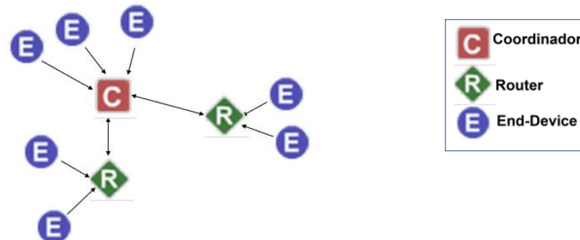


Figura 1.5: Topología de red árbol de grupo de *Routers* del procomi ZB.

Como ya se mencionó en la Subsección 1.1.2, las ventajas de ZB contra las versiones de BLE 5.1 o posteriores, son la distancia entre los nodos y la cantidad de nodos que se pueden conectar en una sola red. Para versiones de BLE anteriores a la 5.1, su mayor desventaja con respecto a ZB es el no poder implementar redes de topología de malla.

En el presente trabajo de tesis, se realizaron experimentos con una WSN ZB en topología de estrella extendida ya que, debido a la falta de recursos, no se implementó una red de Malla,. La red estuvo compuesta de un coordinador, un Mote *Router* y cuatro Motes *end-device*. Queda pendiente comprobar el número máximo de Motes *end-device*, antes de presentar problemas de colisión de paquetes de datos que afecten de forma negativa al desempeño de la red. Lo anterior no se pudo llevar a cabo debido a la falta de recursos disponibles.

1.2 Diseño de sistemas embebidos.

En términos generales, un sistema embebido es un sistema de recursos limitados, dedicado a realizar una tarea específica, compuesto por una computadora que puede ser programada y

trabajar en tiempo real. Un ejemplo de este tipo de sistemas es aquel que se encuentra en electrodomésticos como el horno de microondas. Debido a su naturaleza cambiante y a la complejidad que en la actualidad demanda el diseño y desarrollo de este tipo de sistemas, las metodologías empleadas no pueden ser las mismas que para aquellas empleadas para sistemas de propósito general como las PCs [16].

En el diseño, desarrollo e implementación de este tipo de sistemas, existen dos metodologías principales como las metodologías de diseño de HW/SW y las metodologías para la planeación de proyectos de desarrollo de HW/SW [17]. Con las primeras se intenta armar la arquitectura del sistema y con las segundas, se identifican los pasos a seguir en el diseño y desarrollo de un producto final.

En las siguientes subsecciones solo se mencionarán las metodologías correspondientes al diseño de HW/SW. Las metodologías para la planeación de proyectos no es tema del presente trabajo de tesis, pero es necesario entender la diferencia de estos dos conceptos.

1.2.1 Metodologías para el diseño de sistemas embebidos.

Debido a la gran importancia en nuestra vida cotidiana que tienen hoy los sistemas embebidos, ha ido creciendo el interés por diseñar y desarrollar este tipo de sistemas para que sean robustos, confiables y de calidad que cumplan con requerimientos como tamaño, ahorro de energía y bajos costos de producción [6], entre otros, lo que ha originado idear metodologías y técnicas que permitan mayor rapidez en su diseño, desarrollo e implementación. A continuación, se dará una breve introducción de algunas de las metodologías de diseño de HW/SW de sistemas embebidos que se consideran adecuadas [16, 17] para este tipo de soluciones.

1.2.1.1. *Bottom-up.*

Es una estrategia de diseño donde se plantea la creación del producto, donde no es necesario tener una imagen clara y completa del mismo, pero se requiere de una gran intuición para iniciar creando un componente a detalle y conforme se vaya avanzando en la solución, se van creando subsistemas o módulos que se irán uniendo con el tiempo, hasta llegar al producto final [19]. Lo anterior es posible debido a que las actividades de desarrollo se realizan comenzando una después de terminar una anterior (actividades en serie). Se recomienda utilizar esta metodología cuando haya que agregar componentes nuevos a un sistema funcional o en la creación de productos innovadores. Esta metodología ha sido utilizada por las empresas desde hace mucho tiempo por lo que una ventaja de utilizarla es que no hay que realizar capacitaciones para poder aplicarla, pero su mayor desventaja es que cualquier error detectado se realiza en etapas muy avanzadas del desarrollo. En la Figura 1.6 se puede observar un diagrama general de la metodología.

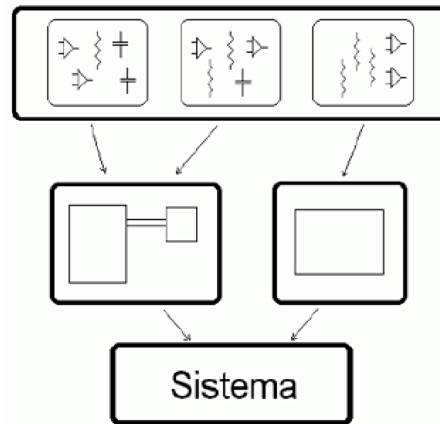


Figura 1.6: Diagrama general de la metodología de diseño y desarrollo *Bottom-up*.

1.2.1.2. *Top-down*.

Es una estrategia de diseño donde se plantea la creación del producto, en donde es necesario tener una imagen clara y completa del mismo para comenzar desde un nivel de abstracción alto y conforme se vaya avanzando en la solución del producto final, se vayan creando sistemas o módulos que producirán con el tiempo, otros subsistemas que darán un mayor nivel de destalle. Al emplear esta metodología, las actividades de desarrollo se realizan de manera paralela por lo que no es necesario terminar una tarea para iniciar otra, lo que optimiza los procesos de producción [18]. Esta metodología se debe emplear en diseño de circuitos y de SW. Hace algunos años, esta metodología no era muy utilizada por las empresas por lo que una desventaja de utilizarla es que quizá haya que realizar capacitaciones y cambiar el modo de pensar de todos los involucrados para poder aplicarla, pero, en contraste, su mayor ventaja es que cualquier error detectado se realiza en etapas tempranas del desarrollo además de permitir la modularidad y reutilización de módulos. En la Figura 1.7 se puede observar un diagrama general de la metodología.

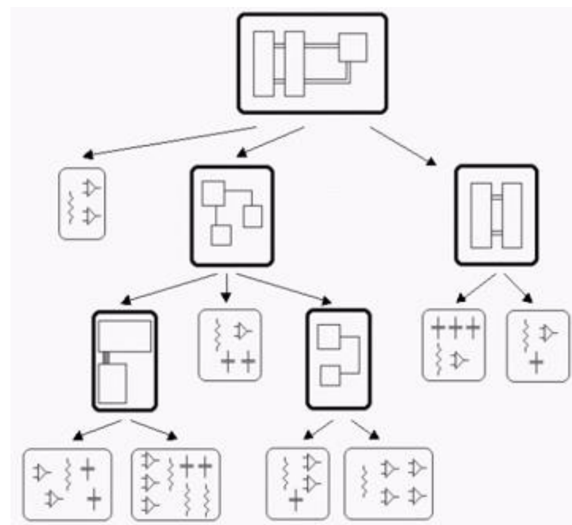


Figura 1.7: Diagrama general de la metodología de diseño y desarrollo *Top-Down*.

1.2.1.3. *Co-design.*

El propósito principal del concepto de la metodología *Co-design* se basa en la interpretación del prefijo co ya que trata de dar a entender que debe haber coordinación, concurrencia, complejidad y correctitud (en inglés, *correctness*) [19].

Por otro lado, cuando se presenta la oportunidad de crear un prototipo de HW/SW es muy probable que algunas personas estén acostumbradas a utilizar metodologías como *bottom-up* o *top-down*, pero no de manera conjunta. *Co-design* es una estrategia de modelado jerárquico mayormente empleada en el diseño de HW/SW de sistemas embebidos debido al diseño e implementación concurrente entre ambas ingenierías, combinando metodologías como *top-down* y *bottom-u*p para evitar incompatibilidades entre módulos, un prototipado más rápido y una de sus grandes ventajas es la simulación antes de la producción en masa [20, 21, 22]. Además, metodologías como esta deben permitir la admisión de requisitos de diseño incompletos o cambios en los mismos, que surgen durante las primeras etapas o durante el desarrollo.

En el diseño de plataformas de HW/SW es importante el costo, consumo energético, usabilidad y portabilidad ya que con ello se logra un buen prototipado, buena documentación, mantenibilidad además de agilizar algún cambio en la implementación, en caso de requerirse.

Las fases generales del desarrollo en la aplicación de la metodología *Co-design*, son las siguientes:

- Co-especificación: Capturar los requerimientos del sistema.
- Definición de la arquitectura: Implementación del estilo y arquitectura del sistema.
- Particionamiento. Dividir la funcionalidad del sistema en módulos de HW y SW.
- Co-síntesis: Integrar como uno solo, todo el particionamiento realizado.
- Co-simulación: Simular el producto que se está desarrollando. Esto se puede llevar a cabo con la ayuda de herramientas Diseño Asistido por Computadora (CAD, *Computer Aided Design*).
- Co-verificación: Para comprobar un correcto acoplamiento de los módulos de HW, SW e interfaces correspondientes. Este paso se debe de realizar en cada fase del desarrollo [23].

De lo anterior se deriva el flujo general de diseño (pasos a seguir) aplicando la metodología *Co-design*, que se encuentra en [23] y se ilustra en la Figura 1.8.

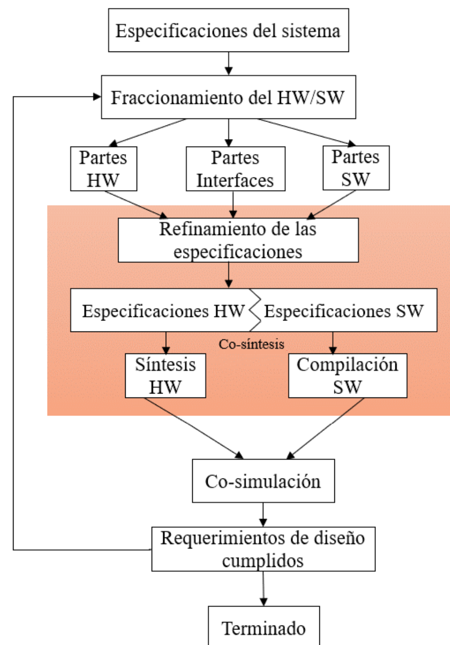


Figura 1.8: Flujo de diseño de la metodología *Co-design*.

Cabe aclarar que el flujo anterior se puede utilizar como modelo, pero hay que adecuarlo según las necesidades de los proyectos a realizar, es decir, puede cambiar para otro tipo de soluciones como se menciona en [24] solo hay que tomar como base el modelo o seguir las fases generales mencionadas con anterioridad.

El flujo de la Figura 1.8, nos indica que primero hay que especificar los requerimientos del sistema a diseñar y desarrollar para en seguida, pasar a la fase de particionamiento donde, basándose en los requerimientos de la fase anterior, ahora se tienen que separar en HW, SW e interfaces; una vez clasificados, la siguiente fase es seleccionar el HW, SW e interfaces posibles y disponibles; la siguiente fase se seleccionan aquellos requerimientos específicos para la solución que se desea proponer; una vez seleccionados el HW, SW e interfaces específicas, lo que seguiría es el desarrollo de la solución propuesta; después, hay que integrar todo lo realizado para posteriormente, pasar a la fase de Co-simulación en donde hay que realizar pruebas y verificación; si todo funciona de acuerdo a los requerimientos especificados, se llegará al final del proceso; sino, se deberán realizar algunas iteraciones más, realizando solo aquellos pasos en donde se haya detectado una falla que evitó que no se pudieran cumplir los requerimientos iniciales.

En el presente trabajo de tesis, se aplicó esta metodología siguiendo cada una de las fases ya mencionadas, para con ello asegurar la compatibilidad entre los diferentes módulos, un prototipado más rápido, usabilidad y escalabilidad del prototipo creado.

1.2.1.4. Sistema en un Chip (SoC).

Desde hace algunos años, la complejidad de los Circuitos Integrados (IC, *Integrated Circuits*) ha ido en aumento esto con el fin de satisfacer las nuevas tendencias tecnológicas. El avance en la tecnología de la impresión de circuitos sobre obleas de silicio permite diseñar y crear chips cada vez más sofisticados, por lo que hubo la necesidad de seguir metodologías de diseño y reutilización como la de los Sistemas en un Chip (SoC, *System on Chip*). Esta metodología consta de la interconexión de lo que se conoce como bloques

de Propiedad Intelectual (IP, *Intellectual Property*) los cuales pueden ser diseñados y desarrollados por diferentes fabricantes y para que sean utilizados en diversos proyectos, deben ser generales, configurables o programables. Los bloques IP deben llegar diseñados y verificados previamente para ser interconectados por el diseñador del SoC. En la parte del SW para SoC, los diferentes módulos también deben ser reutilizables y podrían ser bibliotecas, eventos, entre otros. Un SoC debe contener como mínimo dos o más procesadores, bloques de memoria, bloques analógicos, convertidores Analógico-Digital y Digital Analógico (*Analog to Digital Converter*-ADC y *Digital to Analog Converter*-DCA, respectivamente) y modos de configuración para el ahorro de energía. Debido a las características mínimas mencionadas con anterioridad, el *Core* XBee modelo XBP24BZ7 utilizado en la plataforma propuesta, es considerado un SoC.
Los campos de aplicación de los SoC son muy diversos algunos ejemplos son las WSN, redes neuronales, Inteligencia Artificial (AI, *Artificial Intelligence*), IoT, entre muchos otros.
Una desventaja en su implementación es la seguridad de la información [25] y aunque arquitecturas de procesadores como ARM implementan un nivel de seguridad por HW, continúa siendo vulnerable ante aquel usuario que cuenta con las habilidades necesarias para irrumpir dentro de un sistema.

Como se mencionó, el presente trabajo se basa en trabajos previos como en [7] donde se definió trabajar con la metodología *Co-design* para el diseño y desarrollo del prototipo del Mote por ser la metodología más adecuada para este tipo de proyectos debido a que el diseño y desarrollo del HW y SW se realizan de manera concurrente lo que significa que no hay necesidad de esperar a fabricar al HW para seleccionar las herramientas de SW que ayuden en la programación, además de la sencillez en sus iteraciones permite que se puedan aceptar cambios en fases avanzadas del desarrollo o agregar nuevas funcionalidades al diseño original sin comprometer a todo el diseño. Mas adelante, en el Capítulo 2, se detalla cómo llevar a cabo las diferentes fases de la metodología, aplicándolas el presente trabajo de tesis.

1.2.2. Ingeniería de SW embebido.

Cuando se habla de ingeniería de *Software* [21] se refiere a los métodos, técnicas y herramientas necesarios para el desarrollo de SW de sistemas de cómputo. Los sistemas pueden ser de propósito general o de propósito particular como los sistemas embebidos. El desarrollo de SW para sistemas de propósito general no es el mismo que para el de sistemas embebidos debido principalmente a que en estos últimos, los recursos de HW son limitados y la programación es cercana al mismo HW.
En la presente subsección, se hablará de algunas de las técnicas y herramientas empleadas en el desarrollo del Mote para monitoreo ambiental del trabajo de tesis.

1.2.2.1. Interfaz para la Programación de Aplicaciones (API)

Hoy en día [26], se requiere estar conectados para compartir información e interactuar sin importar la distancia que nos separe y de esta manera obtener una mejor productividad por lo que nace la necesidad de crear aplicaciones de mayor calidad lo que provoca que cada vez sea más necesaria la modularidad de las mismas para optimizar los tiempos de desarrollo y poder reutilizar los módulos en otros proyectos.
Con el pasar del tiempo, las WSN han ido teniendo cada vez más aceptación, esto ha requerido que los nodos sensores sean cada vez más pequeños, portables y sencillos de programar. Una de las razones para que estos nodos sensores puedan comunicarse, son las aplicaciones que se ejecutan dentro de ellos. Estas aplicaciones las crean programadores a

nivel de aplicación, a quiénes se les dificulta programar a nivel de HW debido a su formación académica, por lo que es necesario un nivel de abstracción que les facilite la creación de dichas aplicaciones. Ese nivel de abstracción se obtiene con la creación de una API debido a que estas se diseñan para proveer servicios, información de bases de datos o como su nombre lo indica, permiten interactuar entre sí a aquellas aplicaciones que no tienen algo en común.

Las API son conjuntos de funciones creadas en cierto lenguaje de programación que ofrecen servicios a otras aplicaciones y de esta manera, facilitan la creación de estas últimas. Un ejemplo de servicio que podrían ofrecer las API sería la medición de radiación solar o de presión atmosférica mediante el uso de sensores o la extracción de datos de una base de datos sin tener que aprender lenguajes de consulta como MySQL o proveer de elementos gráficos como lo hace la API del Sistema Operativo Windows conocida como Win32 o Win64.

En el contexto de las WSN, se diseña una API cuando el programador de aplicaciones carece de los conocimientos para la creación de un protocolo de comunicaciones relacionado con las redes de datos o cuando necesita servicios específicos de la misma, como el ruteo de los paquetes de información, estado de la red, entre otros.

Por lo anterior es que se pueden crear API para bibliotecas de SW, lenguajes de programación, sistemas operativos, HW y servicios de Internet (a estas últimas, se les conoce como REST API).

El CWDS v10.2 junto con el XBee SDK contienen API que ayudan a acelerar el tiempo de creación de aplicaciones para el *Core* XBee. Para mayor información sobre el XBee SDK, ver la Subsección 1.2.2.3.

1.2.2.2. Capa de Abstracción de HW (HAL)

La mayoría de los sistemas relacionados con la computación que se basan en una arquitectura de HW/SW, siguen un modelo por capas que puede ir en forma ascendente o descendente, en donde la capa más inferior del modelo es la que va relacionada con el HW (lo tangible) y yendo en forma ascendente, se llega a la capa de aplicación (lo intangible).

La Capa de Abstracción de HW (HAL, *Hardware abstraction layer*) se podría definir como una interfaz de SW para la comunicación entre las capas de aplicación (superiores) con las capas de HW (inferiores) del modelo antes mencionado. Es un grupo de API que realizarán tareas específicas. Una HAL evita que el programador de aplicaciones tenga que conocer a detalle el HW con el que va a interactuar proporcionándole a las funciones que utilizará, el nivel de abstracción que le evitará leer la hoja de datos (en inglés, *datasheet*) del dispositivo, p. ejem. un microprocesador (se le referirá como μp).

Un concepto similar a la HAL son el nano-kernel y los controladores de dispositivos. El primero trata sobre las rutinas de interrupción necesarias en todo sistema computacional y puede ser parte de una HAL o, en el contexto de los OS, es la HAL; el segundo, es un nivel de abstracción para las entradas y salidas de algún dispositivo por lo que parte del código dependerá del HW pero la parte que falte, será independiente del mismo. Es por lo anterior un controlador de dispositivos pertenece parcialmente a una HAL. Las API para HAL pueden dividirse por categorías como la API HAL Kernel que están relacionadas con operaciones sencillas como leer, escribir y modificar; API HAL administradoras de interrupciones y se les conoce como nano-kernel; API HAL de E/S realizan la configuración y acceso a los dispositivos de E/S como las memorias; API HAL que administran los recursos de HW como la verificación de las baterías, configuración de la velocidad del μp; por último, las API HAL del tiempo de diseño.

En el contexto de SoC, una HAL permite la reusabilidad de los módulos de SW [27].

Una de las funciones de la HAL es poder ejecutar SW dentro de un HW para el cuál no fue creado. Ejemplos de uso de una HAL:

- Necesaria en emulaciones como la realizada dentro del IDE CWDS v10.2 con el XBee SDK instalado (ver Subsección 1.2.2.3 sobre lo que es un SDK) que proporciona el fabricante del *Core* XBee para crear aplicaciones óptimas para él.
- Controladores de dispositivos.
- Cuando un sistema embebido utiliza una HAL permitirá un mayor uso de dispositivos que puedan utilizar el sistema sin modificarle nada.

1.2.2.3. Caja de Herramientas para el Desarrollo de SW (SDK)

Un ecosistema de SW [28] es cuando en una empresa se utilizan aplicaciones creadas internamente o adquiridas a terceros, que le permitirán una mayor productividad y soporte para sus actividades empresariales, por tanto, mayor aprovechamiento de los recursos disponibles, ahorros en tiempo y costos. La mayoría de las veces estas aplicaciones crean ecosistemas de SW heterogéneos en donde no se prevé la integración de las mismas. La integración es una parte muy importante en la reutilización de bloques de código ya que estos permitirán la creación de nuevas aplicaciones u optimización de las existentes.

La SDK es un conjunto de herramientas de SW específicas para una plataforma que permite una mayor productividad ya que agiliza el trabajo de los desarrolladores permitiéndoles crear aplicaciones de una manera ágil, sencilla y estandarizada.

Es por lo anterior que, para mejorar la productividad en las empresas, estas tienen que realizar un mantenimiento de sus aplicaciones. La IEEE clasifica al mantenimiento de aplicaciones en 3 tipos: correctivo, perfectivo y adaptativo. Sin adentrarse mucho en el significado de cada uno de ellos, solo se toma el de mayor relevancia para el presente trabajo de tesis como lo es el mantenimiento adaptativo ya que con este tipo de mantenimiento les permitirá a las empresas crear sus propias herramientas de SW reutilizando las herramientas existentes.

En el desarrollo de SW actual, la creación de un SDK permite mejorar la funcionalidad de alguna aplicación de una manera ágil y sencilla, contribuyendo de esta manera, en la rapidez del proceso de desarrollo. Un SDK deberá contener como mínimo: un compilador, proyectos o códigos de ejemplo, bibliotecas, herramientas de análisis y pruebas, documentación, un depurador y al menos una API. Un ejemplo de SDK de código abierto es el que proporciona Microsoft para la creación de aplicaciones para computadoras personales y servicios web, conocido como .NET.

Con base en lo anterior, el CWDS v10.2 es un IDE creado por la empresa *Freescale*/NXP para programar a sus Microcontroladores (MCU, *Microcontrollers*) que se basa en un IDE de código libre como Eclipse. El *Core* XBee contiene internamente un MCU de la empresa *Freescale*/NXP (se le referirá como MCU programable) que puede ser programado con dicho IDE pero para una mayor integración entre el MCU programable y el *Core* XBee, el fabricante de este último proporciona lo que se conoce como XBee SDK que cuenta con las herramientas básicas de todo SDK además de un analizador de XML para la creación de GUI que permitirán agregar y configurar de manera ágil y sencilla, componentes virtuales de HW para todo proyecto que seleccione al *Core* XBee como solución. Consultar el Anexo I para conocer cómo crear una GUI que permita realizar lo anteriormente mencionado.

1.2.2.4. Lenguaje de Marcado Extensible (XML)

De los autómatas programables [29], sabemos que la semántica se relaciona con el significado de los símbolos y la sintaxis es la organización de esos símbolos en relaciones significativas.

XML se utiliza como una sintaxis que inherentemente expresa el concepto del contenido. Sin entrar a detalle sobre lo que es una sintaxis, a continuación, se mostrará un ejemplo de cómo darle concepto a un contenido:

Si tenemos:

```
¡Hola!

¿Cómo estás?
```

En formato XML se expresaría:

```
<?xml version=”1.0” encoding=”utf-8”?>

<humano>

   <saludo> &#33;hola&#161; </saludo>

   <saludo>&#191;C&#243;mo est&#225;s&#63; </saludo>

</humano>
```

Y de esta manera se le estaría dando concepto al contenido.

XML proporciona un enfoque más estructurado y dinámico para la creación de contenido que permite al desarrollador definir y ordenar datos en lugar de simplemente mostrarlos [30]. Por lo anterior, es importante tomar en consideración que XML solo es una forma de construir oraciones de significado específico y definido.

XML se asocia comúnmente con la web pero no es el único contexto en el que se utiliza. Por lo general, se realiza una combinación de XML con algún lenguaje de programación orientado a objetos para una mayor facilidad y nivel de abstracción al momento de crear aplicaciones además de aumentar la portabilidad y estabilidad [31].

El actor más importante de todo documento con formato XML es el *tag* (etiqueta, que para una mayor comodidad, en adelante, se utilizará la palabra en inglés, en género masculino) la cuál es jerárquica [32].

Por si solo, un documento XML no tiene ningún significado. Representa datos no interpretados. Su valor radica en los procesadores que entienden el documento, lo utilizan para realizar alguna acción como lo hacen los navegadores de Internet.

Se pueden crear nuevos lenguajes basados en XML, como los lenguajes de marcado para servicios web siendo el más conocido el Lenguaje de Marcado de Hipertexto (HTML, *HiperText Markup Language*) y con ello proporcionarle estructura a la información. Estos beneficios también se pueden aprovechar en varios otros contextos, por lo que XML tiene un uso generalizado para definir sintaxis para estándares de comunicación, entradas de bases de datos, lenguajes de programación, etc.

Para que uno o varios *tags* tengan significado o se pueda crear un lenguaje como HTML, primero hay que seguir los siguientes pasos generales:

1. Crear la estructura de los *tags*.
2. Crear lo que se conoce como Definición del Tipo de Documento, (DTD, *Document Type Definition*) o XML *Schema* (más eficiente comparándolo con DTD pero difícil de utilizar).
3. Validar el DTD o XML *Schema*. Para poder validarlos se necesitan programas como Editix o Notepad plus plus.
4. Crear una hoja de estilo. Para poder darle un formato amigable al código XML, se tiene que crear una Hoja de Estilo en Cascada (CSS, *Cascading Style Sheet*) o un Lenguaje Extendido para Hojas de Estilo (XSL, *eXtensible Stylesheet Language*) el cual es más eficiente que CSS pero difícil de comprender.
5. Una vez creado y validado el DTD o XML *Schema*, ya se puede utilizar como lenguaje basado en XML como el caso de HMTL.

Una vez que se realizaron los pasos anteriores, para comenzar a escribir un documento XML, se puede utilizar un editor de texto plano como Notepad de Windows y buscar un analizador XML que lo pueda leer (por ejemplo, el navegador Firefox).

El IDE CWDS v10.2 junto con el XBee SDK, permiten agregar componentes virtuales de HW además de funciones de eventos en lenguaje C a las aplicaciones diseñadas para el *Core* XBee, con tan solo crear un archivo XML que permitirá utilizar GUI para las configuraciones comunes de los diferentes componentes virtuales de HW que se creen y de esta manera facilitar el desarrollo de aplicaciones. Para más información, ver el Anexo I.

1.2.2.5 Depuración y verificación de SW embebido

En todo proyecto, el ahorro de tiempo en el diseño y desarrollo de un producto son primordiales para obtener un costo-beneficio positivo. Con lo que respecta al diseño y desarrollo de sistemas embebidos, con el pasar de los años, se han vuelto cada vez más demandantes comparándolos con los primeros sistemas embebidos creados, por lo que el presupuesto disponible para un proyecto de tales magnitudes será consumido en un 80% por los procesos de desarrollo, depuración y verificación, esto provocará retrasos tanto en la productividad como en la verificación, situación que hay evitar ya que se deben de llevar a cabo de manera conjunta.

El proceso de depuración es necesario en todo proyecto de SW [33] en donde hay que encontrar y solucionar aquel código que infrinja una o algunas de las especificaciones en los requerimientos o porque hubo cambios funcionales en las especificaciones debido a las diferentes versiones del sistema que se está desarrollando.

La depuración se puede llevar a cabo dentro de 3 actividades del desarrollo de SW embebido como son: durante el proceso de codificación, durante las últimas etapas de prueba y al momento de activar al dispositivo dentro de su entorno real.

Es por lo anterior que, al momento de depurar se encontrarán con dos de las causas de fallas más comunes en el SW como son: sobreescrituras a memoria y defectos en el HW o SW adquiridos a terceros.

El IDE que se utilice en el proceso de desarrollo, podría proveer pequeñas ayudas con las depuraciones de código, por ejemplo, mostrar errores en las sentencias mediante un código de colores o mensajes de error cuando existen errores en la sintaxis. Si se utiliza como lenguaje de programación a C, una manera antigua de depurar código (aún utilizada) sería añadiendo sentencias de la función *printf()* en puntos estratégicos del código; si la depuración se realiza directamente en el HW destino, por lo general se utiliza HW adicional junto con la interfaz JTAG, que es muy utilizada en la realización de pruebas de PCBs.

Con respecto a la verificación de SW, esta se define como la comprobación de un sistema o programa en proceso de desarrollo que satisface los requerimientos solicitados. Debido a que la capacidad tecnológica sufre un cambio en HW relativamente continuo, esto aumenta la complejidad y cambios continuos en sus requerimientos. Lo mismo sucede con el SW, lo que hace necesario cortos periodos de tiempo en el diseño que obliga a minimizar los tiempos de depuración y verificación de SW. En el desarrollo de sistemas embebidos, es complicado capturar los requerimientos debido a la falta de experiencia del desarrollador ya que podría no estar familiarizado con la ingeniería de SW.

En la verificación realizada del SW para el trabajo de tesis, se realizaron pruebas en tiempo real durante el proceso de desarrollo y la ejecución del programa, resultando todas satisfactorias.

En el presente Capítulo, se describieron las herramientas de HW/SW y temas relevantes que son la base del aprendizaje para la presente tesis de maestría. Se aprendió que, con respecto a la ingeniería de SW, es necesario planear, diseñar y desarrollar de forma ordenada la creación de un producto final, conocer las herramientas que ayudarán aumentar la productividad, disminuir tiempos y costos de producción. En el siguiente Capítulo se continuará con las fases del proceso de desarrollo para cumplir con los objetivos planteados.

Capítulo 2

Metodología y materiales para el desarrollo de la plataforma propuesta

Una vez conocidos los temas más relevantes que ayudaron a la realización del trabajo de tesis, en el presente Capítulo, se seguirá cada una de las fases de la metodología de *Co-design* HW/SW basado en el modelo de la Figura 1.8, adecuándolo al contexto del diseño y desarrollo de una plataforma genérica de HW/SW de componentes virtuales propuesta en el presente proyecto. Se inicia estableciendo las fases a seguir de dicha metodología, continuando a detalle con cada una de ellas ya que con ello se guiará paso a paso en el desarrollo al programador de aplicaciones o usuario del sistema.

2.1 Flujo de trabajo de la plataforma basado en *Co-design* HW y SW.

Debido a la demanda actual en el diseño de sistemas embebidos y que estos son cada vez más complejos, es necesario seguir alguna metodología de diseño de HW/SW para lograr agilidad, usabilidad, mantenibilidad y escalabilidad del producto final.
En el presente trabajo de tesis, se tomó como base la metodología *Co-design* HW/SW [34], ya que el desarrollo del HW y SW para la plataforma propuesta se realiza de manera concurrente y flexible, permitiendo corregir errores o agregar nuevas funcionalidades.
Los trabajos previos permiten utilizar al Mote de una manera básica, sin embargo, era necesario agregar nuevas funcionalidades y corregir inconvenientes detectados, ya mencionados en el Capítulo de introducción. Debido a lo anterior, al seguir la metodología *Co-design* HW/SW se obtienen ventajas como ahorro en tiempo de diseño y costos de producción, además de facilidad en el desarrollo de los Motes para que integren características como usabilidad, mantenibilidad y escalabilidad requeridos en este proyecto.
Nuestra plataforma de HW/SW con componentes virtuales de HW no solo pretende ser una solución para el monitoreo ambiental sino, ser una solución genérica que pueda ser utilizada en otro tipo de proyectos donde sea necesaria una monitorización como son en el cuidado de la salud, en la agricultura, en infraestructura civil, en ciudades inteligentes o en las fábricas [36, 37]. Es por lo anterior que, para su diseño y desarrollo, es necesario establecer los pasos necesarios que nos permitan alcanzar los objetivos propuestos.
En la Figura 2.1, se adapta el esquema de la Figura 1.8 para el diseño y desarrollo de la plataforma HW/SW propuesta, en dicha figura se puede observar el flujo de trabajo a seguir.

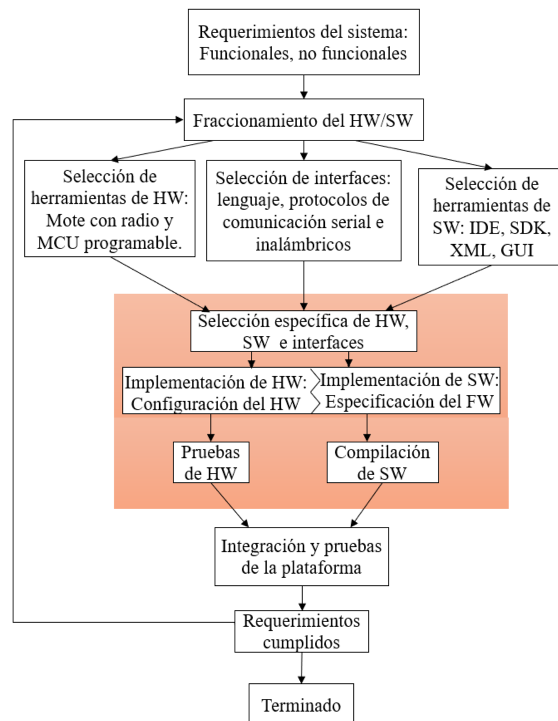


Figura 2.1 Flujo de trabajo para el desarrollo de la plataforma HW/SW propuesta.

Como se puede observar en la Figura 2.1, se inicia el proceso estableciendo los requerimientos de la plataforma que es lo que se ve en la Subsección 2.1.1. En subsecciones posteriores, se irá explicando cómo se va aplicando cada una de las fases del flujo de trabajo mostrado con anterioridad.

2.1.1 Requerimientos del sistema.

La primera fase de la metodología de la Figura 2.1, es establecer los requerimientos de la plataforma, conocer cuales son funcionales, no funcionales y sus restricciones, para generar su especificación general. A continuación, se comentarán los requerimientos funcionales, no funcionales y sus restricciones, de nuestra plataforma:

- Requerimientos funcionales:
    1. Considerar un catálogo elemental de sensores que contemple las variables meteorológicas tales como temperatura ambiental, temperatura del PCB, presión barométrica, humedad y radiación solar.
    2. Agregar y configurar módulos de sensores por medio de una GUI.
    3. Que los módulos de sensores tengan interfaz analógica o digital y que estos últimos tengan protocolos seriales I2C, SPI y *1-Wire*.
    4. Tener la posibilidad de agregar sensores distintos a los señalados en el punto (1).
    5. Que la plataforma sea capaz de generar el FW específico para los Motes, tanto *Routers* como *end-devices*, agilizando su implementación y ahorrando energía mediante la desactivación de los módulos de sensores.
    6. Que la plataforma tenga la opción de configurar el bajo consumo de energía.
    7. Una tarjeta de desarrollo de Motes que integre conectores para el *Core* seleccionado e interfaces tanto digitales como analógicas para los módulos de sensores.
    8. Que el *Router* además de las tareas que realiza en la WSN, tenga la posibilidad de medir variables meteorológicas mediante la conexión de módulos de sensores.
    9. Que la información generada por la WSN se reciba en una sola exhibición.

10. Que se pueda ver en pantalla la información generada por la WSN.
11. Que la información proveniente de la WSN se guarde dentro de un archivo.

- Requerimientos no funcionales:
  1. Que la plataforma sea modular, estética e intuitiva.
  2. Que permita plantear una metodología didáctica para el desarrollo de WSN.
  3. Que la plataforma cumpla con el estándar IEEE 802.15.4 (LoRWPAN) y que el rango de alcance en la intemperie entre los Motes sea de por lo menos 1 km.
  4. Que una imagen sirva como guía que facilite la conexión de los módulos de sensores (*pinmap*).
  5. Que la plataforma permita realizar actualizaciones de HW/SW o corregir errores, es decir, que sea flexible para realizar cambios.
  6. Que la plataforma sea escalable, es decir, que los recursos de HW y SW faciliten el desarrollo de nuevas características.
  7. Que el desarrollo de HW y SW de la plataforma quede documentado para poder reutilizarlo o replicarlo en otros proyectos.
  8. Que la plataforma permita implementar de forma ágil una WSN de topología estrella para incorporar a los Motes de manera sencilla.

Una vez conocidos los requisitos funcionales, no funcionales y restricciones de nuestra plataforma, hay que clasificar los que son de HW y separarlos de los que son de SW. Esto se aborda en la Subsección 2.1.2.

2.1.2 Fraccionamiento del HW y SW.

La segunda fase de la metodología es clasificar como HW o como SW, el listado de requerimientos funcionales y no funcionales, establecidos en los Subsección 2.1.1. A continuación, se muestra la lista de los requerimientos, clasificada por HW o por SW.

- Requerimientos de HW:
  1. Una tarjeta de desarrollo de Motes que integre conectores para el *Core* seleccionado e interfaces tanto digitales como analógicas para los módulos de sensores.
  2. Que la plataforma cumpla con el estándar IEEE 802.15.4 (LoRWPAN) y que el rango de alcance en la intemperie entre los Motes sea de por lo menos 1 km.
  3. Tener la posibilidad de agregar sensores distintos a los señalados en los requerimientos de SW (ver más adelante).
  4. Que los módulos de sensores tengan interfaz analógica o digital y que estos últimos tengan protocolos seriales I2C, SPI y *1-Wire*.
  5. Que la plataforma permita implementar de forma ágil una WSN de topología estrella para incorporar a los Motes de manera sencilla.
- Requerimientos de SW:
  1. Que permita plantear una metodología didáctica para el desarrollo de WSN.
  2. Que la plataforma sea modular, estética e intuitiva.
  3. Agregar y configurar módulos de sensores por medio de una GUI.
  4. Que la plataforma sea capaz de generar el FW específico para los Motes, tanto *Routers* como *end-devices*, agilizando su implementación y ahorrando energía mediante la desactivación de los módulos de sensores.
  5. Considerar un catálogo de sensores elemental que contemple las variables meteorológicas tales como temperatura ambiental, temperatura del PCB, presión barométrica, humedad y radiación solar.
  6. Que la plataforma tenga la opción de configurarle el bajo consumo de energía.

7. Que el *Router* además de las tareas que realiza en la WSN, tenga la posibilidad de medir variables meteorológicas mediante la conexión de módulos de sensores.
8. Que la información generada por la WSN se reciba en una sola exhibición.
9. Que se pueda ver en pantalla la información generada por la WSN.
10. Que la información proveniente de la WSN se guarde dentro de un archivo.
11. Que una imagen sirva como guía que facilite la conexión de los módulos de sensores (*pinmap*).
12. Que la plataforma permita realizar actualizaciones de HW/SW o corregir errores, es decir, que sea flexible para realizar cambios.
13. Que la plataforma sea escalable, es decir, que los recursos de HW y SW faciliten el desarrollo de nuevas características.
14. Que el desarrollo de HW y SW de la plataforma quede documentado para poder reutilizarlo o replicarlo en otros proyectos.

Una vez conocidos los requisitos de HW y SW, ya se puede especificar la plataforma de manera general. Esto se aborda en la Subsección 2.1.3.

2.1.3 Selección de herramientas de HW, SW e Interfaces.

Como tercera fase, hay que investigar que interfaces y herramientas tanto de HW como de SW, pudieran ser necesarias para lograr cumplir todos los requisitos establecidos en la Subsección 2.1.1 lo que nos llevará a cumplir el objetivo general. En la presente Subsección se irán analizando a detalle cada uno de los requerimientos para ir conociendo las diferentes opciones que nos ayudarán a decidir cuáles serán las interfaces y herramientas tanto de HW como de SW que se utilizarán durante el trabajo de tesis.

Por lo anterior, basándonos en los requisitos definidos en la Subsección 2.1.1, se desea una plataforma de HW/SW que permita la distribución ágil de Motes e implementación de una WSN de topología de estrella extendida, donde la WSN deberá tener un funcionamiento intuitivo y los Motes deberán sensar variables meteorológicas como presión barométrica, temperatura interna y externa, humedad y radiación solar. Para obtener versatilidad, que haya dos roles diferentes para los Motes dentro de la WSN. También se puedan desactivar módulos de sensores deseados por medio de una GUI, ya que con ello se podrá personalizar su FW, reducir el tamaño total en Bytes del FW, tema esencial para un sistema embebido debido a las restricciones en los recursos de HW como la memoria, además al no activarse los módulos seleccionados habrá un ahorro de energía.

Uno de los requisitos primordiales que deberá cumplir el Mote es seguir el estándar IEEE 802.15.4 que trata de redes de bajo consumo de energía y baja tasa de transferencia de datos.

Además, debido a que una de las ventajas de las WSN a la intemperie es ser útiles en lugares en los que se le dificulta el acceso a los seres humanos como en un bosque en donde se requiere monitoreo para la prevención de incendios forestales, es por ello que, para reducir costos, cada Mote deberá tener una distancia de separación de aproximadamente 1km.

Los sensores que se le conecten a la plataforma deberán ser uno analógico y tres digitales, pero cada uno de los digitales con diferente protocolo de comunicación serial (se le referirá como procoms) como I2C (*Inter-Integrated Circuits*, protocolo para la comunicación entre IC que se encuentran dentro de un chip), SPI (*Serial Peripheral Interface*, Interfaz Periférica Serial) o *1-Wire*.

Los sensores analógicos deberán contar con tres opciones de configuración de la cantidad de lecturas por muestreo de la señal. Las posibles opciones serán alta, media o baja. Las diferentes opciones deberán ser seleccionadas mediante una GUI. Los sensores deberán ir conectados a pins específicos del Mote.

El Mote *Router* deberá recibir la información conglomerada proveniente de los Motes *end-device*, generar su propia información y enviarla hacia un dispositivo principal (del inglés,

Node Sink) que irá conectado a una PC para poder almacenar la información dentro de un archivo para que pueda ser administrada para fines de análisis. Que el dispositivo principal cumpla con la tarea de unir a los diferentes dispositivos conectados a la WSN, que sea el único dispositivo conectado a una PC y que reciba toda la información conglomerada proveniente del Mote *Router*.
Los dispositivos conectados a la WSN deben poder identificarse, mediante un nombre corto pero descriptivo, ya que esto facilitará su incorporación a la red de una manera simple y sencilla.
Para realizar pruebas y verificación de los diferentes dispositivos, se idearán pruebas individuales de los Motes según su rol y una vez que se compruebe su correcto funcionamiento, se implementará una pequeña WSN con topología de estrella que constará del dispositivo principal, un Mote *Router* y cuatro Motes *end-devices*. Todo deberá quedar documentado para facilitar la usabilidad de la plataforma.
Todo los anteriormente indicado se puede ver de una manera visual en las Figuras 2.2 y 2.3, donde se muestra la forma general de la plataforma de HW/SW.

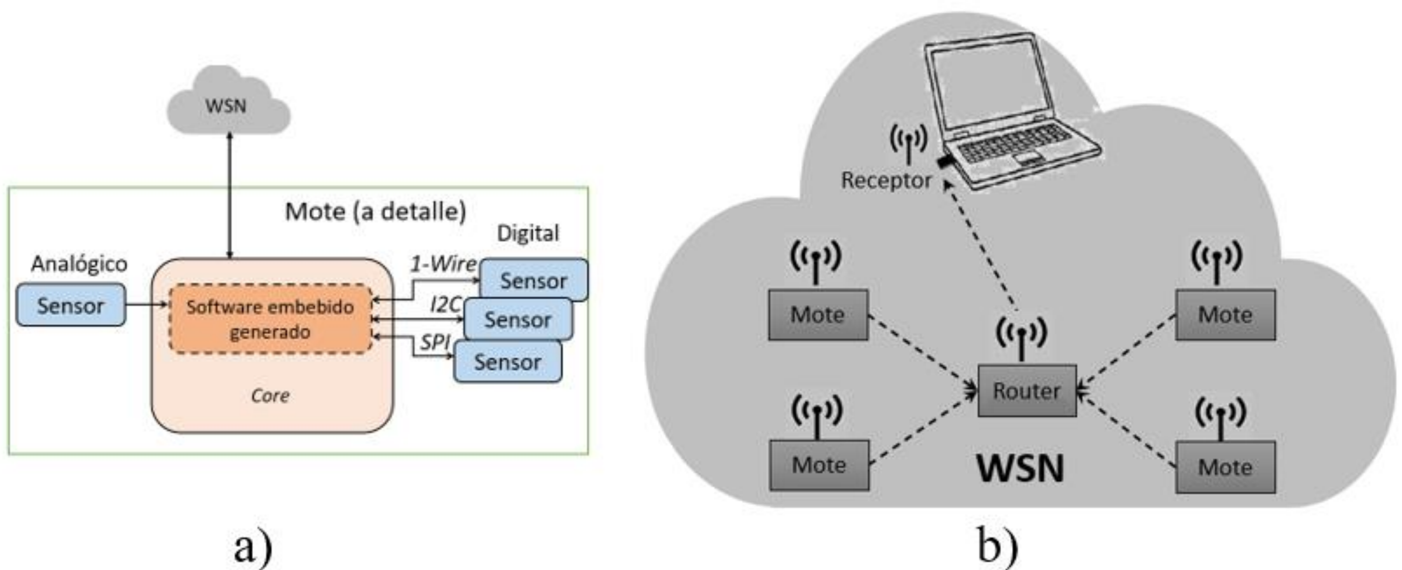


Figura 2.2 Forma general de la plataforma de HW: a) Mote para la WSN. b) WSN con topología de estrella.

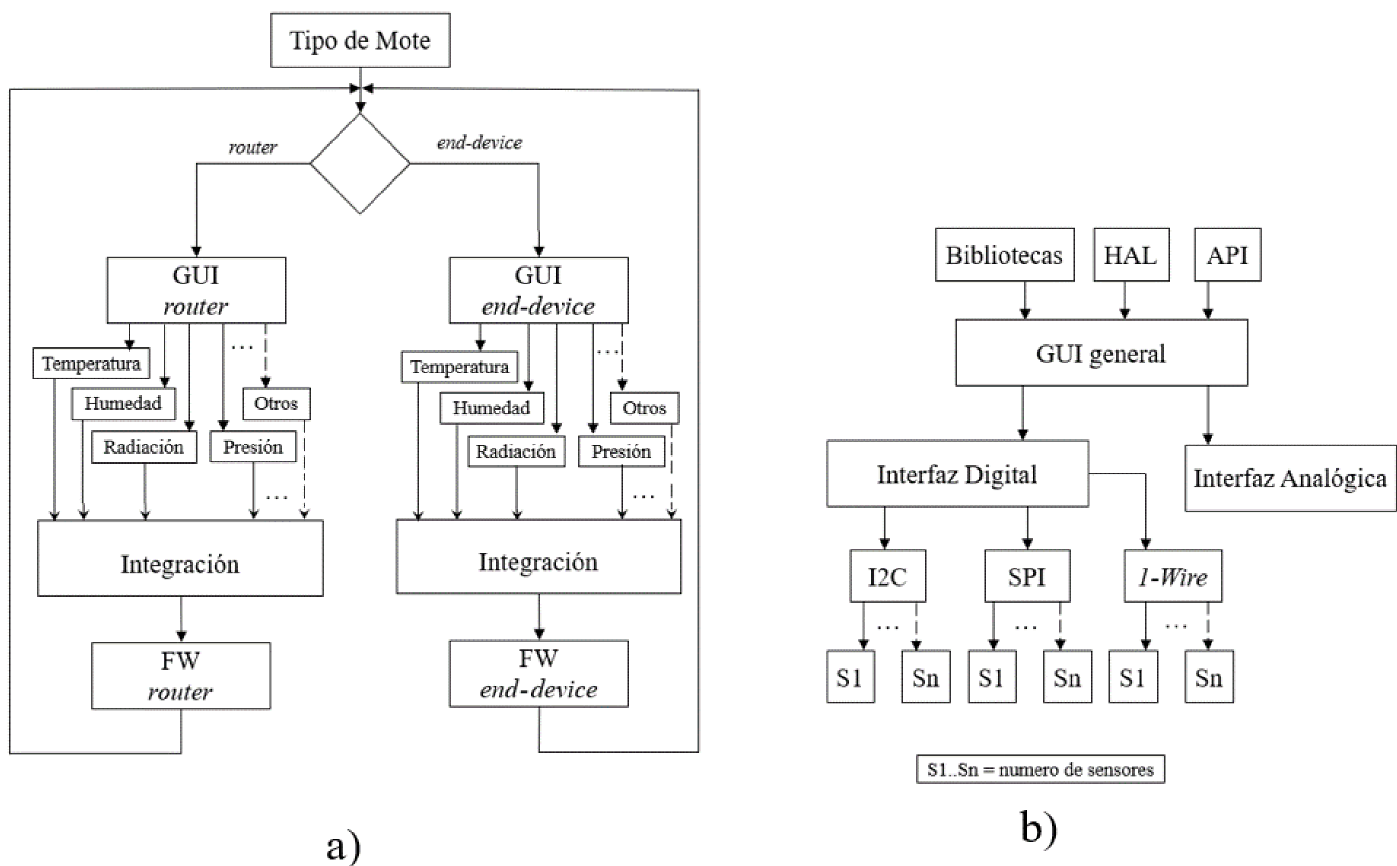


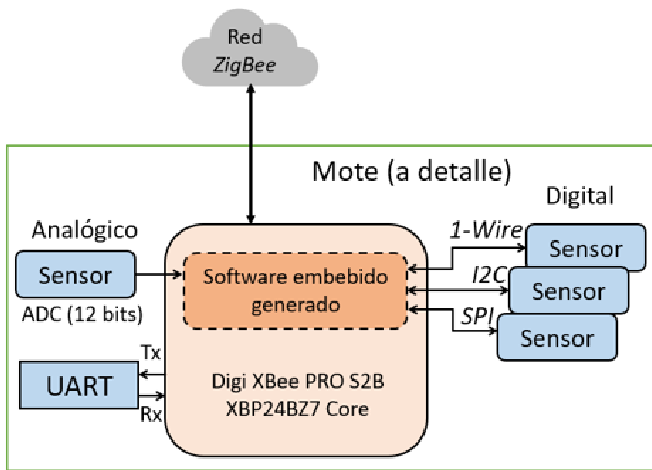
Figura 2.3 Forma general de la plataforma de SW: a) Generación del FW según el rol de los Motes. b) Generación de componentes virtuales de HW.

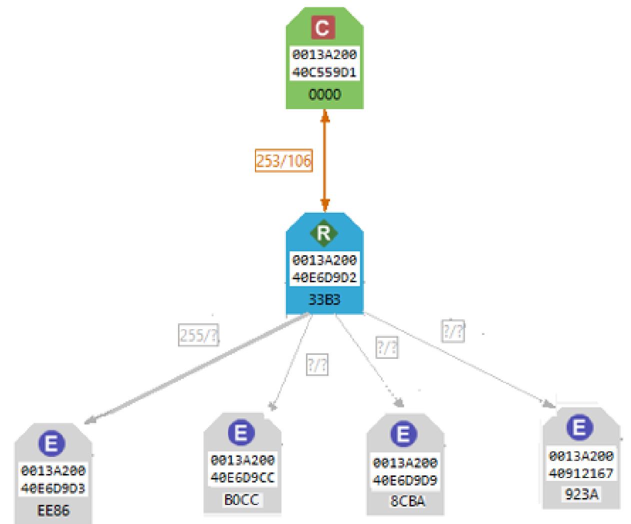
Lo siguiente que hay que realizar es uno de los pasos más importantes y consiste en dividir en módulos de HW, SW o interfaces las diferentes especificaciones de la plataforma. Por lo general, una persona con experiencia en el diseño de HW y SW es quién realiza la división. En el presente trabajo de tesis, dicha división se ha ido realizando conforme avanza el proyecto por lo que en trabajos previos hubo propuestas como los procomi ZB y BLE ya que son los adecuados para cumplir con el estándar IEEE 802.15.4 (aunque cabe aclarar que BLE no se considera dentro del estándar debido a la tasa de transferencia pero sus versiones mayores a la 5.1, cumplen con los requisitos de poder crear redes de malla y bajo consumo de energía) de los cuales se seleccionó a ZB debido a que permite una distancia entre Mote de 1 km y mayor número de Motes conectados a una red; con respecto al Mote, hubo opciones de *Cores* o combinación de ellos como el caso del Arduino UNO R3 cuya tarea sería de coprocesador y un módulo XBee adicional no programable para la etapa de comunicación inalámbrica que en conjunto era una opción de bajo costo pero, debido a su tamaño y alto consumo de energía, no fue viable por lo que se optó por el *Core* XBee; con respecto al bloque de sensores, para la medición de la radiación solar se tenían opciones como PAR Lite y *Texas Instruments* (TI) OPT301 pero se descartaron debido a su alto costo como se menciona en [35] por lo que se seleccionó el TI OPT101 hay que utilizar un filtro de densidad neutral para que funcione como los aparatos profesionales; para la presión barométrica había tres opciones del mismo fabricante como el BME280, el BMP085 y el BMP180 donde se optó por seleccionar este último por ser la opción más accesible en cuanto a precio; para la medición de humedad las opciones más viables eran el DHT22 de Aeosong y el SHT71 de sensirion, donde se seleccionó el primero de ellos debido a su bajo costo.

Por otro lado, en trabajos previos se diseñó la PCB para el Mote por lo que ya no hubo que realizar comparaciones.
Para la etapa de comunicación inalámbrica, se eligió el XStick de Digi por cumplir con el estándar IEEE 802.15.4 y por mantener cierta homogeneidad con el *Core* XBee.
Con respecto al SW, al haber seleccionado el *Core* XBee, el propio fabricante proporciona las herramientas necesarias para el FW del MCU Radio (ver en la Subsección 2.4.1) y para la realización de los requisitos para el FW del MCU programable, evitando la adquisición de licencias y teniendo la ventaja de que son herramientas del propio fabricante que conoce muy bien su HW. Las herramientas a la que nos referimos son el IDE CWDS v10.2 al que se le tiene que instalar el XBee SDK para aprovechar de esta última, otras herramientas útiles como el analizador XML que permitirá la creación de componentes y la selección de configuraciones esenciales por medio de la definición de una GUI.
La mayor aportación del presente trabajo de tesis esta por el lado del SW y el seguimiento de una metodología para obtener una mayor agilidad de implementación y menores iteraciones posibles en la corrección de fallas, por lo que con respecto a los requerimientos solicitados para el presente trabajo de tesis, el trabajo fue agregar nuevas funcionalidades y corregir los inconvenientes detectados en el Capítulo de introducción por lo que en esta etapa había que restructurar el código del FW para el MCU programable e investigar las funciones que nos permitan cumplir con los requisitos solicitados (ver más adelante el Capítulo 3, para mayor detalle).

2.1.4 Selección específica de HW, SW e Interfaces.

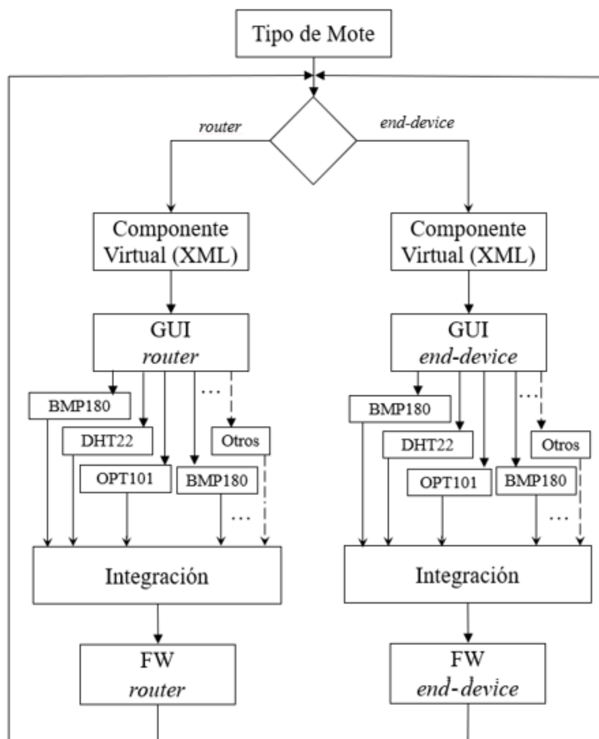
Basado en la Subsección anterior y continuando con la cuarta etapa de la metodología *Co-design* HW/SW que trata sobre la selección específica del HW y SW, como ya se mencionó, en el presente trabajo de tesis se utilizaron las herramientas seleccionadas en trabajos previos como en la parte del HW donde se seleccionó al *Core* XBee, Mote prototipo, la tarjeta de desarrollo XBIB-U-DEV de Digi (se le referirá como U-DEV), P&E USB Multilink (se le referirá como programador HW), XStick, y en la parte del SW, el IDE CWDS v10.2, XBee SDK, *Smart Editor* y funciones de módulos como I2C y en la parte de interfaces, el procomi seleccionado fue ZB, ya que esto ayudará a cumplir con los requisitos establecidos en nuestra primera etapa de la metodología. Debido a lo anterior, las Figuras 2.2 y 2.3 pueden ser actualizadas a las necesidades específicas del presente trabajo de tesis. Esto se puede ver en las Figuras 2.4 y 2.5.

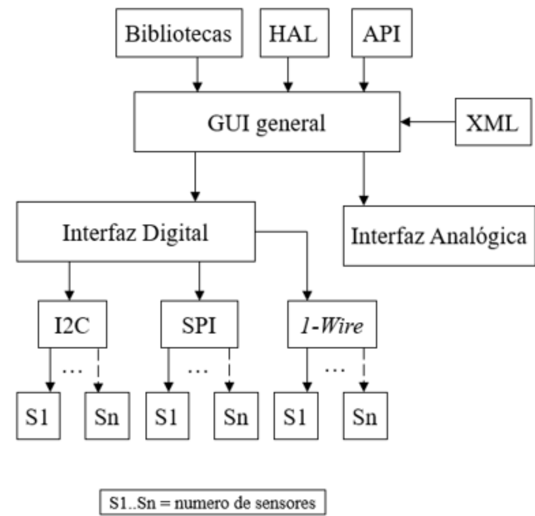




a) b)

Figura 2.4 Plataforma de HW propuesta: a) Mote para la WSN. b) WSN con topología de estrella.





a) b)

Figura 2.5 Plataforma de SW propuesta: a) Generación del FW según el rol de los Motes. b) Generación de componentes virtuales de HW.

Como ya se vio, en la parte del SW se aprovechan las herramientas proporcionadas por el fabricante del *Core* XBee, para construir un FW eficiente para los roles de *Router* y *end-device* escrito en lenguaje C, y por el lado del HW, se utilizaron herramientas como el programador HW para cargar el ejecutable al *Core* XBee o realizar depuraciones, la tarjeta de desarrollo U-DEV para conectar al *Core* XBee a la PC y poder cargarle el FW según el rol del Mote.
Una de las herramientas de SW que ayudó a construir el FW según el rol que tendrá el Mote dentro de la WSN ZB es el IDE CWDS v10.2 y para reutilizar las funciones que proporciona el fabricante del *Core* XBee hay que instalar el *plug-in* XBee SDK, el cual permite una mayor interacción con el *Core* XBee. Un aspecto importante del XBee SDK es que contiene todas las herramientas básicas de un SDK y adicionalmente, cuenta con un analizador de XML que permite agregar componentes virtuales de HW como los módulos de sensores, esto es posible con tan solo crear un archivo XML. Lo anterior tiene la ventaja de que el programador de aplicaciones no necesite un conocimiento total de las herramientas de HW y SW necesarias para poder personalizar un FW según sus necesidades.
Por el lado de las comunicaciones inalámbricas, la pila del procomi ZB se obtiene transfiriendo un FW al MCU Radio del *Core* XBee, según el rol que tendrá el Mote dentro de la WSN ZB. Esto se logra gracias a la herramienta de SW llamada XCTU, proporcionada por el fabricante del *Core* XBee.

2.1.4.1 Documentación de las herramientas de HW.

En la presente subsección, se hablará con mayor detalle sobre las características más comunes de las herramientas de HW que ayudaron a alcanzar el objetivo del presente trabajo de tesis. Estas son:

- XBee PRO S2B: Es el *Core* del Mote de la plataforma propuesta. El modelo es el XBP24BZ7. Es un SoC que consta de dos MCU, el primero de ellos es el MCU Ember EM250 (se le referirá como MCU Radio) que contiene internamente un procesador XAP2b con bus de datos de 16 bits, 5 KBytes de memoria estática de acceso aleatorio (SRAM, *Static Random Access Memory*) y 128 KBytes de memoria *flash,* el MCU Radio llevará el control de las comunicaciones inalámbricas, ya que contiene la pila del procomi ZB que cumple con el estándar ya mencionado; el segundo, es el MCU NXP/Freescale MC9S08QE32 (se le referirá como MCU programable) contiene internamente un procesador HCS08 con bus de datos de 8 bits, 2 KBytes de RAM y 32 KBytes de *flash* que es la que almacenará el FW del MCU programable. El MCU programable se desempeñará como coprocesador ya que le quitará carga de procesamiento al destino de la información. El FW para el MCU programable hará legible las lecturas realizadas por los diferentes sensores conectados a las diferentes interfaces disponibles en el *Core* XBee. La comunicación interna entre MCUs es a través del bus para el protocolo de Transmisión y Recepción Asíncrona Universal (UART, *Universal Asynchronous Receiver-Transmitter*).

  Uno de los motivos que hace posible la comunicación inalámbrica entre Motes a una distancia de separación entre ellos de 1500 m en condiciones ideales, es el conector hembra tipo sma que trae el *Core* XBee y al que hay que conectarle una antena tipo RP-SMA (*Reverse Polarity–Subminiature version A*, Polaridad Inversa-Subminiatura versión A) para lograr dicha distancia de separación. Adicionalmente, el *Core* XBee cuenta con otras dos opciones de antena: una embebida de tipo PCB que en el estado del arte no se recomienda debido a la proximidad entre los componentes que agrega ruido a la señal y una de tipo *wire whip* (que en español se podría traducir como alambre de látigo, muy similar a la antena que traen en el transporte público como los taxis) aunque habría que soldarla al *Core* XBee.
- XBIB-U-DEV: Es una tarjeta de desarrollo de la empresa Digi en la que se conectará el *Core* XBee para, posteriormente, poder transferir los FW para los MCU del *Core* XBee.

Para transferirle el FW al MCU programable, es necesario una herramienta de HW adicional conocida como programador HW (ver en seguida) que se conecta a esta tarjeta a través de un conector JTAG. Para transferir el FW del MCU Radio no es necesario un HW adicional, pero si es necesario tener instalado el programa XCTU provisto por el fabricante. La U-DEV se puede emplear también para ejecutar proyectos de ejemplo que se encuentran dentro del XBee SDK y de esta manera poder comprender el funcionamiento de algunas de las funciones que provee el fabricante.

- P&E USB multilink: Es el HW adicional que necesita la U-DEV para poder transferir el FW al MCU programable. Se debe conectar con la U-DEV a través de un conector JTAG. Contiene como indicadores dos Diodos Emisores de Luz (LED, *Light-Emitting Diode*), uno de color azul y otro amarillo. El LED azul enciende para indicar que las conexiones físicas entre la U-DEV y la PC, están bien. El LED amarillo enciende para indicar que las conexiones físicas entre la U-DEV y el programador HW, están correctas.
- Mote: Es la placa prototipo en la que se conectará el *Core* XBee y los sensores que podrá ser montado en la intemperie. El *Core* XBee se puede programar para que actúe como *Router* o como *end-device*. El Mote debe de ser alimentado a 4.5 VDC por lo que se pueden utilizar tres baterías AA conectadas en serie. Para comprender el funcionamiento del Mote de una manera rápida y sencilla, solo hay que mencionar que contiene un LED de color rojo y otro verde. El LED rojo deberá encender cada que se energice el Mote. El LED verde, deberá encender cuando el Mote tenga información que enviar inalámbricamente y se deberá apagar cuando el *Core* XBee entre en modo de ahorro de energía, el cual es un parámetro modificable para aumentar o disminuir el tiempo que permanecerá el Mote en modo de ahorro de energía. En el caso del Mote *Router*, el LED verde deberá permanecer encendido ya que en una WSN ZB tanto el coordinador como el *Router* siempre deberán estar encendidos, en espera de llegada de paquetes de datos provenientes de los *end-devices*.
  El Mote *Router* deberá poder recibir la información conglomerada proveniente de los Motes *end-device*, generar su propia información y enviarla hacia el coordinador. Por lo anterior, es que es necesario que todos los dispositivos conectados a una misma WSN ZB puedan identificarse mutuamente, mediante un nombre corto pero descriptivo, ya que esto facilitará su incorporación y detección dentro de la red.
- Sensores: al Mote se le podrán conectar tres sensores digitales y un analógico con las siguientes características:
  - ✓ Digitales: el BMP180 de Bosch que sensa presión barométrica y temperatura interna, y para la comunicación entre el sensor y el Mote se utiliza como procoms I2C; el DHT22 de Aeosong que sensa humedad y temperatura externa, con procoms *1-Wire*. Adicionalmente, se podrá conectar un sensor con procoms SPI aunque habrá que crear la biblioteca correspondiente ya que no estuvo dentro del alcance del presente trabajo pero un programador no experimentado deberá poder añadirlo sin necesidad de invertir en tiempo de investigación.
  - ✓ Analógico: El OPT101 de *Texas Instruments* sensa intensidad luminosa, pero se puede adaptar para sensar radiación solar, que era uno de los requisitos.

  Con el fin de utilizar las funciones provistas por el fabricante del *Core* XBee, los sensores hay que conectarlos a pins específicos del mismo. Para la correcta conectividad de los sensores al *Core* XBee, ver más adelante la relación entre números de pin del *Core* XBee y los sensores (a dicha relación se le denominó *pinmap*) correspondiente a los pins mencionados.
- XStick: Fabricado por Digi, es el coordinador de una WSN ZB. Físicamente es parecido a una memoria USB. Para iniciar su funcionamiento, solo se conecta a un puerto USB

disponible de una PC y encenderá un LED de color amarillo que parpadeará continuamente. A este dispositivo no se le puede modificar o transferir SW por lo que sus únicas funciones serán unir a todos los dispositivos que pertenezcan a una misma WSN ZB, establecer de manera automática un parámetro muy importante para la comunicación inalámbrica como es el Canal de Operación (*Operating Channel*) el cual deberá ser el mismo para todos los dispositivos pertenecientes a una misma WSN ZB y por último, recibirá la información conglomerada proveniente del Mote *Router*. Por "misma WSN ZB" nos referimos a que todos los dispositivos deberán tener un mismo Número de Identificador de Red (PAN ID, *Personal Area Network Identifier*). Por otro lado, para ver y guardar la información recibida, hay que abrir una consola de terminal que se encuentra dentro del IDE CWDS v10.2.

2.1.4.2 Documentación de las herramientas de SW.

El fabricante del *Core* XBee proporciona las herramientas de SW que permiten construir un FW eficiente para los roles de *Router* y *end-device* del MCU programable y el FW con la pila del procomi ZB para el MCU Radio.

En la presente Subsección, se dará mayor detalle sobre las características más comunes de las herramientas de SW. Estas son:

- XCTU: Las comunicaciones inalámbricas del presente trabajo son posibles gracias a la pila del procomi ZB que se encuentra en un FW para el MCU Radio. El FW será diferente según el rol que tendrá el Mote dentro de la WSN ZB. Debido a que el coordinador es un dispositivo ya establecido y no se le puede modificar su FW ni cargar un programa, los únicos roles que se podrán transferir hacia el MCU Radio son el de *Router* o *end-device*. Para poder transferir el FW hacia el MCU Radio, es necesario el XCTU y las conexiones físicas necesarias para que esto sea posible. Cabe señalar que los FW para el MCU Radio cuentan con dos modos para la trasmisión de paquetes de datos a través del procomi ZB. Estos modos son AT y API, que son exclusivos del *Core* XBee. En el modo AT, se envían los datos de manera secuencial. Es un método poco seguro debido a que no se asegura la llegada de los datos a su destino, pero es rápido. En este modo se ejecutan los comandos AT que se utilizaban en los antiguos módems. Por otro lado, en el modo API, la información se segmenta en pequeños pedazos, es un método más seguro ya que confirma la llegada de la información a su destino, aunque es lento debido a que se tienen que analizar los paquetes para asegurar el envío y recepción. En nuestro trabajo de tesis se empleó el modo API en todos los dispositivos de la red sin importar su rol, esto con el fin de asegurar la llegada de los paquetes y reducir la perdida de los mismos debido a colisiones. Ejemplo: si se desea que un *Core* XBee tenga el rol de *Router*, hay que transferirle a su MCU Radio el FW ZigBee *Router* API. Un proceso similar se realiza para el rol de *end-device*.
- IDE CWDS v10.2: Esta herramienta de SW permite crear las aplicaciones para el MCU programable del *Core* XBee. Contiene un ensamblador, compilador de C, C++ y un compilador/intérprete de Java (las versiones actuales, contienen un intérprete de MicroPhyton). También se pueden realizar depuraciones de código. En nuestro trabajo de tesis, el FW para el MCU programable fue programado en lenguaje C.

  Por otro lado, al igual que al MCU Radio, al MCU programable habrá que transferirle un FW según el rol del Mote en la red.
- XBee SDK: para reutilizar las funciones que proporciona el fabricante del *Core* XBee hay que instalar el *plug-in* XBee SDK, el cual permite una mayor interacción con el *Core* XBee ya que cuenta con las herramientas que debe de tener todo SDK como APIs, documentación, proyectos de ejemplo, entre otros. Es importante mencionar que todo SDK es específico de la plataforma que se está utilizando. Por otro lado, los proyectos

de ejemplo ayudan a comprender las funciones del fabricante, lo que facilita y agiliza el desarrollo de las aplicaciones. Un aspecto importante del XBee SDK es que cuenta con un analizador de XML llamado *Smart Editor* que permite agregar componentes de HW virtuales.

- XBee *Project Smart Editor*: permite agregar componentes virtuales de HW como los módulos de sensores y con ellos se define una GUI que permitirá agregar los componentes de una manera gráfica además de poder configurar sus parámetros más comunes. Esto es posible con solo crear un archivo XML siguiendo la sintaxis y semántica necesarias para definir las GUIs. Lo anterior tiene la ventaja de que el programador de aplicaciones no necesite un conocimiento total de las herramientas de HW y SW necesarias. Consultar el Anexo I para aprender a crear este tipo de componentes.

  Al poder seleccionar los módulos de sensores por medio de una GUI (que se visualiza dentro del *Smart Editor*) se facilita el poder agregarlos de manera fácil e intuitiva y al mismo tiempo se estaría agregando código en C dentro del FW existente del MCU programable para de esta manera, poder personalizarlo.

  Con lo que respecta al sensor analógico, se le podrá configurar la cantidad de lecturas por muestreo de la señal mediante tres posibles parámetros: alta, media o baja. La opción alta es de 300 lecturas, la media será de 150 lecturas y la baja de 75 lecturas, a mayor cantidad de lecturas, mayor precisión de las lecturas obtenidos por el sensor. Lo anterior es posible gracias al módulo ADC de 12 bits que se encuentra dentro del MCU programable.

  Los archivos XML no solo pueden definir GUIs sino también proyectos de ejemplo para 54724el IDE, que agiliza la programación de los Motes ya que el proyecto se puede configurar *ad-hoc* para poder crear y transferir ejecutables hacia el MCU programable. Los archivos XML para los proyectos son analizados por el IDE no el *Smart Editor*. Consultar el Anexo I para aprender a crear proyectos de ejemplo.

2.1.4.3 Planteamiento de pruebas para la evaluación de la plataforma.

Con el fin de verificar si se cumple con todos los requisitos planteados, es necesario realizar pruebas unitarias o en grupo de los componentes tanto de HW como de SW. Las pruebas unitarias se deben realizar para asegurar que cada uno de los bloques por separado, que componen al sistema, funcionan correctamente y las pruebas de grupo aseguran la correcta integración entre los bloques de HW con los de SW y de esta manera poder comprobar la funcionalidad de la plataforma conforme a los requisitos planteados y así evitar posibles incompatibilidades en etapas más avanzadas del proyecto. Es por lo anterior, que primero se tienen que plantear y detallar las pruebas unitarias a realizar para poder verificar parte del sistema. Las pruebas se detallan en la TABLA 2.1.

TABLA 2.1: Pruebas unitarias para los diferentes módulos de HW y SW de la plataforma.

# prueba	Hipótesis	Herramientas	Desarrollo
1	Al MCU radio se le puede transferir el FW seleccionado.	HW: Una U-DEV, un *Core* XBee, un cable USB tipo A/B (para impresora). SW: XCTU de Digi.	Realizar las conexiones necesarias, ejecutar el XCTU, agregar al *Core* XBee dentro del área de módulos de radio (*Radio Modules*) y transferir hacia el *Core* XBee el FW de "ZigBee *Router* API". Al transferirse el FW seleccionado hacia el MCU radio sin obtener un error, se comprueba el funcionamiento de cada herramienta de HW y SW que se utilice.

2	Los modos AT y API del MCU Radio son sencillos de comprender	HW: Dos U-DEV, dos *Core* XBee, dos cables USB tipo A/B. SW: XCTU.	Realizar las conexiones, ejecutar la aplicación XCTU, agregar los *Core* XBee dentro del área de módulos de radio y transferirles el FW de "ZigBee *Router* API"; dentro del XCTU, abrir un Modo de Trabajo con Consolas (*Consoles Working Mode*) para ambos y enviar mensajes de texto sencillos. Repetir los pasos anteriores, pero ahora transfiriendo el FW "ZigBee *Router* AT". Al enviar mensajes de texto simples, mediante el uso de cualquiera de los dos modos, se podrá entender la diferencia entre los modos AT y API.
3	Es sencillo el uso del IDE para realizar pruebas de envío de paquetes de datos.	HW: Un programador HW, dos U-DEV, dos *Core* XBee, tres cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK de Digi instalado.	Realizar las conexiones necesarias, abrir el XCTU y transferir a los MCU Radio el FW de ZigBee *Router* API. Ejecutar el IDE CWDS v10.2, agregar al explorador de proyectos, el proyecto de ejemplo llamado "*simple_chat_ni_ZB*", crear el ejecutable y transferirlo hacia los dos *Core* XBee; al terminar de transferir, dentro del IDE, abrir dos conexiones de consola de terminal para cada uno de los *Core* XBee y comenzar a enviar mensajes de texto plano. Al realizar un intercambio de mensajes entre dos módulos, se comprueba el funcionamiento del Radio.
4	Se carga correctamente el FW del MCU programable.	HW: Un programador, una U-DEV, un *Core* XBee, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones necesarias, abrir el XCTU y transferir al MCU Radio el FW de ZigBee *End Device* API. Cargar en el *Core* XBee el proyecto de tesis que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2 llamado "*Mote_EndDv_v2*" (si no se encuentra, solicitar el proyecto para cargarlo dentro del IDE), agregar o quitar módulos de sensores deseados, crear el ejecutable y transferirlo hacia el *Core* XBee. Al cargar el ejecutable sin obtener errores, se comprobará que el MCU programable funciona correctamente.
5	Aumenta el número de Bytes de memoria *flash* del MCU programable, al agregar módulos de sensores.	HW: Un programador HW, una U-DEV, un *Core* XBee, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones, abrir el XCTU y transferir al MCU Radio el FW de ZigBee *End Device* API. Cargar en el *Core* XBee el proyecto de tesis que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2 llamado "*Mote_EndDv_v2*", agregar o quitar módulos de sensores deseados, crear el ejecutable y transferirlo hacia el *Core* XBee. Al terminar de transferir el FW personalizado, dentro del archivo con extensión *.map*, dentro de la sección *startup section*, se podrá ver al final la cantidad en Bytes que ocupa la aplicación.

			Al realizar esta prueba se comprobará como la personalización del FW afecta en el número total de Bytes que se transfieren.
6	Los sensores miden con exactitud y precisión.	HW: Un programador HW, una U-DEV, dos *Core* XBee, dos cables USB tipo A/B, un XStick, sensores de DHT22, BMP180 y OPT101, un medidor comercial EXTECH SD700. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones, abrir el XCTU y transferir a uno de los MCU Radios el FW de ZigBee *End Device* API y al otro, el FW de ZigBee *Router* API. Transferirle al Mote *end-device*, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2, configurarle un nombre corto, crear el ejecutable; para el Mote *Router* los pasos son similares con la diferencia de que se le carga el proyecto "*Mote_Router_v2*" y no se le asigna nombre descriptivo. Montar una WSN ZB de un coordinador, un Mote *Router* y un Mote *end-device*. Dentro del CWDS v10.2, abrir una ventana de terminal para el XStick. Comprobar la información recopilada por el XStick contra las lecturas del EXTECH SD700. Al conocer los datos generados por los sensores se comprueba que tan precisas son las lecturas de los sensores y que tan exactas son al compararlas con las lecturas obtenidas por un equipo comercial.
7	Los Motes *end-device* se incorporan fácilmente a la WSN ZB.	HW: Un programador HW, una U-DEV, tres Motes, un XStick, sensores DHT22, BMP180 y OPT101, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones, abrir el XCTU y transferir a dos de los MCU Radios el FW de ZigBee *End Device* API y al tercero, el FW de ZigBee *Router* API. Cargar en los Motes *end-device* y conectarles sus sensores, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2; proceder de manera similar con el Mote *Router* cargándole el proyecto "*Mote_Router_v2*". Montar una WSN ZB donde se conecten los Motes *end-device*, luego el Mote *Router* y por último, el coordinador. Dentro del CWDS v10.2, abrir una ventana de consola de terminal para el XStick y observar la llegada de la información. Si después de salir del rango de cobertura y volver a entrar, se continua recibiendo la información aunque un pequeño retraso, se podrá observar si los dispositivos pierden su conexión.
8	El Mote *Router* recibe la información proveniente de los Motes *end-device*.	HW: Un programador HW, una U-DEV, tres Motes, un XStick, sensores DHT22, BMP180 y OPT101 para cada Mote, dos cables USB tipo A/B.	Realizar las conexiones, abrir el XCTU y transferir a dos de los MCU Radios, el FW de ZigBee *End Device* API y al tercero, el FW de ZigBee *Router* API. Cargar en los Motes *end-device*, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE

		SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	CWDS v10.2 y conectarles sus sensores; proceder de manera similar pero ahora con el Mote *Router* cargándole el proyecto "*Mote_Router_v2*". Montar una WSN ZB de un coordinador, un Mote *Router* y dos Motes *end-device*, ejecutar el XCTU, agregar los *Core* XBee al área de módulos de radio, abrir el modo de trabajo en red (*Network Working Mode*) y escanear la red para ver gráficamente como los Motes *end-devices* se conectan al Mote *Router*. Si se recibe un paquete de datos por cada end-device conectado a la red por lo que al recibirse cada paquete, se comprueba la comunicación entre los dispositivos involucrados.
9	El Mote *Router* está sensando.	HW: Un programador HW, una U-DEV, tres Motes, un XStick, sensores DHT22, BMP180 y OPT101, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones, abrir el XCTU y transferir a dos de los MCU Radios el FW de ZigBee *End Device* API y al tercero, el FW de ZigBee *Router* API. Cargar en los Motes *end-device*, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2; proceder de manera similar pero ahora con el Mote *Router* cargándole el proyecto "*Mote_Router_v2*". Conectar los sensores al Mote *Router*. Montar una WSN ZB de un coordinador, un Mote *Router* y dos Motes *end-device*, abrir una ventana de consola de terminal para el XStick dentro del CWDS v10.2. Comprobar que el coordinador está recibiendo los datos.
10	La WSN ZB con topología de estrella funciona correctamente.	HW: Un programador HW, una U-DEV, cinco Motes, sensores DHT22, BMP180 y OPT101 para cada Mote, un XStick, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	Realizar las conexiones, abrir el XCTU y transferirles a cuatro de los MCU Radios el FW de ZigBee *End Device* API y al quinto, el FW de ZigBee *Router* API. Cargar en los Motes *end-device*, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2; proceder de manera similar pero ahora con el Mote *Router* cargándole el proyecto "*Mote_Router_v2*". Montar una WSN ZB de un coordinador, un Mote *Router* y cuatro Motes *end-device*, ejecutar el XCTU, agregar al XStick al área de módulos de radio y abrir una ventana de modo de red trabajando para el XStick. Existen las versiones 3 y 4 de ambos roles para los Motes, realizarlas para comparar. Si después de un par de horas no existe algún error de comunicación, se comprobará que la red es funcional.
11	Se puede guardar dentro de un archivo,	HW: un programador HW, una U-DEV, cuatro Motes,	Realizar las conexiones, abrir el XCTU y transferir a cuatro de los MCU Radios el FW

	la información recibida por el coordinador.	sensores DHT22, BMP180 y OPT101 para cada de Mote, un XStick, dos cables USB tipo A/B. SW: XCTU, IDE CWDS v10.2 con el XBee SDK instalado.	de ZigBee *End Device* API y al tercero el FW de ZigBee *Router* API. Cargar en los Motes *end-device*, el proyecto de tesis llamado "*Mote_EndDv_v2*" que se encuentra dentro del área de explorador de proyectos del IDE CWDS v10.2; proceder de manera similar pero ahora con el Mote *Router* cargándole el proyecto "*Mote_Router_v2*". Montar una WSN ZB de un coordinador, un Mote *Router* y cuatro Motes *end-device*, crear un archivo de texto plano en una trayectoria conocida, guardar la información generada por la WSN ZB, abriendo una consola de terminal dentro del CWDS v10.2 para el XStick, se solicitará el archivo de texto plano previamente creado. Al terminar, cerrar la captura de información y abrir el archivo para comprobar que se capturó la información.
12	Es preferible crear un proyecto de ejemplo para el IDE, en lugar de exportarlo e importarlo.	HW: Una PC. SW: IDE CWDS v10.2 con el XBee SDK instalado.	Crear un proyecto de ejemplo para el IDE CWDS v10.2 y XBee SDK, una vez creado el proyecto de ejemplo, agregarlo al explorador de proyectos y crear el ejecutable correspondiente (ver Anexo I). Con el IDE abierto, exportar el proyecto recién creado y al terminar, borrarlo del explorador de proyectos. Por último, importar el proyecto recién exportado, crear el ejecutable correspondiente y comparar el proceso de crear un proyecto de ejemplo y de exportarlo e importarlo. Al personalizar el FW y cargarlo sin problemas, se podrá decidir que es más conveniente para agilizar el proceso de programación de los Motes cuando son varios.

A lo largo del presente Capítulo, se detallaron las primeras fases de la metodología *Co-design* HW/SW para lograr un mejor prototipado, estableciendo los requisitos funcionales y no funcionales de nuestra plataforma, así como la selección específica de las herramientas tanto de HW como de SW que nos ayudaron a cumplir con los requisitos planteados. Por último, se establecieron algunas pruebas básicas que ayudaron a verificar el buen funcionamiento e integración de los módulos de HW y SW para que estos cumplan con los requisitos de la plataforma propuesta. En el siguiente Capítulo se continuará con las fases de la metodología *Co-design* HW/SW explicando cómo se fue desarrollando nuestra plataforma.

Capítulo 3

Diseño de la plataforma de Hardware y Software para el desarrollo ágil de redes de sensores inalámbricas

En el Capítulo 2, se detallan las fases de requerimientos del sistema, fraccionamiento del HW y SW, selección y especificación de las interfaces, herramientas de HW y SW de la metodología *Co-design* HW/SW empleada en el presente proyecto. Continuando con el desarrollo del proyecto de la plataforma propuesta, en este Capítulo se continua con las fases de implementación y configuración del HW, especificación del FW, implementación del SW embebido además de la fase de pruebas del HW y compilación del SW, terminando las fases anteriores se tendrá una primera iteración del desarrollo del presente proyecto. A continuación, se detallan dichas fases.

3.1 Implementación del HW: configuración del HW.

Como se comentó en la Subsección 1.1.1, el presente trabajo de tesis se basa en trabajos previos de donde se tomó al Mote prototipo como herramienta. Como quinta fase de la metodología propuesta, en la presente Subsección se describirá la manera como deberá proceder el desarrollador para reparar manualmente los inconvenientes encontrados y realizar las configuraciones de HW necesarias para poder utilizar las herramientas de HW y SW disponibles, sin que se presenten errores de funcionamiento.

3.1.1 Configuración de herramientas de HW.

En la presente Subsección se establecerán pasos a seguir para tener un Mote *Router* o *end-device* disponible para que pueda unirse a una WSN ZB. Como ya se había mencionado, para poder programar y preparar Motes que se unirán a una WSN ZB es necesario HW adicional como el programador de HW, una tarjeta U-DEV y una PCB prototipo por cada Mote de la red. Lo que hay que hacer para realizar pruebas de funcionamiento de las herramientas de HW necesarias (algunas de manera individual otras de manera colectiva) es:

1. Cargar aplicación para el MCU programable del *Core* Xbee del Mote.
   El HW necesario para programar el MCU programable del *Core* XBee es una PC con puertos USB libres, el programador de HW, tarjeta U-DEV y el *Core* XBee. Para ello:
   a. Realizar las conexiones de HW.

Insertar el *Core* XBee en los receptáculos correspondientes de la U-DEV; después, conectar el programador de HW con la U-DEV y por último, conectar la U-DEV y el programador de HW a la PC. Esto se puede observar en la Figura 3.1.

Figura 3.1 Conexión del programador con la U-DEV para programar el *Core* XBee.

b. Crear la aplicación.
c. Construir el ejecutable y transferirlo.
d. Repetir pasos anteriores por cada Mote deseado.

2. Preparar físicamente al Mote.

   El HW necesario para preparar al Mote es un *Core* XBee previamente programado, una PCB prototipo, alimentación para la PCB y los sensores. Uno de los requisitos de la plataforma, es el de la Figura 3.2 donde se tiene un *pinmap* que se utiliza como guía para la conexión de los sensores y en la Figura 3.3 se puede observar cómo quedará preparado el Mote para poder unirlo a una WSN ZB.

```
XPIN1  -|                         |- XPIN20
XPIN2  -|                         |- XPIN19
XPIN3  -|                         |- XPIN18
XPIN4  -| ======================= |- XPIN17
XPIN5  -| #   # ####  #### #### |- XPIN16
XPIN6  -|  # #  #    # #    #     |- XPIN15
XPIN7  -|   #   ####  ###  ###    |- XPIN14
XPIN8  -|  # #  #    # #    #     |- XPIN13
XPIN9  -| #   # ####  #### #### |- XPIN12
XPIN10 -| ======================= |- XPIN11
        |_________________________|

********* PINS LEGEND ***********

*  XPIN1 = VCC                          * XPIN20 = adcBateria [ADC]
*  XPIN2 = UART0 [TX]                   * XPIN19 = I2C [SCL]
*  XPIN3 = UART0 [RX]                   * XPIN18 = SPI [SPSCK]
*  XPIN4 = SPI [MISO]                   * XPIN17 = OPT101
*  XPIN5 = special0 [RST]               * XPIN16 = pinDHT22
*  XPIN6 = special0 [RSSI PWM]          * XPIN15 = special [Assoc]
*  XPIN7 = I2C0 [SDA]                   * XPIN14 = VCC REF
*  XPIN8 = GPIO Pin                     * XPIN13 = pwr_mgnt0
                                                   [On Sleep Pin]
*  XPIN9 = pwr mgnt [Sleep Req]         * XPIN12 = GPIO Pin
*  XPIN10 = GND                         * XPIN11 = SPI [MOSI]
```

Figura 3.2 *pinmap* para la conexión de sensores al Mote.

Figura 3.3 Conexión de los sensores para preparar al Mote.

3. Montar una WSN ZB.
   El HW necesario para montar una WSN ZB son un XStick, un Mote *Router* y cuatro Motes *end-device*. Para ello:
   a. Realizar las conexiones necesarias.
   b. Montar la WSN ZB. En la Figura 3.4 se puede ver el ejemplo de montaje de una WSN ZB.

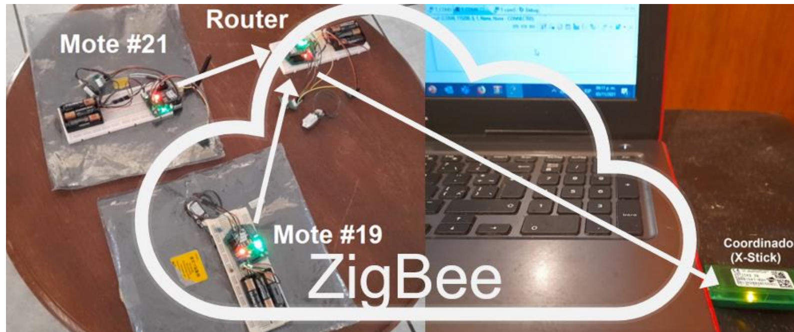


Figura 3.4 Ejemplo de montaje de una pequeña WSN ZB.

Con los pasos uno y dos, queda configurado el HW para poder ser utilizando durante las diferentes etapas de la especificación del SW que es la Subsección 3.1.3. El paso tres será el resultado final de nuestro trabajo.

3.1.2 Correcciones a la PCB del Mote empleado en la tesis de licenciatura.

Con el fin de comprobar el buen funcionamiento de las herramientas de HW a utilizar, se implementó una pequeña WSN ZB de un coordinador, un Mote *Router* y dos Motes *end-devices.* Como resultado de lo anterior, se detectó que la PCB del Mote presentaba algunas fallas como apagado repentino, se batallaba mucho para que los Motes se unieran a la WSN ZB, no se alcanzaban los voltajes necesarios para la alimentación del *Core* XBee lo que provocaba que no hubiera comunicación. Debido a lo anterior, hubo la necesidad de realizar un análisis a nivel electrónico de la PCB para encontrar la causa que provocaba dichas inconsistencias.

Como resultado del análisis realizado al PCB del Mote prototipo, se detectaron cambios en los valores de algunos de los componentes electrónicos como resistencias y capacitores, además, en una de las PCBs, el regulador de voltaje que alimentaba al *Core* XBee no estaba trabajando de manera correcta.

En la TABLA 3.1, se enlistan los problemas que se encontraron en la tarjeta del Mote que se usó en proyectos anteriores a este. Además, se incluyen las soluciones propuestas.

TABLA 3.1: Problemas de HW presentados durante el desarrollo.

Problema	Pasos a realizar	Solución
	1. Se revisaron los esquemáticos del PCB.	Se cambió el regulador de voltaje que alimentaba al *Core* XBee ya que presentaba daño.
Motes no se pueden unir a la WSN ZB	2. Se revisaron los voltajes en puntos específicos para asegurar que el voltaje de trabajo fuera el correcto.	
	3. Pruebas de continuidad en la PCB.	
Apagado repentino de los Motes.	1. Se revisaron los esquemáticos del PCB.	1. Basándonos en el análisis de los esquemáticos, se cambiaron resistencias y capacitores que presentaban valores incorrectos.
	2. Se revisaron los voltajes en puntos específicos.	2. Se reforzaron puntos de soldadura que presentaban desgaste.
	3. Pruebas de continuidad en la PCB.	3. Se aseguró el voltaje mínimo para los reguladores de voltaje que contiene la PCB, utilizando baterías, no fuentes de alimentación.
		4. Se volvió a montar la WSN ZB de un coordinador, un Mote *Router* y dos Motes *end-device*.

Una vez que se comprobó que la PCB del prototipo funcionaba correctamente, se desarrollaron tres PBCs adicionales a las disponibles con el propósito de aumentar la disponibilidad de Motes para la implementación de una WSN ZB.

Las correcciones realizadas y el pequeño aumento en el número de Motes disponibles, permitió analizar el rendimiento y escalabilidad de la WSN ZB planteada dentro de los requisitos funcionales.

3.2 Implementación del SW: especificación del FW

Mediante las herramientas de SW proporcionadas por el fabricante del *Core* XBee se pudo construir un FW eficiente y confiable para el MCU programable, usando como entorno de programación, el IDE CDWS v10.2, el *plug-in* XBee SDK y derivado de este último, el *Smart Editor*. Hay que recordar que el XBee SDK cuenta con herramientas útiles para la interacción con el *Core* XBee como funciones, APIs, el *Smart Editor* y otras más, que permitirán crear componentes virtuales de HW asociados a los sensores requeridos y de esta manera, definir una GUI que permita agilizar el desarrollo de la WSN ZB propuesta, sin la necesidad de que el usuario tenga un conocimiento total de todas estas herramientas de HW y SW.
También se creó una API para el procoms I2C y las bibliotecas necesarias que permiten dar modularidad al sistema. Todo el diseño y desarrollo del FW, API y algunas bibliotecas fueron escritas en lenguaje C.
La pila del procomi seleccionado se obtiene transfiriendo un FW al MCU Radio del *Core* XBee utilizando como herramienta, la aplicación XCTU de *Digi*.
Cabe resaltar que, en el desarrollo del SW embebido, no solo se trabajó de manera abstracta con la virtualización de componentes, también se trabajó a bajo nivel, al reutilizar las funciones proporcionadas por el fabricante (se les referirá como funciones nativas) como las que se encuentran dentro de la biblioteca `i2c_xbee` para el procoms I2C y de esta manera conformar la API para I2C que simplifica el uso de las funciones nativas del módulo I2C del *Core* XBee. Se determinó que, para aprovechar otras funciones nativas, era necesario conectar los sensores en los pines que por defecto son asignados a cierto tipo de interfaz. Además, se reutilizaron las funciones nativas del procomi ZigBee del *Core* XBee como `xbee_transparent_rx` que es útil en la recepción de paquetes de datos provenientes de nodos remotos, `node_dicovery_callback` y `xbee_disc_discover_nodes` útiles en la detección automática de nodos remotos por medio de un nombre descriptivo, lo cual elimina la necesidad de configuraciones complejas basadas en las direcciones MAC de cada dispositivo. Estas últimas funciones nativas permitieron la creación de la biblioteca `m_red_zb` para una mejor estructura y modularidad. Todo lo anterior

era necesario para la reutilización de las diferentes funciones nativas ya que con ellas se puede controlar la plataforma de HW con solo cambiar algunos parámetros que garantizan la correcta operación de la tarea que será ejecutada.

Por otro lado, la metodología de desarrollo que se ha venido realizando tiene la ventaja de permitir que se habiliten o deshabiliten sensores a través de la GUI. Cuando se habilitan los sensores, se crean macros que se guardan dentro de un archivo llamado `xbee_config.h`, reduciendo el uso de recursos y el consumo de energía general del sistema a lo estrictamente necesario. Es decir, el SW embebido resultante que se genera de forma automatizada no contendrá los bloques de código para la interacción con sensores que no se usarán.

Para obtener una idea general del sistema, en la Figura 3.5 se muestra un diagrama de flujo nuestra plataforma.

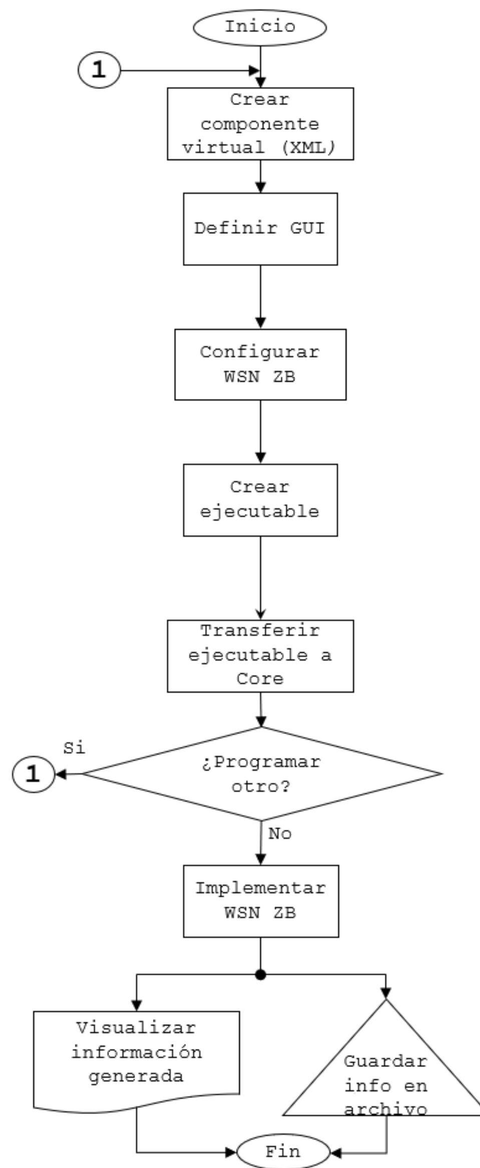


Figura 3.5 Diagrama de flujo de la plataforma de HW/SW.

A continuación, se describen las bibliotecas mencionadas con anterioridad que le brindan el soporte necesario a nuestra plataforma de HW así como sus bloques funcionales que ayudaron a robustecer el FW del MCU programable, útiles en la gestión de los Motes.

a. `i2c_xbee`

El MCU programable de *Core* XBee trae un módulo para el procoms I2C que ayuda mayormente en la comunicación interna con un MCU y se basa en un modelo maestro-esclavo. El I2C del MCU programable siempre será maestro, por lo tanto, es recomendable que los dispositivos con los que se comunique sean esclavos. Para conectar varios dispositivos I2C a un mismo bus, hay que configurarles una dirección con el propósito de que el maestro identifique con quien se comunica. Por ejemplo, el sensor BMP180 cuenta con dos dirección es que corresponden al modo en que se configura; para enviarle información la dirección es `0x77` y dentro del código del FW, se guarda dentro de la macro `I2C_DIR_ESCLAVO`. Por lo anterior, el fabricante del *Core* XBee provee las funciones necesarias para la gestión del módulo I2C del MCU programable, aunque su documentación es poco clara. Es por ello que surgió la necesidad de crear una API que ayuda a abstraer la manera de trabajar con las funciones nativas del módulo I2C. La API se encuentra dentro de la biblioteca `i2c_xbee` y a continuación se explica cómo trabaja cada función.

- `i2c_readByte(`*regRead*`)` : El *Core* XBee lee un byte proveniente del esclavo. Esta función recibe como parámetro a *regRead* que es un número hexadecimal (se le referirá como HEX) de tipo entero sin signo con formato 0x, que representa la dirección del registro del esclavo que el maestro desea leer. Como resultado de ejecución, dicha función regresa el contenido del registro del dispositivo I2C, que al igual que *regRead* es un byte de tipo entero sin signo.
- `i2c_readWord(`*regRead*`)` : El *Core* XBee lee dos bytes provenientes del esclavo. Funciona de manera similar a la función `i2c_readByte`, solo que regresa dos bytes (que equivale a un Word).
- `i2c_writeBytes(`*regWrte*, *dato*`)` : El *Core* XBee envía dos bytes hacia su esclavo. Ambos son de tipo entero sin signo. El parámetro *regWrte* indica el número HEX de la dirección de escritura del esclavo con ella se le indica al esclavo que el maestro quiere enviarle información y con el parámetro *dato*, se configura al esclavo para que realice cierta tarea (por ejemplo, medir temperatura). Como resultado de la ejecución, no se regresa ningún dato.

b. `m_red_zb`

Dentro de esta biblioteca se encontrarán aquellas funciones relacionadas con el procomi ZB que se reutilizaron dentro del proyecto y que son útiles para que un nodo local detecte nodos remotos con tan solo configurarle un Identificador de Nodo (NI, *Node Identifier*) o para que reciba paquetes de datos provenientes de todos los nodos remotos conectados a la misma red. A continuación, se describen las funciones que ayudaron a agregarle usabilidad a la configuración de los Motes.

- `xbee_transparent_rx(`*envelope, context*`)` : Función de evento que se activa dentro del *Smart Editor*, seleccionado el componente “Comunicación Zigbee” (Zigbee *communication*) y activando el parámetro “Proceso de Paquetes de Entrada” (*Process Incoming Frames*). El parámetro *envelope* es de tipo estructura donde se recibe un paquete de datos de los dispositivos remotos pertenecientes a una misma WSN ZB. El parámetro *context* no se utiliza y no afecta en el resultado de la ejecución. Como resultado de ejecución, regresa un 0 para indicar que se tuvo una recepción exitosa de la información. Dentro del proyecto, la función se utiliza específicamente para que el Mote *Router* reenvíe hacia el coordinador de la red la información recibida desde los Motes *end-devices*.

- node_discovery_callback(*xbee*, *node_id*) : Función de evento que se activa dentro del *Smart Editor*, seleccionando el componente "Comunicación Zigbee" (Zigbee *communication*) y activando el parámetro "Soporte para el Descubrimiento de Nodos" (*Node Discovery Support*). Trabaja de manera conjunta con la función xbee_disc_discover_nodes ya que esta última detecta nodos conectados a la WSN ZB y la función node_discovery_callback, selecciona un nodo específico de todos los paquetes de datos que recibe la función xbee_disc_discover_nodes. El parámetro *xbee* es de tipo estructura y guarda información sobre el nodo local; el parámetro *node_id* también es de tipo estructura y guarda la información de un nodo remoto especificado. Todo esto se realiza de manera automática al estar el nodo local dentro de la red. Como resultado de la ejecución, regresa información sobre el nodo remoto.
- xbee_disc_discover_nodes(*xbee, identifier*) : Esta función detecta nodos en la WSN ZB. Trabaja en conjunto con la función node_discovery_callback ya que esta última selecciona un nodo específico de todos los paquetes de datos que recibe esta función. El parámetro *xbee* es de tipo estructura y guarda información sobre el nodo local; el parámetro *identifier* es de tipo caracter y contendrá el NI del nodo remoto para identificarlo de una manera más simple y sencilla y evitar utilizar la dirección física del nodo (también conocida como MAC *Address*). Esta función podría aceptar solo aquellos paquetes provenientes del nodo especificado dentro de *indentifier* o aceptar paquetes de cualquier nodo de la WSN ZB, siempre y cuando, *identifier* se configure como *NULL* (haciendo una analogía con las redes de datos, *NULL* es como hacer un *broadcast* o lo que es lo mismo, enviar paquetes a toda la red). Como resultado de la ejecución, regresa 0 si se detecta un paquete proveniente de los nodos remotos o un código de error, si el parámetro que recibió no es válido u otro código de error, si el *buffer* de transmisión serial del MCU Radio no tiene cupo de almacenamiento.

c. r_red_zb

Muy similar a la biblioteca m_red_zb vista con anterioridad, solo difieren en la función xbee_transparent_rx que se agrega en el FW para el Mote *Router* debido a que es quién recibe la información proveniente de los Motes *end-device* y posteriormente la reenvía hacia el coordinador de la red.

d. m_mote_energia

Dentro de esta biblioteca se encuentran dos funciones creadas en trabajos previos como calcularNivelBatería y sleepMode. No se describe su funcionamiento lo único que se menciona es que no estaban separadas en una biblioteca y se agregaron a esta biblioteca ya que serán útiles en otros proyectos que involucran al *Core* XBee.

e. r_mote_energia

Muy similar a la biblioteca m_mote_energia anterior, pero se diferencian en que esta biblioteca solo contiene la función calcularNivelBatería ya que no es recomendable que el Mote *Router* se apague (es decir, no hay que configurándole un modo de ahorro de energía) debido a que siempre debe de estar a la escucha de los paquetes enviados por los Motes *end-device* dentro de una WSN ZB.

f. commonLibrary

Esta librería fue creada en trabajos previos por lo que únicamente se le añadió la función sensor(*tipoSnsr*, *argum1*, *argum2*) creada con el fin de seleccionar el tipo de sensor con el que se trabajará el Mote. Los parámetros *argum1* y *argum2* dependerán del sensor que se desee utiliza por lo que deberán ser los más utilizados para que el sensor funciones correctamente; lo anterior se hizo con el fin de darle dinamismo a la manera de seleccionar a los diferentes sensores. En el caso de crear una biblioteca para un sensor diferente a los propuestos dentro del proyecto de tesis, solo hay que realizar algunas modificaciones sencillas para poder agregarlo al catálogo de sensores.

3.3 Compilación del SW embebido.

La sexta fase de la metodología está compuesta por dos etapas llamadas: compilación de SW y pruebas de HW. En la presente Subsección se abordará el tema de la compilación del SW.
Desde hace algunos años, las compañías almacenan datos que desean compartir con sus proveedores o socios de negocio. Uno de los beneficios de utilizar XML es facilitar el intercambio de información ya que ofrece una gran simplicidad al momento de compartir datos o acceder a ellos, dándoles un formato más amigable. Al hecho de poder crear una relación entre un documento XML con un lenguaje de programación ayudará a que el programador de aplicaciones ofrezca una mayor calidad en sus implementaciones de código,
El CWDS v10.2 junto con el XBee SDK mantienen esa relación entre XML y el lenguaje C, a través del *Smart Editor* lo que facilita la creación de aplicaciones personalizadas para el *Core* XBee además de acelerar el proceso de implementación. En la presente subsección se describirá como crear componentes virtuales de HW con un simple archivo XML que será parte de la definición de una GUI que permitirá agregar código en C a las aplicaciones para el *Core* XBee, además de realizar configuraciones comunes de los diferentes módulos de HW de la plataforma con la que se está trabajando y agregar algunas funciones de evento al código principal.

3.3.1 Desarrollo de los componentes virtuales de HW.

Como ya se mencionó en la Subsección 1.2.2.4, las entidades más importantes en todo documento XML son los elementos y los *tags* los cuales hay que aprender a utilizarlos para crear los componentes virtuales de HW ya comentados. Se propone el desarrollo de componentes virtuales de HW, que permitan generar SW para llenar espacios en el flujo de trabajo, usando como base las herramientas que proporciona el fabricante del *Core* XBee.
Al comprender la sintaxis de los *tags*, se obtendrá la posibilidad de crear nuevos componentes virtuales de una manera sencilla que permitirá reutilizar las bibliotecas desarrolladas.
En la Figura 3.6, se presenta un diagrama de flujo que explica de manera gráfica los pasos a seguir en la creación de componentes virtuales de HW.

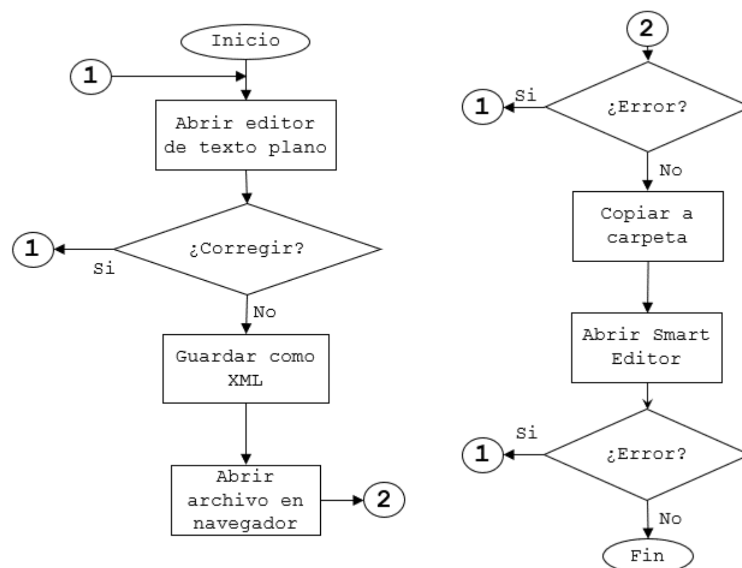


Figura 3.6 Diagrama de flujo para la creación de componentes virtuales.

Como se puede observar en el diagrama de flujo de la Figura 3.6, el primero paso a seguir en la creación de componentes virtuales de HW, es abrir un archivo de texto plano y comenzar a

teclear la sintaxis y semántica de las etiquetas y *tags* propias del *Core* XBee, definidas en el Anexo I. Al finalizar de teclear el código XML, guardar y arrastrar el archivo recién creado hacia un navegador de Internet que pueda leer el tipo de archivo *.xml*. como, por ejemplo, Firefox. Si no se despliega ningún error, es indicativo que el archivo sigue las reglas sintácticas de XML pero no necesariamente significa que vaya a funcionar. En caso de que se despliegue algún error, habrá que iterar hasta corregirlo y al cumplirse todos los requisitos planteados, copiar el archivo dentro de la carpeta de instalación del XBee SDK. Después de realizar la copia, abrir el CWDS v10.2 y luego el proyecto que contendrá el componente creado con anterioridad, abrir el archivo config.xml y si no se despliega ningún error, se habrá creado un componente virtual de HW. Si se despliega un error o hay un comportamiento erróneo, habrá que iterar en la metodología hasta corregir el error.

Como se mencionó, los *tags* son jerárquicos y en la Figura 3.7, se muestra dicha jerarquía que ayudará a comprender su uso para la creación de componentes virtuales como el de la Figura 3.8 que corresponde al sensor de presión atmosférica. Una vez creado el nuevo componente se deberá visualizar dentro del *Smart Editor*. La herramienta que se puede utilizar para teclear el código XML puede ser un editor de texto plano como el *block* de notas de *Windows*, solo hay que guardar el archivo con la extensión .xml.

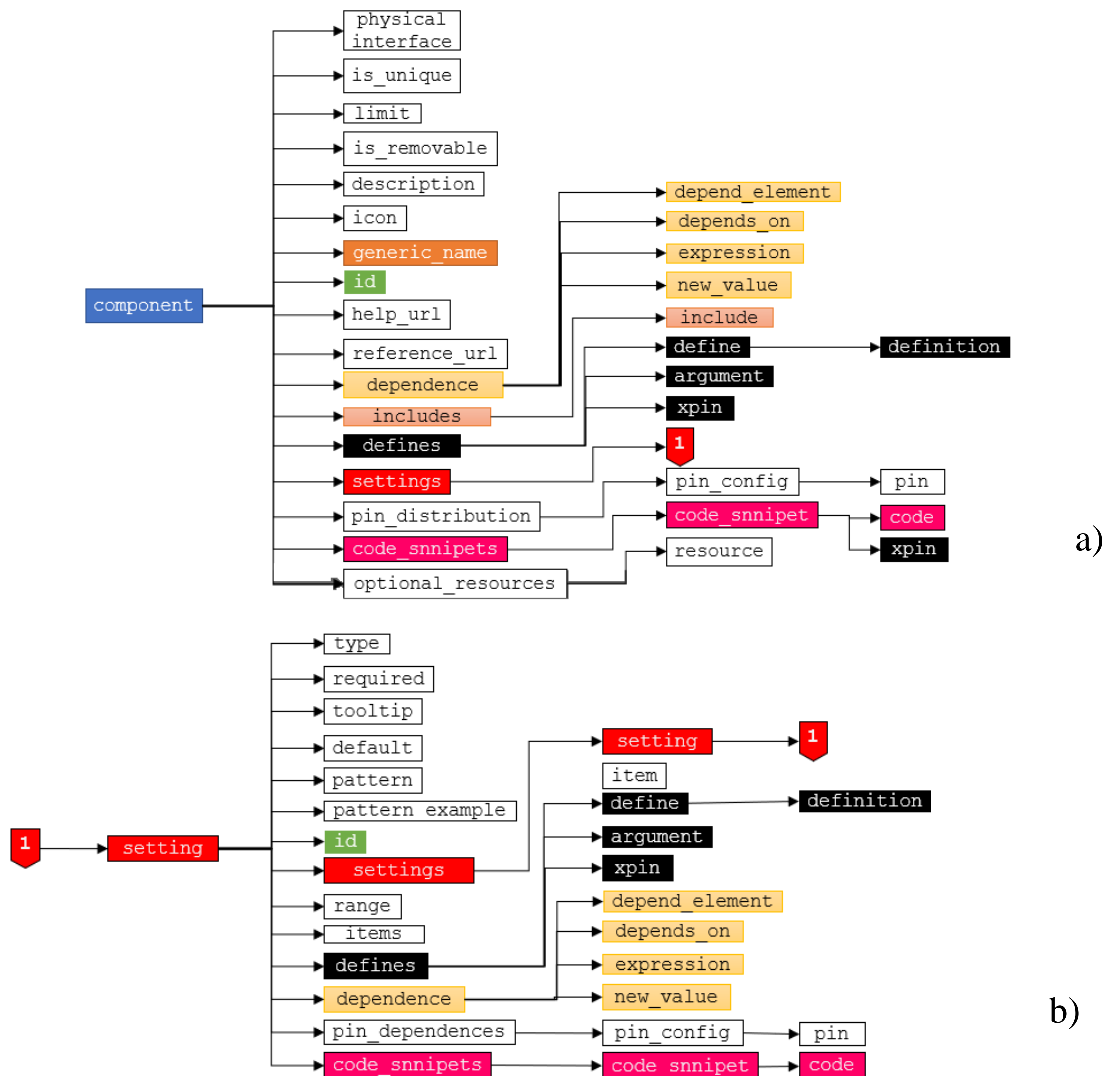


Figura 3.7 Árbol jerárquico de los *tags* para la creación de componentes virtuales de HW: a) Jerarquía principal. b) Continuación de la jerarquía del inciso (a).

El árbol jerárquico de los *tags* para la creación de componentes virtuales de HW, se puede observar en la Figura 3.7a el cuál comienza con el *tag* de mayor jerarquía en color azul, llamada *component*, para continuar con el siguiente nivel de jerarquía donde hay que seguir un orden descendente al momento de utilizarlos, luego se continua con el tercer nivel de jerarquía en donde no todos los *tags* del segundo nivel cuentan con *tags* de tercer nivel. El nivel jerárquico finaliza con un cuarto nivel en donde, de igual manera, no todos los *tags* cuentan con un cuarto nivel jerárquico.

De la Figura 3.7a y b, se puede observar que el *tag settings* (en color rojo) es recursiva, ya que se pueden crear *settings* dentro de otras para conformar grupos de *settings* más grandes.

Los *tags* sombreados en verde y rojo son las que más podrían causar errores al momento de seguir la sintaxis XML por lo que hay que tener cuidado especial con ellos. Como ejemplo de creación de un componente virtual de HW, en la Figura 3.8 se puede observar parte del código del componente para el sensor BMP180 del trabajo de tesis.

```
<subcomponent label="Sensor i2c" visible="true">
  <physical_interface>false</physical_interface>
  <is_unique>false</is_unique>
  <is_removable>true</is_removable>
  <description>
    Agregar un sensor con protocolo de comunicacion serial I2C.
  </description>
  <icon>icons/M_bmp180.png</icon>
  <generic_name>SNSR_I2C</generic_name>
  <id>snsr_iic</id>
  <help_url>html/group__api__24xxx__eeprom.html</help_url>
  <dependence type="existence">
    <depend_element>i2c</depend_element>
    <depends_on>existence</depends_on>
  </dependence>
  <includes>
    <include>#include <BMP180.h> </include>
  </includes>
```

Elemento
tag

Figura 3.8 Parte del código XML para la creación del componente del BMP180. Una visualización de la jerarquía de los *tags* se puede observar en la Figura 3.7.

Del código de la Figura 3.8, se puede observar que el *tag* `subcomponent` no se agregó dentro de la jerarquía de la Figura 3.7, esto es debido a que la estructura es muy similar a la de `component`, con algunos ligeros cambios. El *tag* `subcomponent` se utiliza cuando se desean añadir componentes de igual similitud de comunicación con el *Core* XBee, como por ejemplo, sensores de un mismo procoms como I2C.

Al terminar el documento XML, hay que verificar que se encuentre libre de errores de sintaxis; esto se puede realizar abriendo el archivo XML recién creado dentro de un explorador de Internet como Firefox. Si el navegador despliega el código escrito dentro del archivo, es indicativo de que la sintaxis esta correcta. Si se despliega un error, hay que revisar la sintaxis, por ejemplo, si un elemento no contiene su correspondiente *tag* final.

Una vez que el navegador pueda leer el archivo XML, esto será indicativo de que la sintaxis esta correcta, y el siguiente paso es copiar el archivo dentro de la carpeta `xml_components` que se encuentra dentro de las carpetas de instalación del XBee SDK.

Después, hay que ejecutar el IDE, y dentro del explorador de proyectos, localizar la carpeta que corresponda al proyecto con el que se está trabajando, y buscar el archivo *config.xml* que contiene el código XML para la GUI predeterminada de todo proyecto con el *Core* XBee. Al

abrir dicho archivo, se deberá desplegar el componente virtual recién creado. Si no se visualiza el componente virtual dentro de la GUI o se despliega, pero se comporta de manera poco coherente, es muy probable que exista un error en la jerarquía de *tags* o un error debido al *tag* ID que se visualiza en color verde dentro de la Figura 3.7 con el que se configura el nombre que tendrá el componente virtual dentro del código XML. El error más común es que está duplicado y eso causa conflicto. Habrá que analizar el archivo XML para encontrar el error.

Una vez que se visualice el componente virtual dentro de la GUI y permita realizar configuraciones comunes del mismo, se habrá terminado de crear un componente virtual de HW. Todo lo anterior, se puede observar gráficamente dentro del diagrama de flujo de la Figura 3.6.

Por último, debido a que se documentaron las herramientas de HW y SW de la plataforma propuesta, esto abre la posibilidad de generalizar el proceso para crear nuevos componentes virtuales que representen otros sensores (analógicos o digitales) que robustecerán el catálogo de sensores mencionado en los requerimientos de la Subsección 2.2, lo que permitirá desarrollar de forma ágil y eficiente el SW embebido personalizado para los diferentes roles de los Motes (*Router* y *end-devices*) y de esta manera poder conformar nuevas WSN ZB con topología de estrella, considerando las ventajas de bajo costo, bajo consumo energético, mantenibilidad y escalabilidad de las mismas.

3.3.2 Desarrollo de GUIs para los diferentes roles de la red.

Se propuso definir una GUI que ayude al desarrollador de aplicaciones a seleccionar y configurar los componentes virtuales de HW creados. La GUI está basada en la herramienta *Smart Editor* y se define creando archivos XML que siguen la sintaxis y semántica requeridas como se mencionó en la Subsección 3.3.1 para que después sean interpretados y usados para generar formularios que deben ser llenados por el usuario final.

En la Figura 3.8, se presenta un ejemplo al código XML que corresponde a uno de los componentes virtuales de nuestra plataforma. En la Figura 3.9, se puede observar cómo queda integrado el componente virtual de HW a la GUI lo que permite seleccionarlo entre los componentes virtuales disponibles para que de manera automatizada se identifique e incluyan los fragmentos de código específicos para usar alguno de los sensores. Esto resulta en un SW embebido personalizado que se ejecutará dentro del *Core* XBee permitiendo utilizar únicamente los sensores seleccionados y además tendrá un rol establecido en la WSN ZB, ya sea como *Router* o *end-device*.

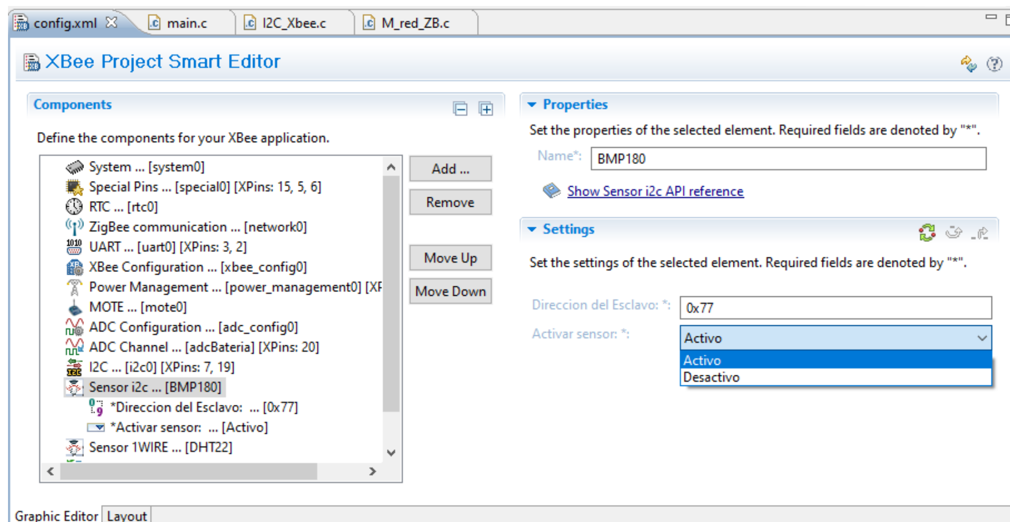


Figura 3.9 GUI con el componente virtual de HW para el sensor BMP180 agregado.

Es importante mencionar que, con el propósito de optimizar la tarea de comunicación inalámbrica tratando de ahorrar energía de las baterías, se requería enviar hacia el coordinador en una sola transmisión, la información generada y conglomerada por los Motes *end-device* y el Mote *Router*, por lo que se crearon tres versiones del código del trabajo previo tanto para el rol de *Router* como el de *end-device*. Lo anterior fue un requerimiento debido a que el coordinador es el único dispositivo de la WSN ZB que está conectado a una PC y en esta se pueden almacenar mayores cantidades de información además de tener conectividad al Internet. Se obtuvieron cuatro versiones diferentes, donde la primera versión fue la creada en trabajos previos; de las tres siguientes, la segunda es la más estable y con mejor escalabilidad, aunque no envía en una sola exhibición toda la información como se requería; las versiones tres y cuatro son muy inestables y poco escalables por lo que se descartaron.

3.4 Descripción de pruebas necesarias de las herramientas disponibles.

Para asegurar el acoplamiento de los módulos de HW con los de SW (séptima fase de la metodología de la Figura 2.1), primero hay que realizar algunas pruebas individuales que nos aseguren el funcionamiento de cada módulo. El orden a seguir para realizar las pruebas individuales se puede ver en el diagrama de flujo de la Figura 3.10. La realización de estas pruebas es necesaria debido a que en la práctica se detectaron *Core* XBee y tarjetas U-DEV que funcionaban a una menor capacidad de la que normalmente deben de trabajar y lo cual provocó retrasos en el desarrollo e implementación del proyecto.

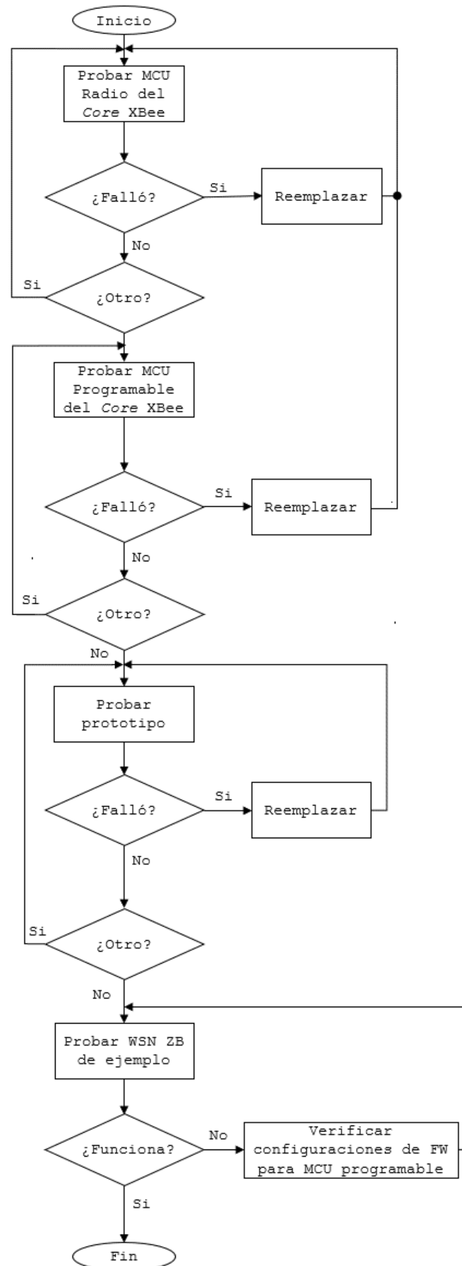


Figura 3.10 Diagrama de flujo para la realización de pruebas individuales de los diferentes componentes de HW de la plataforma propuesta.

3.4.1 Descripción de pruebas de HW.

A continuación, se mencionan algunas pruebas sencillas para las herramientas de HW que se utilizaron en el proyecto.

- MCU Radio. Una prueba relativamente sencilla para la prueba de este módulo es enviar mensajes de texto entre dos *Core* XBee. Para ello, se ocupa como HW dos *Core* XBee con FW ZigBee *Router* API, dos tarjetas U-DEV, una PC y como SW, el programa XCTU. Una vez conectados ambos *Core* XBee a la PC, hay que agregarlos dentro del XCTU al área de Módulos de radio (*Radio modules*) y abrir una conexión por consola serial y, por último, enviar mensajes como paquetes de datos. En la Figura 3.11 se pueden observar los resultados de esta prueba sencilla, donde se encierran dentro de un cuadro en rojo, en el

área Registro de Paquetes de Datos (*Frames log*), los mensajes de texto tanto recibidos como los enviados a través del *Core* XBee local. En el caso de la Figura 3.11, los paquetes enviados por el *Core* XBee *Router* hacia su coordinador (que en este ejemplo, es otro *Core* XBee), están en color azul y los paquetes recibidos, provenientes desde el coordinador, en color rojo.

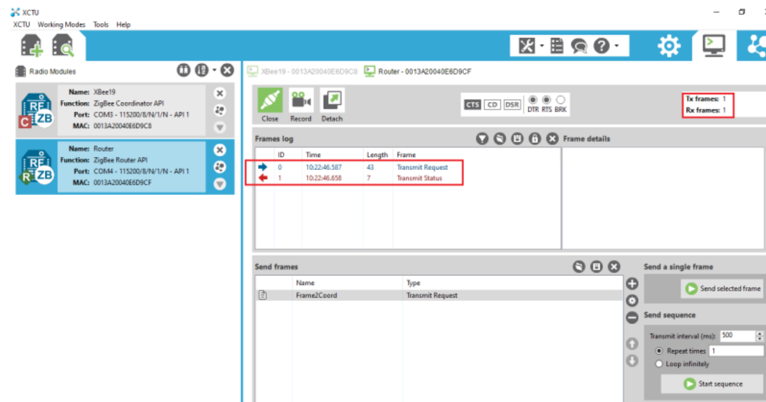


Figura 3.11 Prueba de funcionamiento del MCU Radio utilizando la herramienta XCTU de Digi.

Para mayor detalle, consultar el Anexo I.

- Varias herramientas. Para realizar una prueba de varias de las herramientas de HW al mismo tiempo, hay que transferir el ejecutable de uno de los tantos proyectos de ejemplo para el *Core* XBee que proporciona el fabricante del Core XBee dentro del XBee SDK y que se llama *simple_chat_ni_ZB*. Para este proyecto se ocupan el programador de HW, dos *Core* XBee, dos tarjetas U-DEV. Una vez que se transfiera el proyecto mencionado, hay que abrir dentro del IDE CWDS v10.2, una consola de terminal para cada *Core* XBee y comenzar a enviar mensajes de texto sencillo. El mismo proyecto cuenta con una guía pequeña de su funcionamiento dentro del archivo `ReadMe.txt`. En la Figura 3.12, se puede observar el resultado de ejecución del proyecto *simple_chat_ni_ZB*.

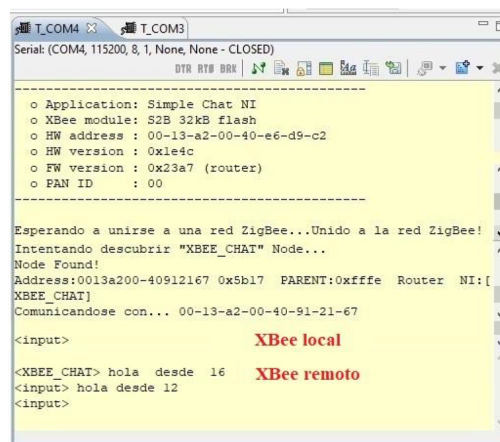


Figura 3.12 Prueba de funcionamiento de varias de las herramientas de HW como el *Core* XBee, el programador, tarjetas U-DEV.

- Todas las herramientas. Al tener listo el ejecutable del proyecto de tesis, se completará las pruebas de todas las herramientas de HW como el prototipo del Mote. Esta prueba solo se podrá realizar al tener dicho ejecutable.

3.4.2 Crear ejecutable del SW.

El proceso de creación de aplicaciones para sistemas embebidos es muy diferente al proceso de creación de aplicaciones para sistemas de propósito general. Una de las diferencias es que en las aplicaciones para sistemas de propósito general como la PC, el proceso de creación del ejecutable y la ejecución de la aplicación, se realiza dentro del mismo sistema; en un sistema embebido como el *Core* XBee, los pasos que se siguen para crear un ejecutable es diseñar y escribir el código de la aplicación, compilarlo y que un enlazador reúna todas las bibliotecas necesarias para crear el ejecutable (en inglés, a esto se le conoce como *build*) pero a diferencia de las aplicaciones de propósito general, las aplicaciones para sistemas embebidos no se ejecutan dentro del mismo sistema (PC) si no que hay que transferirlo al sistema destino (*target*, en inglés) en donde se ejecutará mientras este energizado el dispositivo u ocurra un error de ejecución. El proceso de creación del ejecutable para el MCU programable consta de dos tareas esenciales como son el de compilación y el de enlace (*linker*, en inglés). Es muy probable que el proceso de compilación de la aplicación para el MCU programable sea exitoso si se siguen correctamente las reglas sintácticas del lenguaje que se esté utilizando, pero podría haber problemas con el enlazador ya que una de sus funciones es asignar localidades de memoria de un sistema que no conoce. El IDE CWDS v10.2 cuenta con un Enlazador Inteligente (*Smart Linker*, en inglés) que realiza varias funciones como las mencionadas con anterioridad. Para utilizarlos, lo único que hay que hacer es crear el ejecutable.

Por otra parte, todos los errores relacionados con el enlazador se desplegarán con un código numérico [38] precedido por la letra L.

Los errores más comunes con el enlazador serán debido a que no se incluyeron las bibliotecas necesarias para manejar módulos de HW como el administrador del punto flotante o el manejo de las variables globales o, en el caso del lenguaje C, las variables de tipo puntero.

Un ejemplo de error común que regresa el Enlazador Inteligente es el siguiente:

L1822: symbol ... in file ... is undefined.

que se despliega cuando no se incluye la biblioteca que maneja alguno de los módulos de HW del dispositivo que se está utilizando.

Por lo anterior, para realizar la creación de un ejecutable de manera exitosa, solo hay que tener cuidado de incluir las bibliotecas de manejo de HW que se esté utilizando en el proyecto, conocer el uso de las variables globales o, si se utiliza lenguaje C, las de tipo puntero.

En el presente Capítulo se continuó con las fases de la metodología propuesta para de esta manera completar la primera iteración con respecto al desarrollo de nuestra plataforma. En el siguiente Capítulo se realizarán las pruebas a la plataforma de HW/SW planteadas en la Subsección 2.5 y de esta manera poder validar el cumplimiento de cada uno de los requisitos planteados en la Subsección 2.2 y comprobar que no hubo necesidad de realizar una iteración adicional y de esta manera cerrar el ciclo de la metodología.

Capítulo 4

Pruebas y validación de la plataforma desarrollada

En el Capítulo 3, se realizaron las fases de la metodología *Co-design* HW/SW propuesta correspondientes a la selección e implementación del HW, SW e interfaces sobre el desarrollo de nuestra plataforma con lo que se finalizó el desarrollo del proyecto de tesis. En el presente Capítulo, se realiza la integración de los diferentes módulos de HW y SW que conforman la plataforma, mediante la realización de pruebas de funcionalidad, usabilidad, y mantenibilidad para comprobar el cumplimiento de cada uno de los requerimientos planteados y con ello terminar con todas las fases de la metodología *Co-design* HW/SW de la Figura 2.1. En caso de no cumplirse alguno de los requisitos, habrá que volver a iterar hasta cumplir con cada uno de ellos. A continuación, se detallan las pruebas realizadas que permiten integrar cada Mote con sus respectivos módulos de HW/SW y la conformación de la WSN ZB para probar y validar la plataforma desarrollada.

4.1 Pruebas de funcionalidad de la plataforma.

Una de las ventajas de implementar una WSN es poder hacerlo en ambientes no urbanizados y difícil acceso, como bosques, zonas desérticas, etcétera, donde se cubran áreas extensas con los Motes que conforman a dicha WSN. Cada Mote estará expuesto a cambios climáticos, daños físicos debido a la fauna o personas que puedan vivir en este tipo de ambientes en los que es difícil recibir una señal de celular o de Internet. El procomi más adecuado para este tipo de soluciones es ZB debido a sus características de comunicación como largo alcance y ahorro de energía.

Debido a lo anterior, es importante que los Motes sean pequeños, económicos y ahorradores de energía, esto último es esencial ya que hay que evitar darles mantenimiento correctivo, que se traduce en consumo de recursos humanos y materiales debido a la naturaleza de las WSN en exteriores. En el estado del arte, la mayoría de las plataformas de HW/SW encontradas solo atacan un problema en específico y su arquitectura es cerrada, además de que no indican la metodología de desarrollo empleada.

Por otro lado, la mejor manera de validar la funcionalidad de la plataforma propuesta y comprobar el cumplimiento de los requisitos planteados en la Subsección 2.5, es realizando su distribución en exteriores no sin antes hacerlo dentro de un ambiente controlado como el Laboratorio de Hardware Avanzado de la Facultad de Ingeniería de la UASLP y el entorno que lo rodea. En la siguiente Subsección, se detalla cómo realizar la distribución de una WSN ZB con topología de estrella extendida dentro de este ambiente controlado. La prueba en exteriores en campo libre de interferencias, quedará pendiente como trabajo a futuro.

4.1.1 Conectividad.

Para poder comprobar los resultados esperados la WSN ZB desarrollada estuvo conformada por un Coordinador que consta de un módulo *XStick* de *Digi* conectado a una PC, un Mote *Router* y cuatro Motes *end-device* disponibles. No se realizaron pruebas en ambientes externos, debido a que no se contó con los recursos necesarios para llevarlos a cabo, enfatizando la necesidad de realizar dichas pruebas en ambientes libres de interferencias de

señales inalámbricas como WiFi, *Bluetooth*, antenas transmisoras de celular, estaciones de radio y televisión e incluso factores que degraden el alcance de la señal, como muros de concreto, partes metálicas, entre otros, todo con el fin de analizar el comportamiento de la red. Cabe mencionar que dentro de la Universidad se logró obtener un alcance de hasta 80m, lo cual confirma la capacidad de *Core* XBee dentro de ambientes altamente ruidosos. Más adelante se detalla información al respecto.

Al realizar las pruebas señaladas en la presente Subsección, en la parte del SW, se verifica el aporte que proveen las bibliotecas y API utilizadas dentro del proyecto como son: la API para el módulo I2C del MCU programable que simplifica el uso de las funciones nativas del *Core* XBee o las funciones que facilitan la identificación de los Motes *end-device* dentro de la red de una manera mucho más sencilla, gracias a la configuración de un nombre corto y descriptivo, lo cual elimina la necesidad de configuraciones complejas basadas en la dirección física (MAC *Address*) de cada Mote. También, gracias al uso de la GUI, se comprueba la facilidad con la que los Motes pueden ser programados, sin importar su rol dentro de la red.

Con respecto al ahorro de energía, el hecho de poder seleccionar únicamente los sensores deseados, permitirá activar sólo los módulos de HW/SW del MCU programable que serán necesarios, con lo que se ahorrará energía y memoria *flash*, una parte fundamental en el diseño de este tipo de sistemas ya que es un recurso restringido; aunado a lo anterior, también se aprovechará una característica importante del *Core* XBee, el modo de ahorro de energía, ya que se desactivan algunos módulos de HW por un tiempo programado, lo que ahorra tiempo de vida de las baterías. El ahorro de espacio en memoria y de baterías son algunos de los criterios más importantes dentro del diseño de una WSN debido a la naturaleza restrictiva de recursos de los dispositivos que la conforman. En el caso del Mote desarrollado, entra en modo de bajo consumo de energía por un periodo de tiempo programado, y regresa a la actividad después de pasar dicho tiempo, con el fin de enviar la información generada vía inalámbrica, además de estar a la escucha de paquetes de datos que podrían provenir del Mote *Router* o del Coordinador. La selección de módulos de HW deseados, así como el poder programar el tiempo de permanencia en reposo del Mote, es posible mediante una GUI amigable que facilita las configuraciones de los diferentes módulos de HW.

En las Subsecciones siguientes se mostrarán de manera general, los pasos a seguir para el montaje de la WSN ZB señalada en Subsecciones anteriores.

4.1.2 Componentes virtuales de HW.

Debido a las herramientas proporcionadas por el fabricante del *Core* XBee, se puede generar un *firmware* personalizado para el MCU programable que permitirá llenar los espacios en el flujo de trabajo. Algunas de las entidades que contribuyen en la generación del *firmware* son los componentes virtuales de HW que se crean tecleando código XML propio del *Core* XBee para definir una GUI con la que se podrá añadir bibliotecas y código en lenguaje C además de opciones de configuración de parámetros primordiales de los componentes virtuales que estén siendo creados. Para el presente proyecto, se crearon los componentes virtuales para la configuración de parámetros esenciales de los sensores OPT101, BMP180 y DHT22 además de otros componentes generales como los que permiten la configuración del número de segundos que permanecerán en reposo los Motes (sin importar su rol en la red), así como, el dispositivo con el que se deben conectar. En la Figura 4.1a se puede observar el código XML

para el componente virtual correspondiente al sensor analógico de radiación solar y en la Figura 4.1b, su integración al *Smart Editor*.

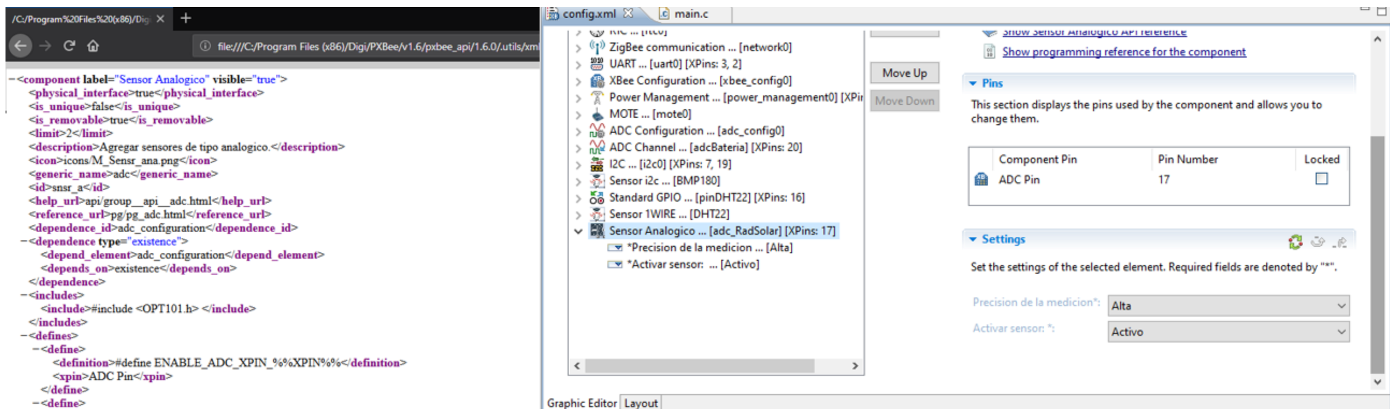


Figura 4.1: a) Código XML de componente virtual OPT101. b) GUI con componente integrado.

4.1.3 Espacio en memoria de los recursos de HW.

Una vez que se crea el ejecutable para el rol que tendrá el Mote en la red, se puede comprobar la cantidad en Bytes que ocupará dentro de la memoria *flash* del MCU programable. De manera predeterminada, en el proyecto se cargan todos los componentes virtuales creados para él y se pueden quitar los deseados, lo que ahorrará espacio en memoria y energía de las baterías ya que no se habilitan los módulos de HW correspondientes.
Para conocer el número de Bytes que ocupa la aplicación de nuestra plataforma, hay que ejecutar el IDE CWDS v10.2 y dentro del Explorador de Proyectos (*Project Explorer*) buscar el nombre del proyecto, click sobre la carpeta Liberado (*Release*) y después, doble click sobre el archivo con la extensión *.map*. Una vez abierto el archivo, buscar la sección Asignación por Secciones (*Section Allocation*) y al final de ella, se desglosará el contenido de la memoria en tres Subsecciones que son: Sólo Lectura (*Read Only*), Lectura y Escritura (*Read_Write*) y No Iniciadas (No_Init), sumar el número de Bytes que despliega por cada Subsección y ese será la cantidad total en Bytes que ocupará la aplicación dentro de la *flash*. En la Figura 4.2a se puede ver el archivo mencionado que corresponde al rol del Mote *end-device* con todos los componentes agregados, ocupando un total de 27576 bytes y en la Figura 4.2b, es el archivo sin sensores, ocupando un total de 21341 bytes.

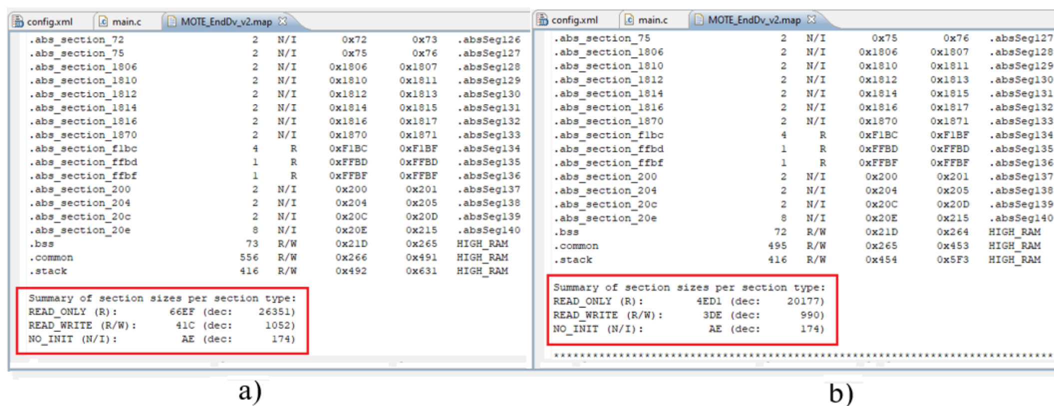


Figura 4.2: a) Cantidad predeterminada en Bytes de la aplicación para el Mote *end-device*. b) Misma aplicación, pero sin los componentes virtuales de los sensores.

El sensor que más espacio ocupa es el BMP180 debido a que manda llamar a más bibliotecas y se tiene que realizar un mayor cómputo para conocer el valor legible recopilado por el sensor. En la siguiente Subsección, se mostrarán las gráficas y análisis del consumo de energía de los Motes según el rol asignado y su ubicación física.

4.1.4 Medición de los sensores de la plataforma.

Para comprobar que los sensores propuestos están midiendo cantidades correctas, se utilizan dos aparatos de medición de uso comercial y se comparan entre sí. El primero de estos aparatos es el Extech SD700 que mide presión atmosférica, humedad y temperatura; el segundo de ellos es el TES 1333 para medir radiación solar. Los resultados obtenidos se verán a continuación.

4.1.4.1 Medición de la presión barométrica.

En la gráfica que se presenta en la Figura 4.3, se comparan las lecturas realizadas por el sensor BMP180 de nuestra plataforma contra las lecturas del aparato comercial Extech SD 700.

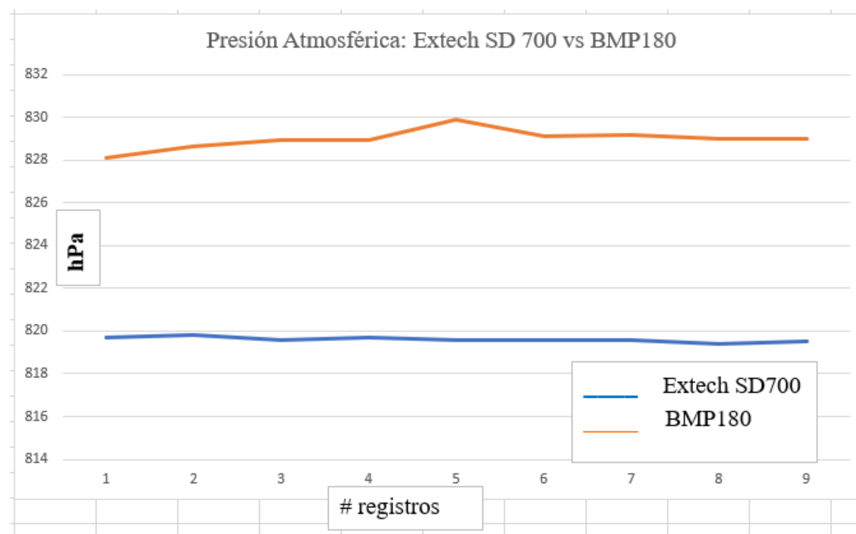


Figura 4.3: Comparación de lecturas sobre presión barométrica hechas con el BMP180 de la plataforma propuesta contra el aparato comercial Extech SD700.

De la Figura 4.3, se obtuvo un porcentaje de error de 1%, lo cual es aceptable, con lo que se comprueba que el BMP180 es recomendable como sensor de presión barométrica para nuestra plataforma.

4.1.4.2 Medición de la humedad relativa.

En la gráfica que se presenta en la Figura 4.4, se comparan las lecturas realizadas por el sensor DTH22 de nuestra plataforma contra las lecturas del aparato comercial Extech SD 700.

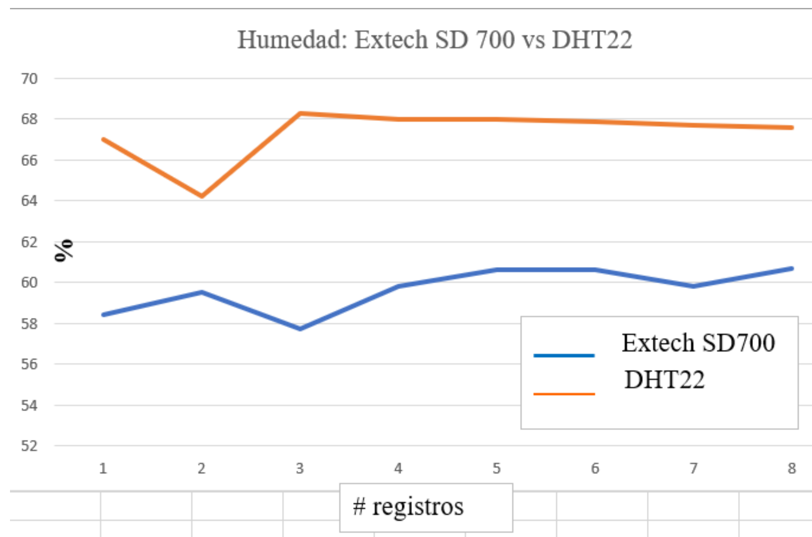


Figura 4.4: Comparación de lecturas de humedad hechas con el DHT22 de la plataforma propuesta contra el aparato comercial Extech SD700.

De la Figura 4.4, sobre las lecturas del DHT22, se obtuvo un porcentaje de error de 13%, lo cual es medianamente aceptable, aunque a pesar de ello, se tomó como sensor de humedad recomendable para nuestra plataforma. Cabe aclarar que las mediciones iniciales se realizaron con un sensor que daba un porcentaje de error del 85%, lo cual era muy alto pero se tomó la decisión de cambiar el sensor por uno nuevo y el porcentaje de error disminuyó de manera significativa, lo cual indica que el que se estaba utilizando inicialmente, estaba dañado

4.1.4.3 Medición de la temperatura.

En la gráfica que se presenta en la Figura 4.5, se comparan las lecturas de temperatura realizadas por los sensores DTH22 y BMP180 de nuestra plataforma contra las lecturas del aparato comercial Extech SD 700. El BMP180 esta considerado para medir la temperatura interna del Mote y el DHT22 para medir la temperatura externa al Mote.

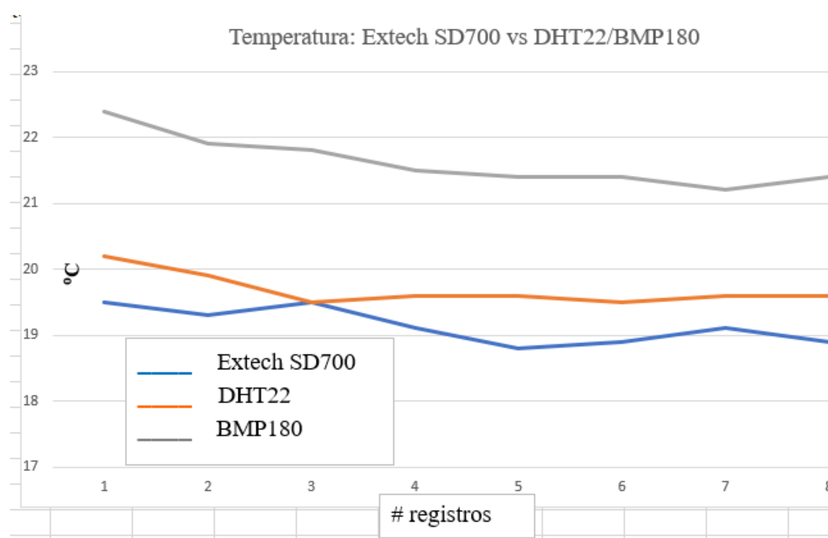


Figura 4.5: Comparación de lecturas de temperatura hechas con los sensores DHT22 y BMP180 de la plataforma propuesta contra el aparato comercial Extech SD700.

De la Figura 4.5, sobre las lecturas de los sensores propuestos; con el BMP180 se obtuvo un porcentaje de error de 13%, lo cual se podría tomar como no aceptable, pero hay que considerar que el sensor se encuentra dentro de una tortillera que encierra el calor (ver más adelante la Figura 4.8) el cual no tiene para donde escaparse, lo que provoca que los dispositivos se comporten de manera distinta. El porcentaje de error del DHT22 fue de 3%, lo que se pude considerar aceptable.

4.1.4.4 Medición de la radiación solar.

En la gráfica que se presenta en la Figura 4.6, se comparan las lecturas de radiación solar realizadas por el sensor TI OPT101 de nuestra plataforma contra las lecturas del aparato comercial TES 1333. Cabe aclarar que para obtener lecturas cercanas a las del aparato comercial, se necesitan cumplir cuando menos un par de requisitos como la utilización de un filtro especial ya que de no utilizarse, el sensor se satura lo que lo vuelve prácticamente inoperable; el otro sería poner en un mismo ángulo de incidencia de los rayos solares tanto al OPT101 como al TES 1333, pero debido a las características físicas de ambos es complicado empatar el ángulo de incidencia.

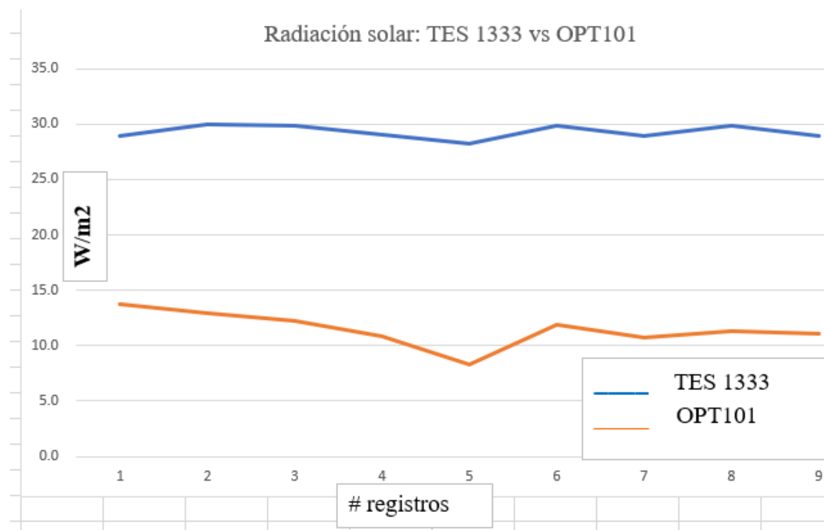


Figura 4.6: Comparación de lecturas de radiación solar hechas con el OPT101 de la plataforma propuesta contra el aparato comercial TES 1333.

Las pruebas se realizaron en horarios en el que los rayos solares presentan mayor potencia, lo que saturaría al sensor y para minimizar el impacto, se utilizó como filtro, un pedazo de acrílico sin obtener los resultados esperados. De la Figura 4.6, sobre las lecturas obtenidas con el sensor OPT101 se obtuvo un porcentaje de error de 61%, lo cual se considera como no aceptable, con lo que se demuestra que este sensor no es apto para medir radiación solar a menos, quizá, que se cumplan los requisitos antes mencionados como el uso de un filtro especial como el utilizado en [35] que hablan de un filtro de densidad neutral para este tipo de sensor. Es por todo lo anterior que se sugiere realizar un análisis a profundidad para encontrar la mejor opción pero que sea económica para su implmentación.

4.1.5 Consumo de energía de los Motes.

Uno de los atributos que debe tener un Mote para WSN en exteriores, es su bajo consumo de energía, debido a la naturaleza de la WSN en este tipo de escenarios. Es por lo anterior, que en la presente Subsección se realizaron pruebas de consumo Corriente en miliAmpers para conocer bajo que circunstancias se consumen más las baterías que los alimentan.

Electrodomésticos como un Refrigerador dan valores altos Corriente, por lo que su medición es una tarea que se puede llevar a cabo sin problemas ya que se puede emplear un amperímetro. Por otro lado, la medición de valores pequeños de Corriente, en sistemas embebidos como el prototipo del Mote propuesto, es un poco más complicado debido a su naturaleza ya que están en el orden de los microAmpers (μA) o miliAmpers (mA), las cuales no todos los amperímetros tienen la capacidad de medir dichas Corrientes. Para situaciones como esta, existen algunas soluciones como amplificadores de Corriente o voltaje, un osciloscopio como el empleado en la Subsección 4.1.5 pero, además, en el caso del osciloscopio, hay que utilizar lo que se conoce como resistencia de *Shunt*. La resistencia de *Shunt* debe de ser de un valor lo más pequeño como sea posible para conectarla en serie al circuito a analizar y con la ayuda de la ley de *Ohm* y el osciloscopio, conocer el valor de la Corriente que fluye, ya que en un circuito en serie la Corriente es la misma.
Las resistencias de *Shunt* utilizadas en las pruebas y que se conectaron tanto para el Mote *Router* como para el Mote *end-device*, tienen un valor comercial de 0.22 Ω pero cuentan con un porcentaje de error de ±5%. Para medir un valor más exacto de cada resistencia, se utilizó un medidor de impedancias, conocido como LCR modelo 1833c de la marca Hantek [39], a diferentes frecuencias, calculando el promedio de las lecturas, que dieron como resultado 0.2188 Ω para la resistencia utilizada para el Mote *end-device* y para el Mote *Router*, el promedio de su resistencia fue de 0.27387 Ω.
En la Figura 4.7, se puede observar el circuito básico utilizado para la medición de la Corriente que circula a lo largo del Mote prototipo.

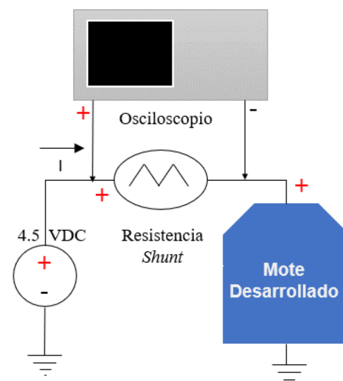


Figura 4.7: Circuito básico con resistencia de *Shunt* para conocer, con el uso de la ley de *Ohm*, la Corriente que circula por el sistema de la plataforma de HW.

Como ya se mencionó, para conocer el valor de la Corriente que circula a través del circuito de la Figura 4.7, se utilizó un osciloscopio para medir la caída de voltaje a través de la resistencia de *Shunt* y se aplicó la ley de *Ohm* que dice Voltaje=Resistencia*Corriente. Se utilizó un osciloscopio debido a que es un aparato de medición utilizado en este tipo de situaciones, aunque se utiliza mayormente para el análisis de señales eléctricas generadas por los circuitos que están siendo objeto de análisis. Los osciloscopios muestran en pantalla, una gráfica de voltaje contra el tiempo, pero una de sus ventajas es que son susceptibles a señales eléctricas pequeñas. Con el valor del voltaje obtenido por el osciloscopio y realizando el cálculo con la Ley de *Ohm*, se obtiene el valor de la Corriente que circula por el circuito. La placa prototipo de nuestra plataforma es un circuito oscilante, por lo que el consumo de Corriente variará por ciertos lapsos de tiempo, pero se puede conocer el valor promedio de consumo, tanto al estar en reposo como al enviar información vía inalámbrica. En la ficha técnica del fabricante del *Core* XBee, se comenta que al momento de transmitir información vía inalámbrica este debe de consumir 200 mA por lo que el Mote *end-device* se debe de programar de tal manera que entre en Modo de Reposo (*Sleep Mode*) por un largo periodo de

tiempo con el fin de no bajar el rendimiento de las baterías que los alimentan, ya que, de no hacerlo, se consumirían en pocos días y lo ideal es que duren meses o quizá, años.
De las dos pruebas realizadas en la Subsección 4.1.5, en la primera se conectó el osciloscopio al Mote *end-device* como se indica en la Figura 4.7, y en la segunda, al Mote *Router*. Los resultados de dichas pruebas se pueden observar en las gráficas de las Figuras 4.8 y 4.9.

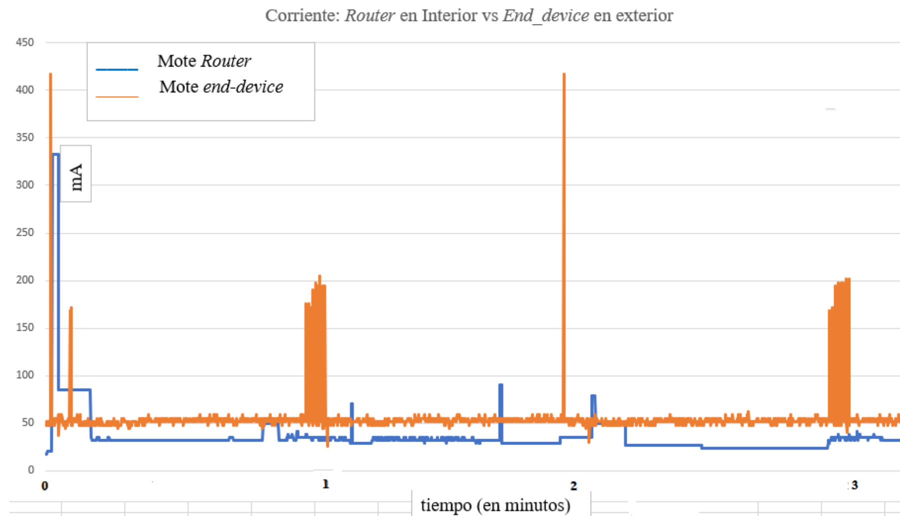


Figura 4.8: Comparación de Corriente del Mote *Router* contra el Mote *end-device*.

De la Figura 4.8, se puede observar que los valores de Corriente del Mote *end-device* (color naranja) son más altos que en el Mote *Router* (color azul), ya sea, en reposo (señal en bajo) o en algunos puntos de envío (señal en alto). El cálculo de la Corriente se pudo realizar gracias a la ley de *Ohm*, con un promedio en reposo, aproximado de 38 mA para el Mote *Router* y de 58 mA para el Mote *end-device* y al momento de transmitir información vía inalámbrica, el Mote *Router* consume poca Corriente (recordar que está en interior), aunque el Mote *end-device*, en algunos puntos, aumenta su consumo a un aproximado de 208 mA o 400 mA lo cual es alto aunque ocurre esporádicamente, esto debido a la naturaleza propia del procomi y a que se encuentra en el exterior. Cabe aclarar que el *Core* XBee seleccionado es una versión programable por lo que consume como mínimo 29 mA, es decir, las versiones que no cuentan con un MCU programable (aquellas que tienen solo el MCU Radio) consumen 14 mA menos de Corriente.

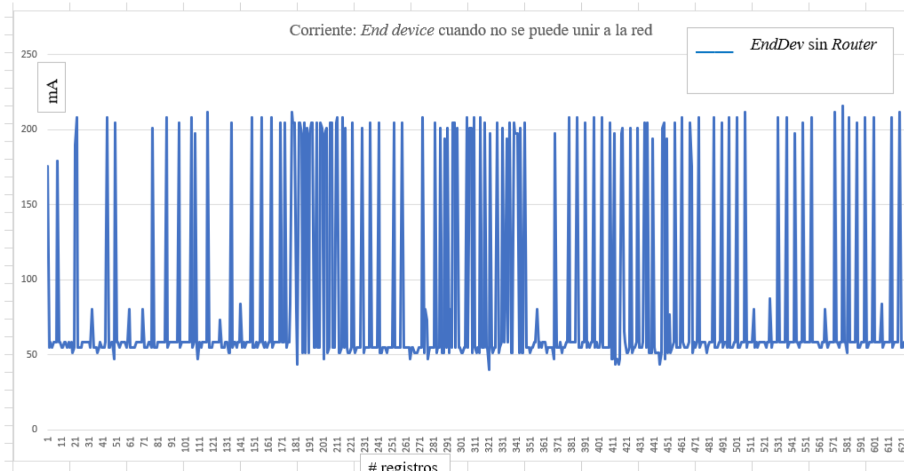


Figura 4.9: Corriente en el Mote *end-device* cuando no se puede conectar a la WSN ZB.

En la Figura 4.9, se puede observar que el Mote *end-device*, consume más Corriente cuando no encuentra al dispositivo con el que se tiene que comunicar para unirse a la red. El promedio del valor máximo de consumo al transmitir es de aproximadamente 200 mA, que concuerda con lo estipulado por el fabricante.
Con gráficas como en la Figura 4.10, se pudo observar que el Mote, sin importar su rol, consume ligeramente más Corriente cuando se expone al exterior. La gráfica corresponde al Mote *Router* dentro de un ambiente controlado como el Laboratorio de Hardware Avanzado (color azul) y, bajo las mismas condiciones, aunque ubicándolo en el exterior (color naranja); se puede ver claramente la afectación cuando se expone al exterior, aumentando ligeramente el consumo.

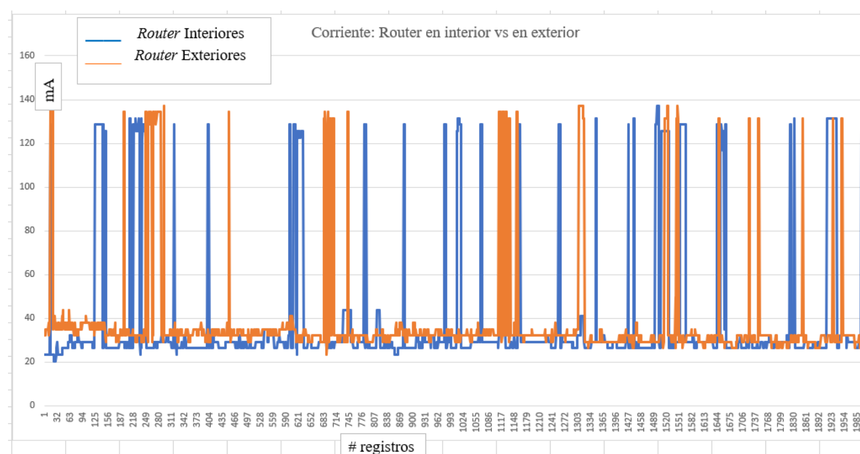


Figura 4.10: Gráfica de consumo de Corriente del Mote *Router* expuesto en el exterior (color naranja) y en el interior (color azul).

Derivado del análisis de los consumos de Corriente de los Motes según su rol en la red, se llega a la conclusión de que el Mote *end-device* consume más corriente que el Mote *Router* pero además, en cualquiera de los roles, si se expone al exterior o no se une a la red, consume un poco más, lo que acortaría el tiempo de vida de las baterías.
Como comentario adicional, al momento de realizar la prueba anterior, se pudo observar que la señal de WiFi emitida por un *Router* WiFi que se encuentra dentro del mismo laboratorio, llegó a un aproximado de 65 m como máximo, esto se pudo comprobar con ayuda de un

teléfono inteligente *Samsung Galaxy* J4, y se tenía la hipótesis de que su distancia máxima sería de un aproximado de 20m. Recordar que alguna de las versiones de la señal de WiFi opera en la misma banda de frecuencia que ZB y su tasa de transferencia es más alta comparándola con cualquier otro procomi que opere en la misma frecuencia.

4.1.5 Establecer la WSN ZB.

El resultado de lo realizado en las Subsecciones 4.1.1, 4.1.2 y 4.1.3 es un SW embebido personalizado que se instalará dentro del MCU programable y permitirá que se utilicen únicamente los sensores seleccionados.

Para poder comprobar que es posible la implementación sencilla de una WSN ZB, como primer paso hay que establecerla dentro de un ambiente controlado para asegurar su funcionalidad y después establecerla en ambientes externos, donde existen mayores retos como el evitar daños debido a la fauna de la zona a monitorear.

Se realizaron dos pruebas de una WSN ZB conformada por un Coordinador *XStick* de *Digi* conectado a una PC y con la consola de terminal del IDE CWDS v10.2 para poder capturar dentro de un archivo .txt la información condensada de todos los Motes *end-device* e incluso del Mote *Router*, un Mote *Router* con los sensores BMP180 y DHT22 conectados y uno de cuatro Motes *end-device* con todos los sensores propuestos y conectados según el *pinmap* de la Figura 3.2. Los Motes se programaron para enviar su información generada cada 25 segundos, se les conectó una antena RPSMA de 4 mm, para obtener un alcance máximo de transmisión, también se protegieron con una tortillera de unicel debido a que había que ubicarlos en un ambiente externo donde existen factores como los rayos solares, lluvia, entre otros; las pruebas tuvieron una duración de una hora cada una y se realizaron dentro de la Zona Poniente de la UASLP y el Laboratorio de Hardware Avanzado. El Mote *end-device* con los sensores, fue el único dispositivo que estuvo a la intemperie, con baterías nuevas, ubicándose a 75 m del Mote *Router* (aunque un día antes, se probó que puede transmitir a 80 m como máximo) que se encontraba dentro del laboratorio. Las condiciones del día fueron seminublado, mayormente despejado al momento de las pruebas, con viento fresco a la sombra y calor a los rayos del sol. El Mote *Router* se ubicó a 3 m del Coordinador, ambos junto con los demás Mote *end-device* dentro del Laboratorio de Hardware Avanzado.

Es importante comentar que se experimentó con cuatro versiones distintas de código del *firmware* del Mote sin importar su rol, con el fin de optimizar la tarea de comunicación inalámbrica siendo la más confiable la versión utilizada en las pruebas, debido a que se conecta de manera automática con los dispositivos con los que se tienen que comunicar, gracias a la configuración de un nombre corto pero descriptivo para cada uno de los Motes que pertenecerán a la red. En la Figura 4.12 se muestran los Motes *Router* y *end-device* desarrollados.

La primera prueba se llevó a cabo en el horario de las 12:56 a las 13:56 hrs a una distancia de 75 m (ver Figura 4.11), conectando el osciloscopio BK *Precision* 2511 [40] al Mote *end-device* para grabar su consumo de Corriente dentro de una memoria USB, utilizando la función *Trend plot* (Tendencia de la gráfica) del osciloscopio. Las grabaciones se realizaron cada 5 minutos debido a las circunstancias de grabación.

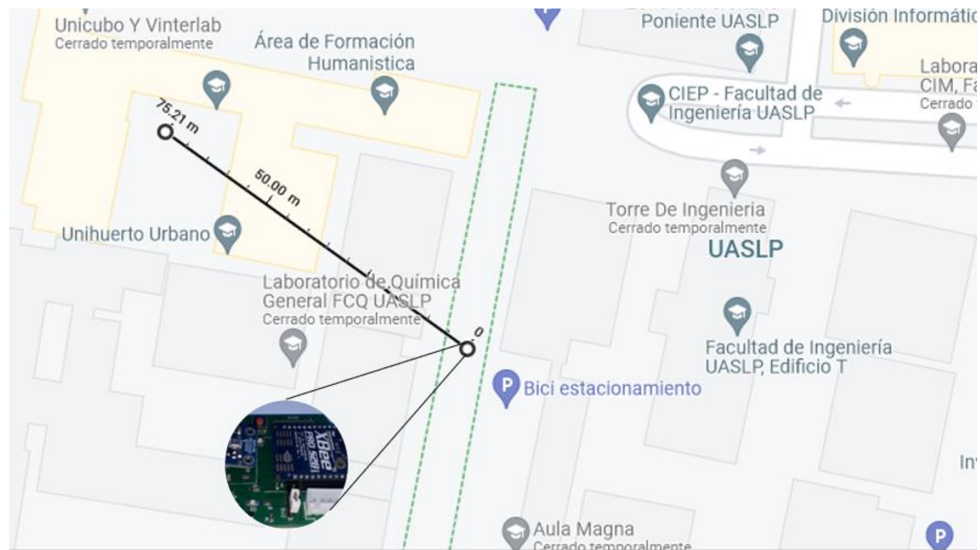


Figura 4.11: Ubicación del Mote *end-device* en las pruebas realizadas en exterior.

La segunda prueba, se llevó a cabo en el horario de las 14:20 a las 15:20 hrs casi en las mismas condiciones que la primera prueba solo que el osciloscopio se conectó al Mote *Router* y debido a que el Mote *end-device* con los sensores, se ubicó en un ambiente externo, se tomó la decisión de realizar las grabaciones del consumo del *Router* cada 15 min (con una de 20 min). Quince minutos antes de finalizar la segunda prueba, se ubicó al *end_device* a 85 m por 7:30 minutos y a 90m por otros 7:30 minutos, pero en ambos casos, el Coordinador no recibió la información generada. En la Figura 4.12a, se pude observar al Mote *end-device* utilizado externamente en ambas pruebas y en la Figura 4.12b, al Mote *Router*.

Figura 4.12: a) Mote *end-device* utilizado en las pruebas. b) Mote *Router*.

4.2 Usabilidad de la plataforma.

La medición de la usabilidad de un producto tanto de HW como de SW no es una tarea fácil, pero nuestra plataforma de HW/SW para monitorización ambiental es sencilla de implementar y de fácil aprendizaje gracias a un manual del programador (ver Anexo I), debido al uso de una metodología de desarrollo como *Co-design* se tiene una estructura, se aumenta la productividad además de ahorrar tiempo de implementación, se evita agregar código innecesario ya que se pueden seleccionar los módulos de HW de una manera intuitiva mediante una GUI con lo que también se ahorrará energía de las baterías ya que sólo se activan los módulos de HW acorde a

las necesidades del usuario final del sistema. El hecho de ser una plataforma genérica pensada en proveer una solución a todo proyecto que lo considere un apoyo para la solución de su problemática conlleva que sea una plataforma flexible en su uso. Además, se pueden crear componentes virtuales de HW que facilitarán el trabajo de programación de los Motes para poder implementar una WSN ZB, esto permite adquirir la habilidad necesaria para su uso.

4.3 Comparación con otras plataformas.

Al igual que la usabilidad, la mantenibilidad de HW y SW no es un atributo sencillo de comprobar. Para seguir ofreciendo una plataforma de HW y SW de calidad, y que su tiempo de vida sea largo, es necesario darle mantenimiento tanto a la parte del SW como la del HW [41]. Uno de los objetivos del presente trabajo, es utilizar una metodología de diseño y desarrollo de HW y SW como Co-*desing* que ayuda a agregar nuevas funcionalidades, corregir las existentes o realizar actualizaciones a la plataforma propuesta. Es por lo anterior que nuestra plataforma es mantenible en comparación con otras plataformas encontradas en el estado del arte como *Sprouts* y *Cookie Nodes* [2, 3].

En la TABLA 4.1 se puede observar una comparación con las plataformas encontradas en el estado del arte; en dicha tabla se tomaron como base cuatro criterios comunes entre ellas.

TABLA 4.1: COMPARACION CON PLATAFORMAS ENCONTRADAS EN EL ESTADO DEL ARTE

Características	Plataforma propuesta	SPROUTS [2]	COOKIE NODE [3]
Número de núcleos	1 por nodo	2 por nodo	3 por nodo
Consumo energético	Bajo	Medio	Medio
Mantenibilidad	Alta	Baja	Media
Usabilidad	Alta	Baja	Media

La TABLA 4.1 confirma que nuestra plataforma es una mejor opción para implementar un WSN de manera ágil, eficiente y a bajo costo.

Las pruebas experimentales que se realizaron confirman que es posible implementar una WSN ZB con una topología de tipo estrella extendida que incluye: un Coordinador, un Mote *Router*, y cuatro Motes *end-device*. Debido a falta de recursos, no se pudo comprobar el número máximo de nodos que puede tener la WSN ZB antes de que se presenten problemas de colisión de paquetes de datos que afecten de forma negativa su desempeño.

En los experimentos realizados, los cuatro *end-devices* hacen el sensado de las variables y envían la información al *Router*. El *Router* está configurado de tal manera que realiza ambas tareas, la de enrutamiento y la de sensado. El Coordinador recibe del *Router* la información condensada de todos los Motes (incluyendo la del propio *Router*). El Coordinador tiene contacto directo con una PC común que usa la consola del IDE CWDS v10.2 (herramienta del fabricante) para permitir al administrador supervisar la información generada por la red, así como registrar todas las lecturas dentro de un archivo de texto plano con el registro histórico.

Con este Capítulo se termina la iteración completa de la metodología *Co-design* HW/SW propuesta, así como la finalización del presente trabajo de tesis. Mediante una serie de pruebas realizadas, se pudo comprobar el cumplimiento de todos los requisitos planteados en la Subsección 2.1 y en caso de requerirse nuevas funcionalidades, reparar o mejorar las existentes, solo hay que seguir la metodología propuesta mostrada en la Figura 2.1. Más adelante, dentro de las conclusiones se comenta sobre la contribución del presente trabajo, que obstáculos se presentaron y el trabajo a futuro del proyecto.

Conclusiones

Conclusiones del desarrollo

La principal conclusión es el uso de Co-*design* para implementar una metodología en el desarrollo y evaluación de una plataforma de HW/SW genérica basada en componentes virtuales de HW, que facilita la implementación de una WSN ZB funcional y fiable, reduciendo el tiempo de desarrollo además de recursos esenciales como la energía de las baterías de los Motes.

En el presente trabajo de tesis se describieron los pasos a seguir en el desarrollo de Motes que puedan ser implementados e integrados de manera sencilla dentro de una WSN ZB para el cuidado del medio ambiente, ya que en su proceso de implementación se puede hacer uso de componentes virtuales de HW que agilizan su programación, todo ello siguiendo una metodología de desarrollo conocida como *Co-design* HW/SW. Los diferentes experimentos que se llevaron a cabo comprobaron como un Mote puede integrarse a una WSN ZB seleccionando algunas configuraciones sencillas al momento de programarlos, lo que permite, entre otras cosas, el poder identificarlos fácilmente dentro de la misma.

La contribución del presente trabajo de tesis es al conocimiento ya que, basado en el análisis del estado del arte, no se encontró una propuesta como la nuestra donde se detallen los pasos a seguir de una metodología además de proponer una plataforma genérica de HW y SW que ayude al programador del mañana a agilizar su trabajo en el desarrollo de prototipos por medio de una interfaz amigable que, en el caso específico del trabajo realizado, permite la implementación de una WSN ZB, un tema relevante de hoy en día. A continuación, se enumeran mis contribuciones derivadas de mi trabajo de investigación:

1. Uso de la metodología Co-*desing* HW/SW para el diseño y desarrollo de un prototipado como el del trabajo de investigación.
2. Plataforma de HW/SW que, debido a su naturaleza genérica, se pude adecuar a las necesidades del cliente final.
3. Manual básico para el programador.
4. Se pone a disposición del código del FW para el MCU programable, así como los archivos XML para los componentes virtuales de HW.
5. El presente documento de tesis.

Y sobre el mismo contexto, al desarrollar una plataforma genérica de HW/SW basada en componentes virtuales de HW, que permite implementar WSN ZB con topología de estrella extendida de una forma eficiente, ágil y versátil enfocadas al monitoreo ambiental, se pueden deducir cuatro ventajas importantes:

1. Uso y seguimiento de una metodología para el diseño, desarrollo e implementación de proyectos, algo muy importante dentro de la ingeniería de SW.
2. Agiliza el desarrollo de Motes para WSN.
3. Se reduce el uso de recursos de cómputo del *Core* XBee, ya que solo se habilitan los módulos y funciones requeridos, lo que también permite reducir el consumo de energía.
4. El costo económico de los recursos de HW se reduce debido a que nuestra plataforma de HW/SW ocupa como elemento principal al *Core* lo cual reduce el tiempo de implementación y costo de los Motes.

La comparación con otras plataformas del estado del arte en lo que se refiere a costo, consumo energético y mantenibilidad, confirma que nuestra propuesta es una mejor opción para implementar este tipo de WSN.

Como se pudo observar en la TABLA 4.1, el comparativo realizado con otras plataformas similares confirma que nuestra plataforma es una mejor opción para implementar una WSN de una manera ágil, eficiente y a bajo costo.

Trabajo a futuro

Debido al constante cambio que sufre la tecnología y el intento por mejorar el rendimiento de un sistema, es lógico pensar que el presente trabajo de investigación tendrá nuevos requerimientos o perfeccionar los existentes. Algunas características en las que podrían cambiar los requerimientos es el consumo energético del sistema, bajos costos de producción, mantenibilidad o lograr una mayor escalabilidad. A continuación, se mencionan algunas oportunidades de mejora para la plataforma que no se encuentran dentro del alcance del presente trabajo de investigación:

1. Realizar pruebas de la WSN ZB en exteriores.
2. Investigar un dispositivo independiente de los llamados *Gateway* que ayude en la integración de la WSN ZB al IoT.
3. Diseñar un Módulo de Energía (EnMo) que monitorice el consumo dentro del Mote para optimizar el consumo de sus baterías.
4. Diseñar una carcasa protectora, ya que el Mote se desarrollo para ser implementado a la intemperie.
5. Realizar un análisis para determinar si es necesario agregar encriptación a los datos que viajan a través de la WSN ZB.
6. Realizar el mantenimiento de los Motes, actualizándolos de manera remota (actualización del FW del MCU programable, a través del procomi ZB).
7. Poder configurar parámetros específicos de los Motes, desde un smartphone o desde la web.
8. Mejorar la medición de los sensores mediante SW o bien buscar otras alternativas de mayor precisión.

Anexo I

Detalles técnicos de la plataforma.

La presente guía pretende ser una herramienta de apoyo para el programador del *Core* XBee que requiera implementar una WSN ZB ya que le permitirá definir una GUI para el IDE CWDS v10.2 con la que se podrán agregar componentes virtuales de HW a las aplicaciones para el *Core* XBee, de una manera ágil y versátil aumentando la productividad y disminuyendo tiempos de desarrollo así como costos de implementación.

A continuación, se indican las herramientas necesarias tanto de HW como de SW que permitirán la implementación de la WSN ZB. En las secciones posteriores, estas herramientas serán explicadas ya sea de manera general o a detalle.

Antes de finalizar, se dará un ejemplo práctico de como implementar una pequeña WSN ZB, a manera de demostración.

Herramientas de HW.

El HW mínimo necesario para pruebas e implementación de una WSN ZB con topología de estrella que utilice al *Core* XBee, es el siguiente:

- **Coordinador**. Para este rol se puede utilizar un *Core* XBee (ver Figura 5.1.a) o un Digi X-Stick (ver Figura 5.1.b). En caso de seleccionar un *Core* XBee, hay que transferirle el FW ZigBee *Coordinator* API a su MCU Radio con la herramienta de SW XCTU (ver más adelante).
- **Router**. Generalmente se selecciona un *Core* XBee para este rol por lo que hay que transferirle el FW de ZigBee *Router* API a su MCU Radio con la herramienta de SW XCTU.
- **End-device**: Para este rol se selecciona un *Core* XBee por lo que hay que transferirle el FW de ZigBee *end-device* API a su MCU Radio, con la herramienta de SW XCTU (ver más adelante).

Figura 5.1: a) *Core* XBee.   b) X-Stick.

- **P&E USB Multilink**: Dispositivo que se utiliza para poder transferir aplicaciones además de poder realizar una depuración directa con el *Core* XBee (ver Figura 5.2.a).
- **Digi XBIB – U – DEV**: Tarjeta de desarrollo para la programación, depuración y pruebas de los *Cores* XBee (ver Figura 5.2.b).

- **PCB Prototipo**: Aquí va conectado el *Core* Xbee y de esta manera se obtiene un Mote. Se le puede conectar un sensor analógico y dos sensores digitales gracias a sus interfaces físicas.

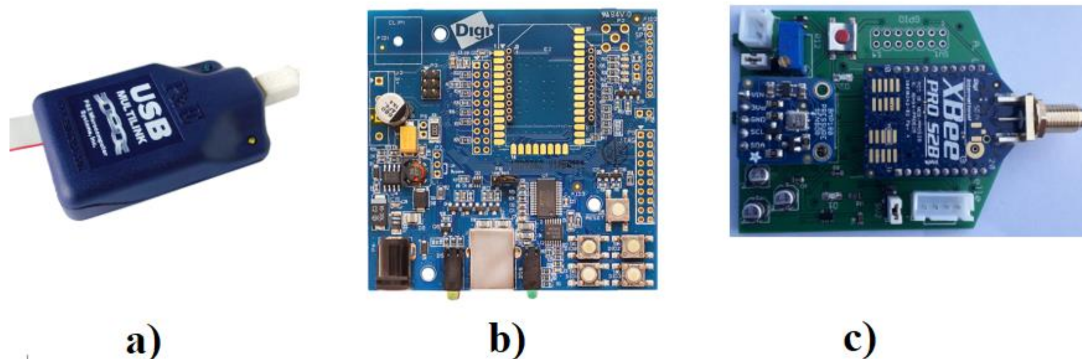


Figura 5.2: a) Programador. b) XBIB. c) Mote.

- **Sensores**:
  - ✓ Digitales: BMP180, DHT22.
  - ✓ Analógicos: OPT101.

  Ver Figura 5.3.

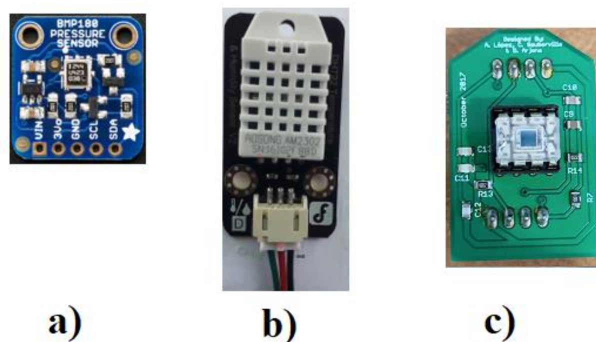


Figura 5.3: Sensores que se pueden conectar al Mote: a) BMP180. b) DHT22. c) OPT101.

- **Accesorios adicionales**: protoboard, cables Dupont, cables USB para impresora, pilas AA (tres por Mote), portapilas (uno por Mote).

Herramientas de SW.

El SW necesario para la programación de los dispositivos que pertenecerán a una WSN ZB, son:

- **Digi XCTU**: Aplicación que proporciona el fabricante del *Core* XBee, necesaria para transferir el FW para el MCU Radio del *Core* XBee según su rol dentro de la WSN ZB; también se utiliza para cambiar algunas configuraciones de red, hacer pequeñas pruebas de funcionamiento del MCU Radio o analizar la llegada de información provenientes de otros dispositivos ZB de una manera sencilla (ver Figura 5.4.a).
- **IDE CWDS v10.2**. Ver Figura 5.4.b.
- **Digi XBee SDK**: Es un *plug-in* para el IDE CWDS v10.2. Ver Figura 5.4.c.

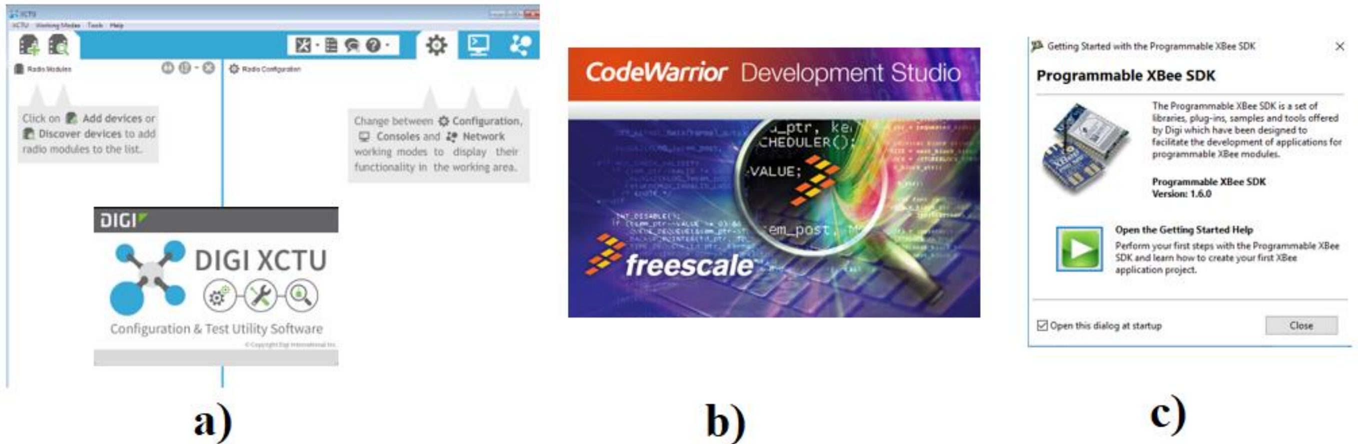


Figura 5.4: Herramientas de SW: a) XCTU. b) CWDS v10.2. c) XBee SDK.

XCTU

Para interactuar con el MCU Radio del *Core* XBee, el fabricante Digi provee como herramienta de SW al XCTU que permitirá transferirle el FW al MCU Radio según el rol que le corresponderá dentro de una WSN ZB. Con esta herramienta, no solo se podrá transferir el FW si no también, enviar mensajes de texto simples de una manera sencilla o analizar paquetes recibidos de dispositivos remotos a través de ZB.

Instalación

Los pasos a seguir son:

a. Instalar los controladores para la U-DEV:
b. Instalar los controladores para la Digi X-Stick.
c. Los controladores para el programador se instalan al instalar el IDE.
d. Una vez instalados los controladores de los diferentes dispositivos, instalar el XCTU. Ver Figura 5.5 sobre la interfaz general del XCTU.

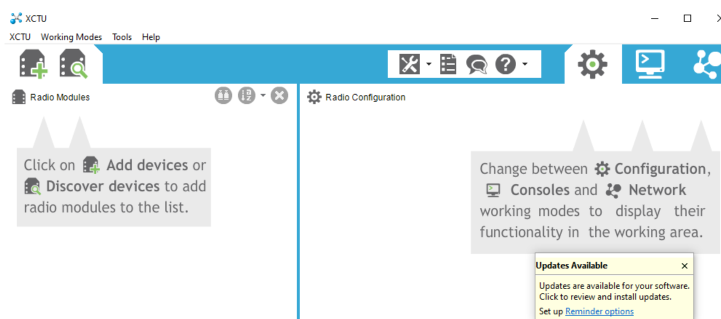


Figura 5.5: Interfaz general del Digi XCTU.

Transferir FW al MCU Radio del *Core* XBee.

Para que los dispositivos de una WSN ZB con *Core* XBee puedan cumplir un rol dentro de la misma y de esta manera poder integrarse a ella, se le tiene que transferir un FW al MCU Radio según el rol que cumplirá el *Core* XBee dentro de ella ya que el FW contendrá las rutinas

específicas necesarias. Con respecto al coordinador, que en nuestro caso es un X-Stick, no es necesario transferirle ningún FW.

Para el *Core* XBee, lo primero que se tiene que saber antes de transferir el FW a los dispositivos correspondientes, son los modos de comunicación AT o API. Estos modos son específicos para el *Core* XBee. A continuación, se da una breve explicación de estos modos de comunicación:

- AT (*Application Transparent*, Aplicación Transparente): En este modo de comunicación, la información se envía tal y como se recibe. Este modo se selecciona cuando se desea comprender algunos conceptos, realizar pruebas simples de envío y recepción de mensajes o simplemente, analizar los paquetes de datos. Dentro de este modo también se puede utilizar el modo comando el cual se utiliza para comunicación entre el XCTU y el MCU Radio a través de UART; para establecer dicha comunicación y teclear los comandos AT deseados. Ejemplo de uso de comandos AT:
  +++ok
  atsl
  40E6D9A2

  Donde, los caracteres en color azul indican lo que el usuario tiene que teclear dentro de la consola de terminal del XCTU y los caracteres en color rojo indican que el *Core* XBee está respondiendo a los comandos que se le están enviando. Después de haber recibido contestación por parte del *Core* XBee, teclear los comandos a utilizar lo más pronto posible ya que después de 3 segundos, se sale de sesión en automático y habría que volver a iniciarla.
- API: En este modo, el envío de información se particiona en Paquetes de Datos (*Frames*) a los que se les agrega información que asegurará su llagada al destino por ello es un modo más seguro con respecto al modo AT.

Una vez que conoce que modo de comunicación es el adecuado para el *Core* XBee además de su rol dentro de la WSN ZB, se explica paso a paso, como transferir el FW con el modo de comunicación necesario al *Core* XBee correspondiente:

¡Precaución!: Para evitar daños irreversibles, antes de actualizar el FW del MCU Radio del *Core* XBee, asegurarse de que la PC o el mismo *Core*, no se apaguen al momento de realizar la actualización.

a. Definir el rol del *Core* XBee dentro de la WSN ZB.
b. Transferirle el FW a su MCU Radio según el rol definido.

¡Importante!: La familia del producto (ver Figura 5.6) cambia según el *Core* XBee que se utilice.

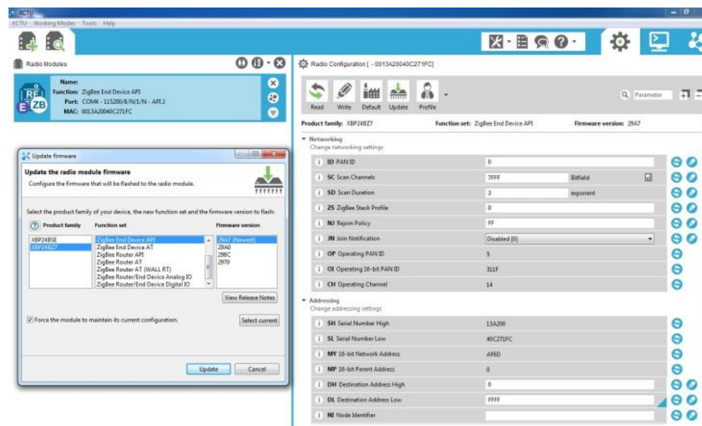


Figura 5.6: Actualización del FW del *Core* XBee con el XCTU.

Ejemplos de uso de modos AT/API

Para entender mejor los diferentes modos de comunicación del *Core* XBee, se mostrarán algunos ejemplos prácticos, utilizando el XCTU.

a. Envío simple de mensajes de texto entre dos *Cores* XBee en modo AT.
   i. Se necesita una PC con XCTU instalado, dos puertos USB disponibles, dos cables USB para impresora, dos U-DEV, dos *Cores* XBee, transferirles el FW ZB *Coordinador* AT al primero de ellos y el FW ZB *Router* AT, al segundo.
   ii. Ejecutar el XCTU, conectar las U-DEV a la PC, y agregados al área de Módulos de Radio (*Radio Modules*), seleccionar al *Core* XBee que tendrá el rol de coordinador -> buscar y configurar los siguientes campos:
      - PAN ID: *<# HEX entre 0-FFFFFFFFFFFFFFFF; debe de ser el mismo para todos los dispositivos pertenezcan a una WSN ZB>*
      - *Operating Channel*: *<Lo asigna el Coordinador por lo que solo hay que verificar que sea el mismo para todos los dispositivos>*
      - *Destination Address High*: *<8 dígitos más significativos de la MAC Address del dispositivo con quien se desea comunicar>*
      - *Destination Address Low*: *<8 dígitos menos significativos de la MAC Address del dispositivo con quien se desea comunicar>*

      Ejemplo:
      ✓ `PAN ID: 11`
      ✓ `Operating Channel:` *<asignado por el coordinador>*
      ✓ `Destination Address High: 0013A200`
      ✓ `Destination Address Low: 40E6D9A2`

   iii. Click en el icono del lapicero para guardar cambios.
   iv. Click en el icono del monitor para conectarse con el Coordinador vía consola de terminal.
   v. Click en el icono Cerrar (*Close*) para establecer comunicación entre el XCTU y el *Core* XBee.
   vi. En el área de texto Consola de Registro (*Console log*) del coordinador, teclear el mensaje que se desee enviar hacia el *Router* (en este último, ya se debieron haber realizado los pasos anteriores, que integran al *Router* dentro de la WSN ZB). En el origen, deberá desplegarse el mensaje en color azul y en el destino, en color rojo.

Si se desea enviar un mensaje como un paquete de datos, en el área de texto Enviar Paquetes (*Send Packets*), click en el símbolo + (más), se desplegará el cuadro de diálogo Agregar paquete de datos a la lista (*Add a data packet to the list*) -> en el campo Nombre del Paquete (*Packet Name*), teclear un nombre corto y descriptivo -> en la pestaña ASCII, teclear el mensaje a enviar -> click en Agregar Paquete (*Add Packet*) -> seleccionarlo de la lista de paquetes -> click en Enviar Paquete Seleccionado (*Send Selected Packet*).

vii. Repetir los pasos para el *Core* XBee *Router*.

viii. La interfaz de la consola de terminal del XCTU en modo AT se puede ver en la Figura 5.7.

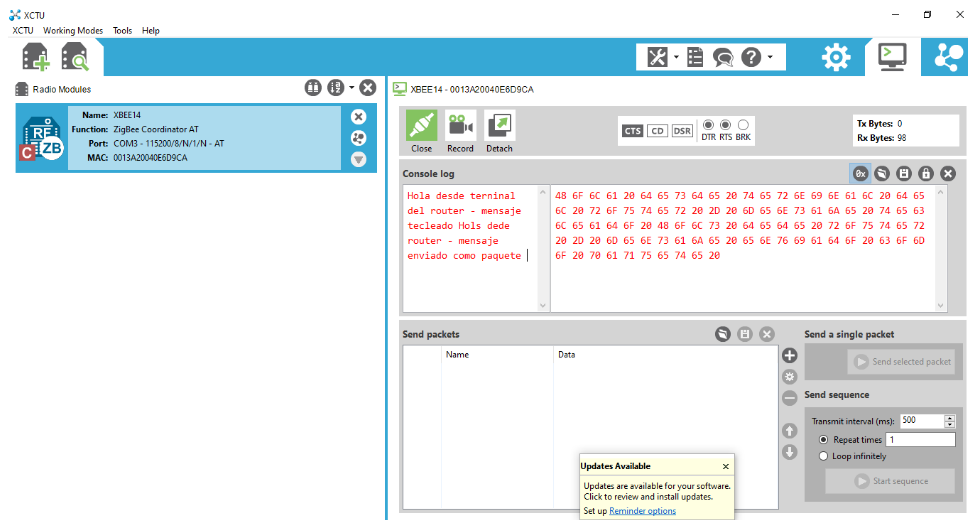


Figura 5.7: Ejemplo de recepción de mensajes en texto plano dentro de la consola del XCTU, para el Coordinador de la WSN ZB en modo AT.

NOTA: La interfaz de la Figura 5.7 se desplegará solo si el FW del MCU Radio está en modo AT.

b. Envío simple de mensajes de texto entre dos *Cores* XBee en modo API.

i. Transferirle al MCU Radio de uno de los dos *Core* XBee que se ocuparán, el FW ZB *Coordinator* API y al segundo, el FW *Router* API.

ii. Conectar los dispositivos involucrados.

iii. Enviar un mensaje hacia un *Core* XBee remoto, dentro del área de texto Enviar Paquetes de Datos (*Send frames*), click en el símbolo + (más), se desplegará el cuadro de diálogo Agregar paquete de datos API a la lista (*Add API frame to the list*) -> en el campo Nombre del Paquete de Datos (*Frame name*), teclear un nombre corto y descriptivo -> click en Crear un paquete de datos utilizando la herramienta generador de paquetes (*Create frame using frames generator tool*) -> se abrirá el cuadro de diálogo Generador de paquetes de datos API para XBee (*Xbee API frames generator*) -> en el campo Tipo de Paquete de Datos (*Frame Type*), seleccionar 0x10 Requerimiento de Transmisión (*0x10 Transmit request*) para poder transmitir el mensaje vía ZB -> en el campo Dirección Destino de 64 bits (*64 bit Dest. Address*) teclear la MAC Address del dispositivo remoto -> en el campo Datos (*RF Data*), pestaña ASCII, teclear el mensaje que se desea enviar. En el área de texto Paquete de Datos Generado (*Generated Frame*) se desplegará el paquete API que se genera

-> click en Aceptar (*Ok*) -> se agregará a la lista disponible, click en Enviar Paquete de Datos Seleccionado (*Send selected frame*) -> verificar que en el destino que haya llegado el paquete de datos.

iv. En el área de texto Registro de Paquetes de Datos (*Frames Log*), se desplegará la entrada y salida de paquetes. Seleccionar un paquete de datos de entrada o salida y dentro del área de texto Detalles del Paquete de Datos (*Frame Details*) se desplegará un desglose a detalle de cada campo que compone un paquete en ZB.

v. Para enviar un mensaje desde el *Router* hacia el Coordinador solo habría que repetir los pasos i e ii.

vi. La interfaz de la consola de terminal en modo API se puede observar en la Figura 5.8.

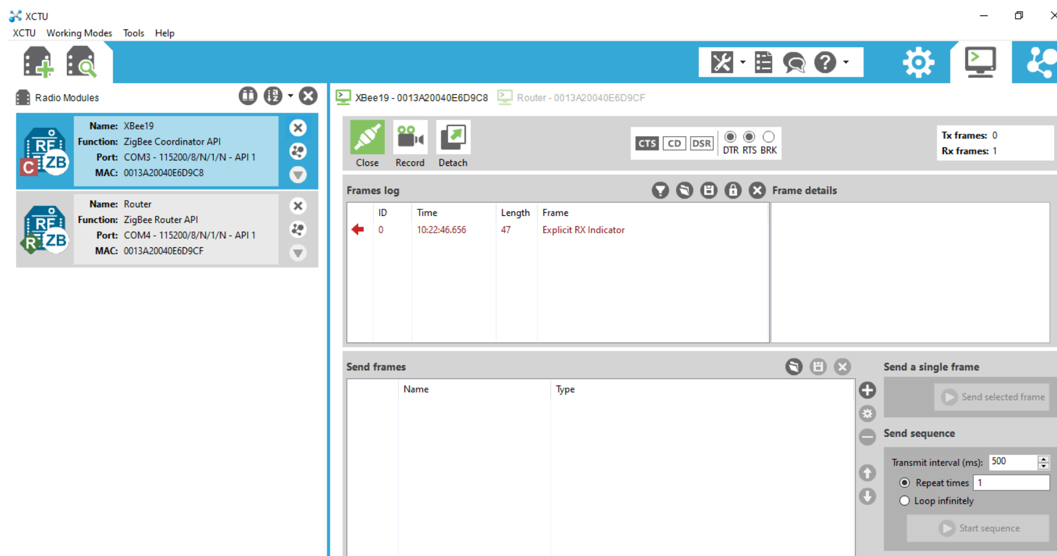


Figura 5.8: Ejemplo de recepción de mensajes en texto dentro de la consola del XCTU para el Coordinador de la WSN ZB en modo API.

NOTA: La interfaz de la Figura 5.8 se desplegará solo si el FW del MCU Radio está en modo API.

c. Envío simple de mensajes de texto entre dos dispositivos, uno en modo AT, otro en modo API.

i. Transferirle al MCU Radio de uno de los dos *Core* XBee que se ocuparán, el FW ZB *Coordinator* API y al segundo, el FW *Router* AT.

ii. Conectar los dispositivos involucrados.

iii. Enviar un paquete desde el coordinador hacia el *Router*. Ver Figura 5.9.

iv. Si se desea, también se puede enviar un mensaje desde el *Router* AT hacia el Coordinador API.

a)

b)

Figura 5.9: Ejemplo de recepción y envió de mensajes por ZB con *Cores* XBee en modo API y AT: a) Coordinador API. b) *Router* AT.

Como se pudo observar, es posible, aunque no recomendable, tener una combinación de MCU Radio del *Core* XBee con FW en modo API o AT.

CWDS v10.2

Toda IDE permite una mejor versatilidad y agilidad al momento de crear aplicaciones ya que integra en una sola aplicación las herramientas necesarias para el desarrollo de SW además de proveer una interfaz gráfica que ayuda a tener una mejor visualización de dichas herramientas.

Como ya se mencionó, el MCU programable es producido por el fabricante NXP *semiconductors*/*Freescale*. Este MCU se encuentra embebido dentro del *Core* XBee que es fabricado por Digi. Para la creación de aplicaciones para el MCU programable se debe de utilizar como herramienta al IDE CWDS v10.2 de *Freescale* junto con el XBee SDK de Digi que contiene las herramientas básicas de todo SDK además de un analizador de XML que permitirá crear componentes virtuales de HW y poder agregarlos mediante una GUI.

El IDE CWDS v10.2 cuenta con un ensamblador, un compilador de C y C++, un intérprete/compilador de micro Java y versiones actuales del *Core* XBee, cuentan con intérprete de *MicroPhyton*.

A la plataforma donde se instala el IDE, se le llama Anfitrión (*host*), y a las aplicaciones que se diseñen se dice que han sido diseñadas para un *target* (que se traduce como destino pero se empleará el término en inglés) refiriéndose al destino de la aplicación y en el que se estará ejecutado la aplicación mientras el dispositivo este energizado.

Por lo anterior, es necesario saber que hay dos tipos de *target*:

- **Simulado**: es donde la aplicación se ejecuta dentro de la PC del IDE.
- **Real**: es cuando la aplicación se ejecuta dentro del *Core* destino mientras el dispositivo se encuentre energizado. Ejemplo: un *target* real es el *Core* XBee.

Instalación

a. Primero, instalar los controladores para la X-Stick y U-DEV.
b. Instalar el IDE.

Interfaz principal del IDE

Para entender al IDE CWDS v10.2, solo se comentan los aspectos más relevantes de su interfaz principal, identificar sus áreas de interés para poder comprender su funcionamiento general; no se profundiza en el IDE debido a que es un tema muy extenso. Para comenzar, ver la Figura 5.10.

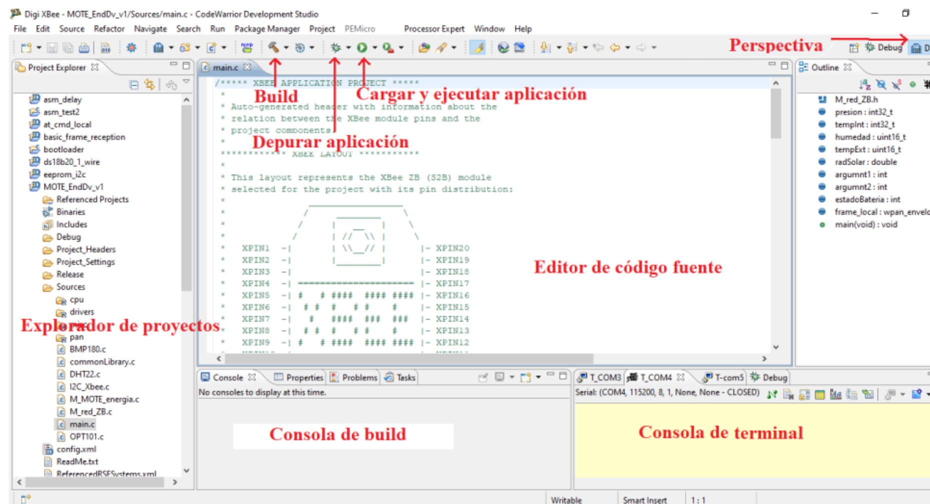


Figura 5.10: Interfaz principal del IDE CWDS v10.2.

De la Figura 5.10, se explicarán de manera general, los puntos más relevantes como son:

- **Crear ejecutable**: Al terminar de teclear el código correspondiente a la aplicación que se esté creando para el *Core* XBee, hay que Crear el ejecutable (*Build*). Al crear el ejecutable, lo que se está realizando son dos procesos en uno solo; estos son el proceso de compilación del código y el proceso de enlace.
  Existen dos maneras de realizar la creación de un ejecutable:
  - Depurar (*Debug*): Seleccionar esta opción si se desea depurar el código antes de transferirlo a un *target* real.
  - Liberar (*release*): Si después de realizar pruebas de la aplicación, todas resultan satisfactorias, solo faltará que se ejecute dentro del *target* real, por lo que hay que seleccionar esta opción.
- **Cargar y depurar aplicación**: Cuando se desea depurar, click en este cuadro combinado para transferir el código hacia el *target* real.
- **Cargar y ejecutar aplicación**: Cuando se desea ejecutar la aplicación, click en este cuadro combinado para transferir el código hacia el *target* real.
- **Perspectiva**: Click en Digi XBee cuando se esté creando una aplicación para el *Core* XBee o en Depurar, cuando se vaya a realizar una depuración directa con el *target* real.
- **Editor de código fuente**: En esta área de la pantalla teclear el código fuente en lenguaje C de la aplicación que se desea a desarrollar.
- **Explorador de proyectos**: En esta área se desplegarán los nombres de proyecto para el *Core* XBee, mediante una estructura de árbol jerárquico.
- **Consola de creación del ejecutable**: En esta área se desplegarán los posibles errores tanto del proceso de compilación como de enlace. Los errores en el proceso de compilación iniciarán con la letra C seguido de un código numérico y los del proceso de enlace con la letra L. Ejemplos:
  - ✓ Un error de compilación muy común sería *C5200: file not found* que indica que se está llamando a un archivo que no está agregado al proyecto.
  - ✓ Un error de enlace muy común sería *L1822: symbol … in file …* es debido a que no se añadieron las bibliotecas para el manejo de módulos de HW.

Los errores más frecuentes serán con el Enlazador Inteligente (*Smart linker*) del CWDS v10.2 cuyo trabajo son recopilar la información de las bibliotecas y asignar bloques de memoria del *target* real que no conoce.

- **Consola de terminal**: En esta área se desplegará la información de salida de UART. Por lo general, la llegada de datos a través de ZB se desplegará dentro de la consola.

Interfaz de depuración.

Cuando una aplicación que está siendo desarrollada produce errores de código no previstos, resultados fuera de lógica, lo que tiene que hacer el desarrollador es realizar una depuración para conocer las posibles causas.

Algunos IDEs como CWDS v10.2 con el XBee SDK proveen una herramienta de depuración que podría desensamblar el código fuente, ver el contenido de las variables, insertar Puntos de detención (*Breakpoint*) o ejecución paso a paso de las líneas de código.

En la Figura 5.11 se puede observar la interfaz de depuración y sus áreas importantes; por ser un tema muy extenso, sólo se comentará lo esencial.

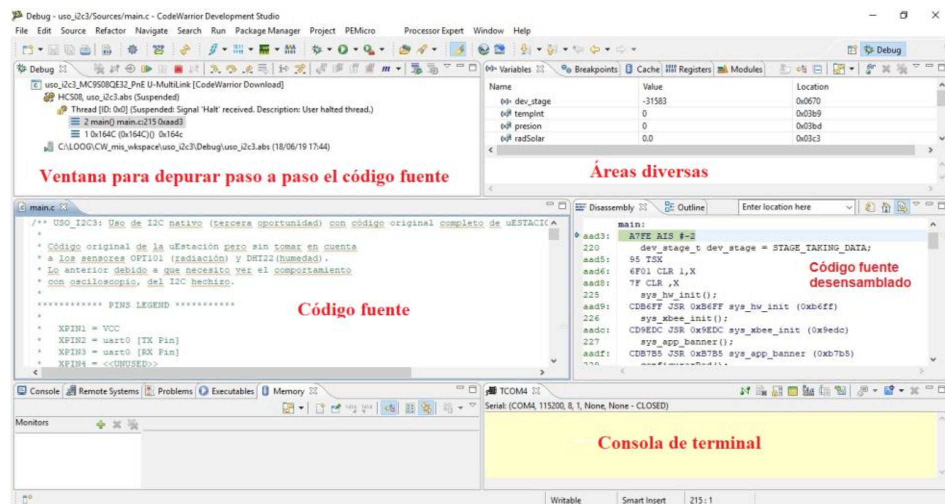


Figura 5.11: Interfaz de depuración del IDE CWDS v10.2.

Para poder ver en pantalla la interfaz de la Figura 5.11, hay que seguir los siguientes pasos:

a. Herramientas de HW necesarias para realizar la depuración de una aplicación para el *Core* XBee que se encuentre en proceso de desarrollo:
   - ✓ PC con dos puertos USB disponibles.
   - ✓ *Core* XBee.
   - ✓ U-DEV.
   - ✓ Programador P&E USB Multilink.
   - ✓ Dos cables USB para impresora.
   - ✓ Un cable JTAG tipo listón con conectores hembra en ambos extremos.

   A la lista anterior se le referirá como HW mínimo de programación.

b. Herramientas de SW necesarias:
   - ✓ Controladores de *Windows* instalados para los dispositivos USB.
   - ✓ CWDS v10.2, instalado.
   - ✓ XBee SDK instalado.

A la lista de anterior se le referirá como SW mínimo de programación.

c. Realizar conexiones de HW:

¡**Precaución**!: Mientras se esté llevando a cabo la depuración, hay que asegurarse de tener siempre energizados a los dispositivos, ya que de lo contrario podría haber daños irreversibles en el *Core* XBee.

d. Dentro del explorador de proyectos, doble click sobre el nombre del proyecto que se desea depurar.
e. Crear el ejecutable, dar click en Crear ejecutable (*Build*) -> Depurar (*Debug*).
f. Click en cargar y depurar aplicación; se tendrán 3 opciones que son:
   - **Adjuntar** (*Attach*): Seleccionar esta opción cuando el *target* real ya contiene dentro de su *flash*, la aplicación que se desea depurar. Es decir, al seleccionar esta opción se ahorrará tiempo debido a que no es necesario Transferir (*Download*) el código completo de la aplicación debido a que se encuentra dentro del *flash* del *target* real y en ejecución, por lo que solo se conecta a ella. Ver Figura 5.12.

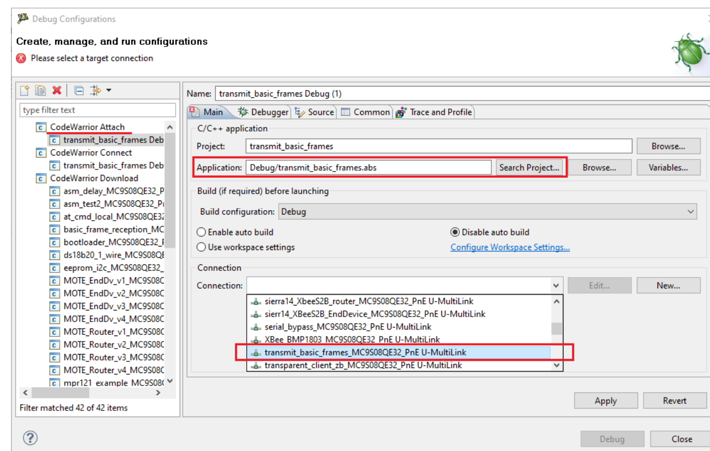


Figura 5.12: Ejemplo de configuración para una depuración con la opción de Adjuntar.

De la Figura 5.12, en el campo Aplicación se selecciona el proyecto que se encuentra dentro del *flash* del MCU programable y en el campo Conexión, se vuelve a seleccionar el nombre del proyecto junto con el HW que se empleará para la depuración.

   - **Conectar** (*Connect*): Seleccionar esta opción cuando se desee depurar una aplicación existente dentro un *target* real pero no hay información requerida para la sesión de depuración. Es decir, si dentro del explorador de proyectos, no se encuentra el proyecto pero el usuario conoce el nombre de la aplicación que se encuentra dentro de la *flash* del MCU programable, hay que seleccionarla. Con esta opción, en el campo aplicación, se selecciona cualquier proyecto del explorador de proyectos y en el campo Conexión, se selecciona el nombre del proyecto que se encuentra dentro de la *flash* del *target* real; la razón de primero seleccionar un proyecto cualquiera, es porque el IDE necesita uno para poder guardar configuraciones. Ver Figura 5.13.

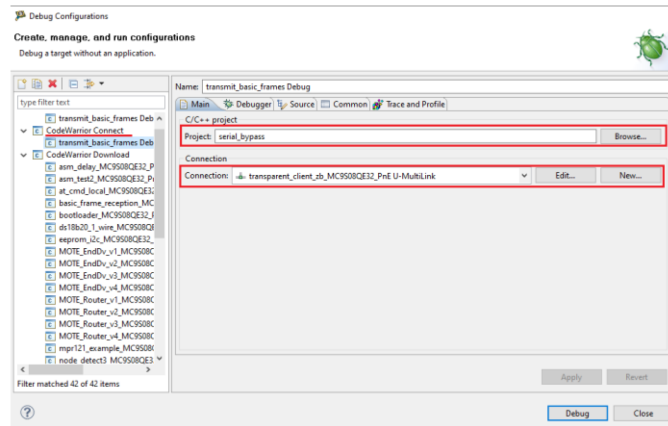


Figura 5.13: Ejemplo de configuración para una depuración con la opción de Conectar.

De la Figura 5.13, en el campo Proyecto (*Project*) se selecciona cualquier proyecto que se encuentre dentro del explorador de proyectos y en el campo Conexión se selecciona el nombre del proyecto que se sabe que se encuentra dentro de la *flash* del MCU programable junto con el HW que se empleará para la depuración.

- **Transferir** (*Download*): Seleccionar esta opción cuando al *target* real se le cargue una aplicación por primera ocasión o en caso de haber realizado cambios en alguno de los archivos del proyecto.

NOTA: Sin importar que opción se seleccione, la depuración se estaría realizando directamente con el *target* real.

g. Automáticamente se abrirá la perspectiva de depuración mostrada en la Figura 5.11.
h. Debido a lo extenso del tema, únicamente se explicarán algunos aspectos importantes del área de depuración paso a paso del código fuente (ver Figura 5.14).

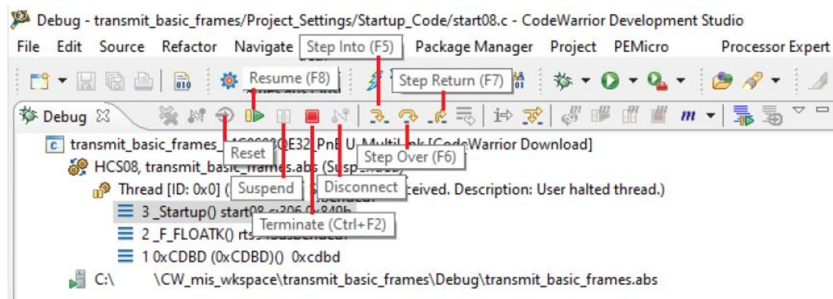


Figura 5.14: Área de depuración paso a paso de la interfaz de depuración del CWDS v10.2.

Explicación de las opciones de depuración más esenciales:

- **Reestablecer** (*Reset*): Presionar este botón cuando se haya detenido la ejecución del programa y se desea que la depuración comience desde la primera línea de código.
- **Continuar** (*Resume*): Seleccionar esa opción para continuar con la ejecución de la aplicación que se encuentra dentro del *flash* del *Core* XBee, después de haberse detenido

debido a que se encontró un punto de detención o porque el usuario presionó el botón de Pausar.

- **Pausar** (*Suspend*): Seleccionar esta opción para hacer una pausa en la ejecución de la aplicación que se encuentra dentro del *flash* del *target* Real.
- **Terminar** (*Terminate*): Cuando se selecciona esta opción, se detiene la aplicación dentro del IDE y dentro del *Target* Real. También se pueden oprimir teclas CTRL + F2 para Terminar la ejecución.
- **Desconectar** (*Disconnect*): Cuando se selecciona esta opción, se detiene la ejecución de la aplicación dentro del IDE pero continúa ejecutándose dentro del *Target* Real.
- **Entrar a** (*Step Into*): Click sobre esta opción, para ejecutar paso a paso cada línea de código de una aplicación y cuando se llegue a una función. También se puede oprimir F5 para le ejecución paso a paso.
- **Saltarse** (*Step Over*): Click sobre esta opción, para ejecutar paso a paso cada línea de código de una aplicación y cuando llegue a una función, ignorarla pasando a la línea de código siguiente a la función. También se puede oprimir F6 para realizar la misma acción.
- **Regresar un paso** (*Step Return*): Click sobre esta opción, para regresar un paso hacia atrás a la línea de comando en la que se encuentre la ejecución actual de la aplicación. También se puede oprimir F7 para realizar la misma acción.

i. Insertar puntos de detención en la ejecución

En una depuración con CWDS v10.2, cuando los programas son extensos, el seleccionar la opción "Entrar dentro de" no es muy recomendable porque habrá un retraso considerable en la ejecución total de la aplicación. En estas situaciones, se puede insertar un punto de detención de la ejecución en alguna de las líneas de código de dicha aplicación; esto es posible de dos maneras, que son:

- En alguna línea de código: dentro de la ventana de código fuente, doble click en la barra izquierda de la ventana de código fuente de lado de la línea de código en la que se desee detener la ejecución del programa.
- Antes de iniciar la ejecución del programa: Para ello, en la interfaz principal del IDE, click en Depurar (*debug*) -> Configuraciones de depuración (*Debug configurations*) -> en la parte derecha de la ventana que se despliega, seleccionar Transferir (*download*) -> seleccionar el nombre del proyecto que se desea transferir o actualizar -> click en la pestaña Depurador (*Debugger*) -> buscar Ejecución del programa (*Program execution*) -> buscar Detenerse en (*Stop on start up at*) -> seleccionar Especificado por el usuario (*user specified*) -> dentro del cuadro de texto, teclear el nombre de una función, variable o un texto poco común dentro de todo el código de la aplicación -> click en Depurar (*Debug*).

NOTA: Se pueden insertar los puntos de detención que se deseen, solo recordar que ese punto será donde se detendrá la ejecución del programa, por lo que para continuar, se puede ir paso a paso o realizar una ejecución continua (presionando F8) hasta el siguiente punto de detención.

j. Borrar punto de detención

Para borrar un punto de detención añadido, solo dar doble click sobre la ubicación donde se encuentra o irse al área de varios (ver Figura 5.11), buscar la pestaña Puntos de detención

(*Breakpoints*) y dar click en el cuadro de verificación para solo deshabilitarlo sin borrarlo o click con el botón secundario sobre él y del menú emergente, seleccionar Borrar (*Remove*).

k. Agregar o remover un punto de observación.
Cuando se realiza una depuración, conforme se ejecuta la aplicación dentro del *target* real es muy probable que se desee ver el contenido de algunas variables. En casos como este, es necesario agregar lo que se conoce como Punto de observación (*Watch point*). Sin entrar a profundidad en el tema, a continuación, se darán los pasos necesarios para agregar un punto de observación para cierta variable de interés:
   a. Iniciar la depuración de la aplicación deseada.
   b. Dentro de la interfaz de depuración, ubicar *áreas diversas* (ver Figura 5.11) y seleccionar la pestaña Variables.
   c. Conforme se vaya ejecutando la aplicación, se irán desplegando algunas de las variables del código.
   d. Pasar a la pestaña Puntos de detención y verificar.
   e. Observar cómo van cambiando los valores de la variable deseada.

Consola de terminal del IDE.

Esta es un área muy importante ya que aquí se verán los resultados de ejecución de la entrada de información a través de ZB. La información que se procesa internamente dentro del MCU programable o se recibe inalámbricamente, se transfiere del *Core* XBee hacia la consola a través de UART. A continuación, se darán algunos puntos importantes que hay que conocer sobre esta consola.

1. Opciones de consola.
   a. **Nueva vista de terminal** (*New terminal view*): Click en el botón para crear y configurar una conexión con un dispositivo que esté conectado a uno de los puertos USB de la PC, por telnet o por SSH (*Secure Shell*, Shell segura). Las dos últimas opciones dependen de alguna conexión a una red de datos.
   b. **Capturar datos en un archivo** (*Capture remote data to file*): Capturar todos los datos que se están recibiendo por consola dentro de un archivo de texto plano.
   c **Configuraciones de terminal** (*Terminal settings*): Para ver y modificar las configuraciones de una conexión creada y seleccionada con anticipación.
   d **Limpiar terminal** (*Clear terminal*): Cuando se despliega una gran cantidad de datos dentro la consola, habrá ocasiones que se desee limpiar su área para visualizar una nueva llegada de datos; dar click aquí para realizar la limpieza de la consola.
   f. **Conectarse** (*Connect*): Una vez que se crea una conexión, dar click aquí para conectarse al dispositivo y de esta manera poder ver los datos que se reciben a través del puerto USB o inalámbricamente por ZB.

   Para una mejor comprensión de lo anterior, ver Figura 5.15.

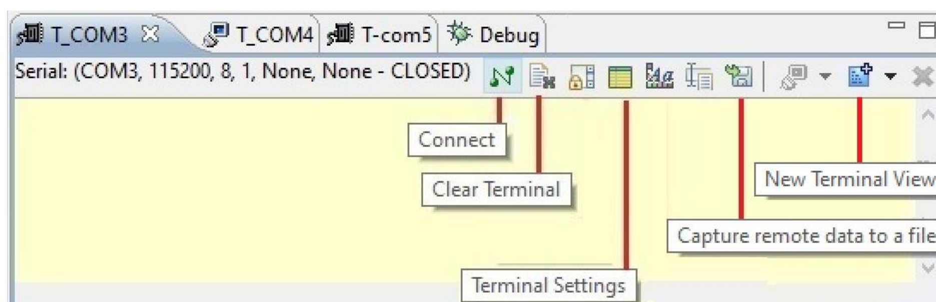


Figura 5.15: Consola de terminal del CWDS v10.2.

2 Conectarse al *Core* XBee.

Para conectarse al *Core* XBee vía puerto USB:

a. Realizar conexiones físicas.
b. Con el CWDS v10.2 abierto, en la sección de la consola de terminal, crear una nueva conexión dando click en *Nueva vista de terminal*. Se desplegará un cuadro de configuraciones (ver Figura 5.16) para la nueva vista de terminal. Las configuraciones más relevantes que siempre se deberán seleccionar cuando se conecta al *Core* XBee por USB, y son:
   - Nombre para la conexión (*View title*).
   - Tipo de conexión (*Connection type*): `Serial`
   - Puerto (*Port*): *<seleccionar puerto USB>*
   - Velocidad de comunicación (*Baud rate*): `115200`
   - Bits de datos (*Data bits*): `8`
   - Bits de parada (*Stop bits*): `1`
   - Paridad (*Parity*): Ninguna
   - Control de flujo (*Flow control*): Ninguno
   - Tiempo de espera en segundos (*Timeout, sec*): `5`
c. Click en Aceptar (*Ok*).

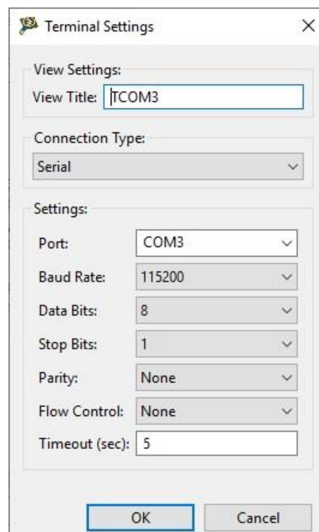


Figura 5.16: Configuraciones de conexión para la consola de terminal.

3. Capturar datos en archivo.

Conforme se reciban datos por consola de terminal, se podría desear analizarlos con alguna aplicación o realizar alguna otra acción con dicha información por lo que lo más conveniente sería guardarlos dentro de un archivo. Solo se permite texto plano. Para ello:

a. Crear un archivo .txt con un nombre corto y descriptivo
b. Dentro de la consola de terminal, click en Capturar datos en un archivo. Se pedirá la trayectoria del archivo creado en (a).
c. Conectarse al dispositivo receptor de la información que esté conectado a un puerto USB.
d. Conforme se vaya recibiendo información, se irá capturando dentro del archivo de (a).
e. Hay que terminar la captura, desconectándose del dispositivo receptor.
f. Dar click en Capturar datos en un archivo.

NOTA: Si por alguna razón se llega a detener la captura y no era lo que se deseaba, solo hay que volver a dar click en Conectarse, click en Capturar datos en un archivo, dar el nombre del mismo archivo y los nuevos datos serán añadidos sin perder o sobrescribir los primeros.

XBee SDK

En la creación de aplicaciones para el *Core* XBee hay que instalar el IDE CWDS v10.2 además del *plug-in* Digi XBee SDK ya que este contiene las herramientas necesarias para poder programar al MCU programable del *Core* XBee además de proyectos de ejemplo y un analizador XML para la creación de componentes virtuales de HW y definición de GUIs, para obtener mayor versatilidad y agilidad de implementación.

Instalación

El proceso de instalación del XBee SDK es sencillo:

a. Asegurarse de haber instalado de los controladores de dispositivos como U-DEV, X-Stick y programador.
b. Asegurarse de haber instalado el IDE CWDS v10.2.
c. Instalar el XBee SDK.

Transferir aplicación de proyecto de ejemplo.

Al instalar el XBee SDK se instalan proyectos de ejemplo que son muy útiles para poder entender el funcionamiento del *Core* XBee. Por lo anterior, una vez instalado del XBee SDK, ya es posible transferir una aplicación de ejemplo creada para el *Core* XBee. Para ello:

a. Realizar conexiones físicas.
b. Debido a que la aplicación de ejemplo que se va a utilizar, necesita dos *Core* XBee y dos U-DEV. Según el rol dentro de la WSN ZB, a los *Cores* XBee hay que transferirles el FW ZigBee *Router* API o ZigBee *end-device* API o una combinación de ambos pero no transferir el FW ZigBee *coordinator* API o cualquiera del modo AT.
c. Asegurarse que el *Core* XBee no tenga protección de seguridad. Si un *Core* XBee es nuevo no será posible utilizarlo para ejecutar alguna aplicación diseñada por algún programador debido a que traen una protección de seguridad que hay que quitar antes de poder utilizarlo. Ver más adelante para aprender cómo quitar dicha protección.
d. Dentro del CWDS v10.2, irse a Archivo (*File*) -> Nuevo (*New*) -> Proyecto XBee de Ejemplo (*XBee Sample Application Project*) -> se abrirá una ventana con el asistente para agregar un proyecto de ejemplo, si la aplicación utiliza operaciones con punto flotante, es muy importante no omitir la activación del módulo de HW que maneja el punto flotante ya que de lo contrario, se obtendrán errores con el Enlazador Inteligente (*Smart Linker*).
e. En la siguiente ventana, buscar y seleccionar wpan -> ZigBee -> *Chat* Sencillo con Identificador de Nodo (*Simple Chat with NI*).
f. Click en Finalizar (*Finish*) para que el proyecto se agregue al explorador de proyectos. Se cargará de manera automática el archivo config.xml predeterminado para el proyecto con los componentes virtuales de HW necesarios para el proyecto en particular.
g. Dentro del explorador de proyectos, click sobre el nombre del proyecto -> buscar la carpeta Archivos fuente (*Sources*) y después, doble click en main.c.
h. Click en el icono del martillo de la parte superior de la pantalla principal del IDE y seleccionar Crear ejecutable (*Build*) -> como es una aplicación de ejemplo, seleccionar Liberado (*Release*).
i. Click en Ejecutar (*Run*) -> Ejecutar como (*Run as*) -> dentro de la ventana que se despliega, seleccionar el nombre del proyecto -> para transferir la aplicación hacia el *Core* XBee, click

en Ejecutar (*Run*) -> al terminar el proceso de transferencia, oprimir el botón de *Reset* de la U-DEV para que inicie la ejecución de la aplicación dentro del *Core* XBee.

j. Después de programar a los *Core* XBee, conectar al coordinador (X-Stick).

k. Repetir los pasos para el segundo *Core* XBee y al finalizar, abrir una conexión de consola de terminal con cada uno de ellos. Al conectarse, se desplegará un prompt por cada conexión, teclear un mensaje deseado y al final, oprimir *ENTER*.

l. Para terminar, dentro de la consola de terminal, desconectarse de cada *Core* XBee y después desconectar físicamente los dispositivos.

Importar / Exportar proyecto

Una vez que se realizaron pruebas de funcionamiento de la aplicación y que resultaron exitosas, lo más conveniente y recomendable es hacer un respaldo del proyecto. Para ello:

a. Exportar (respaldar) proyecto.
   i. Con el CWDS v10.2 abierto, irse a Archivos (*File*) -> Exportar (*Export*).
   ii. De la ventana que se despliega, seleccionar Digi XBee -> Proyecto para el Core XBee (*XBee Project*) -> click en Siguiente (*Next*).
   iii. De la siguiente ventana, en la lista de proyectos, seleccionar el proyecto que se desea respaldar -> click en Siguiente (*Next*).
   iv. De la siguiente ventana que se despliega:
      - Nombre del proyecto (*Project name*): *<puede ser uno existente o uno nuevo>*
      - Versión del proyecto (*Project version*): *<especificar una versión en caso de realizar varias modificaciones al código del proyecto>*
      - Descripción del proyecto (*Project description*): *<breve descripción de la aplicación>*
      - Trayectoria de destino (*Destination*): *<Seleccionar una trayectoria destino. Se creará un archivo .zip>*
   v. Click en Finalizar (*Finish*).
   vi. Repetir los pasos anteriores para cada proyecto a exportar.

b. Importar (restaurar) proyecto:
   i. Con el CWDS v10.2 abierto, irse a Archivo (*File*) -> Importar (*Import*).
   ii. De la ventana que se despliega, seleccionar Digi XBee -> Proyecto para el *Core* XBee (*XBee Project*) -> click en Siguiente (*Next*).
   iii. De la siguiente ventana, dar la trayectoria del archivo .zip que se creó en la exportación -> click en Siguiente (*Next*).
   iv. De la siguiente ventana que se despliega, el único campo relevante es Utilizar un ID de Proyecto Distinto (*Use a Diferent Project ID*).
   v. Click en Finalizar (*Finish*).
   vi. Repetir los pasos anteriores para cada proyecto a importar.

Analizador XML del XBee SDK

Como se ha venido mencionando, entre las herramientas de programación de SW que contiene el XBee SDK esta un analizador XML que permitirá crear componentes virtuales de HW añadiendo versatilidad y agilidad a la implementación de una WSN ZB. En la presente sección se pretende ayudar al programador de aplicaciones del *Core* XBee un mejor entendimiento sobre dicho analizador para que pueda crear componentes virtuales de HW.

En la Subsección 1.2.2.4 se comentó que las entidades más importantes de un documento XML son los elementos y los *tags*.

En todo proyecto de aplicación para el *Core* XBee, al proceso de darle significado a los *tags*, ya no es necesario debido al XBee SDK para el CWDS v10.2 porque contiene un analizador de XML llamado *Smart Editor* que le da concepto a los *tags* del archivo XML, lo que permitirá realizar configuraciones esenciales de los componentes virtuales de HW y agregar algunas funciones de evento en lenguaje C de una manera gráfica. Debido a lo anterior surge la necesidad de comprender cómo utilizar esta herramienta para agilizar el proceso de desarrollo. La presente Subsección tiene como propósito entender mejor esta herramienta.
Dentro del contexto de XML para el *Smart Editor*, un componente es una representación lógica de un HW físico del *Core* XBee para aquel proyecto que lo necesite.
Toda configuración realizada para un componente con el *Smart Editor* se guarda dentro del archivo `xbee_config.h`.
El *Smart Editor* cuenta con una configuración predeterminada de HW virtual que se guarda dentro del archivo `config.xml` y que se agrega de manera automática al crear un proyecto para el *Core* XBee. Conforme se agregan componentes, este archivo cambia su contenido y tamaño. Lo anterior, aunque parece lógico, se menciona ya que es de gran utilidad al momento de crear y desarrollar aplicaciones como las del Mote de tesis de maestría, ya que se pueden personalizar según se desee.
En la Figura 5.17, se puede observar un aspecto general del *Smart Editor*.

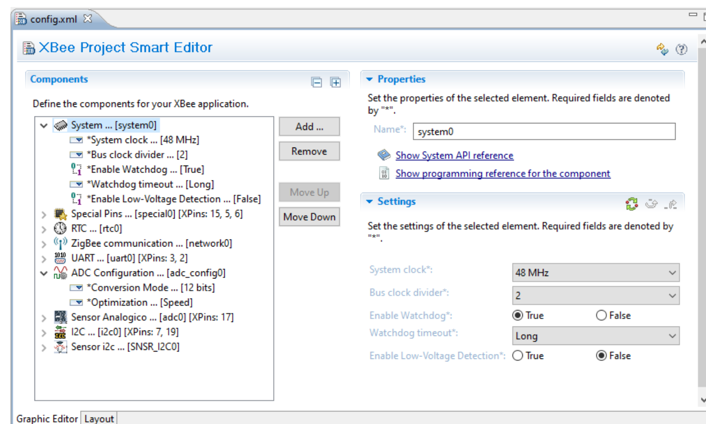


Figura 5.17: *Smart Editor* del XBee SDK para CWDS v10.2.

De la Figura 5.17, de lado izquierdo se encuentra la sección de componentes con la lista de componentes virtuales de HW disponibles, cuyo código en lenguaje C, se agregará a la aplicación que este siendo desarrollada.
De lado derecho, está el espacio para 4 secciones más. Las secciones podrían ser: Propiedades (*Properties*), Pins, Eventos (*Events*), Configuraciones (*Settings*).
De manera predeterminada, se despliegan únicamente las secciones de Propiedades y Configuraciones.
Para la creación de componentes es necesario conocer la sintaxis de los *tags* que pueden utilizarse dentro de cualquier archivo XML para el *Smart Editor* en la creación de componentes de HW, los cuales son jerárquicos y su árbol jerárquico se puede observar en la Figura 3.7.
A continuación, se da una breve descripción de la lista de *tags* disponibles y mayormente utilizados en la creación de componentes para el *Smart Editor*. Si se tiene alguna duda, ir a la Figura 3.7 para conocer el nivel en la que se encuentra dentro del árbol jerárquico.

- **Encabezado de archivo XML**. Aunque no se considera propiamente un *tag*, en todo archivo XML, es buena práctica, agregar la siguiente línea:

```
<?xml version="1.0" encoding="utf-8"?>
```

Dicha línea deberá ir siempre en la parte superior de todo archivo XML. Con la línea anterior, se le está indicando al *Smart Editor*, la versión de XML que se está utilizando y el tipo de codificación Unicode de 8 bits de resolución para los caracteres. Esto le ayuda mucho a cualquier analizador XML a interpretar fácilmente al archivo.

- **<component> </component>**:
  Es el *tag* de mayor jerarquía que permite crear un componente de HW para agregarlo a un proyecto dando click sobre el botón Agregar (*Add*, ver Figura 5.17).

  Este *tag*, va acompañado de dos argumentos de configuración que son *label* (etiqueta) y *visible*. Con *label* se configura un nombre descriptivo que identificará fácilmente al componente que está siendo creado y con *visible*, se estaría permitiendo que pueda visualizarse dicho nombre dentro de una lista de componentes disponibles para un determinado proyecto para el *Core* XBee. Ejemplo:

```
<component label="Sensor Digital" visible="true">
</component>
```

  Del ejemplo anterior, se crea un componente con etiqueta `Sensor digital` y se indica que dicha etiqueta podrá ser visible al dar click sobre el botón Agregar … (*Add*...) dentro del *Smart Editor*; la etiqueta se desplegará dentro de la lista Agregar nuevo elemento (*Add new element*).

- **<subcomponent> </subcomponent>**
  Permite crear un componente hijo dentro de un componente padre. Es el *tag* con segunda mayor jerarquía. Ejemplo:

```
<component label="Sensor Digital" visible="true">
   <subcomponent label="Sensor i2c" visible="true">
   </subcomponent>
</component>
```

  Del ejemplo anterior, dentro del componente `Sensor Digital` se configura el subcomponente `Sensor i2c`; lo anterior se puede visualizar más claramente dentro del *Smart Editor*, dando click sobre el botón Agregar ... (*Add*…), click sobre el nombre del componente y después en el subcomponente.

- **<description> </description>**
  Se utiliza cuando se desean agregar frases o párrafos como texto descriptivo. Un ejemplo de utilidad es cuando se desea agregar una breve descripción para informar al usuario final sobre el componente que desea agregar a su proyecto.

```
<description> Agregar sensor con protocolo I2C.
</description>
```

- **<physical_interface> </physical_interface>**
  Se utiliza para especificar si el componente o subcomponente contendrá alguna interfaz física como los módulos i2c, spi o los pines del *Core* XBee. En el caso de los pines se podrá seleccionar un *pin* en particular (ver más adelante como hacerlo). Los posibles valores para este *tag* son `true` o `false`. El valor `true` se utiliza cuando se desea una asignación dinámica de pines, mediante el uso de la variable `%%XPIN%%`. Ejemplo:
  `<physical_interface> true </physical_interface>`

  Con la línea anterior se está indicando que el componente que está siendo diseñado tendrá la opción de configurar pines de manera dinámica.
- **<is_unique> </is_unique>**
  Se utiliza para indicar si un componente o subcomponente podrá agregarse una sola vez (se debe configurar con valor `true`) o varias veces (valor `false`) dentro de un proyecto para el *Core* XBee. Ejemplo:
  `<is_unique> true </is_unique>`

  La línea anterior, indica que el componente que se está configurando solo se puede agregar una vez al proyecto.
- **<limit> </limit>**
  Se utiliza para limitar el número de veces que se puede agregar un componente o subcomponente a un proyecto dando click sobre el botón *Agregar*... Su valor debe ser un número natural. Ejemplo:
  `<limit> 2 </limit>`

  La línea anterior permitirá que un componente se pueda agregar al proyecto, máximo 2 veces.
- **<is_removable> </is_removable>**
  Se utiliza si se desea permitir que un componente o subcomponente pueda eliminarse de un proyecto al dar click sobre el botón Quitar (*Remove*) del *Smart Editor*. Posibles valores: `true/false`. Ejemplo:
  `<is_removable> false </is_removable>`

  La línea anterior indica que el componente no se puede quitar al dar click sobre el botón *Quitar* del *Smart Editor*.

- **<icon> </icon>**
  Se utiliza para asignar un icono a alguno de los componentes o subcomponentes, lo que permitirá identificarlo visualmente dentro de la lista disponible de componentes. El archivo de icono deberá estar en formato .png y tener un tamaño máximo de 28 x 28 pixeles. Ejemplo:
  `<icon> icons/bmp180.png </icon>`

  La línea anterior indica que existe un archivo de icono llamado bmp180.png dentro de la carpeta *icons* que está dentro de la carpeta de instalación del XBee SDK.
- **<generic_name> </generic_name>**
  Lleva como valor el nombre que se desplegará de manera predeterminada dentro del Cuadro de texto (*Textbox*) de la sección de propiedades de un componente o subcomponente del *Smart Editor*; se le añadirá de manera automática un número secuencial, comenzando en 0, que aumentará una unidad conforme se vayan agregando

componentes o subcomponentes similares, como sensores I2C y permitidos por los *tags* <limit> o <is_unique>. Ejem.:

```
 <generic_name> SNSR_I2C </generic_name>
```

La línea anterior permitirá que dentro de las propiedades del componente o subcomponente, al visualizarlo con el *Smart Editor*, se despliegue el nombre SNSR_I2C0.

- **<id> </id>**
  Se utiliza para asignar un nombre identificador al componente, subcomponente o configuración. El nombre debe de ser único dentro de la programación XML. Ejemplo: <id> snsr_spi </id>.

¡Precaución!: Evitar duplicar los ID´s de los componentes, subcomponentes o configuraciones, ya que de lo contrario, el *Smart Editor* se comportará de manera errónea al abrirlo.

- **<help_url> </help_url>**
  El XBee SDK contiene la documentación de sus funciones en formato html. Este *tag* se utiliza para crear un enlace a la documentación que corresponda a alguna función específica de un componente, por lo que el valor del argumento debe ser la trayectoria del archivo .html que la contiene. El enlace se visualizará dentro de la sección de propiedades del componente. Ejemplo:

```
<help_url> html/group__api__24xxx__eeprom.html </help_url>
```

  La línea anterior permitirá visualizar dentro de las propiedades del componente, un enlace hacia las funciones de una memoria eeprom del tipo 24xxx.

- **<reference_url> </reference_url>**
  El XBee SDK contiene la documentación de las funciones en formato html. Este *tag* es muy similar a la anterior ya que también se utiliza para crear un enlace a la documentación que corresponda a alguna función específica de un componente, pero se diferencia que la documentación será a cerca de algún protocolo de comunicación serial. Hay que especificar la trayectoria del archivo .html que la contiene. El enlace se visualizará dentro de la sección de propiedades del componente. Ejemplo:

```
<reference_url> pg/pg__one__wire.html </reference _url>
```

  La línea anterior desplegará un enlace dentro de las propiedades del componente hacia las funciones del procoms *1-Wire*.

- **<includes> </includes>**
  Por lo general, se establecen antes de cualquier *tag* <settings>. Es tag padre del *tag* <include> por lo que podría contener varios de ellos.

- **<include> </include>**
  Se utiliza cuando se desea agregar una directiva #include de lenguaje C dentro del archivo de cabecera xbee_config.h. Ejemplo:

```
<include>#include &lt;s digital.h&gt;</include>
```

  Con la línea anterior se agregará dentro del xbee_config.h la línea:

```
#include <s_digital.h>; .
```

¡**Advertencia**!: No intentar utilizar el *tag* `<include>` como *tag* hijo del *tag* `<setting>` porque podría dejar de funcionar el componente. Si se desean realizar pruebas, utilizándolo como *tag* hijo del *tag* `<setting>`, primero hay que respaldar el archivo .xml correspondiente.

- **<defines> </defines>**
  Es *tag* padre del *tag* `<define>` por lo que podría contener varios de ellos.
- **<define> </define>**
  Se utiliza por cada directiva `#define` del lenguaje C que se necesite agregar. Es *tag* hijo del *tag* `<defines>` y padre del *tag* `<defininition>` por lo que contendrá varios de estos últimos.
- **<definition> </definition>**
  Se utiliza para agregar la directiva `#define` del lenguaje C dentro del archivo de cabecera `xbee_config.h`, en otras palabras, para agregar macros. Se le puede agregar un argumento *value* como condicional. Si se desea, puede ir relacionada con el *tag* `<argument>`. Ejemplo:

```
<defines>
   <define>
     <definition value="2 Wires">#define UART_CFG_MODE_2W
 </definition>
     <argument>1</argument>
   </define>
   <define>
     <definition value="3 Wires">#define UART_CFG_MODE_3W
 </definition>
     <argument>1</argument>
   </define>
</defines>
```

  Las líneas anteriores agregarán las líneas `#define UART_CFG_MODE_2W` y `#define UART_CFG_MODE_3W` dentro del archivo `xbee_config.h` que crearán las macros mencionadas asignándole un valor de 1 (*argument*), siempre y cuando, se seleccione una de dos opciones como "2 *Wires*" o "3 *Wires*" al dar click, por ejemplo, sobre una combo.
- **<argument> </argument>**
  Es el argumento para una macro definida dentro del *tag* `<definition>`. Puede definirse un valor fijo o puede contener una variable (ver TABLA 5.1, para conocer las variables disponibles) cuyo valor podría ser asignado, p. ejemplo, al seleccionarlo desde una combo. Ejemplo:

```
<definition>#define %%NAME%%</definition>
<argument>%%RESOURCE_UPPER%%</argument>
```

  Las líneas anteriores agregarán al `xbee_config.h` la directiva `#define`, seguida del nombre de la macro (almacenado en `%%NAME%%`), el cual se teclea dentro del cuadro de texto que se encuentra dentro de las propiedades del componente dentro del *Smart Editor*; además de lo anterior, la macro podría tener como argumento cualquier valor, que en el caso del ejemplo, es el nombre de un recurso

(%%RESOURCE_UPPER%%, donde el UPPER se refiere a que el nombre se creará en mayúsculas) del *Core* XBee; un recurso podría ser un *timer* (ver TABLA 5.2, para posibles nombres de recursos), por lo que la línea que podría agregarse dentro del xbee_config.h deberá ser muy similar a la siguiente:

```
#define MiMACRO TMP1.
```

Donde TMP1 es el argumento de la macro MiMACRO; el nombre TMP1 ya está definido para un *timer* del *Core* XBee.

- **<settings> </settings>**
  Todo lo que se despliega en la sección *settings* del *Smart Editor*, corresponde a las configuraciones esenciales del componente que se está agregando. A partir de la inclusión de este *tag* dentro del archivo XML, comienza la sección de configuraciones para el componente que está siendo diseñado. Podría contener varios *tags* <setting>.
- **<setting> </setting>**
  Se utiliza para agregar diferentes opciones de configuración de algún componente que se desea agregar a un proyecto; la flexibilidad de este *tag* permite agregar elementos gráficos como combos, cuadros de texto, Botones de Radio (*Radio Button*), entre otros. Contiene el argumento *label* (etiqueta) para que el usuario final pueda identificar o conocer que es lo que va a configurar. Ejemplo:

  ```
  <setting label="Editar alarma a bajo nivel">  </setting>
  ```

  Con la línea anterior, dentro de la sección de configuraciones, se deberá desplegar una opción de configuración para la edición de una alarma a bajo nivel.

NOTA: Ocurre un caso especial en el que el *tag* <settings> podrá ser *tag* hijo de un *tag* <setting> y es cuando se utiliza el *tag* <type> con valor object (ver Figura 3.7) permitiendo una recursividad.

- **<required> </required>**
  Se utiliza para comprometer al usuario a teclear un argumento estrictamente requerido dentro de un cuadro de texto, seleccionar un valor dentro de una combo o seleccionar un estado dando click sobre un botón de radio. Si no se teclea o selecciona el argumento requerido, la configuración correspondiente se mostrará en color rojo y no permitirá avanzar hasta que se cumpla la condición. Valores posibles: true o false. Ejemplo:

  ```
  <required> false </required>
  ```

  Con la línea anterior, se está indicando que no es estrictamente necesario seleccionar o teclear el valor de alguna configuración.
- **<tooltip> </tooltip>**
  Se utiliza para orientar al usuario con un texto de ayuda, a cerca de que valores debe teclear o seleccionar en una configuración para algún componente o subcomponente. Dicho texto se visualizará al posicionar el cursor del ratón sobre la configuración con la que está relacionada. Se pueden agregar saltos de línea (\n) para agregar líneas en blanco al texto de ayuda. Ejemplo:

  ```
  <tooltip> Teclear un número del 0 al 60 para configurar el
  numero en segundos que se dormirá el Xbee </tooltip>
  ```

  La línea anterior, permitirá que al posicionar el cursor del ratón sobre la configuración relacionada, se despleigue el mensaje.
- **<type> </type>**

Toda configuración que se desee agregar para algún componente que se esté creando, puede llevar elementos gráficos para tener una interfaz amigable. Dichos elementos podrían ser cuadros de texto (para la entrada desde teclado, con valores de tipo *integer* o *string*), combo (que se utilizan para proveer varias opciones disponibles en la selección de valores), botones de selección como los botones de radio (para la selección de valores de tipo *boolean*), etiquetas informativas y agrupación de configuraciones mediante lo que se denomina objetos. Los posibles valores más utilizados para este *tag* son:

- `text`: para agregar un elemento gráfico "cuadro de texto" que solo aceptará texto.
- `integer`: para agregar un elemento gráfico "cuadro de texto" que solo aceptará números enteros.
- `String`: para agregar un elemento gráfico "cuadro de texto" que acepte caracteres alfanuméricos.
- `combo`: para agregar un elemento gráfico "cuadro de texto combinado" (*combobox*) que permita seleccionar valores disponibles de una lista de ellos.
- `boolean`: para agregar un elemento gráfico "botón de radio" que permita seleccionar un estado.
- `label`: Para ver en pantalla, el resultado de alguna operación matemática o un mensaje; es decir, no se agrega un elemento gráfico como los mencionados con anterioridad, solo una línea de texto alfanumérica e informativa.
- `object`: se utiliza cuando se desean agrupar varias configuraciones para un componente.

Ejemplo:

```
<type> String </type>
```

Con la línea anterior se agregará un cuadro de texto, que aceptará únicamente valores de tipo alfanumérico. Para restringir la entrada de caracteres no deseados, se pueden utilizar expresiones regulares como valor para el *tag* `<pattern>` (ver más adelante) con la que se relacionará este *tag*.

- **<default> </default>**

Se utiliza para configurar un valor predeterminado a un elemento gráfico como un cuadro de texto, un combo o un botón de radio. Este *tag* está relacionado con el *tag* `<type>`. Ejemplo:

```
<setting label="Activar sensor: ">
  <type>combo</type>
  <required>true</required>
  <tooltip>
    Seleccionar Activo para agregar sensor I2C a proyecto
    MOTE.
  </tooltip>
  <default>Desactivado</default>
  <id>snsr_twi</id>

</setting>
```

Con las líneas anteriores se está creando una configuración `Activar sensor` para algún componente o subcomponente que tendrá una combo con un valor predeterminado de desactivado.

- **<items> </items>**

Cuando se agrega una combo como elemento gráfico, es necesario definir los valores que podrán ser seleccionados.

- **<item> </item>**
  Se utiliza para definir los valores del elemento gráfico combo que permitirá seleccionar de una manera sencilla alguna configuración específica para un componente. Hay que agregar *tags* `<item>` por cada valor que se desee mostrar dentro de la combo. Ejemplo:

```
<setting label="UART Mode">
  <type>combo</type>
  <required>true</required>
  <default>2 Wires</default>
  <id>uart_mode</id>
  <items>
    <item>2 Wires</item>
    <item>3 Wires</item>
    <item>4 Wires</item>
  </items>
  <defines>
    <define>
      <definition         value="2         Wires">#define
UART_CFG_MODE_2W </definition>
      <argument>1</argument>
    </define>
.
</setting>
```

  De las líneas anteriores, se crea la configuración `UART Mode`, que tendrá una combo que al dar click sobre ella, se visualizarán los posibles valores a seleccionar que son: `2 Wires`, `3 Wires` o `4 Wires`, siendo el valor predeterminado `2 Wires`.

- **<xpin> </xpin>**
  Se pueden crear variables con algún nombre descriptivo para los números de *pin* del *Core* XBee. Dicha variable se puede definir con este *tag*. Cada que se agregue un *tag* `<xpin>` deberá haber un *tag* `<pin_distribution>` o `<pin_dependences>` con el que va relacionado. Ejemplo:

```
<xpin> Input Capture Pin </xpin>
```

  La línea anterior indica que `Input Capture Pin` es una variable a la que se le puede asignar de manera dinámica, algún número de *pin* disponible del *Core* XBee, especificado dentro del *tag* `<pin_distribution>` o `<pin_dependence>`.

- **<pin_distribution> </pin_distribuiton>**
  Al agregar este *tag*, se añadirá la sección *pins* dentro del *Smart Editor*. Este *tag* se utiliza cuando la configuración de algún componente necesita que se seleccione automáticamente o manualmente, el número de *pin* del *Core* XBee de una lista disponible. Se podría seleccionar uno de la lista o uno en específico (por ejemplo, el módulo i2c del *Core* XBee, necesita específicamente el *pin* 7 para transmitir datos de manera serial). Su comportamiento es muy similar al *tag* `<pin_dependence>` con la diferencia que este *tag* se utiliza cuando no se necesita establecer una condición.

- **<pin_config name = ″*<variable>*″ dynamic = ″true″> </pin_config>**
  Se utiliza para la selección dinámica de un *pin* disponible del *Core* XBee para un componente. El argumento `Dynamic = true` significa que el *pin* será seleccionado

de manera automática según la disponibilidad (sin incluir a los pines de Vcc y GND, se podrían seleccionar cualquiera de 18 pines disponibles del *Core* XBee aunque también dependerá de algunas restricciones como el caso de los pines para señales analógicas en las que solo estarían disponibles 6 pins específicos de los 18 mencionados). Ejemplo:

```
<pin_config name="Input Capture Pin" dynamic="true">
```

La línea anterior nos indica que dentro del código XML deberán existir varios *tag* `<pin>` que representan a la lista de pines del *Core* XBee que se seleccionarán de manera automática según su disponibilidad. El número de *pin* seleccionado, se asignará a la variable `Input Capture Pin`.

- **<pin> </pin>**
  Deberá haber un *tag* como este por cada *pin* del *Core* XBee que se desee asignar, por lo que sus posibles valores serán números naturales que corresponderán a un número de *pin* del *Core* XBee. Ejemplo:

```
<pin> 20 </pin>
```

  La línea anterior hace referencia al *pin* 20 del *Core* XBee.
- **<pin_dependences> </pin_dependences>**
  Al agregar este *tag*, se añadirá la sección *pins* dentro del *Smart Editor*. Este *tag* se utiliza cuando, en la configuración de algún componente, es necesario seleccionar automáticamente, pines del *Core* XBee de una lista disponible. Su comportamiento es muy similar al *tag* `<pin_distribution>` con la diferencia de que `<pin_dependences>` es más dinámico al momento de asignar números de *pin* ya que para asignar automáticamente el número de *pin*, se basa en una condición (designada en el argumento `value` del *tag* `<pin_config>`). Ejemplo:

```
<pin_dependences>
  <pin_config value="2" name="Bit0 Pin" dynamic="true">
     <pin>4</pin>
     <pin>7</pin>
     <pin>9/pin>
  </pin_config>
</pin_dependences>
```

  Para este ejemplo hay que suponer que se creó una combo con valores 2, 3 y 4. Las líneas anteriores nos indican que, una vez que el usuario final seleccione el valor indicado por el argumento `value` (la condición) del *tag* `<pin_config>`, se seleccionará un número de *pin* de una lista disponible (pines 4, 7 o 9) al que se le asignará el nombre `Bit0 Pin`.

- **<preferred> </preferred>**
  Este *tag* se utiliza cuando la configuración de algún componente necesita un *pin* específico del *Core* XBee, como por ejemplo, el módulo i2c necesita estrictamente los pines 7 y 19. El valor posible debe ser un número natural. Ejemplo:

```
<preferred> 20 </preferred>
```

  Con la línea anterior, se seleccionará de una lista disponible de pines, el pin 20 para el componente con el que se le relacione,
- **<code_snippets> </code_snippets>**

Permitirá agregar *tags* <code_snippet> con las que se podrá agregar código en C o funciones de evento en puntos específicos del código principal.

- **<code_snippet> </code_snippet>**
  Este *tag* lleva los argumentos `name` y `callback`. Con `name` se le da un nombre descriptivo a una función de evento o bloque de código y con `callback`, desplegará la sección de eventos dentro del *Smart Editor*, se agregará dentro de ella un enlace al código del evento que se encuentra en una ubicación específica dentro del archivo main.c. El *tag* <code> (ver enseguida) contendrá la función de evento o líneas de código en lenguaje C que se activarán, siempre y cuando, se active una opción dentro del *Smart Editor*. Ejemplos de funciones de eventos son las funciones para el protocolo ZB del *Core* Xbee `xbee_transparent_rx` y `node_discovery_callback`
  Ejemplo:
  ```
  <code_snippet name="IRQ Callback" callback="true">
  </code_snippet>
  ```

  La línea anterior indica que habrá un evento que hará un llamado a interrupción, desplegará la sección de eventos, contendrá el nombre `IRQ callback` que tendrá en un lado, un enlace hacia la función correspondiente dentro del `main.c` del proyecto con el que se esté trabajando.
- **<code> </code>**
  Se utiliza para insertar una función de evento o líneas de código dentro del archivo de código `main.c`, siempre y cuando, se active una opción dentro del *Smart Editor*. El código debe de ir dentro de alguna de las directivas del lenguaje C como `#if`, `#ifdef` o `#if defined`. Para insertar la función dentro del `main.c`, se toma como punto de referencia a la función `main`, por lo que, si por alguna razón no la encuentra, insertará el código donde crea necesario, lo cual no es conveniente ya que causará errores al armar el ejecutable. Va acompañada de un argumento `value` y va relacionada con los *tags* <setting> y <type>. Por ejemplo, el *tag* <type> debe contener un valor `boolean` para poder relacionarse con este *tag*. Se recomienda ubicar a este *tag* después de un *tag* <defines>.

¡Advertencia!: No utilizar este *tag* para insertar directivas de C como `#include` o `#define` porque podría borrar código del archivo principal debido a que el *tag* no está definida para dichas directivas.

Ejemplo:
```
 <setting label="Enable serial console">
   <type>boolean</type>
   <required>True</required>
   <default>True</default>
 .
   <defines>
     <define>
       <definition>#define ENABLE_STDIO_PRINTF_SCANF
       </definition>
       <argument value="True">1</argument>
       <argument value="False">0</argument>
     </define>
   </defines>
```

```
<code_snippets>
  <code_snippet     name="Data     read     callback"
callback="true">
    <code value="False">
      #if (UART_CFG_RX_WATERMARK > 0) &&
          (ENABLE_STDIO_PRINTF_SCANF == 0) &&
           defined(ENABLE_UART)
           void uart_rx_data(void) {  } #endif
    </code>
    </code_snippet>
  </code_snippets>
```

De las líneas de código anteriores, se creará una configuración llamada `Enable serial console` (Habilitar consola de terminal) para algún componente que contendrá como elemento gráfico un botón de radio cuyo valor predeterminado será `True` que habilitará el uso de la función `printf`; en caso de seleccionar `False`, se agregará el evento `uart_rx_data` antes de la función `main` del código principal.

¡Advertencia!: Si por alguna razón se modifica o borra el punto de inserción del código definido dentro de este *tag*, CWDS v10.2 podría borrar bloques del código fuente, por lo que se recomienda respaldar todo el proyecto antes de agregar un *tag* `<code>`.

- **`<dependence>  </dependence>`**
  Se utiliza para configurar si un elemento gráfico como un cuadro de texto, una etiqueta o una combo puede ser visible después de seleccionar un componente padre (referido por su id) por lo que el *tag* va acompañado de su argumento `type`. Los posibles valores para el argumento `type` son:
  - ✓ `Existence`. Verifica si el componente especificado en `<depend_element>` (ver más adelante), ha sido añadido al proyecto; en caso de que no, dará un error informando que es necesario añadirlo. Al configurar el argumento `type` con este valor, solo podrá ser hijo de los *tags* `<component>` o `<subcomponent>`.
  - ✓ `Visibility`. Permitirá que la configuración correspondiente al componente que está siendo diseñado, se visualice dentro del *Smart Editor* solo hasta que se seleccione un valor en el componente, subcomponente o configuración especificado dentro de la *tag* `<depend_element>`. Este argumento va relacionado con el *tag* `<type>` del componente o configuración especificados dentro del *tag* `<depend_element>` cuyo valor deberá ser `combo`, `text` o `integer`.
  - ✓ `Enablement`. La configuración actual, visualmente debe encontrarse dentro del *Smart Editor* deshabilitada y se habilita hasta que se seleccione `True` con un elemento gráfico como un botón de radio. Este argumento está relacionado con el *tag* `<type>` con valor `boolean` del elemento de configuración especificado dentro del *tag* `<depend_element>` cuyo valor predeterminado debe de ser `False`.
  - ✓ `Value`. El elemento de configuración actual se podrá visualizar dentro del *Smart Editor* junto con su valor actualizado después de teclear un número natural dentro de un elemento gráfico como un cuadro de texto. Este argumento está relacionado con

un *tag* `<type>` con valor `integer` del elemento de configuración especificado dentro del *tag* `<depend_element>`.

Ejemplo:
```
<dependence type=”existence”> </dependence>
```

La línea anterior, verificará si ya se agregó al proyecto un componente o subcomponente especificado en el *tag* `<depend_element>`.

- **`<depend_element>  </depend_element>`**
  Se utiliza para especificar el ID del componente, subcomponente o configuración del cual dependerá la configuración actual. Ejemplo:
  ```
  <depend_element> i2c </depend_element>
  ```

  La línea anterior nos indica que para que se visualice el componente, subcomponente o configuración dependiente tendrá primero que haberse agregado el componente, subcomponente o configuración con ID `i2c`.
- **`<depends_on>  </depends_on>`**
  Se utiliza como condicional para poder visualizar la configuración dependiente. Posibles valores son `value` si se está creando una configuración o `existence` si se está creando un componente o subcomponente. Ejemplo:
  ```
  <depends_on> value </depends_on>
  ```

  La línea anterior indica para que se visualice un elemento gráfico (como una combo o cuadro de texto) para alguna configuración, primero hay que seleccionar un valor de una componente o configuración con la que está relacionada.
- **`<expression>  </expression>`**
  Este *tag* se utiliza para especificar el valor exacto que debe seleccionarse en una configuración relacionada con el ID especificado dentro del *tag* `<depend_element>`. Va relacionada con una configuración en la que el usuario seleccionará el valor de una combo o un cuadro de texto. Este *tag* va junto con el *tag* `<new_value>`. Ejemplo
  ```
  <expression> Edge </expression>
  ```

  Con la línea anterior, se establece que hay que seleccionar o teclear el valor `Edge` dentro de la configuración con id especificado en `<depend_element>`.
- **`<new_value>  </new_value>`**
  Este *tag* se utiliza para especificar un nuevo valor para una configuración que dependerá de otro valor especificado o seleccionado dentro de un elemento gráfico como una combo o cuadro de texto. Se le pueden configurar valores como `true`, `false` u algún otro. Por lo general se le asigna el valor `true`, con lo que el elemento gráfico con el que va relacionado se despliega dentro del *Smart Editor* después de haber seleccionado el valor especificado dentro del *tag* `<expression>`. Si se le asigna un valor `false`, se invierte el comportamiento anteriormente mencionado, es decir, el elemento gráfico con el que va relacionado permanece visible mientras no se seleccione el valor especificado dentro del *tag* `<expression>`. La mayoría de las veces, este *tag* va junto con el *tag* `<expression>`. Ejemplo:
  ```
   <setting>
     <dependence type="visibility">
       <depend_element>routr_version</depend_element>
       <depends_on>value</depends_on>
  ```

```
      <expression>MOTE_Router_v4</expression>
      <new_value>true</new_value>
    </dependence>
  .
  </setting>
```

Las líneas anteriores indican que la configuración actual se desplegará hasta que en la configuración con ID `routr_version` se seleccione el valor `MOTE_Router_v4`.

¡Advertencia!: Si se omite uno de los *tag* `<expression>` o `<new_value>` cuando estos deben de ir juntos, el *Smart Editor* se comportará de manera errónea.

- **`<range>  </range>`**
  Este *tag* se utiliza para especificar rangos válidos de valores de tipo entero. Va relacionado con un elemento gráfico de cuadro de texto. Ejemplo:
  `<range> [320,28000] </range>`

  Con la línea anterior se indica que se puede teclear o seleccionar un valor que se encuentre dentro del rango que va de 320 hasta 28000.
- **`<optional_resources>    </optional_resources>`**
  Tag padre del *tag* `<resources>`.
- **`<resources>    </resources>`**
  Este *tag* se utiliza para especificar recursos del *Core* XBee como un *timer* o *pins* para el componente `SPI`. Los únicos valores que puede contener se encuentran dentro de la TABLA 5.2. Ejemplo:
  `<resources> TMP1 </resources>`

  La línea anterior se utiliza para especificar un *timer* interno del *Core* XBee que se especificó de fábrica como `TMP1`.
- **`<pattern>   </pattern>`**
  Este *tag* se utiliza para especificar los caracteres alfanuméricos válidos para un cuadro de texto de este tipo y de esta manera, asegurar que el usuario final teclee solo caracteres permitidos. Los caracteres que serán válidos se configuran mediante una expresión regular. Se recomienda acompañarlo con el *tag* `<pattern_example>`. Ejemplo:
  `<pattern> ()|\S([\da-zA-Z]{1,9}) </pattern>`

  De la línea anterior, el cuadro de texto con el que va relacionado solo aceptará quedar vacío o que el usuario final ingrese un máximo de 9 caracteres alfanuméricos.
- **`<pattern_example>   </pattern_example>`**
  Se recomienda añadir este *tag* cuando se utilice un *tag* `<pattern>`. Su valor deberá ser un texto de ayuda que permita darle una idea al usuario final de lo que tiene que teclear dentro del cuadro de texto con la que está relacionada este *tag*. El texto solo se muestra si el usuario final tecleó caracteres no permitidos por la expresión regular. Ejemplo:
  `<pattern_example> bMp180 </pattern_example>`

  Con la línea anterior, el usuario final deberá teclear algo similar a bMp180 o una combinación de ellas; esto debido a que eso se especificó dentro del *tag* `<pattern>` mediante una expresión regular.
- **`<label_link>   </label_link>`**

Este *tag* se utiliza para crear un enlace hacia la documentación que se encuentra dentro del XBee SDK. La diferencia con los *tag* `<help_url>` y `<reference_url>` antes vistas, es que se visualizará dentro de la sección de configuraciones del *Smart Editor*, como una etiqueta y no como un enlace dentro de la sección de propiedades del componente. Ejemplo:

```
<setting label="WARNING">
  <type>label</type>
  <required>false</required>
  <label_link  type=  "api">pg/pg_spi.html#secondary_s2c
  </label_link>
  <default>
    If  using  secondary  pinout  please  read  the
    documentation \n on how to configure the radio
    firmware with X-CTU.
  </default>
</setting>
```

Las líneas anteriores crean una configuración de nombre `WARNING` que contendrá una etiqueta con un enlace hacia la documentación que se encuentra dentro del XBee SDK referente a la API del procoms SPI. La etiqueta se desplegará dentro de la configuración y solo hay que dar click sobre ella para que abra la documentación correspondiente.

- **`<template>   </template>`**

  Este *tag* se utiliza para crear plantillas de pines especiales para un determinado componente. La mayoría de las veces se utiliza en *Cores* con número de pines mayor a 30 aunque se puede utilizar para una cantidad de menor a la mencionada. Ejemplo:

```
<pin_distribution>
  <template    name="Pinout    principal"    type="0"
  active="true">
    <defines>
      <define>
        <definition>#define SPI_PINOUT_MUX</definition>
        <argument>1</argument>
      </define>
    </defines>
    <pin_config name="MISO Pin" dynamic="false">
      <pin>5</pin>
    </pin_config>
    <pin_config name="MOSI Pin" dynamic="false">
      <pin>24</pin>
    </pin_config>
    <pin_config name="SPSCK Pin" dynamic="false">
      <pin>31</pin>
    </pin_config>
  </template>
  .
</pin_distribution>
```

De las líneas anteriores, se creó una plantilla con nombre `Pinout` principal para indicar los pines que se configurarán al seleccionar esta plantilla. Una segunda plantilla (no mostrada dentro del código) podría tomar otros pines distintos.

Con el *tag* anterior se terminan las breves explicaciones de uso de cada uno de ellos dentro de un archivo XML para el *Smart Editor*.

TABLA 5.1: Variables útiles para archivos XML del *Core* XBee.

`%%VALUE%%`	Almacena un valor alfanumérico proveniente de un cuadro de texto o una combo. Es una variable local de un *tag* `<setting>`. Se puede utilizar para crear macros o asignarle un valor a la macro.
`%%NAME%%`	Almacena el nombre del componente que se especifica dentro del *tag* `<generic name>` o un texto tecleado dentro del cuadro de texto de la sección propiedades de un componente. Solo se puede utilizar con un *tag* `<definition>`, no utilizarla con un *tag* `<include>`.
`%%INDEX%%`	Almacena un valor de tipo entero que comienza en 0 y aumenta en 1 unidad.
`%%RESOURCEUPPER%%`	Almacena en mayúsculas el nombre de un recurso de HW del *Core* XBee (ver TABLA 5.2). Ejem. para el *Core* XBee, el *timer* es un HW cuyo nombre por código es `TPM1`. Otro recurso podría ser un *pin* del procoms SPI. El `UPPER` indica que almacenará el nombre en mayúsculas.
`%%RESOURCELOWER%%`	También almacena en minúsculas el nombre de un recurso de HW del *Core* XBee de la TABLA 5.2 y `LOWER` es lo que indica que lo guardará en minúsculas. Ejemplo: `tmp1`.
`%%XPIN%%`	Almacena un valor de tipo entero relacionado con la asignación dinámica de números de pin disponibles en el *Core* XBee.

TABLA 5.2: Lista de recursos disponibles para el *Core* XBee.

Timer	**SPI**
TPM1	SS0
TPM2	SS1
TPM3	SS2
N/A	SS0

Crear componente virtual de HW

Como ya se mencionó, el *Smart Editor* permitirá definir GUIs para poder agregar y configurar de manera fácil y rápida, componentes virtuales de HW para cualquier proyecto para el *Core* XBee, aumentando la productividad ya que agiliza el trabajo de los desarrolladores permitiéndoles crear aplicaciones de una manera rápida, sencilla y estandarizada además de poder implementar fácil y eficientemente una WSN ZB.

Recordar que dentro del *Smart Editor*, un componente se define como una representación virtual del HW del *Core* XBee.

En la presente Subsección, se explicará con un ejemplo práctico, del uso de los *tags* anteriormente vistas.

a. Pasos generales en la creación de componentes virtuales de HW para el *Smart Editor* del CWDS v10.2:
   1. Herramientas necesarias:

- HW: una PC.
- SW: Un programa que pueda leer archivo XML (como el navegador *Firefox*) y un programa que permita crear archivos de texto plano (como *Notepad* de *Windows*).

2. Ejecutar un editor de texto plano.
3. Teclear código, basado en los *tags* aprendidos.
4. Si se utiliza *Notepad*, guardar el archivo con la extensión .xml.
5. Después de terminar el diseño y desarrollo del componente, copiar el (os) archivo (s) .xml recién creado (s), dentro algunas trayectorias específicas que se encuentran dentro de la carpeta de instalación del XBee SDK.
   La trayectoria general de instalación del XBee SDK en plataformas *Windows*, es:
   `%programfiles(x86)%\Digi\PXBee\v1.6\pxbee_api\1.6.0\`

   Lo anterior se comenta con el fin de no tener repetir la trayectoria general dentro de las trayectorias que vienen a continuación por lo que, aunque no se menciona, las carpetas que vienen en seguida, se encuentran dentro de la trayectoria general.
   - Trayectorias específicas para archivos .xml y .png
     Copiar el (os) archivo(s) .xml creado(s) por el usuario y .png seleccionado(s) para el (os) componente(s) creados, dentro de las siguientes trayectorias:
     - ✓ Archivos de Iconos (.png):
       `\.utils\xml_components\icons\`
     - ✓ Archivos .xml:
       `\.utils\xml_components\common\`
       `\.utils\xml_components\s2b\`
       `\.utils\xml_components\tht\`

       Donde:
       - ➢ **Common**: esta carpeta deberá contener aquellos archivos .xml que son comunes para cualquier proyecto de aplicación de cualquier modelo de XBee.
       - ➢ **S2b**: esta carpeta deberá contener aquellos archivos .xml que son exclusivos del *Core* XBee.
       - ➢ **Tht**: esta carpeta deberá contener aquellos archivos .xml que son exclusivos de los diferentes modelos del XBee con tecnología *through-hole*, como el *Core* XBee.
     - ✓ Proyectos de ejemplo:
       `\examples\`

   A las trayectorias general y específicas, se les llamará en conjunto trayectorias del *Smart Editor*.
6. Una vez copiados los archivos dentro de las trayectorias del *Smart Editor*, hay que comprobar que todo funciona correctamente, para ello, abrir el CWDS v10.2, crear un proyecto para el *Core* XBee. Se abrirá el *Smart Editor* con la configuración predeterminada que se encuentra dentro del archivo `config.xml`.
7. Agregar al proyecto, el componente recién creado. Dentro del *Smart Editor*, click en Agregar (*Add...*) -> se abrirá el cuadro de diálogo Agregar nuevo elemento (*Add new element*) con la lista de componentes disponibles -> seleccionar el componente recién creado -> click Agregar (*Add*).
8. En el componente recién agregado, realizar las configuraciones, que se adecuen a las necesidades del proyecto.
9. Si no se requiere agregar más componentes, click en Crear ejecutable-Liberar para posteriormente, transferir el código hacia el *Core* XBee.
10. Realizar las pruebas de funcionamiento correspondientes.

11. Repetir los pasos anteriores para nuevos componentes.

b. Ejemplo de cómo crear un componente para *Smart Editor*.
El ejemplo de creación de un componente que a continuación se explicará, es un archivo XML creado para el proyecto del presente trabajo de maestría.

1. Como primera línea, agregar aquella que es recomendable agregar en todo archivo XML.
```
<?xml version="1.0" encoding="utf-8"?>
```

2. Al componente que se va a crear se le llamará `Sensor Analogico` y es el nombre que se visualizará dentro de la lista de componentes y subcomponentes de Agregar nuevo elemento (*Add new element*) dentro del *Smart Editor*; esto se realiza con la siguiente línea de código:
```
<component label="Sensor Analogico" visible="true">
   <!-- Aquí va el código de los siguientes pasos -->
</component>
```

3. Se utilizará asignación dinámica de pines por lo que el siguiente *tag* debe llevar valor `true`:
```
<physical_interface>true</physical_interface>
```

4. El prototipo de Mote de tesis de maestría permite agregar 2 sensores analógicos como máximo por lo que para evitar limitarlo solo a uno:
```
<is_unique>false</is_unique>
```

5. Con la siguiente línea, se limita en 2 el número máximo de sensores analógicos que se pueden conectar al *Core* XBee:
```
<limit>2</limit>
```

6. Para permitir quitar al componente del proyecto dando click sobre el botón Quitar (*Remove*) del *Smart Editor*:
```
<is_removable>true</is_removable>
```

¡**Precaución**!: Tener cuidado del uso del valor `false`, porque después es complicado quitar el componente de algún proyecto en el que se agregó.

7. Para visualizar dentro del cuadro de diálogo Agregar nuevo elemento -> Descripción (*Description*), una descripción a cerca del componente que se va a agregar al proyecto:
```
<description> Agregar sensores de tipo analógico.
</description>
```

8. Para identificar visualmente al componente, asignándole un icono:
```
<icon>icons/M_Sensr_ana.png</icon>
```

9. Nombre predeterminado que se visualizará dentro del cuadro de texto que se encuentra en la sección de Propiedades del componente:
```
<generic_name>adc</generic_name>
```

10. Nombre con el que se identificará el componente dentro del código XML para *Smart Editor*:

```
<id>snsr_a</id>
```

11. Agregar a la sección de propiedades, un enlace hacia la documentación de ayuda sobre el componente o temas relacionados con él:
```
<help_url>api/group__api__adc.html</help_url>
<reference_url>pg/pg_adc.html</reference_url>
```

12. El componente que se está creando, necesita un módulo ADC del *Core* XBee por lo que depende del componente `adc_configuration`; hay que verificar si el componente del cual depende, ya fue agregado de manera automática al proyecto; esto se logra con:
```
<dependence type="existence">
  <depend_element>adc_configuration</depend_element>
  <depends_on>existence</depends_on>
</dependence>
```

13. Para agregar de manera automática dentro del `xbee_config.h`, la línea correspondiente al archivo de cabecera sobre un sensor analógico para sensar radiación como el caso del sensor OPT101 de Texas Instruments (puede ser otro sensor, solo que sea analógico y se tenga su correspondiente archivo de cabecera), es necesaria la siguiente línea de código:
```
<includes>
      <include>#include <OPT101.h> </include>
</includes>
```

14. Para agregar las macros `ENABLE_ADC_XPIN_`*<# de pin seleccionado automáticamente del Core XBee para dispositivo analógico>* y *<Nombre tecleado en el campo name de la sección de propiedades del componente>* este último con valor `XPIN_`*<# de pin del Core XBee para dispositivo analógico, seleccionado automáticamente>* dentro del archivo `xbee_config.h` y asignarles un # de *pin* seleccionado de los pines disponibles del *Core* XBee y almacenado dentro de la variable `ADC pin`, las siguientes líneas realizarán dicha acción:
```
<defines>
  <define>
    <definition>#define ENABLE_ADC_XPIN_%%XPIN%% </definition>
    <xpin>ADC Pin</xpin>
  </define>
  <define>
    <definition>#define %%NAME%%</definition>
    <argument>XPIN_%%XPIN%%</argument>
    <xpin>ADC Pin</xpin>
  </define>
</defines>
```

15. Creación de diferentes configuraciones para el componente.
```
<settings>
    <!-- Aquí va el código para todas las configuraciones del
    componente que se deseen agregar -->
</settings>
```

   a. Crear una configuración llamada Precisión de la medición que, mediante un elemento gráfico combo, permita seleccionar la calidad del muestreo para las lecturas analógicas con las siguientes opciones de configuración: calidad baja que representa 74 muestreos,

media para 150 muestreos y alta para 300 muestreos, tomando como valor predeterminado la opción de `Media`:

```
<setting label="Precision de la medicion">
   <type>combo</type>
   <required>true</required>
    <tooltip>Especificar la calidad del muestreo
    (sampling) de una señal analógica entrante.</tooltip>
   <default>Media</default>
   <id>sampling_a</id>
   <items>
     <item>Baja</item>
     <item>Media</item>
     <item>Alta</item>
   </items>
   <defines>
     <define>
       <definition>#define CTE_nLecturasADC</definition>
       <argument value="Baja">75</argument>
       <argument value="Media">150</argument>
       <argument value="Alta">300</argument>
     </define>
   </defines>
 </setting>
```

b. La segunda configuración permitirá activar o desactivar al sensor.

```
<setting label="Activar sensor: ">
  <type>combo</type>
  <required>true</required>
  <tooltip>
     Seleccionar Activo para agregar sensor analógico a
     proyecto MOTE.
   </tooltip>
   <default>Desactivo</default>
   <id>snsr_analog</id>
   <items>
     <item>Activo</item>
     <item>Desactivo</item>
   </items>
   <defines>
     <define>
 <definition value="Activo"> #define ANALOGO_EN_USO
 </definition>
      </define>
    </defines>
    </setting>
```

c. Por último, se añaden las líneas que permitirán seleccionar de manera automática uno de los pines 17 o 20 del *Core* XBee para la conexión del sensor analógico, aunque si el *pin* 20 se encuentra disponible, este es el que será seleccionado primero por lo que, si no se encuentra disponible, se seleccionará el 17. Sin importar cual *pin* se seleccione, el numero seleccionado se guardará dentro de la variable `ADC Pin`:

```
<pin_distribution>
  <pin_config name="ADC Pin" dynamic="true">
    <preferred>20</preferred>
    <pin>17</pin>
    <pin>20</pin>
  </pin_config>
</pin_distribution>
```

16. Con lo anterior, se ha creado un componente que permitirá conectar un sensor analógico en un pin específico del *Core* XBee.
17. Para asegurarse que no hay errores de sintaxis, arrastrar el archivo .xml dentro de un navegador como *Firefox*. Si no se despliegan errores, es muy probable que el *Smart Editor* lo pueda leer sin comportarse de manera errática.

Ejemplo de implementación de WSN ZB

La mejor manera de comprobar que se puede implementar una WSN ZB de manera ágil y eficiente, es implementando una, por lo que se utilizará el trabajo realizado en Mote de tesis de maestría.

En la presente Subsección, se implementará una WSN ZB de topología de estrella con un coordinador, un Mote *Router* y tres Motes *end-devices*.

Implementación de WSN ZB

1. Para la implementación de la WSN ZB, las herramientas de SW que se necesitan son:
   a. XCTU para transferir el FW para el MCU Radio correspondiente al rol que tendrá cada dispositivo dentro de la WSN ZB.
   b. CWDS v10.2 junto con el XBee SDK para cargar los códigos de aplicación de la plataforma propuesta.
   c. Las herramientas de HW necesarias son:
      - ✓ PC con dos puertos USB libres.
      - ✓ Un Digi X-Stick (coordinador).
      - ✓ Una tarjeta U- DEV.
      - ✓ Un programador P&E USB Multilink.
      - ✓ Dos cables USB para impresora.
      - ✓ Cuatro PCB de la plataforma.
      - ✓ Cuatro portapilas.
      - ✓ Doce pilas AA (tres por Mote)
      - ✓ Tres sensores TI OPT101 para la medición de radiación solar
      - ✓ Tres sensores AEOSONG DHT22 .
      - ✓ Tres sensores BOSCH BMP180.
      - ✓ Dos Protoboards.
      - ✓ Cables Dupont.
   d. Realizar las conexiones necesarias para programar los *Cores* XBee.
   e. Según el rol que cumplirán dentro de la WSN ZB, cargarles a los *Core* XBee, el FW en modo API y la aplicación según su rol dentro de la red.
   f. Una vez terminado todo lo anterior, conectar al coordinador de la WSN ZB y dentro de la consola de terminal del IDE, abrir una conexión con el coordinador.
   g. Conectar los sensores a los Motes *Router* o *end-device* y energizar a los Motes para que comiencen a funcionar cada uno de ellos dentro de la WSN ZB.

- Para mayor seguridad en la conexión de los sensores, en la Figura 3.2, se tiene el *pinmap* que es una representación gráfica de los pines del *Core* XBee en los que van conectados cada uno de ellos.
- Lo siguiente que se tiene que hacer es la conexión física de los sensores por lo que hay que verificar la Figura 5.18 donde se da un diagrama general de las conexiones.

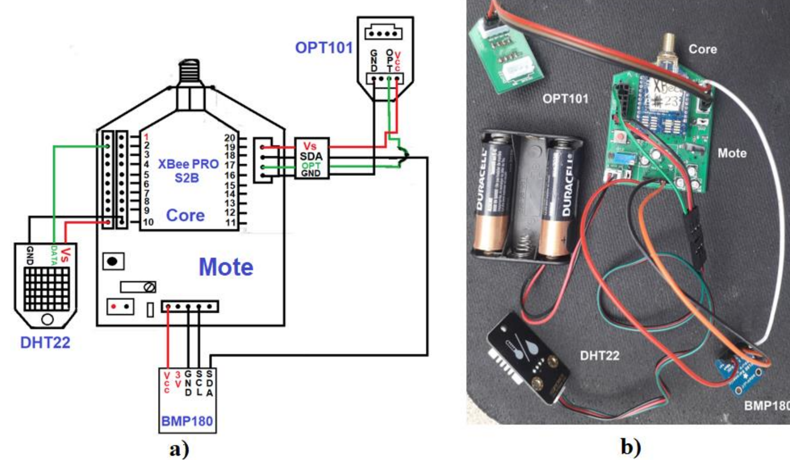


Figura 5.18: a) Diagrama general de conexiones. b) Conexión física de sensores en prototipo.

h. Después de unos minutos, el coordinador comenzará a recibir la información generada por los Motes *end-device* vía el Mote *Router* y al mismo tiempo, el Mote *Router* también enviará la información que genere. La información recibida sin formato, se podrá observar dentro de la consola de terminal del CWDS v10.2.
i. En la Figura 3.4, se puede observar la WSN ZB en funcionamiento y en la Figura 5.19, la conexión con el Coordinador por medio de la consola de terminal del CWDS v10.2 recibiendo datos provenientes de los Motes de la WSN ZB.

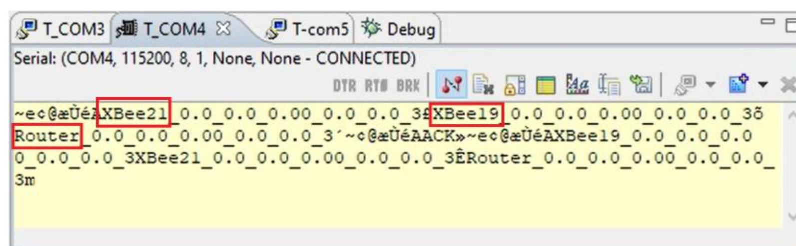


Figura 5.19: Coordinador recibiendo datos provenientes de la WSN ZB.

Hasta aquí se termina la explicación de como utilizar las herramientas de HW/SW para la creación de componentes virtuales de HW con XML e implementación de una WSN ZB de topología de estrella.

Solución de problemas

Como en toda planeación, diseño y desarrollo de un proyecto, ocurren situaciones que retrasan los planes de su finalización en los tiempos programados. En la presente Subsección, se mencionarán algunos de los problemas más comunes que podrían suceder al momento de crear componentes virtuales de HW, implementar una WSN ZB u otros, así como su posible solución para tratar de evitar cualquier posible retraso.

Core XBee nuevo

El primer detalle con el que hay que enfrentarse es con el *Core* XBee que se desea utilizar por primera vez para ejecutar alguna aplicación diseñada por el programador. Todo *Core* XBee llega de fábrica con una protección de seguridad que evita realizar depuraciones por lo que hay que eliminarla para poder utilizarlo. Para ello hay que modificar su *bootloader* de la siguiente manera:

1. Realizar las conexiones necesarias para poder programar al *Core* XBee.
2. Dentro del CWDS v10.2, cargar la aplicación de ejemplo llamada *bootloader* dentro del Explorador de Proyectos.
3. Crear el ejecutable del proyecto de (2) con la opción de Depurar.
4. Una vez creado el ejecutable con éxito, transferirlo hacia el *Core* XBee con la opción de Depuración-Descargar (*Debug-Download*).
5. Cuando el *Core* XBee es nuevo, al estar transfiriendo la aplicación, se despliega el mensaje de la Figura 5.20.

Figura 5.20: Mensaje que indica que el *bootloader* del *Core* XBee no puede ser depurado.

Click en Si (*Yes*).

6. Desconectar y volver a conectar la U-DEV de la PC para que se tomen los cambios recién realizados. Con esto, ya se puede utilizar el *Core* XBee para diferentes proyectos de aplicación.

XBIB no se detecta dentro de *Windows* 10 de 64 bits

Cuando este error ocurre es debido a que los controladores para la tarjeta U-DEV no fueron diseñados para esta plataforma. Esto se soluciona de la siguiente manera:

1. En la PC con *Windows* 10 instalado, donde se desea conectar la U-DEV, instalar los controladores de la tarjeta como lo indica el fabricante.
2. Conectar la U-DEV a un puerto USB libre de la PC.
3. Irse a Este equipo (*This device*) -> Propiedades (*Properties*) -> Sistema (*System*) -> Administrador de dispositivos (*Device manager*) -> Controladores de bus serie universal -> Convertidor Serial USB (*USB Serial Converter*) -> Propiedades (*Properties*) -> pestaña Avanzado (*Advanced*) -> click en el cuadro de verificación Cargar VCP (*Virtual COM Port*) -> click en Aceptar (*Ok*).

Fallas al querer transferir código o depurar una aplicación en el *Core* XBee

Este error (ver Figura 5.21) ocurre cuando se está programando el *Core* XBee o se está depurando una aplicación dentro de él. La causa del error es que no existe comunicación con el programador debido a que:

- No se conectó el programador.

- Cable JTAG mal conectado en ambos extremos.
- No se instaló correctamente el CWDS v10.2 ya que contiene sus controladores del programador.
- Esta dañado el programador.
- Cable USB para impresora, dañado.
- Cable JTAG, dañado.

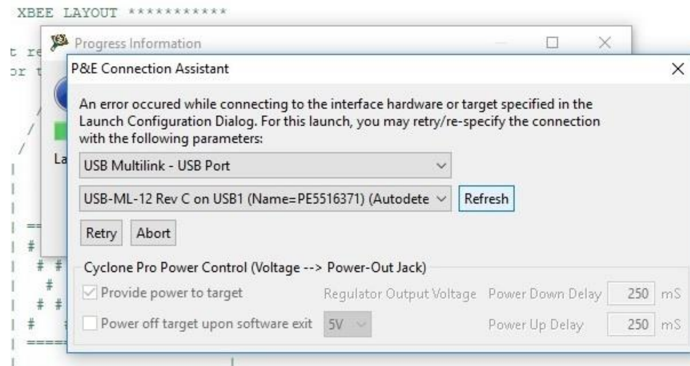


Figura 5.21: Error de comunicación entre el programador y la PC.

Las dos primeras causas son las más frecuentes por lo que solo hay que verificar que todo este correcto.

Enlazador inteligente del IDE envía el mensaje de error *L1822: Symbol ... in file* al crear el ejecutable

Este error es debido a que no se habilitó el módulo de HW y al crear el ejecutable, el Enlazador Inteligente (*Smart Linker*) marca el error L1822: símbolo … dentro del archivo …. no fue definido (*L1822: symbol* <símbolo de la tabla de símbolos> *in File ... is undefined*). Una solución es volver a cargar el proyecto y cuando se solicite, activar al módulo para el manejo de número de punto flotante. La segunda solución es activar el módulo dentro del proyecto, con tan solo cambiar un archivo de biblioteca de la siguiente manera:

1. Click con el botón secundario del ratón sobre el nombre del proyecto.
2. Del menú contextual, seleccionar Propiedades (*Properties*).
3. Se desplegará el cuadro de diálogo Propiedades para (*Properties for*) <nombre del proyecto seleccionado en el punto (1)>.
4. Del árbol jerárquico que aparece en la parte izquierda del cuadro, seleccionar Crear ejecutable C/C++ (*C/C++ Build*) -> Configuraciones (*Settings*) -> pestaña Configuraciones de herramienta (*Tool settings*) -> Enlazador para el S08 (*S08 Linker*) -> Entrada (*Input*) -> Bibliotecas (*Libraries*) -> Editar trayectoria de archivo (*Edit file path*) -> click en Sistema de Archivos (*File System*) -> seleccionar la trayectoria `${MCUToolsBaseDir}` que es la trayectoria en donde se instaló el CWDS v10.2; cambiar la biblioteca `ansiis.lib` por la biblioteca `ansifs.lib`.
5. Click en Aceptar (*Ok*) y por último, crear el ejecutable para el proyecto.

Error de comunicación entre CWDS v10.2 y el *Core* XBee

Al querer transferir un proyecto de aplicación desde la PC hacia el *Core* XBee, se podría obtener el mensaje de error de la Figura 5.22 que podría ser debido a diversas causas como:

- *Core* XBee dañado.
- Cables USB y/o JTAG, dañados.
- P&E USB Multilink, dañado.

- U-DEV, dañada.
- Falso contacto en las conexiones.
- La falla más probable en la mayoría de las ocasiones es la comunicación entre la aplicación y el MCU programable del *Core* XBee, debido al uso continuo del *Core* XBee.

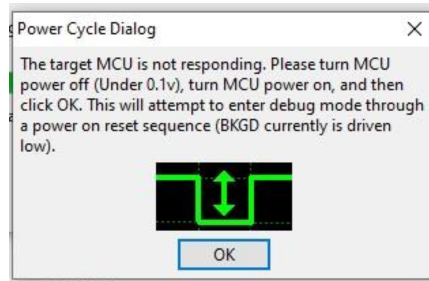


Figura 5.22: Falla en la comunicación entre la PC y el MCU programable del *Core* XBee.

Las posibles soluciones son:

- Tener al alcance un HW nuevo o asegurarse que todo el HW que se está utilizando funciona correctamente.
- Pruebas de continuidad en los cables.
- Volver a realizar todas las conexiones.
- Para la última opción de posible falla, dar click en el botón de Aceptar (*ok*) de la Figura 5.22 y casi de inmediato que desaparece la imagen de pantalla, click en el botón de reconfigurar (*reset*) de la U-DEV, Este último paso es complicado de realizar al principio pero con el tiempo se aprende a realizarlo.

Mote *end-device* enciende pero se apaga de inmediato

Todo dispositivo con este rol dentro de la WSN ZB, debe apagarse para ahorrar energía por lo que por código está configurado para sense el estado de las baterías y esto se realiza utilizando físicamente un *pin* específico del *Core* XBee como entrada analógica. Entre las optimizaciones realizadas al código original del Mote de tesis de maestría, se cambió por código, el número de *pin* pero físicamente no es posible hacer el cambio porque se requiere un rediseño de la placa prototipo. Una solución temporal a lo anterior es configurar dentro del archivo de código `M_Mote_energia.c` el valor de `return` de 0 a 3.

Cuando se tenga la nueva placa prototipo, por código, solo hay que volver a regresar el valor antes mencionado (de 3 a 0).

Smart Editor no muestra el componente virtual de HW creado por el programador

Este error se presenta cuando existe un error dentro del archivo .xml creado por el programador y que corresponde al componente. Todo archivo .xml puede ser leído por un navegador como *Firefox*, solo hay abrir o arrastrar el archivo, según corresponda. Si se utiliza *Firefox* para leer el archivo, este mostrará la línea de código donde se podría encontrar el error (ver Figura 5.23).

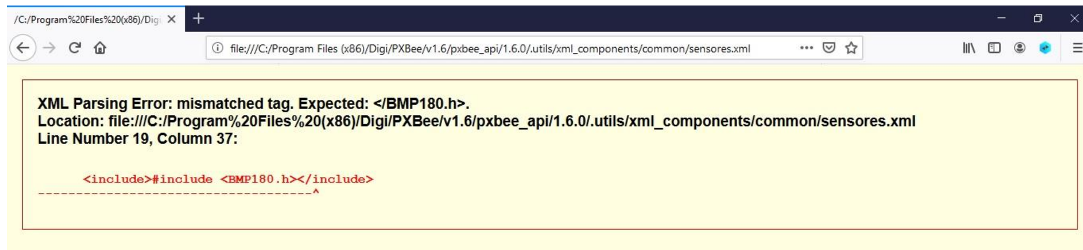


Figura 5.23: Error de lectura de un archivo .xml con el navegador *Firefox*.

La solución es corregir el error, guardar los cambios, arrastrar el archivo el navegador y si se lee correctamente, actualizar el archivo correspondiente dentro de las carpetas que lee el *Smart Editor* (trayectoria general).

Comportamiento errático del *Smart Editor*

La mayoría de las veces este problema se debe a que se ha utilizado de manera incorrecta uno de los *tag* vistos con anterioridad, dentro de uno de los archivos .xml creados por el programador o porque se realizaron modificaciones a alguno de ellos pero no se actualizaron los archivos que lee el *Smart Editor*.

Corregir el error o actualizar los archivos que sufran cambios.

XCTU falla al leer las propiedades del *Core* XBee

Cuando se intenta agregar a un *Core* XBee dentro del XCTU, se leen sus propiedades actuales. Si al leer dichas propiedades se despliega un mensaje de error como el de la Figura 5.24 es debido a las siguientes causas posibles:

- Falsos contactos.
- U-DEV, dañada.
- Cables dañados.
- *Core* XBee dañado o apunto de fallar (el más común).

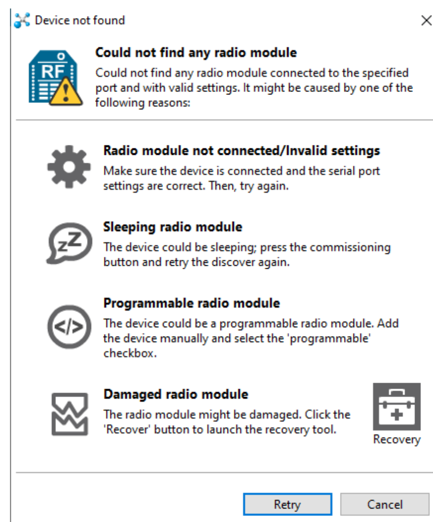


Figura 5.24: Error común al intentar leer las propiedades del MCU Radio con el XCTU.

Si se obtiene el error de la Figura 5.24, es probable que se haya dañado el *Core* XBee aunque habrá ocasiones en las que se puedan leer las propiedades después de algunos intentos o también podría ser debido a daño en la U-DEV.

En algunas ocasiones podría funcionar cargar un FW distinto para el MCU Radio al que tenga actualmente.

De no funcionar con nada de lo anteriormente sugerido, remplazar el *Core* XBee.

Dispositivos no se conectan a la WSN ZB

La razón más común por la cual no se conectan los diferentes dispositivos a la WSN ZB son debido a que se configuró un número de PAN ID distinto en cada uno de los dispositivos de la WSN ZB y/o el número de Canal de Operación (*Operating Channel*) no es el mismo entre ellos. La solución al primero de ellos es sencilla ya que solo hay que configurar el mismo PAN ID para todos los dispositivos con el *Smart Editor* o con el XCTU. Si se desean crear dos o más WSN ZB totalmente independientes, no olvidar que por cada WSN ZB creada, hay que configurar un PAN ID distinto.

El segundo lo asigna de manera automática el coordinador de la WSN ZB por lo que si no se está actualizando el argumento de manera automática, podría ser debido al FW del MCU Radio. Probar cambiando el FW del radio.

Los anteriores son algunos de los tantos problemas que se podrían presentar al momento de programar *Cores* XBee, o implementar una WSN ZB. Algunos errores son muy descriptivos que facilitan su solución, pero los mencionados en la presente sección son los que más se presentan y pueden no ser tan intuitivos.

Con lo anterior, se finaliza la descripción de los detalles técnicos de la plataforma propuesta.

Referencias

[1] Raghavan, V., & Shahnasser, H. (2015). Embedded wireless sensor network for environment monitoring. *Journal of Advances in Computer Networks*, 3(1), 13-17.

[2] El Kouche, A. (2012, June). Towards a wireless sensor network platform for the Internet of Things: Sprouts WSN platform. In *2012 IEEE International Conference on Communications (ICC)* (pp. 632-636). IEEE.

[3] Mujica, G., Rosello, V., Portilla, J., & Riesgo, T. (2012, October). Hardware-software integration platform for a WSN testbed based on cookies nodes. In *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society* (pp. 6013-6018). IEEE.

[4] Yawut, C., & Kilaso, S. (2011, May). A wireless sensor network for weather and disaster alarm systems. In *International Conference on Information and Electronics Engineering, IPCSIT* (Vol. 6, pp. 155-159).

[5] Li, Y., Wang, Z., & Song, Y. (2006, June). Wireless sensor network design for wildfire monitoring. In *2006 6th World Congress on Intelligent Control and Automation* (Vol. 1, pp. 109-113). IEEE.

[6] Tovar, A. (2017). Sistema Embebido Inalámbrico para la Monitorización de Radiación Solar. *Tesis de Ing. en Computación. Universidad Autónoma de San Luis Potosí.*

[7] López, A. (2018). Diseño de una micro - estación meteorológica capaz de generar una red de sensores inalámbrica. *Tesis para obtener el grado de Ingeniero en computación. Universidad Autónoma de San Luis Potosí.*

[8] Salvatore Filippo Di Gennaro, Alessandro Matese, Beniamino Gioli, Piero Toscano, Alessandro Zaldei, Alberto Palliotti, and Lorenzo Genesio (2017). Multisensor approach to assess vineyard thermal dynamics combining high-resolution unmanned aerial vehicle (uav) remote sensing and wireless sensor network (wsn) proximal sensing. *Scientia horticulturae*, 221:83{87.

[9] Spachos, P., & Gregori, S. (2019). Integration of wireless sensor networks and smart uavs for precision viticulture. *IEEE Internet Computing*, 23(3), 8-16.

[10] Lifländer, J., & Oy, P. F. (2010). Ceramic chip antennas vs. PCB trace antennas: a comparison. *Microwave product digest*, 2.

[11] Divsalar, D., Simon, M. K., & Raphaeli, D. (1998). Improved parallel interference cancellation for CDMA. *IEEE Transactions on Communications*, *46*(2), 258-268.

[12] Darroudi, S. M., & Gomez, C. (2017). Bluetooth low energy mesh networks: A survey. *Sensors*, *17*(7), 1467.

[13] Redes de malla con bluetooth 5 (consultado el 15/ene/2022), https://www.digikey.com.mx/es/articles/take-the-fast-track-to-bluetooth-5-mesh-networking

[14] Bluetooth mesh & home automation (consultado el 15/ene/2022), https://www.mouser.hn/blog/bluetooth-mesh-home-automation

[15] Faludi, R. (2010). *Building wireless sensor networks: with ZigBee, XBee, arduino, and processing*. " O'Reilly Media, Inc."

[16] Cordeiro, L., Barreto, R., Barcelos, R., Oliveira, M., Lucena, V., & Maciel, P. (2007, March). Agile development methodology for embedded systems: A platform-based design approach. In *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)* (pp. 195-202). IEEE.

[17] Greene, B. (June, 2004). Agile methods applied to embedded firmware development. In *Agile Development Conference* (ADC'04).

[18] Crespi, V., Galstyan, A., & Lerman, K. (2008). Top-down vs bottom-up methodologies in multi-agent system design. *Autonomous Robots*, 24(3), 303-313.

[19] Teich, J. (2012). Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, *100*(Special Centennial Issue), 1411-1430.

[20] Liggesmeyer, P., & Trapp, M. (2009). Trends in embedded software engineering. *IEEE software*, *26*(3), 19-25.

[21] Oshana, R., & Kraeling, M. (Eds.). (2019). *Software engineering for embedded systems: Methods, practical techniques, and applications*. Newnes.

[22] Staunstrup, J., & Wolf, W. (1997). *Hardware/software co-design: principles and practice*. Springer Science & Business Media.

[23] Gazis, V., Houssos, N., Koutsopoulou, M., Pantazis, S., & Alonistioti, N. (2003, September). Towards Reconfigurable 4G Mobile Environments. In *ANWIRE Workshop on Reconfigurability, Mykonos, Greece*.

[24] De Michell, G., & Gupta, R. K. (1997). Hardware/software co-design. *Proceedings of the IEEE*, *85*(3), 349-365.

[25] Ishtiaq, A., Khan, M. U., Ali, S. Z., Habib, K., Samer, S., & Hafeez, E. (2021, January). A Review of System on Chip (SOC) Applications in Internet of Things (IOT) and Medical. In *ICAME21, International Conference on Advances in Mechanical Engineering, Pakistan* (pp. 1-10).

[26] Azevedo, J., Pereira, R. L., & Chainho, P. (2015, April). An API proposal for integrating sensor data into web apps and WebRTC. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems* (pp. 1-5).

[27] Yoo, S., & Jerraya, A. A. (2003). Introduction to hardware abstraction layers for SoC. In *Embedded software for SoC* (pp. 179-186). Springer, Boston, MA.

[28] Frantz, R. Z., & Corchuelo, R. (2012, March). A software development kit to implement integration solutions. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (pp. 1647-1652).

[29] Shriram Krishnamurthi, Kathryn E Gray, and Paul T Graunke (2000, January). Transformation-by-example for xml. In *International Symposium on Practical Aspects of Declarative Languages*, pages 249{262. Springer, Berlin, Heidelberg.

[30] Terence Ahern, Mark Jamison, and Arturo Olivarez (2000). Designing the next generation of tools using an open systems approach: A usability study. In *Society for Information Technology & Teacher Education International Conference*, pages 2390{2395. Association for the Advancement of Computing in Education (AACE).

[31] Scott A McDermott (2006, March). Astrologic: using xml in a spacecraft-focused client server system. In *2006 IEEE Aerospace Conference*, pages 15{pp. IEEE.

[32] Marchal, B. (2002). *XML by Example*. Que Publishing.

[33] Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. IBM *Systems Journal*, *41*(1), 4-12.

[34] Wolf, W. H. (1994). Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, *82*(7), 967-989.

[35] Achleitner, S., Kamthe, A., Liu, T., & Cerpa, A. E. (2014, April). SIPS: Solar irradiance prediction system. In *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks* (pp. 225-236). IEEE.

[36] Garcia-Font, V., Garrigues, C., & Rifà-Pous, H. (2016). A comparative study of anomaly detection techniques for smart city wireless sensor networks. *sensors*, *16*(6), 868

[37] Salam, H. A., & Khan, B. M. (2016). Use of wireless system in healthcare for developing countries. *Digital Communications and Networks*, *2*(1), 35-46

[38] Smart Linker (consultado el 15/ene/2022), https://www.nxp.com/docs/en/user-guide/SMARTLINKERUG.pdf

[39] Handheld LCR Meter, User´s Manual (consultado el 10/nov/2022), http://www.hantek.com/Product/Hantek183X/Hantek183X%20Series%20Manual_EN.pd

[40] BK Precision 2510 Series Handheld, User Manual (consultado el 10/nov/2022), https://bkpmedia.s3.amazonaws.com/downloads/manuals/en-us/251x_manual.pdf

[41] Alsolai, H., & Roper, M. (2020). A systematic literature review of machine learning techniques for software maintainability prediction. *Information and Software Technology*, *119*, 106214