UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN

Diseño de una interfaz para sistemas de control en Java con comunicación inalámbrica utilizando un microcontrolador AVR con interfaz USB

TESIS

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO MECÁNICO ELECTRICISTA

PRESENTA:

Rubén Flores Altamirano

ASESOR:

José Luis Barbosa Pacheco

CUAUTITLÁN IZCALLI, ESTADO DE MÉXICO

2014

VNIVERSIDAD NACIONAL AVPNOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN
UNIDAD DE ADMINISTRACIÓN ESCOLAR
DEPARTAMENTO DE EXÁMENES PROFESIONALES

ASUNTO: VOTO APROBATORIO

M. en C. JORGE ALFREDO CUÉLLAR ORDAZ
DIRECTOR DE LA FES CUAUTITLAN
PRESENTE

ATN: M. en A. ISMAEL HERNÁNDEZ MAURICIO
Jefe del Departamento de Exámenes
Profesionales de la FES Cuautitlán.

Con base en el Reglamento General de Exámenes, y la Dirección de la Facultad, nos permitimos comunicar a usted que revisamos **La Tesis:**

"DISEÑO DE UNA INTERFAZ PARA SISTEMAS DE CONTROL EN JAVA CON COMUNICACIÓN INALÁMBRICA UTILIZANDO UN MICROCONTROLADOR AVR CON INTERFAZ USB"

Que presenta el pasante: **RUBÉN FLORES ALTAMIRANO**
Con número de cuenta: **40900386-6** para obtener el Título de: **Ingeniero Mecánico Electricista**

Considerando que dicho trabajo reúne los requisitos necesarios para ser discutido en el **EXAMEN PROFESIONAL** correspondiente, otorgamos nuestro **VOTO APROBATORIO**.

ATENTAMENTE
"POR MI RAZA HABLARA EL ESPÍRITU"
Cuautitlán Izcalli, Méx. a 04 de Agosto de 2014.

PROFESORES QUE INTEGRAN EL JURADO

	NOMBRE	FIRMA
PRESIDENTE	Ing. José Luis Rivera López	
VOCAL	Ing. José Luis Barbosa Pacheco	
SECRETARIO	Ing. Oscar Carmona Islas	
1er SUPLENTE	Ing. Anatolio Mendoza González	
2do SUPLENTE	Ing. José Alberto Song Flores	

NOTA: Los sinodales suplentes están obligados a presentarse el día y hora del Examen Profesional (art. 127).
En caso de que algún miembro del jurado no pueda asistir al examen profesional deberá dar aviso por anticipado al departamento. (Art 127 REP)
HHA/Vc

"Cuando un científico distinguido pero de edad avanzada afirma que algo es posible, casi con toda seguridad está en lo cierto. Cuando afirma que algo es imposible, muy probablemente se equivoca. La única manera de descubrir los límites de lo posible es aventurarse un poco más allá de dichos límites, en lo imposible. Toda tecnología lo suficientemente avanzada es indistinguible de la magia."

Arthur C. Clarke.

Agradecimientos

A mis padres y hermanos

Por su incondicional apoyo y afecto, este trabajo es el resultado de la suma de todos sus esfuerzos y su amor brindado a lo largo de toda mi vida, no existen suficientes palabras para agradecerles todo, los amo con todo mi ser. Gracias Claudia, mi ángel que me cuida desde el cielo.

A mis amigos

Por esos momentos increíbles en la Universidad, su afecto y su gran amistad.

A mi novia

Por toda su ayuda y amor durante más de 5 años, gracias por compartir conmigo este logro y muchos más que nos esperan, te amo.

A mis maestros

Por toda su labor realizada para mi desarrollo profesional. En especial a mi asesor José Luis Barbosa Pacheco por su amistad, paciencia y dedicación para la realización de este trabajo.

A la Facultad de Estudios Superiores Cuautitlán Campo 4 y la Universidad Nacional Autónoma de México

Por brindarme una educación gratuita y los conocimientos necesarios para lograr convertirme en un Ingeniero Mecánico Electricista. *"Por mi raza hablará el espíritu"*.

Contenido

Capítulo 1
Introducción

En este capítulo se describe el objetivo de este proyecto, su planteamiento y las bases del mismo.

Actualmente existe una gran necesidad del uso de sistemas de control y automatización en la industria y en áreas de investigación y desarrollo, en donde estos sistemas, utilizan dispositivos que se encargan de la supervisión, control y/o adquisición de datos de sus diversos procesos involucrados. Esto se realiza a través de sensores y actuadores, los cuales permiten un monitoreo constante y pueden realizar acciones al equipo, la información es recibida por un dispositivo lógico de control, éste se encarga de gestionar a los sensores y actuadores, lo que permite realizar acciones preventivas para evitar accidentes o daños en el equipo y regular los componentes involucrados en el proceso para su correcto funcionamiento.

Estos sistemas de control pueden trabajar independientemente o pueden enviar toda la información recabada hacia la computadora, para que a través de un software específico se puedan supervisar todos los procesos a distancia, este tipo de software es comúnmente llamado SCADA (Supervisory Control and Data Acquisition) y tiene la particularidad de proveer de toda la información que se genera en un proceso productivo y permite su gestión e intervención. Se caracterizan por ser sistemas de lazo cerrado en dónde existe una retroalimentación con el sistema, es decir, la salida regresa al principio para que se analice la diferencia junto con un valor de referencia y se ajuste el sistema para reducir el error.

Este tipo de sistemas SCADA son muy deseables y permiten una mayor facilidad para el usuario al controlar sus procesos, de esta manera es un factor indispensable tener una interfaz con la computadora que permita comunicarse con el sistema de control. La comunicación entre el sistema de control y el software SCADA puede implementarse de forma alámbrica o inalámbrica, siendo en muchas ocasiones la inalámbrica la más apropiada ya que evita cableado, posibles fallos eléctricos y reduce los costos.

La comunicación inalámbrica se logra mediante envío de datos a través de ondas electromagnéticas las cuales se envían por un dispositivo transmisor y son captadas por un receptor, de esta manera se logra enviar y recibir información a distancias cortas o medianas sin necesidad de cables directos entre el transmisor y el receptor, sin embargo, la comunicación directa con la computadora suele ser al final mediante un puerto físico de la misma como, lo es el puerto Ethernet o el estándar USB, también existe la posibilidad de protocolos inalámbricos como el bluetooth o el wifi, sin embargo, no necesariamente se encuentran en la mayoría de las computadoras y para el caso del wifi se requiere de un punto de acceso o router que asigne las direcciones adecuadas al dispositivo para poder lograr dicha comunicación. De las opciones mencionadas, el puerto USB sigue siendo el estándar para la mayoría de las computadoras actuales. El protocolo USB es confiable, rápido, versátil, permite un ahorro de energía, es barato y es soportado por la mayoría de los sistemas operativos.

El Universal Serial Bus (USB) es un estándar industrial desarrollado a mediados de los años 1990 que define los cables, conectores y protocolos usados en un bus para conectar, comunicar y proveer de alimentación eléctrica entre ordenadores, periféricos y dispositivos electrónicos [1].

Desde 2004, aproximadamente 6 mil millones de dispositivos se encuentran actualmente en el mercado global, y alrededor de 2 mil millones se venden cada año.[1]

[1] Fuente: http://www.pcworld.com/article/156494/superspeed_usb.html

En la actualidad se ha vuelto un estándar y su campo de aplicación se extiende a cualquier dispositivo electrónico como computadoras, dispositivos portátiles (laptops, celulares, tabletas, etc.). Los dispositivos que cuentan con USB van desde mouse, teclados, controles, escáneres, cámaras, impresoras, entre otros. [2]

Adicionalmente se debe considerar la gestión y el manejo del flujo de los datos hacia la computadora, esta parte consiste en la recepción de los datos inalámbricos para posteriormente enviarlos a través del protocolo USB. Ésto se logra mediante un dispositivo de control que se encargue de dichas funciones, tal como un microcontrolador o un microprocesador, debido a que la gestión de los datos no demanda mucho procesamiento, un microprocesador excede las necesidades y además incrementa el costo, por lo que, un microcontrolador es la opción óptima para éste propósito.

Los microcontroladores permiten ejecutar tareas específicas dentro de un sistema y cuentan con una unidad aritmética lógica, memoria y puertos de entrada y de salida, integrados en el mismo chip, lo que permite reducir costos al no incluir hardware externo para su funcionamiento y evitar fallas debidas al cableado externo, a diferencia de un microprocesador.

Los microcontroladores han mejorado desde su invención, actualmente cuentan con una gran variedad de características adicionales, desde convertidores analógicos digitales hasta protocolos de comunicación integrados, permiten su fácil programación y cuentan con herramientas de desarrollo en distintos lenguajes como C, Basic, ensamblador, etc. En la actualidad existen diversas marcas especializadas en la manufactura y venta de microcontroladores tales como Microchip®, ATMEL®, NXP®, Freescale®, Texas Instruments®, entre otras. Todas ofrecen distintas opciones en cuanto a características, puertos, memoria, número de bits, etc., permitiendo una flexibilidad al momento de escoger qué microcontrolador utilizar para alguna aplicación específica.

Entre los más reconocidos, se encuentran los microcontroladores PIC de Microchip® y los AVR de ATMEL®, su familia de microcontroladores de 8 bits es extensa y cuenta con herramientas de desarrollo gratuitas, las diferencias más significativas existen en su procesamiento, en donde, los microcontroladores AVR realizan una instrucción por cada ciclo de reloj, mientras que los PIC necesitan 4 ciclos de reloj para realizar una sola instrucción, debido a esto los PIC suelen ser más lentos, por ejemplo, si se utiliza un reloj con una frecuencia de oscilación de 4MHz, el AVR logra 4 millones de instrucciones por segundo mientras que el PIC sólo lograría 1 millón de instrucciones por segundo, adicionalmente las instrucciones de los AVR son más completas ya que permiten realizar operaciones más complejas que los PIC y en una sola instrucción. Otra característica importante en los AVR es su soporte directo con el lenguaje C, siendo de gran ayuda para desarrollar aplicaciones con mayor facilidad y sin ocupar gran cantidad de memoria o tiempo de procesamiento.

Dentro de los microcontroladores AVR de 8 bits se encuentra la familia ATMEGA, que se emplea en aplicaciones de propósito general y cuenta con un rango de características que van desde 4KB - 256KB de memoria de programa tipo flash, 28 - 100 pines y hasta 20 MHz de velocidad de reloj. Dentro de esta familia se encuentra el microcontrolador ATMEGA8 el cual cuenta con 8KB de memoria Flash, 1KB de memoria RAM estática, 512KB de memoria EEPROM, 28 o 32 pines según el empaquetado, 23 pines de entrada o salida, 2 interrupciones externas, además de contadores y convertidor analógico digital. Se optó por utilizar este microcontrolador para la gestión del flujo de

datos debido a su bajo costo, capacidades de memoria y velocidad intermedias, su fácil obtención en el mercado y todo el soporte que se obtiene con las herramientas de desarrollo de ATMEL®. [3]

Además de la comunicación inalámbrica, el manejo de los datos y la comunicación con la computadora; se requiere también de una interfaz con el usuario, dicha interfaz es un programa que sirve como medio entre el usuario y la computadora, generalmente este tipo de son intuitivos y amigables con el usuario para que se logre un fácil entendimiento del proceso, dichas interfaces de usuario suelen ser del tipo gráficas o denominadas en inglés GUI (Graphic User Interfaces).

Para lograr crear estas interfaces de usuario se requieren de lenguajes de programación de alto nivel y existen varias opciones disponibles como C++, C#, Visual Basic, Java, entre otros. Cada lenguaje ofrece diferentes ventajas y limitantes y de los cuales el lenguaje Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, con unos 10 millones de usuarios reportados. [4]

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems© (la cual fue adquirida por la compañía Oracle©) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems©. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a bytecode (clase Java) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora.

Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa.

El objetivo de esta tesis es desarrollar una interfaz para sistemas de control, que pueda ser una herramienta para cualquier ingeniero que requiera de una comunicación inalámbrica entre su sistema de control desarrollado y la computadora, pudiendo así lograr un control del proceso más intuitivo y amigable para el usuario final.

Para lograr este objetivo, se realizó un diseño arriba-abajo en donde se partió de una idea general, ésta se fue dividiendo en sus distintas secciones o apartados para poder desglosar cada componente de la misma, de esta forma desarrollar cada parte del sistema de forma separada y al final integrarse como la interfaz para sistemas de control deseada. La Figura 1.1 muestra el diagrama del diseño arriba-abajo de la interfaz desarrollada.

El diseño de cada sección se conformó de la siguiente manera: el estándar USB se utilizará para la comunicación con la computadora, un microcontrolador AVR ATMEGA8 de ATMEL para la gestión y el manejo del flujo de datos, un transceptor de radiofrecuencia para la comunicación inalámbrica con el sistema de control, el cual será definido en capítulos posteriores y finalmente para el

desarrollo de la interfaz de usuario, se utilizará el lenguaje de programación Java. Además de estas condiciones, se consideró que el diseño fuera de bajo costo y con pocos componentes, para que permitiera ser una opción viable frente a otras opciones en el mercado.

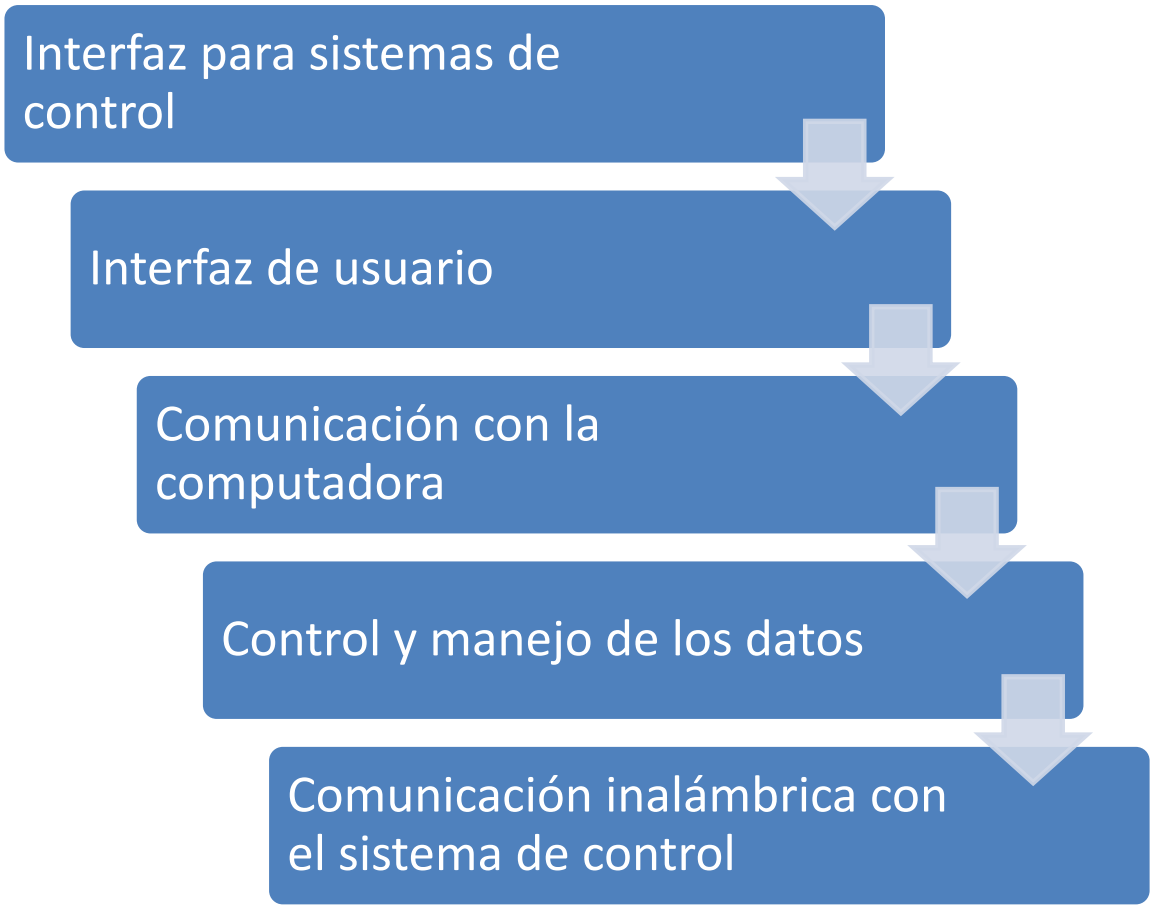

Figura 1.1.- Diagrama del diseño arriba-abajo utilizada para la interfaz.

Primeramente, se trataran los conceptos técnicos y teóricos del sistema en general; los capítulos posteriores definirán cada componente del sistema, como lo es la comunicación inalámbrica, la interfaz con la computadora y la interfaz de usuario; posteriormente se incluirá en un capítulo el diseño físico y la integración de cada aspecto desarrollado en los capítulos anteriores a éste, para después conformar la aplicación final de la interfaz del sistema de control. Se continúa con una serie de pruebas realizadas y finalmente se presentarán los resultados obtenidos sobre el proyecto desarrollado y sus conclusiones. En la sección de apéndices se podrá encontrar el código desarrollado, así como información técnica del sistema tales como extractos de hojas de datos y especificaciones. Todos los archivos de esta tesis, incluyendo librerías, documentación y archivos de programación están disponibles en línea para descargar desde la página web [http://sourceforge.net/projects/usb-rf-avr/files/].

Capítulo 2
Marco Teórico

En el presente capítulo se dan a conocer los conceptos que se requieren conocer para el desarrollo del proyecto y diseño de la aplicación final.

En esta sección abordaremos algunos conceptos técnicos y teóricos que nos ayudarán a comprender los criterios utilizados a lo largo de esta Tesis, no es la intención la de abarcar en detalle cada tópico involucrado, pero si la de dar una idea general sobre los elementos involucrados y sus particularidades.

Primeramente se definirán algunos conceptos sobre la comunicación inalámbrica y flujo de datos, posteriormente abordaremos las especificaciones del microcontrolador a utilizar, el AVR ATMEGA8, después se abordarán algunos conceptos sobre la interfaz SPI (Serial Peripheral Interface) y el protocolo USB (Universal Serial Bus), finalmente se tratarán algunas generalidades sobre el lenguaje de programación Java.

2.1 COMUNICACIÓN INALÁMBRICA

La comunicación mediante dos o más dispositivos puede tener distintas clasificaciones de acuerdo a lo que estemos evaluando, si nos referimos al medio físico en el que la comunicación se realiza entre el emisor y el receptor, podemos clasificarlas en dos tipos: alámbrica e inalámbrica. La comunicación inalámbrica es aquella que utiliza la modulación de las ondas electromagnéticas a través del espacio.

En general, se utilizan ondas de radiofrecuencia y una banda de frecuencia específica, para transmitir entre dispositivos. Las bandas de frecuencia son intervalos de frecuencia del espectro electromagnético asignados a diferentes usos dentro de las radiocomunicaciones. Su uso está regulado por la Unión Internacional de Telecomunicaciones y puede variar según el lugar. En México está gestionado por el IFT (Instituto Federal de Telecomunicaciones) y es el encargado de realizar todas las regulaciones y designaciones de las bandas de frecuencia en el país.

Existen bandas de frecuencia de uso libre que permiten transmitir sin necesidad de una licencia, esto ha propiciado el aumento en el número de equipos que utilicen medios inalámbricos para su comunicación.

La modulación consiste en enviar información o datos a través de una onda denominada portadora, la onda portadora la cual es de mayor frecuencia tiene la función de transportar a la onda moduladora (datos). Posteriormente de que se envía la información, ocurre el paso inverso conocido como demodulación, en donde el receptor recibe la señal y se encarga de quitar la onda portadora y solo procesar la señal moduladora y de esa forma obtener los datos enviados por el transmisor.

La modulación puede clasificarse en dos tipos: analógica y digital, dependiendo del tipo de datos que contenga la señal moduladora, en el caso de valores de señales digitales existen modulaciones del tipo ASK (modulación por desplazamiento de Amplitud), FSK (modulación por desplazamiento de frecuencia), PSK (modulación por desplazamiento de fase), por mencionar algunas, cada modulación modifica una característica en la onda portadora dependiendo de cada dato en la señal moduladora.

La comunicación inalámbrica ofrece muchos campos de aplicación, desde radio y televisión, telefonía, internet, entre otros.

Las ventajas que nos ofrece la comunicación inalámbrica respecto a la alámbrica es la movilidad y flexibilidad, además de reducir costos en cableado e instalación, siendo particularmente más beneficioso a grandes distancias, sin embargo, también existen desventajas, tales como mayor susceptibilidad a interferencias, retraso y atenuación en la señal.

2.2 MICROCONTROLADOR AVR ATMEGA8

Los microcontroladores son dispositivos programables que integran la arquitectura básica incluida en una computadora, en donde tenemos una unidad aritmética lógica (ALU), memoria de programa, memoria de almacenamiento y puertos de entrada/salida, además de esto, cada microcontrolador cuenta con distintas características adicionales, que van desde convertidores analógicos digitales hasta protocolos de comunicación integrados.

Tienen la ventaja de ser fáciles de programar y cuentan con herramientas de desarrollo en distintos lenguajes de programación como C, Basic, ensamblador, etc. En la actualidad existen diversas marcas especializadas en la manufactura y venta de microcontroladores tales como Microchip®, ATMEL®, NXP®, Freescale®, Texas Instruments®, entre otras, todas ofrecen distintas opciones en cuanto a características, puertos, memoria, número de bits, etc., permitiendo una flexibilidad al momento de escoger qué microcontrolador utilizar para alguna aplicación específica.

Entre los más destacados, se encuentran los microcontroladores PIC de Microchip® y los AVR de ATMEL®, su familia de microcontroladores de 8 bits es extensa y cuenta con herramientas de desarrollo gratuitas, las diferencias más significativas existen en su procesamiento, en donde, los microcontroladores AVR realizan una instrucción por cada ciclo de reloj, mientras que los PIC necesitan 4 ciclos de reloj para realizar una sola instrucción, debido a esto los PIC suelen ser más lentos, por ejemplo, si se utiliza un reloj con una frecuencia de oscilación de 4MHz, el AVR logra 4 millones de instrucciones por segundo mientras que el PIC sólo lograría 1 millón de instrucciones por segundo, adicionalmente las instrucciones de los AVR son más completas ya que permiten realizar operaciones más complejas que los PIC y en una sola instrucción. Otra característica importante en los AVR es su soporte directo con el lenguaje C, siendo de gran ayuda para desarrollar aplicaciones con mayor facilidad y sin ocupar gran cantidad de memoria o procesamiento.

Dentro de los microcontroladores AVR de 8 bits se encuentra la familia ATMEGA, la cual es de propósito general y cuenta con un rango de características que van desde 4KB - 256KB de memoria de programa tipo flash, 28 a 100 pines y hasta 20 MHz de velocidad de reloj. Dentro de esta familia se encuentra el microcontrolador ATMEGA8, el cual cuenta con 8KB de memoria Flash, 1KB de memoria RAM estática, 512KB de memoria EEPROM, 28 o 32 pines según el empaquetado, 23 pines de entrada o salida, 2 interrupciones externas, además de contadores y convertidor analógico digital. [3]

Se optó por utilizar este microcontrolador para la gestión del flujo de datos, debido a su bajo costo, capacidades de memoria y velocidad intermedias, su fácil obtención en el mercado y todo el soporte que se obtiene con las herramientas de desarrollo de ATMEL®.

2.3 INTERFAZ SPI (SERIAL PERIPHERAL INTERFACE)

Una de las características adicionales con que se cuenta en casi todos los microcontroladores es la interfaz periférica serial o SPI, esta interfaz es un estándar de comunicaciones serial usado ampliamente en la transferencia de información entre dispositivos electrónicos.

El flujo de los datos es síncrono, esto quiere decir, que se cuenta con una señal de reloj idéntica en todos los dispositivos involucrados en la comunicación, de esta manera se mantienen todos a la misma velocidad de operación y el flujo de datos siempre esta sincronizado.

La comunicación la realiza a través de 4 señales:

- **SCLK** (Clock): Es el pulso que marca la sincronización. Con cada pulso de este reloj, se lee o se envía un bit.
- **MOSI** (Master Output Slave Input): Salida de datos del Maestro y entrada de datos al Esclavo.
- **MISO** (Master Input Slave Output): Salida de datos del Esclavo y entrada al Maestro.
- **SS** (Slave Select): Señal que permite seleccionar un Esclavo, o para que el Maestro le diga al Esclavo que se active.

El dispositivo Maestro (Master), es aquel que gestiona a los demás dispositivos en el bus SPI, generalmente es el microcontrolador o microprocesador el dispositivo Maestro, sin embargo, en algunas aplicaciones pueden llegar a ser incluso Esclavos (Slave). Este dispositivo también es el encargado de generar la señal de reloj SCLK que alimenta a todos los Esclavos.

Los Esclavos (Slave), son todos aquellos dispositivos que se conectan al Maestro, se les llama así porque es el Maestro el que designa con quien comunicarse y cuándo comunicarse. El dispositivo Esclavo no puede realizar ninguna comunicación a menos que el Maestro se lo indique.

El proceso de comunicación consiste en que el Maestro envía una señal de activación por la señal SS (Slave Select) mediante un cero lógico, esto hace que sólo el Esclavo activado se pueda comunicar con el Maestro, posteriormente inicia el proceso de envío y recepción de datos, esto se hace de forma paralela a través de las señales MOSI y MISO, mientras el Maestro envía un bit a través de la señal MOSI, simultáneamente el Esclavo envía un bit por la señal MISO, esto permite enviar y recibir datos al mismo tiempo, a este concepto se le conoce como comunicación Full-dúplex.

Es muy importante la gestión de la señal de activación del Esclavo SS, ya que si el Maestro se está comunicando con varios Esclavos, tendrá que activar sólo al que desee comunicarse ya que todos los Esclavos comparten el mismo Bus, tal como se observa en la Figura 2.2, de no ser así podría causar un corto circuito entre los componentes involucrados.

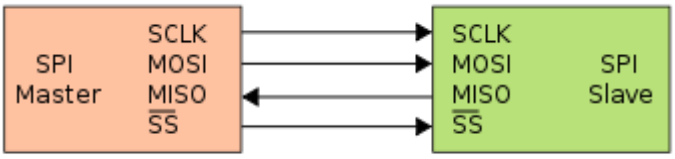

Figura 2.1.- Bus SPI con un Maestro y un Esclavo.[2]

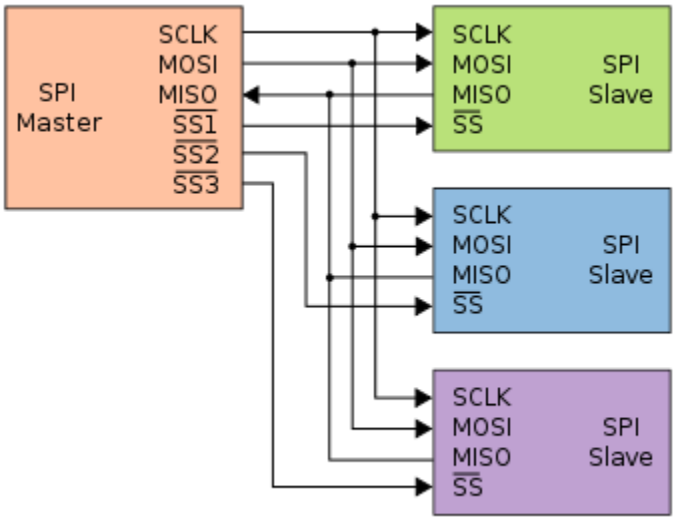

Figura 2.2.- Bus SPI con un Maestro y 3 Esclavos.[3]

Cada dispositivo gestiona la interfaz SPI de diferentes maneras, se deberá referir a las hojas de especificación del dispositivo para habilitar la interfaz y realizar la comunicación. Para el caso del microcontrolador ATMEGA 8, la información referente a la interfaz SPI puede consultarse desde su hoja de especificación, capítulo "Serial Peripheral Interface – SPI", páginas 121-128. [3]

[2] Imagen tomada de http://commons.wikimedia.org/wiki/File:SPI_single_slave.svg
[3] Imagen tomada de http://commons.wikimedia.org/wiki/File:SPI_three_slaves.svg

2.4 PROTOCOLO USB (UNIVERSAL SERIAL BUS)

El Universal Serial Bus (USB) es un estándar industrial desarrollado a mediados de los años 1990 que define los cables, conectores y protocolos usados en un bus para conectar, comunicar y proveer de alimentación eléctrica entre ordenadores, periféricos y dispositivos electrónicos [1].

En la actualidad se ha vuelto un estándar y su campo de aplicación se extiende a cualquier dispositivo electrónico como computadoras, dispositivos portátiles (laptops, celulares, tabletas, etc.). Los dispositivos que cuentan con USB van desde mouse, teclados, controles, escáneres, cámaras, impresoras, entre otros. [2]

Características

- **Una interfaz para muchos dispositivos**. El USB es bastante versátil, en lugar de tener diferentes conectores y tipos de cables para cada función del periférico, una interfaz sirve para muchas.

- **Configuración automática.** Cuando el usuario conecta un dispositivo USB a la PC, el sistema operativo detecta al dispositivo y carga el controlador de software apropiado. Si es la primera vez que se conecta el dispositivo el sistema operativo intentará utilizar algún controlador predefinido o se tendrá que instalar, sin embargo, este proceso sólo se realiza una vez y sin reiniciar el equipo.

- **Fácil de conectar.** Una computadora típicamente tiene múltiples puertos USB, y mediante HUBS[4] se puede ampliar la capacidad.

- **Cables convenientes.** Los conectores USB son compactos comparados con otras interfaces como el RS-232. Para permitir una operación confiable, la especificación USB define las características eléctricas que deben tener los cables, un segmento de cable puede llegar a ser hasta de 5 metros dependiendo de la velocidad del bus.

- **Conexión en caliente o Hot Plug.** Los usuarios pueden conectar y desconectar un dispositivo USB cuando se desee, sin importar si el sistema o el dispositivo estén energizados, sin dañar la computadora o el dispositivo. El sistema operativo detecta cuando el dispositivo está conectado y listo para usarse.

- **Sin necesidad de configurarse por el usuario.** Los dispositivos USB no tienen opciones de configuración tales como direcciones de puerto, línea de petición de interrupción (IRQ), así que los usuarios no tienen que ajustar el hardware o correr utilidades de configuración.

Funcionamiento

Cada dispositivo que reside en el USB tiene asignada una dirección solo conocida por el subsistema USB y no consume ningún recurso del sistema. USB soporta hasta 127 dispositivos

[4] Un hub USB es un dispositivo que permite concentrar varios puertos USB, permitiendo la conexión con la computadora mediante un solo bus o cable.

simultáneamente mediante la utilización de un HUB. Los dispositivos USB contienen números de registros individuales o puertos que pueden ser accedidos indirectamente por los controladores del dispositivo USB. Estos registros son conocidos como los “endpoints”.

Cuando una transacción es enviada en el USB, todos los dispositivos (exceptuando los de baja velocidad) verán la transacción. Cada transacción comienza con la trasmisión de un paquete que define el tipo de transacción a ser realizada por el dispositivo USB y la dirección del endpoint.

Esta dirección es manejada por el software USB. Cada dispositivo USB debe tener una dirección interna por defecto (llamada endpoint cero) que es reservada para configurar el dispositivo. Por medio del endpoint cero, el software del sistema USB lee los descriptores estándar del dispositivo. Estos descriptores proveen la información de configuración necesaria para la inicialización del software y el hardware. De esta manera el software del sistema puede detectar el tipo de dispositivo (o información de la clase) y determinar cómo el dispositivo se intenta acceder.

USB soporta hasta cuatro velocidades en el Bus: SuperSpeed a 5Gbps (disponible únicamente en USB 3.0), highspeed a 480Mbps, full speed a 12Mbps y low speed a 1.5Mbps.

Actualmente el USB cuenta con las especificaciones USB 1.0, 1.1, 2.1, 3.0 y 3.1, cada una fue mejorando aspectos en velocidad y rendimiento, y las más actuales son retro compatibles con versiones anteriores, es decir, un USB 3.0 es compatible con dispositivos USB 2.0 y 1.0.

La interfaz USB realiza su comunicación mediante transferencias, las cuales de acuerdo a la especificación se dividen en:

- Transferencias de Control
- Transferencias tipo Bulto o Bulk
- Transferencias de Interrupción
- Transferencias Isócronas

Cada una ofrece distintas opciones en cuanto a tamaño de datos por paquete, estructura de los datos (mensajes o difusión continua), corrección de errores, etc., algunas transferencias sólo están disponibles a velocidad full speed en adelante, tales como las de tipo bulto e isócronas.

Toda la información relevante sobre especificaciones, productos y desarrollo de la interfaz USB, puede ser consultada desde el portal de internet: [http://www.usb.org/home]

2.5 LENGUAJE DE PROGRAMACIÓN JAVA

El lenguaje de programación Java fue originalmente desarrollado por James Gosling de Sun Microsystems© (la cual fue adquirida por la compañía Oracle©) y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems©. Su sintaxis deriva mucho de C y C++, pero tiene menos facilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a bytecode (clase Java) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora.

Java es la base para prácticamente todos los tipos de aplicaciones de red, además del estándar global para desarrollar y distribuir aplicaciones móviles y embebidas, juegos, contenido basado en web y software de empresa.

Es un lenguaje de programación de propósito general, concurrente, orientado a objetos y basado en clases que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo (conocido como WORA, o "write once, run anywhere"), lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. [4]

El lenguaje Java tiene las siguientes características:

- Programación orientada a objetos y clases.
- Permite la ejecución de un mismo programa en múltiples sistemas operativos.
- Incluye por defecto soporte para trabajo en red.
- Ejecuta código en sistemas remotos de forma segura.
- Es fácil de usar y toma lo mejor de otros lenguajes orientados a objetos, como C++.

La sintaxis de Java se deriva en gran medida de C++. Pero a diferencia de éste, que combina la sintaxis para programación genérica, estructurada y orientada a objetos, Java fue construido desde el principio para ser completamente orientado a objetos. Todo en Java es un objeto (salvo algunas excepciones), y todo en Java reside en alguna clase (una clase es un molde a partir del cual pueden crearse varios objetos).

Actualmente existen varias plataformas de desarrollo de Software gratuitas que permiten la programación en Java de manera más sencilla y práctica, tales como Eclipse o NetBeans, éstas ofrecen herramientas de depuración y compilación, en ambientes totalmente gráficos que permiten un entorno más amigable para el usuario.

Además de esto, debido al amplio uso de Java muchos desarrolladores han realizado librerías de distintas funciones que permiten ser implementadas a un proyecto propio y ampliar el alcance del mismo.

Capítulo 3
Desarrollo de la comunicación inalámbrica

En éste capítulo se expone la forma en la que se realizó la comunicación inalámbrica, se retoman algunos conceptos explicados en el capítulo anterior y cómo estos conceptos se emplean para el desarrollo de dicha comunicación.

La primera parte a desarrollar en el proyecto fue la de lograr una comunicación inalámbrica entre el dispositivo propuesto en este proyecto, el AVR ATMEGA 8 y cualquier otro microcontrolador o microprocesador, esto con motivo de que la aplicación de control fuera flexible para el diseñador.

Las características deseadas para la comunicación inalámbrica del sistema fueron las siguientes:

1. Comunicación bidireccional entre el sistema y la aplicación de control, para poder aplicarse en sistemas de lazo cerrado.
2. Alcance de por lo menos 50 m., para tener una distancia aceptable entre el sistema de control y la computadora.
3. Velocidades de transferencia de por lo menos 256Kbps para que permita una gran transferencia de datos en poco tiempo.
4. Selección de la frecuencia de transmisión para que permita al sistema adecuarse a las posibles interferencias de distintas fuentes de frecuencia.
5. Codificación en los datos para contrarrestar interferencias.
6. Interfaz estándar que permita comunicación con múltiples microcontroladores o microprocesadores, de esta forma se vuelve flexible la selección del controlador para el sistema de control.

Dados los requerimientos del sistema, se procedió a elegir la forma de implementar ésta comunicación, que no sólo cumpliera con lo especificado, sino que además no tuviera un costo elevado.

El alcance de este trabajo no comprende el diseño del dispositivo de comunicación, sólo su implementación. Por tal motivo, nos enfocamos en la tarea de investigar la factibilidad de usar un dispositivo que cumpliera con las características. El dispositivo seleccionado para ésta tarea fue el NRF24L01+, que a continuación definiremos con mayor detalle.

3.1 CIRCUITO TRANSCEPTOR DE RADIOFRECUENCIA NRF24L01+

El dispositivo NRF24L01+ es un circuito integrado transceptor diseñado por *NORDIC SEMICONDUCTOR* (Ver Figura 3.1), que permite realizar una comunicación inalámbrica vía radiofrecuencia utilizando la banda de frecuencia de los 2.4 Ghz a los 2.5Ghz, la cuál está comprendida en la banda ISM (Industrial, Scientific and Medical), el uso de esta banda de frecuencia es de uso libre en todo el mundo.[5]

Las características principales del dispositivo son las siguientes [5]:

- Transceptor tipo GFSK (Gaussian Frequency-Shift Keying) de RF de bajo costo con banda de operación ISM de 2.4GHz.
- Transferencia de datos en el aire de 250Kbps, 1Mbps y 2Mbps.
- Operación de bajo consumo ULP (Ultra Low Power).

[5] No necesita de alguna licencia siempre que se respeten las regulaciones que limitan los niveles de potencia transmitida.

- Voltaje de operación de 1.9 V– 3.6 V.
- Tecnología Enhanced ShockBurst™ [6].
- Manejo de paquetes de datos automático.
- Hasta 6 canales de comunicación (pipes) para recepción de datos.
- Chip compacto de 20 pines con empaquetado de 4x4mm QFN (Quad Flat No-Lead).
- Interfaz digital SPI (Serial Peripheral Interface) de hasta 10Mbps.

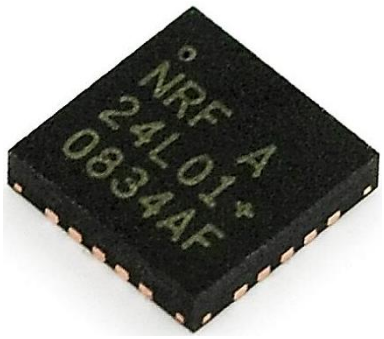

Figura 3.1 - Transceptor NRF24L01+ de Nordic Semiconductor[7].

De acuerdo a las características descritas, se analiza si se cumplen con las características deseadas del sistema.

1. Comunicación bidireccional entre el sistema y la aplicación de control.

Debido a que el NRF24L01+ es un circuito transceptor, permite configurarse como transmisor o receptor, con el único inconveniente de que no puede ser ambos al mismo tiempo, por lo tanto se puede lograr una comunicación bidireccional del tipo half-duplex.

2. Alcance de por lo menos 50m.

De acuerdo a las especificaciones encontradas en la hoja de datos del NRF24L01+, el alcance varía entre los 50-100m dependiendo del entorno y la antena utilizada.

3. Velocidades de transferencia de por lo menos 256Kbps.

Las velocidades de transmisión del NRF24L01+ permiten transferencias de 256Kbps, 1Mbps y hasta 2Mbps, configurables desde un registro.

[6] La tecnología Enhanced Shockburst es una capa de enlace de datos patentada por *Nordic Semiconductor*.
[7] Figura tomada de [http://www.tenettech.com/product/854/24ghz-transceiver-ic-nrf24l01]

4. Selección de la frecuencia de transmisión.

Una de las características importantes del NRF24L01+, es la posibilidad de seleccionar la frecuencia de transmisión de un rango de 127 canales espaciados 1MHz partiendo desde 2.4GHz, con tan sólo modificar un registro.

5. Codificación en los datos para contrarrestar interferencias.

El circuito NRF24L01+ cuenta con la tecnología patentada Enhanced ShockBurst™, que tiene la particularidad de manejar paquetes automáticamente además de utilizar codificación CRC (comprobación de redundancia cíclica) que puede ser de 1 o 2 bytes, esto permite una comunicación más segura sin errores en los datos.

6. Interfaz estándar que permita la comunicación con múltiples microcontroladores o microprocesadores.

La interfaz utilizada por el NRF24L01+ es el bus SPI (Serial Peripherial Interface) el cuál es un estándar de comunicaciones, usado ampliamente en una gran variedad de microprocesadores y microcontroladores.

Se observa que las características del dispositivo cumplen perfectamente con las necesidades del sistema incluyendo otras que mencionaremos más adelante.

Otro aspecto importante que no se enlistó pero que es importante mencionar, es el precio, el circuito tiene un precio relativamente bajo, llegando a precios desde $4.00 USD con antena integrada en el PCB o hasta $10.00 USD con antena externa, los cuales además de incluir el circuito, viene soldado en éste PCB con todo el hardware adicional necesario para utilizar el transceptor (cristal de cuarzo, antena y otros elementos pasivos), además de tener sus salidas soldadas a un pin macho del tipo header de 4x2 o 5x2 con espaciamiento de 2.54mm (Ver Figura 3.2).

Existen otros modelos de placas con diferente esquema y relación de pines, pero se optó por el mostrado en la Figura 3.2, ya que es el que se encuentra ampliamente en el mercado.

Figura 3.2 - Placa de circuito impreso que incluye: Transceptor nRF24L01+, antena (antena impresa en el mismo PCB foto Izquierda, con antena externa foto derecha), elementos pasivos y header de 4x2.[8]

3.1.1 Información sobre los pines

Como se mencionó anteriormente el dispositivo NRF24L01+ cuenta con un total de 20 pines en un empaquetado QFN de 4x4mm, la asignación de pines y su descripción pueden observarse en la Figura 3.3 y la Tabla 3.1.

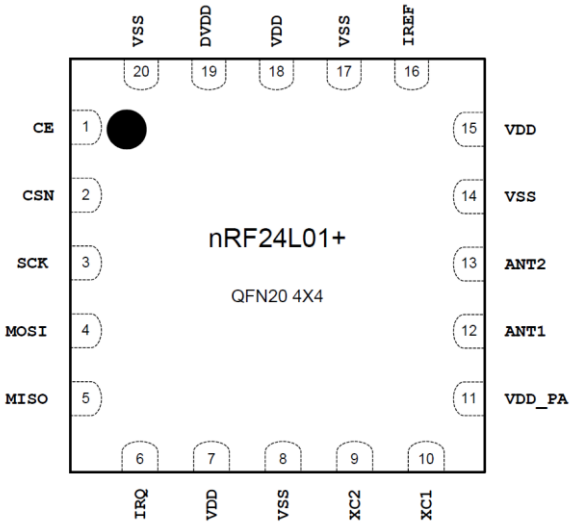

Figura 3.3.- Asignación de pines del NRF24L01+ (vista desde arriba) para el empaquetado QFN. [6]

[8] Figura tomada de: [http://www.elecfreaks.com/wiki/index.php?title=2.4G_Wireless_nRF24L01p]

Pin	Nombre	Función del Pin	Descripción
1	CE	Entrada Digital	Chip Enable. Activa Modo RX o TX.
2	CSN	Entrada Digital	Chip Select para la interfaz SPI.
3	SCK	Entrada Digital	Clock para la interfaz SPI.
4	MOSI	Entrada Digital	Slave Data Input para la interfaz SPI.
5	MISO	Salida Digital	Slave Data Output para la interfaz SPI.
6	IRQ	Salida Digital	Interrupción mascarada. Se activa en nivel bajo
7	VDD	Voltaje	Voltaje de alimentación (+1.9V – +3.6V)
8	VSS	Voltaje	Tierra (0V)
9	XC2	Salida Analógica	Pin de cristal 1
10	XC1	Entrada Analógica	Pin de cristal 2
11	VDD_PA	Voltaje de Salida	Voltaje de salida (+1.8V) para amplificador interno
12	ANT1	RF	Interfaz para Antena 1
13	ANT2	RF	Interfaz para Antena 2
14	VSS	Voltaje	Tierra (0V)
15	VDD	Voltaje	Voltaje de alimentación (+1.9V – +3.6V)
16	IREF	Entrada Analógica	Voltaje de referencia. Conectar a tierra con una resistencia de 22KΩ
17	VSS	Voltaje	Tierra (0V)
18	VDD	Voltaje	Voltaje de alimentación (+1.9V – +3.6V)
19	DVDD	Voltaje de Salida	Voltaje de salida digital interno
20	VSS	Voltaje	Tierra (0V)

Tabla 3.1- Función de los pines en el NRF24L01+. [6]

Debido a que para el desarrollo del sistema se utilizó la placa soldada, como se muestra en la Figura 3.2, el sistema se enfocará solamente en 8 pines: CE, CSN, SCK, MOSI, MISO, IRQ, VDD y VSS.

Los primeros 5 pines corresponden a la interfaz SPI, que se trató con mayor detalle en el Capítulo 2; el pin IRQ corresponde a una interrupción opcional que puede configurarse para conocer si se ha recibido un dato cuando se está en el modo receptor o si se envió un dato satisfactoriamente estando en el modo transmisor; finalmente, los pines VDD y VSS son para la alimentación del circuito (recordar que su voltaje de operación es de entre 1.9V – 3.6V).

3.1.2 Modos de operación

A continuación describiremos los modos de operación y los parámetros utilizados para el control por radiofrecuencia en el NRF24L01+.

Modo Apagado (Power Down).

En el modo apagado o Power Down el NRF24L01+ consume una corriente mínima. Todos los valores de los registros se mantienen y la interfaz SPI se mantiene activa, permitiendo cambiar la

configuración y la lectura y escritura de los registros. Se entra al modo apagado al configurar el bit PWR_UP del registro CONFIG en nivel bajo.

Modo en espera-I (Standby-I).

Al configurar el bit PWR_UP del registro CONFIG en 1, el dispositivo entra en el modo de espera-I o Standby-I. El modo Standby-I se utiliza para minimizar la corriente promedio de consumo y mantener los tiempos de arranque cortos. En este modo sólo una parte del cristal oscilador está activo. El cambio a algún modo activo sólo sucede si el pin CE se activa en nivel alto y cuando CE está en nivel bajo, el NRF24L01 regresa al modo Standby-I de ambos modos RX y TX.

Modo en espera-II (Standby-II).

En el modo Standby-II se consume más corriente comparado con el modo Standby-I. Se entra en este modo cuando el pin CE se mantiene en nivel alto mientras se configura como modo TX y el FIFO[9] de TX está vacío. Si se inserta un paquete de datos al FIFO de TX, el PLL[10] inmediatamente empieza a transmitir el paquete después de un retraso de 130µs.

Modo Receptor (RX).

El modo receptor o RX es un modo activo en el cual el NRF24L01+ se configura como receptor de datos. Para entrar a este modo se debe colocar el bit PWR_UP, el bit PRIM_RX y el pin CE en nivel alto.

En el modo RX, el receptor demodula las señales del canal de RF y constantemente envía los datos demodulados al motor del protocolo de banda base. El motor de protocolo de banda base constantemente busca un paquete válido, si encuentra un paquete válido (al coincidir la dirección y el CRC), los datos del paquete son presentados en un espacio libre en los FIFO's del RX. Si los FIFO's del RX están llenos, el paquete recibido se descarta.

El NRF24L01+ permanece en modo RX hasta que se configura como modo Standby-I o modo Power Down.

Modo Transmisor (TX).

El modo TX es un modo activo que permite transmitir paquetes. Para entrar a este modo se deben configurar el bit PWR_UP en nivel alto, el bit PRIM_RX en nivel bajo, debe existir un bloque de datos en el FIFO de TX y un pulso de nivel alto en el pin CE de por lo menos 10µs.

El NRF24L01+ permanece en modo TX hasta que termina de enviar el paquete de datos. Si CE = 0, regresa al modo Standby-I. Si CE = 1, el estado del FIFO de TX determina la siguiente acción. Si el FIFO de TX no está vacío el NRF24L01+ permanece en modo TX y transmite el siguiente paquete. Si no está vacío el NRF24L01+ entra en modo Standby-II.

[9] El registro tubo o FIFO es una memoria de acceso serie utilizada para el registro de datos. FIFO es el acrónimo en inglés de First In, First Out (primero en entrar, primero en salir).
[10] El PLL (Phase Loop Locked) es un circuito que permite que una señal de referencia externa, controle la frecuencia y la fase de un oscilador.

La Tabla 3.2 describe como configurar los distintos modos de operación.

Modo	Bit PWR_UP[11]	Bit PRIM_RX	Pin de entrada CE	Estado del FIFO
RX	1	1	1	-
TX	1	0	1	Datos en el FIFO de TX. Vaciara todos los niveles del FIFO[12]
TX	1	0	Pulso mínimo en nivel alto de 10µs	Datos en el TX FIFO. Vaciará un nivel del FIFO.[13]
Standby-II	1	0	1	FIFO de TX vacío
Standby-I	1	-	0	No hay ninguna transmisión de paquete pendiente
Power Down	0	-	-	-

Tabla 3.2.- Modos principales del NRF24L01+.

3.1.3 Tiempos de operación

Los tiempos de transiciones entre modos y los tiempos para el pin CE se muestran en la Tabla 3.3.

NRF24L01+	Max.	Min.
Power Down → Standby	4.5ms	1.5ms
Standby → TX/RX	130µs	
Tiempo mínimo CE en alto		10µs
Retraso desde el flanco positivo de CE a CSN en nivel bajo		4µs

Tabla 3.3.- Tiempos de operación del NRF24L01+.

[11] El bit PWR_UP y el bit PRIM_RX se encuentran en el registro CONFIG.
[12] Si CE se mantiene en alto todos los FIFO's del TX se vacían, si CE sigue en alto cuando se vacíen todos los niveles del FIFO de TX, el NRF24L01+ entrará al modo standby-II.
[13] Esta operación permite transmitir sólo un paquete. Éste es el modo normal de operación. Después de enviar el paquete, el NRF24L01+ entra en modo standby-I.

3.1.4 Velocidad de transferencia en el aire

La velocidad de transferencia en el aire está dada por la velocidad de modulación de la señal del NRF24L01+, y puede ser de 250Kbps, 1Mbps o 2Mbps. Utilizando una menor velocidad de transferencia, aumenta la sensibilidad del receptor y por ende el alcance, pero a mayor velocidad de transferencia disminuye la corriente de consumo y reduce la probabilidad de colisiones en el aire.

La velocidad de transferencia se configura en el bit RF_DR del registro RF_SETUP. Tanto el transmisor como el receptor deben programarse a la misma velocidad de transferencia para lograr comunicarse entre sí.

3.1.5 Canal de frecuencia RF

El canal de frecuencia RF determina el centro del canal usado por el NRF24L01+. El canal ocupa un ancho de banda menor a 1MHz a 250Kbps y 1Mbps, y un ancho de banda de menos de 2MHz a 2Mbps. El NRF24L01+ puede operar a frecuencias desde los 2.400GHz hasta 2.525GHz. La resolución de ajuste del canal de frecuencia RF es de 1MHz.

A 2Mbps el canal ocupa un ancho de banda más ancho que la resolución de ajuste del canal de RF, para evitar traslapes en el modo de 2Mbps, el espaciamiento de canales deberá ser de 2MHz o más, mientras que a velocidades de 1Mbps y 250Kbps el ancho de banda es igual o menor a la resolución de ajuste.

El canal de frecuencia RF se configura a través del registro RF_CH de acuerdo a la Ecuación 3.1:

$$F_0 = 2400 + RF\_CH \ \ [MHZ]$$

Ecuación 3.1.- Ecuación para determinar el canal de frecuencia de RF en el dispositivo NRF24L01+. [6]

Se deben programar tanto el transmisor como el receptor al mismo canal de frecuencia para que puedan comunicarse entre sí.

3.1.6 Control de ganancia

El control del amplificador de potencia en el NRF24L01+ permite configurar la potencia de transmisión y regular el consumo de corriente, existen cuatro ajustes en total, mostrados en la Tabla 3.4.

El control del amplificador de potencia se realiza mediante la configuración del bit RF_PWR del registro RF_SETUP.

SPI RF-SETUP (RF_PWR)	Potencia de salida de RF	Consumo de corriente
11	0dBm	11.3mA
10	-6dBm	9.0mA
01	-12dBm	7.5mA
00	-18dBm	7.0mA

Tabla 3.4.- Ajustes de potencia para el NRF24L01+ [6].[14]

3.1.7 Control de RX/TX

El control RX/TX se programa en el bit PRIM_RX del registro CONFIG y configura al NRF24L01+ en modo transmisor/receptor.

3.1.8 Tecnología Enhanced Shockburst™

La tecnología Enhanced Shockburst™ es una capa de enlace de datos basada en paquetes que se caracteriza por el ensamble automático de paquetes, su sincronización y el reconocimiento y retransmisión automático de paquetes.

Características

Las características principales de Enhanced Shockburst™ son:

- Tamaño de bloques de datos (Payload) dinámicos desde 1 hasta 32 bytes.
- Manejo automático de paquetes.
- Manejo automático de transacción de paquetes.
  - Auto reconocimiento con bloques de datos.
  - Auto retransmisión
- 6 canales de datos MultiCeiver™[15] para redes tipo estrella 1:6.

Enhanced Shockburst™ utiliza Shockburst™ para el manejo automático de paquetes y su sincronización. Durante la transmisión, Shockburst™ ensambla el paquete y envía los pulsos de

[14] Condiciones: VDD = 3.0V, VSS = 0V, T_A = 27°C, Impedancia de antena = 15Ω + j88Ω

[15] La tecnología MultiCeiver™ es una función utilizada por el nRF24L01+ que consiste en seis canales de comunicación con direcciones únicas.

reloj en cada bit del paquete de datos para la transmisión. Durante la recepción, Shockburst™ constantemente busca una dirección válida en la señal demodulada, cuando encuentra una dirección valida, procesa el resto del paquete y lo valida por medio del CRC. Si el paquete es válido mueve el bloque de datos a un espacio vacío en la FIFO de RX.

Formato de los paquetes

El paquete de Enhanced Shockburst™ contiene un encabezado o preámbulo, una dirección, un paquete de control, un bloque de datos y un campo de CRC. La Figura 3.4 muestra la trama de paquetes utilizada, con el bit más significativo (MSB) a la izquierda.

Encabezado (1 byte)	Dirección (3 – 5 bytes)	Control de paquetes (9 bits)	Bloque de Datos (0 – 32 bytes)	CRC (1-2 bytes)

Figura 3.4.- Trama con bloque de datos (0 – 32 bytes) utilizada en Enhanced Shockburst™ [6].

Encabezado

El encabezado es una secuencia de bits utilizada para sincronizar el demodulador del receptor para la llegada de flujo de bits. El encabezado tiene una longitud de 1 byte y puede ser `01010101` o `10101010`. Si el primer bit de la dirección es un 1, el preámbulo se configura automáticamente a `10101010`. Si es `0`, se configura a `01010101`. Esto se hace para asegurar que existan suficientes transiciones en el preámbulo para estabilizar al receptor.

Dirección

Esta dirección es refiere a la del receptor. Una dirección asegura que el paquete sea detectado y recibido por el receptor correcto, previniendo diafonía[16] entre múltiples sistemas nRF24L01+.

La longitud del campo de dirección puede configurarse en el registro `AW` y puede ser de 3, 4 o hasta 5 bytes. (Ver Apéndice 1).

Se debe tener cuidado de utilizar direcciones en donde los niveles lógicos sólo se intercambian una vez (por ejemplo, `000FFFFFFF`), de lo contrario pueden presentar ruido y dar una falsa detección, el cuál, a su vez, puede resultar en una elevada tasa de paquetes con errores. Direcciones con una continuación del preámbulo (alternando 1's y `0`'s) también incrementa esta tasa.

[16] La diafonía o Crosstalk se refiere a la perturbación de un nodo de comunicación con respecto a otro. En donde parte de las señales presentes en uno de ellos (considerado perturbador), aparece en el otro.

Control de paquetes

La Figura 3.5 muestra el formato de 9 bits para el control de paquetes, con el bit más significativo a la izquierda.

Longitud del bloque de datos (6bits)	PID (2 bits)	NO_ACK (1 bit)

Figura 3.5.- Campo de control de paquetes.

El campo de control de paquetes contiene 6 bits que corresponden a la longitud del bloque de datos, 2 bits para el identificador de paquetes PID y un bit NO_ACK como bandera.

La longitud del bloque de datos especifica la longitud en bytes para los bloques de datos que pueden ser desde 0 hasta 32.

Codificación: `000000 = 0 bytes, 100000 = 32 bytes, 100001 = Don't care`

Este campo sólo se utiliza si la opción DPL (Dynamic Payload Length) está habilitada.

El campo PID (Paquete Identificador de Bits) se utiliza para detectar si el paquete que se envía es un nuevo paquete o es una retransmisión. El PID evita que el receptor reciba dos veces el mismo paquete de datos.

La bandera NO_ACK (No Acknowledgment) sólo se utiliza cuando la opción de auto reconocimiento está habilitada. Al colocar la bandera en nivel alto le dice al receptor que el paquete no será auto reconocido.

Bloques de datos

Los bloques de datos o Payload se refieren al contenido definido por el usuario. Puede ser de 0 hasta 32 bytes de ancho y se transmite en el aire cuando se cargan los datos al nRF24L01+.

Enhanced Shockburst™ provee dos alternativas para el manejo de las longitudes en los bloques de datos: estáticos y dinámicos.

Por defecto se usan los estáticos, con una longitud estática todos los paquetes entre el transmisor y el receptor tienen la misma longitud, la longitud se define en los registros `RX_PW_Px` del lado receptor y del lado transmisor por el número de bytes enviados, éstos deben ser idénticos a los definidos por el respectivo valor `RX_PW_Px` en los registros del lado receptor.

La otra opción es utilizar una longitud dinámica o DPL (Dynamic Payload Length), ésta alternativa permite al transmisor enviar paquetes con una longitud variable al receptor, esto significa que, en un sistema con bloques de datos variables, no es necesario ajustar la longitud de paquetes a la máxima utilizada.

Con la función DPL, el nRF24L01+ puede decodificar la longitud del paquete recibido automáticamente en lugar de utilizar los registros RX_PW_Px. El microcontrolador puede entonces leer la longitud del paquete de datos utilizando el comando R_RX_PL_WID.

Es importante que si se utiliza el DPL, siempre se debe verificar si el paquete es de 32 bytes o menor. Si es mayor a 32 bytes entonces el paquete contiene errores y debe ser descartado utilizando el comando FLUSH_RX.

Comprobación de Redundancia Cíclica (CRC)

La CRC es un mecanismo obligatorio para la detección de errores en los paquetes. Puede ser de 1 o 2 bytes y se calcula de acuerdo a la dirección, el campo de control de paquetes y el bloque de datos.

El valor del polinomio para la CRC de 1 byte es $X^8 + X^2 + X + 1$. Con valor inicial de 0xFF.

El valor del polinomio para la CRC de 2 bytes es $X^{16} + X^{12} + X^5 + 1$. Con valor inicial de 0xFFFF.

El número de bytes en la CRC se configura en el bit CRCO del registro CONFIG. Ningún paquete es aceptado si la CRC falla.

3.1.9 Auto-reconocimiento (Auto acknowledgment)

El auto-reconocimiento es una función que permite transmitir automáticamente un paquete de acuse de recibo (ACK) al dispositivo transmisor después de haber recibido y validado un paquete. Esto permite reducir la carga en el microcontrolador de la aplicación de control y elimina la necesidad de un hardware dedicado para el SPI, además de reducir el costo y la corriente de consumo promedio. Para habilitar la característica de auto-reconocimiento se debe configurar en el registro EN_AA.

Adicionalmente un paquete ACK puede contener un bloque de datos del dispositivo receptor para el dispositivo transmisor. Para poder utilizar esta opción se debe tener activada la longitud dinámica de bloques de datos DPL. El controlador del lado receptor tiene que cargar el bloque de datos en el FIFO de TX utilizando el comando W_ACK_PAYLOAD. El bloque de datos queda pendiente en la FIFO de TX hasta que un nuevo paquete es recibido por el transmisor, el nRF24L01+ puede tener hasta tres bloques de datos en el ACK pendientes en la FIFO al mismo tiempo.

La Figura 3.6 muestra cómo la FIFO de TX opera cuando controla un bloque de datos en ACK pendiente. Del controlador, el bloque de datos es cargado con el comando W_ACK_PAYLOAD, la dirección del decodificador y el buffer del controlador se aseguran de que el bloque de datos quede guardado en un espacio vacío dentro de la FIFO de TX. Cuando el paquete es recibido, la dirección del decodificador y el buffer del controlador son notificados con la dirección del dispositivo transmisor. Esto asegura que el bloque de datos correcto se presente al generador de ACK.

Si el FIFO de TX contiene más de un bloque de datos en el ACK, los bloques de datos se enviarán con el principio de: “el primero en entrar es el primero en salir” (FIFO). El controlador puede limpiar el FIFO de TX con el comando FLUSH_TX.

Para habilitar el auto-reconocimiento con bloques de datos es necesario poner en 1 el bit EN_ACK_PAY del registro FEATURE.

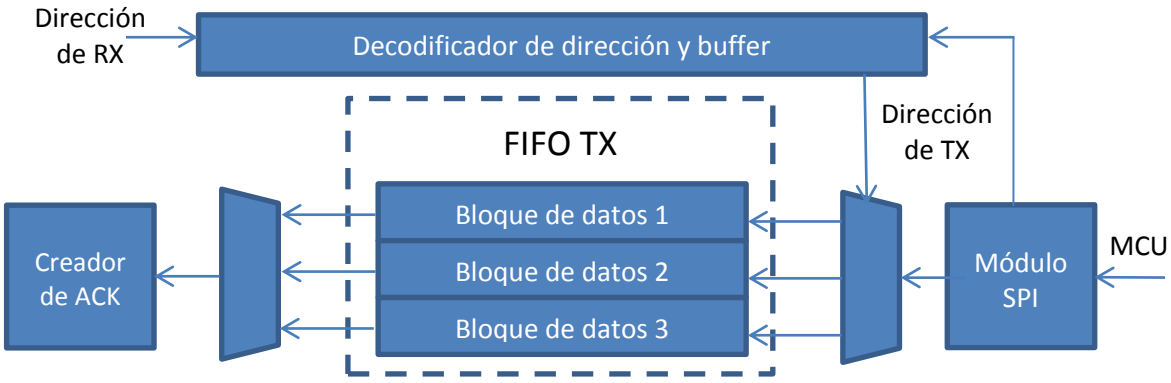

Figura 3.6.- FIFO de TX del dispositivo receptor con bloques de datos pendientes.

3.1.10 Auto-retransmisión (ART)

La auto-retransmisión es una función que permite retransmitir un paquete cuando no se recibe un acuse de recibo (ACK). Se utiliza en un sistema de auto-reconocimiento para el transmisor. Cuando el paquete no tuvo acuse de recibo, se puede configurar el número de veces que intentará retransmitir el paquete en los bits ARC del registro SETUP_RETR. El dispositivo transmisor entrará al modo receptor por un periodo breve de tiempo y esperará por el acuse de recibo cada vez que se envía un paquete. El periodo de tiempo de espera se basa en las siguientes condiciones:

- El tiempo de retraso de auto-retransmisión (ARD) haya terminado.
- No coincide la dirección en un lapso de 250µs (o 500µs en el modo de 250Kbps).
- Después de recibir un paquete (CRC correcto o no).

El nRF24L01+ activa el bit TX_DS IRQ cuando el acuse de recibo ha llegado.

Si no llega el acuse de recibo, el nRF24L01+ regresa al modo transmisor después del retraso ARD y retransmite los datos. Esto continúa hasta que llegue el acuse de recibo o se alcance el valor máximo de retransmisiones definidas en los bits ARC del registro SETUP_RETR[17].

El tiempo de retraso de auto-retransmisión (ARD) define el tiempo desde que se termina de transmitir un paquete hasta que empieza una retransmisión. ARD se define en el registro SETUP_RETR en incrementos de 250µs. Recordando que una retransmisión ocurre cuando no se recibe un paquete de acuse de recibo.

Existe una restricción en la longitud del ARD cuando se utilizan acuses de recibo ACK con bloques de datos. El tiempo del ARD nunca debe ser menor a la suma del tiempo de encendido y el tiempo en el aire para el paquete de ACK:

[17] La retransmisión puede ser desde 0 (retransmisión deshabilitada) hasta 15 intentos.

- Para velocidades de transferencia de 2Mbps y una dirección de 5 bytes; 15 bytes será el máximo posible para un bloque de datos en ACK para ARD = 250µs.
- Para velocidades de transferencia de 1Mbps y una dirección de 5 bytes; 5 bytes será el máximo posible para un bloque de datos en ACK para ARD = 250µs.

Con ARD=500µs es tiempo suficiente para cualquier tamaño de paquete de datos en ACK para velocidades de 1Mbps o 2Mbps.

- Para velocidades de 250Kbps y 5 bytes de dirección referirse a la Tabla 3.5.

ARD	Tamaño del paquete de ACK (bytes)
1500µs	Todos los tamaños
1250 µs	≤ 24
1000 µs	≤ 16
750 µs	≤ 8
500 µs	ACK vacío sin bloque de datos

Tabla 3.5.- Máximo número de bytes de datos permitidos en ACK para diferentes retrasos de transmisión a 250Kbps.

Como una alternativa para la retransmisión, es posible ajustar manualmente el número de veces a retransmitir un paquete. Esto se logra utilizando el comando REUSE_TX_PL. El microcontrolador debe iniciar cada transmisión del paquete con un pulso en el pin CE cuando se utiliza éste comando.

3.1.11 Multi-receptor (MultiCeiver™)

La tecnología MultiCeiver™ es una función utilizada en el modo receptor, consiste en seis canales de comunicación (data pipes) con direcciones únicas. Un data pipe consiste de un canal lógico ubicado en el canal de RF. Cada canal tiene su propia dirección física decodificada en el nRF24L01+.

Un nRF24L01+ configurado como receptor primario puede recibir datos de seis diferentes canales en un solo canal de frecuencia tal como se muestra en la Figura 3.7 . Cada canal de datos puede configurarse para tener un comportamiento individual.

Esto permite que hasta seis mRF24L01+ configurados como transmisores puedan comunicarse con un nRF24L01+ configurado como receptor.

Las siguientes características son comunes para todos los canales:

- CRC habilitado/deshabilitado (CRC siempre estará habilitado cuando se tenga activado Enhanced ShockBurst™).
- Esquema CRC.
- Longitud de dirección de RX.

- Canal de Frecuencia.
- Velocidad de transferencia.
- Ganancia.

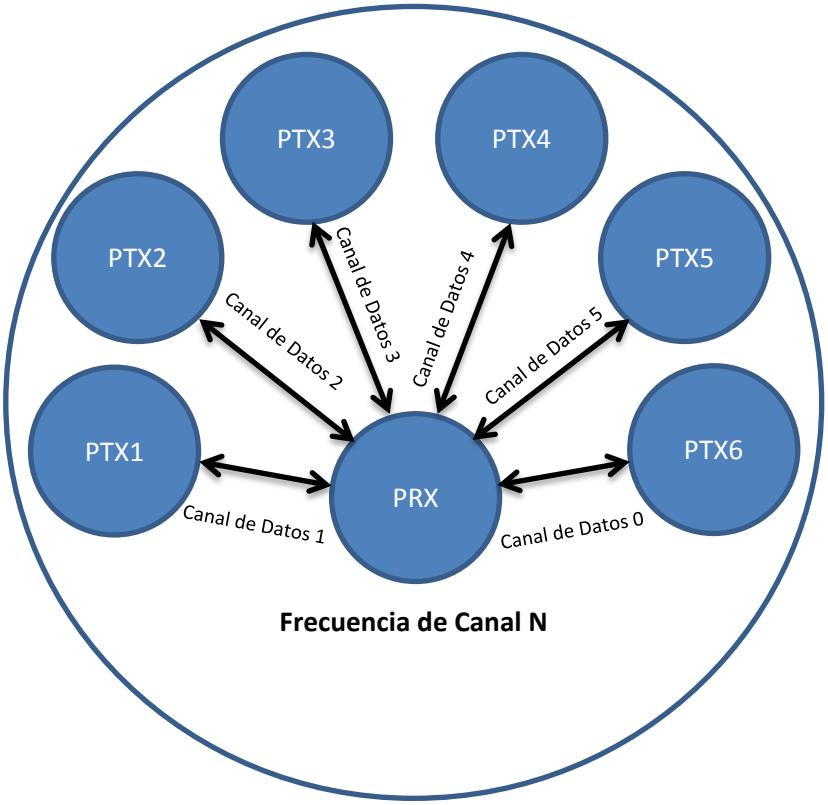

Figura 3.7.- Dispositivo Receptor utilizando MultiCeiver™.

Los canales de datos se habilitan en el registro EN_RXADDR. Por omisión sólo el canal de datos 0 y 1 están activados. Cada dirección de canal de datos se configura en los registros RX_ADDR_PX.

Cada canal puede tener hasta 5 bytes configurables. El canal de datos 0 tiene una dirección de 5 bytes única, mientras que los canales 1 al 5 comparten los cuatro bytes más significativos de la dirección del canal 1. El byte menos significativo debe ser único para cada canal. La Tabla 3.6 es un ejemplo de cómo se nombran las direcciones para los canales 0 al 5.

Canal de datos	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Canal 0 (RX_ADDR_P0)	0xE7	0xD3	0xF0	0x35	0x77
Canal 1 (RX_ADDR_P0)	0xC2 ⬇	0xC2 ⬇	0xC2 ⬇	0xC2 ⬇	**0xC2**
Canal 2 (RX_ADDR_P0)	0xC2	0xC2	0xC2	0xC2	**0xC3**
Canal 3 (RX_ADDR_P0)	0xC2	0xC2	0xC2	0xC2	**0xC4**
Canal 4 (RX_ADDR_P0)	0xC2	0xC2	0xC2	0xC2	**0xC5**
Canal 5 (RX_ADDR_P0)	0xC2	0xC2	0xC2	0xC2	**0xC6**

Tabla 3.6.- Asignación de direcciones para los canales de datos: 0 – 5.

3.1.12 Tiempos

Los tiempos de operación y transmisión de datos del nRF24L01+ varían dependiendo del número de bytes que se envían, del número de bytes de dirección, así como el de la CRC. Los tiempos más importantes son T_{OA}, T_{ACK} y T_{UL}.

$$Tiempo\ en\ aire = T_{OA} = \frac{longitud\ paquete}{velocidad\ de\ transferencia}$$

$$= \frac{8\left[\frac{bit}{byte}\right](1[byte]_{Preámbulo} + 3{,}4\ ó\ 5[bytes]_{Dirección} + N[bytes]_{Datos} + 1\ ó\ 2[bytes]_{CRC}) + 9[bit]_{Control}}{velocidad\ de\ transferencia\left[\frac{bit}{seg}\right]}$$

Ecuación 3.2.- Tiempo en el aire, T_{OA}.

$$Tiempo\ en\ aire\ con\ datos\ en\ ACK = T_{ACK} = \frac{longitud\ paquete}{velocidad\ de\ transferencia}$$

$$= \frac{8\left[\frac{bit}{byte}\right] * (1[byte]_{Preámbulo} + 3{,}4\ ó\ 5[bytes]_{Dirección} + N[bytes]_{Datos\ en\ ACK} + 1\ ó\ 2[bytes]_{CRC}) + 9[bit]_{Control}}{velocidad\ de\ transferencia\left[\frac{bit}{seg}\right]}$$

Ecuación 3.3.- Tiempo en el aire con datos en ACK, T_{ACK}.

$$Tiempo\ de\ carga = T_{UL} = \frac{longitud\ del\ dato}{velocidad\ de\ transferencia\ de\ SPI}$$

$$= \frac{8\left[\frac{bit}{byte}\right] * N[bytes]_{Datos}}{velocidad\ de\ transferencia\ de\ SPI\left[\frac{bit}{seg}\right]}$$

Ecuación 3.4.- Tiempo de carga, T_{UL}.

Un ejemplo de tiempo en el aire y tiempo de carga para un mensaje enviado de: 10 Bytes de longitud, dirección de 5 Bytes, utilizando un CRC de 2 Bytes, una velocidad de 2Mbps y una velocidad de transferencia de la interfaz SPI de 4Mbps, se muestra a continuación:

$$T_{OA} = \frac{8\left[\frac{bit}{byte}\right] * (1[byte]_{Preámbulo} + 5[bytes]_{Dirección} + 10[bytes]_{Datos} + 2[bytes]_{CRC}) + 9[bit]_{Control}}{2{,}000{,}000\left[\frac{bit}{seg}\right]}$$

$$T_{OA} = 76.5\ \mu seg$$

$$T_{UL} = \frac{8\left[\frac{bit}{byte}\right] * 10[bytes]_{Datos}}{4{,}000{,}000\left[\frac{bit}{seg}\right]} = 20\ \mu seg$$

3.1.13 Interfaz de datos y control

La interfaz de datos y control permite el acceso a todas las características del nRF24L01+, ésta interfaz consiste de seis pines (más atrás) compatibles con 5V:

- **IRQ** (Señal de interrupción activada en nivel bajo y controlada por 3 fuentes de interrupción enmascarables).
- **CE** (Señal que activa el circuito en modo RX o TX).
- **CSN** (Señal de interfaz SPI).
- **SCK** (Señal de interfaz SPI).
- **MOSI** (Señal de interfaz SPI).
- **MISO** (Señal de interfaz SPI).

Utilizando los comandos de 1 byte de SPI (ver Tabla 3.7), se pueden activar los FIFO's de datos en el nRF24L01+ o su mapa de registros (ver Apéndice 1) en todos los modos de operación.

La interfaz SPI (como se vio en el Capítulo 2), es una interfaz estándar con la que cuenta el nRF24L01+ y permite velocidades de hasta 10Mbps.

Los comandos SPI se muestran en la Tabla 3.7. Cada comando introducido deberá empezar con una transición de nivel alto a nivel bajo en el pin **CSN**.

En paralelo con el comando enviado en el pin **MOSI**, el nRF24L01+ devuelve el valor actual del registro STATUS por el pin **MISO**.

Los comandos SPI llevan el siguiente formato:

Comando: <Bit más significativo a Bit menos significativo (1 byte) >

Bytes de Datos: <Byte menos significativo a Byte más significativo, Bit más significativo primero para cada byte>

Comando	Valor (binario)	# Bytes de Datos	Operación
R_REGISTER	000A AAAA	1-5	Comando de lectura para los registros. AAAA = dirección del registro en 5 bits.
W_REGISTER	001A AAAA	1-5	Comando de escritura para los registros. AAAA = dirección del registro en 5 bits.
R_RX_PAYLOAD	0110 0001	1-32	Comando de lectura de bloque de datos. Después del comando se lee el bloque de datos 1-32 Bytes.
W_TX_PAYLOAD	1010 0000	1-32	Comando de escritura de bloque de datos. Después del comando se escribe el bloque de datos 1-32 Bytes.
FLUSH_TX	1110 0001	0	Comando que limpia la FIFO de TX, se utiliza en el modo TX.
FLUSH_RX	1110 0010	0	Comando que limpia la FIFO RX, se utiliza en el modo RX.[18]
REUSE_TX_PL	1110 0011	0	Comando que reutiliza el último bloque de datos transmitido, se seguirá reutilizando el paquete hasta que se ejecute el comando W_TX_PAYLOAD o FLUSH_TX.
R_RX_PL_WID	0110 0000	1	Comando de lectura de longitud de bloque de datos. Se debe tener activado el DPL.
W_ACK_PAYLOAD	1010 1PPP	1-32	Comando de bloque de datos para ACK. PPP = canal de datos. (Valores válidos desde 000 hasta 101). Después del comando se escribe el bloque de datos 1-32 Bytes.
W_TX_PAYLOAD_NOACK	1011 0000	1-32	Comando que deshabilita el Auto-reconocimiento para el paquete especificado.
NOP	1111 1111	0	No realiza operación. Puede ser usado para leer el registro STATUS.

Tabla 3.7.- Lista de comandos SPI del NRF24L01+.

[18] No debe utilizarse durante la transmisión de un ACK o el acuse de recibo no será completado.

3.1.14 Mapa de registros

Se puede configurar y controlar la comunicación RF accediendo al mapa de registros a través de la interfaz SPI. La Tabla 3.8 contiene el mapa de registros general del nRF24L01+, para la información completa de cada registro ver el Apéndice 1.

Dirección (Hex)	Nemónico	Descripción
00	CONFIG	Registro de configuración.
01	EN_AA Enhanced Shockburst™	Habilita la función de "Auto reconocimiento" .
02	EN_RXADDR	Habilita las direcciones para el receptor
03	SETUP_AW	Configuración de la longitud de la dirección (común para todas las direcciones).
04	SETUP_RETR	Configuración para la retransmisión automática.
05	RF_CH	Selección del canal de RF.
06	RF_SETUP	Registro para la configuración de RF.
07	STATUS	Registro de estado (En paralelo con el comando SPI aplicado en el pin MOSI, el registro STATUS es regresado por el pin MISO).
08	OBSERVE_TX	Registro de observación de la transmisión.
09	RPD	Detector de potencia recibida.
0A	RX_ADDR_P0	Dirección de recepción para el canal de datos 0. Longitud máxima de 5 Bytes. (El Byte menos significativo se escribe primero. Se escribe el número de bytes definidos en SETUP_AW).
0B	RX_ADDR_P1	Dirección de recepción para el canal de datos 1. Longitud máxima de 5 Bytes (El Byte menos significativo se escribe primero. Se debe escribir el número de bytes definidos en SETUP_AW).
0C	RX_ADDR_P2	Dirección de recepción para el canal de datos 2. Sólo el Byte menos significativo. Los Bytes más significativos son iguales a los de RX_ADDR_P1.

Dirección (Hex)	Nemónico	Descripción
0D	RX_ADDR_P3	Dirección de recepción para el canal de datos 3. Sólo el Byte menos significativo. Los Bytes más significativos son iguales a los de RX_ADDR_P1.
0E	RX_ADDR_P4	Dirección de recepción para el canal de datos 4. Sólo el Byte menos significativo. Los Bytes más significativos son iguales a los de RX_ADDR_P1.
0F	RX_ADDR_P4	Dirección de recepción para el canal de datos 5. Sólo el Byte menos significativo. Los Bytes más significativos son iguales a los de RX_ADDR_P1.
10	TX_ADDR	Dirección de transmisión. Utilizada en modo transmisor (El Byte menos significativo se escribe primero). El registro RX_ADDR_P0 debe ser idéntico al valor de este registro para poder manejar el reconocimiento automático, si se configura el dispositivo con Enhanced Shockburst™ habilitada.
11	RX_PW_P0	Número de Bytes de datos utilizados para el canal de datos de recepción 0 (1 a 32 Bytes. 0 →Desactiva el canal de datos).
12	RX_PW_P1	Número de Bytes de datos utilizados para el canal de datos de recepción 1 (1 a 32 Bytes. 0 →Desactiva el canal de datos).
13	RX_PW_P2	Número de Bytes de datos utilizados para el canal de datos de recepción 2 (1 a 32 Bytes. 0 →Desactiva el canal de datos).
14	RX_PW_P3	Número de Bytes de datos utilizados para el canal de datos de recepción 3 (1 a 32 Bytes. 0 →Desactiva el canal de datos).
15	RX_PW_P4	Número de Bytes de datos utilizados para el canal de datos de recepción 4 (1 a 32 Bytes. 0 →Desactiva el canal de datos).

Dirección (Hex)	Nemónico	Descripción
16	RX_PW_P5	Número de Bytes de datos utilizados para el canal de datos de recepción 5 (1 a 32 Bytes. 0 →Desactiva el canal de datos).
17	FIFO_STATUS	Registro de estado del FIFO.
1C	DYNPD	Habilita la longitud de bloque de datos dinámica.
1D	FEATURE	Registro especial.

Tabla 3.8.- Registros del nRF24L01+.

3.1.15 Interrupciones

El nRF24L01+ cuenta con un pin de interrupción que se activa en nivel bajo (IRQ). El pin IRQ se activa cuando TX_DS, RX_DR o MAX_RT del registro STATUS están puestas en nivel alto. El pin IRQ se resetea cuando se escribe un '1' en el bit correspondiente de la fuente de la interrupción. La interrupción de IRQ es enmascarable, y puede configurarse en el registro CONFIG.

3.2 COMUNICACIÓN Y CONTROL DE RF

Con la información que se presentó en el subtema anterior y el Capítulo 2, se procede a definir la fase de control y comunicación inalámbrica, utilizando el microcontrolador AVR ATMEGA8 y el nRF24L01+ con su interfaz SPI.

3.2.1 Comunicación y control del nRF24L01+ con el microcontrolador AVR

Tal como se comentó anteriormente, la comunicación entre el microcontrolador AVR y el nRF24L01+ se realiza mediante la interfaz SPI, la cuál se definió con detalle en el capítulo 2. A continuación definiremos las librerías creadas para lograr la comunicación y control del dispositivo mencionado, su código e implementación.

Librerías de control para el nRF24L01+

Las funciones de control y archivos de cabecera descritos a continuación para el nRF24L01+ se implementaron en el lenguaje de programación C, utilizando como compilador la plataforma de desarrollo Atmel® Studio 6.

Para no tener que recurrir a los valores hexadecimales de cada comando y registro del nRF24L01+, se generó el archivo de cabecera *NRF24L01.h* (Ver Apéndice 2).

Las funciones de control y manejo del nRF24L01+ se crearon en el archivo fuente *NRF24L01.c* (Ver Apéndice 3), el cuál utiliza las definiciones del archivo de cabecera *NRF24L01.h*, para implementar sus funciones. La librería incluye funciones que permiten configurar los registros de forma automática y otras que requieren de datos dados por el usuario; la documentación pertinente se incluye para cada función en el archivo fuente.

Configuración de registros

En general las funciones utilizadas en el archivo fuente *NRF24L01.c* se derivan de la función: `unsigned char SPI_Send(unsigned char tData)`. Esta función permite enviar y recibir datos utilizando la interfaz SPI, utiliza el parámetro "*tData*", un dato del tipo carácter sin signo que equivale al byte que se enviará por el pin MOSI, la función a su vez, regresa un valor del mismo tipo que representa el byte recibido por el pin MISO.

Previamente se llama la función: `SPI_Init()`. La cuál modifica los registros internos del AVR para configurar y habilitar la interfaz SPI y adecuarse a las características del nRF24L01+. Además se requiere configurar los pines ya sea como entrada o salida digital, según corresponda, propios a la interfaz SPI (SCK, MISO, MOSI, CSN, CE). Este proceso se hace una sola vez y debe hacerse antes de enviar cualquier dato por el SPI.

Para acceder y configurar los registros del nRF24L01+ es necesario aplicar el comando correspondiente de lectura o de escritura (`R_REGISTER`, `W_REGISTER`) de la lista de comandos (ver Tabla 3.7), colocando en el valor de los últimos 4 bits del comando, el valor binario del registro (Ver Tabla 3.8 y Apéndice 1) al que se desea escribir o leer.

Un ejemplo de escritura y lectura de un registro utilizando las librerías *NRF24L01.c* y *NRF24L01.h*, se muestra a continuación:

```
int main(void)
{
	/* Configura como salidas y entradas los pines correspondientes.
	*  CE, CSN, MOSI, SCK, MISO, IRQ deberán declararse en el archivo
	*  NRF24L01.h
	*/
	DDRB |= (1<<CE) | (1<<CSN) | (1<<MOSI) | (1<<SCK);
	DDRB &= ~(1<<MISO);
	DDRD &= ~(1<<IRQ);
```

```
        /*Inicializa la interfaz SPI*/
        SPI_Init();
        /* Coloca en cero el pin CSN del nRF24L01+*/
        RF_Select;
        /* Envia el comando de escritura del registro RF_CH*/
        SPI_Send(W_REGISTER|RF_CH);
        /* Escribe el registro RF_CH con el valor 0x02*/
        SPI_Send(0x02);
        /* Coloca en uno el pin CSN del nRF24L01+*/
        RF_Select_Not;
        /* Declara una variable para almacenar el valor de lectura*/
        unsigned char Canal;
        /* Coloca en cero el bit CSN del nRF24L01+*/
        RF_Select;
        /* Envia el comando de lectura del registro RF_CH*/
        SPI_Send(R_REGISTER|RF_CH);
        /* Lee el registro y almacena el resultado en la variable "Canal"
        (Se debe enviar cualquier dato por el SPI aunque sólo estemos
        leyendo)*/
        Canal = SPI_Send(0xFF);
        /* La variable "Canal" tendrá el valor 0x02*/
        /* Coloca en uno el bit CSN del nRF24L01+*/
        RF_Select_Not;
}
```

El proceso de lectura y escritura de registros del nRF24L01+ se ilustra en la Figura 3.8.

Adicionalmente se generaron funciones predefinidas que configuran al nRF24L01+ de forma automática, una de ellas es la función: `void NRF24L01_Init(bool MODE, unsigned char ADDR_P0[5], unsigned char ADDR_P1[5], uint8_t RF_CHANNEL, uint8_t P0_LENGTH, uint8_t P1_LENGTH, uint8_t RF_SPEED)`. La cuál inicializa el nRF24L01+ habilitando los bloques de datos 0 y 1 y el auto-reconocimiento hasta con 5 retransmisiones con un ARD de 1 ms.

Adicionalmente se debe configurar: el modo con el parámetro "`MODE`" (true para RX, false para TX), la dirección de transmisión "`ADDR_P0`" (5 Bytes), la dirección del receptor "`ADDR_P1`" (5 Bytes), el canal de radiofrecuencia "`RF_CHANNEL`" (0-127), la longitud de los bloques de datos a enviar y recibir "`P0_LENGTH`" y "`P1_LENGTH`" (0-32 Bytes) y la velocidad de transferencia de datos "`RF_SPEED`" (1 → 2Mbps, 2→ 1Mbps y 3→ 250Kbps).

Existen más funciones de asignación para configurar aspectos individuales del dispositivo como el canal de frecuencia, el modo, velocidad, etc. Todas vienen debidamente documentadas en el mismo archivo fuente *NRF24L01.c*, para mayor referencia véase el Apéndice 3.

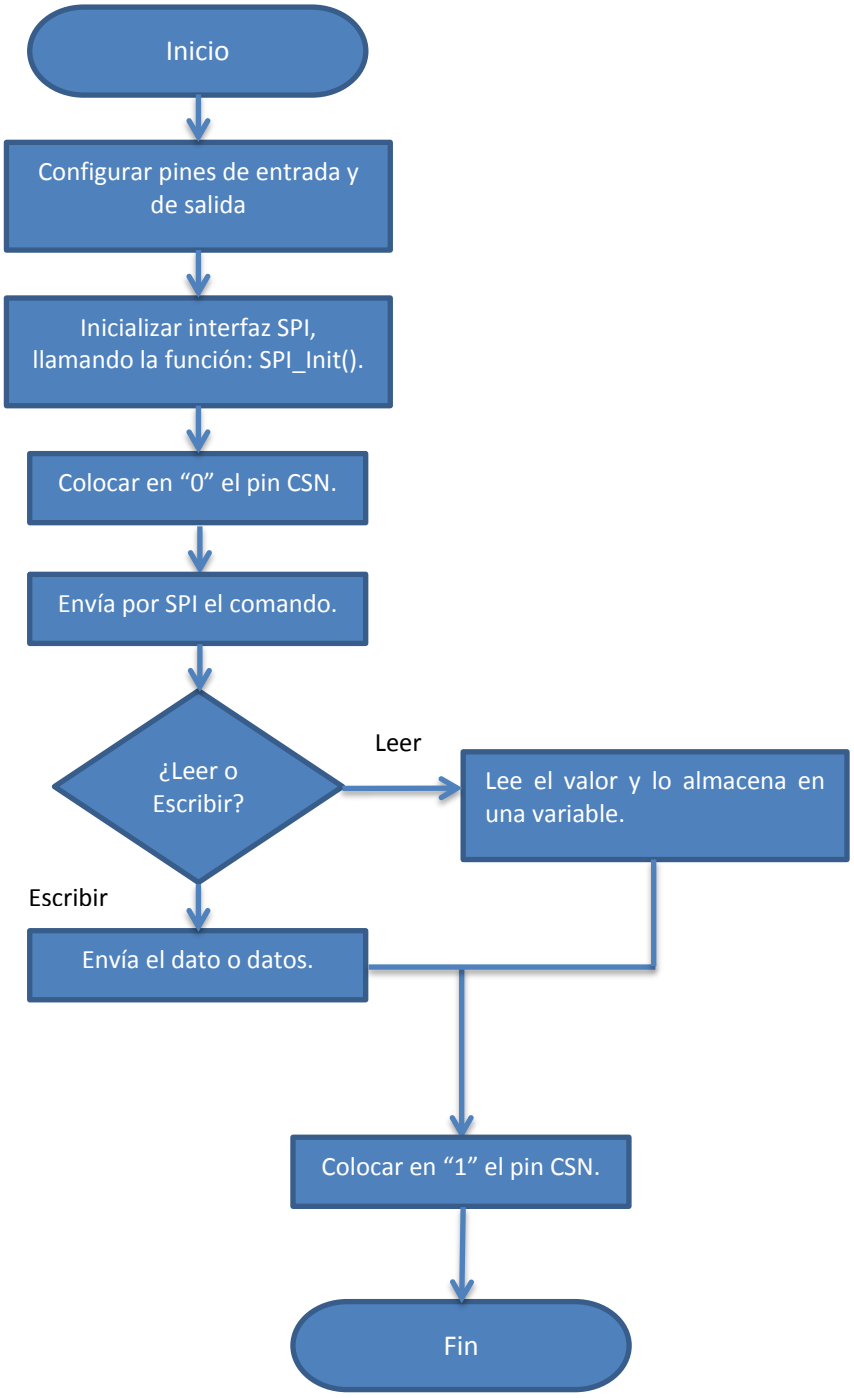

Figura 3.8.- Proceso de lectura y escritura de los registros del nRF24L01+.

Lectura y envío de datos RF

Para el envío y recepción de datos estáticos en el dispositivo nRF24L01+ se utilizan básicamente los comandos `R_RX_PAYLOAD`, para leer los datos recibidos, y `W_TX_PAYLOAD` para escribir los datos que se desean enviar por RF. Previamente se debe haber seleccionado la longitud de los bloques de datos. En el caso de bloques de datos dinámicos el proceso es similar pero en vez de asignar un bloque de datos de longitud definida, se envían datos de longitud variable y se debe leer la cantidad de datos recibidos con el comando `R_RX_PL_WID`.

En la librería se incluyen las funciones que simplifican el proceso de transferencia y recepción de datos. Existen cuatro funciones principales que permiten enviar y recibir datos RF; dependiendo de si se envían bloques de datos estáticos o dinámicos se deberán utilizar las funciones correspondientes.

Las funciones de lectura y escritura para bloques de datos estáticos son: `Read_RF_Data(unsigned char readData[])` y `Transmit_RF_Data(unsigned char writeData[])`, éstas funciones permiten leer y escribir datos sin la necesidad de incluir ningún registro, simplemente se proporciona un arreglo de caracteres que vendrá siendo para el caso de "Transmit_RF_Data", el valor de los datos a enviar por RF; y para "Read_RF_Data", el parámetro donde se almacenarán los datos de lectura RF (el valor se asigna por referencia). El tamaño del arreglo no debe exceder los 32 Bytes y debe ser igual al número de Bytes configurados en la función `Set_Rx_Length(uint8_t P1_LENGTH)`, para lectura; y `Set_Tx_Length(uint8_t P0_LENGTH)`, para envío de datos.

Las funciones de lectura y escritura para bloques de datos dinámicos son: Send_Dynamic_Data_RF(unsigned char writeData[], unsigned int8 dataLength) y unsigned int 8 Receive_Dynamic_Data_RF(unsigned char readData[]). Para el caso de "Send_Dynamic_Data_RF", el parámetro "dataLength" corresponde al número total de datos a enviar, si el número de datos a enviar es igual al tamaño del arreglo se puede utilizar como valor del parámetro la función "sizeof" aplicada al valor del arreglo; para el caso de la función "Receive_Dynamic_Data_RF", el arreglo "readData" deberá ser del tamaño suficiente como para almacenar hasta 32 posibles Bytes de llegada, la función a su vez, regresa un valor entero sin signo de 8 bits que representa el número total de Bytes recibidos[19] (correspondientes al comando `R_RX_PL_WID` del nRF24L01+).

Antes de poder utilizar cualquiera de las dos funciones anteriores se debe activar la característica Dynamic Payload Length del nRF24L01+, esto se hace llamando a la función Enable_DPL().

Si se desea cambiar la dirección de transmisión (para cualquier tipo de bloques de datos), se debe llamar a la función: `Set_TX_Address(unsigned char ADDR_P0[5])` y para la dirección de recepción, la función: `Set_RX_Address(unsigned char ADDR_P1[5])`.

Para el caso de la lectura de datos de RF se debe considerar que exista un dato en el FIFO de RX, esto se puede hacer de dos maneras:

[19] Si el número de Bytes recibidos es mayor a 32 la función automáticamente desechará los datos recibidos utilizando el comando FLUSH_RX del nRF24L01+.

- Configurar el pin IRQ para activar una interrupción de nivel bajo cuando se reciba un dato.
- Muestrear continuamente el registro STATUS para determinar si se tiene un dato o no.

El primer método puede realizarse con la función incluida Enable_RX_IRQ() para activar la interrupción por recepción de datos, o la función Enable_All_IRQ() para activar todas las interrupciones por IRQ.[20]

La segunda forma es la de leer constantemente el registro STATUS hasta que el bit RX_DR tenga un valor de 1. Para leer el registro STATUS se llama la función Read_Status().

Una vez que se tiene un dato en el FIFO de RX, el procedimiento de lectura consta de 4 pasos:

1. Leer los datos con la función `Read_RF_Data(unsigned char readData[])`.
2. Limpiar el bit RX_DR del registro STATUS con la función `Clean_Status()`.
3. Leer el registro FIFO_STATUS con la función Get_Fifo_Status() y revisar que no existan más datos pendientes por leer en el FIFO de RX.
4. Si existen más datos en el FIFO de RX, repetir desde el paso 1.

[20] Las funciones Enable_RX_IRQ(), Enable_TX_IRQ() y Enable_All_IRQ(), sólo configuran el comportamiento del pin IRQ del NRF24L01+ más no activa la interrupción del AVR. Si se desea utilizar como una interrupción del AVR, se debe configurar en el programa principal.

Capítulo 4
Desarrollo de la Interfaz USB

En el siguiente capítulo se abordará el desarrollo e implementación de la interfaz USB utilizando un microcontrolador AVR. Se retoman algunos conceptos explicados en el marco teórico y se explican a detalle las herramientas que se requirieron para dicha interfaz.

Tal como se trató en el capítulo 2, se utilizó para el proyecto un microcontrolador AVR ATMEGA 8. A continuación se verán algunos puntos importantes sobre la interfaz USB y un dispositivo AVR.

La comunicación entre una PC y un dispositivo AVR ATMEGA 8 a través del protocolo USB no puede realizarse directamente como en otro tipo de dispositivos, tales como los PIC18F4550 de Microchip o la serie AT90USB de ATMEL, en donde éstos ya cuentan con un controlador interno para manejar el protocolo USB, lo cuál permite la comunicación directa.

Entonces, ¿Por qué utilizar un microcontrolador sin soporte USB directo? La respuesta se deriva de dos factores que fueron el objetivo de éste proyecto: bajo costo y simplicidad.

Los dispositivos antes mencionados llegan a ser 3 o hasta 5 veces más caros que el dispositivo propuesto: el ATMEGA 8. Además de que algunos dispositivos no permiten su fácil implementación, es decir, utilizan empaquetados Quad Flat Package (QFP) o Thin Quad Flat Package (TQFP) que limitan ampliamente su uso a menos que se cuente con la tecnología adecuada o en su defecto tener que comprar kits armados que son mucho más costosos.

Utilizando un microcontrolador como el ATMEGA 8 que incluye un empaquetado DIP, además de otras características, reducimos el costo y volvemos más accesible el sistema, sin embargo, permanece un problema: no se cuenta con la interfaz USB directamente desde el dispositivo, por lo que el siguiente paso para el sistema es el de definir una solución que permita implementar la interfaz USB mediante software y que no implique añadir mucho hardware para que el costo permanezca siendo accesible. Tal solución se encontró en el firmware de uso libre *V-USB* creado por Christian Starkjohann [7]. La cuál explicaremos con mayor detalle en el siguiente apartado.

4.1 V-USB: VIRTUAL USB PARA MICROCONTROLADORES AVR

V-USB es una implementación por software de un dispositivo USB de baja velocidad (USB 1.1) para microcontroladores AVR, sin la necesidad de requerir un chip adicional.

Las ventajas de utilizar el firmware V-USB sobre algún dispositivo con el hardware integrado de USB son las siguientes:

- Los microcontroladores AVR estándar son fáciles de conseguir.
- La mayoría de los microcontroladores con soporte USB sólo están disponibles en tecnología de montaje de superficial (SMD), haciéndolos casi imposibles para estudiantes o desarrolladores aficionados.
- V-USB viene con un Vendor y Product Id.[21] de uso libre.
- Los microcontroladores AVR son más rápidos que la mayoría que tienen integrados el USB y son de menor costo.

El firmware ofrece la ventaja de utilizar poco hardware para su implementación, además que permite integrar la interfaz USB a casi cualquier AVR. Inicialmente se pretendía utilizar el ATTINY2313 por su bajo costo, sin embargo, debido a que su memoria flash es tan solo de 2KB, al

[21] Identificadores de 16 bits del dispositivo USB.

cargar el firmware de V-USB, éste dejaba poca memoria para el programa principal, por lo que se optó por un microcontrolador AVR de mayor capacidad: el ATMEGA8.

4.1.1 Características de V-USB

El firmware V-USB ofrece una comunicación USB a través de diferentes endpoints[22], utiliza el endpoint 0 para transferencias de control, dos configurables para transferencias de entrada del tipo bulk o del tipo interrupción y hasta otros 7 para transferencias de salida del tipo bulk o del tipo interrupción.

Trabaja en cualquier microcontrolador AVR que contenga por lo menos 2KB de memoria flash, 128 bytes de RAM y un reloj de por lo menos 12 MHz. Sólo utiliza una interrupción del AVR.

V-USB se encarga de realizar todos los paquetes del tipo acuse de recibo, control, sincronización (ACK, CRC, SYNC), etc., que se utilizan de manera regular al enviar o recibir paquetes en la comunicación USB.

Un aspecto importante que se debe tomar en cuenta antes de empezar a utilizar el firmware, es el Product Id y Vendor Id, estos dos se encargan de identificar el dispositivo USB y son dos valores únicos de 16 bits que se asignan a éste (similar al MAC address). Estos valores se asignan desde la librería *usbconfig.h*, sin embargo, no deben de asignarse valores arbitrarios ya que éstos podrían pertenecer a una empresa u otra organización, es por eso que deben utilizarse valores únicos que son proporcionados por la organización que se encarga de gestionar dichos permisos: [www.usb.org] y que pueden adquirirse desde su página de internet en la siguiente ruta: [http://www.usb.org/developers/vendor/] o bien, de otras empresas que tengan derechos sobre las licencias, pero que de igual forma fueron otorgadas por [www.usb.org].

Una ventaja de V-USB es que cuenta con un par de identificadores de uso libre, que permiten ser asignados al dispositivo USB, siempre y cuando no se comercialice el producto final o se distribuya.

Las características eléctricas de la interfaz USB, es decir, los niveles de voltaje, son diferentes que los de un microcontrolador AVR, es por eso que se debe añadir hardware para ajustar dichos niveles. Christian Starkjohann [7] ofrece tres opciones de implementación para éste propósito:

- Regulador de voltaje de 3.3V. (Ver Figura 4.1)
- Dos diodos simples (rectificadores de silicio) y 4 resistencias. (Ver Figura 4.2)
- Dos diodos zener de 3.3V y 3 resistencias. (Ver Figura 4.3)

El firmware V-USB dispone de varios archivos que deben incluirse al proyecto en C. Siendo de los más importantes el archivo de cabecera *usbconfig.h* (Ver Apéndice 5). En éste archivo se configuran los endpoints, el product ID, el vendor Id y la configuración de hardware (pines D+ y D-), entre otras características.

Las librerías deberán modificarse de acuerdo al microcontrolador AVR utilizado así como la configuración deseada del dispositivo USB.

[22] Nodo de comunicación que sirve como memoria de bloque de datos (buffer) en el controlador.

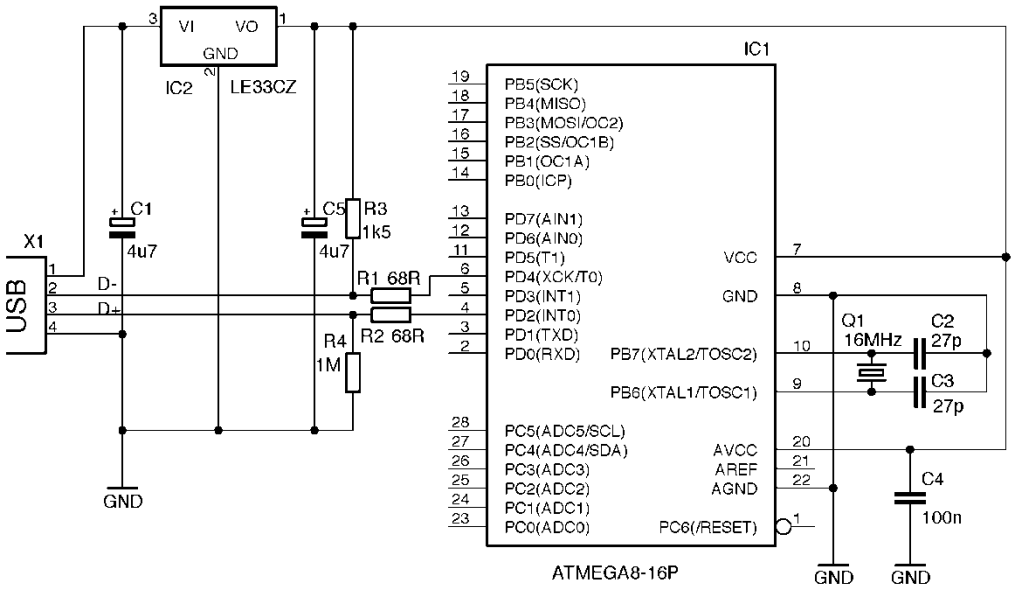

Figura 4.1.- Diagrama de conexiones V-USB utilizando un microcontrolador ATMEGA 8 y un regulador de voltaje de 3.3V. [7]

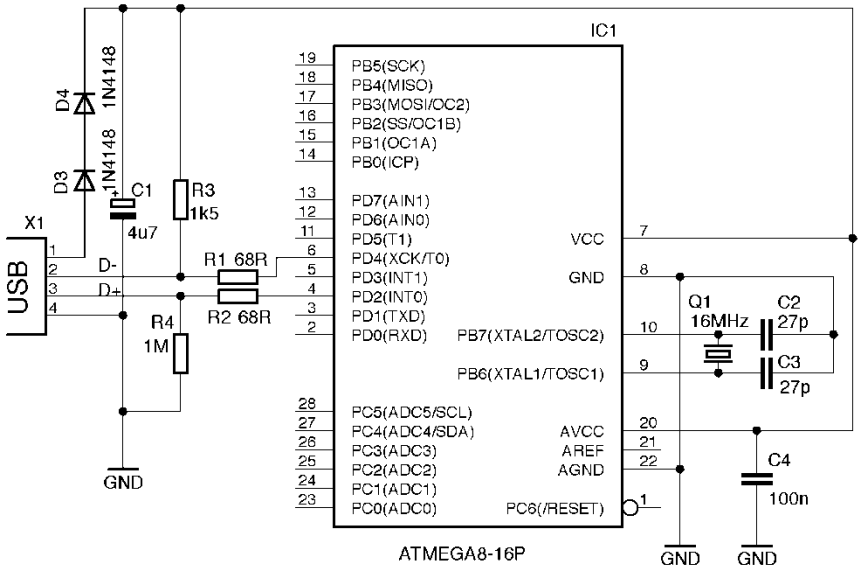

Figura 4.2.- Diagrama de conexiones V-USB utilizando un microcontrolador ATMEGA 8 y diodos rectificadores. [7]

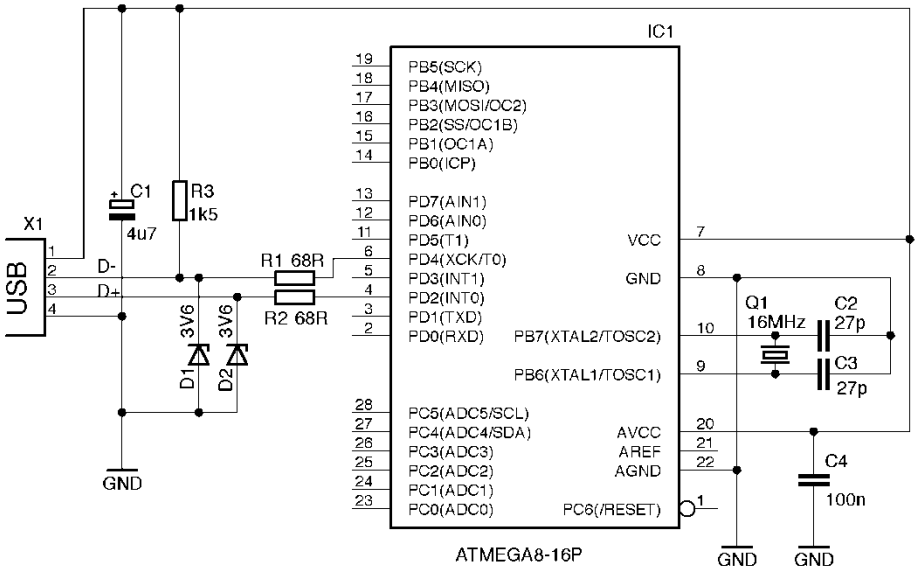

Figura 4.3.- Diagrama de conexiones V-USB utilizando un microcontrolador ATMEGA 8 y diodos zener. [7]

Se optó por utilizar el diagrama mostrado en la Figura 4.3 para el proyecto, debido a su bajo costo y simplicidad, sin embargo, bien se puede utilizar otro de los arreglos mostrados anteriormente.

Para implementar la interfaz USB utilizando el firmware USB se deben tomar en cuenta las siguientes restricciones:

- Sólo se puede utilizar un cristal de cuarzo de: 12 MHz, 12.8 MHz, 15 MHz, 16 MHz, 16.5 MHz, 18 MHz o 20 MHz.
- El pin **"D+"** del protocolo USB debe estar conectado (en cualquiera de los 3 arreglos), al pin de interrupción **"INT0"** del microcontrolador AVR.
- El pin "D-" puede conectarse a cualquier otro pin del microcontrolador siempre y cuando se configure en la librería *usbconfig.h* (Véase Apéndice 5).
- El microcontrolador deberá tener por lo menos 2 KB de memoria para almacenar el programa.

4.1.2 Comunicación por transferencias de control en V-USB

Las transferencias de control son la funcionalidad básica de la comunicación USB, utiliza una trama definida como "mensaje de control" la cuál se envía típicamente a través del endpoint 0, estos mensajes de control constan de un paquete de 8 Bytes denominado "Setup" y un bloque de datos opcional, de longitud variable, que puede ser de hasta 64 Bytes de información y permite ser enviado del host al dispositivo o del dispositivo al host[23]. Es importante destacar que es el host el único encargado de enviar los mensajes de control, el dispositivo sólo se encarga de recibir y

[23] El host se define como el aparato al cuál es conectado el dispositivo USB, para este caso lo definiremos como la computadora o PC.

procesar los mensajes de control recibidos del host, independientemente de que el mensaje de control requiera de un bloque de datos.

La estructura del paquete "Setup" (Ver Apéndice 6), está definida en la hoja de especificaciones del USB revisión 2.0, el cuál indica que la primera entrada "bmRequestType", es un Byte que contiene la dirección del bloque de datos ya sea del dispositivo al host (control-in) o del host al dispositivo (control-out), además también describe el tipo de mensaje (Standard, Class o Vendor) y el destino del mensaje (dispositivo, interfaz, endpoint u otros).

La segunda entrada "bRequest", está dada en 1 Byte e identifica la petición. Para el caso de mensajes de control del tipo "Vendor", éste parámetro se utiliza indistintamente para poder asignarle alguna función de acuerdo al valor recibido.

Las entradas "wValue" y "wIndex" de igual forma que "bRequest", se utilizan de forma arbitraria para mensajes de control del tipo "Vendor" y son de 2 Bytes de longitud cada uno. Un uso común en mensajes del tipo "Vendor" es el de asignarles un dato que pueda utilizar la función establecida a "bRequest".

Finalmente, la entrada "wLength" define en 2 Bytes la longitud del bloque de datos (control-in o control-out), pudiendo ser cero (si no se enviarán o recibirán datos).

La estructura del paquete "Setup" utilizada en la librería V-USB está declarada en *usbdrv.h* que está definida como "usbRequest", y es la siguiente:

```
typedef struct usbRequest{
      uchar bmRequestType;
      uchar bRequest;
      usbWord_t wValue;
      usbWord_t wIndex;
      usbWord_t wLength;
}usbRequest_t;
```

Cuando el host envía mensajes de control, la función usbFunctionSetup() de la librería V-USB es llamada. Dicha función contiene un apuntador de 8 Bytes que corresponde a la estructura usbRequest.

Un ejemplo de un mensaje de control utilizando la librería V-USB, se muestra a continuación, en éste caso se utiliza el valor del parámetro "bRequest" para asignar la función de encender o apagar un LED de acuerdo al valor dado en "wValue".

```
usbMsgLen_t usbFunctionSetup(uchar setupData[8])
{
      // Obtiene los valores del Setup y los asigna al apuntador rq.
      usbRequest_t *rq = (void *)setupData;
      // Evalúa el Byte bRequest para poder asignar la función
      // correspondiente.
      switch(rq->bRequest){
            // Si bRequest = 1, realizará la function setLED.
```

```
            case 1:
            // Realiza la función setLED y utiliza como parámetro el
            // valor del Byte menos significativo recibido en wValue.
            setLED(rq->wValue.bytes[0]);
            // No se envía ningún bloque de datos al host.
            return 0;
      }
      // Ignora todas las peticiones diferentes a 1.
      return 0;
}
```

Mensajes de control con bloques de datos

Como se mencionó anteriormente, los mensajes de control permiten tener un bloque de datos. Los mensajes de control con bloques de datos se refieren siempre respecto al host con el dispositivo, por lo que si se utiliza el término de bloques de datos de entrada se refiere a datos que recibe el host del dispositivo y los bloques de datos de salida son datos que envía el host hacia el dispositivo.

Se denominan los mensajes de control con bloques de datos de entrada como “control—in” y los mensajes de control bloques de datos de salida como “control—out”. La forma de procesar los bloques de datos en V-USB es a través de la misma función usbFunctionSetup() y la función usbFunctionWrite() (para el caso de transferencias “control – out”).

Mensajes de control con bloques de datos de entrada (control—in)

La forma en que el dispositivo envía transferencias “control—in” en V-USB es a través de la misma función usbFunctionSetup(). La forma más simple es utilizar un buffer estático en la RAM que almacene todos los datos y regresar el valor en la función usbFunctionSetup().

Un ejemplo de un mensaje “control—in” utilizando un buffer en la RAM, se muestra a continuación:

```
// Se declara un buffer que almacene los datos que se envían al host.
static uchar buffer[64];

// Constante cualquiera que define la petición en “bRequest”.
#define     VENDOR_RQ_READ_BUFFER   1

usbMsgLen_t usbFunctionSetup(uchar setup Data[8])
{
      usbRequest_t *rq = (void *)setupData;
      switch(rq->bRequest){
            case VENDOR_RQ_READ_BUFFER:
```

```
            // Se declara una variable que contenga la longitud del
            // buffer.
            usbMsgLen_t len = sizeof(buffer);
            // Si el host requiere menos datos que el tamaño del buffer,
            // entonces ajusta su valor a la cantidad requerida por el
            // host.
            if(len > rq->wLength.word)
            len = rq->wLength.word;
            // Carga el contenido del buffer al apuntador utilizado
            // en el Firmware V-USB.
            usbMsgPtr = buffer;
            // Le dice al Firmware cuantos bytes debe enviar.
            return len;
      }
      // Ignora todas las peticiones desconocidas.
      return 0;
}
```

Mensajes de control con bloques de datos de salida (control –out)

Si el dispositivo recibe datos del host a través del mensaje de control se utiliza la función usbFunctionWrite(). Para poder utilizar esta función, es necesario editar el archivo de cabecera *usbconfig.h,* se debe definir el valor USB_CFG_IMPLEMENT_FN_WRITE a un valor de '1'.

La forma en que se llama a la función usbFunctionWrite() es a través de la función usbFunctionSetup(), para esto se utiliza la constante USB_NO_MSG como parámetro de retorno, de esta forma automáticamente se llamará a la función usbFunctionWrite(). La forma más simple de recibir datos es la de utilizar un buffer estático guardado en la RAM, éste debe ser de longitud suficiente para recibir los datos del host, se recomienda tenerlo al valor máximo posible (64 Bytes).

El siguiente ejemplo muestra una forma de recibir datos del host en un mensaje de control y almacenarlos en un buffer estático:

```
// Constante cualquiera que define la petición en "bRequest".
#define     VENDOR_RQ_WRITE_BUFFER  1

// Se declara un buffer que almacene los datos que se reciben del host.
static uchar buffer[64];
// Variables utilizadas para leer los Bytes de información.
static uchar currentPosition, bytesRemaining;

usbMsgLen_t usbFunctionSetup(uchar setupData[8])
{
      // Obtiene los valores del Setup y los asigna al apuntador rq.
      usbRequest_t *rq = (void *)setupData;
      // Evalúa el Byte bRequest para poder asignar la función
      // correspondiente.
```

```
	switch(rq->bRequest){
		// Si bRequest = 1, Leerá los datos recibidos del Host.
		case VENDOR_RQ_WRITE_BUFFER:
		// Inicializa la posición.
		currentPosition = 0;
		// Almacena la cantidad de datos requerida
		bytesRemaining = rq->wLength.word;
		if(bytesRemaining > sizeof(buffer))
		bytesRemaining = sizeof(buffer);
		// Llama a la función usbFunctionWrite().
		return USB_NO_MSG;
	}
	return 0;
}



uchar usbFunctionWrite(uchar *data, uchar len)
{
	uchar i;
	// Evalúa si éste es el último bloque de datos.
	if(len > bytesRemaining)
	// Limita a la cantidad que podemos almacenar.
	len = bytesRemaining;
	bytesRemaining -= len;
	// Lee los datos y los almacena en el buffer.
	for(i = 0; i < len; i++)
	buffer[currentPosition++] = data[i];
	// Regresa 1 si obtuvimos todos los datos.
	return bytesRemaining == 0;
}
```

4.2 COMUNICACIÓN Y CONTROL DE LA INTERFAZ USB

A continuación se definirá la forma en que el microcontrolador AVR, el cuál denominaremos como dispositivo, se comunicará con la computadora, la cual definiremos como host, con un sistema operativo Windows.

Retomaremos lo visto en la sección anterior para definir la forma de comunicarnos con el Firmware V-USB y el sistema operativo. Es importante hacer notar que para este proyecto se utilizan transferencias de control del tipo “Vendor”, de tal forma que, es necesario definir un controlador (driver) que permita al sistema operativo detectar nuestro dispositivo y realizar la comunicación adecuada.

4.3.1 Drivers USB para Windows

Se define como controlador (driver) de un dispositivo, a un componente de software que permite a los programas de aplicación tener acceso al hardware. Una de las principales funciones de un driver es ocultar a las aplicaciones todos los detalles referentes a la conexión física, señales y protocolos requeridos para poder comunicarse con el dispositivo.

Existen varias opciones de poder definir los drivers en un sistema operativo Windows, algunas requieren de programación avanzada a través de código núcleo (kernel) y otras ofrecen una forma más simple: mediante una aplicación que genera los drivers automáticamente utilizando como referencia el "Vendor" y "Product" Id.

La solución se encontró en la librería de uso libre Libusb [8], en su versión para Windows libusb-win32 [9], la cual se definirá con mayor detalle a continuación.

4.3.2 Libusb-win32

Libusb-win32 [9] es una implementación de la librería USB Libusb 0.1 [8], para los sistemas operativos Windows (Windows 98 SE, Windows ME, Windows 2000, Windows XP, Windows Vista y Windows 7). La librería permite al usuario acceder a una gran cantidad de dispositivos USB en Windows mediante una forma genérica sin la necesidad de escribir ninguna línea de código núcleo (kernel).

Características:

- Puede usarse como filtro de algún Driver existente para dispositivos que ya cuentan con un driver definido, esto permite a libusb-win32 comunicarse con dispositivos USB previamente instalados.

- Puede usarse como un Driver normal para dispositivos en los cuales no exista algún Driver definido (dispositivos propios, hardware de desarrollo, etc.). También permite reemplazar el Driver existente por un nuevo Driver libusb-win32.

- Soporta todas las transferencias USB: Control, bulk, interrupción e isócronas.

- Soporta todas las peticiones de dispositivo estándar (mensajes de control) descritas en el capítulo 9 de la especificación USB.

- Soporta mensajes de control del tipo Vendor.

Instalación:

- Descargar la última versión disponible de libusb-win32 de la página de internet: [http://sourceforge.net/projects/libusb-win32/files/]

- Extraer el archivo (libusb-win32-device-bin-x.x.x.x.zip) a la computadora que se desea instalar.

- Utilizar el programa ejecutable INF-Wizard, el cuál generará el archivo INF de acuerdo a los valores de "Vendor" y "Product" Id., estos se pueden introducir manualmente o si el dispositivo está conectado éste podrá visualizarse desde el instalador y ser seleccionado para generar los respectivos drivers (Ver Figura 4.4).

- Una vez que se generen los drivers, se podrán instalar inmediatamente a través del mismo instalador, o bien, instalar los drivers manualmente utilizando los archivos generados.

    - Si se opta por la instalación manual se debe acceder al administrador de dispositivos de Windows, ubicar el dispositivo, seleccionar la opción de instalar software de controlador mediante una ubicación desde la PC, seleccionar la ubicación donde se generaron los drivers e instalarlos.

- Desconectar el dispositivo y reconectarlo.

- Verificar en el administrador de dispositivos de Windows que el driver esté instalado correctamente. Adicionalmente puede correrse el programa de prueba (testlibusb-win.exe) incluido en el folder "bin", el cual permite visualizar los descriptores del dispositivo (Ver Figura 4.5).

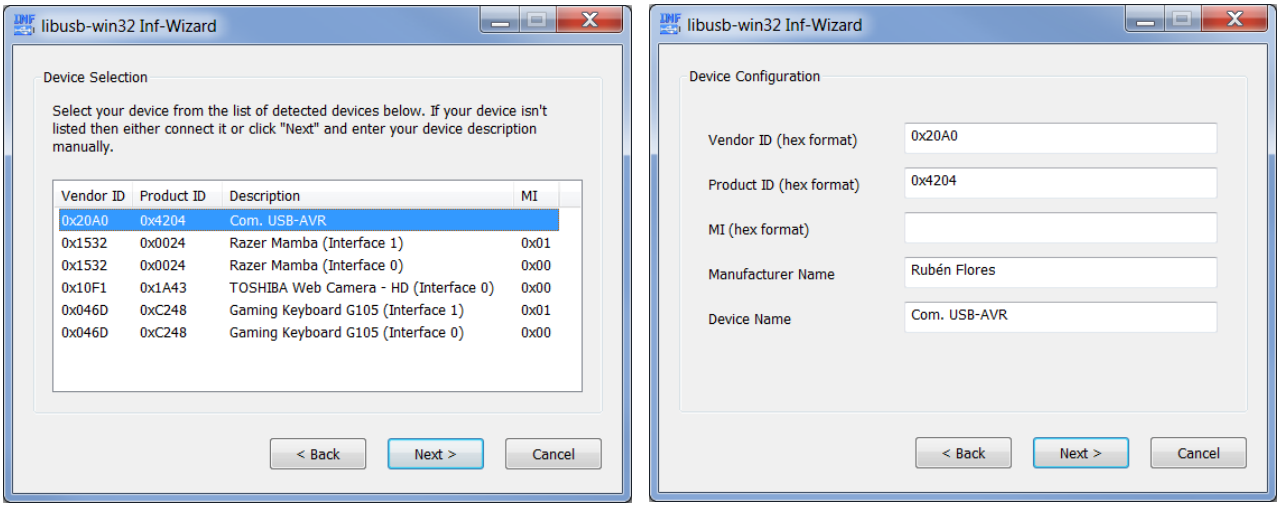

Figura 4.4.- Generación de Drivers utilizando libusb-win32 INF-Wizard.

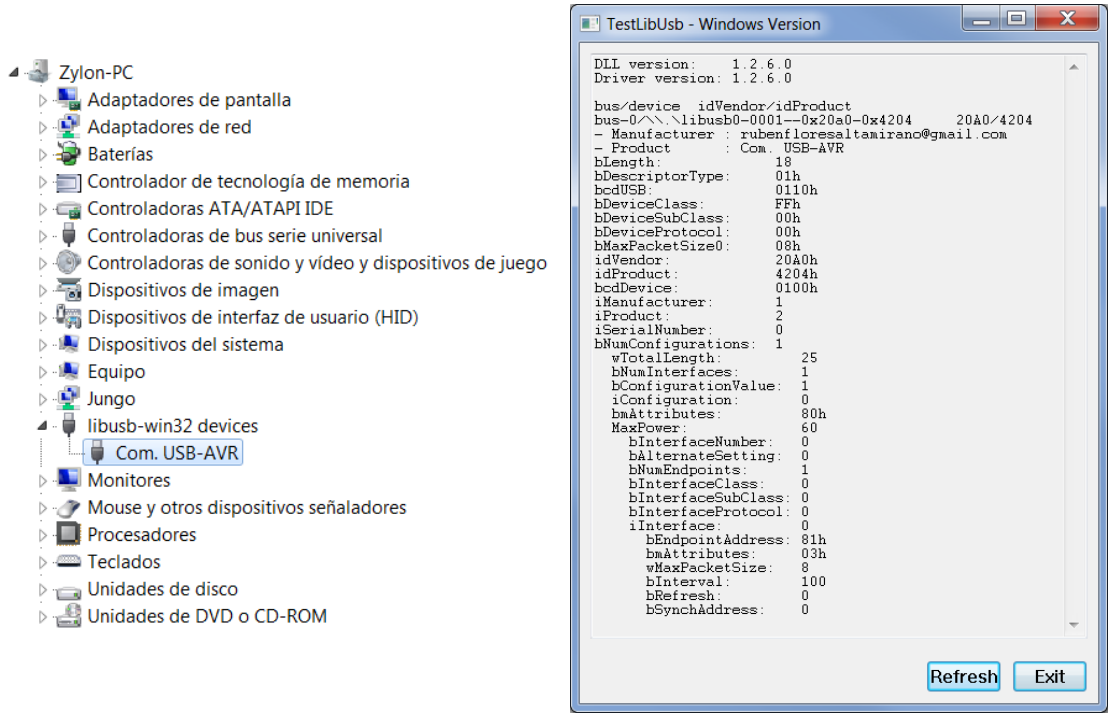

Figura 4.5.- Comprobación de instalación de drivers libusb-win32.

Comunicación USB utilizando Libusb-win32

Libusb-win32 implementa una serie de funciones definidas en lenguaje C++ que permiten la comunicación USB, la documentación completa puede encontrarse en: [http://sourceforge.net/apps/trac/libusb-win32/wiki/libusbwin32_documentation].

Las funciones principales son:

- usb_init()
- usb_find_busses()
- usb_find_devices()
- usb_get_busses()
- usb_open()
- usb_close()
- usb_set_configuration()

Las funciones para transferencias de control son las siguientes:

- usb_control_msg()
- usb_get_string()
- usb_get_string_simple()
- usb_get_descriptor()
- usb_get_descriptor_by_endpoint()

La comunicación entre un dispositivo USB y el host mediante Libusb-win32 consiste en:

- Identificar el dispositivo mediante sus descriptores.
- Abrir el dispositivo
- Realizar transferencias USB: control, interrupción, isócronas o bulk, a algún endpoint del dispositivo.

Antes de que cualquier comunicación pueda ocurrir entre el host y el dispositivo, el dispositivo debe ser encontrado. Esto se logra al buscar todos los buses y después buscar todos los dispositivos dentro de los buses, la aplicación deberá realizar un bucle manual a través de todos los buses y todos los dispositivos y realizar una comparación para encontrar el dispositivo al cuál se requiere comunicar, la página de libusb-win32: [http://sourceforge.net/apps/trac/libusb-win32/wiki/libusbwin32_examples], ofrece una gran gama de ejemplos y aplicaciones. Se recomienda al lector que desee tener conocimiento a mayor detalle de las funciones que implementa libusb-win32, lea la documentación antes indicada.

Debido al alcance del proyecto no nos detendremos a analizar las funciones y modos de comunicarse con la librería libusb-win32, ya que la aplicación final será mostrada utilizando el lenguaje de programación Java.

4.3.3 Mensajes de control USB implementados

Utilizando como referencia las funciones implementadas que se discutieron en el capítulo anterior para el manejo y control del transceptor nRF24L01+ (Ver Apéndice 3), se procedió a implementar dichas funciones mediante transferencias de control USB, de tal forma que, permitieran lograr una comunicación con la computadora y así poder enviar y recibir datos de manera inalámbrica.

La forma en que las funciones se implementaron fue mediante mensajes de control USB del tipo “Vendor”, en donde los parámetros “bRequest”, “wValue” y “wIndex” del formato Setup (Véase Apéndice 6), se manejan de la siguiente manera:

El parámetro “bRequest” conteniendo el comando o petición respectiva, es decir, la operación que debe realizar el microcontrolador, tales como: configurar o leer un registro del nRF24L01+ o enviar y recibir datos vía radiofrecuencia.

El parámetro “wValue” se utilizó para aquellas funciones en las que se requería de un dato (no mayor a 2 Bytes) para realizar su operación, como por ejemplo: Leer registros (el valor “wValue” contiene el valor del registro el cuál se desea leer) u otras funciones en las que utilizan éste parámetro para indicar el valor que desea escribirse a un registro específico del nRF24L01+.

Los bloques de datos de salida de los mensajes de control se utilizaron para asignar los datos a enviar por RF o para configurar registros que requieran más de 2 Bytes para configurarse. Por ejemplo configurar la dirección de TX o RX del nRF24L01+, la cuál es de 5 Bytes.

Los bloques de datos de entrada se utilizaron para asignar los datos recibidos por RF o para leer registros cuya longitud es mayor a los 2 Bytes.

La Figura 4.6 muestra el proceso de comunicación USB que se realiza entre la computadora y la interfaz del sistema.

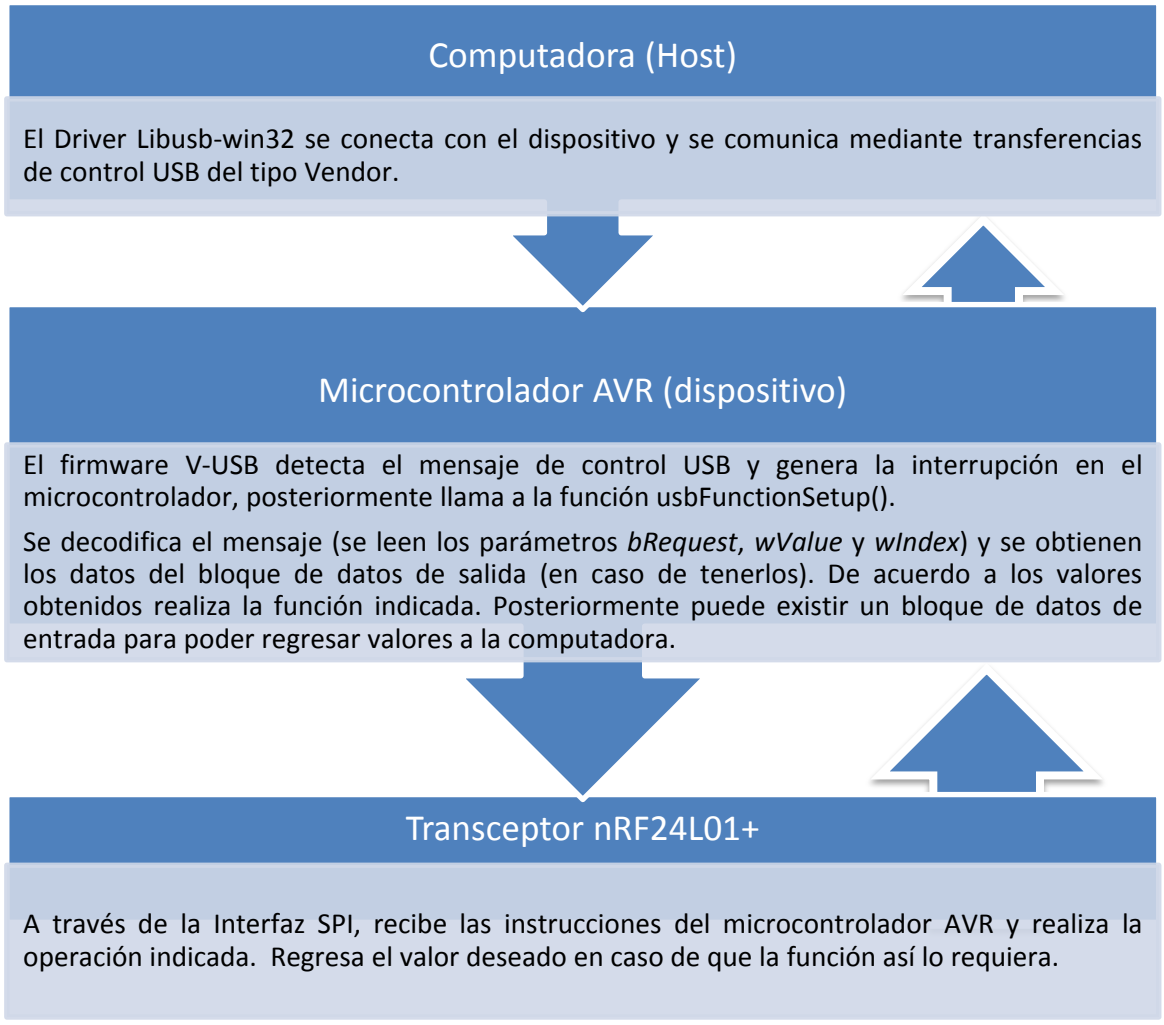

Figura 4.6.- Diagrama de comunicación USB de la interfaz del sistema.

Las funciones implementadas del sistema pueden verse en el programa principal de la interfaz USB –AVR *"Avr_Interfaz_USB.c"* (Ver Apéndice 4).

Capítulo 5
Desarrollo de la Interfaz en Java

En el presente capítulo se expone el desarrollo de la interfaz implementada en el lenguaje de programación Java, se dan a conocer algunos conceptos sobre el lenguaje de programación y se explican a detalle las herramientas utilizadas para ésta interfaz.

En el presente apartado partiremos de que nuestro sistema está conectado y configurado utilizando los drivers libusb-win32 tratados en el capítulo anterior. Se abordará el desarrollo de la interfaz desarrollada en el lenguaje de programación Java, la cual permita comunicarse con nuestro dispositivo USB y gestionar las funciones implementadas en la sección anterior. Para el desarrollo del mismo se utilizará la plataforma de desarrollo integrado de Oracle: NetBeans.

Posteriormente se explicará el proceso de control y comunicación, así también, el de incluir las librerías creadas en este proyecto para que el desarrollador pueda incluirlas como parte de su sistema.

El primer paso fue el de encontrar una herramienta que nos permitiera comunicar los drivers ya existentes de libusb-win32 con Java, ya que ésta librería se encuentra desarrollada en C. La solución se encontró en la librería envoltorio (wrapper) Java libusb, la cual consiste en una adaptación de la librería libusb.

De tal forma tendremos acceso a las funciones de libusb-win32, pero con un acercamiento desde el lenguaje de programación Java, para este propósito definiremos con mayor detalle la librería envoltorio o “wrapper” Java libusb.

5.1 JAVA LIBUSB

Java libusb es una librería envoltorio o “wrapper” que implementa las funciones de libusb en el lenguaje de programación Java, tiene la ventaja de soportar sistemas operativos Microsoft Windows, Mac OS X y GNU/Linux para 32 y 64 bits. [10]

Implementación

Java libusb provee las librerías y clases necesarias para acceder a libusb/libusb–win32 a través de la interfaz nativa de Java. La clase *ch.ntb.usb.LibusbJava* contiene las librerías compartidas y provee la interfaz nativa para acceder a libusb.

En libusb la estructura de buses (con dispositivos, configuración, interfaces y endpoints) está representada por estructuras en C. Para cada estructura, un objeto de Java es creado (nombrado como Usb_xxx) y la información es copiada a este objeto. Esto se realiza cuando se llama a la función LibusbJava.usb_get_busses(). La información sobre los buses y dispositivos conectados puede ser obtenida como un objeto de la estructura Java.

Java libusb provee de una clase simple denominada “Device” la cual representa un dispositivo USB y vuelve fácil la lectura y escritura del mismo, los errores y tiempos de espera largos resultarán en excepciones de Java.

Instalación

- Se debe instalar previamente un dispositivo con drivers libusb-win32 y revisar que estén correctamente instalados (Ver subsección de instalación de la 4.3.2Libusb-win32).

- Descargar la última versión del archivo "*ch.ntb.usb-x.x.x.jar"* y el archivo "*LibusbJava.dll"* de la librería Java libusb. Disponible en: [http://sourceforge.net/projects/libusbjava/files/]

- Ubicar ambos archivos en la misma carpeta y añadir el archivo "*ch.ntb.usb-x.x.x.jar"* como parte del proyecto de Java al que se desea incluir[24].

Una vez realizado estos pasos, se deberá tener acceso a las clases y métodos definidos de la librería Java libusb desde una nueva clase en Java. Para esto será necesario crear un objeto de la clase "Device", el cual se declara a partir del Product Id. y Vendor Id. del dispositivo USB al cual deseamos comunicarnos y que previamente fue instalado con los drivers libusb-win32.

Interfaz de programación de aplicaciones (API)

La interfaz de programación de aplicaciones (API) es el conjunto de funciones y métodos además de toda la documentación de una aplicación, se almacena como una biblioteca para que pueda ser utilizado por otro software como una capa de abstracción.

Para el caso de la librería Java libusb / libusb – win32, el API puede accederse desde la siguiente página web: [http://libusbjava.sourceforge.net/wp/res/doc/index.html].

A continuación definiremos el uso de estas librerías para el proyecto, en donde se utilizará el entorno de desarrollo integrado (IDE) NetBeans de Oracle, esto debido a que es una herramienta para programadores pensada para escribir, compilar, depurar y ejecutar programas en Java además de ser un producto de uso libre, gratuito y de tener un número importante de usuarios en todo el mundo. Esto ayuda a que el sistema tenga mayor soporte y más herramientas para el diseño.

5.2 ENTORNO DE DESARROLLO INTEGRADO NETBEANS

El Entorno de Desarrollo Integrado (IDE) NetBeans permite una forma rápida y sencilla de desarrollar aplicaciones de escritorio en Java, así como también aplicaciones web y móviles. Adicionalmente provee de un gran número de herramientas para PHP y desarrolladores en C / C++. El Software es de uso libre y tiene una gran comunidad de usuarios y desarrolladores en todo el mundo. [11]

El IDE NetBeans provee de un amplio soporte así como de una gran variedad de herramientas, librerías y ejemplos.

[24] Este proceso varía dependiendo del entorno de desarrollo, tal como Eclipse IDE o NetBeans IDE.

El IDE NetBeans puede descargarse gratuitamente desde su página web en: [https://netbeans.org/downloads/index.html]. Existen varias versiones si se desea trabajar en otros lenguajes de programación, sin embargo, para el proyecto la versión Java SE es suficiente.

El IDE NetBeans ya cuenta con todo lo necesario para programar en Java desde que se instala, ya que incluye la plataforma JDK (Java Development Kit) y JVM (Java Virtual Machine).

Instalación

El proceso de instalación para sistemas operativos Windows consiste en descargar el instalador de Windows de: [https://netbeans.org/downloads/index.html] y seguir las instrucciones del instalador.

Una vez instalado se accede al Software dando doble clic sobre el icono de NetBeans.

Interfaz

El entorno de desarrollo NetBeans ofrece una interfaz amigable al usuario y permite la programación de forma más intuitiva.

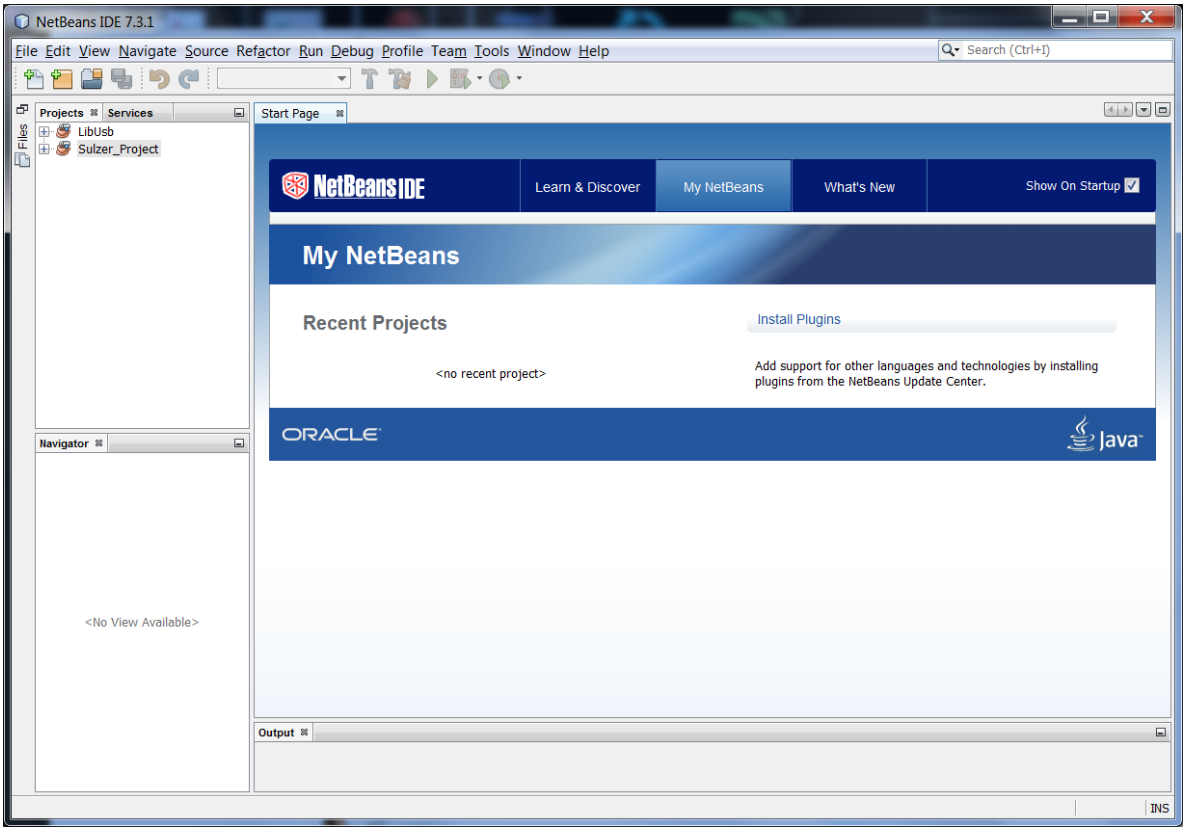

Figura 5.1.- Interfaz del Entorno de Desarrollo Integrado NetBeans.

En la Figura 5.1 se aprecia la interfaz de NetBeans, del lado izquierdo se visualiza la sección de proyectos en donde podremos acceder a proyectos previamente creados, además de ver los archivos que contiene, en la parte inferior izquierda se encuentra el panel de navegación, y en la parte inferior el panel de salida de datos Output.

Creación de un proyecto

Antes de empezar la programación en Java en el entorno NetBeans, es necesario crear un proyecto, el cuál contendrá todas las clases, métodos y librerías que requiera nuestro programa.

Para crear un nuevo proyecto en NetBeans se realizan los siguientes pasos:

1. Se selecciona el menú *Archivo*, y posteriormente la opción *Proyecto Nuevo…*

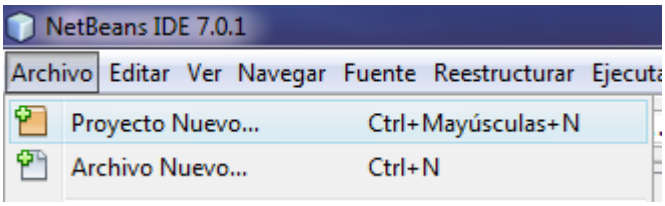

Figura 5.2.- Creación de un nuevo proyecto en NetBeans.

2. Seleccionar la categoría *Java*, escoger el proyecto *Aplicación Java* y dar clic en siguiente.

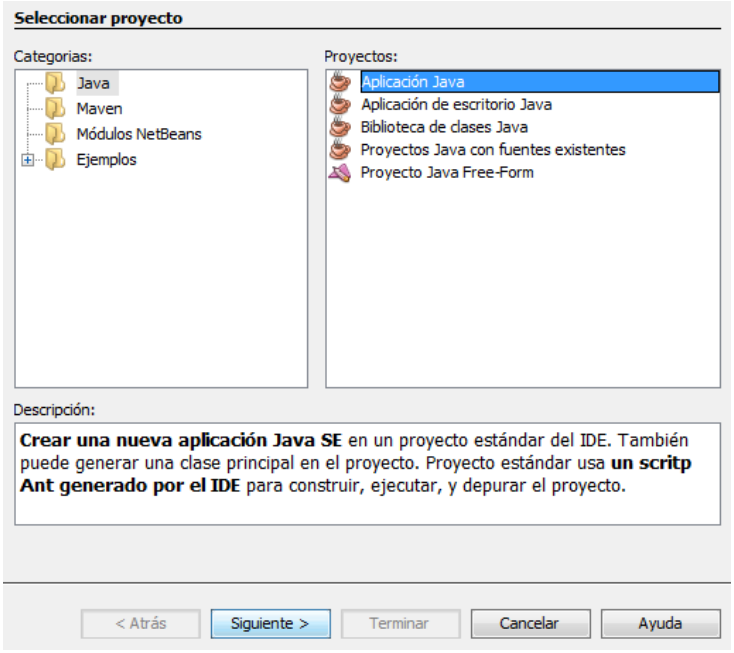

Figura 5.3.- Creación de una nueva aplicación de Java en NetBeans.

3. Introducir un nombre al proyecto y una ubicación, adicionalmente se puede crear una clase principal.

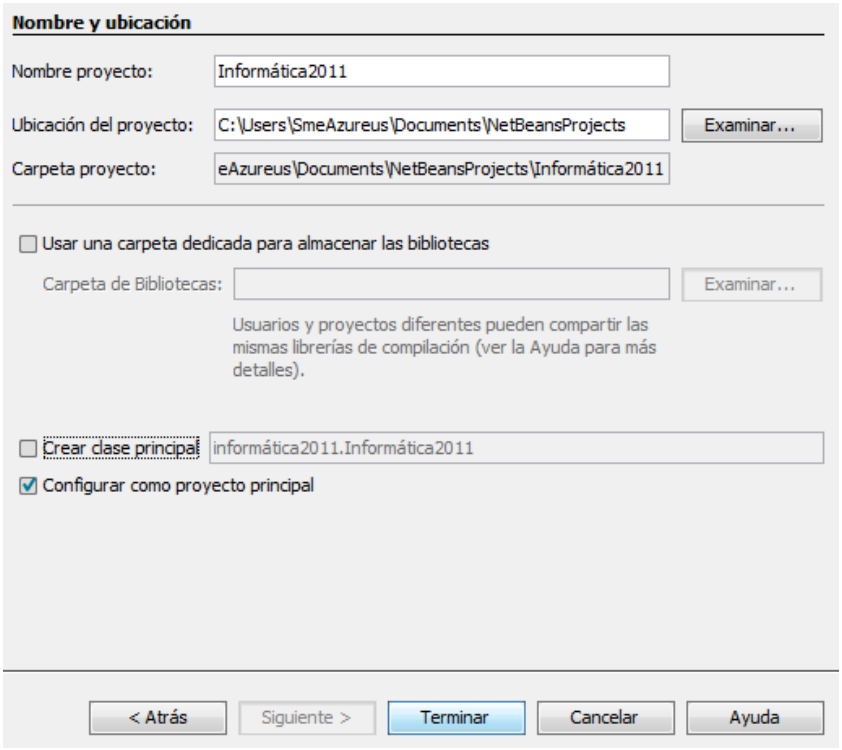

Figura 5.4.- Generación de nombre y ubicación de un proyecto nuevo en NetBeans.

Una vez generado el proyecto, se podrán generar paquetes y clases dentro de éste, además de incluir librerías o archivos java del tipo "*.jar*". Los archivos JAR provienen del acrónimo *java archive*, y es un archivo genérico de Java que empaqueta los programas y permite ejecutarlos desde la Máquina Virtual de Java (JVM).

5.3 CONTROL Y COMUNICACIÓN USB EN JAVA

Como se mencionó en las secciones anteriores, la librería Java libusb permite incluir en nuestro proyecto las funciones de control y comunicación USB de la librería libusb.

Partiendo del concepto anterior incluiremos dichas librerías a un proyecto nuevo desarrollado en la IDE de NetBeans y se generarán los métodos de Java correspondientes para poder comunicarnos con nuestro dispositivo USB y a su vez enviar datos de forma inalámbrica.

El primer paso para tal propósito es el de generar un proyecto nuevo en IDE NetBeans (Ver procedimiento en página 69) y posteriormente agregar las librerías Java libusb.

El proyecto generado se nombró "*USBRFCOM*" y se creó la clase "*UsbRfDevice*" dentro del mismo. En esta clase se generaron todos los métodos que implementan las funciones USB y que permiten la comunicación inalámbrica con otro dispositivo, con la condición de que el dispositivo con el que se desee comunicar, cuente con el transceptor nRF24L01+. A continuación describiremos el proceso de cómo añadir las librerías Java libusb al proyecto.

5.3.1 Instalación e implementación de la librería Java libusb al proyecto

Para poder implementar las funciones USB se incluyó primeramente la librería Java libusb al proyecto, para esto se realizó lo siguiente:

1. Se despliega el menú "File" o "Archivo" y se selecciona la opción de "Project Properties" o "Propiedades del proyecto", véase Figura 5.5.

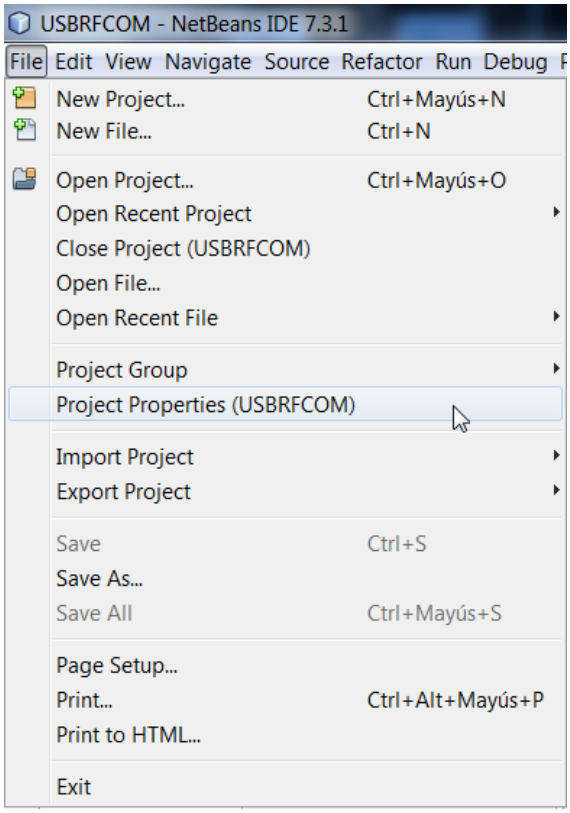

Figura 5.5.- Proceso para acceder a las propiedades del proyecto en NetBeans.

2. Se selecciona la categoría "Libraries" o "Librerías" y se da clic en la opción "Add JAR/Folder" o "Agregar JAR/Folder", véase Figura 5.6.

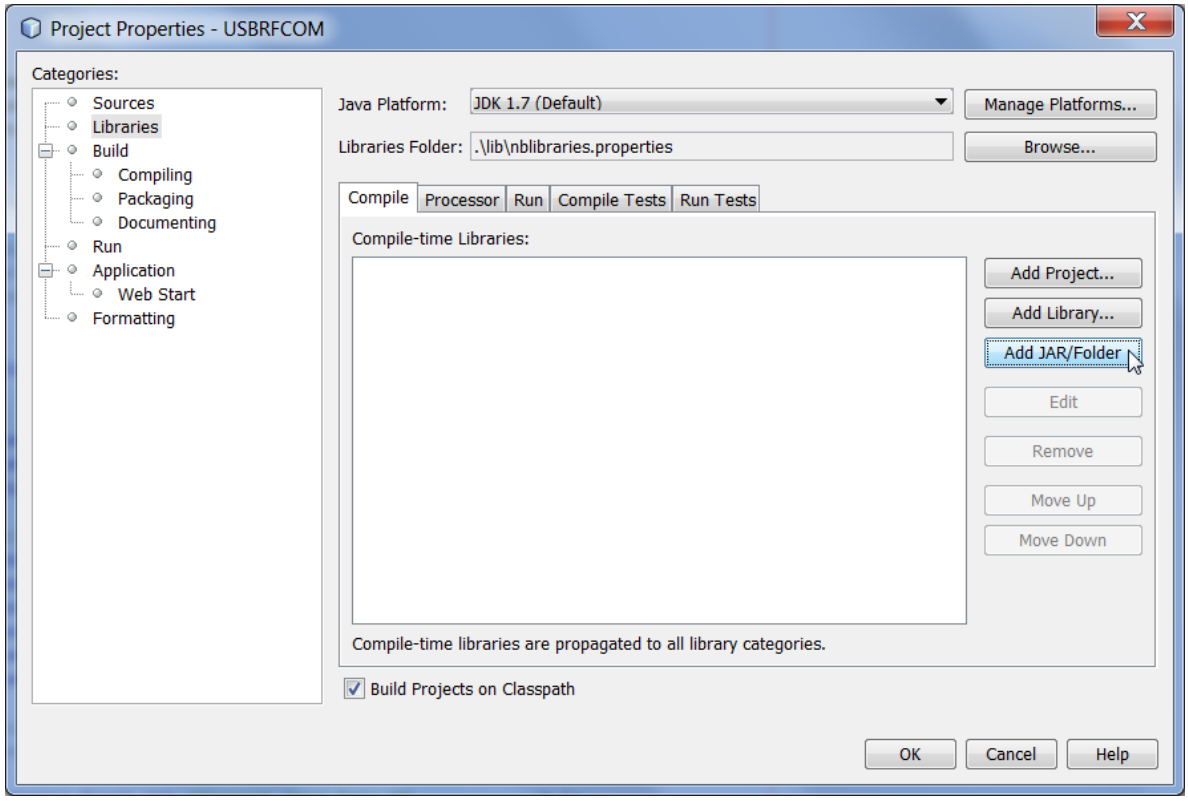

Figura 5.6.- Proceso para añadir una librería JAR en NetBeans.

3. Se busca la ruta del archivo JAR de la librería Java libusb y se agrega al proyecto, tal como se muestra en la Figura 5.7.

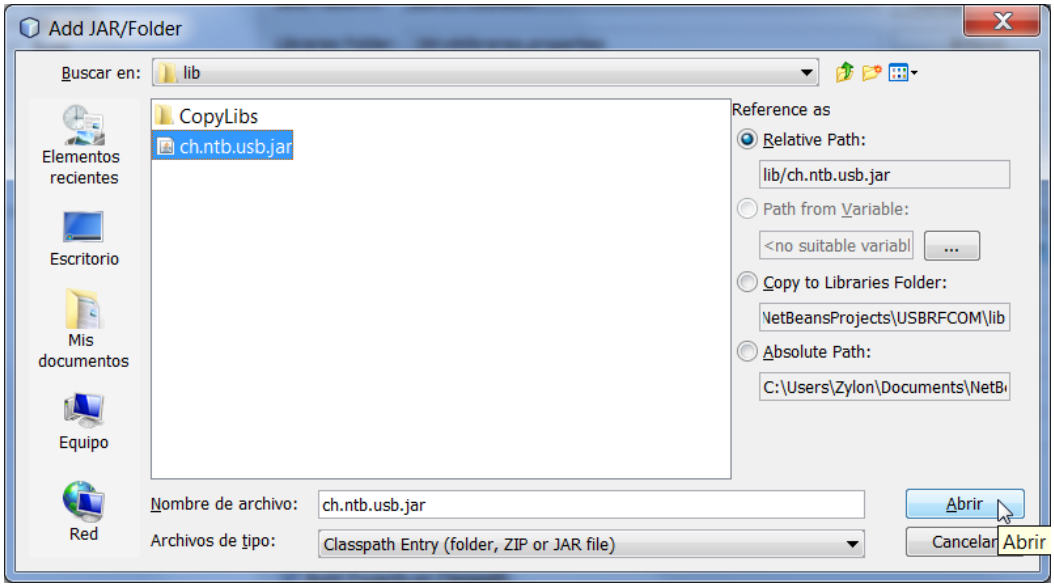

Figura 5.7.- Agregando el archivo JAR de la librería Java libusb al proyecto.

4. Una vez agregado el archivo JAR, será necesario también adicionar el archivo "*LibusbJava.dll*" en la ruta del proyecto, de otra forma no funcionará la librería adecuadamente. Para esto sólo se debe copiar el archivo mencionado y pegarlo en la ruta del proyecto al mismo nivel donde se encuentra el archivo "build.xml" y "manifest.mf".

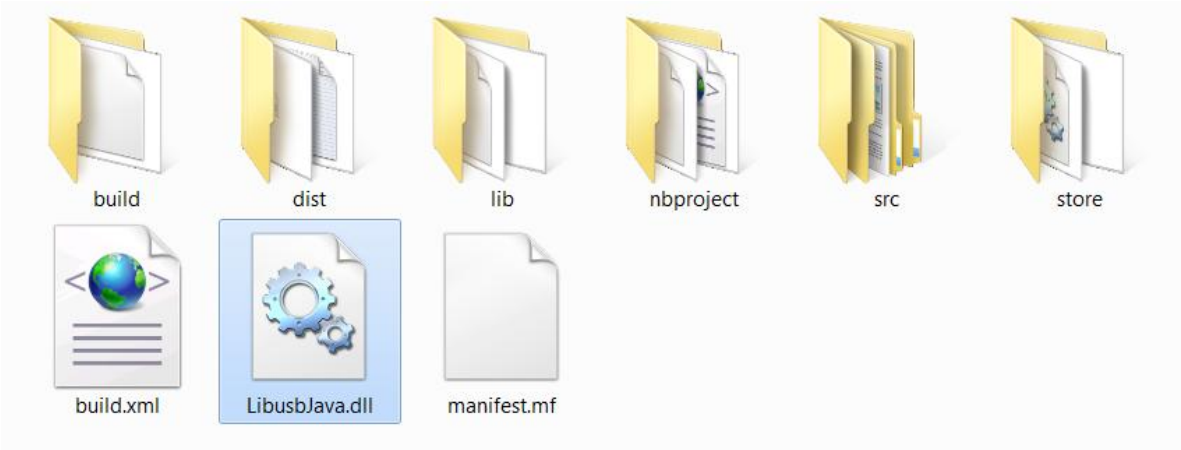

Figura 5.8.- Agregando el archivo "LibusbJava.dll" de la librería Java libusb al proyecto.

5. Finalmente se deberá incluir la siguiente línea de código a la clase en donde deseemos utilizar los métodos de la librería Java libusb:

```
Import      ch.ntb.usb.*;
```

Una vez realizado esto, tendremos acceso a las clases y métodos de la librería Java libusb. En el apartado siguiente se abordará el proceso para implementar las funciones USB mediante dicha librería.

5.3.2 Comunicación y control USB mediante Java libusb

Para este punto consideraremos que ya se tienen instalados los drivers libusb-win32 y que se agregaron las librerías Java libusb al proyecto "*USBRFCOM*" en Java, de tal forma que se trabajará sobre la clase "*UsbRFDevice*" definida dentro de dicho proyecto.

La clase "*UsbRFDevice*" se definió dentro de un paquete denominado "*Javausb*", ésta clase contiene todos los métodos que permiten la comunicación inalámbrica mediante el protocolo USB.

Se definieron dentro de esta clase los parámetros a utilizar para las peticiones "*bRequest*" de los mensajes de control USB.

El proceso de comunicación USB mediante la librería Java libusb consiste en los siguientes pasos:

- Definir un dispositivo USB, creando una instancia de la clase "*Device*". Éste se genera a partir del Vendor y Product Id. y para poder visualizarlo deberá estar conectado a algún puerto USB de la computadora.

Para definir el dispositivo, se llama a un constructor y se genera una instancia de la case "*Device*" con los Vendor y Product Id. deseados. Este proceso se muestra a continuación:

```
// Se define el Product y Vendor Id.
private static final short  idVendor  =    0x20A0;
private static final short  idProduct =    0x4204;
// Se crea una instancia de la clase Device, la cual representa el
//  dispositivo USB.
Device dev = USB.getDevice(idVendor,  idProduct);
```

En el ejemplo anterior se definen dos valores hexadecimales de 2 Bytes correspondientes al Vendor y Product Id. Posteriormente se llama a un constructor que genera una instancia de la clase "*Device*", el cual se nombra como: dev. Este objeto se construye a partir de la función USB.getdevice() y tiene como parámetros los valores Vendor y Product Id. previamente definidos.

- Una vez definido el dispositivo, se procede a "abrir".

El término de "abrir" el dispositivo USB, se refiere al proceso de enumerar el bus para posteriormente, una vez que el dispositivo sea detectado, leer los descriptores y el valor máximo de datos por paquete permitido. En el caso de que no se encuentre el dispositivo, automáticamente se llama una excepción en Java.

Todo este proceso se realiza por medio de la función "*open*", de la clase "*Device*". Dicha función realiza el proceso antes descrito y requiere como parámetros un valor de configuración, de interfaz y de interfaz alterna. Estos parámetros permiten distintos comportamientos en el dispositivo, para el caso de éste proyecto, los parámetros deberán inicializarse con la configuración estándar y sin ninguna interfaz, para esto se debe colocar como valor de configuración: "1", interfaz "0" e interfaz alterna: "-1". Partiendo del ejemplo anterior, el dispositivo se abriría de la siguiente manera:

```
try {
dev.open(1, 0, -1);
```

```
} catch (USBException e) {

// Si ocurre un error durante la conexión, se presenta una
excepción y ejecutará el código de esta sección.

System.exit(0);
}
```

- Una vez que el dispositivo está abierto, se podrá realizar la comunicación con éste, se pueden realizar los distintos tipos de transferencias USB (control, interrupción, isócronas o bulto).

La comunicación mediante Java libusb permite los distintos tipos de transferencias de acuerdo a las especificaciones del protocolo USB. Para el caso de transferencias de control, la clase "*Device*" ofrece el método "*controlMsg*", el cual realiza una petición de control al dispositivo. Los parámetros son nombrados de igual forma como se muestran en la especificación USB (Ver Apéndice 6).

La función "*controlMsg"* está compuesta de la siguiente forma:

```
public int controlMsg(int requestType,
                      int request,
                      int value,
                      int index,
                      byte[] data,
                      int size,
                      int timeout,
                      boolean reopenOnTimeout)
               throws USBException
```

Parámetros:

requestType – Tipo de petición USB (tal como viene identificada en la especificación USB, ver Apéndice 6). Se pueden usar las constantes definidas en la clase "*USB*" (REQ_TYPE_xxx).

request – Petición específica USB. Para el caso de peticiones del tipo "Vendor" este parámetro es arbitrario. Lo utilizaremos para identificar las funciones específicas del proyecto.

value – Campo variable que depende de la petición (request).

index - Campo variable que depende de la petición (request).

data – Buffer de envío/recepción de datos.

size – Tamaño o longitud del buffer. El valor "0" es válido pero aun así debe ingresarse un buffer (data).

timeout – Cantidad de tiempo expresada en milisegundos, en la cual el dispositivo intentará enviar/recibir datos hasta que exista una excepción por tiempo de espera agotado.

reopenOnTimeout – Si se coloca en valor true, el dispositivo intentará abrir nuevamente la conexión y enviar/recibir datos antes de que ocurra una excepción por tiempo de espera agotado.

Regresa:

Número de bytes enviados/leidos.

Tipo de excepción que lanza:

USBException

- Finalmente, una vez realizado el envío o recepción de datos, el dispositivo se cierra para dejar libre la interfaz reclamada.

A continuación se muestra todo el proceso de cómo poder enviar un mensaje de control con datos del host al dispositivo.

```
// Se importan las librerías Java libusb a nuestra clase.
import ch.ntb.usb.*;
public class UsbControlMsgWrite {
// Se define el Product y Vendor Id.
private static final short  idVendor  =    0x20A0;
private static final short  idProduct =    0x4204;
// Se crea una instancia de la clase Device, la cual representa el
//  dispositivo USB.
Device dev = USB.getDevice(idVendor,  idProduct);
try {
dev.open(1, 0, -1);
int bRequest_Send_Data  = 1;
```

```
int wValue = 0;

int wIndex = 0;

byte[] Datos = new byte [2];

Datos[0] = 10;

Datos[1] = 50;

int size = Datos.length;

//          Estructura del Mensaje de Control USB mostrado:
//                      ---requestType---
// Dirección de los datos: Host al dispositivo.
// Tipo de mensaje: Tipo Vendor.
// Recipiente del mensaje: Dispositivo.
//                        ---bRequest---
// Valor de la petición: 1
//                         ---wValue---
// Valor 'wValue' asignado a la petición: 0
//                         ---wIndex---
// Valor 'wIndex' asignado a la petición: 0
//                          ---data---
// Valor de los datos a enviar: 10, 50.
//                          ---size---
// Cantidad de datos a enviar: 2.
//                         ---timeout---
// tiempo de espera antes de excepción: 2 seg.
//                      ---reopenOnTimeout---
// El dispositivo no intentará reconectarse nuevamente si el lapso
// de tiempo de espera termina.

dev.controlMsg(USB.REQ_TYPE_DIR_HOST_TO_DEVICE |
USB.REQ_TYPE_TYPE_VENDOR | USB.REQ_TYPE_RECIP_DEVICE,
bRequest_Send_Data, wValue, wIndex, Datos, size, 2000, false);

// Cerramos el dispositivo y dejamos libre la interfaz.

dev.close();

} catch (USBException e) {


// Si ocurre un error durante la conexión, se presenta una
excepción y ejecutará el código de esta sección.

System.exit(0);
}

}
```

Para el caso de recepción de datos, el proceso es similar, sólo que esta vez la dirección de los datos debe ser del dispositivo al host (`USB.REQ_TYPE_DIR_DEVICE_TO_HOST)`, se debe definir el buffer para los datos que servirá para almacenarlos[25] y finalmente para determinar el número de datos que se obtuvieron en el mensaje, el mismo método "*controMsg*" regresará éste valor como un entero, por lo que deberá leerse el valor de retorno de dicha función para determinar cuántos datos se recibieron.

Adicional a las transferencias de control, se pueden realizar los otros tipos de transferencias USB. Para la información completa sobre las clases y métodos disponibles de la librería Java libusb se puede visitar la página web: [http://libusbjava.sourceforge.net/wp/res/doc/index.html].

Para lograr la implementación de la comunicación inalámbrica por medio del transceptor nRF24L01+ y nuestro dispositivo USB, se implementaron los métodos que permitieran controlar al dispositivo por medio de transferencias de control USB como la mostrada en el ejemplo anterior, en la siguiente sección se tratará con mayor detalle los métodos de la clase "*UsbRfDevice*" desarrollada.

5.3.3 Métodos de Java implementados

Ahora que se ha explicado la forma de cómo lograr la comunicación USB mediante la librería Java libusb, proseguiremos con la implementación de los métodos en Java para la clase que se denominó "*UsbRFDevice*" del proyecto "*USBRFCOM*".

Antes de comenzar a definir y explicar los métodos implementados, se abordarán algunas diferencias a considerar respecto al tipo de datos utilizados en Java y los utilizados en C.

En el lenguaje de programación Java, toda la información que se maneja en un programa está representada por dos tipos principales de datos[26]:

a) Datos de tipo básico o primitivo.

b) Referencias a objetos.

Los tipos de datos básicos o primitivos no son objetos y se pueden utilizar directamente en un programa sin necesidad de crear objetos de este tipo. La biblioteca Java proporciona clases asociadas a estos tipos que proporcionan métodos que facilitan su manejo.

La Tabla 5.1 muestra una comparativa de los tipos de datos primitivos que soporta Java y los tipos de datos que soporta el lenguaje C:

[25] El buffer deberá ser de longitud definida de valor igual o mayor al máximo número de datos por paquete, típicamente 64 Bytes.
[26] Extraído de [http://puntocomnoesunlenguaje.blogspot.mx/2012/04/tipos-de-datos-java.html].

Tipo de dato	Lenguaje	Representación	Tamaño (Bytes)	Rango de Valores	Valor por defecto
byte	Java	Número Entero con signo	1	-128 a 127	0
	C	No disponible en C	-	-	-
short	Java	Número Entero con signo	2	-32,768 a 32,767	0
	C	Número Entero con signo	2	-32,768 a 32,767	0
int	Java	Número Entero con signo	4	-2,147,483,648 a 2,147,483,647	0
	C	Número Entero con signo	2	-32,768 a 32,767	0
long	Java	Número Entero con signo	8	-9,372,036,854,775,808 a 9,223,372,036,854,775,807	0
	C	Número Entero con signo	4	-2,147,483,648 a 2,147,483,647	0
float	Java	Número de punto flotante de precisión simple de acuerdo a Norma IEEE 754	4	$\pm 1.4x10^{-45}$ a $\pm 3.4x10^{38}$	0
	C	Número de punto flotante con precisión de 6 decimales sin signo	4	$1.2x10^{-38}$ a $3.4x10^{38}$	0
double	Java	Número de punto flotante de precisión doble de acuerdo a Norma IEEE 754	8	$\pm 4.9x10^{-324}$ a $\pm 1.8x10^{308}$	0
	C	Número de punto flotante con precisión de 15 decimales	8	$2.3x10^{-308}$ a $1.8x10^{308}$	0
char	Java	Carácter Unicode	2	\u0000 a \uFFFF	\u0000
	C	Carácter código ASCII	1	\u00 a \uFF	\u00

boolean	Java	Dato lógico	-	true ó false	false
	C	Dato lógico	1	Representa el boolean como un Int en donde false es igual a cero y true es cualquier valor diferente de cero.	0 (false)
void	Java	-	-	-	-
	C	-	-	-	-

Tabla 5.1.- Tabla comparativa de los tipos de datos primitos en Java y C.

Es importante tomar en cuenta los tipos de datos que maneja un lenguaje de programación respecto a otro ya que pueden ocurrir errores al interpretarlos.

En Java los datos primitivos son siempre con signo, por lo que en el caso particular de este proyecto, se observa que el método "*controlMsg"* de la clase "*Device"* de la librería Java libusb, envía y recibe datos del tipo byte, los cuales van desde -127 a 127, y se debe considerar que manejamos sólo datos positivos en el programa en lenguaje C desarrollado, por lo que se tuvo que utilizar datos del tipo int, con valores limitados (desde 0 a 255), y dentro de cada método implementado reconvertir los datos al tipo byte para poderlos utilizar en el método "*controlMsg*" para las transferencias de control USB.

Tomando en cuenta lo anterior, se procedieron a generar los métodos en Java para todas las funciones USB implementadas en el programa "*AVR_Interfaz_USB.c"* (Ver Apéndice 4).

Todos los métodos implementados fueron creados para la clase *"UsbRFDevice"* contenidos en el proyecto "*USBRFCOM".* La interfaz de programación de aplicaciones (API) de los métodos creados para la clase "*UsbRFDevice*" se encuentra en el Apéndice 7 y la clase completa puede consultarse en el Apéndice 8.

Capítulo 6
Diseño e implementación del sistema

En éste capítulo se muestra la implementación y diseño físico del sistema utilizando los módulos desarrollados en los capítulos anteriores. Se presenta el diseño de la placa de circuito impreso y las pruebas realizadas.

En este capítulo consideraremos toda la información presentada en los apartados anteriores y la utilizaremos para realizar todo el diseño e implementación del sistema; se tratará el diseño del circuito y la placa de circuito impreso así como su elaboración; posteriormente se abordará el cómo integrar la interfaz desarrollada a otro proyecto o sistema de control el cuál requiera de comunicación inalámbrica con la computadora; y finalmente se mostrarán algunas pruebas realizadas al sistema.

Como primera sección se abordará la parte de diseño del circuito para posteriormente realizar el diseño de la placa de circuito impreso y la implementación del mismo.

6.1 DISEÑO DEL CIRCUITO

Para el diseño del circuito se consideraron los siguientes puntos:

- El sistema debía ser pequeño para que fuera práctico y portátil.
- De bajo costo que permita ser una opción viable, en comparación con los sistemas existentes en el mercado.
- Permitir flexibilidad para poder colocar distintos tipos de modelos disponibles de nrf24L01+.
- Poder ser programado desde el mismo circuito sin retirar algún componente, de esta forma actualizar el firmware o realizar ajustes al programa de forma simple y rápida.
- Hardware requerido por la interfaz USB, considerando los requerimientos tratados en el capítulo 4 (Ver Figura 4.3).

Tomando en cuenta estos puntos, se procedió a realizar el diagrama esquemático del circuito y considerar la lista de componentes del mismo. En la Figura 6.1 se muestran de forma general los elementos involucrados en el circuito del sistema.

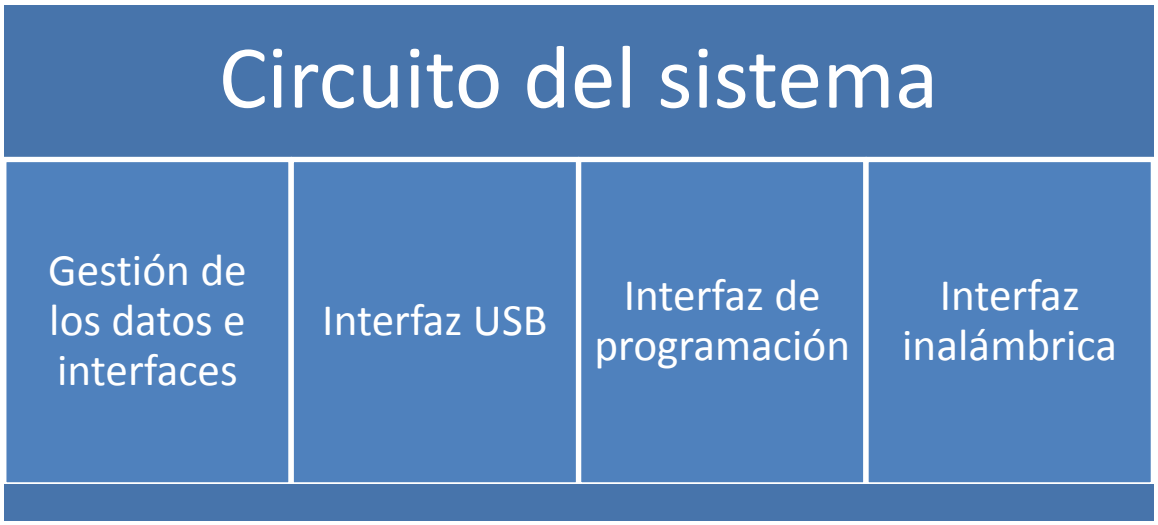

Figura 6.1.- Diagrama de bloques del circuito del sistema.

La gestión de los datos e interfaces se refiere a la parte de control y sus elementos, es decir, el microcontrolador ATMEGA 8 y los componentes para su funcionamiento (reloj y elementos pasivos); la interfaz USB está compuesta por los elementos involucrados para dicha comunicación mediante V-USB (ver Figura 4.3); la interfaz de programación sólo involucra las conexiones y el conector que permiten la programación mediante ICSP; y por último, la interfaz inalámbrica compete a las conexiones, el conector de la interfaz SPI, y el ajuste de voltaje a 3.3V mediante un regulador, esto permite la comunicación con el dispositivo transceptor nRF24L01, además de proveerlo de una fuente de alimentación aprovechando los 5V del Bus USB.

6.1.1 Diseño del diagrama esquemático del circuito

A continuación se mostrará cómo se llevó a cabo el diagrama del sistema diseñado. Para este propósito se utilizó la herramienta de diseño EAGLE desarrollada por CadSoft [12].

La herramienta EAGLE permite la creación de diagramas esquemáticos, placas de circuitos impreso y creador de rutas, entre otras funciones. Su editor de esquemáticos permite, entre otras cosas:

- Crear esquemáticos de hasta 999 hojas.
- Referencias cruzadas para los nodos.
- Copiado simple de partes.
- Generador automático de placa.
- Generador automático de voltajes de alimentación.
- Librerías de componentes.

El Software EAGLE tiene un costo, sin embargo, CadSoft permite utilizarlo gratuitamente con una licencia del tipo Freeware[27] que presenta algunas limitaciones, como son:

- La placa del circuito está limitada a un área de 100 x 80 mm.
- Sólo se permite utilizar dos capas de señales (superior o “top” e inferior o “bottom”).
- El editor de esquemáticos permite crear una sola hoja.

Para nuestro sistema estas limitantes no afectan al diseño ya que la placa no excederá dichas dimensiones y sólo requerirá de una hoja y dos capas para las conexiones de los componentes.

El circuito está constituido por los siguientes componentes:

- Microcontrolador AVR ATMEGA 8.
- Conector USB.
- Cristal de cuarzo de 12MHz.

[27] Es un tipo de software que se distribuye sin costo, disponible para su uso y por tiempo ilimitado.

- Conectores header hembra de 0.1”, 2x4 (interfaz nrf24L01+) y 2x3 (interfaz de programación).
- Regulador de voltaje de 3.3V (para alimentación del nrf24l01+).
- Led de encendido y led indicador (conectado a un pin del microcontrolador).
- Diodos Zener de 3.3V o 3.6V.
- Elementos pasivos (resistencias y capacitores).

El diagrama esquemático del sistema realizado en EAGLE, se muestra en la Figura 6.2.

El diseño del circuito se diseñó tomando en cuenta los módulos disponibles a la venta del transceptor nRF24L01+, los cuáles cuentan con un conector tipo “header macho” de 4x2 con espaciamiento de 0.1”, que permiten la comunicación con el microcontrolador.

Adicionalmente se cuenta con otro conector tipo “header hembra” de 2x3, el cuál permite la programación del dispositivo de forma serial a través de la interfaz SPI, este tipo de programación es estándar en estos microcontroladores y se le conoce como ICSP (In Circuit Serial Programming), sólo requiere de 6 pines y permite la programación del dispositivo sin desconectarlo.

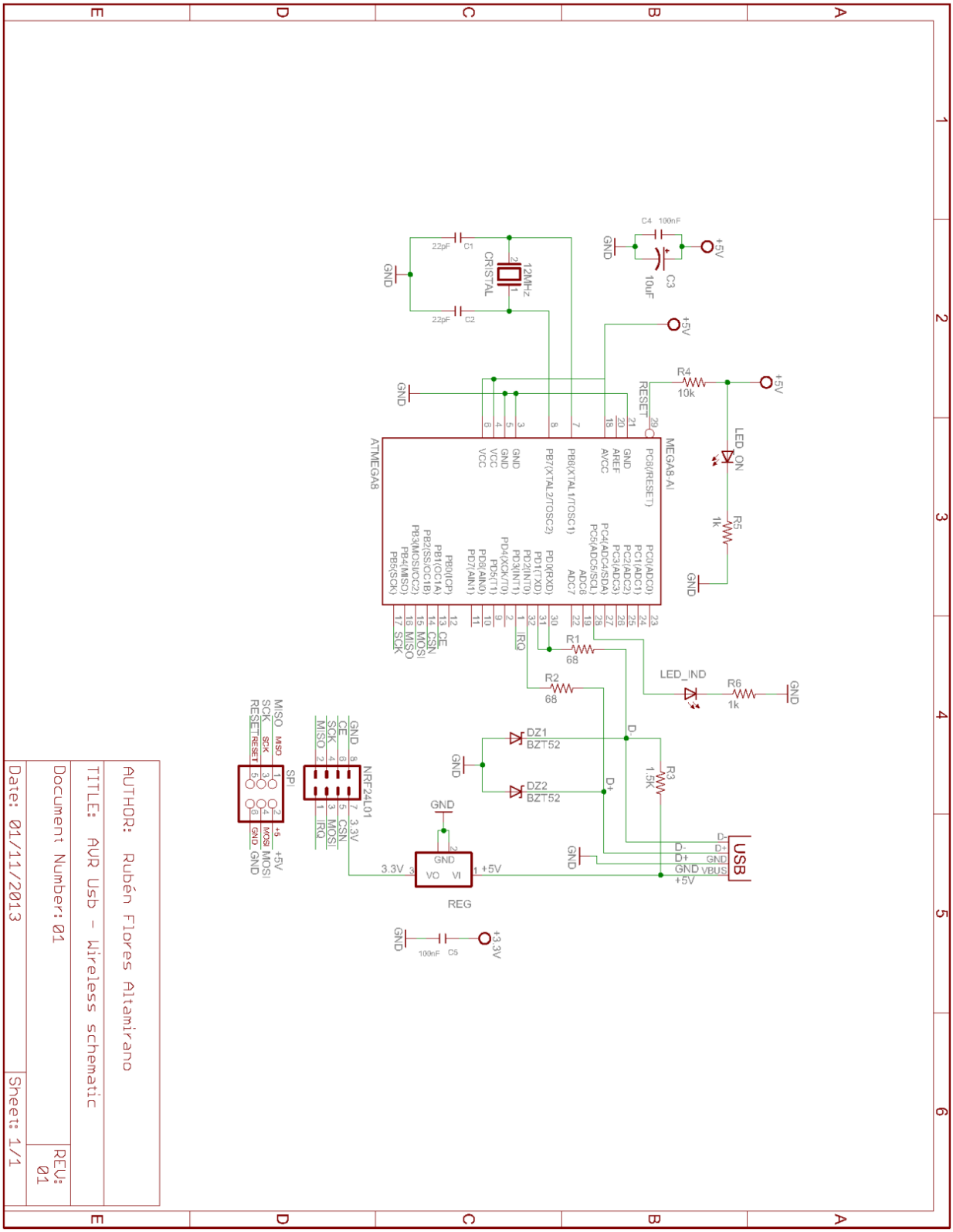

Figura 6.2.- Diagrama esquemático del sistema.

6.1.2 Diseño de la placa de circuito impreso (PCB)

Partiendo del diagrama esquemático, se prosigue con el diseño físico del sistema, esto es, realizar el arreglo físico de los componentes y la placa de circuito impreso, tomando en cuenta las consideraciones del diseño mencionadas anteriormente.

El tamaño del circuito se diseñó para que fuera relativamente pequeño y permitiera portabilidad y practicidad, de tal forma que, se consideraron unas dimensiones similares a una memoria USB Flash promedio, las cuales oscilan entre 57mm-80mm de largo y 16.5mm-17.5mm de ancho.

Estas condiciones requerían que los componentes utilizados fueran de menor tamaño, por lo que se optó por utilizar dispositivos de montaje superficial (SMD) para componentes como resistencias, capacitores, microcontrolador, leds, regulador de voltaje y diodos. Para el caso del cristal de cuarzo y conectores (USB, interfaz del nRF24L01+ e interfaz de programación ICSP) se utilizaron componentes de agujeros pasantes (Through-Hole). Adicionalmente se requirió del uso de dos capas una superior (top) y una inferior (bottom) para lograr todas las conexiones.

Dadas las características de la placa, fue necesario recurrir a una empresa que proporcionara el servicio de manufactura de la misma, tras haber analizado varios proveedores se optó por el proporcionado por la página web “Seeedstudio”, el servicio es adicional a la venta de productos electrónicos y se accede en su página de internet: [http://www.seeedstudio.com/service/] a través de la opción “Fusion PCB”.

En cuestiones de diseño, “Seeedstudio” proporciona un archivo compatible con EAGLE el cuál representa una serie de reglas que permiten identificar errores que pudieran causar problemas al manufacturarse las placas, de esta forma se asegura un producto de calidad y sin problemas. Existen métodos adicionales que ayudan a revisar que el diseño este correcto, para obtener una guía más detallada de este proceso se puede acceder a la siguiente página web: [http://dangerousprototypes.com/docs/Get_your_PCBs_made].

Una vez que se realizan todas las rutas de la placa y la comprobación de errores con EAGLE, se procede a generar los archivos de tipo fichero “Gerber”, éstos contienen toda la información necesaria para la fabricación de la placa de circuito impreso (PCB), estos archivos deberán ser proporcionados junto con el pedido para que el proveedor pueda manufacturar la placa.

“Seeedstudio” ofrece el servicio de manufactura de PCB a precios accesibles, para el caso de este proyecto el costo de 5 placas (+1 gratis) de dos capas y dimensiones entre 5cm x 10 cm en color verde, fue de $18.90 USD más $8.22 USD por gastos de envío, dando un total de $27.12 USD teniendo un precio por unidad de $4.52 USD o aproximadamente $58.7 MXN al tipo de cambio de $13.0 MXN al momento de realizar este trabajo.

La placa de circuito impreso diseñada para el sistema se aprecia en la Figura 6.3.

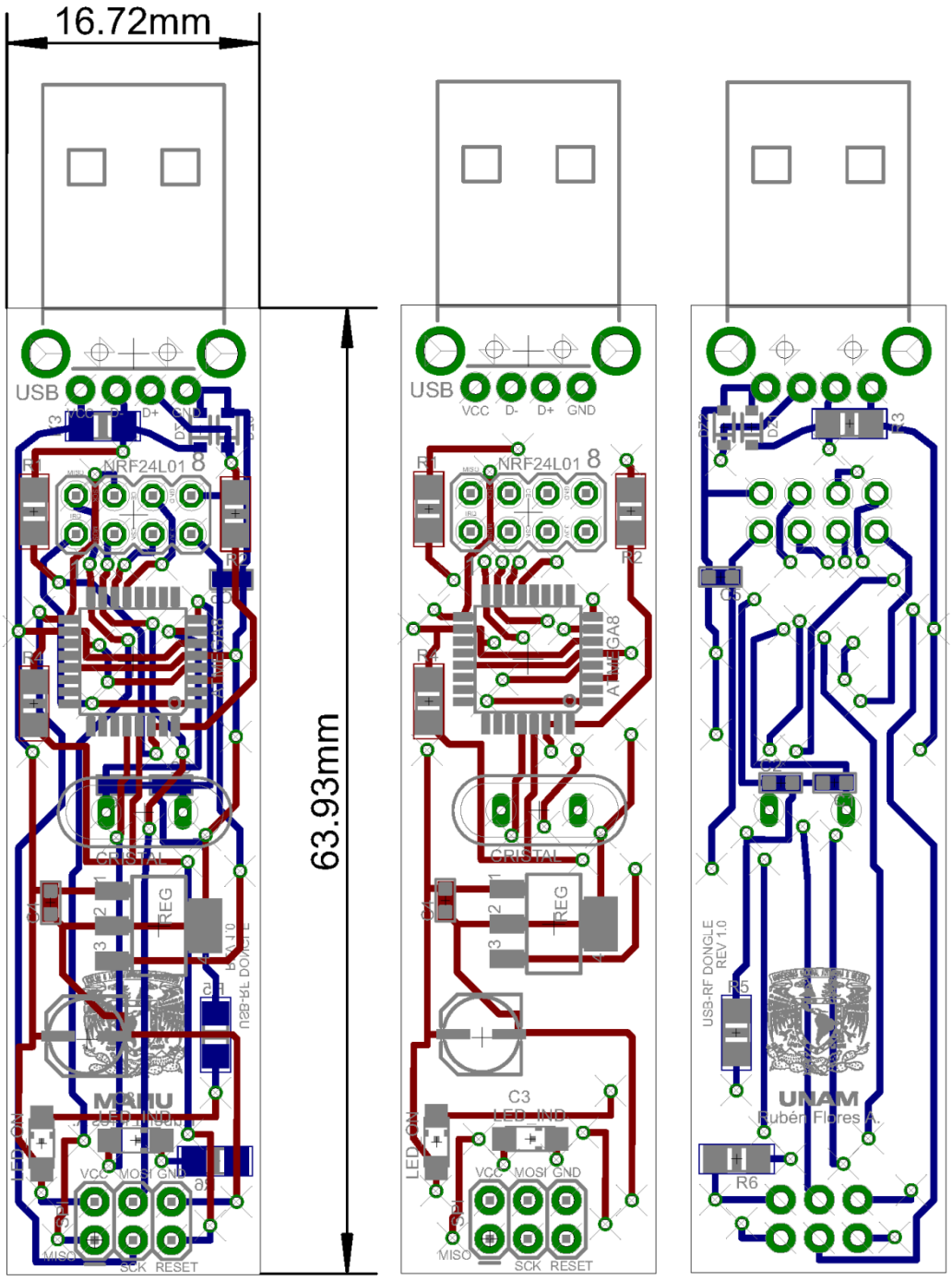

Figura 6.3.- Placa de circuito impreso diseñada en EAGLE CadSoft, (de izquierda a derecha) se observa la placa completa y dimensiones, la vista de la capa superior (Top) y la vista de la capa inferior (Bottom).

El diseño mostrado se mandó manufacturar a “Seeedstudio” y en un periodo aproximado de 30 días se recibieron las placas de circuito impreso, las cuáles pueden apreciarse en la Figura 6.4.

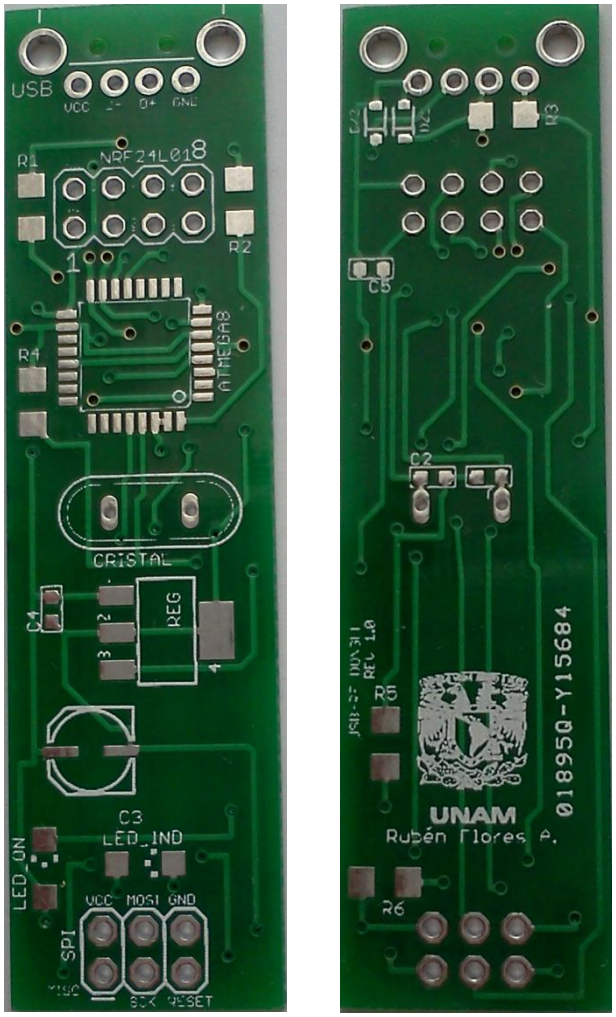

Figura 6.4.- Placas de circuito impreso (PCB) del sistema.

Finalmente se procedieron a soldar todos los componentes en la placa, teniendo especial cuidando en aquellos que deben mantener una posición tales como el microcontrolador, capacitor electrolítico, diodos y leds. En la Figura 6.5 se puede apreciar la placa de circuito impreso con sus elementos soldados.

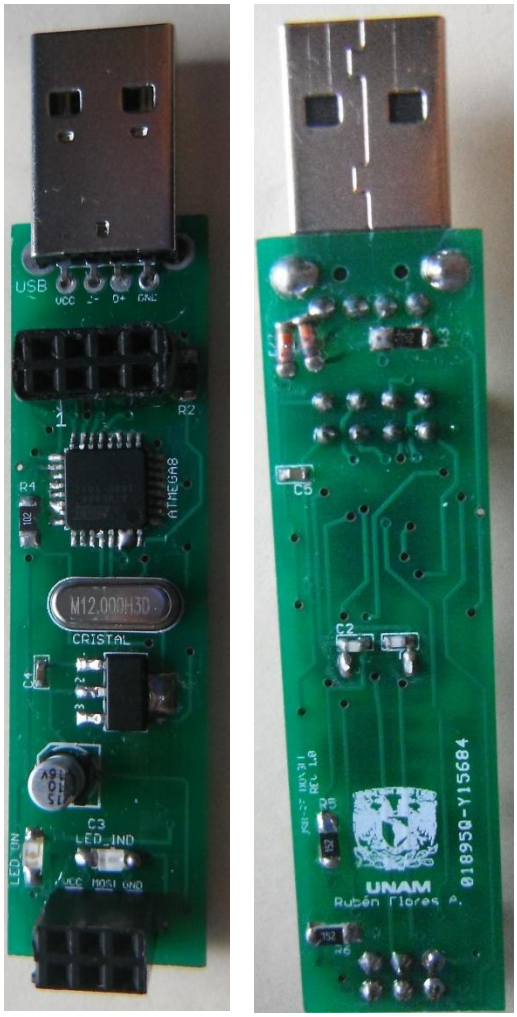

Figura 6.5.- Placa de circuito impreso (PCB) con sus componentes soldados.

En la Figura 6.5 se observan los conectores del tipo “header hembra” utilizados para la comunicación con el transceptor y la interfaz de programación, en el caso de la comunicación inalámbrica, tenemos flexibilidad para poder utilizar distintos modelos disponibles en el mercado del transceptor nRF24L01+, pudiendo ser uno con antena integrada en el PCB o con antena externa (Ver Figura 3.2). Para el caso de la interfaz de programación se cuenta con un conector de 6 pines que se requieren para la programación serial ICSP, los pines requeridos son: VCC, GND, SCK, MISO, MOSI y RESET.

6.2 IMPLEMENTACIÓN DE LA INTERFAZ DEL SISTEMA DE CONTROL

En este apartado se explicará el proceso de cómo integrar la interfaz desarrollada al sistema de control deseado, para que pueda gestionarse a través de la computadora. Este proceso se divide en dos partes: la primera se refiere a la implementación de la interfaz de usuario, la cuál representa la inclusión de las librerías en Java creadas que permiten la comunicación inalámbrica con el sistema de control; la segunda parte representa la implementación física, ésta explica las modificaciones que deberán hacerse al sistema de control para lograr la comunicación inalámbrica y la carga del programa al microcontrolador ATMEGA8, encargado de la comunicación USB y el control y manejo de los datos.

6.2.1 Implementación de la interfaz de usuario en Java

En esta sección abordaremos la implementación de las librerías en Java creadas para la comunicación y control inalámbrico desde la computadora, las cuales se trataron en el capítulo anterior. Con ayuda de estas librerías se podrán crear fácilmente interfaces de usuario gráficas en Java, que se ajusten a las necesidades del usuario y del sistema de control.

Debido a que el proceso para incluir las librerías desarrolladas es similar al visto en el capítulo 5 sobre la integración de las librerías Java libusb (Ver Sección 5.3.1), tomaremos los mismos pasos como referencia, sólo que en este caso se considera el archivo "*USBRFCOM.jar*" para añadir como librería al proyecto de Java y de igual forma copiaremos el archivo "*LibusbJava.dll*" en la ruta de dicho proyecto[28]. Previamente se debieron haber instalado los drivers USB del dispositivo en la computadora (Ver sección 4.3.2).

Una vez que se hayan añadido ambos archivos al proyecto, se podrá tener acceso a las clases y métodos creados de la librería USBRFCOM, dentro de ésta se encuentra la clase UsbRFDevice, ésta contiene los métodos necesarios para la comunicación inalámbrica (Ver Apéndice 7), para poder acceder a ellos se debe generar una instancia de la clase a través de un constructor, como se muestra a continuación:

```
// Importa la librería que contiene la clase UsbRFDevice.
import Javausb.*;

public class ejemploClase {

// Se llama al Constructor y genera una instancia de la clase UsbRFDevice
// la cual se denomina AVR (El nombre lo decide el usuario).

    UsbRFDevice AVR = new UsbRFDevice();

}
```

[28] No es necesario agregar la librería Java libusb al proyecto ya que viene incluida en el archivo "*USBRFCOM.jar*".

Una vez realizado lo anterior se tendrán acceso a los métodos de la clase “UsbRFDevice” a través de la instancia generada, el siguiente paso será el de configurar las opciones del dispositivo: velocidad de transmisión, canal de radiofrecuencia, dirección asignada para recibir datos por radiofrecuencia, dirección asignada para transmitir (esta dirección será la dirección del receptor en el sistema de control), número de bytes a enviar y recibir por cada paquete (este valor será fijo para todos los paquetes) o en su defecto habilitar la longitud dinámica de paquetes (de esta manera se podrán recibir paquetes de longitud variable desde 1 hasta 32 Bytes). Para este propósito se utilizan los métodos: “setRFChannel”, “setRFMode”, “setTXAddress”, “setRXAddress”, “setTXLength”, “setRXLength”, “setRFSpeed”, “enableDPL”, “disableDPL”.

La configuración de estos parámetros puede hacerse dinámicamente durante el programa y también se les puede asignar un valor inicial fijo, estos valores iniciales se almacenan en la EEPROM para que siempre que se conecte el dispositivo se configuren automáticamente, esto permite que no sea necesario configurarlos cada vez que se corra el programa principal, sin embargo, deberán asignarse por lo menos una vez y correr el programa, de esta forma se guardarán en la EEPROM y posteriormente se podrá borrar este código del programa. Para este propósito se cuentan con los métodos: “setInitialRFChannel”, “setInitialRFMode”, “setInitialTXAddress”, “setInitialRXAddress”, “setInitialTXLength”, “setInitialRXLength”, “setInitialRFSpeed”, “setInitialDPLState”.

Una vez que el dispositivo esté configurado para poder comunicarse adecuadamente con el sistema de control, se podrán utilizar los métodos para enviar o recibir datos por radiofrecuencia, si se desean enviar datos de longitud fija, se debe utilizar el método “sendDataRf”; para mensajes de longitud variable se utiliza el método “sendDynamicDataRf”; y adicionalmente se puede usar el método “sendCommandDataRf” que permite enviar un comando de 2 Bytes junto con un dato de 2 Bytes, por lo tanto se debe configurar la longitud de datos a transmitir “setTxLength” a 4 Bytes o tener habilitado el envío de datos dinámico “enableDPL”.

Para el caso de la recepción de datos, se cuenta con los métodos “readDataRf”, “readDynamicRfData” y “readCommandDataRf”, éstos funcionan de manera similar a su contraparte para enviar datos, sin embargo, debido a su naturaleza asíncrona (no se sabe exactamente cuándo se puede recibir un dato) será necesario realizar un muestreo continuo o “polling”, ésto consiste en que cada determinado tiempo se verifique si se tiene algún dato recibido, de ser así, el método regresará un valor booleano de True y se podrán leer los datos obtenidos. Se recomienda utilizar la clase “Timer” de las librerías nativas de Java, la cuál permite asignar un temporizador y realizar ejecuciones de código repetitivas en intervalos de tiempo definidos.

Para mayor referencia sobre todos los métodos disponibles y su utilización, se recomienda revisar la documentación API (Ver Apéndice 7). Todos los archivos utilizados para este trabajo, incluyendo librerías, documentación y archivos de programación están disponibles para su descarga desde la página web [http://sourceforge.net/projects/usb-rf-avr/files/].

6.3.1 Implementación física

A continuación veremos cómo implementar físicamente la comunicación inalámbrica al sistema de control y posteriormente la programación del microcontrolador ATMEGA8 encargado del flujo de los datos inalámbricos y la comunicación USB.

La primera parte consiste en ajustar el sistema de control que se tenga, para poder lograr una comunicación con el transceptor nRF24L01+, para esto será necesario adaptar el sistema, en caso que el sistema cuente con la interfaz SPI se podrá realizar la comunicación directa con el transceptor, de no ser así, se tendrá que adicionar un dispositivo que se encargue del flujo de datos entre el transceptor y el sistema de control.

Para el primer caso, se deberá adicionar el transceptor nRF24L01+ al sistema de control y comunicarse con éste por medio de la interfaz SPI. Debido a las diferentes posibilidades en el diseño del sistema de control se recomienda revisar las hojas de especificaciones del nRF24L01+ y el Capítulo 3 de este trabajo, para conocer las condiciones de operación y la forma de interacción con el transceptor. Sí el sistema de control cuenta con un microcontrolador AVR, se pueden utilizar las librerías creadas en el proyecto (Ver Apéndice 2 y Apéndice 3), de no ser así, se tendrán que generar o adaptar las librerías de acuerdo al dispositivo que tenga el sistema de control, otra opción es la de utilizar librerías disponibles en sitios web, que sean compatibles con el dispositivo que se cuente.

Para el segundo caso, se tendrá que adicionar un dispositivo que sirva como puente entre la interfaz SPI del transceptor nRF24L01+ y alguna interfaz de comunicación disponible en el sistema de control, para esto existen varias posibilidades de acuerdo al dispositivo que se ocupe y de las interfaces de comunicación disponibles en el sistema de control (RS232, I^2C, UART, etc.). Debido a esto, se recomienda buscar un dispositivo tipo puente o “Bridge” que sólo se encargue de realizar el intercambio de datos de una interfaz a otra, ya que así se evita añadir más programación de la necesaria, de otra manera se tendrá que utilizar un dispositivo programable como un microcontrolador. Sí se opta por añadir un microcontrolador, se recomienda utilizar un AVR para así poder utilizar las librerías generadas en este proyecto (Ver Apéndice 2: Archivo de cabecera *(NRF24L01.h)* y Apéndice 3), de no ser así, se tendrán que generar las librerías para tal dispositivo.

Para cualquiera de los dos casos, una vez que se logre la comunicación con el transceptor nRF24L01+ y el sistema de control, se deberán configurar los parámetros del dispositivo, tales como: frecuencia y velocidad de transmisión, dirección de envío y recepción y la cantidad de datos por mensaje para envío y recepción, o en su defecto habilitar la longitud de datos dinámica. Todo esto para poder realizar la comunicación inalámbrica adecuada, con el transceptor nRF24L01+ que se encuentra del lado de la computadora.

En el sistema de control se deberá realizar toda la programación necesaria para el envío y recepción de datos, para tal caso se recomienda utilizar los métodos que ofrece el dispositivo nRF24L01+, tal como la interrupción por medio del pin IRQ, ésta permite en modo receptor conocer cuándo se recibe un dato y en modo transmisor determina si un dato se envió correctamente.

Debido a que cada sistema es diferente, será necesario tener en cuenta las necesidades del mismo para determinar el mejor método para comunicarse, ya sea si se requiere una longitud fija de

datos, o si se requiere mayor velocidad de transferencia al costo de un menor alcance en distancia, utilizar un canal de radiofrecuencia que no tenga tanta interferencia en el ambiente que se encuentre, cambiar al modo transmisor y receptor de acuerdo a las necesidades del mismo o permanecer siempre como modo receptor y utilizar datos en el acuse de recibo para evitar cambiar a modo transmisor y realizar una comunicación más fluida. Será necesario acudir a las hojas de datos del nRF24L01+ para información detallada del mismo, de igual forma se puede consultar el Capítulo 3 de este trabajo en donde se encuentra un resumen de lo más relevante sobre el mismo.

Finalmente, se debe incorporar el sistema mostrado en la Figura 6.2 el cuál es el encargado de la comunicación entre la computadora y la comunicación inalámbrica, si se desea realizar el diseño mostrado, será necesario mandar manufacturar dichas placas y soldar sus componentes, sin embargo, debido a que los componentes del diseño permiten reemplazarse por el tipo “through hole”, se puede cambiar el diseño y elaborar distintas placas de circuito impreso, o en su defecto se pueden realizar pruebas desde una “protoboard”.

Una vez que se tenga armado el circuito, se procede a cargar el archivo de programación de formato “.hex” al microcontrolador AVR. Para este propósito, utilizaremos un programador serial que tenga la interfaz ICSP, de esta manera se podrá programar el dispositivo sin desconectar algún componente. Cada programador utiliza distintas formas de realizar este proceso, debido a esto, se recomienda seguir las instrucciones que indica el manual del producto. Para la programación se deberá cargar el archivo “*main.hex*” al dispositivo y utilizar como fusibles los siguientes valores:

- Low Fuse : 0xFF
- High Fuse: 0xC1
- Lock: 0xFF

Estos fusibles permiten configurar características adicionales del microcontrolador y será necesario que tenga estos valores para que el dispositivo funcione correctamente, estos valores se configuran desde el mismo programador.

Opcionalmente, se recomienda cargar el archivo de configuración inicial del transceptor nRF24L01+, almacenado en la EEPROM, para esto se debe cargar el archivo “*init_val.eep*” en conjunto con el archivo “*main.hex*”. Estos parámetros iniciales también se pueden configurar desde la interfaz en Java y se pueden ajustar dinámicamente a las necesidades del sistema.

El proceso de carga del programa y datos iniciales a la memoria EEPROM se ilustra en la Figura 6.6.

Finalmente, el diseñador del sistema de control deberá realizar la interfaz de usuario en Java mediante las librerías creadas en este trabajo, teniendo la posibilidad de añadir librerías creadas por otros, además de las nativas de Java. Esto permite tener mayor flexibilidad y así poder adaptarse a las necesidades del sistema, además de lograr una interfaz de usuario más intuitiva y amigable.

En el siguiente apartado se mostrarán algunas pruebas realizadas al sistema, para posteriormente analizar los resultados y dar a conocer las conclusiones del proyecto.

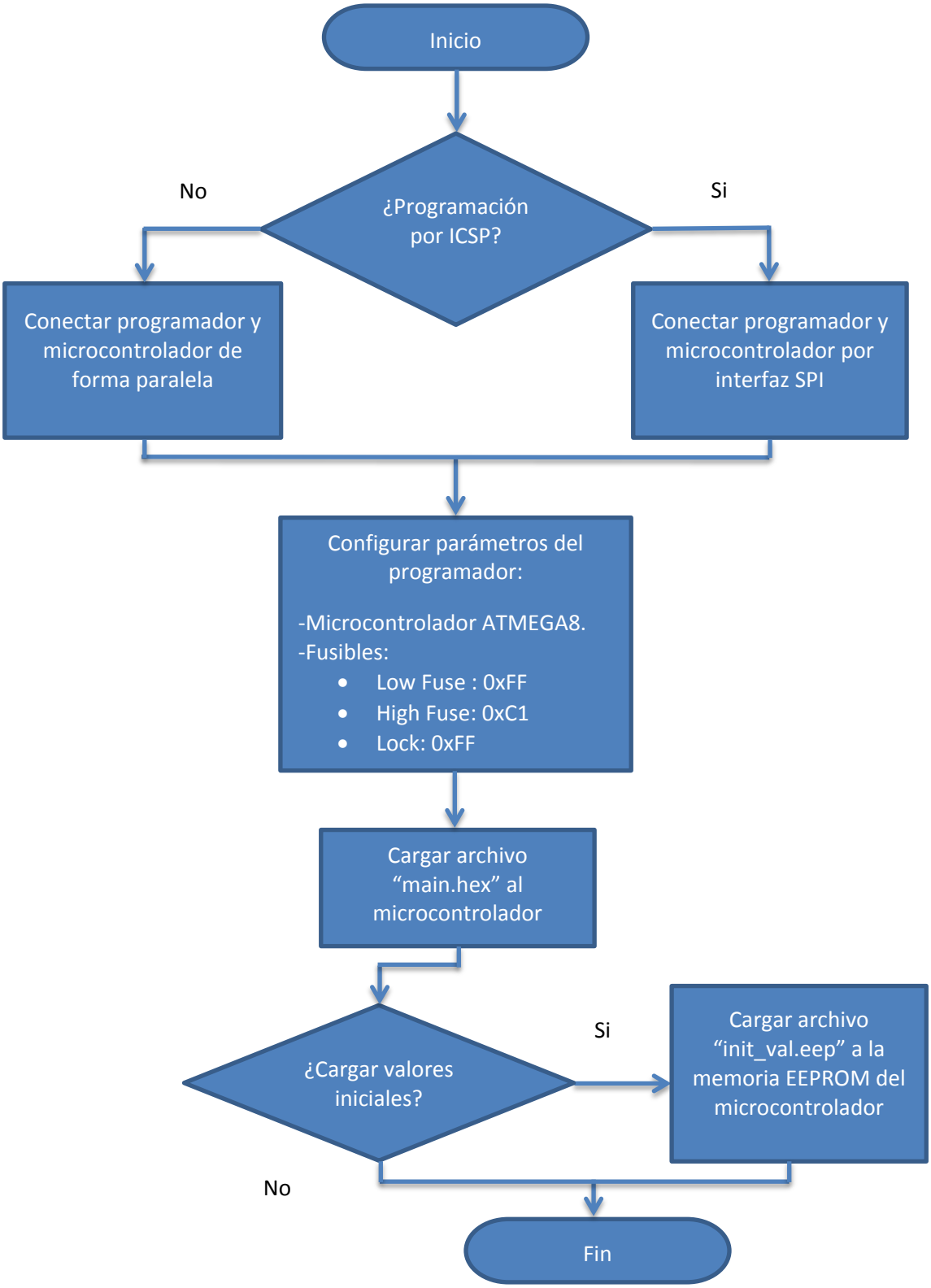

Figura 6.6.- Diagrama de bloques para programar el microcontrolador utilizado para la interfaz de comunicación con la computadora.

Capítulo 7
Metodología Experimental

En el presente capítulo se realizan diversas pruebas al sistema, para posteriormente analizar el alcance de la interfaz diseñada.

A continuación se tratarán una serie de pruebas realizadas a la interfaz desarrollada, para posteriormente analizar los resultados obtenidos y concluir el proyecto.

Para las pruebas a realizar, se consideró analizar la comunicación inalámbrica, tomando en cuenta los siguientes parámetros:

- Distancia o alcance, para poder determinar el rango máximo que podemos obtener antes de perder la comunicación.
- Cantidad de paquetes no entregados o fallidos, de esta forma analizar las interferencias y retransmisiones que a su vez generan tiempos muertos.

Para obtener un mejor análisis, se modificarán las distintas variables que puedan afectar el comportamiento de los resultados, tales como la velocidad de transmisión, canal de frecuencia, cantidad de datos a enviar, además ,se harán pruebas utilizando los módulos de radiofrecuencia con antena externa y sin antena externa.

Debido a que es necesario contar con otro dispositivo que nos permita realizar las pruebas de comunicación inalámbrica, se utilizó la tarjeta de desarrollo Arduino Uno, ya que cuenta con todas las herramientas necesarias para la comunicación con el transceptor, además de ser de bajo costo y permite realizar cambios rápidamente desde su interfaz de programación haciéndolo ideal para hacer las pruebas necesarias.

Antes de continuar a definir las pruebas realizadas, describiremos algunas generalidades sobre la tarjeta de desarrollo Arduino, utilizada como dispositivo para realizar las pruebas pertinentes.

7.1 TARJETA DE DESARROLLO ARDUINO

Arduino es una herramienta de desarrollo que permite medir y controlar diversos procesos en el mundo físico. Es una plataforma de código abierto (open source) basada en una marca específica de microcontroladores y una plataforma de desarrollo para escribir software directamente sobre la placa. [13]

Arduino puede ser usada para desarrollar objetos interactivos, tomando entradas de distintos sensores o interruptores y también controlar diversas salidas como motores, luces, relés, entre otros. Los proyectos desarrollados en Arduino pueden ser autónomos (stand-alone), o pueden comunicarse por medio de Software a través de la computadora (ejemplo: Flash, Processing, MaxMSP). Las placas pueden ser ensambladas a mano o compradas ya ensambladas, la interfaz de programación es de uso libre y se descarga de manera gratuita.

El lenguaje de programación Arduino es una implementación de Wiring, la cual es una plataforma de computación física basada en el ambiente de programación de multimedia denominado Processing.

Existen otras plataformas basadas en microcontroladores disponibles para computación física, tales como Parallax Basic Stamp, Netmedia’s BX-24, Phidgets, Handyboard, entre otras. Todas ofrecen herramientas de desarrollo similares, sin embargo, toman la misma programación

utilizada en microcontroladores y la envuelven en una interfaz más amigable. Arduino simplifica todo el proceso de trabajar con microcontroladores y además ofrece ventajas para maestros, estudiantes e incluso aficionados. Las ventajas sobre otros sistemas son:

- Barato – Las tarjetas Arduino son relativamente baratas en comparación con otras plataformas. Las versiones más baratas de Arduino pueden ser ensambladas a mano, e incluso algunos módulos ensamblados cuestan menos de $350.00 MXN.

- Multi-plataforma – El Software de Arduino corre en sistemas operativos Windows, Macintosh OSX y Linux. La mayoría de sistemas basados en microcontrolador están limitados sólo a Windows.

- Ambiente de programación simple y claro – El ambiente de programación de Arduino es fácil de usar para principiantes, además de ser suficientemente flexible para usuarios avanzados.

- Código abierto y software extensible –El software de Arduino es de código abierto, permitiendo ser extendido por programadores experimentados. El lenguaje puede ser expandido a través de librerías en C++. Aquellos que deseen entender los detalles técnicos pueden saltar de Arduino al lenguaje de programación AVR-C en el cual está basado. De manera similar se puede agregar código AVR-C directamente al programa Arduino si se desea.

- Hardware extensible – Las placas Arduino están basadas en los microcontroladores Atmel (Por ejemplo: ATMEGA328, ATMEGA32u, etcétera). Los diagramas de los módulos están publicados bajo una licencia libre, por lo que diseñadores de circuitos experimentados pueden crear su propia versión del módulo, extendiéndolo o mejorándolo. Incluso usuarios sin mucha experiencia pueden armar el circuito desde una protoboard, ya sea para entender cómo funciona o ahorrar dinero.

La plataforma Arduino cuenta con diversas placas de desarrollo para distintas necesidades, algunos ejemplos de estas son: Arduino Uno, Mega, Duemilanove, Fio, Lilypad, Diecimlia, entre otros. Entre los más populares se encuentra la tarjeta Arduino Uno, mostrada en la Figura 7.1.

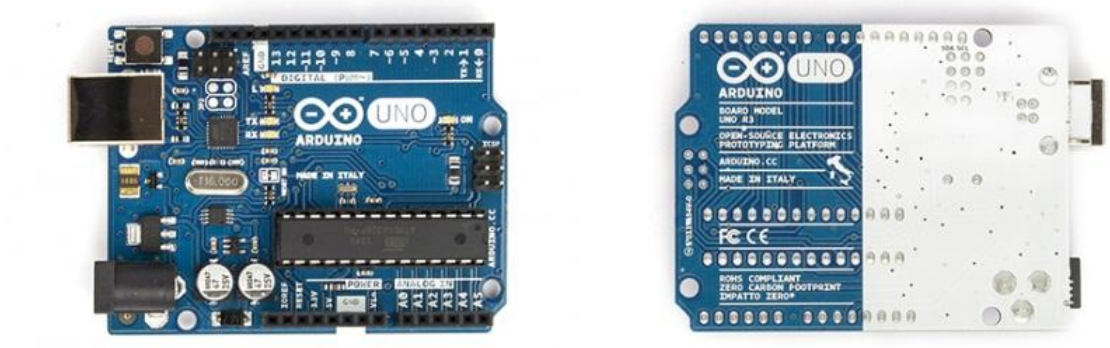

Figura 7.1.- Tarjeta de desarrollo Arduino Uno (imagen de lado izquierdo vista superior, imagen del lado derecho vista inferior).[29]

[29] Fuente: http://arduino.cc/en/Main/arduinoBoardUno

La tarjeta Arduino Uno está basada en el microcontrolador ATMEGA328, cuenta con 14 pines de entrada/salida (de los cuales 6 pueden usarse como salidas PWM), 6 entradas analógicas, un cristal de 16MHz, una conexión USB, un conector tipo Jack para alimentación, un conector tipo header para interfaz ICSP (In Circuit Serial Programming) y un botón de reset. Además de esto, contiene todo lo necesario para soportar al microcontrolador, sólo es necesario conectarlo a la computadora con el cable USB o alimentarlo mediante un adaptador de voltaje AC – DC o una batería para hacerlo funcionar.

Todas las tarjetas Arduino pueden ser programadas desde el software Arduino (disponible para su descarga gratuita desde: [http://arduino.cc/en/Main/Software]), el microcontrolador incluido en las tarjetas Arduino viene precargado con un gestor de arranque o "bootloader" que permite la carga de nuevo código al dispositivo sin utilizar un programador externo.

7.2 PRUEBAS REALIZADAS

A continuación definiremos las pruebas realizadas al sistema y se mostrarán los resultados de las mismas, para posteriormente ser analizados y dar una conclusión al respecto.

Se consideraron dos parámetros a evaluar para las pruebas realizadas, estos fueron los siguientes:

- Distancia máxima posible entre los transceptores antes de perder la comunicación. Ésta se considera en donde el receptor ya no logra recibir ningún paquete de datos en un lapso no mayor a 500 milisegundos, tomando en cuenta que se envía un paquete de datos cada 10 milisegundos, es decir, se han enviado 50 paquetes y no se ha podido recibir ninguno.

- Porcentaje de paquetes recibidos correctamente a una distancia aproximada de 50 metros. Para determinar este valor, se utiliza la retroalimentación entre los dispositivos o acuse de recibo (ACK), como referencia para medir la cantidad de paquetes recibidos.

Para realizar esto se tomaron en cuenta algunas variables que pudieran afectar los resultados de estos dos parámetros y que pueden configurarse desde el transceptor, estos son:

- Canal de radiofrecuencia. Esto permite conocer qué canales ofrecen una mejor recepción, aunque esto también dependerá de la interferencia en el ambiente.

- Cantidad de datos por mensaje enviado. De esta forma se podrá ver el impacto en distancia que tiene al enviar paquetes de diferentes longitudes (número de Bytes por paquete)[30].

- Velocidad de transmisión. De acuerdo a las hojas de especificación del transceptor, éste parámetro afecta de manera directa en la distancia alcanzada, por lo que se analizará su impacto.

[30] Para el caso de la prueba de porcentaje de paquetes recibidos, se realizó a 16 Bytes por paquete, ya que es una cantidad intermedia, considerando que el máximo de datos permitidos por paquete es de 32 Bytes.

- Tipo de antena utilizada en el transceptor (interna o externa). Debido a que el dispositivo permite estas dos posibilidades en el transceptor, se realizarán las pruebas utilizando dispositivos con antena interna (integrada en el PCB) y con antena externa.

Las pruebas aquí mostradas se realizaron en las instalaciones de la Facultad de Estudios Superiores Cuautitlán Campo 4, en un espacio abierto con pocos obstáculos entre ambos transceptores, además, no se hizo ningún intento de aislar los dispositivos de otras posibles interferencias en el ambiente, debido a esto, los valores obtenidos sólo permiten ser una referencia aproximada y pueden diferir de acuerdo a la ubicación y ambiente de operación.

Cabe hacer notar que las distancias mostradas son aproximadas y se obtuvieron mediante una aplicación para dispositivos celulares denominada "GPS Tape Measure"[31], ésta utiliza la geo localización (GPS) para determinar la distancia entre dos puntos y de acuerdo a su desarrollador, tiene un rango de error de ± 8m en condiciones ideales.

Otra consideración a tomar en cuenta, es que en las pruebas realizadas no hubo verificación alguna de la integridad de los datos, es decir, el transceptor "device" no comprobó que el contenido de los datos recibidos haya sido el correcto, de acuerdo a lo enviado por el transceptor "host", sin embargo, se considera cubierto debido al código de detección de errores de comprobación de redundancia cíclica (CRC), incluido automáticamente en todos los paquetes enviados por el transceptor nRF24L01.

7.2.1 Prueba de distancia máxima alcanzada

El procedimiento para realizar la prueba de distancia máxima alcanzada, fue el siguiente:

1. Se conectaron los transceptores con antena interna en cada dispositivo, uno conectado a la computadora con el sistema desarrollado en esta tesis, a éste lo denominaremos "host", y otro transceptor conectado a la placa Arduino, al que llamaremos "device".

2. Se colocó en un lugar fijo el transceptor "host" y se tomó este punto como referencia para medir la distancia.

3. El transceptor "host" se configuró como transmisor mientras que el "device", se configuró como receptor.

4. Se fijó un valor de frecuencia, velocidad de datos y cantidad de datos por paquete en ambos dispositivos transceptores.

5. Se inició la transmisión de paquetes de datos continua, con un espacio entre cada paquete de 10 milisegundos.

[31] Disponible de forma gratuita para su descarga en dispositivos Android desde la página web: [https://play.google.com/store/apps/details?id=com.gpstapemeasure]

6. El transceptor “device”, se alejó gradualmente del punto de referencia y con ayuda de un LED indicador conectado al mismo[32] y una aplicación móvil de geo-posicionamiento, se determinó la distancia máxima alcanzada.

7. Se repitió todo el proceso desde el punto 3, para cada valor indicado en la Tabla 7.1.

8. Se realizó todo el proceso nuevamente, pero esta vez utilizando transceptores con antena externa.

7.2.2 Prueba de paquetes exitosos

El procedimiento para realizar la prueba de porcentaje de paquetes exitosos a 50 metros, fue el siguiente:

1. Se conectaron los transceptores con antena interna en cada dispositivo, uno conectado a la computadora con el sistema desarrollado en esta tesis, a éste lo denominaremos “host”, y otro transceptor conectado a la placa Arduino, al que llamaremos “device”.

2. Se colocó en un lugar fijo el transceptor “host” y se tomó este punto como referencia para medir la distancia.

3. Se fijó un valor de frecuencia, velocidad de datos y cantidad de datos por paquete en ambos dispositivos transceptores.

4. Se colocó el transceptor “device” a una distancia aproximada de 50 metros desde el punto de referencia.

5. Se inició el proceso de envío de datos, este proceso consistió en una transmisión continua de 1,000 paquetes de datos espaciados entre sí 10 milisegundos.

6. Con ayuda de una interfaz realizada en Java (ver Figura 7.2) y los acuses de recibo (ACK) del transceptor, se registró el total de datos enviados satisfactoriamente por el “host”, el programa contabilizó y calculó el porcentaje de paquetes exitosos.

7. Se repitió el proceso desde el paso 3, para todos los valores de la Tabla 7.2.

8. Se realizó todo el proceso nuevamente, pero esta vez utilizando transceptores con antena externa.

En el capítulo siguiente se analizan los datos aquí obtenidos y se dan a conocer las conclusiones sobre el sistema desarrollado en esta Tesis.

[32] El LED se configuró en el dispositivo Arduino para que encendiera al recibir un paquete de datos exitoso y se apagara automáticamente después de no recibir algún dato dentro de un lapso de 500 milisegundos.

Antena	Canal (MHz)	Datos por paquete (Bytes)	Velocidad de transmisión	Distancia máxima alcanzada (m)
Interna	2400	2	250 Kbps	78
			1 Mbps	50
			2 Mbps	36
		8	250 Kbps	81
			1 Mbps	50
			2 Mbps	43
		16	250 Kbps	72
			1 Mbps	47
			2 Mbps	33
		32	250 Kbps	70
			1 Mbps	49
			2 Mbps	31
	2430	2	250 Kbps	83
			1 Mbps	60
			2 Mbps	38
		8	250 Kbps	81
			1 Mbps	62
			2 Mbps	35
		16	250 Kbps	82
			1 Mbps	56
			2 Mbps	34
		32	250 Kbps	70
			1 Mbps	53
			2 Mbps	33
	2460	2	250 Kbps	73
			1 Mbps	69
			2 Mbps	38
		8	250 Kbps	70
			1 Mbps	65
			2 Mbps	19
		16	250 Kbps	70
			1 Mbps	60
			2 Mbps	21
		32	250 Kbps	65
			1 Mbps	54
			2 Mbps	25
	2500	2	250 Kbps	65
			1 Mbps	60
			2 Mbps	13
		8	250 Kbps	63

Antena	Canal (MHz)	Datos por paquete (Bytes)	Velocidad de transmisión	Distancia máxima alcanzada (m)
Interna	2500	8	1 Mbps	51
			2 Mbps	14
		16	250 Kbps	65
			1 Mbps	49
			2 Mbps	15
Externa	2400	2	250 Kbps	144
			1 Mbps	85
			2 Mbps	65
		8	250 Kbps	150
			1 Mbps	88
			2 Mbps	68
		16	250 Kbps	150
			1 Mbps	90
			2 Mbps	71
		32	250 Kbps	150
			1 Mbps	102
			2 Mbps	72
	2430	2	250 Kbps	180
			1 Mbps	130
			2 Mbps	80
		8	250 Kbps	170
			1 Mbps	125
			2 Mbps	73
		16	250 Kbps	170
			1 Mbps	127
			2 Mbps	73
		32	250 Kbps	165
			1 Mbps	105
			2 Mbps	68
	2460	2	250 Kbps	240
			1 Mbps	108
			2 Mbps	75
		8	250 Kbps	220
			1 Mbps	130
			2 Mbps	80
		16	250 Kbps	210
			1 Mbps	124
			2 Mbps	95
		32	250 Kbps	205
			1 Mbps	138

Antena	Canal (MHz)	Datos por paquete (Bytes)	Velocidad de transmisión	Distancia máxima alcanzada (m)
Externa	2460	32	2 Mbps	110
	2500	2	250 Kbps	129
			1 Mbps	115
			2 Mbps	65
		8	250 Kbps	165
			1 Mbps	130
			2 Mbps	66
		16	250 Kbps	173
			1 Mbps	131
			2 Mbps	68
		32	250 Kbps	182
			1 Mbps	131
			2 Mbps	71

Tabla 7.1.- Valores de distancia máxima alcanzada (medidos en el transceptor nRF24L01).

Antena	Canal (MHz)	Datos por paquete (Bytes)	Velocidad de transmisión	% de paquetes correctos a 50 m.
Interna	2400	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	0%
	2430	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	91.9%
	2460	16	250 Kbps	95.4%
			1 Mbps	94.7%
			2 Mbps	96.3%
	2500	16	250 Kbps	100%
			1 Mbps	93.1%
			2 Mbps	68.3%
Externa	2400	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	100%
	2430	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	100%
	2460	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	100%
	2500	16	250 Kbps	100%
			1 Mbps	100%
			2 Mbps	100%

Tabla 7.2.- Valores calculados de porcentaje de paquetes recibidos correctamente a una distancia de 50 metros entre transceptores.

Tipo de Antena	Velocidad de transmisión	Distancia Mínima alcanzada (m)	Distancia máxima alcanzada (m)	Distancia promedio (m)
Interna	255 Kbps	61	83	71.81
	1 Mbps	30	69	54
	2 Mbps	13	43	28.25
Externa	255 Kbps	129	240	175.19
	1 Mbps	85	138	116.19
	2 Mbps	65	110	75

Tabla 7.3.- Valores promedio de distancia de acuerdo a la Tabla 7.1.

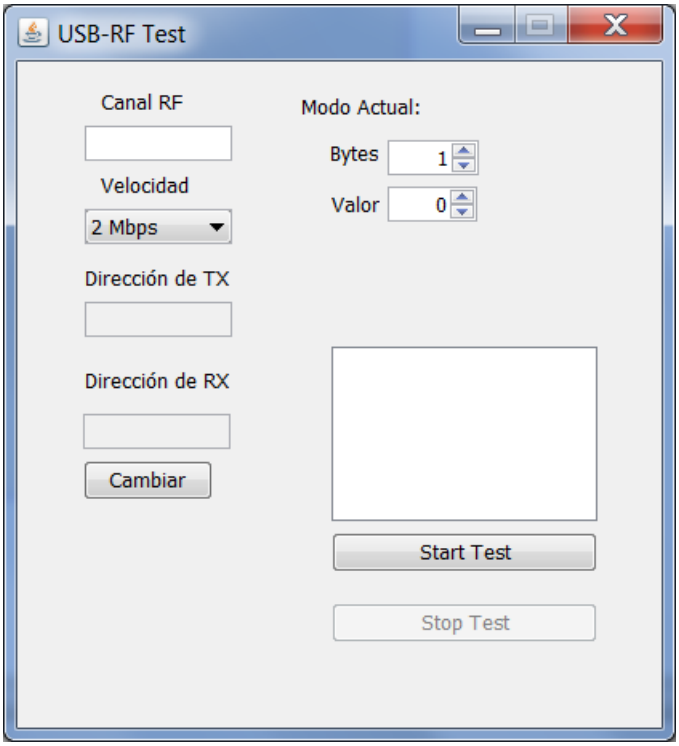

Figura 7.2.- Interfaz de programa realizada en Java para medir el porcentaje de paquetes correctos.

Capítulo 8
Resultados y Conclusiones

En el siguiente capítulo se analizan los resultados obtenidos y se dan a conocer las conclusiones del proyecto de acuerdo al objetivo planteado.

De acuerdo a los resultados obtenidos en los capítulos anteriores, se analiza el alcance y se dan a conocer las conclusiones sobre el proyecto desarrollado en esta Tesis.

8.1 ANÁLISIS DE RESULTADOS

Examinando los resultados obtenidos en las pruebas realizadas del capítulo anterior (Tabla 7.1 y Tabla 7.2), podemos resumir los siguientes puntos:

- La distancia máxima permitida tiene una relación directa con la velocidad de transmisión, tal como lo indican las hojas de datos del transceptor. De acuerdo a los valores obtenidos, en promedio se obtiene una ganancia en distancia de aproximadamente el 60% en velocidades de 1Mbps y hasta del 100% para velocidades de 255 Kbps, tomando como comparación una velocidad de 2Mbps. La Tabla 7.3 muestra algunos promedios para distintas configuraciones de velocidad.

- El canal de radiofrecuencia parece no afectar en gran medida los valores de distancia, aunque, los valores más bajos se encontraron en la frecuencia más alta probada, que fue de 2500 MHz, además, se obtuvo un mayor alcance para valores de frecuencia entre 2430-2460 MHz. Hay que recordar que el canal de radiofrecuencia se verá afectado por la ubicación y el entorno en donde se encuentre el dispositivo, además hay que tomar en cuenta las regulaciones de telecomunicaciones de cada país para la selección del canal.

- El número de datos (Bytes) por paquete no contribuyó de manera importante en las pruebas de distancia, por lo que podemos considerarlo irrelevante en este ámbito, aunque habrá que tomar en cuenta que a mayor número de Bytes por paquete también aumenta el tiempo de transmisión, para mayor detalle referirse a la sección 3.1.12.

- El utilizar una antena externa en el dispositivo transceptor aumenta ampliamente la distancia alcanzada, mejorando el alcance aproximadamente al doble de lo que se podría con una antena interna, llegando a alcanzar distancias incluso superiores a los 200 metros en velocidades de 255Kbps.

- De acuerdo a la prueba de porcentaje de paquetes correctos, podemos determinar que para una distancia entre transceptores mayor o igual a 50 metros, velocidades de 1Mbps y 255Kbps de transmisión presentan una comunicación sin pérdidas, y para velocidades de 2Mbps habrá pérdida de datos e incluso una comunicación nula, sin embargo, esto puede compensarse utilizando una antena externa que permita aumentar el rango de distancia.

8.2 ANÁLISIS DE COSTOS

Otro aspecto importante a examinar, es el de determinar el costo promedio que implica implementar el sistema propuesto, para esto se tomará en cuenta el material utilizado.

La Tabla 8.1 muestra el total de costos aproximados involucrados en el sistema, debido a que el diseño del sistema permite cierta flexibilidad, se incluyeron las opciones que puedan ver afectado el costo.

Cant.	Material	Costo Unitario (MXN)	Costo Total (MXN)
2	(Opción 1) Transceptor nRF24L01+ con antena interna	$40.00	$80.00
2	(Opción 2) Transceptor nRF24L01+ con antena externa	$130.00	$260.00
2	Capacitor de 100nF	$2.00	$4.00
1	Capacitor de 10uF	$3.00	$3.00
2	Capacitor de 22pF	$1.00	$2.00
1	Conector Header Hembra Doble	$15.00	$15.00
1	Conector USB Macho Tipo "A"	$10.00	$10.00
1	Costos indirectos (Soldadura, etc.)	$25.00	$25.00
1	Cristal de 12 MHZ	$12.00	$12.00
2	Diodos Zener	$1.00	$2.00
2	Led	$3.00	$6.00
1	Microcontrolador ATMEGA 8	$50.00	$50.00
1	Placa PCB doble capa	$58.70	$58.70
1	Regulador de Voltaje de 3.3V	$10.00	$10.00
1	Resistencia de 1.5KΩ	$1.00	$1.00
1	Resistencia de 10KΩ	$1.00	$1.00
2	Resistencia de 1KΩ	$1.00	$2.00
2	Resistencia de 68Ω	$1.00	$2.00
		Total con Opción 1	**$283.70**
		Total con Opción 2	**$487.40**

Tabla 8.1.- Tabla de costos del sistema diseñado.

Al comparar este sistema con otros disponibles en el mercado, como por ejemplo el ATAVRUSBRF01 de ATMEL®, éste tiene un costo en el mercado de $94.00 USD o aprox. 1,220.00 MXN (tomando una tasa de conversión de 13 MXN). Se observa que el sistema desarrollado es más barato y además permite mayor flexibilidad para el diseño.

8.3 CONCLUSIONES

En general se considera que los resultados obtenidos cumplieron los requerimientos de la comunicación inalámbrica, aunque al final dependerá de las necesidades del sistema de control así como el entorno en que éste se encuentre, se recomienda utilizar velocidades bajas (1 Mbps o 255Kbps) para distancias largas (mayores a 50 metros) o con muchos obstáculos, sí la velocidad y la distancia es crítica, se puede optar por utilizar transceptores con antena externa y una velocidad de 2Mbps.

La interfaz USB utilizada permite tener una mayor compatibilidad para distintos sistemas debido a que es un estándar ampliamente utilizado, el único inconveniente serán los controladores o drivers que deberán instalarse en el sistema, adicionalmente, debido a que la versión USB utilizada es la 1.1, esto limita la velocidad, en comparación con sus posteriores versiones y deberá ser considerada para la aplicación final. En general se considera que el sistema cubre una amplia gama de aplicaciones, sin embargo, no se considera apto para aquellas en la que es necesario enviar una gran cantidad de datos a velocidades mayores a las permitidas por la interfaz USB versión 1.1, aplicaciones como transmisión de video o audio, por ejemplo.

Aunque el sistema obliga a que la interfaz de usuario sea desarrollada en el lenguaje de programación Java, el utilizar este lenguaje ofrece diversas ventajas ya que permite tener flexibilidad en cuanto al sistema operativo a utilizar, además de tener una amplia gama de librerías, soporte y actualizaciones, entorno de programación gráfico, entre otras cosas. Aunque el desarrollo en esta Tesis se enfocó sobre el sistema operativo Microsoft® Windows®, se espera que en un futuro se pueda probar en otros sistemas operativos.

Los costos pueden variar entre $300.00 a $500.00 MXN dependiendo el tipo de transceptor utilizado (antena interna o externa) y de los dispositivos que se utilicen (montaje superficial o “through hole”), estos costos siguen siendo mucho menores comparados con otras opciones, que oscilan entre los $1,000 y $1,500 MXN, en donde utilizan un software propio para su desarrollo, por lo que no permiten flexibilidad pero con la única ventaja de que utilizan el USB versión 2.0.

A futuro se espera que este trabajo permita a ingenieros y desarrolladores que deseen ampliar o mejorar su sistema al añadir una comunicación inalámbrica y software SCADA desde la computadora, esto implica realizar algunas modificaciones y aumentar el hardware, pero el beneficio que se obtendría justificaría los costos adicionados y agregaría un valor importante al proyecto final.

Se espera que este proyecto sea un punto de partida para futuros proyectos propios y que también permita serlo para otros ingenieros y desarrolladores.

Apéndices

TABLA DE APÉNDICES

APÉNDICE 1: MAPA DE REGISTROS DEL NRF24L01+[33]

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
00	CONFIG				Configuration Register
	Reserved	7	0	R/W	Only '0' allowed
	MASK_RX_DR	6	0	R/W	Mask interrupt caused by RX_DR 1: Interrupt not reflected on the **IRQ** pin 0: Reflect RX_DR as active low interrupt on the **IRQ** pin
	MASK_TX_DS	5	0	R/W	Mask interrupt caused by TX_DS 1: Interrupt not reflected on the **IRQ** pin 0: Reflect TX_DS as active low interrupt on the **IRQ** pin
	MASK_MAX_RT	4	0	R/W	Mask interrupt caused by MAX_RT 1: Interrupt not reflected on the **IRQ** pin 0: Reflect MAX_RT as active low interrupt on the **IRQ** pin
	EN_CRC	3	1	R/W	Enable CRC. Forced high if one of the bits in the EN_AA is high
	CRCO	2	0	R/W	CRC encoding scheme '0' - 1 byte '1' – 2 bytes
	PWR_UP	1	0	R/W	1: POWER UP, 0:POWER DOWN
	PRIM_RX	0	0	R/W	RX/TX control 1: PRX, 0: PTX
01	EN_AA Enhanced ShockBurst™				Enable 'Auto Acknowledgment' Function Disable this functionality to be compatible with nRF2401, see page 75
	Reserved	7:6	00	R/W	Only '00' allowed
	ENAA_P5	5	1	R/W	Enable auto acknowledgement data pipe 5
	ENAA_P4	4	1	R/W	Enable auto acknowledgement data pipe 4
	ENAA_P3	3	1	R/W	Enable auto acknowledgement data pipe 3
	ENAA_P2	2	1	R/W	Enable auto acknowledgement data pipe 2
	ENAA_P1	1	1	R/W	Enable auto acknowledgement data pipe 1
	ENAA_P0	0	1	R/W	Enable auto acknowledgement data pipe 0
02	EN_RXADDR				Enabled RX Addresses
	Reserved	7:6	00	R/W	Only '00' allowed
	ERX_P5	5	0	R/W	Enable data pipe 5.
	ERX_P4	4	0	R/W	Enable data pipe 4.
	ERX_P3	3	0	R/W	Enable data pipe 3.
	ERX_P2	2	0	R/W	Enable data pipe 2.

[33] Extracto de las hojas de especificación del transceptor nRF24L01+, Rev. 1.0, Septiembre 2008, páginas 57-63.

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
	ERX_P1	1	1	R/W	Enable data pipe 1.
	ERX_P0	0	1	R/W	Enable data pipe 0.
03	SETUP_AW				Setup of Address Widths (common for all data pipes)
	Reserved	7:2	000000	R/W	Only '000000' allowed
	AW	1:0	11	R/W	RX/TX Address field width '00' - Illegal '01' - 3 bytes '10' - 4 bytes '11' – 5 bytes LSByte is used if address width is below 5 bytes
04	SETUP_RETR				Setup of Automatic Retransmission
	ARD[a]	7:4	0000	R/W	Auto Retransmit Delay ‘0000’ – Wait 250µS ‘0001’ – Wait 500µS ‘0010’ – Wait 750µS …….. ‘1111’ – Wait 4000µS (Delay defined from end of transmission to start of next transmission)[b]
	ARC	3:0	0011	R/W	Auto Retransmit Count ‘0000’ –Re-Transmit disabled ‘0001’ – Up to 1 Re-Transmit on fail of AA …… ‘1111’ – Up to 15 Re-Transmit on fail of AA
05	RF_CH				RF Channel
	Reserved	7	0	R/W	Only '0' allowed
	RF_CH	6:0	0000010	R/W	Sets the frequency channel nRF24L01+ operates on
06	RF_SETUP				RF Setup Register
	CONT_WAVE	7	0	R/W	Enables continuous carrier transmit when high.
	Reserved	6	0	R/W	Only '0' allowed
	RF_DR_LOW	5	0	R/W	Set RF Data Rate to 250kbps. See RF_DR_HIGH for encoding.
	PLL_LOCK	4	0	R/W	Force PLL lock signal. Only used in test
	RF_DR_HIGH	3	1	R/W	Select between the high speed data rates. This bit is don’t care if RF_DR_LOW is set. Encoding: [RF_DR_LOW, RF_DR_HIGH]: ‘00’ – 1Mbps ‘01’ – 2Mbps ‘10’ – 250kbps ‘11’ – Reserved

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
	RF_PWR	2:1	11	R/W	Set RF output power in TX mode '00' – -18dBm '01' – -12dBm '10' – -6dBm '11' – 0dBm
	Obsolete	0			Don't care
07	STATUS				Status Register (In parallel to the SPI command word applied on the **MOSI** pin, the STATUS register is shifted serially out on the **MISO** pin)
	Reserved	7	0	R/W	Only '0' allowed
	RX_DR	6	0	R/W	Data Ready RX FIFO interrupt. Asserted when new data arrives RX FIFO[c]. Write 1 to clear bit.
	TX_DS	5	0	R/W	Data Sent TX FIFO interrupt. Asserted when packet transmitted on TX. If AUTO_ACK is activated, this bit is set high only when ACK is received. Write 1 to clear bit.
	MAX_RT	4	0	R/W	Maximum number of TX retransmits interrupt Write 1 to clear bit. If MAX_RT is asserted it must be cleared to enable further communication.
	RX_P_NO	3:1	111	R	Data pipe number for the payload available for reading from RX_FIFO 000-101: Data Pipe Number 110: Not Used 111: RX FIFO Empty
	TX_FULL	0	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.
08	OBSERVE_TX				Transmit observe register
	PLOS_CNT	7:4	0	R	Count lost packets. The counter is overflow protected to 15, and discontinues at max until reset. The counter is reset by writing to RF_CH. See page 75.
	ARC_CNT	3:0	0	R	Count retransmitted packets. The counter is reset when transmission of a new packet starts. See page 75.
09	RPD				
	Reserved	7:1	000000	R	
	RPD	0	0	R	Received Power Detector. This register is called CD (Carrier Detect) in the nRF24L01. The name is different in nRF24L01+ due to the different input power level threshold for this bit. See section 6.4 on page 25.
0A	RX_ADDR_P0	39:0	0xE7E7E7E7E7	R/W	Receive address data pipe 0. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
0B	RX_ADDR_P1	39:0	0xC2C2C2C2C2	R/W	Receive address data pipe 1. 5 Bytes maximum length. (LSByte is written first. Write the number of bytes defined by SETUP_AW)
0C	RX_ADDR_P2	7:0	0xC3	R/W	Receive address data pipe 2. Only LSB. MSBytes are equal to RX_ADDR_P1[39:8]
0D	RX_ADDR_P3	7:0	0xC4	R/W	Receive address data pipe 3. Only LSB. MSBytes are equal to RX_ADDR_P1[39:8]
0E	RX_ADDR_P4	7:0	0xC5	R/W	Receive address data pipe 4. Only LSB. MSBytes are equal to RX_ADDR_P1[39:8]
0F	RX_ADDR_P5	7:0	0xC6	R/W	Receive address data pipe 5. Only LSB. MSBytes are equal to RX_ADDR_P1[39:8]
10	TX_ADDR	39:0	0xE7E7E7E7E7	R/W	Transmit address. Used for a PTX device only. (LSByte is written first) Set RX_ADDR_P0 equal to this address to handle automatic acknowledge if this is a PTX device with Enhanced ShockBurst™ enabled. See page 75.
11	RX_PW_P0				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P0	5:0	0	R/W	Number of bytes in RX payload in data pipe 0 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
12	RX_PW_P1				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P1	5:0	0	R/W	Number of bytes in RX payload in data pipe 1 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
13	RX_PW_P2				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P2	5:0	0	R/W	Number of bytes in RX payload in data pipe 2 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
14	RX_PW_P3				
	Reserved	7:6	00	R/W	Only '00' allowed

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
	RX_PW_P3	5:0	0	R/W	Number of bytes in RX payload in data pipe 3 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
15	RX_PW_P4				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P4	5:0	0	R/W	Number of bytes in RX payload in data pipe 4 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
16	RX_PW_P5				
	Reserved	7:6	00	R/W	Only '00' allowed
	RX_PW_P5	5:0	0	R/W	Number of bytes in RX payload in data pipe 5 (1 to 32 bytes). 0 Pipe not used 1 = 1 byte … 32 = 32 bytes
17	FIFO_STATUS				FIFO Status Register
	Reserved	7	0	R/W	Only '0' allowed
	TX_REUSE	6	0	R	Used for a PTX device Pulse the rfce high for at least 10µs to Reuse last transmitted payload. TX payload reuse is active until W_TX_PAYLOAD or FLUSH_TX is executed. TX_REUSE is set by the SPI command REUSE_TX_PL, and is reset by the SPI commands W_TX_PAYLOAD or FLUSH_TX
	TX_FULL	5	0	R	TX FIFO full flag. 1: TX FIFO full. 0: Available locations in TX FIFO.
	TX_EMPTY	4	1	R	TX FIFO empty flag. 1: TX FIFO empty. 0: Data in TX FIFO.
	Reserved	3:2	00	R/W	Only '00' allowed
	RX_FULL	1	0	R	RX FIFO full flag. 1: RX FIFO full. 0: Available locations in RX FIFO.
	RX_EMPTY	0	1	R	RX FIFO empty flag. 1: RX FIFO empty. 0: Data in RX FIFO.

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
N/A	ACK_PLD	255:0	X	W	Written by separate SPI command ACK packet payload to data pipe number PPP given in SPI command. Used in RX mode only. Maximum three ACK packet payloads can be pending. Payloads with same PPP are handled first in first out.
N/A	TX_PLD	255:0	X	W	Written by separate SPI command TX data payload register 1 - 32 bytes. This register is implemented as a FIFO with three levels. Used in TX mode only.
N/A	RX_PLD	255:0	X	R	Read by separate SPI command. RX data payload register. 1 - 32 bytes. This register is implemented as a FIFO with three levels. All RX channels share the same FIFO.
1C	DYNPD				Enable dynamic payload length
	Reserved	7:6	0	R/W	Only '00' allowed
	DPL_P5	5	0	R/W	Enable dynamic payload length data pipe 5. (Requires EN_DPL and ENAA_P5)
	DPL_P4	4	0	R/W	Enable dynamic payload length data pipe 4. (Requires EN_DPL and ENAA_P4)
	DPL_P3	3	0	R/W	Enable dynamic payload length data pipe 3. (Requires EN_DPL and ENAA_P3)

Address (Hex)	Mnemonic	Bit	Reset Value	Type	Description
	DPL_P2	2	0	R/W	Enable dynamic payload length data pipe 2. (Requires EN_DPL and ENAA_P2)
	DPL_P1	1	0	R/W	Enable dynamic payload length data pipe 1. (Requires EN_DPL and ENAA_P1)
	DPL_P0	0	0	R/W	Enable dynamic payload length data pipe 0. (Requires EN_DPL and ENAA_P0)
1D	FEATURE			R/W	Feature Register
	Reserved	7:3	0	R/W	Only '00000' allowed
	EN_DPL	2	0	R/W	Enables Dynamic Payload Length
	EN_ACK_PAY[d]	1	0	R/W	Enables Payload with ACK
	EN_DYN_ACK	0	0	R/W	Enables the W_TX_PAYLOAD_NOACK command

a. Please take care when setting this parameter. If the ACK payload is more than 15 byte in 2Mbps mode the ARD must be 500µS or more, if the ACK payload is more than 5byte in 1Mbps mode the ARD must be 500µS or more. In 250kbps mode (even when the payload is not in ACK) the ARD must be 500µS or more. Please see section 7.4.2 on page 33 for more information.

b. This is the time the PTX is waiting for an ACK packet before a retransmit is made. The PTX is in RX mode for 250µS (500µS in 250kbps mode) to wait for address match. If the address match is detected, it stays in RX mode to the end of the packet, unless ARD elapses. Then it goes to standby-II mode for the rest of the specified ARD. After the ARD it goes to TX mode and then retransmits the packet.

c. The RX_DR IRQ is asserted by a new packet arrival event. The procedure for handling this interrupt should be: 1) read payload through SPI, 2) clear RX_DR IRQ, 3) read FIFO_STATUS to check if there are more payloads available in RX FIFO, 4) if there are more data in RX FIFO, repeat from step 1).

d. If ACK packet payload is activated, ACK packets have dynamic payload lengths and the Dynamic Payload Length feature should be enabled for pipe 0 on the PTX and PRX. This is to ensure that they receive the ACK packets with payloads. If the ACK payload is more than 15 byte in 2Mbps mode the ARD must be 500µS or more, and if the ACK payload is more than 5 byte in 1Mbps mode the ARD must be 500µS or more. In 250kbps mode (even when the payload is not in ACK) the ARD must be 500µS or more.

APÉNDICE 2: ARCHIVO DE CABECERA *(NRF24L01.H)*

```
/*
 *      Nombre: NRF24L01.h
 *
 *      Autor: Rubén Flores A.
 *
 *      Proyecto: Diseño de una interfaz para sistemas de control en Java
 *      con comunicación inalámbrica utilizando un microcontrolador AVR
 *      con interfaz USB.
 *
 *      Descripción: Archivo de cabecera para NRF24L01.c
 *      contiene definiciones y constantes utilizadas para
 *      el control y configuración del NRF24L01+.
 *
 */

/*  Configuración de pines. */

#define IRQ     PD3     // Interrupción del NRF24L01. Conectada a INT1.
#define IRQ_PIN PIND3
#define SPI_PORTPORTB   // Puerto del SPI en el AVR. Generalmente es el PORTB.
#define CE      PB1     // Pin15 del ATMEGA 8.  CE del NRF24L01. (configur de salida).
#define CSN     PB2     // Pin16 del ATMEGA 8.  Chip Select del NRF24L01. (configurar de salida).
#define MOSI    PB3     // Pin17 del ATMEGA 8.  MOSI del AVR (configurar de salida).
#define MISO    PB4     // Pin18 del ATMEGA 8.  MISO del AVR (configurar de entrada).
#define SCK     PB5     // Pin19 del ATMEGA 8.  SCK del AVR(configurar de salida).

#define RF_Select       SPI_PORT &= ~(1<<CSN)
#define RF_Select_Not   SPI_PORT |= (1<<CSN)

#define RF_CE_Low       SPI_PORT &= ~(1<<CE)
#define RF_CE_High      SPI_PORT |= (1<<CE)


#define Payload_Data_Pipe0      1       // Poner en 1 (No se utiliza mas que para el AA.)


/* COMANDOS nRF24L01+*/

#define R_REGISTER      0x00
#define W_REGISTER      0x20
#define R_RX_PAYLOAD    0x61
#define W_TX_PAYLOAD    0xA0
#define FLUSH_TX0xE1
#define FLUSH_RX0xE2
#define REUSE_TX_PL     0xE3
#define R_RX_PL_WID     0x60
#define W_ACK_PAYLOAD_PP0       0xA8
#define W_ACK_PAYLOAD_PP1       0xA9
#define W_ACK_PAYLOAD_PP2       0xAA
#define W_ACK_PAYLOAD_PP3       0xAB
#define W_ACK_PAYLOAD_PP4       0xAC
#define W_ACK_PAYLOAD_PP5       0xAD
#define W_TX_PAYLOAD_NO_ACK     0xB0
#define NOP_RF  0xFF
#define ACTIVATE0x50

/*      REGISTROS NRF24L01
        Usar  operador lógico | (OR) con comando R_REGISTER o W_REGISTER
        para leer o escribir respectivamente dicho registro.
*/

#define CONFIG_RF       0x00 //Configuration Register
#define EN_AA   0x01 //Enable 'Auto Acknowledgment' Function
#define EN_RXADDR       0x02 //Enabled RX Addresses
#define SETUP_AW0x03 //Enabled RX Addresses
#define SETUP_RETR      0x04 //Setup of Automatic Retransmission
```

```
#define RF_CH    0x05 //RF Channel
#define RF_SETUP0x06 //RF Setup Register
#define RF_SETUP_SPEED_1Mbps     0b00000110 // 0x06
#define RF_SETUP_SPEED_2Mbps     0b00001110 // 0x0E
#define RF_SETUP_SPEED_250Kbps   0b00100110 // 0x26
#define STATUS_RF        0x07 //Status Register
#define OBSERVE_TX       0x08 //Transmit observe register
#define RX_ADDR_P0       0x0A //Receive address data pipe 0. 5 Bytes max length.
#define RX_ADDR_P1       0x0B //Receive address data pipe 1. 5 Bytes max length.
#define RX_ADDR_P2       0x0C //Receive address data pipe 2. 5 Bytes max length.
#define RX_ADDR_P3       0x0D //Receive address data pipe 3. 5 Bytes max length.
#define RX_ADDR_P4       0x0E //Receive address data pipe 4. 5 Bytes max length.
#define RX_ADDR_P5       0x0F //Receive address data pipe 5. 5 Bytes max length.
#define TX_ADDR 0x10 //Transmit address.
#define RX_PW_P00x11 // # of bytes in RX payload in data pipe 0 (1 to 32 bytes).
#define RX_PW_P10x12 // # of bytes in RX payload in data pipe 1 (1 to 32 bytes).
#define RX_PW_P20x13 // # of bytes in RX payload in data pipe 2 (1 to 32 bytes).
#define RX_PW_P30x14 // # of bytes in RX payload in data pipe 3 (1 to 32 bytes).
#define RX_PW_P40x15 // # of bytes in RX payload in data pipe 4 (1 to 32 bytes).
#define RX_PW_P50x16 // # of bytes in RX payload in data pipe 5 (1 to 32 bytes).
#define FIFO_STATUS      0x17 // FIFO Status Register
#define DYNPD    0x1C // Enable dynamic payload length
#define FEATURE 0x1D // Feature Register


/* Bits del registro CONFIG*/
#define MASK_RX_DR       6       // Mask interrupt caused by RX_DR (Recepcion exitosa)
//1: Interrupt not reflected on the IRQ pin
//0: Reflect RX_DR as active low interrupt on the IRQ pin
#define MASK_TX_DS       5       //Mask interrupt caused by TX_DS (Transmision exitosa)
//1: Interrupt not reflected on the IRQ pin
//0: Reflect TX_DS as active low interrupt on the IRQ pin
#define MASK_MAX_RT      4  //Mask interrupt caused by MAX_RT (Max. Reintentos)
//1: Interrupt not reflected on the IRQ pin
//0: Reflect MAX_RT as active low interrupt on the IRQ pin

/* Bits del registro STATUS*/
#define RX_DR    6       //Data Ready RX FIFO interrupt. Set high
//when new data arrives RX FIFO. Write 1 to clear bit.

#define TX_DS    5       //Data Sent TX FIFO interrupt. Set high
//when packet sent on TX. If AUTO_ACK is activated, this bit will be set high only
//when ACK is received. Write 1 to clear bit.

#define MAX_RT   4       //Maximum number of TX retries interrupt
//Write 1 to clear bit. If MAX_RT is set it must be cleared to enable
//further communication.

#define RX_P_NO33        // Data pipe number for the payload

#define RX_P_NO22        // available for reading from RX_FIFO
// 000-101: Data Pipe Number
// 110: Not Used
// 111: RX FIFO Empty
#define RX_P_NO11

#define TX_FULL 0        // TX FIFO full flag. 1: TX FIFO full. 0:
// Available locations in TX FIFO.
```

APÉNDICE 3: ARCHIVO FUENTE *(NRF24L01.C)*

```
/*
 *      Nombre: NRF24L01.c
 *
 *      Autor: Rubén Flores A.
 *
 *      Proyecto: Diseño de una interfaz para sistemas de control en Java
 *      con comunicación inalámbrica utilizando un microcontrolador AVR
 *      con interfaz USB.
 *
 *      Descripción: Archivo fuente que contiene las funciones de
 *      control y configuración del NRF24L01+, requiere del archivo
 *      de cabecera NRF24L01.h
 *
 */

/* Definiciones de reloj y librerias utilizadas*/
#define F_CPU 12000000UL

#include "NRF24L01.h"
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <stdbool.h>

/*      Variables que alojan el valor de longitud de los bloques de datos de
        recepción y transmisión (para bloques de datos estáticos) */

uint8_t PP0_Length, PP1_Length;

/* Inicializa la interfaz SPI */
void SPI_Init()
{
        SPCR |= (1<<SPE)|(1<<MSTR);
}

/* Envía un dato utilizando el SPI */
unsigned char SPI_Send(unsigned char tData)
{
        unsigned char datain;
        /* Envia Transmisión SPI */
        SPDR = tData;
        /* Espera a que termine de enviar la transmision */
        while(!(SPSR & (1<<SPIF)));
        /* Obtiene valor del pin MISO. */
        datain = SPDR;
        /* Regresa el valor adquirido. */
        return datain;
}

/* Regresa el valor del registro STATUS del nRF24L01+ */
unsigned char Get_Status()
{
        unsigned char Status_temp;
        RF_Select;
        SPI_Send(R_REGISTER|STATUS_RF);
        RF_Select_Not;

        RF_Select;
        Status_temp = SPI_Send(NOP_RF);
        RF_Select_Not;
        return Status_temp;
}

/* Limpia el FIFO de TX del nRF24L01+ */
```

```
void Clean_Tx_Fifo()
{
        RF_Select;
        SPI_Send(FLUSH_TX);
        RF_Select_Not;
}

/* Limpia el valor del registro STATUS del nRF24L01+ */
void Clean_Status()
{
        unsigned char Status_temp;
        Status_temp=Get_Status();
        Status_temp &= 0x7F;
        RF_Select;
        SPI_Send(W_REGISTER|STATUS_RF);
        SPI_Send(Status_temp);
        RF_Select_Not;
}


/* Regresa el valor del registro FIFO_STATUS del nRF24L01+ */
unsigned char Get_Fifo_Status()
{
        unsigned char fifo_temp;
        RF_Select;
        SPI_Send(R_REGISTER|FIFO_STATUS);
        fifo_temp = SPI_Send(NOP_RF);
        RF_Select_Not;
        return fifo_temp;
}

/* Limpia el FIFO de RX del nRF24L01+ */
void Clean_Rx_Fifo()
{
        RF_Select;
        SPI_Send(FLUSH_RX); //Limpia FIFO RX.
        RF_Select_Not;
}

/* Entra al modo apagado del nRF24L01+ */
void Set_Power_Down_Mode()
{
        unsigned char Config_Temp;

        RF_Select;
        SPI_Send(R_REGISTER|CONFIG_RF);
        Config_Temp = SPI_Send(0xFF);
        RF_Select_Not;

        Config_Temp &= ~(1<<1);

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(Config_Temp);
        RF_Select_Not;
}

/* Entra al modo encendido del nRF24L01+ */
void Set_Power_Up_Mode()
{
        unsigned char Config_Temp;

        RF_Select;
        SPI_Send(R_REGISTER|CONFIG_RF);
        Config_Temp = SPI_Send(0x00);
        RF_Select_Not;

        Config_Temp |= (1<<1);
```

```
        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(Config_Temp);
        RF_Select_Not;
}

/* Entra al modo RX del nRF24L01+ */
bool Set_RX_Mode()
{
        RF_CE_Low;

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(0x3F);
        RF_Select_Not;

        RF_CE_High;

        _delay_us(130);

        return false;
}

/* Entra al modo TX del nRF24L01+ */
bool Set_TX_Mode()
{
        RF_CE_Low;

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(0x4E);
        RF_Select_Not;
        return true;
}

/* Configura el canal RF (valores permitidos de 0 - 127)*/
void Set_RF_Channel(uint8_t RF_CHANNEL)
{
        if(RF_CHANNEL>127 || RF_CHANNEL< 0)
        {
        return;
        }

        RF_Select;
        SPI_Send(W_REGISTER|RF_CH);
        SPI_Send(RF_CHANNEL);
        RF_Select_Not;
}

/* Regresa el canal RF*/
uint8_t Get_RF_Channel()
{
        uint8_t canal;
        RF_Select;
        SPI_Send(R_REGISTER|RF_CH);
        canal= SPI_Send(NOP_RF);
        RF_Select_Not;

        return canal;
}

/* Configura la velocidad de tranmisión RF
        Valores permitidos (definidos en nRF24L01.h) :
        RF_SETUP_SPEED_1Mbps
        RF_SETUP_SPEED_2Mbps
        RF_SETUP_SPEED_250Kbps
```

```
*/
void Set_RF_Dataspeed(uint8_t velocidad)
{
        RF_Select;
        SPI_Send(W_REGISTER|RF_SETUP);
        SPI_Send(velocidad);
        RF_Select_Not;
}

/* Regresa la velocidad de tranmisión RF
        Valores de regreso:
        2Mbps   =       1
        1Mbps   =       2
        250Kbps =       3
*/
uint8_t Get_RF_Dataspeed()
{
        uint8_t velocidad;
        RF_Select;
        SPI_Send(R_REGISTER|RF_SETUP);
        velocidad = SPI_Send(NOP_RF);
        RF_Select_Not;

        velocidad &= 0xFE;

        if (velocidad==RF_SETUP_SPEED_2Mbps)
        {
        velocidad=1;
        }
        else if (velocidad==RF_SETUP_SPEED_1Mbps)
        {
        velocidad=2;
        }
        else if (velocidad==RF_SETUP_SPEED_250Kbps)
        {
        velocidad=3;
        }

        return velocidad;

}


/* Configura la dirección de TX (5 Bytes) */
void Set_TX_Address(unsigned char ADDR_P0[5])
{
        uint8_t i;

        RF_Select;
        SPI_Send(W_REGISTER|TX_ADDR);
        for (i=0;i<5;i++)
        {
        SPI_Send(ADDR_P0[i]);
        }
        RF_Select_Not;

        RF_Select;
        SPI_Send(W_REGISTER|RX_ADDR_P0);
        for (i=0;i<5;i++)
        {
        SPI_Send(ADDR_P0[i]);
        }
        RF_Select_Not;
}

void Get_TX_Address(unsigned char ADDR_P0[5])
{
        uint8_t i;
```

```
        unsigned char *ptr = ADDR_P0;
        RF_Select;
        SPI_Send(R_REGISTER|TX_ADDR);
        for (i=0;i <5; i++)
        {
        ptr[i] = SPI_Send(NOP_RF);
        }
        RF_Select_Not;
}

void Get_RX_Address(unsigned char ADDR_P1[5])
{
        uint8_t i;
        unsigned char *ptr = ADDR_P1;
        RF_Select;
        SPI_Send(R_REGISTER|RX_ADDR_P1);
        for (i=0;i <5; i++)
        {
        ptr[i] = SPI_Send(NOP_RF);
        }
        RF_Select_Not;
}


/* Configura la longitud de los bloques de datos de TX (0-32 Bytes) */
void Set_TX_Length(uint8_t P0_LENGTH)
{
        RF_Select;
        SPI_Send(W_REGISTER|RX_PW_P0); //Envia Comando de configuracion de Pipe 0.
        SPI_Send(P0_LENGTH); //Pipe 0.
        RF_Select_Not;

        PP0_Length = P0_LENGTH;
}

/* Configura la dirección de RX (5 Bytes) */
void Set_RX_Address(unsigned char ADDR_P1[5])
{
        uint8_t i;
        RF_Select;
        SPI_Send(W_REGISTER|RX_ADDR_P1);
        for (i=0;i<5;i++)
        {
        SPI_Send(ADDR_P1[i]);
        }
        RF_Select_Not;
}

/* Configura la longitud de los bloques de datos de RX (0-32 Bytes)*/
void Set_RX_Length(uint8_t P1_LENGTH)
{
        RF_Select;
        SPI_Send(W_REGISTER|RX_PW_P1);
        SPI_Send(P1_LENGTH);
        RF_Select_Not;

        PP1_Length = P1_LENGTH;
}


/*      Configura los canales de RX habilitados.
        (No modificar a menos que se desee tener más canales de RX) */
Set_Enabled_RX_Addresses(uint8_t cfg_Rx)
{
        RF_Select;
        SPI_Send(W_REGISTER|EN_RXADDR); //Envia Comando de configuracion de RX addresses.
        SPI_Send(cfg_Rx);
```

```
        RF_Select_Not;
}

/* Inicializa el nRF24L01+ con una velocidad de 2Mbps, y direcciones de 5 Bytes
        MODE    ->      Configura el modo RX o TX. True para TX, False para RX.
        ADDR_P0 ->      Dirección de TX (5-Bytes).
        ADDR_P1 ->      Dirección de RX (5-Bytes).
        RF_CHANNEL      ->      Canal de RF 2400MHZ - 2527 MHZ (introducir valor entre 0-127)
*/
void NRF24L01_Init(bool MODE, unsigned char ADDR_P0[5],
unsigned char ADDR_P1[5], uint8_t RF_CHANNEL, uint8_t P0_LENGTH,
uint8_t P1_LENGTH, uint8_t RF_SPEED)
{
        Set_RF_Channel(RF_CHANNEL);

        Set_TX_Address(ADDR_P0);

        Set_RX_Address(ADDR_P1);

        Set_Enabled_RX_Addresses(0x03);

        Set_TX_Length(P0_LENGTH);

        Set_RX_Length(P1_LENGTH);

        switch (RF_SPEED)
        {
        case 1:
        RF_SPEED = RF_SETUP_SPEED_2Mbps;
        break;

        case 2:
        RF_SPEED = RF_SETUP_SPEED_1Mbps;
        break;

        case 3:
        RF_SPEED = RF_SETUP_SPEED_250Kbps;
        break;

        default:
        RF_SPEED = RF_SETUP_SPEED_2Mbps;
        }


        Set_RF_Dataspeed(RF_SPEED);

        RF_Select;
        SPI_Send(W_REGISTER|SETUP_RETR);
        SPI_Send(0x55);
        RF_Select_Not;

        if (MODE==false)
        {
        Set_RX_Mode();

        }

        else if (MODE==true)
        {

        Set_TX_Mode();

        }


        Clean_Rx_Fifo();
        Clean_Tx_Fifo();
        _delay_ms(2);
```

```
}


/* Lee los datos recibidos vía RF (bloques de datos estáticos)
        readData[]      ->      Arreglo donde se almacenará (por referencia)    el valor
        leído por RF. Su tamaño debe ser igual o mayor al
        definido en Set_RX_Length().
*/
void Read_RF_Data(unsigned char readData[])
{
        int8_t i;
        unsigned char *ptr= readData;

        if(sizeof(readData)<PP1_Length)
        {
        return;
        }

        RF_CE_Low;
        RF_Select;
        SPI_Send(R_RX_PAYLOAD);
        for(i=0;i<PP1_Length;i++)
        {
        ptr[i]=SPI_Send(NOP_RF);
        }
        RF_Select_Not;
}



/* Lee el valor de Bytes recibidos vía RF (bloques de datos dinámicos) */
uint8_t Read_ACK_Data_Width()
{
        uint8_t width;

        RF_Select;
        SPI_Send(R_RX_PL_WID);
        width = SPI_Send(NOP_RF);
        RF_Select_Not;

        return width;
}

/* Lee los datos recibidos vía RF (bloques de datos dinámicos)

        RX_buffer[]     ->      Arreglo donde se almacenará (por referencia)    el valor
        leído por RF. Su tamaño debe ser igual o mayor al
        recibido (se recomienda que su valor sea de 32 Bytes)

        Regresa el valor de los bytes recibidos por RF.
*/
uint8_t Read_Dynamic_RF_Data(unsigned char RX_buffer[])
{

        uint8_t i,long_datos_ACK;
        unsigned char *ptr= RX_buffer;

        long_datos_ACK = Read_ACK_Data_Width();


        if (long_datos_ACK > 32)
        {
        Clean_Rx_Fifo();
        return 0;
        }
```

```
        RF_Select;
        SPI_Send(R_RX_PAYLOAD);
        for(i=0;i<long_datos_ACK;i++)
        {
        ptr[i]=SPI_Send(0x00);
        }
        RF_Select_Not;



        return long_datos_ACK;

}

/*      Asigna un bloque de datos en el paquete de acuse de recibo (ACK)
        Se debe tener habilitado DPL con la función Enable_DPL() */
void Set_ACK_Data(uint8_t ACK_Data[])
{

        if (sizeof(ACK_Data)>32)
        {
        return;
        }

        uint8_t i;

        RF_Select;
        SPI_Send(W_ACK_PAYLOAD_PP0);
        for(i=0;i<sizeof(ACK_Data);i++)
        {
        SPI_Send(ACK_Data[i]);
        }
        RF_Select_Not;

}

/*      Habilita la función Dynamic Payload Length      y ACK Payload del nRF24L01+*/
bool Enable_DPL()
{

        unsigned char temp, buffer;

        RF_Select;
        SPI_Send(R_REGISTER|DYNPD);
        temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        buffer = temp | 0x3F;

        RF_Select;
        SPI_Send(W_REGISTER|DYNPD);
        SPI_Send(buffer);
        RF_Select_Not;

        RF_Select;
        SPI_Send(R_REGISTER|DYNPD);
        temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        if (temp == 0)
        {
        buffer = 0x73;

        RF_Select;
        SPI_Send(ACTIVATE);
        SPI_Send(buffer);
        RF_Select_Not;
```

```
        RF_Select;
        SPI_Send(R_REGISTER|DYNPD);
        temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        buffer = temp | 0x3F;

        RF_Select;
        SPI_Send(W_REGISTER|DYNPD);
        SPI_Send(buffer);
        RF_Select_Not;

        }

        RF_Select;
        SPI_Send(R_REGISTER|FEATURE);
        temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        buffer = temp | 0x06;

        RF_Select;
        SPI_Send(W_REGISTER|FEATURE);
        SPI_Send(buffer);
        RF_Select_Not;

        return true;
}


/*      Deshabilita la función Dynamic Payload Length    y ACK Payload del nRF24L01+*/
bool Disable_DPL()
{

        RF_Select;
        SPI_Send(W_REGISTER|DYNPD);
        SPI_Send(0x00);
        RF_Select_Not;

        RF_Select;
        SPI_Send(W_REGISTER|FEATURE);
        SPI_Send(0x00);
        RF_Select_Not;

        return false;
}

/* Habilita las interrupciones por recepción en IRQ. */
void Enable_RX_IRQ()
{
        unsigned char Config_Temp;

        RF_Select;
        SPI_Send(R_REGISTER|CONFIG_RF);
        Config_Temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        Config_Temp &= 0x3F;

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(Config_Temp);
        RF_Select_Not;
}

/* Habilita las interrupciones por tranmsión y retransmisión en IRQ. */
void Enable_TX_IRQ()
```

```
{
        unsigned char Config_Temp;

        RF_Select;
        SPI_Send(R_REGISTER|CONFIG_RF);
        Config_Temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        Config_Temp &= 0x4F;

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(Config_Temp);
        RF_Select_Not;
}

/* Habilita todas las interrupciones en IRQ. */
void Enable_All_IRQ()
{
        unsigned char Config_Temp;

        RF_Select;
        SPI_Send(R_REGISTER|CONFIG_RF);
        Config_Temp = SPI_Send(NOP_RF);
        RF_Select_Not;

        Config_Temp &= 0x0F;

        RF_Select;
        SPI_Send(W_REGISTER|CONFIG_RF);
        SPI_Send(Config_Temp);
        RF_Select_Not;
}


/*      Transmite un dato de longitud dinámica (1-32 bytes).
        Se debe activar el Dynamic Payload Length para poder utilizar ésta función.

        writeData[]     ->      Arreglo que contiene    los datos a enviar por RF.
        Su tamaño no puede ser mayor a 32 Bytes.

        bytes   ->      Número de Bytes a envíar del arreglo writeData[].
        Si se desean enviar todos, se puede utilizar la
        función sizeof(writeData[]).

*/
void Transmit_Dynamic_RF_Data(uint8_t writeData[], uint8_t bytes)
{
        uint8_t i;

        if (sizeof(writeData)>32||bytes >32)
        {
        return;
        }

        RF_Select;
        SPI_Send(W_TX_PAYLOAD);
        for(i=0;i<bytes;i++)
        {
        SPI_Send(writeData[i]);
        }
        RF_Select_Not;

        RF_CE_High;
        _delay_us(30);
        RF_CE_Low;
}
```

```
/*      Transmite un dato de longitud fija (1-32 bytes).

        writeData[]     ->      Arreglo que contiene    los datos a enviar por RF.
        Su tamaño debe ser igual al definido en Set_TX_Length().

*/
void Transmit_RF_Data(unsigned char writeData[])
{
        uint8_t i;

        if (sizeof(writeData)>32)
        {
        return;
        }

        RF_Select;
        SPI_Send(W_TX_PAYLOAD);
        for(i=0;i<PP0_Length;i++)
        {
        SPI_Send(writeData[i]);
        }
        RF_Select_Not;

        RF_CE_High;
        _delay_us(30);
        RF_CE_Low;
}
```

APÉNDICE 4: PROGRAMA PRINCIPAL INTERFAZ USB – AVR
(AVR_INTERFAZ_USB.C)

```
/*
 *      Avr_Interfaz_USB.c
 *
 *      Autor: Rubén Flores A.
 *
 *      Proyecto: Diseño de una interfaz para sistemas de control en Java
 *      con comunicación inalámbrica utilizando un microcontrolador AVR
 *      con interfaz USB.
 *
 *      Microcontrolador: AVR ATMEGA 8
 *      Fusibles uilizados:
 *
 *      Low Byte:       0xFF
 *      High Byte:      0xC9
 *
 */

/* Frecuencia del cristal: 12 MHz */
// Nota: Los valores permitidos para V-USB pueden ser: 12 MHz, 12.8 MHz, 15 MHz, 16 MHz, 16.5 MHz,
18 MHz y 20 MHz.

#define F_CPU 12000000UL

/* Librerías genéricas utilizadas       */

#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/wdt.h>
#include <avr/sfr_defs.h>
#include <avr/pgmspace.h>
#include <avr/eeprom.h>
#include <stdbool.h>
#include <stdlib.h>
#include <math.h>

/* Librerías especiales utilizadas      */

// Librería creada, contiene definiciones y funciones utilizadas para configurar el Transceiver
NRF24L01.
// Nota: modificar libreria si se desea modificar los pines utilizados para otro tipo de AVR.
#include "NRF24L01.h"

/* Librerías utilizadas por VUSB.*/
#include "usbdrv.h"
#include "oddebug.h"
#include "usbconfig.h"
#include "usbportability.h"

/* Definiciones de constantes utilizadas. */

// constantes que identifican el comando recibido en bRequest.
#define usbReq_Send_RF_Data     1
#define usbReq_Receive_USB_Data 2
#define usbReq_Receive_RF_Data  3 //Obsoleto. Utilizar usbReq_Read_RF_Data.
#define usbReq_Send_Dynamic_RF_Data     4
#define usbReq_Read_Dynamic_RF_Data     5
#define usbReq_Read_RF_Data     8
#define usbReq_Enable_DPL       10
#define usbReq_Disable_DPL      11
#define usbReq_Set_TX_Mode      12
#define usbReq_Set_TX_Address   13
#define usbReq_Get_TX_Address   14
#define usbReq_Set_RX_Mode      15
#define usbReq_Set_RX_Address   16
```

```
#define usbReq_Get_RX_Address   17
#define usbReq_Set_TX_Length    18
#define usbReq_Set_RX_Length    19
#define usbReq_Get_TX_Length    20
#define usbReq_Get_RX_Length    21
#define usbReq_Get_DPL_State    22
#define usbReq_Get_RF_Mode      23
#define usbReq_Read_nRF24L01_Register   30
#define usbReq_Write_nRF24L01_Register  31
#define usbReq_Set_RF_Channel   50
#define usbReq_Get_RF_Channel   51
#define usbReq_Set_Data_Speed   60
#define usbReq_Get_Data_Speed   61
#define usbReq_Turn_LED_On      200
#define usbReq_Turn_LED_Off     201
#define usbReq_Blink_LED202
#define usbReq_Set_Initial_TX_Address   210
#define usbReq_Set_Initial_RX_Address   215
#define usbReq_Set_Initial_RF_Channel   220
#define usbReq_Set_Initial_RF_Speed     225
#define usbReq_Set_Initial_RF_Mode      230
#define usbReq_Set_Initial_TX_Length    235
#define usbReq_Set_Initial_RX_Length    240
#define usbReq_Set_Initial_DPL_State    245
#define usbReq_Get_Initial_Values       250

// constantes para identificación de errores.
#define Error   64
#define Ok      20
#define sensores6

#define LED_IND PC5
#define LED_ON  PORTC |= 0X20
#define LED_OFF PORTC &= 0XDF

/* Definiciones de variables globales utilizadas. */
static unsigned char USB_Buffer[32];
static unsigned char RF_Data_Buffer[32];
static unsigned char ACK_Data_Buffer[32];
static unsigned char Initial_Values_Buffer[16];
static unsigned char RF_Channel[1];
static unsigned char RF_Speed[1];
static unsigned char Read_Register[1];
static unsigned char TX_AD[5];
uint8_t TX_Length;
static unsigned char RX_AD[5];
uint8_t RX_Length;

// Utilizados para Leer Datos del USB
static unsigned char dataReceived = 0, dataLength = 0;

unsigned char Status_temp;
bool DPL_State = false;
// TX = true; RX = false
bool RF_Mode = false;
#define TX_Mode true
#define RX_Mode false

uint8_t ACK_Data_Width;
unsigned char i;
uint8_t reg_data[1];

/* Variables de inicio almacenadas en la EEPROM*/
uint8_t EEMEM   NonVolatileRFMode = 1;
uint8_t EEMEM   NonVolatileRFChannel = 60;
unsigned char   EEMEM   NonVolatileRFSpeed = 1;
unsigned char   EEMEM   NonVolatileTXAddress [5] = "COM01";
```

```
uint8_t EEMEM   NonVolatileTXDataLength = 4;
unsigned char   EEMEM   NonVolatileRXAddress [5] = "USBRF";
uint8_t EEMEM   NonVolatileRXDataLength = 32;
uint8_t EEMEM   NonVolatileDPLState = 1;



uint8_t bRequest_Send_Dynamic_RF_Data(uint8_t leng);
uint8_t bRequest_Read_Dynamic_RF_Data();
bool bRequest_Send_Data_RF();
bool bRequest_Read_RF_Data();



/*      Función usbFunctionSetup.
        Esta función es llamada automáticamente cuando existe un
        mensaje de control      por parte del host.
*/
USB_PUBLIC unsigned char usbFunctionSetup(unsigned char data[8])
{

        // Obtiene los 8 Bytes del paquete Setup
        // y los almacena en el apuntador rq.
        usbRequest_t *rq = (void *)data;



        /*      El valor del comando se encuentra en el campo 'bRequest', por lo que
        se evalúa su valor y de acuerdo a éste se realiza la función deseada.
        */

        switch(rq->bRequest)
        {
        case usbReq_Send_RF_Data:
        if (bRequest_Send_Data_RF() == true)
        {
        usbMsgPtr = "1";
        return 1;
        }

        else
        return 0;


        case usbReq_Receive_USB_Data:
        bRequest_Receive_Data_USB((unsigned char)rq->wLength.word);
        return USB_NO_MSG;      // Llama la función usbFunctionWrite.

        case usbReq_Receive_RF_Data:
        bRequest_Receive_Data_RF();
        usbMsgPtr = RF_Data_Buffer;
        return sizeof(RF_Data_Buffer);

        case usbReq_Send_Dynamic_RF_Data:
        ACK_Data_Width = bRequest_Send_Dynamic_RF_Data((uint8_t) rq->wValue.word);

        if (ACK_Data_Width== Error || ACK_Data_Width >32)
        {
        ACK_Data_Buffer[0] = 'E';
        ACK_Data_Buffer[1] = 'R';
        ACK_Data_Buffer[2] = 'R';
        ACK_Data_Buffer[3] = '0';
        ACK_Data_Buffer[4] = '0';
        usbMsgPtr = ACK_Data_Buffer;
        return 5;
        }
```

```
usbMsgPtr = ACK_Data_Buffer;
return ACK_Data_Width;


case usbReq_Set_RF_Channel:
bRequest_Set_Channel_RF((uint8_t) rq->wValue.word);
usbMsgPtr = RF_Channel;
return sizeof(RF_Channel);

case usbReq_Get_RF_Channel:
bRequest_Get_Channel_RF();
usbMsgPtr = RF_Channel;
return sizeof(RF_Channel);

case usbReq_Set_Data_Speed:
bRequest_Set_Data_Speed(rq->wValue.word);
usbMsgPtr = RF_Speed;
return sizeof(RF_Speed);

case usbReq_Get_Data_Speed:
bRequest_Get_RF_Speed();
usbMsgPtr = RF_Speed;
return sizeof(RF_Speed);

case usbReq_Enable_DPL:
bRequest_Enable_DPL();
return 0;

case usbReq_Disable_DPL:
bRequest_Disable_DPL();
return 0;

case usbReq_Set_TX_Mode:
RF_Mode = Set_TX_Mode();
return 0;

case usbReq_Set_RX_Mode:
RF_Mode = Set_RX_Mode();
return 0;

case usbReq_Set_TX_Address:
bRequest_Set_TX_Address();
return 0;

case usbReq_Set_Initial_TX_Address:
bRequest_Set_Initial_TX_Address();
return 0;

case usbReq_Set_Initial_RF_Channel:
bRequest_Set_Initial_RF_Channel(rq->wValue.word);
return 0;

case usbReq_Set_Initial_RF_Speed:
bRequest_Set_Initial_RF_Speed(rq->wValue.word);
return 0;

case usbReq_Set_Initial_RX_Length:
bRequest_Set_Initial_RX_Length(rq->wValue.word);
return 0;

case usbReq_Set_Initial_TX_Length:
bRequest_Set_Initial_TX_Length(rq->wValue.word);
return 0;

case usbReq_Set_Initial_RF_Mode:
```

```
bRequest_Set_Initial_RF_Mode(rq->wValue.word);
return 0;

case usbReq_Set_Initial_DPL_State:
bRequest_Set_Initial_DPL_State(rq->wValue.word);
return 0;

case usbReq_Get_TX_Address:
bRequest_Get_TX_Address();
usbMsgPtr = TX_AD;
return sizeof(TX_AD);

case usbReq_Set_RX_Address:
bRequest_Set_RX_Address();
return 0;

case usbReq_Set_Initial_RX_Address:
bRequest_Set_Initial_RX_Address();
return 0;

case usbReq_Get_RX_Address:
bRequest_Get_RX_Address();
usbMsgPtr = RX_AD;
return sizeof(RX_AD);

case usbReq_Set_TX_Length:
bRequest_Set_TX_Length(rq->wValue.word);
usbMsgPtr = TX_Length;
return 1;

case usbReq_Get_TX_Length:
usbMsgPtr = TX_Length;
return 1;

case usbReq_Set_RX_Length:
bRequest_Set_RX_Length(rq->wValue.word);
usbMsgPtr = RX_Length;
return 1;

case usbReq_Get_RX_Length:
usbMsgPtr = RX_Length;
return 1;

case usbReq_Get_DPL_State:
if (DPL_State == false)
{
return 0;
}
else
usbMsgPtr = "1";
return 1;

case usbReq_Get_RF_Mode:
if (RF_Mode == false)
{
return 0;
}
else
usbMsgPtr = "1";
return 1;

case usbReq_Write_nRF24L01_Register:
bRequest_Write_Register(rq->wValue.bytes[0], rq->wIndex.bytes[0]);
return 0;

case usbReq_Read_nRF24L01_Register:
bRequest_Read_Register(rq->wValue.word, &reg_data);
```

```
        usbMsgPtr = reg_data;
        return sizeof(reg_data);

        case usbReq_Turn_LED_On:
        LED_ON;
        return 0;

        case usbReq_Turn_LED_Off:
        LED_OFF;
        return 0;

        case usbReq_Blink_LED:
        Blink_LED();
        return 0;

        case usbReq_Get_Initial_Values:
        bRequest_Get_Initial_Values(&Initial_Values_Buffer);
        usbMsgPtr = Initial_Values_Buffer;
        return sizeof(Initial_Values_Buffer);

        case usbReq_Read_RF_Data:
        if (bRequest_Read_RF_Data() == false)
        {
        return 0;
        }
        usbMsgPtr = RF_Data_Buffer;
        return RX_Length;

        case usbReq_Read_Dynamic_RF_Data:
        ACK_Data_Width = bRequest_Read_Dynamic_RF_Data();
        if (rq->wValue.word > 0)
        {
        Set_ACK_Data(USB_Buffer);
        }

        if (ACK_Data_Width == 0)
        {
        return 0;
        }
        usbMsgPtr = RF_Data_Buffer;
        return ACK_Data_Width;


        default:
        return 0;
        }

        return 0;
}


/*      Función usbFunctionWrite
        Esta función es llamada cuando se reciben datos del host (PC) hacia el dispositivo.
*/

USB_PUBLIC unsigned char usbFunctionWrite(unsigned char *data, unsigned char len) {
        unsigned char i;

        for(i = 0; dataReceived < dataLength && i < len; i++, dataReceived++)
        USB_Buffer[dataReceived] = data[i];
        // regresa 1 sí recibimos todo, 0 sí no.
        return (dataReceived == dataLength);


}
```

```
/* Función main*/

int main(void)
{
        // Variable local, utilizada para la re-enumeración.
        unsigned char i;

        /* Configuración de los puertos */

        DDRB |= (1<<CE) | (1<<CSN) | (1<<MOSI) | (1<<SCK);
        DDRB &= ~(1<<MISO);
        DDRD &= ~(1<<IRQ);
        DDRC |= (1<<LED_IND);


        // Se asegura que no esté activado el pin Slave del nRF24L01.
        RF_Select_Not;

        // Habilita el Watchdog Timer con 1 segundo.
        wdt_enable(WDTO_1S);

        usbInit();

        // Fuerza la re-enumeración.
        usbDeviceDisconnect();
        for(i = 0; i<250; i++)
        {
        // Espera 500ms.
        wdt_reset();
        _delay_ms(2);
        }
        usbDeviceConnect();

        // Inicializa la interfaz SPI

        SPI_Init();

        /* Obtiene los valores iniciales de la EEPROM*/

        Get_Initial_Values();

        /* Inicializa el módulo NRF24L01+
        *       En caso de no tener algun valor definido,
        *       los valores por Default son los siguientes:
        *       Modo: Transmisor
        *       Dirección de TX: COM01
        *       Dirección de RX: USBRF
        *       Canal de RF: 60 (2460 MHz)
        */

        NRF24L01_Init(RF_Mode, TX_AD, RX_AD, RF_Channel[0], TX_Length, RX_Length, RF_Speed[0]);

        // Habilita el modo "bloques de datos dinámicos" (DPL)
        if (DPL_State == true)
        {
        DPL_State = Enable_DPL();
        }


        // Parpadea LED como indicador de que todo está configurado

        Blink_LED();

        // Habilitación general de interrupciones

        sei();
```

```
        // Bucle infinito, a la espera de peticiones USB.

        while (1)
        {
        wdt_reset();
        usbPoll();
        }
}

void Get_Initial_Values()
{
        uint8_t RFMode;
        RFMode = eeprom_read_byte (& NonVolatileRFMode);
        if (RFMode == 0xFF)
        {
        RF_Channel[0] = 60;
        RF_Mode = true;
        TX_Length =4;
        RX_Length =32;
        DPL_State = true;
        RF_Speed[0] = 1;

        TX_AD[0] = "C";
        TX_AD[1] = "O";
        TX_AD[2] = "M";
        TX_AD[3] = "0";
        TX_AD[4] = "1";

        RX_AD[0] = "U";
        RX_AD[1] = "S";
        RX_AD[2] = "B";
        RX_AD[3] = "R";
        RX_AD[4] = "F";
        return;
        }
        else if (RFMode == 0x00)
        {
        RF_Mode = false;
        }
        else if (RFMode == 0x01)
        {
        RF_Mode = true;
        }
        RF_Channel[0] = eeprom_read_byte(& NonVolatileRFChannel);
        RF_Speed[0] = eeprom_read_byte(& NonVolatileRFSpeed);
        TX_Length = eeprom_read_byte(& NonVolatileTXDataLength);
        RX_Length = eeprom_read_byte(& NonVolatileRXDataLength);
        eeprom_read_block (( void *) TX_AD , ( const void *) NonVolatileTXAddress , sizeof(TX_AD));
        eeprom_read_block (( void *) RX_AD , ( const void *) NonVolatileRXAddress , sizeof(RX_AD));
        uint8_t DPL;
        DPL = eeprom_read_byte (& NonVolatileDPLState);
        if (DPL == 0)
        {
        DPL_State = false;
        }
        else
        DPL_State = true;
}

bool bRequest_Send_Data_RF()
{
        if (RF_Mode==RX_Mode)
        {
        RF_Mode = Set_TX_Mode();
        _delay_ms(2);
        }
```

```
        Transmit_RF_Data(USB_Buffer);
        while(bit_is_set(PIND,IRQ_PIN));
        Status_temp=Get_Status();
        if(bit_is_set(Status_temp,MAX_RT))
        {
        Clean_Status();
        return false;
        }
        Clean_Status();
        return true;
}

void bRequest_Receive_Data_USB(unsigned char leng)
{
        dataLength      = leng;
        dataReceived = 0;
        if(dataLength   > sizeof(USB_Buffer))
        dataLength      = sizeof(USB_Buffer);
}

void bRequest_Receive_Data_RF()
{
        uint16_t j;
        Transmit_RF_Data(USB_Buffer);
        while(bit_is_set(PIND,IRQ_PIN));
        Status_temp=Get_Status();
        Clean_Status();
        RF_Data_Buffer[16]=Error;


        if(bit_is_set(Status_temp,MAX_RT))
        {
        RF_Data_Buffer[16]=Error;
        return 0;
        }
        Set_RX_Mode();

        _delay_ms(2);

        for (j=0;j<=60000;j++)
        {
        if((bit_is_clear(PIND,IRQ_PIN)))
        {
        RF_CE_Low;
        RF_Data_Buffer[16]=Ok;
        Read_RF_Data(&RF_Data_Buffer);
        Clean_Status;
        j=61000;
        }
        }
        Set_TX_Mode();
        _delay_ms(2);
}

uint8_t bRequest_Send_Dynamic_RF_Data(uint8_t leng)
{

        if ((DPL_State == true) && (RF_Mode == true))
        {
        uint16_t j;
        ACK_Data_Width = 0;
        Transmit_Dynamic_RF_Data(USB_Buffer,leng);
        while(bit_is_set(PIND,IRQ_PIN));
        Status_temp=Get_Status();
```

```
        if((bit_is_set(Status_temp,MAX_RT)))
        {
        Clean_Tx_Fifo();
        Clean_Status();
        return Error;
        }

        if((bit_is_set(Status_temp,RX_DR)))
        {
        ACK_Data_Width = Read_Dynamic_RF_Data(&ACK_Data_Buffer);
        Clean_Rx_Fifo();
        }

        Clean_Status();
        return ACK_Data_Width;
        }

        return 0;
}

void bRequest_Set_Channel_RF(uint8_t Can_RF)
{
        if (Can_RF >= 0 && Can_RF <=127)
        {
        Set_RF_Channel(Can_RF);
        }

        RF_Channel[0] = Get_RF_Channel();
}

void bRequest_Get_Channel_RF()
{
        RF_Channel[0] = Get_RF_Channel();
}

void bRequest_Set_Data_Speed(uint8_t Velocidad_RF)
{
        if (Velocidad_RF==1)
        {
        RF_Speed[0]=RF_SETUP_SPEED_2Mbps;
        }
        else if (Velocidad_RF==2)
        {
        RF_Speed[0]=RF_SETUP_SPEED_1Mbps;
        }
        else if (Velocidad_RF==3)
        {
        RF_Speed[0]=RF_SETUP_SPEED_250Kbps;
        }
        else
        {
        return;
        }

        Set_RF_Dataspeed(RF_Speed[0]);
}

void bRequest_Get_RF_Speed()
{
        RF_Speed[0] = Get_RF_Dataspeed();
}

void Blink_LED()
{
        PORTC |= 0x20;
        _delay_ms(50);
        PORTC &= 0xDF;
```

```
}

void bRequest_Enable_DPL()
{
        DPL_State = Enable_DPL();
}

void bRequest_Disable_DPL()
{
        DPL_State = Disable_DPL();
}

void bRequest_Set_TX_Address()
{
for(i=0;i<5;i++)
{
        TX_AD[i] = USB_Buffer[i];
}
Set_TX_Address(TX_AD);
}

void bRequest_Get_TX_Address()
{
        Get_TX_Address(&TX_AD);
}

void bRequest_Set_RX_Address()
{
        for(i=0;i<5;i++)
        {
        RX_AD[i] = USB_Buffer[i];
        }
        Set_RX_Address(RX_AD);
}

void bRequest_Get_RX_Address()
{
        Get_RX_Address(&RX_AD);
}

void bRequest_Write_Register(uint8_t reg, uint8_t register_data)
{
        RF_Select;
        SPI_Send(W_REGISTER | reg);
        SPI_Send(register_data);
        RF_Select_Not;
}

void bRequest_Read_Register(uint8_t reg, uint8_t register_data[])
{
        uint8_t *rgdata = register_data;
        RF_Select;
        SPI_Send(R_REGISTER|reg);
        rgdata[0] = SPI_Send(NOP_RF);
        RF_Select_Not;
}

void bRequest_Set_Initial_TX_Address()
{
        unsigned char temp_TX_Add[5];
        for(i=0;i<5;i++)
        {
        temp_TX_Add[i] = USB_Buffer[i];
        }
        eeprom_update_block (( const void *) temp_TX_Add , ( void *) NonVolatileTXAddress, 5);
}
```

```
void bRequest_Set_Initial_RX_Address()
{
        unsigned char temp_RX_Add[5];
        for(i=0;i<5;i++)
        {
        temp_RX_Add[i] = USB_Buffer[i];
        }
        eeprom_update_block (( const void *) temp_RX_Add , ( void *) NonVolatileRXAddress, 5);
}

void bRequest_Set_Initial_RF_Channel(uint8_t Channel)
{
        eeprom_update_byte( &NonVolatileRFChannel, Channel);
}

void bRequest_Set_Initial_RF_Speed(unsigned char Speed)
{
        eeprom_update_byte( &NonVolatileRFSpeed, Speed);
}

void bRequest_Set_Initial_RX_Length(uint8_t AD_P1_Length)
{
        eeprom_update_byte( &NonVolatileRXDataLength, AD_P1_Length);
}

void bRequest_Set_Initial_TX_Length(uint8_t AD_P1_Length)
{
        eeprom_update_byte( &NonVolatileTXDataLength, AD_P1_Length);
}

void bRequest_Set_Initial_RF_Mode(uint8_t Mode)
{
        eeprom_update_byte( &NonVolatileRFMode, Mode);
}

void bRequest_Set_Initial_DPL_State(uint8_t DPL)
{
        eeprom_update_byte( &NonVolatileDPLState, DPL);
}

void bRequest_Set_TX_Length(uint8_t AD_P0)
{
        Set_TX_Length(AD_P0);
        TX_Length = AD_P0;
}

void bRequest_Set_RX_Length(uint8_t AD_P1)
{
        Set_TX_Length(AD_P1);
        RX_Length = AD_P1;
}

void bRequest_Get_Initial_Values(unsigned char Values[16])
{
        unsigned char *val = Values;
        unsigned char Temp_AD[5];

        val[0] = eeprom_read_byte(&NonVolatileRFMode);
        val[1] = eeprom_read_byte(&NonVolatileRFChannel);
        val[2] = eeprom_read_byte(&NonVolatileRFSpeed);
        eeprom_read_block((void*)Temp_AD, (const void*)NonVolatileTXAddress,5);

        for (i=0;i<5;i++)
        {
        val[i+3]= Temp_AD[i];
        }
```

```
        val[8] = eeprom_read_byte(&NonVolatileTXDataLength);

        eeprom_read_block((void*)Temp_AD, (const void*)NonVolatileRXAddress,5);
        for (i=0;i<5;i++)
        {
        val[i+9]= Temp_AD[i];
        }

        val[14] = eeprom_read_byte(&NonVolatileRXDataLength);
        val[15] = eeprom_read_byte(&NonVolatileDPLState);

}

bool bRequest_Read_RF_Data()
{
        if (RF_Mode == TX_Mode)
        {
        RF_Mode = Set_RX_Mode();
        _delay_ms(2);
        }

        Status_temp=Get_Status();

        if(bit_is_set(Status_temp,RX_DR))
        {
        RF_CE_Low;
        Read_RF_Data(&RF_Data_Buffer);
        Clean_Status();
        Clean_Rx_Fifo();
        return true;
        }

        return false;
}

uint8_t bRequest_Read_Dynamic_RF_Data()
{
        if (RF_Mode == TX_Mode)
        {
        RF_Mode = Set_RX_Mode();
        _delay_ms(2);
        }

        Status_temp=Get_Status();

        if(bit_is_set(Status_temp,RX_DR))
        {
        RF_CE_Low;
        uint8_t qty;
        qty = Read_Dynamic_RF_Data(&RF_Data_Buffer);
        Clean_Status();
        Clean_Rx_Fifo();
        RF_CE_High;
        return qty;
        }

        return 0;
}
```

APÉNDICE 5: LIBRERÍA DE CONFIGURACIÓN V-USB
(USBCONFIG.H)

```
 * Name: usbconfig.h
 * Project: V-USB, virtual USB port for Atmel's(r) AVR(r) microcontrollers
 * Author: Christian Starkjohann
 * Creation Date: 2005-04-01
 * Tabsize: 4
 * Copyright: (c) 2005 by OBJECTIVE DEVELOPMENT Software GmbH
 * License: GNU GPL v2 (see License.txt), GNU GPL v3 or proprietary (CommercialLicense.txt)
 * This Revision: $Id$
 */

#ifndef __usbconfig_h_included__
#define __usbconfig_h_included__

/*
General Description:
This file is an example configuration (with inline documentation) for the USB
driver. It configures V-USB for USB D+ connected to Port D bit 2 (which is
also hardware interrupt 0 on many devices) and USB D- to Port D bit 4. You may
wire the lines to any other port, as long as D+ is also wired to INT0 (or any
other hardware interrupt, as long as it is the highest level interrupt, see
section at the end of this file).
+ To create your own usbconfig.h file, copy this file to your project's
+ firmware source directory) and rename it to "usbconfig.h".
+ Then edit it accordingly.
*/

/* ---------------------------- Hardware Config ---------------------------- */

#define USB_CFG_IOPORTNAME      D
/* This is the port where the USB bus is connected. When you configure it to
 * "B", the registers PORTB, PINB and DDRB will be used.
 */
#define USB_CFG_DMINUS_BIT      0
/* This is the bit number in USB_CFG_IOPORT where the USB D- line is connected.
 * This may be any bit in the port.
 */
#define USB_CFG_DPLUS_BIT       2
/* This is the bit number in USB_CFG_IOPORT where the USB D+ line is connected.
 * This may be any bit in the port. Please note that D+ must also be connected
 * to interrupt pin INT0! [You can also use other interrupts, see section
 * "Optional MCU Description" below, or you can connect D- to the interrupt, as
 * it is required if you use the USB_COUNT_SOF feature. If you use D- for the
 * interrupt, the USB interrupt will also be triggered at Start-Of-Frame
 * markers every millisecond.]
 */
#define USB_CFG_CLOCK_KHZ       12000
/* Clock rate of the AVR in kHz. Legal values are 12000, 12800, 15000, 16000,
 * 16500, 18000 and 20000. The 12.8 MHz and 16.5 MHz versions of the code
 * require no crystal, they tolerate +/- 1% deviation from the nominal
 * frequency. All other rates require a precision of 2000 ppm and thus a
 * crystal!
 * Since F_CPU should be defined to your actual clock rate anyway, you should
 * not need to modify this setting.
 */
#define USB_CFG_CHECK_CRC       0
/* Define this to 1 if you want that the driver checks integrity of incoming
 * data packets (CRC checks). CRC checks cost quite a bit of code size and are
 * currently only available for 18 MHz crystal clock. You must choose
 * USB_CFG_CLOCK_KHZ = 18000 if you enable this option.
 */

/* ----------------------- Optional Hardware Config ------------------------ */

/* #define USB_CFG_PULLUP_IOPORTNAME   D */
```

```
/* If you connect the 1.5k pullup resistor from D- to a port pin instead of
 * V+, you can connect and disconnect the device from firmware by calling
 * the macros usbDeviceConnect() and usbDeviceDisconnect() (see usbdrv.h).
 * This constant defines the port on which the pullup resistor is connected.
 */
/* #define USB_CFG_PULLUP_BIT          4 */
/* This constant defines the bit number in USB_CFG_PULLUP_IOPORT (defined
 * above) where the 1.5k pullup resistor is connected. See description
 * above for details.
 */

/* --------------------------- Functional Range ---------------------------- */

#define USB_CFG_HAVE_INTRIN_ENDPOINT    1
/* Define this to 1 if you want to compile a version with two endpoints: The
 * default control endpoint 0 and an interrupt-in endpoint (any other endpoint
 * number).
 */
#define USB_CFG_HAVE_INTRIN_ENDPOINT3   0
/* Define this to 1 if you want to compile a version with three endpoints: The
 * default control endpoint 0, an interrupt-in endpoint 3 (or the number
 * configured below) and a catch-all default interrupt-in endpoint as above.
 * You must also define USB_CFG_HAVE_INTRIN_ENDPOINT to 1 for this feature.
 */
#define USB_CFG_EP3_NUMBER              3
/* If the so-called endpoint 3 is used, it can now be configured to any other
 * endpoint number (except 0) with this macro. Default if undefined is 3.
 */
/* #define USB_INITIAL_DATATOKEN           USBPID_DATA1 */
/* The above macro defines the startup condition for data toggling on the
 * interrupt/bulk endpoints 1 and 3. Defaults to USBPID_DATA1.
 * Since the token is toggled BEFORE sending any data, the first packet is
 * sent with the oposite value of this configuration!
 */
#define USB_CFG_IMPLEMENT_HALT          0
/* Define this to 1 if you also want to implement the ENDPOINT_HALT feature
 * for endpoint 1 (interrupt endpoint). Although you may not need this feature,
 * it is required by the standard. We have made it a config option because it
 * bloats the code considerably.
 */
#define USB_CFG_SUPPRESS_INTR_CODE      0
/* Define this to 1 if you want to declare interrupt-in endpoints, but don't
 * want to send any data over them. If this macro is defined to 1, functions
 * usbSetInterrupt() and usbSetInterrupt3() are omitted. This is useful if
 * you need the interrupt-in endpoints in order to comply to an interface
 * (e.g. HID), but never want to send any data. This option saves a couple
 * of bytes in flash memory and the transmit buffers in RAM.
 */
#define USB_CFG_INTR_POLL_INTERVAL      10
/* If you compile a version with endpoint 1 (interrupt-in), this is the poll
 * interval. The value is in milliseconds and must not be less than 10 ms for
 * low speed devices.
 */
#define USB_CFG_IS_SELF_POWERED         0
/* Define this to 1 if the device has its own power supply. Set it to 0 if the
 * device is powered from the USB bus.
 */
#define USB_CFG_MAX_BUS_POWER           120
/* Set this variable to the maximum USB bus power consumption of your device.
 * The value is in milliamperes. [It will be divided by two since USB
 * communicates power requirements in units of 2 mA.]
 */
#define USB_CFG_IMPLEMENT_FN_WRITE      1
/* Set this to 1 if you want usbFunctionWrite() to be called for control-out
 * transfers. Set it to 0 if you don't need it and want to save a couple of
 * bytes.
 */
```

```
#define USB_CFG_IMPLEMENT_FN_READ       0
/* Set this to 1 if you need to send control replies which are generated
 * "on the fly" when usbFunctionRead() is called. If you only want to send
 * data from a static buffer, set it to 0 and return the data from
 * usbFunctionSetup(). This saves a couple of bytes.
 */
#define USB_CFG_IMPLEMENT_FN_WRITEOUT   0
/* Define this to 1 if you want to use interrupt-out (or bulk out) endpoints.
 * You must implement the function usbFunctionWriteOut() which receives all
 * interrupt/bulk data sent to any endpoint other than 0. The endpoint number
 * can be found in 'usbRxToken'.
 */
#define USB_CFG_HAVE_FLOWCONTROL        0
/* Define this to 1 if you want flowcontrol over USB data. See the definition
 * of the macros usbDisableAllRequests() and usbEnableAllRequests() in
 * usbdrv.h.
 */
#define USB_CFG_DRIVER_FLASH_PAGE       0
/* If the device has more than 64 kBytes of flash, define this to the 64 k page
 * where the driver's constants (descriptors) are located. Or in other words:
 * Define this to 1 for boot loaders on the ATMega128.
 */
#define USB_CFG_LONG_TRANSFERS          0
/* Define this to 1 if you want to send/receive blocks of more than 254 bytes
 * in a single control-in or control-out transfer. Note that the capability
 * for long transfers increases the driver size.
 */
/* #define USB_RX_USER_HOOK(data, len)     if(usbRxToken == (uchar)USBPID_SETUP) blinkLED(); */
/* This macro is a hook if you want to do unconventional things. If it is
 * defined, it's inserted at the beginning of received message processing.
 * If you eat the received message and don't want default processing to
 * proceed, do a return after doing your things. One possible application
 * (besides debugging) is to flash a status LED on each packet.
 */
/* #define USB_RESET_HOOK(resetStarts)     if(!resetStarts){hadUsbReset();} */
/* This macro is a hook if you need to know when an USB RESET occurs. It has
 * one parameter which distinguishes between the start of RESET state and its
 * end.
 */
/* #define USB_SET_ADDRESS_HOOK()              hadAddressAssigned(); */
/* This macro (if defined) is executed when a USB SET_ADDRESS request was
 * received.
 */
#define USB_COUNT_SOF                   0
/* define this macro to 1 if you need the global variable "usbSofCount" which
 * counts SOF packets. This feature requires that the hardware interrupt is
 * connected to D- instead of D+.
 */
/* #ifdef __ASSEMBLER__
 * macro myAssemblerMacro
 *     in      YL, TCNT0
 *     sts     timer0Snapshot, YL
 *     endm
 * #endif
 * #define USB_SOF_HOOK                    myAssemblerMacro
 * This macro (if defined) is executed in the assembler module when a
 * Start Of Frame condition is detected. It is recommended to define it to
 * the name of an assembler macro which is defined here as well so that more
 * than one assembler instruction can be used. The macro may use the register
 * YL and modify SREG. If it lasts longer than a couple of cycles, USB messages
 * immediately after an SOF pulse may be lost and must be retried by the host.
 * What can you do with this hook? Since the SOF signal occurs exactly every
 * 1 ms (unless the host is in sleep mode), you can use it to tune OSCCAL in
 * designs running on the internal RC oscillator.
 * Please note that Start Of Frame detection works only if D- is wired to the
 * interrupt, not D+. THIS IS DIFFERENT THAN MOST EXAMPLES!
 */
```

```
#define USB_CFG_CHECK_DATA_TOGGLING     0
/* define this macro to 1 if you want to filter out duplicate data packets
 * sent by the host. Duplicates occur only as a consequence of communication
 * errors, when the host does not receive an ACK. Please note that you need to
 * implement the filtering yourself in usbFunctionWriteOut() and
 * usbFunctionWrite(). Use the global usbCurrentDataToken and a static variable
 * for each control- and out-endpoint to check for duplicate packets.
 */
#define USB_CFG_HAVE_MEASURE_FRAME_LENGTH   0
/* define this macro to 1 if you want the function usbMeasureFrameLength()
 * compiled in. This function can be used to calibrate the AVR's RC oscillator.
 */
#define USB_USE_FAST_CRC                0
/* The assembler module has two implementations for the CRC algorithm. One is
 * faster, the other is smaller. This CRC routine is only used for transmitted
 * messages where timing is not critical. The faster routine needs 31 cycles
 * per byte while the smaller one needs 61 to 69 cycles. The faster routine
 * may be worth the 32 bytes bigger code size if you transmit lots of data and
 * run the AVR close to its limit.
 */

/* -------------------------- Device Description --------------------------- */

#define  USB_CFG_VENDOR_ID       0xc0, 0x16 /* = 0x16c0 = 5824 = voti.nl */
/* USB vendor ID for the device, low byte first. If you have registered your
 * own Vendor ID, define it here. Otherwise you may use one of obdev's free
 * shared VID/PID pairs. Be sure to read USB-IDs-for-free.txt for rules!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define  USB_CFG_DEVICE_ID       0xdc, 0x05 /* = 0x05dc = 1500 */
/* This is the ID of the product, low byte first. It is interpreted in the
 * scope of the vendor ID. If you have registered your own VID with usb.org
 * or if you have licensed a PID from somebody else, define it here. Otherwise
 * you may use one of obdev's free shared VID/PID pairs. See the file
 * USB-IDs-for-free.txt for details!
 * *** IMPORTANT NOTE ***
 * This template uses obdev's shared VID/PID pair for Vendor Class devices
 * with libusb: 0x16c0/0x5dc.  Use this VID/PID pair ONLY if you understand
 * the implications!
 */
#define USB_CFG_DEVICE_VERSION  0x00, 0x01
/* Version number of the device: Minor number first, then major number.
 */
#define USB_CFG_VENDOR_NAME     'r', 'u', 'b', 'e', 'n', 'f', 'l', 'o', 'r', 'e', 's', 'a', 'l', 't',
'a', 'm', 'i', 'r', 'a', 'n', 'o', '@', 'g', 'm', 'a', 'i', 'l', '.', 'c', 'o', 'm'
#define USB_CFG_VENDOR_NAME_LEN 31
/* These two values define the vendor name returned by the USB device. The name
 * must be given as a list of characters under single quotes. The characters
 * are interpreted as Unicode (UTF-16) entities.
 * If you don't want a vendor name string, undefine these macros.
 * ALWAYS define a vendor name containing your Internet domain name if you use
 * obdev's free shared VID/PID pair. See the file USB-IDs-for-free.txt for
 * details.
 */
#define USB_CFG_DEVICE_NAME     'C', 'o', 'm', '.', ' ', 'U', 'S', 'B', '-', 'A','V','R'
#define USB_CFG_DEVICE_NAME_LEN 12
/* Same as above for the device name. If you don't want a device name, undefine
 * the macros. See the file USB-IDs-for-free.txt before you assign a name if
 * you use a shared VID/PID.
 */
/*#define USB_CFG_SERIAL_NUMBER   'N', 'o', 'n', 'e' */
/*#define USB_CFG_SERIAL_NUMBER_LEN   0 */
/* Same as above for the serial number. If you don't want a serial number,
 * undefine the macros.
```

```
 * It may be useful to provide the serial number through other means than at
 * compile time. See the section about descriptor properties below for how
 * to fine tune control over USB descriptors such as the string descriptor
 * for the serial number.
 */
#define USB_CFG_DEVICE_CLASS        0xff    /* set to 0 if deferred to interface */
#define USB_CFG_DEVICE_SUBCLASS     0
/* See USB specification if you want to conform to an existing device class.
 * Class 0xff is "vendor specific".
 */
#define USB_CFG_INTERFACE_CLASS     0   /* define class here if not at device level */
#define USB_CFG_INTERFACE_SUBCLASS  0
#define USB_CFG_INTERFACE_PROTOCOL  0
/* See USB specification if you want to conform to an existing device class or
 * protocol. The following classes must be set at interface level:
 * HID class is 3, no subclass and protocol required (but may be useful!)
 * CDC class is 2, use subclass 2 and protocol 1 for ACM
 */
/* #define USB_CFG_HID_REPORT_DESCRIPTOR_LENGTH    42 */
/* Define this to the length of the HID report descriptor, if you implement
 * an HID device. Otherwise don't define it or define it to 0.
 * If you use this define, you must add a PROGMEM character array named
 * "usbHidReportDescriptor" to your code which contains the report descriptor.
 * Don't forget to keep the array and this define in sync!
 */

/* #define USB_PUBLIC static */
/* Use the define above if you #include usbdrv.c instead of linking against it.
 * This technique saves a couple of bytes in flash memory.
 */

/* ------------------- Fine Control over USB Descriptors ------------------- */
/* If you don't want to use the driver's default USB descriptors, you can
 * provide our own. These can be provided as (1) fixed length static data in
 * flash memory, (2) fixed length static data in RAM or (3) dynamically at
 * runtime in the function usbFunctionDescriptor(). See usbdrv.h for more
 * information about this function.
 * Descriptor handling is configured through the descriptor's properties. If
 * no properties are defined or if they are 0, the default descriptor is used.
 * Possible properties are:
 *   + USB_PROP_IS_DYNAMIC: The data for the descriptor should be fetched
 *     at runtime via usbFunctionDescriptor(). If the usbMsgPtr mechanism is
 *     used, the data is in FLASH by default. Add property USB_PROP_IS_RAM if
 *     you want RAM pointers.
 *   + USB_PROP_IS_RAM: The data returned by usbFunctionDescriptor() or found
 *     in static memory is in RAM, not in flash memory.
 *   + USB_PROP_LENGTH(len): If the data is in static memory (RAM or flash),
 *     the driver must know the descriptor's length. The descriptor itself is
 *     found at the address of a well known identifier (see below).
 * List of static descriptor names (must be declared PROGMEM if in flash):
 *   char usbDescriptorDevice[];
 *   char usbDescriptorConfiguration[];
 *   char usbDescriptorHidReport[];
 *   char usbDescriptorString0[];
 *   int usbDescriptorStringVendor[];
 *   int usbDescriptorStringDevice[];
 *   int usbDescriptorStringSerialNumber[];
 * Other descriptors can't be provided statically, they must be provided
 * dynamically at runtime.
 *
 * Descriptor properties are or-ed or added together, e.g.:
 * #define USB_CFG_DESCR_PROPS_DEVICE   (USB_PROP_IS_RAM | USB_PROP_LENGTH(18))
 *
 * The following descriptors are defined:
 *   USB_CFG_DESCR_PROPS_DEVICE
 *   USB_CFG_DESCR_PROPS_CONFIGURATION
 *   USB_CFG_DESCR_PROPS_STRINGS
```

```
 *   USB_CFG_DESCR_PROPS_STRING_0
 *   USB_CFG_DESCR_PROPS_STRING_VENDOR
 *   USB_CFG_DESCR_PROPS_STRING_PRODUCT
 *   USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER
 *   USB_CFG_DESCR_PROPS_HID
 *   USB_CFG_DESCR_PROPS_HID_REPORT
 *   USB_CFG_DESCR_PROPS_UNKNOWN (for all descriptors not handled by the driver)
 *
 * Note about string descriptors: String descriptors are not just strings, they
 * are Unicode strings prefixed with a 2 byte header. Example:
 * int  serialNumberDescriptor[] = {
 *     USB_STRING_DESCRIPTOR_HEADER(6),
 *     'S', 'e', 'r', 'i', 'a', 'l'
 * };
 */

#define USB_CFG_DESCR_PROPS_DEVICE                  0
#define USB_CFG_DESCR_PROPS_CONFIGURATION           0
#define USB_CFG_DESCR_PROPS_STRINGS                 0
#define USB_CFG_DESCR_PROPS_STRING_0                0
#define USB_CFG_DESCR_PROPS_STRING_VENDOR           0
#define USB_CFG_DESCR_PROPS_STRING_PRODUCT          0
#define USB_CFG_DESCR_PROPS_STRING_SERIAL_NUMBER    0
#define USB_CFG_DESCR_PROPS_HID                     0
#define USB_CFG_DESCR_PROPS_HID_REPORT              0
#define USB_CFG_DESCR_PROPS_UNKNOWN                 0

/* ------------------------ Optional MCU Description ------------------------ */

/* The following configurations have working defaults in usbdrv.h. You
 * usually don't need to set them explicitly. Only if you want to run
 * the driver on a device which is not yet supported or with a compiler
 * which is not fully supported (such as IAR C) or if you use a differnt
 * interrupt than INT0, you may have to define some of these.
 */
/* #define USB_INTR_CFG            MCUCR */
/* #define USB_INTR_CFG_SET        ((1 << ISC00) | (1 << ISC01)) */
/* #define USB_INTR_CFG_CLR        0 */
/* #define USB_INTR_ENABLE         GIMSK */
/* #define USB_INTR_ENABLE_BIT     INT0 */
/* #define USB_INTR_PENDING        GIFR */
/* #define USB_INTR_PENDING_BIT    INTF0 */
/* #define USB_INTR_VECTOR         INT0_vect */

#endif /* __usbconfig_h_included__ */
```

APÉNDICE 6: FORMATO DE DATOS "SETUP" EN MENSAJES DE CONTROL USB[34]

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

[34] Extracto de "*Universal Serial Bus Specification Revision 2.0, Tabla 9-2 "Format of Setup Data*"

Javausb

Class UsbRFDevice

java.lang.Object

Javausb.UsbRFDevice

public class **UsbRFDevice**

extends java.lang.Object

Ésta clase permite la comunicación inalámbrica entre la computadora y otro dispositivo de control que cuente con el transceptor nRF24L01+. Se requiere de un dispositivo AVR con V-USB, LibUSb y un dispositivo NRF24L01+. Vendor Id: 0x20A0 Product Id: 0x4204.

Resumen de Campos

Campos	
Modificador y Tipo	**Campo y Descripción**
static int	RF_Speed_1Mbps Velocidad de RF de 1 Mbps.
static int	RF_Speed_1Mbps Velocidad de RF de 1 Mbps.
static int	RF_Speed_2Mbps Velocidad de RF de 2 Mbps.
static javax.swing.JPanel	panelerror
int	RX_Length Numero de bytes recibidos RF.
int	TX_Length Numero de bytes enviados RF.

Resumen de Constructores

Constructores
Constructor
UsbRFDevice()

Resumen de Métodos

Métodos	
Modificador y Tipo	**Método y Descripción**
void	disableDPL() Deshabilita la función de datos de longitud dinámica (DPL).
void	enableDPL() Habilita la función de datos de longitud dinámica (DPL).
boolean	getDPLState() Regresa el estado de la función de datos de longitud variable (DPL).
java.lang.String	getInitialValues() Lee los valores de configuración iniciales almacenados en la EEPROM del dispositivo.
int	getRFChannel() Lee el canal de frecuencia del transceptor.
boolean	getRFMode() Regresa el modo de RF en el que se encuentra actualmente el dispositivo.
int	getRFSpeed() Método que permite obtener la velocidad de transmisión del transceptor.
java.lang.String	getRXAddress() Obtiene el valor de dirección asignado para recibir (RX).
int	getRXLength() Regresa la longitud de datos fijos asignados para recibir (RX).
java.lang.String	getTXAddress() Obtiene el valor de dirección asignado para transmitir (TX).

Métodos	
Modificador y Tipo	**Método y Descripción**
int	getTXLength() Regresa la longitud de datos fijos asignados para transmitir (TX).
boolean	readDynamicRFData(int[] Received_Data) Lee los datos de longitud dinámica (DPL) recibidos por RF.
boolean	readDynamicRFData(int[] Received_Data, java.lang.String ACK_Data)Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.
boolean	readDynamicRFData(java.lang.String[] Received_Data) Lee los datos de longitud dinámica (DPL) recibidos por RF.
boolean	readDynamicRFData(java.lang.String[] Received_Data, java.lang.String ACK_Data) Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.
int	readnrf24l01register(int register) Lee el registro indicado del nRF24L01.
boolean	sendCommandDataRF(int Comando, int Dato) Envía un dato y un comando de 2 Bytes vía RF.
boolean	sendDataRF(int[] dato_RF) Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF.
boolean	sendDataRF(java.lang.String dato_RF) Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF.
int	sendDynamicDataRF(int[] dato_RF, char[] ACK_Data, boolean[] confirm) Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF.

Métodos	
Modificador y Tipo	**Método y Descripción**
int	sendDynamicDataRF(java.lang.String dato_RF, char[] ACK_Data, boolean[] confirm) Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF.
void	setInitialDPLState(boolean DPL_State) Configura el valor inicial de la función longitud de datos dinámica (DPL).
void	setInitialRFChannel(int RF_Channel) Configura el valor inicial del canal de frecuencia.
void	setInitialRFMode(boolean RF_Initial_Mode) Configura el valor inicial del transceptor (RX o TX).
void	setInitialRFSpeed(int RF_Speed) Configura el valor inicial de la velocidad de transmisión de datos de radiofrecuencia.
void	setInitialRxAddress(java.lang.String RX_Address) Configura el valor inicial de la dirección de recepción (RX).
void	setInitialRXLength(int RX_Length) Configura el valor inicial de longitud de datos fija para recibir.
void	setInitialTxAddress(java.lang.String TX_Address) Configura el valor inicial de la dirección de transmisión (TX).
void	setInitialTXLength(int TX_Length) Configura el valor inicial de longitud de datos fija para transmitir.
int	setRFChannel(int Canal_RF) Configura el canal de frecuencia del transceptor.
void	setRFMode(boolean Modo) Configura el valor del transceptor.

Métodos	
Modificador y Tipo	**Método y Descripción**
int	setRFSpeed(int vel) Método que permite configurar la velocidad de transmisión del transceptor.
void	setRXAddress(java.lang.String RX_Address) Configura el valor de dirección asignado para recibir (RX).
int	setRXLength(int RX_Len) Configura el valor de longitud de datos fija para recibir (RX).
void	setTXAddress(java.lang.String TX_Address) Configura el valor de dirección asignado para transmitir (TX).
int	setTXLength(int TX_Len) Configura el valor de longitud de datos fija para transmitir (TX).
void	turnLedOff() Apaga el Led conectado al dispositivo.
void	turnLedOn() Enciende el Led conectado al dispositivo.
void	writenrf24l01register(int register, int reg_value) Configura el canal de frecuencia del transceptor.

Métodos Detallados

disableDPL

public void disableDPL()

Deshabilita la función de datos de longitud dinámica (DPL).

enableDPL

public void enableDPL()

Habilita la función de datos de longitud dinámica (DPL).

getDPLState

public boolean getDPLState()

Regresa el estado de la función de datos de longitud variable (DPL).

Regresa:

boolean.

True - La función DPL esta activada.

False - La función DPL esta desactivada.

getInitialValues

public java.lang.String getInitialValues()

Lee los valores de configuración iniciales almacenados en la EEPROM del dispositivo.

Regresa:

String.

Regresa los valores que tendrá el dispositivo cada que vez que es conectado, éstos son:

- Modo de RF. Para configurar éste valor usar el método setInitialRFMode.
- Canal de RF. Para configurar éste valor usar el método setInitialRFChannel.
- Velocidad de RF. Para configurar éste valor usar el método setInitialRFSpeed.
- Dirección de transmisión (TX). Para configurar éste valor usar el método setInitialTxAddress.
- Longitud de datos fija para transmitir (TX). Para configurar éste valor usar el método setInitialTXLength.
- Dirección de recepción (RX). Para configurar éste valor usar el método setInitialRxAddress.
- Longitud de datos fija para recibir (RX). Para configurar éste valor usar el método setInitialRXLength.

- Longitud de datos dinámicos habilitados (DPL). Para configurar éste valor usar el método setInitialDPLState.

getRFChannel

public int getRFChannel()

Lee el canal de frecuencia del transceptor.

Regresa:

int

Regresa el canal de frecuencia actual del transceptor como un valor de 0 a 127. (2400MHz - 2527MHz).

getRFMode

public boolean getRFMode()

Regresa el modo de RF en el que se encuentra actualmente el dispositivo.

Regresa:

boolean.

True - El dispositivo se encuentra en modo transmisor (TX).

False - El dispositivo se encuentra en modo receptorr (RX).

getRFSpeed

public int getRFSpeed()

Método que permite obtener la velocidad de transmisión del transceptor.

Regresa:

Regresa la velocidad del transceptor (Valores del 1 al 3).

Velocidad de 2Mbps - 1

Velocidad de 1Mbps - 2

Velocidad de 250Kbps - 3

getRXAddress

public java.lang.String getRXAddress()

Obtiene el valor de dirección asignado para recibir (RX).

Regresa:

String

Regresa en una cadena de texto de 5 Bytes el valor asignado como dirección de recepción (RX).

getRXLength

public int getRXLength()

Regresa la longitud de datos fijos asignados para recibir (RX).
Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud variable (DPL).

Regresa:

int.

Valor asignado al registro "RX_ADDR_P0" del nRF24L01+

getTXAddress

public java.lang.String getTXAddress()

Obtiene el valor de dirección asignado para transmitir (TX).

Regresa:

String

Regresa en una cadena de texto de 5 Bytes el valor asignado como dirección de transmisión (TX).

getTXLength

public int getTXLength()

Regresa la longitud de datos fijos asignados para transmitir (TX).
Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud variable (DPL).

Regresa:

int.
Valor asignado al registro "RX_ADDR_P1" del nRF24L01+

readDynamicRFData

public boolean readDynamicRFData(java.lang.String[] Received_Data)

Lee los datos de longitud dinámica (DPL) recibidos por RF.

Parámetros:

Received_Data - Buffer que almacenará los datos en caso de que se hayan recibido.

Regresa:

boolean.
True - Se recibieron datos.
False - No se recibieron datos.

readDynamicRFData

public boolean readDynamicRFData(int[] Received_Data)

Lee los datos de longitud dinámica (DPL) recibidos por RF.

Parámetros:

Received_Data - Buffer que almacenará los datos en caso de que se hayan recibido.

Regresa:

boolean.
True - Se recibieron datos.

False - No se recibieron datos.

readDynamicRFData

public boolean readDynamicRFData(java.lang.String[] Received_Data,
java.lang.String ACK_Data)

Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.

Parámetros:

Received_Data - Buffer que almacenará los datos en caso de que se hayan recibido.

ACK_Data - Datos a enviar a través del acuse de recibo. La longitud no debe exceder los 32 Bytes.

Nota: Los datos en acuse de recibo se enviarán al siguiente mensaje recibido solamente.

Regresa:

boolean.

True - Se recibieron datos.

False - No se recibieron datos.

readDynamicRFData

public boolean readDynamicRFData(int[] Received_Data,
java.lang.String ACK_Data)

Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.

Parámetros:

Received_Data - Buffer que almacenará los datos en caso de que se hayan recibido.

ACK_Data - Datos a enviar a través del acuse de recibo. La longitud no debe exceder los 32 Bytes.

Nota: Los datos en acuse de recibo se enviarán al siguiente mensaje recibido solamente.

Regresa:

boolean.

True - Se recibieron datos.

False - No se recibieron datos.

readnrf24l01register

public int readnrf24l01register(int register)

Lee el registro indicado del nRF24L01.

Parámetros:

register - Registro al que se desea leer.

Regresa:

int

Regresa el valor del registro

sendCommandDataRF

public boolean sendCommandDataRF(int Comando,
int Dato)

Envía un dato y un comando de 2 Bytes vía RF.

Parámetros:

Comando - Comando de 2 Bytes que se envia al receptor, utilizar valores de 0 a 65,535.

Dato - Dato a enviar para el comando dado, valores válidos de 0 a 65,535.

Regresa:

boolean.

True - Se envío correctamente el dato.

False - No se logró enviar el dato.

sendDataRF

public boolean sendDataRF(int[] dato_RF)

Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF. El dispositivo deberá estar en modo transmisor (TX) y estar definido un valor para la longitud de datos de transmisión, usar el método setTxLength.

Parámetros:

dato_RF - Datos a envíar por RF. El mensaje deberá estar compuesto por segmentos de Bytes (valores de 0 a 255), no ser mayor a 32 Bytes y de longitud idéntica al definido en el método setTxLength.

Regresa:

boolean.

True - Se envió correctamente el dato.

False - No se logró enviar el dato.

sendDataRF

public boolean sendDataRF(java.lang.String dato_RF)

Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF. El dispositivo deberá estar en modo transmisor (TX) y estar definido un valor para la longitud de datos de transmisión, usar el método setTxLength.

Parámetros:

dato_RF - Datos a envíar por RF. El mensaje deberá estar compuesto por caracteres ASCII y no ser mayor a 32. longitud idéntica al definido en el método setTxLength.

Regresa:

boolean.

True - Se envió correctamente el dato.

False - No se logró enviar el dato.

sendDynamicDataRF

public int sendDynamicDataRF(int[] dato_RF,
char[] ACK_Data,
boolean[] confirm)

Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF. El dispositivo deberá estar en modo transmisor (TX) y estar activada la función de datos de longitud variable (DPL), usar el método enableDPL.

Parámetros:

dato_RF - Datos a envíar por RF. El mensaje deberá estar compuesto por segmentos de Bytes de tipo entero (valores de 0 a 255) y no ser mayor a 32 Bytes.

ACK_Data - Buffer que almacenará los datos de acuse de recibo (ACK) en caso de que existan.

confirm - Buffer que indicará si el dato llegó al receptor (RX).

True - El dato se envió correctamente. False - El dato no llegó al receptor (No se recibió acuse de recibo ACK).

Regresa:

int.

Regresa cuantos datos se recibieron en el acuse de recibo (ACK), si es que se recibió alguno.

sendDynamicDataRF

public int sendDynamicDataRF(java.lang.String dato_RF,
char[] ACK_Data,
boolean[] confirm)

Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF. El dispositivo deberá estar en modo transmisor (TX) y estar activada la función de datos de longitud variable (DPL), usar el método enableDPL.

Parámetros:

dato_RF - Datos a envíar por RF. El mensaje tipo cadena no debe ser mayor a 32 caracteres y deberán ser del tipo ASCII.

ACK_Data - Buffer que almacenará los datos de acuse de recibo (ACK) en caso de que existan.

confirm - Buffer que indicará si el dato llegó al receptor (RX).

True - El dato se envió correctamente. False - El dato no llegó al receptor (No se recibió acuse de recibo ACK).

Regresa:

int.

Regresa cuantos datos se recibieron en el acuse de recibo (ACK), si es que se recibió alguno.

setInitialDPLState

public void setInitialDPLState(boolean DPL_State)

Configura el valor inicial de la función longitud de datos dinámica (DPL).

Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

DPL_State - Estado inicial del DPL.

True: La función DPL estará habilitada desde que el dispositivo se conecta.

False: La función DPL no estará activada cuando el dispositivo se conecte.

setInitialRFChannel

public void setInitialRFChannel(int RF_Channel)

Configura el valor inicial del canal de frecuencia.

Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

RF_Channel - Dirección de radiofrecuencia, valores válidos de 0 a 127 (2400Mhz - 2527MHz).

setInitialRFMode

public void setInitialRFMode(boolean RF_Initial_Mode)

Configura el valor inicial del transceptor (RX o TX).

Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

RF_Initial_Mode - Modo inicial de RF para el transceptor nRF24L01+.

True: El dispositivo iniciara como transmisor (TX). False: El dispositivo iniciara como receptor (RX).

setInitialRFSpeed

public void setInitialRFSpeed(int RF_Speed)

Configura el valor inicial de la velocidad de transmisión de datos de radiofrecuencia.

Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

RF_Speed - Velocidad de radiofrecuencia. Valores válidos:
-RF_Speed_2Mbps - 1
-RF_Speed_1Mbps - 2
-RF_Speed_250Kbps - 3

setInitialRxAddress

public void setInitialRxAddress(java.lang.String RX_Address)

Configura el valor inicial de la dirección de recepción (RX).
Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

RX_Address - Dirección de recepción (RX). La dirección deberá ser igual a 5 Bytes.

setInitialRXLength

public void setInitialRXLength(int RX_Length)

Configura el valor inicial de longitud de datos fija para recibir.
Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

RX_Length - Cantidad de Bytes fijos que contendrá cada mensaje de recepción (RX). Valores válidos de 1 a 32.

setInitialTxAddress

public void setInitialTxAddress(java.lang.String TX_Address)

Configura el valor inicial de la dirección de transmisión (TX).
Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

TX_Address - Dirección de transmision (TX). La dirección deberá ser igual a 5 Bytes.

setInitialTXLength

public void setInitialTXLength(int TX_Length)

Configura el valor inicial de longitud de datos fija para transmitir.
Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.

Parámetros:

TX_Length - Cantidad de Bytes fijos que contendrá cada mensaje de transmisión (TX). Valores válidos de 1 a 32.

setRFChannel

public int setRFChannel(int Canal_RF)

Configura el canal de frecuencia del transceptor.

Parámetros:

Canal_RF - Canal que se asigna al tansceptor, utilizar sólo valores de 0 a 127. (2000MHz - 2127MHz).

Regresa:

int.

Regresa el canal de frecuencia actual del transceptor como un valor de 0 a 127. (2000MHz - 2127MHz).

setRFMode

public void setRFMode(boolean Modo)

Configura el valor del transceptor.

Parámetros:

Modo - Representa el modo del transceptor.

True: Transmisor (TX).

False: Receptor (RX).

setRFSpeed

public int setRFSpeed(int vel)

Método que permite configurar la velocidad de transmisión del transceptor.

Parámetros:

vel - Velocidad que se asigna al tansceptor, los valores válidos son:

RF_Speed_2Mbps - 1

RF_Speed_1Mbps - 2

RF_Speed_250Kbps - 3

Regresa:

Regresa la velocidad del transceptor como un valor del 1 al 3.

Velocidad de 2Mbps - 1

Velocidad de 1Mbps - 2

Velocidad de 250Kbps - 3

setRXAddress

public void setRXAddress(java.lang.String RX_Address)

Configura el valor de dirección asignado para recibir (RX).

Parámetros:

RX_Address - Dirección de recepción (TX). La dirección deberá ser igual a 5 Bytes.

setRXLength

public int setRXLength(int RX_Len)

Configura el valor de longitud de datos fija para recibir (RX). Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud dinámica (DPL).

Parámetros:

RX_Len - Cantidad de Bytes fijos que contendrá cada mensaje de recepción(RX). Valores válidos de 1 a 32.

setTXAddress

public void setTXAddress(java.lang.String TX_Address)

Configura el valor de dirección asignado para transmitir (TX).

Parámetros:

TX_Address - Dirección de transmisión (TX). La dirección deberá ser igual a 5 Bytes.

setTXLength

public int setTXLength(int TX_Len)

Configura el valor de longitud de datos fija para transmitir (TX). Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud dinámica (DPL).

Parámetros:

TX_Len - Cantidad de Bytes fijos que contendrá cada mensaje de transmisión (TX). Valores válidos de 1 a 32.

turnLedOff

public void turnLedOff()

Apaga el Led conectado al dispositivo.

turnLedOn

public void turnLedOn()

Enciende el Led conectado al dispositivo.

writenrf24l01register

public void writenrf24l01register(int register,
int reg_value)

Configura el canal de frecuencia del transceptor.

Parámetros:

register - Registro a escribir.

reg_value - Valor del registro.

APÉNDICE 8: CLASE "USBRFDEVICE.JAVA"

```
package Javausb;



 //importa libreria Java libusb.
import ch.ntb.usb.*;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

/**
* Ésta clase permite la comunicación inalámbrica entre la computadora
* y otro dispositivo de control que cuente con el transceptor nRF24L01+.
* Se requiere de un dispositivo AVR con V-USB, LibUSb y un dispositivo NRF24L01+.
* Vendor Id: 0x20A0
* Product Id: 0x4204
* @author Rubén Flores Altamirano.
* @version 1.0
*/

public class UsbRFDevice {


public final static JPanel panelerror = new JPanel();

// Definiciones para mensajes de control
static int bRequest=0;
static int nBytes=0;
static int wValue=0;
static int wIndex=0;
private static final short idVendor = 0x20A0;
private static final short idProduct = 0x4204;

// Peticiones creadas en bRequest (Deben ser definididas con los
// mismos valores en el dispositivo USB)

private static final int bRequest_Send_Data_RF   = 1;
private static final int bRequest_Send_Data_USB  = 2;
private static final int bRequest_Receive_Data_RF        = 3;
private static final int bRequest_Send_Dynamic_Data_RF   = 4;
private static final int bRequest_Read_Dynamic_Data_RF   = 5;
private static final int bRequest_Read_RF_Data   = 8;
private static final int bRequest_Enable_DPL     = 10;
private static final int bRequest_Disable_DPL    = 11;
private static final int bRequest_Set_TX_Mode    = 12;
private static final int bRequest_Set_TX_Address = 13;
private static final int bRequest_Get_TX_Address = 14;
private static final int bRequest_Set_RX_Mode    = 15;
private static final int bRequest_Set_RX_Address = 16;
private static final int bRequest_Get_RX_Address = 17;
private static final int bRequest_Set_TX_Length  = 18;
private static final int bRequest_Set_RX_Length  = 19;
private static final int bRequest_Get_TX_Length  = 20;
private static final int bRequest_Get_RX_Length  = 21;
private static final int bRequest_Get_DPL_State  = 22;
private static final int bRequest_Get_RF_Mode    = 23;
private static final int bRequest_Read_nRF24L01_Register = 30;
private static final int bRequest_Write_nRF24L01_Register = 31;
private static final int bRequest_Set_Channel_RF = 50;
private static final int bRequest_Get_Channel_RF = 51;
private static final int bRequest_Set_Data_Speed = 60;
private static final int bRequest_Get_Data_Speed = 61;
private static final int bRequest_Turn_LED_On    = 200;
private static final int bRequest_Turn_LED_Off   = 201;
private static final int bRequest_Blink_LED      = 202;
private static final int bRequest_Set_Initial_TX_Address = 210;
```

```
private static final int bRequest_Set_Initial_RX_Address= 215;
private static final int bRequest_Set_Initial_RF_Channel= 220;
private static final int bRequest_Set_Initial_RF_Speed  = 225;
private static final int bRequest_Set_Initial_RF_Mode   = 230;
private static final int bRequest_Set_Initial_TX_Length = 235;
private static final int bRequest_Set_Initial_RX_Length = 240;
private static final int bRequest_Set_Initial_DPL_State = 245;
private static final int bRequest_Get_Initial_Values    = 250;

// Parámetros de RF.

/** Numero de bytes recibidos RF. (Si no se tiene habilitado DPL)
 *
 */
public int RX_Length    =       32;
/** Numero de bytes enviados RF. (Si no se tiene habilitado DPL)
 *
 */
//
public int TX_Length    =       4;

/** Velocidad de RF de 250 Kbps. <br>
 * Se utiliza para configurar {@link #setRFSpeed(int) setRFSpeed} y {@link #setInitialRFSpeed(int)
SetInitialRFspeed}
 */
public static final int RF_Speed_250Kbps= 3;
/** Velocidad de RF de 1 Mbps. <br>
 * Se utiliza para configurar {@link #setRFSpeed(int) setRFSpeed} y {@link #setInitialRFSpeed(int)
SetInitialRFspeed}
 */
public static final int RF_Speed_1Mbps  = 2;
/** Velocidad de RF de 2 Mbps. <br>
 * Se utiliza para configurar {@link #setRFSpeed(int) setRFSpeed} y {@link #setInitialRFSpeed(int)
SetInitialRFspeed}
 */
public static final int RF_Speed_2Mbps  = 1;

// Parámetros de estado.
private static boolean DPL_State = false;


/**
* Envía un dato y un comando de 2 Bytes vía RF.
* @param Comando Comando de 2 Bytes que se envia al receptor, utilizar valores de 0 a 65,535.
* @param Dato Dato a enviar para el comando dado, valores válidos de 0 a 65,535.
* @return boolean.
* <br> True - Se envío correctamente el dato.
* <br> False - No se logró enviar el dato.
*/
public boolean sendCommandDataRF(int Comando,int Dato)
{
        Dato &= 0xFFFF;
        Comando &= 0xFFFF;



        if(getDPLState() != true)
        {
        if (getTXLength() != 4 )
        {
        setTXLength(4);
        }
        }

        Device dev = USB.getDevice(idVendor, idProduct);


        try {
```

```
        dev.open(1, 0, -1);
        byte[] dato_RF= new byte [4];

        dato_RF[0]=(byte) ((Comando >> 8) & 0xFF);
        dato_RF[1] = (byte) (Comando & 0xFF);
        dato_RF[2]=(byte) ((Dato >> 8) & 0xFF);
        dato_RF[3]=(byte) (Dato & 0xFF);

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, dato_RF, dato_RF.length, 2000, false);



        byte [] data= new byte[1];
        int confirm =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Send_Data_RF, wValue, wIndex, data, data.length, 2000, false);
        dev.close();

        if (confirm == 0)
        {
        return false;
        }

        else
        {
        return true;
        }

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}


/**
* Regresa la longitud de datos fijos asignados para transmitir (TX). <br>
* Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud variable (DPL).
* @return int.
* <br> Valor asignado al registro "RX_ADDR_P1" del nRF24L01+
*/

public int getTXLength()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] TX_Len= new byte [1];


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Get_TX_Length, wValue, wIndex, TX_Len, TX_Len.length, 2000, false);

        dev.close();

        TX_Len[0] &= 0xFF;

        TX_Length = TX_Len[0];
```

```
        return (int) TX_Len[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}

/**
* Regresa la longitud de datos fijos asignados para recibir (RX). <br>
* Éste valor sólo es válido si no se tiene habilitada la función de datos de longitud variable (DPL).
* @return int.
* <br> valor asignado al registro "RX_ADDR_P0" del nRF24L01+
*/


public int getRXLength()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] RX_Length= new byte [1];

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Get_RX_Length, wValue, wIndex, RX_Length, RX_Length.length, 2000, false);

        dev.close();

        RX_Length[0] &= 0xFF;

        return (int) RX_Length[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}


/**
* Regresa el estado de la función de datos de longitud variable (DPL).
* @return boolean.
* <br>  True - La función DPL esta activada.
* <br>  False - La función DPL esta desactivada.
*/

public boolean getDPLState()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] DPL_State= new byte [1];
```

```
        int confirm =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Get_DPL_State, wValue, wIndex, DPL_State, DPL_State.length, 2000, false);

        dev.close();

        if (confirm == 0)
        {
        return false;
        }

        else
        {
        return true;
        }
        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}

/**
* Regresa el modo de RF en el que se encuentra actualmente el dispositivo.
* @return boolean.
* <br> True - El dispositivo se encuentra en modo transmisor (TX).
* <br> False - El dispositivo se encuentra en modo receptorr (RX).
*/

public boolean getRFMode()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] RF_Mode= new byte [1];


        int confirm =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Get_RF_Mode, wValue, wIndex, RF_Mode, RF_Mode.length, 2000, false);

        dev.close();

        if (confirm == 0)
        {
        return false;
        }

        else
        {
        return true;
        }
        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}
```

```
/**
* Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF.
* El dispositivo deberá estar en modo transmisor (TX) y estar definido
* un valor para la longitud de datos de transmisión, usar el método
* {@link #setTxLength(int) setTxLength}.
* @param dato_RF Datos a envíar por RF. El mensaje deberá estar compuesto por
* segmentos de Bytes (valores de 0 a 255), no ser mayor a 32 Bytes y de
* longitud idéntica al definido en el método {@link #setTxLength(int) setTxLength}.
* @return boolean.
* <br> True - Se envió correctamente el dato.
* <br> False - No se logró enviar el dato.
*/

public boolean sendDataRF(int[] dato_RF)
{
        if (dato_RF.length > TX_Length)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de los datos debe ser igual a TX
Length: " + TX_Length+ " Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
        }

        int i;
        for(i=0; i < dato_RF.length ;i++)
        {
        if (dato_RF[i]>255 || dato_RF[i]<0)
        {
        JOptionPane.showMessageDialog(panelerror, "El valor de cada dato debe ser entre 0 y 255",
"Error", JOptionPane.ERROR_MESSAGE);
        return false;
        }
        dato_RF[i] &= 0xFF;
        }

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        dev.open(1, 0, -1);

        byte[] datos_RF= new byte [dato_RF.length];
        for(i=0; i < dato_RF.length ;i++)
        {
        datos_RF[i]= (byte) dato_RF[i];
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, datos_RF, datos_RF.length, 2000, false);



        byte [] data= new byte[TX_Length];

        int confirm =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Send_Data_RF, wValue, wIndex, data, data.length, 2000, false);
        dev.close();

        if (confirm == 0)
        {
        return false;
        }

        else
        {
        return true;
        }
```

```
        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}

/**
* Envía un mensaje de datos de longitud fija de hasta 32 Bytes vía RF.
* El dispositivo deberá estar en modo transmisor (TX) y estar definido
* un valor para la longitud de datos de transmisión, usar el método
* {@link #setTxLength(int) setTxLength}.
* @param dato_RF Datos a envíar por RF. El mensaje deberá estar compuesto por
* caracteres ASCII y no ser mayor a 32.
* longitud idéntica al definido en el método {@link #setTxLength(int) setTxLength}.
* @return boolean.
* <br> True - Se envió correctamente el dato.
* <br> False - No se logró enviar el dato.
*/

public boolean sendDataRF(String dato_RF)
{
        if (dato_RF.length() > TX_Length)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de los datos debe ser igual a TX
Length: " + TX_Length+ " Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return false;
        }

        int i;


        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        dev.open(1, 0, -1);

        byte[] datos_RF= new byte [dato_RF.length()];
        for(i=0; i < dato_RF.length() ;i++)
        {
        datos_RF[i]= (byte) dato_RF.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, datos_RF, datos_RF.length, 2000, false);



        byte [] data= new byte[TX_Length];

        int confirm =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Send_Data_RF, wValue, wIndex, data, data.length, 2000, false);
        dev.close();

        if (confirm == 0)
        {
        return false;
        }

        else
        {
        return true;
```

```
        }

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}

/**
* Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF.
* El dispositivo deberá estar en modo transmisor (TX) y estar activada
* la función de datos de longitud variable (DPL), usar el método
* {@link #enableDPL() enableDPL}.
* @param dato_RF Datos a envíar por RF. El mensaje deberá estar compuesto por
* segmentos de Bytes de tipo entero (valores de 0 a 255) y no ser mayor a 32 Bytes.
* @param ACK_Data Buffer que almacenará los datos de acuse de recibo (ACK) en
* caso de que existan.
* @param confirm Buffer que indicará si el dato llegó al receptor (RX). <br>
* True - El dato se envió correctamente.
* False - El dato no llegó al receptor (No se recibió acuse de recibo ACK).
* @return int.
* <br> Regresa cuantos datos se recibieron en el acuse de recibo (ACK),
* si es que se recibió alguno.
*/

public int sendDynamicDataRF(int[] dato_RF, char [] ACK_Data, boolean confirm[])
{


        if (DPL_State == false)
        {
        if(getDPLState() != true || DPL_State == false)
        {
        enableDPL();
        DPL_State = true;
        }
        }

        if (dato_RF.length>32)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de los datos no debe exceder los 32
Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return 0;
        }

        int i;
        for(i=0; i < dato_RF.length ;i++)
        {
        if (dato_RF[i]>255 || dato_RF[i]<0)
        {
        JOptionPane.showMessageDialog(panelerror, "El valor de cada dato debe ser entre 0 y 255",
"Error", JOptionPane.ERROR_MESSAGE);
        return 0;
        }
        dato_RF[i] &= 0xFF;
        }



        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);
```

```
        byte[] datos_RF= new byte [dato_RF.length];
        for(i=0; i < dato_RF.length ;i++)
        {
        datos_RF[i]= (byte) dato_RF[i];
        }

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, datos_RF, datos_RF.length, 2000, false);


        wValue= dato_RF.length;
        byte [] data_read= new byte[32];
        int data_length;
        data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Send_Dynamic_Data_RF, wValue, wIndex, data_read, data_read.length, 2000, false);
        dev.close();

        confirm[0] = true;

        if (data_length>0)
        {


        for(i=0; i < data_length ;i++)
        {
        ACK_Data[i]= (char) data_read[i];
        ACK_Data[i] &= 0xFF;
        }

        confirm[0] = true;

        if (new String(ACK_Data).contains("ERR00"))
        {
        data_length = 0;
        confirm[0] = false;
        }
        }

        return data_length;


        }
        catch (USBException e)
        {
        // if an exception occurs during connect or read/write an exception
        // is thrown
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}


/**
* Envía un mensaje de longitud variable (DPL) de hasta 32 Bytes vía RF.
* El dispositivo deberá estar en modo transmisor (TX) y estar activada
* la función de datos de longitud variable (DPL), usar el método
* {@link #enableDPL() enableDPL}.
* @param dato_RF Datos a envíar por RF. El mensaje tipo cadena no debe ser
* mayor a 32 caracteres y deberán ser del tipo ASCII.
* @param ACK_Data Buffer que almacenará los datos de acuse de recibo (ACK) en
* caso de que existan.
* @param confirm Buffer que indicará si el dato llegó al receptor (RX). <br>
* True - El dato se envió correctamente.
```

```
* False - El dato no llegó al receptor (No se recibió acuse de recibo ACK).
* @return int.
* <br> Regresa cuantos datos se recibieron en el acuse de recibo (ACK),
* si es que se recibió alguno.
*/

public int sendDynamicDataRF(String dato_RF, char [] ACK_Data, boolean confirm[])
{

        if (DPL_State == false)
        {
        if(getDPLState() != true || DPL_State == false)
        {
        enableDPL();
        DPL_State = true;
        }
        }

        if (dato_RF.length() >32)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de los datos no debe exceder los 32
Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return 0;
        }

        int i;




        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);
        byte[] datos_RF= new byte [dato_RF.length()];
        for(i=0; i < dato_RF.length() ;i++)
        {
        datos_RF[i]= (byte) dato_RF.charAt(i);
        }

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, datos_RF, datos_RF.length, 2000, false);


        wValue= dato_RF.length();
        byte [] data_read= new byte[32];
        int data_length;
        data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Send_Dynamic_Data_RF, wValue, wIndex, data_read, data_read.length, 2000, false);
        dev.close();

        confirm[0] = true;

        if (data_length>0)
        {


        for(i=0; i < data_length ;i++)
        {
        ACK_Data[i]= (char) data_read[i];
        ACK_Data[i] &= 0xFF;
        }

        confirm[0] = true;
```

```
        if (new String(ACK_Data).contains("ERR00"))
        {
        data_length = 0;
        confirm[0] = false;
        }
        }

        return data_length;


        }
        catch (USBException e)
        {
        // if an exception occurs during connect or read/write an exception
        // is thrown
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}


/**
* Configura el canal de frecuencia del transceptor.
* @param Canal_RF Canal que se asigna al tansceptor, utilizar sólo valores de 0 a 127. (2000MHz -
2127MHz).
* @return int.
* <br> Regresa el canal de frecuencia actual del transceptor como un valor de 0 a 127. (2000MHz -
2127MHz).
*/
public int setRFChannel(int Canal_RF)
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        Canal_RF &= 0xFF;
        if (Canal_RF<0 || Canal_RF>127)
        {
        JOptionPane.showMessageDialog(panelerror, "Seleccione una valor entre 0 y 127", "Error",
JOptionPane.ERROR_MESSAGE);
        return 0;
        }

        dev.open(1, 0, -1);

        wValue=Canal_RF;
        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Channel_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
```

```
}


/**
* Configura el canal de frecuencia del transceptor.
* @param register Registro a escribir.
* @param reg_value Valor del registro.
*/
public void writenrf24l01register(int register, int reg_value)
{


        if(register < 0 || reg_value <0)
        {
        register = 0;
        reg_value = 0;
        }

        if(register >255 || reg_value > 255)
        {
        register = 255;
        reg_value = 255;
        }

        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);

        register &= 0xFF;
        reg_value &= 0xFF;


        wValue = register;
        wIndex = reg_value;
        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Write_nRF24L01_Register, wValue, wIndex, data, data.length, 2000, false);

        dev.close();




        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        }


}


/**
* Lee el canal de frecuencia del transceptor.
* @return int
* <br> Regresa el canal de frecuencia actual del transceptor como un valor de 0 a 127. (2400MHz -
2527MHz).
*/
public int getRFChannel()
```

```
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Get_Channel_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        data[0] &= 0xFF;

        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}

/**
* Lee el registro indicado del nRF24L01.
* @param register Registro al que se desea leer.
* @return int
* <br> Regresa el valor del registro
*/
public int readnrf24l01register(int register)
{
        register &= 0xFF;

        if (register <0)
        {
        register =0;
        }
        if (register >255)
        {
        register =255;
        }

        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        wValue = register;
        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Read_nRF24L01_Register, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return (int)    data[0]&0xFF;

        }
        catch (USBException e)
        {
```

```
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
}


 /**
* Lee los valores de configuración iniciales almacenados en la EEPROM del dispositivo.
* @return String.
* <br> Regresa los valores que tendrá el dispositivo cada que vez que es conectado, éstos son:
* <br> - Modo de RF. Para configurar éste valor usar el método {@link #setInitialRFMode(boolean)
setInitialRFMode}.
* <br> - Canal de RF. Para configurar éste valor usar el método {@link
#setInitialRFChannel(int)setInitialRFChannel}.
* <br> - Velocidad de RF. Para configurar éste valor usar el método {@link
#setInitialRFSpeed(int)setInitialRFSpeed}.
* <br> - Dirección de transmisión (TX). Para configurar éste valor usar el método {@link
#setInitialTxAddress(java.lang.String) setInitialTxAddress}.
* <br> - Longitud de datos fija para transmitir (TX). Para configurar éste valor usar el método
{@link #setInitialTXLength(int)setInitialTXLength}.
* <br> - Dirección de recepción (RX). Para configurar éste valor usar el método {@link
#setInitialRxAddress(java.lang.String)setInitialRxAddress}.
* <br> - Longitud de datos fija para recibir (RX). Para configurar éste valor usar el método {@link
#setInitialRXLength(int)setInitialRXLength}.
* <br> - Longitud de datos dinámicos habilitados (DPL). Para configurar éste valor usar el método
{@link #setInitialDPLState(boolean)setInitialDPLState}.
*/

public String getInitialValues()
        {
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] eepromValues= new byte [16];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Get_Initial_Values, wValue, wIndex, eepromValues, eepromValues.length, 2000,
false);

        dev.close();

        char Val[] = new char [16];

        int i;

        for (i=0;i<15;i++)
        {
        Val[i] = (char) eepromValues[i];
        Val[i] &= 0xFF;
        }


        String RF_Mode, RF_Channel, RF_Speed, TX_Ad, TX_Len, RX_Ad, RX_Len, DPL_State;

        if(eepromValues[0] ==0)
        RF_Mode = "RX";
        else
        RF_Mode = "TX";

        RF_Channel = Integer.toString((int) Val[1]);

        if(eepromValues[2] ==RF_Speed_2Mbps)
        {
        RF_Speed = "2 Mbps";
```

```
        }
        else if(eepromValues[2] ==RF_Speed_1Mbps)
        {
        RF_Speed = "1 Mbps";
        }
        else
        {
        RF_Speed = "250 Kbps";
        }
        TX_Ad = ""+Val[3]+Val[4]+Val[5]+Val[6]+Val[7];

        TX_Len = Integer.toString((int) Val[8])+" Bytes";

        RX_Ad = ""+Val[9]+Val[10]+Val[11]+Val[12]+Val[13];

        RX_Len = Integer.toString((int) Val[14])+" Bytes";

        if(eepromValues[15] ==0)
        DPL_State = "DPL Disabled";
        else
        DPL_State = "DPL Enabled";

        return new String("RF Mode: "+RF_Mode+"\nRF Channel: "+
        RF_Channel+"\nRF Speed: "+RF_Speed+"\nTX Address: "+
        TX_Ad+"\nTX Data Length: "+TX_Len+"\nRX Address: "+
        RX_Ad+"\nRX Data Length: "+RX_Len+"\nDPL State: "+DPL_State);




        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return null;
        }



}

/**
* Método que permite configurar la velocidad de transmisión del transceptor.
* @param vel Velocidad que se asigna al tansceptor, los valores válidos son: <br>
* {@link #RF_Speed_2Mbps RF_Speed_2Mbps} - 1 <br>
* {@link #RF_Speed_1Mbps RF_Speed_1Mbps} - 2 <br>
* {@link #RF_Speed_250Kbps RF_Speed_250Kbps} - 3
* @return Regresa la velocidad del transceptor como un valor del 1 al 3. <br>
* Velocidad de 2Mbps- 1 <br>
* Velocidad de 1Mbps- 2 <br>
* Velocidad de 250Kbps - 3
*/
public int setRFSpeed(int vel)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        vel &= 0xFFFF;
        if (vel<1 || vel>3)
        {
        JOptionPane.showMessageDialog(panelerror, "Seleccione un valor de velocidad permitido.",
"Error", JOptionPane.ERROR_MESSAGE);
        return 0;
        }
```

```
        dev.open(1, 0, -1);

        wValue=vel;
        byte[] data= new byte [1];

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Data_Speed, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }



}

/**
* Método que permite obtener la velocidad de transmisión del transceptor.
* @return Regresa la velocidad del transceptor (Valores del 1 al 3). <p>
* Velocidad de 2Mbps- 1 <br>
* Velocidad de 1Mbps- 2 <br>
* Velocidad de 250Kbps - 3
*/

public int getRFSpeed()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Get_Data_Speed, wValue, wIndex, data, data.length, 2000, false);


        dev.close();


        data[0] &= 0xFF;

        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }
```

```
}


/**
* Enciende el Led conectado al dispositivo.
*/

 public void turnLedOn()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Turn_LED_On, wValue, wIndex, data, data.length, 2000, false);


        dev.close();

        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

/**
* Apaga el Led conectado al dispositivo.
*/
 public void turnLedOff()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Turn_LED_Off, wValue, wIndex, data, data.length, 2000, false);


        dev.close();

        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
```

```
        System.exit(0);
        return;
        }



}

/**
* Parpadea el Led conectado al dispositivo.
*/
 void blinkLed()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);

        byte[] data= new byte [1];

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Blink_LED, wValue, wIndex, data, data.length, 2000, false);


        dev.close();

        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Obtiene el valor de dirección asignado para transmitir (TX).
* @return String
* <br> Regresa en una cadena de texto de 5 Bytes el valor asignado como
* dirección de transmisión (TX).
*/

 public String getTXAddress()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] data= new byte [5];
        int addressLength =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Get_TX_Address, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        char[] chrs = new char [addressLength];
        int i;
        for (i=0;i<addressLength;i++)
        {
```

```
        chrs[i] = (char) data[i];
        }
        String TX_Address = new String (chrs);

        return TX_Address;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return null;
        }
}

 /**
* Obtiene el valor de dirección asignado para recibir (RX).
* @return String
* <br> Regresa en una cadena de texto de 5 Bytes el valor asignado como
* dirección de recepción (RX).
*/

 public String getRXAddress()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] data= new byte [5];
        int addressLength =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Get_RX_Address, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        char[] chrs = new char [addressLength];
        int i;
        for (i=0;i<addressLength;i++)
        {
        chrs[i] = (char) data[i];
        }
        String RX_Address = new String (chrs);

        return RX_Address;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return null;
        }
}

 /**
* Configura el valor inicial de la dirección de transmisión (TX). <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param TX_Address Dirección de transmision (TX). La dirección deberá ser igual a 5 Bytes.
*/

 public void setInitialTxAddress(String TX_Address)
{

        Device dev = USB.getDevice(idVendor, idProduct);
```

```
        try {
        if (TX_Address.length() != 5)
        {
        JOptionPane.showMessageDialog(panelerror, "La Dirección de TX debe ser igual a 5 Bytes.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        dev.open(1, 0, -1);
        int i;

        byte[] data= new byte [5];
        for (i=0;i<5;i++)
        {
        data[i] = (byte) TX_Address.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, data, data.length, 2000, false);



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_TX_Address, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor inicial de la dirección de recepción (RX). <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param RX_Address Dirección de recepción (RX). La dirección deberá ser igual a 5 Bytes.
*/

 public void setInitialRxAddress(String RX_Address)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (RX_Address.length() != 5)
        {
        JOptionPane.showMessageDialog(panelerror, "La Dirección de TX debe ser igual a 5 Bytes.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        dev.open(1, 0, -1);
```

```
        int i;

        byte[] data= new byte [5];
        for (i=0;i<5;i++)
        {
        data[i] = (byte) RX_Address.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, data, data.length, 2000, false);



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_RX_Address, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

/**
* Configura el valor inicial del canal de frecuencia. <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param RF_Channel Dirección de radiofrecuencia, valores válidos de 0 a 127 (2400Mhz - 2527MHz).
*/

 public void setInitialRFChannel(int RF_Channel)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (RF_Channel <0 || RF_Channel >127)
        {
        JOptionPane.showMessageDialog(panelerror, "El canal de RF debe estar entre 0 y 127.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        RF_Channel &= 0xFF;

        dev.open(1, 0, -1);

        byte[] data= new byte [1];

        wValue = RF_Channel;
```

```
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_RF_Channel, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor inicial del transceptor (RX o TX). <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param RF_Initial_Mode Modo inicial de RF para el transceptor nRF24L01+. <br>
* True: El dispositivo iniciara como transmisor (TX).
* False: El dispositivo iniciara como receptor (RX).
*/

 public void setInitialRFMode(boolean RF_Initial_Mode)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        if (RF_Initial_Mode == false)
        {
        wValue = 0;
        }

        if (RF_Initial_Mode == true)
        {
        wValue = 1;
        }

        dev.open(1, 0, -1);

        byte[] data= new byte [1];



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_RF_Mode, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
```

```
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor inicial de longitud de datos fija para transmitir. <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param TX_Length Cantidad de Bytes fijos que contendrá cada mensaje de transmisión (TX). Valores
válidos de 1 a 32.
*/

 public void setInitialTXLength(int TX_Length)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (TX_Length <1 || TX_Length >32)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de datos para transmitir debe ser de
1 a 32 Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        TX_Length &= 0xFF;

        dev.open(1, 0, -1);

        byte[] data= new byte [1];

        wValue = TX_Length;

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_TX_Length, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor inicial de longitud de datos fija para recibir. <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
```

```
* @param RX_Length Cantidad de Bytes fijos que contendrá cada mensaje de recepción (RX). Valores
válidos de 1 a 32.
*/

 public void setInitialRXLength(int RX_Length)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (RX_Length <1 || RX_Length >32)
        {
        JOptionPane.showMessageDialog(panelerror, "La longitud de datos para recibir debe ser de 1 a
32 Bytes.", "Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        RX_Length &= 0xFF;

        dev.open(1, 0, -1);

        byte[] data= new byte [1];

        wValue = RX_Length;

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_RX_Length, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor inicial de la velocidad de transmisión de datos de radiofrecuencia. <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param RF_Speed Velocidad de radiofrecuencia. Valores válidos: <br>
* -{@link #RF_Speed_2Mbps RF_Speed_2Mbps} - 1 <br>
* -{@link #RF_Speed_1Mbps RF_Speed_1Mbps} - 2 <br>
* -{@link #RF_Speed_250Kbps RF_Speed_250Kbps} - 3
*/

 public void setInitialRFSpeed(int RF_Speed)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (RF_Speed <1 || RF_Speed >3)
        {
        JOptionPane.showMessageDialog(panelerror, "La velocidad de RF debe ser entre 1 y 3.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
```

```
        }


        dev.open(1, 0, -1);

        byte[] data= new byte [1];

        wValue = RF_Speed;

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_RF_Speed, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

/**
* Configura el valor inicial de la función longitud de datos dinámica (DPL). <br>
* Éste valor se guarda en la EEPROM y sólo tendrá efecto al desconectar y reconectar el dispositivo.
* @param DPL_State Estado inicial del DPL. <br>
* True: La función DPL estará habilitada desde que el dispositivo se conecta.    <br>
* False: La función DPL no estará activada cuando el dispositivo se conecte.
*/

 public void setInitialDPLState(boolean DPL_State)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (DPL_State == false)
        {
        wValue = 0;
        }

        if (DPL_State == true)
        {
        wValue = 1;
        }


        dev.open(1, 0, -1);

        byte[] data= new byte [1];



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_Initial_DPL_State, wValue, wIndex, data, data.length, 2000, false);
```

```
        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

 /**
* Configura el valor de dirección asignado para transmitir (TX).
* @param TX_Address Dirección de transmisión (TX). La dirección deberá ser igual a 5 Bytes.
*/

 public void setTXAddress(String TX_Address)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (TX_Address.length() != 5)
        {
        JOptionPane.showMessageDialog(panelerror, "La Dirección de TX debe ser igual a 5 Bytes.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        dev.open(1, 0, -1);
        int i;

        byte[] data= new byte [5];
        for (i=0;i<5;i++)
        {
        data[i] = (byte) TX_Address.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, data, data.length, 2000, false);



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_TX_Address, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
```

```
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }



}

/**
* Configura el valor de dirección asignado para recibir (RX).
* @param RX_Address Dirección de recepción (TX). La dirección deberá ser igual a 5 Bytes.
*/

        public void setRXAddress(String RX_Address)
{

        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        if (RX_Address.length() != 5)
        {
        JOptionPane.showMessageDialog(panelerror, "La Dirección de RX debe ser igual a 5 Bytes.",
"Error", JOptionPane.ERROR_MESSAGE);
        return;
        }

        dev.open(1, 0, -1);
        int i;

        byte[] data= new byte [5];
        for (i=0;i<5;i++)
        {
        data[i] = (byte) RX_Address.charAt(i);
        }

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, data, data.length, 2000, false);



        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_RX_Address, wValue, wIndex, data, data.length, 2000, false);



        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }
```

```
}

 /**
* Configura el valor de longitud de datos fija para transmitir (TX). Éste valor sólo es válido si
* no se tiene habilitada la función de datos de longitud dinámica (DPL).
* @param TX_Len Cantidad de Bytes fijos que contendrá cada mensaje de transmisión (TX). Valores
válidos de 1 a 32.
*/

 public int setTXLength(int TX_Len)
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        TX_Len &= 0xFFFF;
        if (TX_Len<1 || TX_Len>32)
        {
        JOptionPane.showMessageDialog(panelerror, "Seleccione una valor entre 1 y 32", "Error",
JOptionPane.ERROR_MESSAGE);
        return 0;
        }

        dev.open(1, 0, -1);

        wValue=TX_Len;
        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_TX_Length, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }



}

/**
* Configura el valor de longitud de datos fija para recibir (RX). Éste valor sólo es válido si
* no se tiene habilitada la función de datos de longitud dinámica (DPL).
* @param RX_Len Cantidad de Bytes fijos que contendrá cada mensaje de recepción(RX). Valores válidos
de 1 a 32.
*/

        public int setRXLength(int RX_Len)
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        RX_Len &= 0xFFFF;
        if (RX_Len<1 || RX_Len>32)
        {
        JOptionPane.showMessageDialog(panelerror, "Seleccione una valor entre 1 y 32", "Error",
JOptionPane.ERROR_MESSAGE);
        return 0;
        }

        dev.open(1, 0, -1);
```

```
        wValue=RX_Len;
        byte[] data= new byte [1];

        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_RX_Length, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return (int) data[0];

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return 0;
        }



}

/**
* Configura el valor del transceptor como transmisor (TX).
*/

        private void setTXMode()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);


        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_TX_Mode, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return ;
        }



}

/**
* Configura el valor del transceptor.
* @param Modo Representa el modo del transceptor. <br>
```

```
*       True: Transmisor (TX). <br>
*       False: Receptor (RX).
*
*/

        public void setRFMode(boolean Modo)
        {
        if (Modo == false)
        setRXMode();

        if (Modo == true)
        setTXMode();
        }


/**
* Configura el valor del transceptor como receptor (RX).
*/

        private void setRXMode()
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {


        dev.open(1, 0, -1);


        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Set_RX_Mode, wValue, wIndex, data, data.length, 2000, false);

        dev.close();



        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return ;
        }



}

/**
* Habilita la función de datos de longitud dinámica (DPL).
*/

        public void enableDPL()
        {
        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Enable_DPL, wValue, wIndex, data, data.length, 2000, false);

        dev.close();
```

```
        DPL_State = true;

        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }
        }

/**
* Deshabilita la función de datos de longitud dinámica (DPL).
*/

        public void disableDPL()
        {
        Device dev = USB.getDevice(idVendor, idProduct);
        try {
        dev.open(1, 0, -1);

        byte[] data= new byte [1];
        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO
_HOST,
        bRequest_Disable_DPL, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        DPL_State = false;

        return;

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return;
        }
        }


/**
* Lee los datos de longitud dinámica (DPL) recibidos por RF.
* @param Received_Data Buffer que almacenará los datos en caso de que se hayan recibido.
* @return boolean.
* <br> True - Se recibieron datos.
* <br> False - No se recibieron datos.
*/

        public boolean readDynamicRFData(String Received_Data[])
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] data= new byte [32];
        wValue = 0;

        int data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
```

```
        bRequest_Read_Dynamic_Data_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        if (data_length > 0)
        {
        char[] chrs = new char [data_length];
        int i;

        for (i=0;i<data_length;i++)
        {
        chrs[i] = (char) data[i];
        }

        Received_Data[0] = new String (chrs);
        return true;
        }

        else
        {
        return false;
        }

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}


/**
* Lee los datos de longitud dinámica (DPL) recibidos por RF.
* @param Received_Data Buffer que almacenará los datos en caso de que se hayan recibido.
* @return boolean.
* <br> True - Se recibieron datos.
* <br> False - No se recibieron datos.
*/

        public boolean readDynamicRFData(int Received_Data[])
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        byte[] data= new byte [32];
        wValue = 0;

        int data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Read_Dynamic_Data_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        if (data_length > 0)
        {
        char[] chrs = new char [data_length];
        int i;

        for (i=0;i<data_length;i++)
        {
        Received_Data[i] = (int) data[i];
        Received_Data[i] &= 0xFF;
        }
```

```
        return true;
        }

        else
        {
        return false;
        }

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}


/**
* Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.
* @param Received_Data Buffer que almacenará los datos en caso de que se hayan recibido.
* @param ACK_Data       Datos a enviar a través del acuse de recibo. La longitud no debe exceder los
32 Bytes. <br><br>
* Nota: Los datos en acuse de recibo se enviarán al siguiente mensaje recibido
* solamente.
* @return boolean.
* <br> True - Se recibieron datos.
* <br> False - No se recibieron datos.
*/

        public boolean readDynamicRFData(String Received_Data[], String ACK_Data)
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        if(ACK_Data.length() > 32)
        {
        JOptionPane.showMessageDialog(panelerror, "Los datos en acuse de recibo (ACK) no pueden
exceder los 32 Bytes.", "Error 404", JOptionPane.ERROR_MESSAGE);
        return false;
        }

        byte[] ack_data= new byte [ACK_Data.length()];


        int i;


        for (i=0;i<ACK_Data.length();i++)
        {
        ack_data[i] = (byte) ACK_Data.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, ack_data, ack_data.length, 2000, false);



        byte[] data= new byte [32];

        wValue = ACK_Data.length();
```

```
        int data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Read_Dynamic_Data_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        if (data_length > 0)
        {
        char[] chrs = new char [data_length];


        for (i=0;i<data_length;i++)
        {
        chrs[i] = (char) data[i];
        }

        Received_Data[0] = new String (chrs);
        return true;
        }

        else
        {
        return false;
        }

        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}


/**
* Lee los datos de longitud dinámica (DPL) recibidos por RF y carga datos al acuse de recibo.
* @param Received_Data Buffer que almacenará los datos en caso de que se hayan recibido.
* @param ACK_Data       Datos a enviar a través del acuse de recibo. La longitud no debe exceder los
32 Bytes. <br><br>
* Nota: Los datos en acuse de recibo se enviarán al siguiente mensaje recibido
* solamente.
* @return boolean.
* <br> True - Se recibieron datos.
* <br> False - No se recibieron datos.
*/

        public boolean readDynamicRFData(int Received_Data[], String ACK_Data)
{
        Device dev = USB.getDevice(idVendor, idProduct);
        try {

        dev.open(1, 0, -1);

        if(ACK_Data.length() > 32)
        {
        JOptionPane.showMessageDialog(panelerror, "Los datos en acuse de recibo (ACK) no pueden
exceder los 32 Bytes.", "Error 404", JOptionPane.ERROR_MESSAGE);
        return false;
        }

        byte[] ack_data= new byte [ACK_Data.length()];


        int i;
```

```
        for (i=0;i<ACK_Data.length();i++)
        {
        ack_data[i] = (byte) ACK_Data.charAt(i);
        }


        dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_HOST_TO_D
EVICE,
        bRequest_Send_Data_USB, wValue, wIndex, ack_data, ack_data.length, 2000, false);



        byte[] data= new byte [32];

        wValue = ACK_Data.length();

        int data_length =
dev.controlMsg(USB.REQ_TYPE_TYPE_VENDOR|USB.REQ_TYPE_RECIP_DEVICE|USB.REQ_TYPE_DIR_DEVICE_TO_HOST,
        bRequest_Read_Dynamic_Data_RF, wValue, wIndex, data, data.length, 2000, false);

        dev.close();

        if (data_length > 0)
        {
        char[] chrs = new char [data_length];


        for (i=0;i<data_length;i++)
        {
        Received_Data[i] = (int) data[i];
        Received_Data[i] &= 0xFF;
        }

        return true;
        }

        else
        {
        return false;
        }
        }
        catch (USBException e)
        {
        JOptionPane.showMessageDialog(panelerror, "No se encontró el dispositivo USB.", "Error 404",
JOptionPane.ERROR_MESSAGE);
        System.exit(0);
        return false;
        }
}


}
```

Bibliografía

[1]	Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips., «Universal Serial Bus Specification Rev. 2.0,» Abril 2007. [En línea]. Available: http://www.usb.org/developers/docs/usb20_docs/.
[2]	J. Axelson, USB Complete. The Developer's Guide., Madison, WI: Lakeview Research LLC, 2009.
[3]	ATMEL Corporation, «ATMEGA 8 Datasheet,» nº Rev. 8159D, Febrero 2011.
[4]	Oracle, «Conozca más sobre la tecnología Java,» [En línea]. Available: http://www.java.com/es/about/. [Último acceso: Noviembre 2013].
[5]	Nordic Semiconductor, «nRF24L01+ Specifications,» Mayo 2013. [En línea]. Available: http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01P.
[6]	Nordic Semiconductor, «nRF24L01+ Datasheet,» [En línea]. Available: http://www.nordicsemi.com/eng/content/download/2726/34069/file/nRF24L01P_Product_Specification_1_0.pdf. [Último acceso: Noviembre 2013].
[7]	C. Starkjohann, «V-USB, virtual USB port for Atmel's AVR microcontrollers,» Objective Development Software GmbH, [En línea]. Available: http://www.obdev.at/products/vusb/index.html. [Último acceso: Febrero 2013].
[8]	LibUsb, [En línea]. Available: http://www.libusb.org/. [Último acceso: Marzo 2013].
[9]	LibUSBWin32, [En línea]. Available: http://sourceforge.net/apps/trac/libusb-win32/. [Último acceso: Marzo 2013].
[10]	Java libusb / libusb-win32 wrapper, Febrero 2014. [En línea]. Available: http://libusbjava.sourceforge.net/wp/.
[11]	Oracle, «NetBeans IDE,» Febrero 2014. [En línea]. Available: https://netbeans.org/.
[12]	CadSoft, «EAGLE PCB Software,» Marzo 2014. [En línea]. Available: http://www.cadsoftusa.com/eagle-pcb-design-software/.
[13]	Atmel, «Arduino - Introduction,» [En línea]. Available: http://arduino.cc/en/Guide/Introduction. [Último acceso: Marzo 2014].