UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

TESIS

DISEÑO DE UN SISTEMA DE COMUNICACIÓN MAESTRO-ESCLAVO RS-485, CON APLICACIÓN EN LABVIEW

PRESENTADA POR:

ANTONIO GARCÍA SANTIAGO

PARA OBTENER EL TÍTULO DE:

INGENIERO ELÉCTRICO ELECTRÓNICO

DIRECTOR DE TESIS:

M. en I. JORGE ISRAEL CRUZ MORALES

CIUDAD UNIVERSITARIA, 2013

JURADO ASIGNADO:

Presidente: ING. GLORIA MATA HERNADEZ

Vocal: M.I. JORGE ISRAEL CRUZ MORALES

Secretario: M.I. RAUL RUVALCABA MORALES

1er Suplente: ING. MOISES EUGENIO RUEDA GUTIERREZ

2o Suplente: FÍS. SERGIO ROBERTO ARZAMENDI PEREZ

A mis padres

y

a mis hermanos

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

INTRODUCCIÓN

En la presente tesis con el título “Diseño de un sistema de comunicación maestro-esclavo RS-485, con aplicación en LabVIEW”, se desarrolló un protocolo de comunicación genérico y adaptable a la mayoría de los dispositivos esclavos que se conectarán al dispositivo maestro, esto con el fin de que la comunicación se realice de manera eficiente y se detecten errores en la transmisión de datos. La implementación del protocolo de este sistema es adaptable a cualquier microcontrolador que contenga un módulo USART, en este caso se implementó con microcontroladores de Microchip.

La interfaz entre el dispositivo maestro y los dispositivos esclavos se realizó con una comunicación RS485, con ello se resolvió el problema de la distancia entre el dispositivo maestro y los dispositivos esclavos, y también la comunicación multipunto.

El diseño realizado se puede extender a 32 dispositivos esclavos, y manipularlos desde una computadora personal con una aplicación desarrollada en LabVIEW.

Planteamiento del problema

En el Instituto de Física de la UNAM se necesita contar con un sistema de comunicación que permita interactuar con diversos elementos tales como equipos de medición, sensores, actuadores, motores, entre otros dispositivos.

Se tiene que en estas aplicaciones la distancia es el principal problema, porque muchos de los dispositivos indicados anteriormente no cuentan con una comunicación, o si la tienen, ésta es del tipo RS-232 o USB, y dado que estos tipos de comunicación no son las más idóneas en determinados laboratorios del Instituto; tal es el caso, del laboratorio Acelerador tipo Van de Graaff de 5.5 MeV, donde el panel de control del acelerador se encuentra en una sala independiente a las líneas de trabajo, ahí es donde se localizan los dispositivos de monitoreo de los procesos que se ejecutan. Cabe destacar que este tipo de acelerador tiene haces de iones positivos $^{3}He^{+}$, $^{3}He^{++}$, $^{4}He^{+}$, $^{4}He^{++}$, Deuterones y Protones. Es importante saber que cuando se trabaja con Protones se produce radiación que es peligrosa para el ser humano, por eso surge la importante necesidad de diseñar e implementar el sistema de comunicación, para monitorearlos desde la sala de control.

Por lo anteriormente mencionado, la presente tesis tiene como objetivo realizar un sistema de comunicación bidireccional capaz de manejar un mínimo de 4 dispositivos esclavos a través de un dispositivo maestro, para ello se desarrollará un protocolo de comunicación adaptable al sistema, considerando que la distancia entre el dispositivo maestro y los dispositivos esclavos debe ser mayor a veinte metros.

Las características del sistema de comunicación son las siguientes:

- El dispositivo maestro puede establecer comunicación con varios dispositivos esclavos, además de que también se puede comunicar con una PC.

INTRODUCCIÓN

- La información que se envía a los dispositivos esclavos puede provenir de dos fuentes: de las entradas (botones) que posea el dispositivo maestro o de la PC a través del software LabVIEW.
- La información que se envía a los dispositivos esclavos tiene como objetivo obtener información de los sensores o manipular el estado de los dispositivos actuadores que se encuentran adjuntos en cada uno de ellos.
- La información que es solicitada a los dispositivos esclavos por el dispositivo maestro se visualizará en un display LCD o en la PC mediante el uso de LabVIEW.
- La razón de establecer comunicación con la PC es para ampliar la visualización gráfica de la información que se envía y recibe de los dispositivos esclavos, cuando se tengan varios elementos esclavos adjuntos, así, como para incluir la capacidad de almacenamiento y manipulación de datos del sistema. En la figura 1 se muestra el esquema general del proyecto.
- El sistema también cuenta con la capacidad de detectar errores en la transmisión de datos digitales.

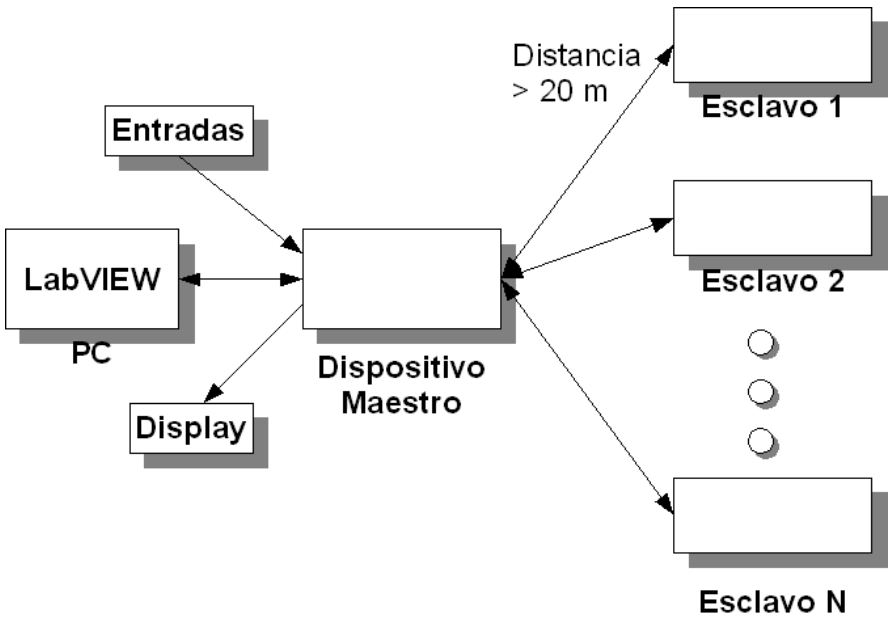


Figura 1: Esquema general del sistema de comunicación.

Para desarrollar este sistema, en el capítulo 1 se dará una revisión sobre los conceptos básicos que se utilizarán a lo largo del proyecto, así como una descripción de varias interfaces de comunicación serial, para justificar el uso de la interfaz RS-232 entre la PC y el dispositivo maestro así como RS-232 y RS-485 entre el dispositivo maestro y los dispositivos esclavos.

INTRODUCCIÓN

En el capítulo 2 se hablará sobre los microcontroladores, que son los elementos básicos de los que se componen el dispositivo maestro y los esclavos.

En el capítulo 3 se desarrollará el proyecto como tal, definiendo el protocolo a usar, el diagrama de bloques de la arquitectura, tanto del dispositivo maestro como de los esclavos, así como los diagramas de flujo de la programación de los microcontroladores que conforman al sistema.

En el capítulo 4 se dará una revisión sobre el entorno de programación LabVIEW y se explicará cómo es la forma en que se establece la comunicación con el dispositivo maestro. Y por último; en el capítulo 5, se presentarán las pruebas y resultados del proyecto.

1. CONCEPTOS GENERALES

Siempre ha existido la necesidad de transmitir información, primero entre las personas mediante señas, después mediante sonidos guturales, comunicándose siempre con las personas que se encontraban alrededor, pero después nació la necesidad de poder transmitir información hacia personas que se encontraran muy alejadas de la persona que emitía el mensaje, para esto se inventaron las telecomunicaciones, entonces se tuvieron que desarrollar sistemas que pudieran procesar las señales de información de manera adecuada. Y siguiendo este camino los sistemas de comunicación que se implementaron también tuvieron que desarrollar una forma de comunicación hasta llegar al punto en donde no sólo los seres humanos han necesitado comunicarse, sino también los sistemas que el mismo ha desarrollado.

Así, los sistemas de comunicación comparten información con dispositivos que se pueden encontrar a milímetros de distancia sobre la misma tarjeta del sistema o a varios cientos o miles de kilómetros de ellos. Estos sistemas pueden procesar tanto señales analógicas como digitales. En este trabajo el sistema de comunicación será digital ya que procesará solamente señales digitales.

Un sistema de comunicación se compone de un transmisor, un receptor y un medio de comunicación como el que se muestra en la figura 2. El medio a través del cual se lleva a cabo la transmisión de información puede ser la atmósfera utilizando señales de radiofrecuencia, los alambres conductores mediante el uso de señales de voltaje o corriente, o la fibra óptica mediante el uso de señales luminosas. En el sistema de la figura 2 la estación 1 solamente puede transmitir información a la estación 2, por lo que el flujo de información solamente ocurre en una dirección, pero a veces es necesario que tanto la estación 1 como la estación 2 puedan transmitir y recibir información. A continuación se dará una clasificación de acuerdo a esta característica.

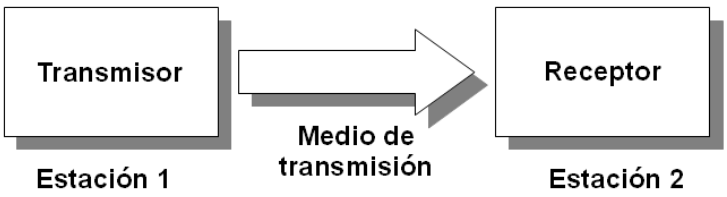


Figura 2: Sistema de comunicación

1.1 Modos de Transmisión

Los sistemas de comunicación pueden clasificarse de acuerdo a su modo de transmisión, lo cual implica tomar en cuenta, el número de líneas de transmisión, la dirección en que se da el flujo de la información y los instantes en que este flujo ocurre. Entonces tenemos tres modos posibles: *simplex, half duplex, full duplex* [1].

SIMPLEX

En este modo sólo hay una línea de comunicación por lo que las transmisiones solamente se hacen en una dirección. Una estación solamente puede ser transmisora o receptora pero no ambos. Como se muestra la figura 2, en donde sólo hay flujo de información de la estación 1 a la estación 2.

HALF DUPLEX

En este modo también hay una sola línea para la comunicación, pero a diferencia de la simplex aquí se pueden realizar transmisiones de información en ambas direcciones, pero no al mismo tiempo, como se muestra en la figura 3.

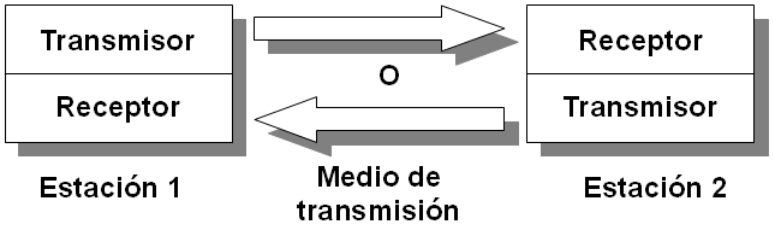


Figura 3: Sistema Half Duplex

FULL DUPLEX

En este modo existen dos líneas de comunicación, una que se ocupa para transmitir y otra para recibir información, teniendo la capacidad de poder realizar las dos operaciones simultáneamente como lo muestra la figura 4.

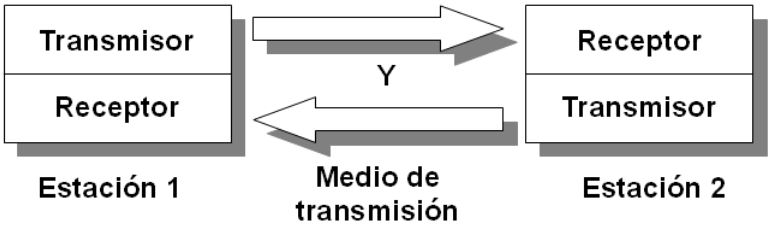


Figura 4: Sistema Full Duplex

1.2 La información

Los elementos mediante los cuales se construye la información son los llamados datos. Estos datos están representados por símbolos los cuales forman un código único, el cual es compartido tanto por el transmisor como por el receptor. En el caso de la información digital, sus elementos constructivos son los datos binarios, los cuales están representados por señales que solamente pueden tomar dos estados '0' y '1', a los que se les ha llamado bits (de *binary digits*). Así, mediante cantidades binarias formadas a partir de estos dos estados, se pueden codificar gran cantidad de información como lo es el código alfanumérico ASCII.

1.2.1 Código ASCII

Debido a que toda la información que es manejada por un sistema digital está representada por '0´s' y '1´s', a cada cantidad numérica representada por estos dos símbolos se le pueden asignar letras y caracteres para que un ser humano pueda interactuar de una forma sencilla con un sistema digital.

En 1960 se estableció el estándar llamado ASCII (*American Standar Code for Information Interchange*), el cual asigna una cantidad numérica binaria para las letras del alfabeto, los números del 0 al 9, y muchos códigos de control y signos de puntuación [2]. La gran ventaja de este código es que al ser implementado en diversos dispositivos, se puede intercambiar información de manera simple sin tener que hacer conversiones entre distintos códigos. Los sistemas ASCII utilizan una cadena de 7 bits para representar cada código. Aunque, frecuentemente se agrega un cero en la posición más significativa para hacer un código de 8 bits, que definen un byte. Por ejemplo, para la letra “A” el número binario asignado es el '0100 0001'.

1.3 Transmisión digital: paralela y serial

Dentro de los sistemas digitales hay dos formas mediante las que se puede recibir o transmitir información que son: paralela y serial. Por ejemplo para transmitir el numero binario '1001' de un sistema A a un sistema B en forma paralela, se necesitan de 4 líneas para que la recepción ocurra de manera simultánea. En cambio en una transmisión serial sólo se necesita de una línea que vaya del sistema A al sistema B, pero la transmisión ocurre bit a bit utilizando intervalos regulares. En este trabajo emplearemos estos dos tipos de transmisión.

1.4 Elementos de un sistema de comunicación robusto

Para que un sistema de comunicación trabaje de manera eficiente se deben de considerar los siguientes puntos [1]:

- La forma de las señales y la magnitud de las mismas.
- El tipo del enlace de la comunicación (par trenzado, coaxial, fibra óptica, radiofrecuencia, etc.)
- El arreglo de las señales para formar los caracteres del código, a partir del cual los mensajes pueden ser construidos.
- Un protocolo de comunicación.

La forma de las señales se refiere a los niveles de voltajes utilizados para la representación de la lógica binaria. En este trabajo se utilizarán señales TTL y RS-232. En el caso de la lógica TTL un voltaje que se encuentre dentro del rango que va de 0 a 0.4 [V] es un '0' lógico y un voltaje dentro del rango de 2.4 a 5 [V] es un '1' lógico. Los valores lógicos de las señales manejadas por el protocolo RS-232 se proporcionarán más adelante.

En lo que se refiere al arreglo de las señales para formar el código, serán utilizados algunos caracteres del código ASCII.

1.5 Protocolo

Un protocolo, definido como un conjunto de reglas, es esencial para la definición del formato de los mensajes comunes y procedimientos para la transferencia de datos entre todos los dispositivos de una red de comunicación. Este protocolo debe incluir las siguientes características [1]:

- Inicialización .- Se inicializan los parámetros del protocolo y comienza la comunicación.
- Bloques de datos y sincronización.- Definen el inicio y fin de un bloque de datos, así como las características de los bloques de datos y la forma en que el receptor puede sincronizarse con el flujo de datos.
- Control de flujo.- Asegura que el receptor este habilitado para informarle al transmisor la regulación del flujo de datos, para asegurar que no haya pérdida de datos.
- Control de línea.- Utilizado con enlaces *half duplex* para revertir el rol del transmisor y del receptor para continuar con la transmisión en otra dirección.
- Control de error.- Provee técnicas para revisar la precisión de los datos recibidos para identificar errores en la transmisión.
- Control del *time out*.- Procedimiento para transmitir, retransmitir o abortar la transmisión de información cuando no haya una señal que indique que la operación se ha llevado a cabo con éxito dentro del límite de tiempo acordado.

1.6 Detección de errores en la transmisión

Todos los canales de comunicación de datos están sujetos a ruido, particularmente donde los equipos están situados en ambientes industriales con ruido eléctrico elevado, tales como radiación electromagnética de los equipos adjuntos o inducción electromagnética de los cables adjuntos. Como consecuencia los datos que llegan al receptor pueden contener errores. Para asegurar que la comunicación sea confiable necesitamos revisar la precisión de cada mensaje.

Entre los métodos que se tienen para detección de errores, el más sencillo es el método de paridad, el cual consiste en agregar un bit a un grupo de código que se transfiere de una localidad a otra que generalmente son mensajes de siete u ocho bits. El bit extra llamado bit de paridad puede ser un '1' o un '0' dependiendo del número de unos que haya en el mensaje [3].

Hay dos tipos de paridad: par e impar. En la paridad par si el mensaje tiene un número de unos impar entonces el bit de paridad debe de ser '1' para que el total de unos sea par, y si el número de unos es par entonces el bit de paridad es un '0'. En la paridad impar por el contrario el bit de paridad es '1' cuando el número de unos del mensaje sea par, y es '0' cuando el número de unos es impar. El bit de paridad generalmente se agrega a la parte más significativa del mensaje.

El bit de paridad se emplea para detectar cualquier error en un sólo bit que ocurra durante la transmisión de un código de una localidad a otra, por ejemplo si se transmite el código ASCII “A” que tiene asignada la codificación binaria '1100 0001' utilizando paridad impar en donde el séptimo bit corresponde al bit de paridad y debido a ruido en el entorno del sistema se recibe '1100 0000', el receptor después de realizar la verificación de paridad determinará que hay un

error ya que tanto el receptor como el transmisor han acordado la paridad impar; puede entonces el receptor solicitar al transmisor que le reenvíe la información. Pero, si hubieran ocurrido dos cambios durante la transmisión, entonces no se habría afectado la condición impar en el número de unos del mensaje, por lo que este método es empleado cuando la probabilidad de que ocurra un error es muy baja y cuando el tamaño de los mensajes es pequeño tales como 7 u 8 bits.

Para mensajes más largos se requieren de cálculos de revisión más complejos para ser más efectivos. Por ejemplo, la LRC (L*ongitudinal Redundancy Check*) calcula un byte adicional cubriendo el contenido de un mensaje de hasta 15 bytes, mientras que una revisión aritmética calcula dos bytes adicionales que pueden ser usados para mensajes de hasta 50 bytes de longitud. El CRC (*Cyclic Redundancy Check*) es otro método que puede detectar errores con mucha precisión en mensajes de longitudes variadas. Por ejemplo, puede detectar un error en paquetes de datos de 36000 bits [1].

1.7 Líneas balanceadas y desbalanceadas

Una línea desbalanceada es aquella que transporta señales por un sólo cable referidas a tierra, como la que se muestra en la figura 5. Las ventajas de este tipo de transmisión son la simplicidad y el bajo costo de su implementación. La principal desventaja es que no es inmune al ruido. Debido a que el cable de tierra forma parte del sistema, un voltaje transitorio puede inducir altas frecuencias o corrientes altas que provoquen la degradación de la señal. Estos problemas limitan la distancia y la velocidad a la que puede operar el sistema con una sola línea.

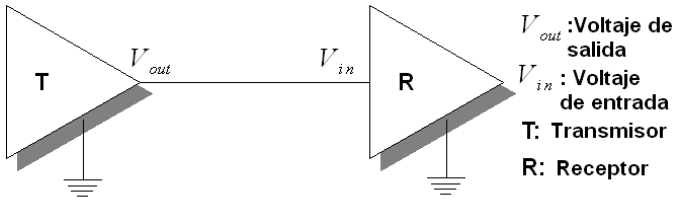


Figura 5: Línea desbalanceada

Una línea balanceada o diferencial como la que se muestra en la figura 6 es aquella que utiliza dos cables para transmitir una sola señal. Estas líneas son utilizadas para transmitir información a grandes distancias ya que tienen una excelente inmunidad al ruido. En estos sistemas el voltaje que mide el receptor es igual a la diferencia de voltajes entre las dos líneas.

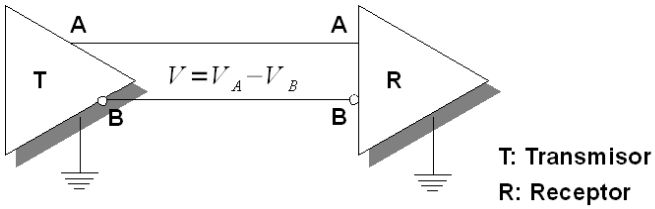


Figura 6: Línea balanceada o diferencial

1.8 Comunicaciones síncronas y asíncronas

En las comunicaciones seriales siempre debe de haber alguna forma de sincronización entre el transmisor y el receptor para que la información transmitida pueda ser interpretada de forma correcta por el receptor.

En aplicaciones en donde los datos se transmiten a cortas distancias, es posible que la señal de reloj acompañe a los datos, lo cual facilita la sincronización. Cuando la señal de reloj acompaña los datos, la comunicación es síncrona.

Para cuando la transmisión de datos se tenga que realizar a distancias muy grandes, transmitir la señal de reloj aumentaría el costo de la transmisión. Así que, el receptor debe de ser capaz de conocer el tiempo de duración de cada bit y el momento en que comienza cada bloque de datos transmitidos (los bloques de datos generalmente están compuestos por bytes) sin la señal de reloj. Entonces, la duración de cada bit se puede sincronizar utilizando la misma frecuencia de reloj tanto en el transmisor como en el receptor y el inicio de cada bloque de datos puede indicarse con una marca cada vez que se transmita un byte o un conjunto de bytes. Cuando la sincronización se realiza byte a byte, se dice que la comunicación es asíncrona, mientras que, si se realiza por bloques de bytes la comunicación es síncrona [4].

Así, la comunicación síncrona es aquella en la que se transmite la señal de reloj o la sincronización se realiza por bloques de bytes (sin señal de reloj), y la comunicación asíncrona es aquella en donde la sincronización se realiza byte por byte y no se transmite la señal de reloj.

1.9 Topologías de las transmisiones de datos

A continuación se explicarán las topologías punto a punto y multipunto.

PUNTO A PUNTO

En esta configuración únicamente existe un transmisor y un receptor por línea y la transmisión solamente es posible en una dirección (*simplex*), como se muestra en la figura 7.

Figura 7: Topologías punto a punto.

MULTIPUNTO

Esta configuración se puede clasificar en dos tipos: distribución simple y múltiple.

En la distribución simple solamente existe un transmisor y muchos receptores, y la información sólo fluye en una dirección, como se muestra en la figura 8. En la distribución múltiple pueden existir muchos transmisores-receptores, y las transmisiones pueden ser bidireccionales como se observa en la figura 9.

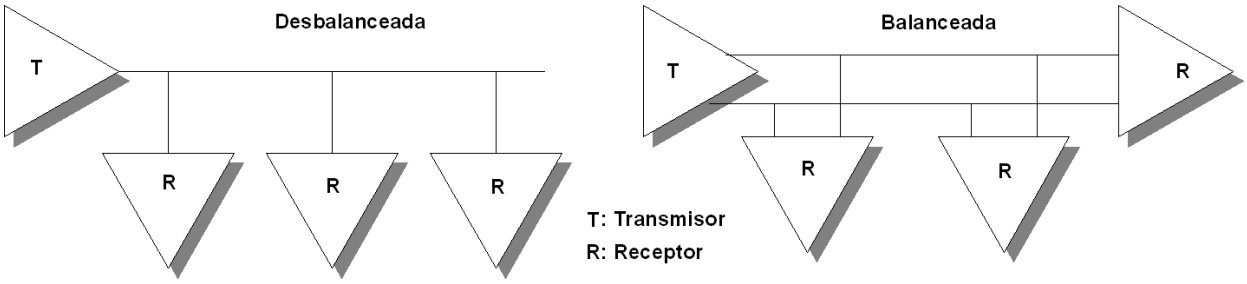


Figura 8: Topología multipunto con distribución simple.

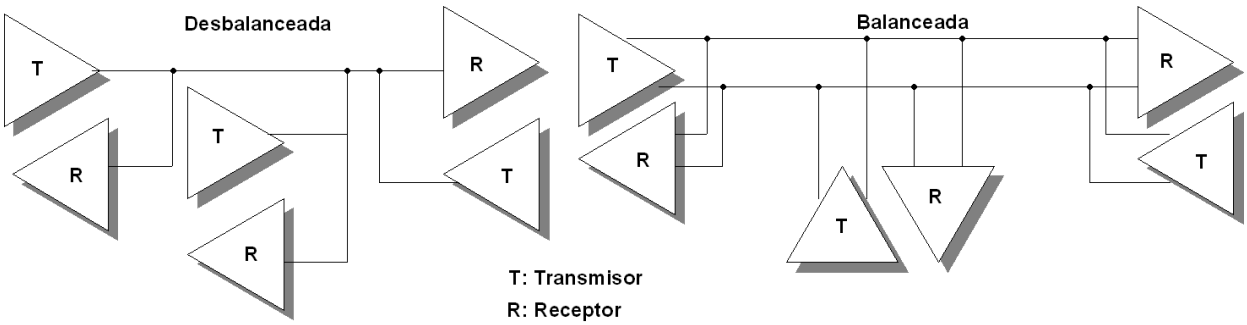


Figura 9: Topología multipunto con distribución múltiple.

1.10 Interfaces de comunicación serial

A continuación se revisarán las interfaces de comunicación serial, RS-232, RS-485, SPI, CAN y Ethernet; para justificar el uso de las dos primeras en este trabajo. Cabe mencionar que, todas las interfaces anteriores pueden implementarse con el uso de microcontroladores ya que estos cuentan con hardware que permiten manejar cada una de las interfaces mencionadas.

1.10.1 SPI

La interfaz SPI (*Serial Peripheral Interface*) desarrollada por Motorola es un protocolo síncrono. Este protocolo es implementado entre microcontroladores o entre un microcontrolador y periféricos que cuenten con un hardware que maneje el protocolo SPI. Los dispositivos periféricos pueden ser memorias EEPROM´s seriales, registros de desplazamiento, convertidores analógico-digitales, etc. Es utilizada cuando pocas líneas de entrada-salida están disponibles en un microcontrolador, pero también cuando la comunicación debe ser rápida y fácil de implementar.

Es un protocolo maestro-esclavo, en donde el dispositivo maestro es el que inicia la comunicación con los dispositivos esclavos. Este *bus* opera en modo *full duplex,* en donde la transferencia y recepción de datos se realiza de manera simultánea. El tamaño de los paquetes de datos que se envían y reciben son de 8 bits, un byte.

La interfaz SPI especifica cuatro señales lógicas como las que se muestran en la figura 10.

SD0.- Salida serial de datos.

SDI.-Entrada serial de datos.

SCK.- Señal de reloj.

SS.- Selector del dispositivo esclavo.

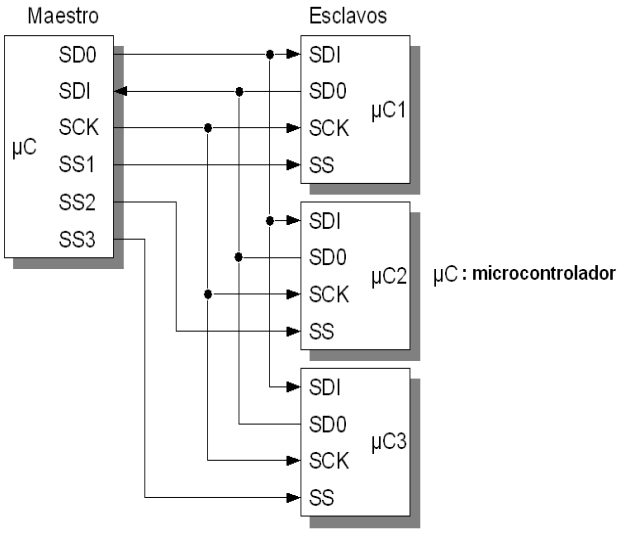


Figura 10: Configuración SPI maestro-esclavos

Cuando las señales SS se encuentran en estado alto ningún dispositivo esclavo se encuentra habilitado para la comunicación, ésta comienza cuando la línea SS de alguno de los esclavos es llevada al estado bajo, la cual siempre es controlada por el dispositivo maestro. Los dispositivos esclavos no seleccionados mantienen a la línea SDO en estado de alta impedancia y por lo tanto inactiva.

La señal de reloj es suministrada por el dispositivo maestro para efectuar la sincronización, esta señal de reloj controla el momento en que un bit puede ser enviado o recibido.

La interfaz SPI emplea un registro de desplazamiento para transmitir la información. Los datos sobre esta interfaz pueden ser transferidos desde una razón de 0 bits/segundo hasta 1 Mbit/segundo.

1.10.2 RS-232

De entre los estándares más populares de comunicación serial que se utilizan hoy en día debido a su simplicidad y poco hardware usado en su implementación se tiene el EIA/TIA-232, el cual fue introducido en 1960. Este estándar ha sido desarrollado por la Asociación de Industrias Electrónicas y la Asociación de Industrias de Telecomunicaciones (EIA/TIA), que popularmente se conoce como RS-232 donde RS significa Estándar Recomendado.

El nombre oficial del estándar EIA/TIA-232 es “Interfaz entre equipos terminales de datos (DTE) y equipos de comunicación de datos (DCE), empleando intercambio de datos seriales binarios”, por lo que este estándar se ocupa de la comunicación serial entre un sistema servidor DTE y un periférico DCE.

Para asegurar que haya compatibilidad entre el sistema servidor y un periférico, el estándar RS-232 especifica:

- Los niveles de las señales y los voltajes que son comunes.
- La configuración del cableado de las terminales comunes.
- La cantidad mínima de información de control entre el sistema servidor y el periférico.

Entonces RS-232 especifica las características eléctricas, funcionales y mecánicas para satisfacer los tres criterios mencionados arriba.

CARACTERÍSTICAS ELÉCTRICAS

En esta sección del estándar se especifican los niveles de voltaje, la velocidad de cambio de los voltajes, y la impedancia de la línea.

Como el estándar original RS-232 fue definido en 1962, antes de que estableciera la lógica TTL, los niveles lógicos de voltaje usados en este estándar no corresponden a 0 y 5 [V]. En vez de esto, un nivel alto de voltaje válido para el transmisor o receptor está definido dentro del rango de +3 a +15 [V], y un nivel bajo de voltaje válido para el transmisor o receptor está definido dentro del rango de -3 a -15 [V]. El área muerta entre -3 y +3 [V] está diseñada para absorber el ruido de la línea. En este estándar, un nivel de voltaje bajo corresponde a un 1 lógico, que históricamente ha sido conocido como una marca y un nivel de voltaje alto corresponde a un 0 lógico, conocido también como un espacio, como se muestra en la figura 11.

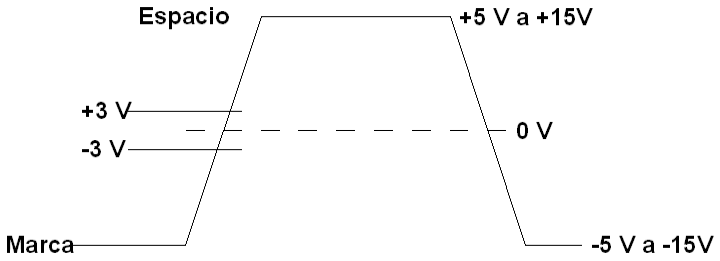


Figura 11: Niveles lógicos RS-232

El máximo *slew rate* (máxima variación en el voltaje de salida por unidad de tiempo) permitido por el estándar es de 30V/μs y la máxima velocidad de transmisión de datos es de 20 kbits/segundo. Estas características permiten reducir la probabilidad de interferencias [5].

La carga vista por el receptor está especificada dentro del rango de 3kΩ a 7kΩ. En la especificación original, la longitud máxima del cable entre el receptor y el transmisor se definió como 15 metros, aunque en las revisiones posteriores del estándar se cambió el parámetro de longitud máxima del cable por una carga de capacitancia máxima que corresponde a 2500 pF, el cual es un parámetro más adecuado. Así, la longitud máxima del cable está determinada por la capacitancia del cable por unidad de longitud, la cual es provista en las especificaciones del cable a utilizar.

CARACTERÍSTICAS FUNCIONALES

El estándar RS-232 también define las funciones de las diferentes señales que son utilizadas en la interfaz. Estas señales se dividen en cuatro diferentes categorías: comunes, datos, control y sincronización. Estas señales son mostradas en la tabla 1 [5].

Nombre de la señal*	Dirección de la señal	Tipo de señal
Signal Common	-	Común
Transmitted Data (TD) Received Data (RD)	Hacia DCE Desde DCE	Datos
Request to Send (RTS) Clear to Send (CTS) Data Set Ready (DSR) Data Terminal Ready (DTR) Ring Indicator (RI) Data Carrier Detect (DCD) Signal Quality Detector Data Signal Rate Detector Data Signal Rate Detector Ready for Receiving Remote Loopback Local Loopback Test Mode	Hacia DCE Desde DCE Desde DCE Hacia DCE Desde DCE Desde DCE Desde DCE Hacia DCE Desde DCE Hacia DCE Hacia DCE Hacia DCE Desde DCE	Control
Transmitter Signal Element Timing Transmitter Signal Element Timing Receiver Signal Element Timing	Hacia DCE Desde DCE Desde DCE	Sincronización
Secondary Transmitted Data Secondary Received Data	Hacia DCE Desde DCE	Datos
Secondary Request to Send Secondary Clear to Send Secondary Data Carrier Detect	Hacia DCE Desde DCE Desde DCE	Control

* Las señales que se encuentran entre paréntesis corresponden a las ocho señales más usadas.

Tabla 1: Definición de las señales RS-232

El número de señales definidas en el estándar RS-232 es muy amplio ya que puede soportar canales de comunicación primarios y secundarios, pero son pocas las aplicaciones que requieren del uso de todas las señales. Por ejemplo, para un módem se utilizan nueve señales, algunas aplicaciones simples requieren solamente de cinco señales, dos de datos, dos de control y la señal de tierra, pero también puede haber aplicaciones en donde sólo se utilicen las señales de datos sin señales de control.

CARACTERÍSTICAS MECÁNICAS

En esta sección se especifica la interfaz mecánica, el cual es un conector de 25 terminales en el cual se pueden acomodar las 25 señales que se definen en la parte funcional del estándar. Pero como no todas las señales son utilizadas, en algunas aplicaciones se utilizan otros conectores diferentes con menor número de terminales, en donde el más popular es el conector DB9; este conector provee terminales para las señales de recepción y transmisión de datos y las señales de control necesarias para las aplicaciones de módem. La asignación de las señales a cada una de las terminales de los conectores se muestra en la figura 12. Estos conectores son tipo hembra y corresponden a los que poseen los dispositivos DCE, para los dispositivos DTE el conector debe ser macho.

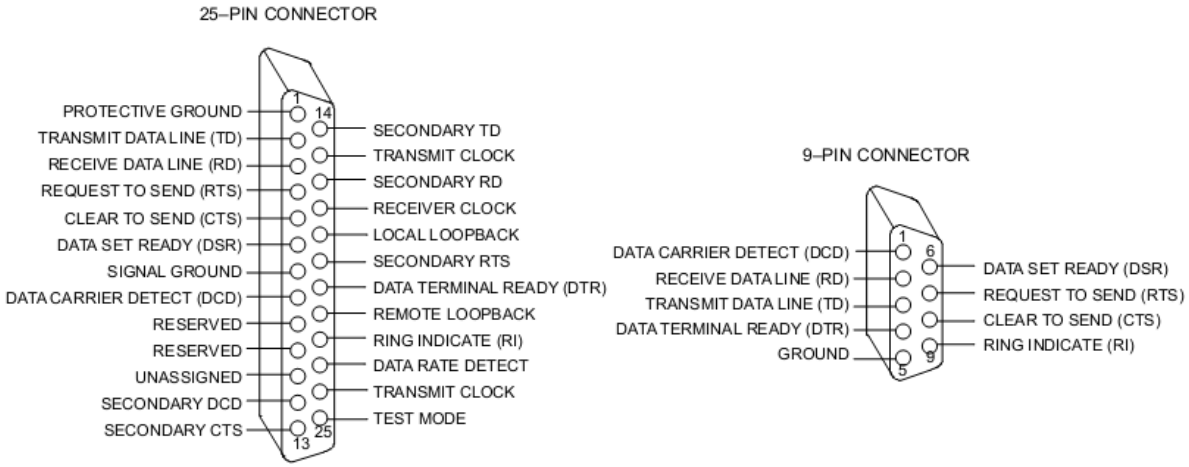


Figura 12: Conectores RS-232

VELOCIDAD DE TRANSFERENCIA DE DATOS

La velocidad de transferencia de los datos en el protocolo RS-232 depende del sistema en donde se maneja este estándar, por ejemplo en una computadora personal la velocidad de transmisión de datos puede variar desde 100 bits/segundo hasta 56 kbits/segundo [2].

PUERTOS COM

En las computadoras personales el protocolo RS-232 es implementado en los puertos COM, los cuales pueden ser accedidos por programas de aplicación tales como LabVIEW, o BASIC. Recientemente estos puertos fueron desplazados por puertos USB, por lo que, en ausencia de estos se pueden utilizar módulos convertidores de USB a COM.

UART

La UART (*Universal Asynchronous Receiver/Transmitter*) es un hardware con el que cuentan los dispositivos con puerto serial como la PC o los microcontroladores. La función de la UART

es convertir de datos paralelos a datos seriales y viceversa, así como manejar los detalles a bajo nivel de las comunicaciones seriales.

IMPLEMENTACIONES PRÁCTICAS RS-232

Muchos de los sistemas diseñados hoy en día no operan usando niveles de voltaje RS-232. Debido a esto los conversores de niveles lógicos son necesarios para implementar una comunicación RS-232. La conversión de niveles se realiza mediante circuitos integrados especiales. Por ejemplo hay circuitos integrados que convierten niveles de voltaje RS-232 a niveles de voltaje TTL tales como el MAX232. Así, se podría realizar una comunicación entre una PC y un microcontrolador manejando el protocolo RS-232 como se muestra en la figura 13, donde la PC es el dispositivo DTE y el microcontrolador el dispositivo DCE.

Figura 13: Implementación de una comunicación RS-232

1.10.3 RS-485

RS-485 es una solución para aplicaciones que requieran grandes distancias o velocidades de transmisión altas, ya que es una línea balanceada que trabaja en modo *half duplex* o *full duplex.*

Características de RS-485:

Bajo costo. Los receptores y transmisores son baratos y sólo requieren de una fuente de 5 V para generar una diferencia mínima de 1.5 V entre sus terminales diferenciales a diferencia del protocolo RS-232 que requiere del uso de voltajes simétricos.

Capacidad de interconexión. En vez de estar limitada a una conexión de sólo dos dispositivos como la RS-232, la RS-485 es una interfaz multipunto que puede manejar múltiples receptores y transmisores. Generalmente esta red puede manejar 32 nodos, pero con una alta impedancia en los receptores se puede tener una red RS-485 que puede manejar hasta 256 nodos.

Longitud de enlace. La longitud máxima del cable para la transmisión de datos es de 1.2 km.

Velocidad. La velocidad de transmisión puede ser tan alta como 10 Mbits/segundo.

La velocidad de transmisión está relacionada con la longitud de transmisión, así para una velocidad de 90 kbits/segundo se tiene que la máxima longitud de transmisión es de 1.2 km, y trabajando a 10 Mbits/segundo la máxima longitud a la que se puede transmitir es de 15 m [6].

En la figura 14 se muestra una red RS-485 multipunto trabajando en modo *half duplex,* con dos hilos en donde se pueden compartir hasta 32 transmisores-receptores.

Los dos estados que se pueden dar en la línea diferencial son los siguientes:

Cuando el voltaje de la terminal “A” del transmisor es menor con respecto al voltaje de la terminal “B”, la línea está en estado binario 0.

Cuando el voltaje de la terminal “A” del transmisor es mayor con respecto al voltaje de la terminal “B”, la línea se encuentra en el estado binario 1.

Como la interfaz observada en la figura 14 está trabajando en modo half duplex, el flujo de datos solamente puede ocurrir en una sola dirección en un instante dado, por esta razón los elementos transmisores-receptores son elementos triestado, que cuentan con una línea de control para seleccionar el modo en el que deben de trabajar durante un instante dado, ya sea como receptores o como transmisores.

Un método para terminar un par de líneas de una red RS-485 es mediante el uso de dos resistores situados en los extremos de la línea diferencial. El valor del resistor debe de coincidir con la impedancia característica de la línea de transmisión, la cual usualmente varía en el rango de 100 a 120 Ω [7].

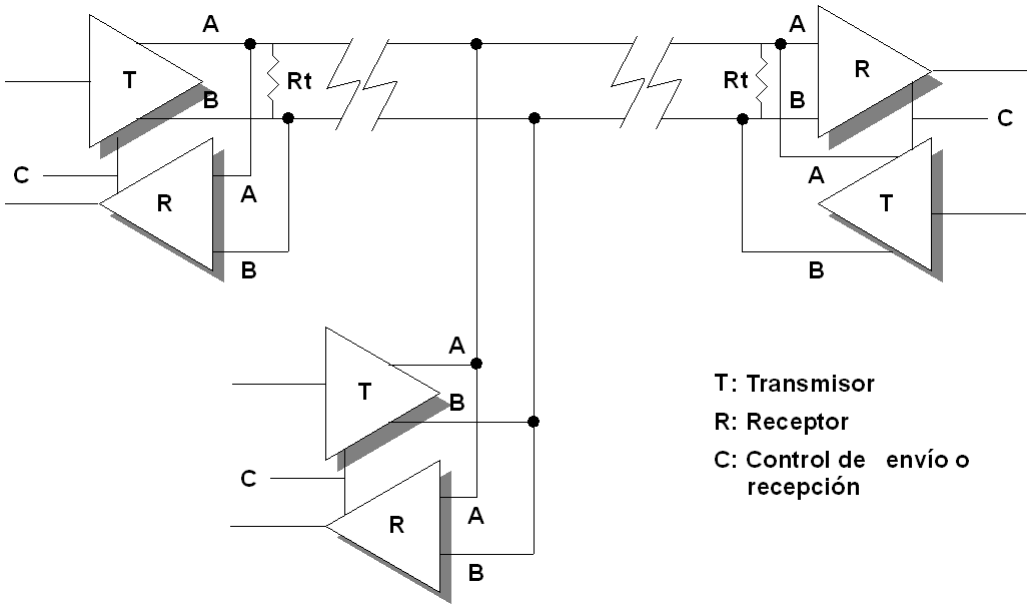


Figura 14: Red RS-485 multipunto half duplex

1.10.4 CAN

El protocolo/interfaz CAN (Controller Area Network) fue desarrollado por BOSCH como un sistema multimaestro, el cual puede trabajar a una velocidad máxima de 1 Mbit/segundo, a una longitud máxima de 40 metros y con un máximo de 30 nodos.

CAN es un sistema que se encuentra conformado por varios nodos, los cuales pueden operar como maestros o esclavos. La principal característica de este *bus* es que cuando un nodo envía

información, el destino de ésta no es un sólo nodo, sino todos los demás nodos del sistema, posteriormente cada nodo decide si el mensaje debe ser aceptado o rechazado.

Originalmente fue desarrollado para industria automotriz para remplazar las conexiones complejas de cables por una sola línea diferencial. Mejoras como, la alta inmunidad a la interferencia eléctrica y la capacidad para autodiagnosticarse y reparar los errores en los datos, permitieron extender el uso de esta interfaz en las industrias médicas y de manufactura.

CAN tiene un protocolo de comunicación serie asíncrono del tipo CSMA/CD (Carrier Sense Multiple Access with Collision Detection). CSMA significa que, antes de que cada nodo intente enviar un mensaje, cada nodo debe permanecer inactivo cierto tiempo, para evaluar el estado de la red (activa o inactiva); de esta manera, solamente puede transmitirse un mensaje cuando la red este inactiva. CD significa que cuando ocurre una colisión entre los datos, ésta es resuelta mediante reglas de prioridad preprogramadas en el campo de identificación de cada mensaje.

En la figura 15 se muestra el diagrama de bloques de una interfaz CAN, en donde se observa la línea diferencial que es compartida por todos los nodos. La impedancia característica de esta línea es del orden de 120 Ω, por lo que se emplean resistencias de este valor en los extremos de las líneas para evitar que haya ondas reflejadas. Cada una de las dos líneas posee un nombre, CANH y CANL respectivamente, las cuales definen un cierto estado dependiendo de los niveles de voltaje que presenten. El estado recesivo se presenta cuando las dos líneas CANH y CANL se encuentran polarizadas a 2.5 [V] y el estado dominante cuando CANH = 3.5 [V] y CANL = 1.5 [V], por lo que hay una diferencia de 2 [V] entre las dos señales.

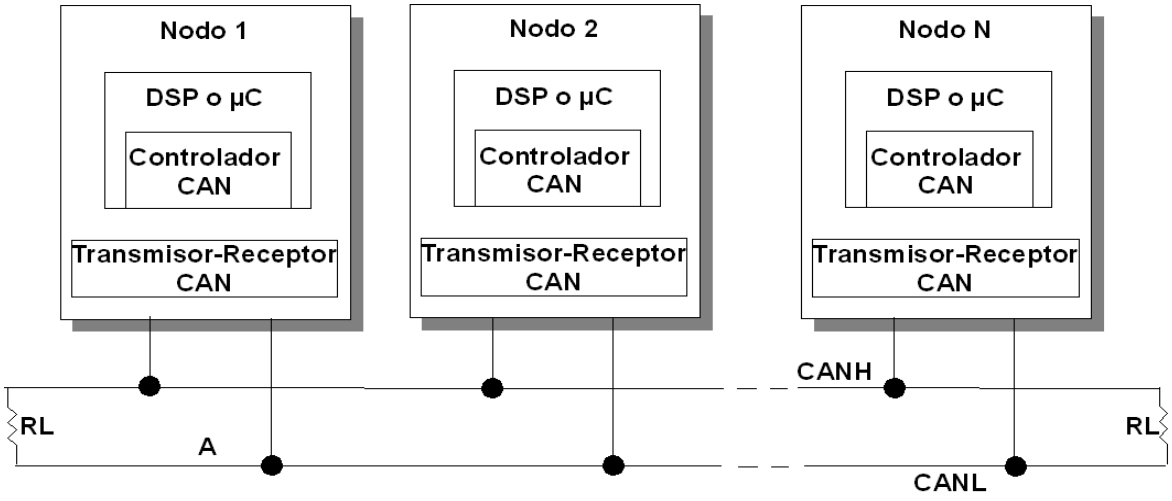


Figura 15: Interfaz CAN

1.10.5 Ethernet

Ethernet es un protocolo/interfaz de comunicación asíncrono CSMA/CD que puede enviar desde 46 hasta 1500 bytes de datos por paquete, con una velocidad de transmisión de cientos de Mbits/segundo.

Ethernet está definido en la especificación IEEE 802.3, en la que se describen la capa física y el enlace de los datos. Ethernet tiene varias presentaciones, que dependen de la combinación entre la velocidad de transmisión, el modo de transmisión y el medio de transmisión físico.

Velocidad máxima de bits (Mbits/segundo): 10, 100, 1000 etc.

Modo de transmisión: Banda ancha, banda base.

Medio de transmisión física: Coaxial, fibra óptica, UTP, etc

Así, la Ethernet 10-Base-T se refiere a un protocolo/interfaz que soporta una velocidad de 10 Mbits/segundo, utiliza cable UTP y trabaja en banda base.

La parte de la especificación que se refiere a la capa física describe la forma en que los bits son transmitidos a través de las conexiones físicas, incluyendo la codificación, multiplexaje, sincronización, etc. La parte referida al enlace de datos describe las características de los bloques de datos transmitidos, incluyendo los paquetes de control de flujo de datos.

Las primeras redes Ethernet fueron implementadas en un medio compartido como el de la figura 16 pero ahora muchas redes son configuradas en una red punto a punto como el de la figura 17 o en una red estrella como el de la figura 18, la cual puede ser una colección de conexiones punto a punto. En estas dos últimas configuraciones, como cada nodo está conectado a máximo un nodo, cada nodo puede operar en modo *full duplex,* de esta manera ya no es necesario usar CSMA/CD, ya que no existen colisiones entre los datos.

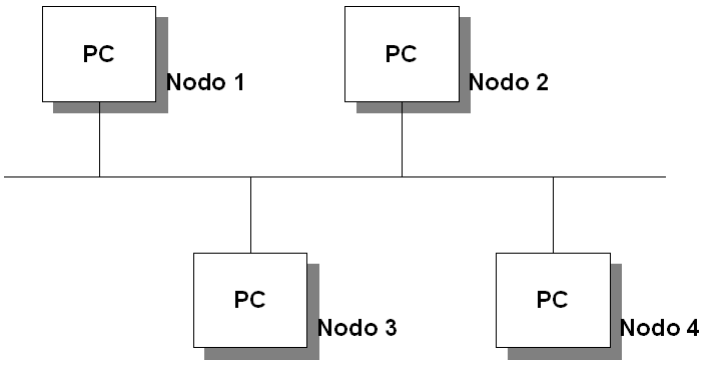


Figura 16: Ethernet en una red compartida

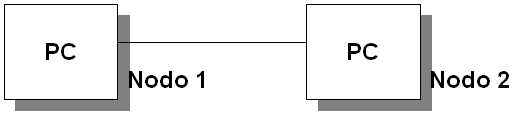


Figura 17: Ethernet en una configuración punto a punto

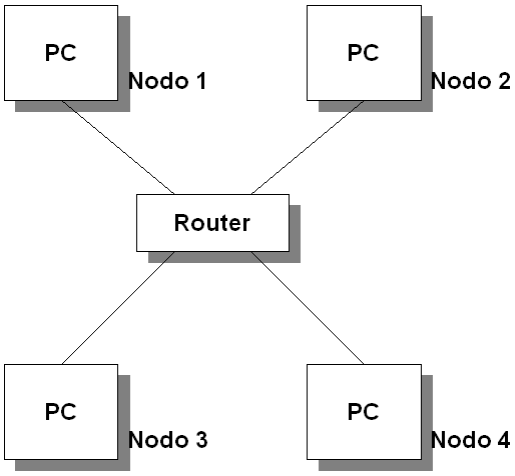


Figura 18: Ethernet en una configuración estrella

2. MICROCONTROLADORES

2.1 Introducción

Los microcontroladores se popularizaron mucho en la industria debido a que el tener concentrados varios elementos en un sólo encapsulado permitió emplearlos en la solución tanto de tareas muy simples como de muy complejas, a un costo reducido y en un tiempo menor, en comparación con un sistema discreto que se puede construir con los mismos elementos de los que se compone el microcontrolador. La diferencia entre un sistema concentrado y un sistema discreto es que este último, está enfocado a realizar tareas mucho más complejas tales como el manejo de un sistema operativo.

Un microcontrolador es un encapsulado integrado que incluye como mínimo, un microprocesador (CPU - Central Processing Unit), una memoria, y puertos de entrada-salida, como se muestra en la figura 19.

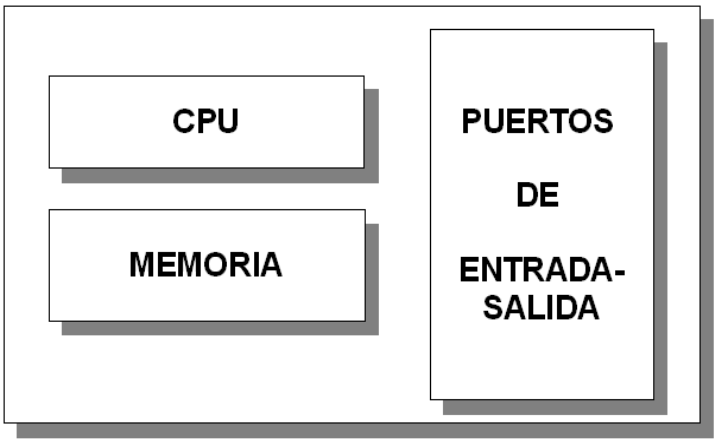


Figura 19: Microprocesador básico

La Unidad Central de Proceso (CPU, por sus siglas en inglés) es el bloque que se encarga de ejecutar una a una las instrucciones que se encuentran almacenadas en la memoria, así como de realizar las operaciones lógicas y aritméticas. Los puertos de entrada-salida son vías mediante las cuales el microcontrolador tiene contacto con el exterior ya que se pueden introducir datos, instrucciones o recibir resultados de sus procesos internos.

En la actualidad los microcontroladores han ido incorporando un mayor número de periféricos en el mismo encapsulado, llegando a conformar un sistema con una mayor funcionalidad. Entre los periféricos que podemos encontrar en los microcontroladores tenemos: temporizadores-contadores, módulos de captura, módulos de comparación, PWM, convertidores analógicos-digitales (ADC), módulos de comunicación serial como USART, SPI, I2C, Ethernet, CAN, USB, memorias EEPROM, etc, como el que se muestra en la figura 20.

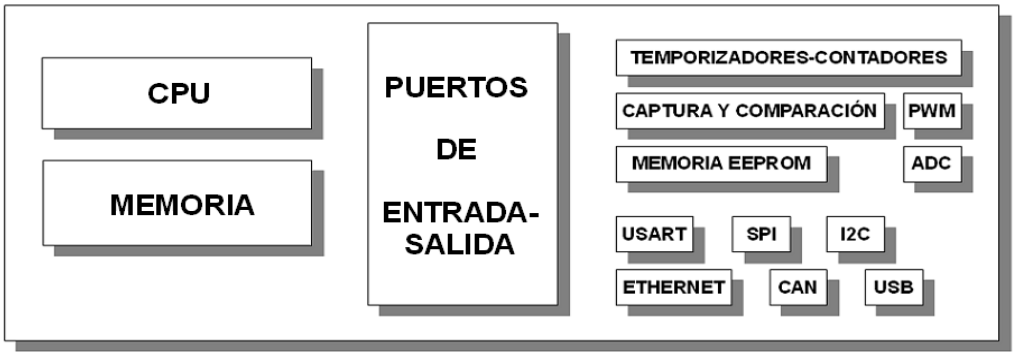


Figura 20: Microprocesador genérico

2.2 Memorias RAM y ROM

En cuanto a la memoria existen dos tipos que son la RAM y la ROM. La memoria RAM (Random Access Memory) es usada para almacenar información temporalmente, es decir, los datos se retienen mientras el sistema se encuentra polarizado y se pierden cuando ya no lo está, por lo que también es llamada memoria volátil. Por otro lado, la memoria ROM (Read Only Memory) se usa para almacenar información permanentemente incluso cuando no hay polarización, debido a esto también se le llama memoria no volátil. Ejemplos de este tipo de memoria son la flash y la EEPROM.

2.3 Buses internos de los microcontroladores

Un bus es un conjunto de cables o líneas mediante los cuales se transporta información de un lugar a otro. En un microcontrolador los *buses* permiten la interacción interna entre el CPU y la memoria, y sus puertos de entrada-salida. Estos *buses* se clasifican en tres tipos: *bus* de direcciones, *bus* de datos, *bus* de control.

2.3.1 Bus de datos

Este *bus* es utilizado para transportar datos entre el CPU y la memoria o sus dispositivos de entrada-salida. Generalmente el tamaño de los *buses* de datos en los CPU´s varía entre 8 y 64 líneas. El tamaño del *bus* de datos indica el número de bits que se pueden procesar simultáneamente. Así la capacidad de procesamiento de un CPU depende de los tamaños del *bus* de datos y del *bus* de direccionamiento, el cual se revisa a continuación.

2.3.2 Bus de direcciones

Para ser reconocidos por el CPU, cada elemento de la memoria o puerto de entrada-salida debe poseer un registro asociado, el cual tiene asignada una dirección binaria única. Así el número de líneas de este *bus* determina el número de registros direccionados y por ende el

tamaño del direccionamiento. Por ejemplo un CPU con 13 líneas de direccionamiento puede acceder a un total de 8 192 (2^{13}) localidades o registros.

2.3.3 Bus de control

Estas líneas son utilizadas para realizar acciones de control, como por ejemplo, indicarle a la memoria o a un puerto de entrada-salida que se le va a escribir o a leer.

2.4 Arquitectura de microcontroladores

Existen dos formas de clasificar la arquitectura de los microcontroladores, una desde el punto de vista de los *buses* que son utilizados entre el microprocesador y la memoria y la otra desde el punto de vista de la estructura interna del microprocesador.

2.4.1 Harvard y von Newmann

La arquitectura Harvard es aquella en donde los datos e instrucciones se almacenan en memorias diferentes por tanto existe una memoria de datos y una memoria de programa, al igual que existe un *bus* diferente para cada una de ellas. Esto permite que se puedan ejecutar distintas operaciones simultáneamente.

Por otro lado, la arquitectura von Newmann posee solamente una memoria para almacenar tanto datos como instrucciones. De tal forma que, solamente hay un *bus* para acceder tanto a datos como a instrucciones.

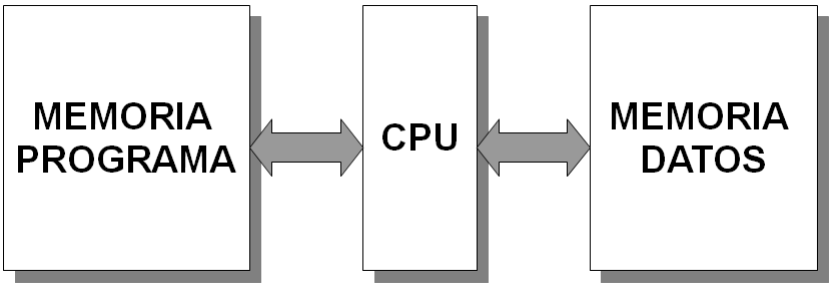


Figura 21: Arquitectura Harvard

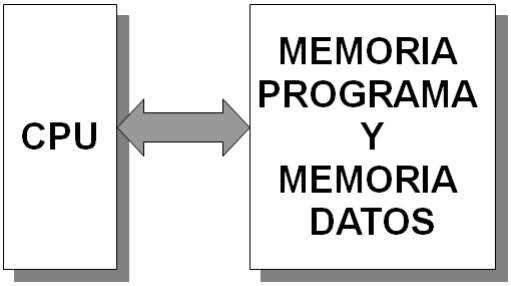


Figura 22: Arquitectura von Newmann

2.4.2 CISC Y RISC

En los inicios de los diseños de computadoras el número de las instrucciones que se diseñaban para el microprocesador eran muchas, las cuales realizaban desde tareas simples hasta muy complejas. A los microprocesadores con estas características se les dice que poseen una arquitectura CISC (C*omplex Instruction Set Computer*). De acuerdo a muchos estudios que se realizaron en los años 70´s muchas de estas instrucciones que eran implementadas en el microprocesador nunca eran usadas por los programadores y compiladores, además de que buena parte de los transistores eran usados para decodificar las instrucciones. Entonces se pensó en reducir el número de instrucciones y simplificar la tarea que cada una de ellas realizaba, con esto se llegó al diseño de los microprocesadores con arquitectura RISC (*Reduced Set Instruction Computer*) [2].

2.5 Fabricantes de microcontroladores

Realizando una comparación entre microcontroladores de 8 bits de los fabricantes, Renesas (78K0), Freescale (MC68HC11), Atmel (AVR), Texas Instrument (TMS370) y Microchip (PIC); tomando en cuenta que debían de contar con por lo menos un módulo USART (*Universal Synchronous Asynchronous Receiver/Transmitter*), que permitiera la implementación de una interfaz RS-485, se encontró que todos los fabricantes mencionados poseen microcontroladores con este módulo, por lo que, considerando lo anterior y tomando en cuenta los siguientes puntos:

- Disponibilidad de herramientas de desarrollo.
- Velocidad.
- Cantidad de memoria ROM y RAM.
- Número de terminales de entrada-salida.

se optó por elegir microcontroladores de la familia PIC16 de Microchip, para la construcción del sistema.

2.6 Programación de los microcontroladores

Existen tres tipos de lenguajes mediante los cuales se pueden programar a los microcontroladores que son: el lenguaje máquina, el lenguaje ensamblador y el lenguaje de alto nivel.

En los inicios de la programación el lenguaje utilizado fue el lenguaje máquina, que es el que manejan directamente los microprocesadores y los microcontroladores. En este lenguaje un programador debe manejar directamente patrones binarios para cada instrucción que quiera ser almacenada en la memoria de programa. Esto conlleva a manejar lógica de alto nivel, y mucho tiempo de trabajo, además de que si hubiera un error en el programa, la depuración resultaría muy complicada debido a las cadenas de '0´s' y '1´s' que deben de ser revisadas. Por ejemplo, todas las instrucciones del lenguaje máquina de los microcontroladores PIC16 tienen una longitud de 14 bits.

Debido a las dificultades del lenguaje máquina, fue inventado el lenguaje ensamblador para simplificar la programación a bajo nivel. Un lenguaje ensamblador consiste de un conjunto de símbolos mnemotécnicos, los cuales sustituyen a los bits del lenguaje máquina. Por ejemplo, los microcontroladores PIC16 cuentan con un conjunto de 35 instrucciones en ensamblador, a partir de las cuales se puede elaborar un programa.

Pero incluso el lenguaje ensamblador tiene algunas desventajas como las que se mencionan a continuación:

- El programador debe de estar familiarizado con la arquitectura del hardware en el cual el programa será ejecutado.
- Un programa largo puede ser muy difícil de entender por alguien que no sea el autor, además de que utiliza lógica de alto nivel.

Por las razones anteriores fueron inventados los lenguajes de alto nivel como C, C++. Los lenguajes de alto nivel son parecidos a los utilizados por los humanos, por lo que son más fáciles de entender e implementar. Por ejemplo, una sentencia en lenguaje de alto nivel, podría requerir de la implementación de decenas o inclusive centenas de instrucciones en lenguaje ensamblador. Así, con el uso de estos lenguajes se reduce también el tiempo de programación.

La desventaja de los lenguajes de alto nivel es que el código máquina generado al compilar un programa escrito en alto nivel resulta mucho más grande, por lo que no puede ejecutarse tan rápido como si se hubiese implementado en lenguaje ensamblador.

Un programa escrito en lenguaje de alto nivel es llamado código fuente, el cual requiere de un programa llamado compilador para trasladarlo a código máquina.

Los PIC16 también soportan la programación mediante lenguajes de alto nivel [8], por lo que en este trabajo se utiliza el lenguaje C para dicho propósito, empleando el compilador CCS, del cual se proporciona información en el apéndice.

2.7 Arquitectura de los PIC16

En la figura 23 se muestra el diagrama de bloques del PIC16F887 el cual es uno de los utilizados en este trabajo. Se puede observar que este microcontrolador posee un *bus* de datos de 8 bits y un *bus* de direcciones de 13 bits. Este microcontrolador posee arquitectura Harvard y RISC.

Las características de este microcontrolador son las siguientes:

- Memoria ROM de 4096 palabras de 14 bits.
- Memoria RAM de 256 bytes.
- Memoria EEPROM de 256 bytes.
- 5 puertos de entrada/salida.
- 2 módulos CCP (comparación, captura y PWM).

- 14 canales de conversión A/D de 10 bits.
- 3 temporizadores.
- 1 módulo de comunicación USART.
- 1 módulo de comunicación SPI/I2C.

Las características del segundo microcontrolador (PIC16F628A) mostrado en la figura 24 son las siguientes:

- Memoria ROM de 2048 palabras de 14 bits.
- Memoria RAM de 224 bytes.
- Memoria EEPROM de 128 bytes.
- 2 puertos de entrada/salida.
- 1 módulos CCP (comparación, captura y PWM).
- 3 temporizadores.
- 1 módulo de comunicación USART.

2.8 Puertos en los PIC16

En la figura 23 se observa que el PIC16F887 cuenta con 5 puertos de los cuales 4 son de 8 bits y 1 de 4 bits. Cada una de líneas que compone al puerto pueden realizar varias funciones. Por ejemplo, una línea puede funcionar como una entrada o salida digital, como una entrada analógica del canal ADC, o puede ser una línea de control del módulo SPI.

Los puertos también pueden utilizarse como un *bus* que puede manejar una comunicación paralela, para tamaños de información que van desde 2 a 8 bits.

2.9 Interrupciones

Un microcontrolador está encargado de atender a sus periféricos o a sus puertos, de manera que hay dos métodos mediante los cuales se puede hacer esto: las interrupciones y la consulta (*polling)*. Utilizando el método de interrupción un dispositivo puede solicitar en cualquier momento al CPU que se le atienda, mediante el envío de una señal de interrupción. En el momento en que el CPU recibe la señal de interrupción, el microcontrolador deja de ejecutar la tarea actual para brindarle servicio al dispositivo. Por otro lado en el método de consulta el CPU monitorea continuamente uno por uno, el estado de todos los dispositivos que tiene conectados, a los cuales sólo se les brinda servicio cuando se cumple el estado en el cual deben de atenderse. De estos dos métodos el que más se prefiere es el de interrupción, ya que no hay pérdida de tiempo como ocurre con el método de consulta, en donde se debe de estar revisando el estado de requerimiento de atención de todos los dispositivos aun cuando no lo soliciten. Otra ventaja del uso de interrupciones es que mientras no se requiera atender a un dispositivo se pueden realizar otras tareas para aprovechar de manera más eficiente al microcontrolador.

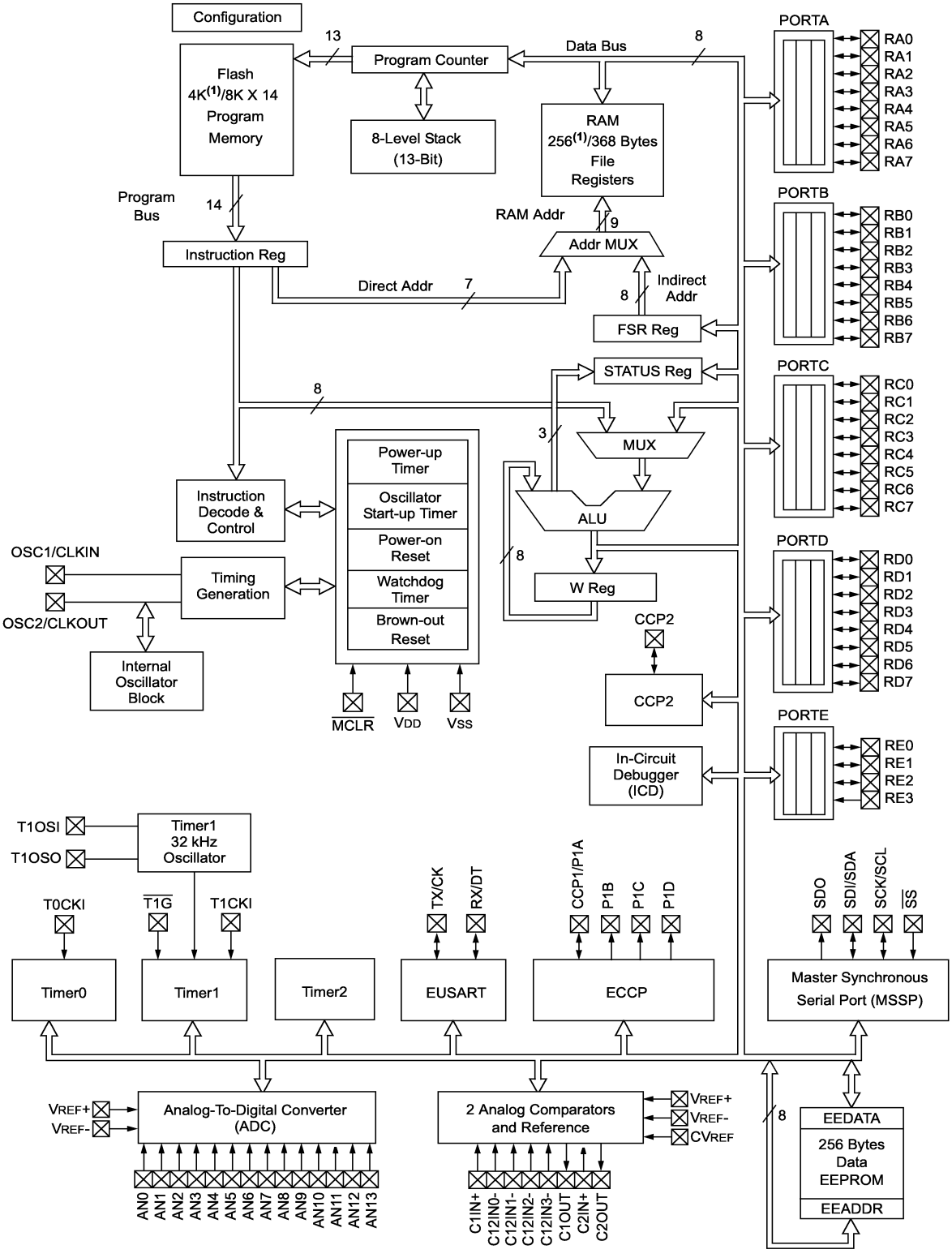


Figura 23: Diagrama de bloques del PIC16F887

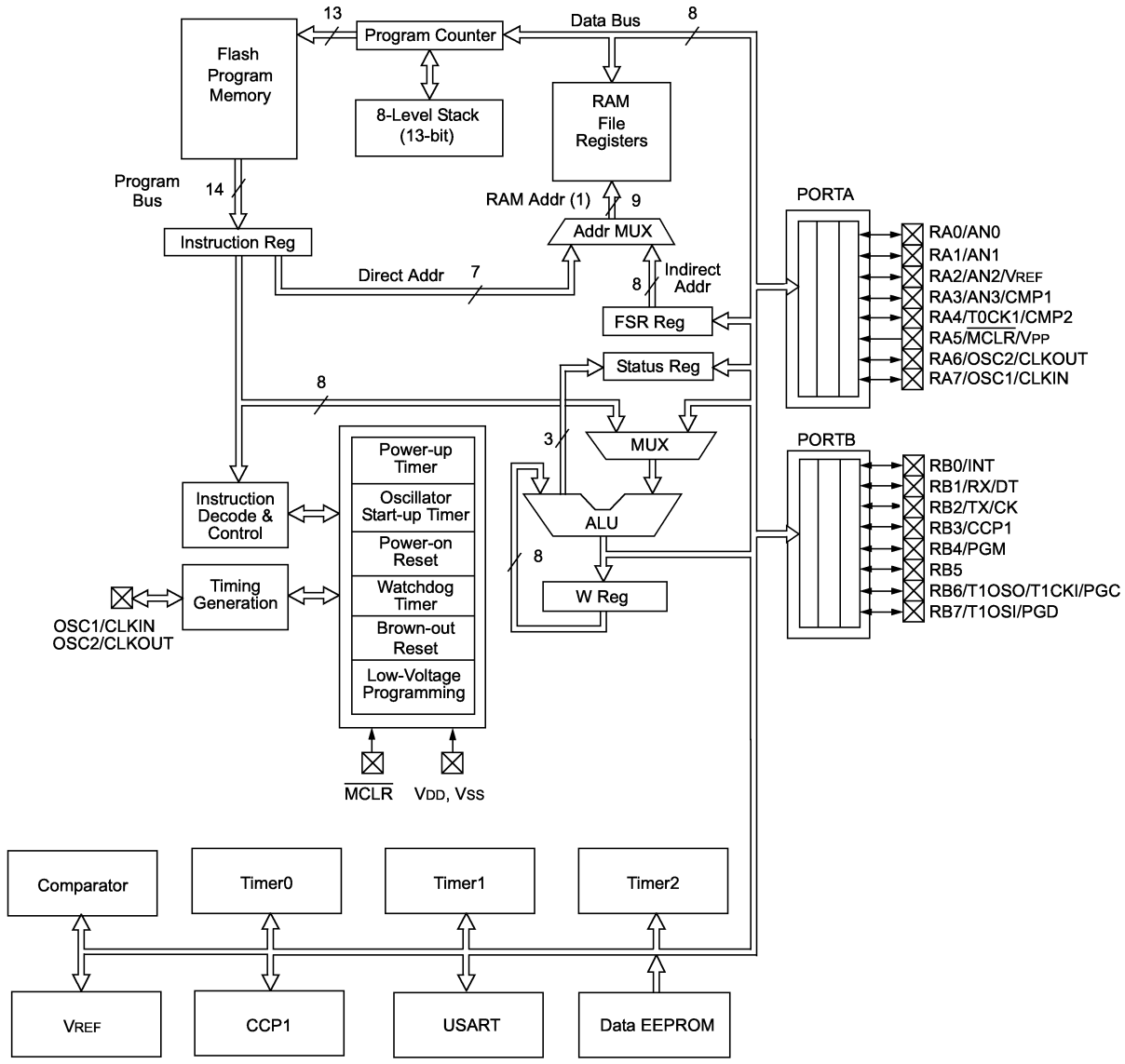


Figura 24: Diagrama de bloques del PIC16F628A

Algunos de los dispositivos que pueden interrumpir al CPU en los PIC16 son la recepción y transmisión de datos en la USART, los temporizadores, los puertos, el convertidor analógico-digital, los módulos de comparación, etc.

2.10 USART

La USART es un periférico con el que cuentan algunos microcontroladores de la familia PIC16, PIC17 y PIC18 de Microchip, el cual es un módulo de comunicación serial que puede usarse para manejar el protocolo RS-232 o el protocolo RS-485. La función principal de este módulo es recibir y transmitir datos en forma serial y puede operar en forma síncrona o asíncrona. Este módulo cuenta con dos líneas, así en la operación síncrona, una línea opera como reloj y la otra como transmisión y recepción de datos (modo *half duplex)*, mientras que en la operación asíncrona una línea es para transmitir (Tx) y la otra para recibir datos (Rx); en este modo no hay un reloj que acompañe a los datos, como se muestra en la figura 25.

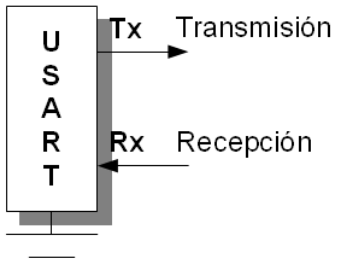


Figura 25: Operación asíncrona

El tamaño del paquete de datos (*frame*) que se pueden manejar por cada transferencia es de 8 o 9 bits. Generalmente las aplicaciones se realizan utilizando bloques de 8 bits aunque a veces se puede agregar un bit de más, para manejar datos de 9 bits, para agregar un bit más de STOP, para usarlo como bit de paridad o para crear direccionamientos de dispositivos.

La USART también puede generar interrupciones cuando se ha completado una recepción o una transmisión.

2.10.1 Operación asíncrona

En este modo de operación en el que no hay una línea de reloj que acompañe a los datos por estar operando en modo asíncrono, se puede transmitir y recibir simultáneamente por lo que trabaja en modo *full duplex*.

Dado que no hay un reloj separado en la operación asíncrona, el receptor necesita un método de sincronización con el transmisor. Esto se logra teniendo una velocidad fija (*baud rate)* de transmisión de datos, un bit de comienzo (*START*) y uno de fin (*STOP*).

El estado de las líneas Tx (transmisión) y Rx (recepción) cuando no hay transmisión o recepción de datos es '1' lógico. Cuando ellas transitan al estado '0' lógico, comienza la transmisión o recepción del bit de comienzo. Entonces el dispositivo receptor usa la duración del bit de comienzo para poder sincronizarse con los bits siguientes.

El orden en que se transmiten los bits de datos es desde el menos significativo hasta el octavo o noveno bit más significativo según sea el caso. El bit de *STOP* sigue después del último bit de

datos y es siempre alto, por lo tanto, una transmisión siempre termina con la línea en estado alto o '1' lógico. Un ejemplo de la señal transmitida en la línea Tx o recibida en la línea Rx se muestra en la figura 26.

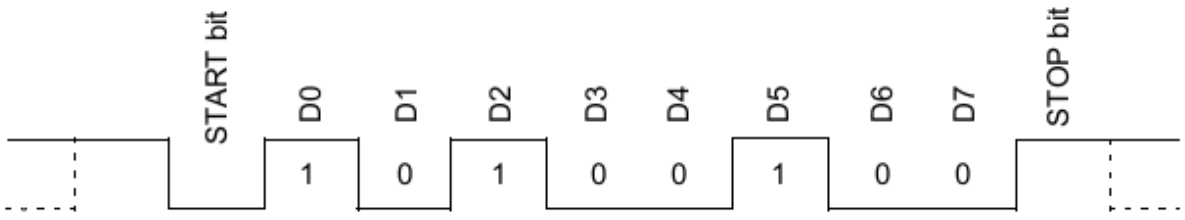


Figura 26: Señal asíncrona transmitida o recibida por la USART

La USART está compuesta por un transmisor y un receptor. El corazón del transmisor es un registro de desplazamiento, el cual transmite en forma serial a los datos que son colocados en él en forma paralela mediante software. El receptor también utiliza un registro de desplazamiento para recibir los datos seriales uno por uno; una vez que todos los bits son recibidos se verifica el bit de STOP y son movidos a un buffer FIFO (primeras entradas primeras salidas), en donde podrá ser leído de forma paralela mediante software.

Las velocidades de transmisión que pueden ser programadas en la USART van desde 300 hasta 115200 bits/segundo.

3. DISEÑO DEL SISTEMA DE COMUNICACIÓN

El sistema de comunicación de esta tesis trabaja en una configuración maestro-esclavos, por lo que todo el control de la comunicación es llevado a cabo por el dispositivo maestro. La comunicación del sistema se realiza mediante el envío o recepción de bloques de bytes, utilizando los módulos USART de los microcontroladores PIC, que son los elementos que conforman tanto al dispositivo maestro como a los esclavos.

En este capítulo, primeramente se revisará la estructura del protocolo de comunicación, posteriormente las interfaces utilizadas en el sistema y, por último, la arquitectura y programación del dispositivo maestro y dos esclavos.

3.I Estructura del protocolo de comunicación

Tomando en cuenta lo revisado en el capítulo 1 (***1.5 Protocolo)*** sobre las características con las que debe de contar un protocolo de comunicación, en el sistema de comunicación que se desarrolla en este trabajo, el protocolo cuenta con las siguientes características: bloques de datos, control de línea, control de error y control del *time out*. Cada una de estas características serán detalladas a lo largo del trabajo, comenzando con la descripción de los bloques de datos.

BLOQUES DE DATOS

En este apartado, se revisa el formato de cada uno de los mensajes que toman parte en el proceso de comunicación. En la figura 27 puede observarse que existen cinco tipos diferentes de bloques de datos, los cuales transitan entre la PC, el dispositivo maestro y los dispositivos esclavos. Estos bloques reciben los siguientes nombres: bloque de comandos PC, bloque de información PC, bloque de comandos, bloque de información y bloque de error. La estructura de estos bloques será explicada en los siguientes subtemas.

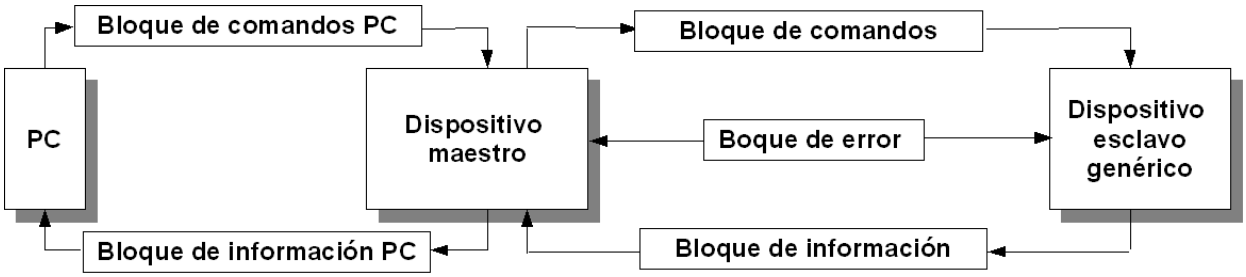


Figura 27: Esquema del protocolo de comunicación

3.1.1 Comunicación entre el dispositivo maestro y los dispositivos esclavos

Cuando la comunicación ocurre entre el dispositivo maestro y los dispositivos esclavos, el maestro es el que siempre inicia y termina la comunicación. Como se observa en la figura 27, los bloques de datos implicados en esta comunicación son: los bloques de comandos, los bloques de información y los bloques de error, que serán detallados a continuación.

BLOQUE DE COMANDOS

Los bloques de comandos solamente pueden viajar del dispositivo maestro a algún esclavo y tienen una estructura como la que se muestra en la figura 28. Cada uno de estos bloques está formado por 4 tipos de bytes: dirección, comando n, paridad n y retorno de carro (ASCII '13'). El byte "dirección" contiene la dirección del esclavo al cual va a ser enviado un bloque de comandos. Los bytes con la palabra "comando" corresponden a las instrucciones que se envían al esclavo seleccionado por el primer byte. El número de comandos que pueden enviarse a algún esclavo, pueden variar desde 1 hasta n (comando enésimo), dependiendo de las funciones que tenga cada dispositivo esclavo. Los bytes "paridad" corresponden a la paridad de cada uno de los comandos que constituye el bloque. Por ejemplo, "paridad 1" es el byte de paridad que corresponde al "comando 1". Los bytes de paridad sirven, como se revisó en el capítulo 1, para detectar errores en la transmisión de la información. El último byte que corresponde al retorno de carro o ASCII '13' permite al esclavo determinar el final de este bloque.

Dirección	Comando1	Paridad 1	Comando 2	Paridad 2	- - -	Comando n	Paridad n	Retorno de carro(ASCII 13)

Figura 28: Bloque de comandos

BLOQUE DE INFORMACIÓN

El bloque de información mostrado en la figura 29 es similar al bloque de comandos, excepto porque, este bloque sólo se transmite de algún esclavo al dispositivo maestro y porque en lugar de contener bytes de comandos este bloque contiene bytes de información. Así, el primer byte "dirección", contiene la dirección del esclavo que envía el bloque hacia el dispositivo maestro. Los bytes "byte inf n" corresponden a la información que el dispositivo esclavo transmite al dispositivo maestro y la cantidad de este tipo de bytes en un bloque de información puede variar desde 1 hasta n (byte de información enésimo), dependiendo de la naturaleza de la información que maneje cada esclavo. Nuevamente son usados bytes de paridad para cada uno de los bytes de información para la detección de algún error que pudiese ocurrir. El último byte, que es el retorno de carro permite al dispositivo maestro detectar el momento en que el bloque de información finaliza.

Dirección	Byte inf 1	Paridad 1	Byte inf 2	Paridad 2	- - -	Byte inf n	Paridad n	Retorno de carro(ASCII 13)

Figura 29: Bloque de información

BLOQUE DE ERROR

Un bloque de error está compuesto por tres bytes como se muestra en la figura 30. Este bloque puede viajar de algún dispositivo esclavo al dispositivo maestro o viceversa. Un

dispositivo esclavo siempre envía al dispositivo maestro un bloque de error, después de que el esclavo ha recibido un bloque de comandos, para informarle al dispositivo maestro si el bloque de comandos llegó con o sin errores. Y a la inversa, el dispositivo maestro siempre envía un bloque de error, al dispositivo esclavo que le haya enviado un bloque de información, para informarle si el bloque llegó con o sin errores. Revisando a los tres bytes que conforman al bloque de error, el primero de ellos corresponde a la dirección, e indica de qué esclavo procede el bloque de error o hacia qué esclavo se dirige. El segundo byte puede ser el carácter ASCII 'F' o 'V'. Cuando el byte es el ASCII 'F' indica que ocurrió un error en el bloque de comandos o en el bloque de información, dependiendo en qué dirección se esté transmitiendo este bloque. Y al contrario, si el byte es el carácter ASCII 'V' indica que no ha ocurrido ningún error. Finalmente el tercer byte que corresponde al retorno de carro indica el fin de este bloque.

Figura 30: Bloque de error

3.1.2 Comunicación entre la PC y el dispositivo maestro

La información que viaja del dispositivo maestro a los dispositivos esclavos, puede provenir de las entradas propias que posee el dispositivo maestro o de la PC, y la información que se recibe de los esclavos puede ser mostrada en un display o puede ser enviada a la PC para ser visualizada e incluso procesarla. De esta manera, la PC se convierte en un dispositivo de entrada-salida que se puede anexar al dispositivo maestro para aumentar la capacidad gráfica, capacidad de almacenamiento y manipulación de datos del sistema, con el uso de una aplicación elaborada en LabVIEW. En este sentido, la información que se envía o recibe de los esclavos puede ser visualizada en una PC, teniendo como bloque intermedio al dispositivo maestro, el cual toma la información que proviene de la PC, la procesa y la envía a los esclavos o toma la información que provienen de los esclavos, la procesa y la envía a la PC.

Los bloques de datos involucrados entre la PC y el dispositivo maestro son, como se mostraron en la figura 27, el bloque de comandos PC y el bloque de información PC, los cuales serán descritos a continuación.

BLOQUE DE COMANDOS PC

Los bloques de comandos PC (figura 31) siempre transitan de la PC al dispositivo maestro y su objetivo es solicitar la comunicación con algún esclavo desde la PC, a través de LabVIEW. El primer byte que conforma al bloque de comandos PC, puede ser, o el carácter ASCII 'W' o el ASCII 'X', que indica el tipo de comunicación que se va a llevar a cabo con algún esclavo: simple o compuesta. Si el byte corresponde al ASCII 'W' la comunicación es compuesta y si es el carácter ASCII 'X' la comunicación es simple. Estos tipos de comunicación serán explicados más adelante. El segundo byte contiene la dirección del esclavo, con el que se va a establecer una comunicación. Los bytes “comando” contienen las instrucciones que serán enviadas al esclavo. La cantidad de bytes “comando” que puede contener un bloque de comandos PC puede variar desde 1 hasta n (enésimo byte comando) dependiendo de las instrucciones que ejecute cada esclavo. El último byte (retorno de carro) permite al dispositivo maestro identificar el final de este bloque.

Una vez que el dispositivo maestro ha recibido un bloque de comandos PC, este le elimina al bloque el primer byte y le agrega bytes de paridad a cada uno de los comandos para que el bloque tenga la misma estructura que un bloque de comandos, que pueda ser enviado al esclavo con el que se quiera entablar una comunicación.

ASCII W o X	Dirección	Comando1	Comando 2	- - -	Comando n	Retorno de carro(ASCII 13)

Figura 31: Bloque de comandos PC

BLOQUE DE INFORMACIÓN PC

El bloque de información PC sólo se transmite del dispositivo maestro a la PC y está compuesto solamente por bytes de información. Este bloque puede contener desde 1 hasta n bytes de información, como lo muestra la figura 32.

Byte inf 1	Byte inf 2	- - -	Byte inf n

Figura 32: Bloque de información PC

3.1.3 Flujo de los bloques de datos en una comunicación

La forma en que interactúan los bloques de datos que se revisaron anteriormente, en una comunicación entre el dispositivo maestro y algún esclavo, así como, la implementación del control de línea, el control de error y el control del *time out* del protocolo de comunicación son mostrados en el diagrama de flujo de la figura 33.

CONTROL DE LÍNEA

Dado que el sistema es *half duplex* se necesita de un control de línea que permita invertir el sentido en que viaja la información entre el dispositivo maestro y los esclavos. En el diagrama de flujo de la figura 33 esta señal de control se llama CFD (control de flujo de datos), y puede observarse que en el caso del dispositivo maestro, esta señal pasa del estado enviar al estado recibir inmediatamente después de que el maestro ha enviado un bloque de comandos, para poder recibir un bloque de error. En el esclavo esta señal de control cambia de modo recibir a modo enviar inmediatamente después de que ha recibido un bloque de comandos para poder enviar un bloque de error al dispositivo maestro.

CONTROL DE ERROR

El control de error es implementado en este sistema para la detección de errores tanto en los bloques de comandos como en los de información, y se realiza mediante la revisión de los bytes de paridad de los bloques mencionados. En la figura 33 se observa que la revisión de paridad es realizada por el esclavo después de que ha recibido un bloque de comandos y por el dispositivo maestro después de que ha recibido un bloque de información, estas operaciones determinan el carácter ASCII que contendrá el bloque de error, un carácter ASCII 'V' cuando no hubieran errores y un carácter ASCII 'F' cuando hubiesen errores, de manera que, si ocurriera un error en los bloques de comandos o de información estos serían reenviados (el bloque de comandos al

esclavo respectivo y el bloque de información al dispositivo maestro) tantas veces hasta que ya no se presentasen errores.

CONTROL DEL *TIME OUT*

El *time out* (figura 33) es el tiempo mínimo que espera el dispositivo maestro a que llegue un bloque de error, tanto si la comunicación es solicitada desde la PC o de las entradas propias del dispositivo maestro, para conocer el estado en que llegó el bloque de comandos enviado por el maestro hacia algún esclavo. El tiempo mínimo de espera corresponde a 4 segundos, por lo que si después de este tiempo no se ha recibido un bloque de error, el sistema procede a mostrar en un display el mensaje “ERROR DISPOSITIVO”, para indicar que el esclavo no ha recibido el bloque de comandos o que no se encuentra conectado al sistema de comunicación.

COMUNICACIÓN SIMPLE Y COMPUESTA

Antes de que el dispositivo maestro inicie una comunicación con algún esclavo, primero se define si la comunicación va a ser simple o compuesta. En el diagrama de flujo de la figura 33, la comunicación simple o compuesta se define con el carácter ASCII que toma la variable 'com', si es simple 'com=X' y si es compuesta 'com=W'.

Una comunicación es simple (figura 33) cuando solamente intervienen dos tipos de bloques de datos, un bloque de comandos que es enviado por el dispositivo maestro hacia algún esclavo y la respuesta de este con un bloque de error. Si el bloque de error que recibe el dispositivo maestro contiene el carácter ASCII 'V', entonces la comunicación finaliza debido a que no se presentaron errores en el bloque de comandos que se envió al esclavo, en caso contrario el dispositivo maestro reenviaría al esclavo, el bloque de comandos tantas veces mientras se presentasen errores. Dado que, en este tipo de comunicación el dispositivo maestro no recibe información del esclavo que sea diferente a la de un bloque de error, su objetivo es solamente actualizar el estado de algún esclavo, como podría ser la simple apertura o cierre de una válvula conectada al esclavo.

Por otro lado, en una comunicación compuesta (figura 33) con algún esclavo, se repite el proceso de la comunicación simple, es decir, se envía al esclavo un bloque de comandos y se recibe un bloque de error de él, y una vez que ya no haya errores o no se hayan presentado errores en el bloque de comandos enviado al esclavo; el dispositivo maestro procede a esperar un bloque de información de ese esclavo. Cuando se recibe el bloque de información, el dispositivo maestro ejecuta la revisión de paridad de este bloque, para que después envíe un bloque de error al esclavo, que le comunique el estado en que llegó el bloque de información, si este no presentó errores, entonces se procede a procesar la información del bloque para mostrarlo en un display. En caso de que hubiese errores, el esclavo reenviaría el bloque de información al maestro hasta que ya no hubiese errores. Este tipo de comunicación es el que se realiza cuando el dispositivo maestro solicita información a algún dispositivo esclavo, como es el caso del esclavo tipo 1 que se verá, más adelante.

Dado que en el sistema pueden haber conectados hasta 32 esclavos, todos ellos recibirían el bloque de comandos que envía el dispositivo maestro, pero como se observa en la figura 33 solamente el esclavo que coincida con la dirección que contiene el bloque de comandos es el que procesa al bloque.

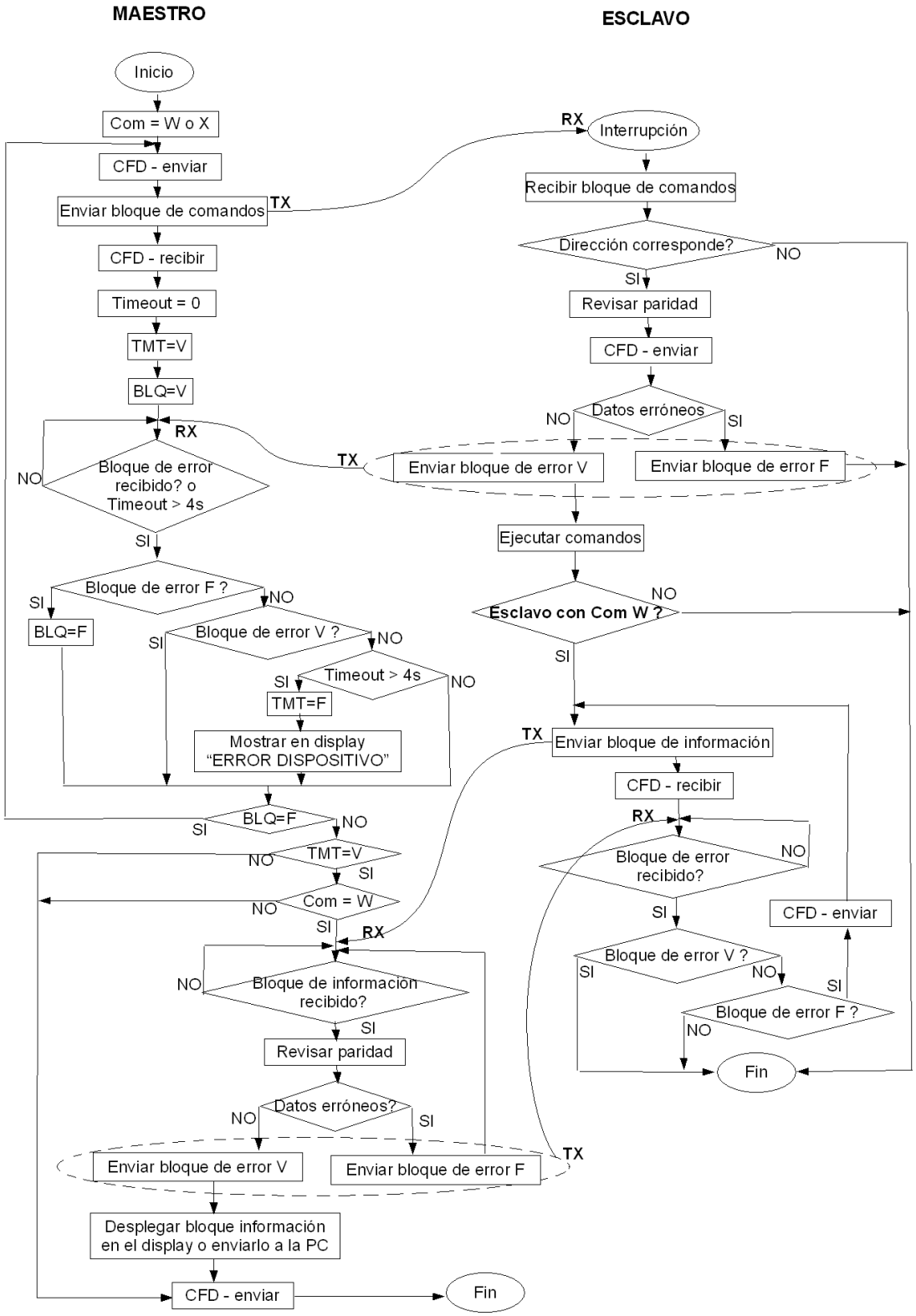


Figura 33: Diagrama de flujo de una comunicación entre el dispositivo maestro y algún esclavo

3.1.4 Paridad de los datos

La paridad de un byte en este trabajo no consiste de un sólo bit como el revisado en el capítulo 1, sino que consta de un byte. Pero este byte sigue conservando los valores binarios '0' y '1' de la paridad como si se tratará de un bit. El hecho de manejar bytes en lugar de bits es para que todos los bloques de datos estén constituidos por bytes. La paridad solamente se aplica, como se explicó en la descripción de los bloques de datos, a los bytes de comandos de los bloques de comandos y a los bytes de información de los bloques de información.

El tipo de paridad manejado en este sistema es par, lo cual quiere decir que si se tiene el comando '00110001' que corresponde al carácter ASCII '1' entonces el byte de paridad asignado a este comando es el siguiente '00000001', dado que la suma de los bits '1´s' del comando con los bits del byte de paridad tiene que dar un número par, por estar utilizando la paridad par.

La revisión de paridad es realizada tanto por el esclavo como por el maestro, cuando se trata de una comunicación compuesta, debido a que intervienen tanto bloques de comandos como bloques de información, y solamente por el esclavo cuando se trata de una comunicación simple, en donde únicamente intervienen bloques de comandos.

3.2 Interfaces de comunicación serial

Dado que el sistema de comunicación trabaja en un esquema maestro-esclavo, el control de toda la comunicación es llevada a cabo por el dispositivo maestro. En la figura 34 se observa que el dispositivo maestro debe establecer comunicación tanto con los esclavos como con la PC.

De las interfaces seriales de comunicación revisadas en el capítulo 1, que pueden implementarse con el uso de microcontroladores, se eligieron dos para el sistema de comunicación de este trabajo; la interfaz RS-232 y la interfaz diferencial RS-485.

En el caso de la comunicación entre el dispositivo maestro y los esclavos, los puntos a considerar fueron: la trasmisión de información a una distancia mayor de 20 metros, la comunicación multipunto y la inmunidad al ruido. Ésto llevó a la elección de la interfaz diferencial RS-485. Esta interfaz diferencial combinada con los módulos USART de los microcontroladores PIC (componentes del maestro y el esclavo) permitió transportar, transmitir y recibir respectivamente los bloques de datos que transitan entre el dispositivo maestro y los esclavos.

Por otro lado, la interfaz utilizada para la comunicación entre el maestro y la PC es la interfaz RS-232, ya que, además de que la distancia máxima soportada por esta interfaz no se excede, se puede implementar de manera sencilla con el uso de un módulo USART de los microcontroladores que componen al dispositivo maestro. De modo que los módulos USART se utilizaron para implementar las interfaces RS-485 y RS-232 como lo muestra la figura 34.

Debido a que la interfaz RS-485 maneja señales diferenciales y éstas no pueden ser manejadas directamente por los microcontroladores que conforman al dispositivo maestro y los esclavos, se tuvo que realizar una conversión de señales diferenciales a señales referidas a tierra con los transmisores-receptores 485 (MAX487).

3. DISEÑO DEL SISTEMA DE COMUNICACIÓN

De las 25 señales que puede manejar el protocolo RS-232, para enlazar a la PC con el dispositivo maestro solamente se utilizaron 3, una para transmitir datos, una para recibir datos y la señal de tierra.

Dado que, los niveles de voltaje utilizados por la USART difieren de los manejados por la PC, se utilizó un convertidor de niveles de voltaje (MAX232) como lo muestra la figura 34.

Debido a que, los puertos de comunicación COM de las computadoras que permitían el manejo de la interfaz RS-232 han desaparecido prácticamente, hoy en día para seguir trabajando con este tipo de interfaz, en el mercado se pueden encontrar cables USB que permiten la emulación de estos puertos.

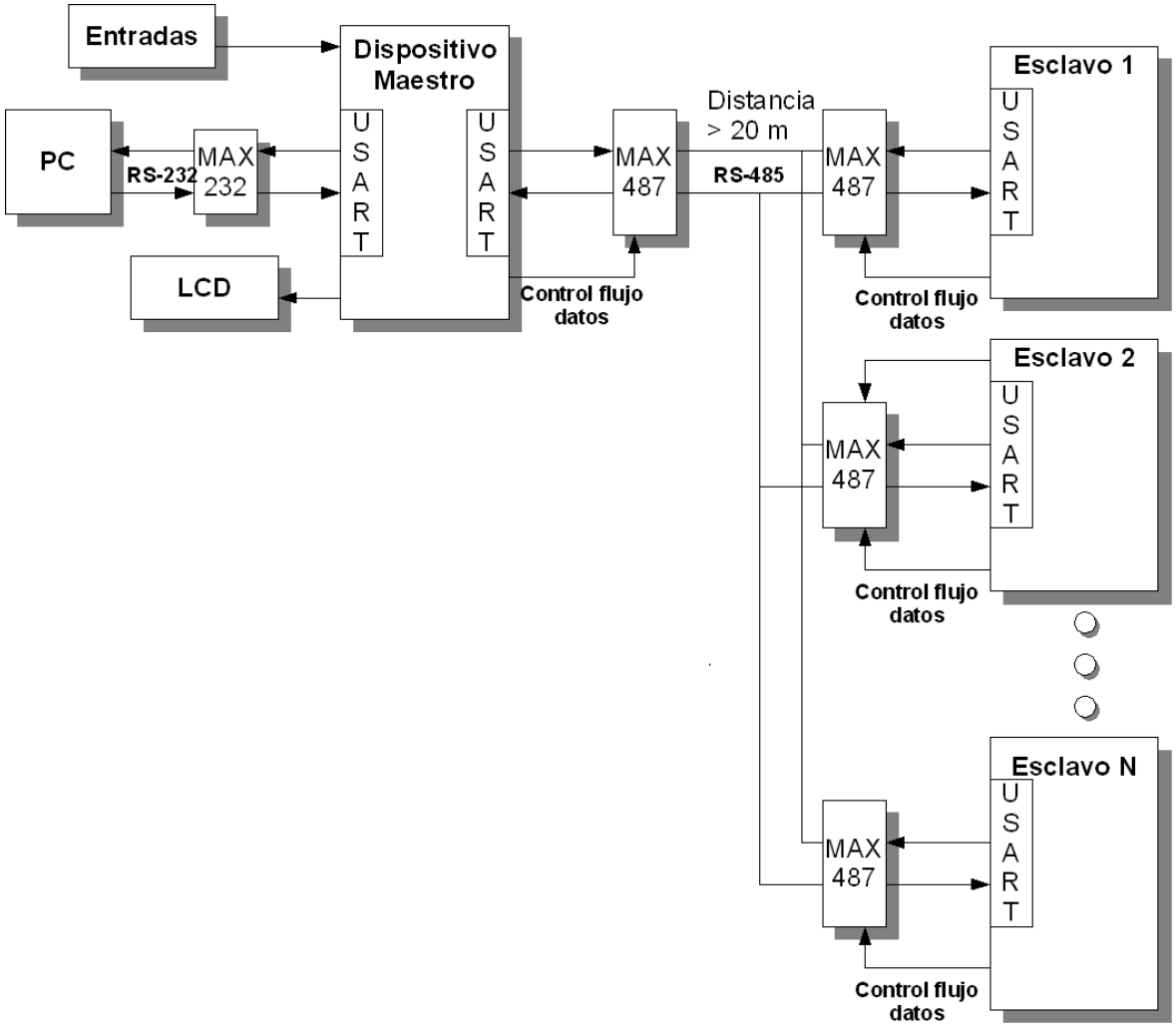


Figura 34: Esquema de las interfaces seriales utilizadas en el sistema de comunicación

3.3 Diseño del dispositivo maestro

3.3.1 Arquitectura del dispositivo maestro

En cada uno de los diseños primero se revisará la arquitectura y posteriormente la programación del dispositivo. En la figura 35 se muestra el esquema de la arquitectura del dispositivo maestro. Debido a que el dispositivo maestro tiene que establecer comunicación tanto con los dispositivos esclavos, como con la PC, se hizo un arreglo de dos microcontroladores utilizando el módulo USART que posee cada uno de ellos.

A continuación se describirá la función que posee cada uno de los elementos del dispositivo maestro.

MAX232. Se encarga de realizar la conversión de voltajes entre señales RS-232 que provienen de la PC a señales TTL manejadas por los microcontroladores PIC´s.

MAX487. Este circuito integrado es un transmisor-receptor 485 que se encarga de realizar la conversión de señales referidas a tierra a señales diferenciales.

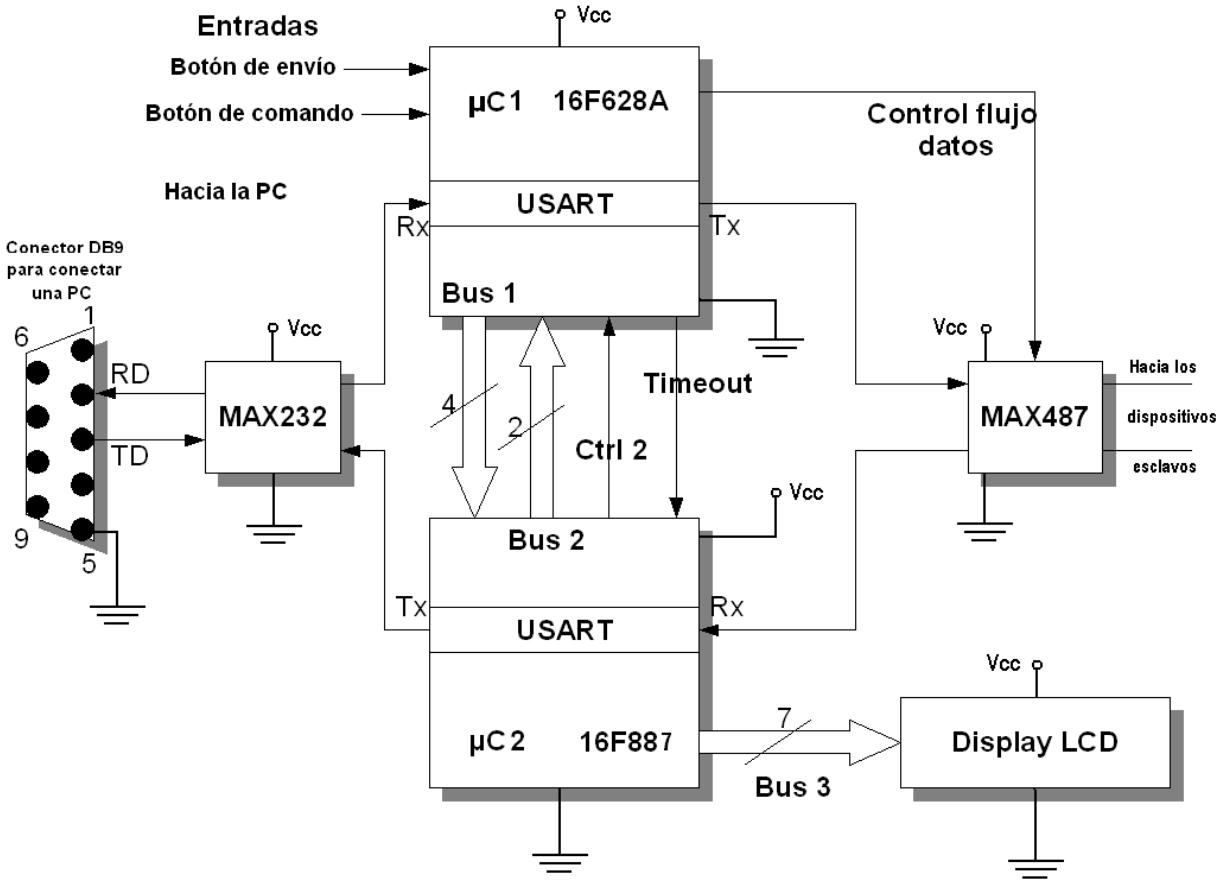


Figura 35: Arquitectura del dispositivo maestro

Microcontrolador 1 (μC 1). Se encarga, de recibir el bloque de comandos PC que proviene de la PC a través de la terminal Rx de su USART, de atender sus dos entradas digitales (botón de comandos y botón de envío), de enviar los bloques de comandos o de error a los esclavos por medio de la terminal Tx de su USART, de controlar el flujo de datos (CFD) del circuito integrado MAX487 y de enviar información al microcontrolador 2 a través del *bus* 1 y la línea *time out*.

Microcontrolador 2 (μC 2). Se encarga, de recibir los bloques de información o de error que proviene de los esclavos a través de la terminal Rx de su USART, de transmitir los bloques de información PC a la PC mediante la terminal Tx de su USART, de desplegar información en el display LCD por medio del *bus* 3 ya sea que ésta provenga de los esclavos o del μC 1, y de enviar información al μC 1 a través del *bus* 2 y la línea ctrl 2.

Botón de comandos. Es un *push botton*, que cada vez que es oprimido incrementa un contador interno en el μC 1. El valor de este contador determina un bloque de comandos, el cual contiene tanto la dirección del esclavo con el que quiere establecer comunicación, como el comando o comandos que le serán enviados.

Botón de envío. Es un *push botton,* que cada vez que se presiona envía un bloque de comandos, al esclavo que indique el primer byte de este bloque. De manera que, también al presionarse se indica que se inicia una comunicación con ese esclavo.

Bus 1. Conjunto de 4 líneas que transfieren el valor del contador interno de comandos en forma paralela del μC 1 al μC 2 para que sea mostrado en el display que maneja el μC 2.

Bus 2. Conjunto de 2 líneas que el μC 2 usa para informar al μC 1, si el bloque de comandos que le fue enviado por el μC 1 a algún esclavo presentó o no errores. También por este mismo *bus* el μC 2 comunica al μC 1, si el bloque de información que fue recibido de algún esclavo presentó errores. Cuando no transmite información este *bus* siempre permanece en el estado binario '11', de modo que, las transiciones de estados binarios '11' a '10' y luego a '11' le indican al μC 1 que, el bloque de comandos enviado o el bloque de información recibido presentaron errores, y las transiciones binarias de '11' a '01' y luego a '11' le informan al μC 1 que no hubo errores en cualquiera de estos dos bloques.

Bus 3. Conjunto de 7 líneas que muestran en el display la información contenida en el bloque de información previamente procesada que se recibió de algún esclavo. También se encarga de mostrar el valor actual del contador interno del μC 1.

Ctrl 2. Es una línea de control del microcontrolador 2, que siempre se mantiene en '1' binario, la cual tiene la función de informarle al μC 1, el momento en que la comunicación con un esclavo ha terminado, mediante los cambios binarios '1', '0', '1'. Esto sirve para que no haya un traslape de solicitudes de comunicación con los esclavos.

Time out. Es una línea de control del μC 1 que se utiliza para informar al μC 2, que el tiempo, en que debe responder un esclavo al envió de un bloque de comandos, se ha sobrepasado, lo cual indica que hay un error con el esclavo. En caso de no sobrepasar el tiempo de espera esta línea no tiene efecto. Esta señal siempre permanece en el estado binario '1' por lo que los cambios binarios '1', '0', '1' son los que permiten informar al μC 2 que se ha sobrepasado el tiempo de espera.

3.2.2 Programación del dispositivo maestro

Dado que el dispositivo maestro está compuesto por dos microcontroladores, las tareas que se realizan en el proceso de comunicación, se distribuyen tal y como lo muestran los diagramas de flujo de las figuras 36, 37, 38 y 39, que son una implementación del diagrama de flujo que se revisó en la figura 33. Así, el programa principal del microcontrolador 1 es el que se encarga de llevar a cabo las comunicaciones que son solicitadas desde sus dos entradas (botón de comandos y botón de envío), mientras que la rutina de interrupción lleva a cabo las comunicaciones con los esclavos que sean solicitadas desde la PC. De manera que, tanto el programa principal como la rutina de interrupción del microcontrolador 1 envían bloques de comandos y de error a los esclavos, mientras que sólo la rutina de interrupción (figura 37) recibe bloques de comandos PC de la PC. En el caso del microcontrolador 2, su programa principal (figura 38) se encarga de atender los cambios en el *bus* 1 y la línea *time out*, y su rutina de interrupción (figura 39) se encarga de recibir los bloques de información y de error que provienen de los esclavos, así como de enviar bloques de información PC a la PC.

En el programa principal del microcontrolador 1 (figura 36), lo primero que se realiza al igual que en el programa principal del microcontrolador 2 (figura 38) es, la configuración de: las terminales que van a funcionar como entradas y como salidas, las variables a utilizar en el programa y la habilitación de la interrupción por recepción de datos en la USART. Después de la configuración, el microcontrolador 1 procede a revisar el estado de las entradas 'cmd' (botón de comandos) y 'env' (botón de envío) y el estado de la variable 'clv'. Las entradas (botones) 'cmd' y 'env' cuando no están presionadas permanecen en '1' lógico, por lo que en este caso el microcontrolador 1 procede a encender y apagar un LED, para poder verificar que el programa esté funcionando correctamente.

Cada vez que es oprimido ('0' lógico) el botón de comandos (entrada 'cmd') se incrementa un contador (índice), cuyo valor es enviado por el *bus* 1 hacia el microcontrolador 2 (los cambios en el *bus* 1 son detectados continuamente por el programa principal del microcontrolador 2, figura 38) para que sea mostrado en el display del sistema. El valor de este contador (índice) determina la dirección y comando que serán enviados hacía algún esclavo, tal y como lo muestra la rutina de envío de bloque de comandos de la figura 41. Una vez que se haya elegido el valor del contador que determine el comando y la dirección del esclavo con el que se va a establecer comunicación, al oprimir ('0' lógico) el botón de envío (entrada 'env'), empieza una comunicación (figura 36).

El proceso de comunicación comienza ajustando el control de flujo de datos (CFD) a modo enviar, para poder transmitir datos a los esclavos, después es enviado un bloque de comandos (figura 40) hacia el esclavo elegido y en la siguiente instrucción el control de flujo de datos es cambiado a modo recibir, para que el microcontrolador 2 a través de su rutina de interrupción (figura 39) reciba el bloque de error que enviará el esclavo, con el objeto de conocer el estado en el que ha llegado el bloque de comandos que fue enviado. Debido a que, el que recibe el bloque de error es el microcontrolador 2 y el microcontrolador 1 debe de conocer la información de este bloque, el microcontrolador 2 se encarga de informar esto al microcontrolador 1 por medio del *bus* 2. Anteriormente se revisó que la información que caracteriza a un bloque de error es, el carácter ASCII 'F' o el carácter ASCII 'V', en donde una 'F' indica que el bloque enviado presentó errores y una 'V' que no los hubo. De modo que, dependiendo de qué carácter

ASCII contenga el bloque de error, es la señal que el microcontrolador 2 genera en el *bus* 2, señales que fueron revisadas anteriormente. Una vez que el microcontrolador 1 conoce el estado con el que llegó el bloque de comandos al esclavo, si hubo errores el microcontrolador 1 procede a reenviar el bloque de comandos y a esperar nuevamente las señales del *bus* 2, este proceso se repetiría hasta que ya no se presentasen errores. Cuando ya no hubiesen errores, si la comunicación es simple (com=X) el microcontrolador procedería a cambiar el control de flujo de datos a modo enviar y finalizar la comunicación. En el caso de que se tratase de una comunicación compuesta (com=W) el microcontrolador 2 procedería a esperar un bloque de información por interrupción(figura 39), del esclavo con el que se está entablando la comunicación y una vez recibido el bloque procedería a realizar la revisión de paridad de sus bytes de información. Tanto, como si hubieran errores, como si no, en el bloque de información recibido, debe de ser informado al microcontrolador 1 por medio del *bus* 2, para que el microcontrolador 1 (figura 36) envíe al esclavo un bloque de error que le indique que proceso seguir. Si el esclavo recibe un bloque de error con el carácter ASCII 'F', el esclavo procedería a reenviar el bloque de información y a esperar otro bloque de error, proceso que se repetiría mientras se presentasen errores. En caso contrario, si el esclavo recibe un bloque de error con el carácter ASCII 'V' entonces la comunicación finaliza.

Cuando no hubiesen o ya no hubiesen errores en el bloque de información, el microcontrolador 2 (figura 39) procede a procesar el bloque y mostrar la información en el display del sistema, una vez que se haya terminado de mostrar la información, el microcontrolador 2 informa esto al microcontrolador 1 por la línea ctrl2, para que finalice el proceso de comunicación y el dispositivo se encuentre listo para otra comunicación.

Otra cosa que ocurre inmediatamente después de que se ha enviado un bloque de comandos (figura 36), es que, se incializa un contador (*time out*). Este contador permite verificar que la comunicación con un esclavo se lleve a cabo de manera correcta, es decir el esclavo con el que se ha iniciado una comunicación debe de responder con un bloque de error en un tiempo menor (aproximadamente de 1 segundo) al límite fijado en el contador (*time out*) que es de 4 segundos, ya que si el esclavo no responde antes de este tiempo quiere decir, que el esclavo no recibió el bloque de comandos o que no está conectado al sistema. Cuando no hubo respuesta del esclavo, el microcontrolador 1 informa esto al microcontrolador 2 por medio de la línea *time out* (este proceso es realizado por el programa principal del microcontrolador 2, figura 38), por lo que el microcontrolador 2 al recibir la información muestra en su display “ERROR DISPOSITIVO”.

La variable 'key2' en el programa principal del microcontrolador 1 permite definir dos estados: ocupado (key2=A) y desocupado (key2=B). Esto permite que en el caso de que hubiera conectada una PC y ésta solicitara una comunicación cuando estuviera en curso una, no se llevaría a cabo hasta después de concluida la comunicación actual. De modo que, en la rutina de interrupción del microcontrolador 1 (figura 37) hay una variable llamada 'clv' que en el momento en que ocurre una interrupción y se esté llevando a cabo una comunicación, esta variable (que puede tomar dos estados: clv=E y clv=D), permite posponer la comunicación que se ha solicitado desde la interrupción hasta que la comunicación actual finalice asignando a 'clv' el valor de 'E' (espera). Así, cuando el programa principal del microcontrolador 1 (figura 36) revisa el valor de esta variable y encuentra que 'clv=E' entonces procede a realizar esta comunicación que quedó en espera y al finalizar se asigna a 'clv' el valor de 'D' (desocupado).

Cuando ocurre una interrupción en el microcontrolador 1 (figura 37), es decir se solicita una comunicación desde la PC y no hay una comunicación en curso (key2=B), entonces la rutina de interrupción del microcontrolador 1 realiza un proceso de comunicación con el esclavo respectivo, similar al que se realiza cuando se solicita desde el botón de envío. Los únicos procesos diferentes en esta comunicación ocurren al principio, cuando al bloque de comandos PC se le quita el primer byte que indica el tipo de comunicación (simple ASCII 'X' o compuesta ASCII 'W') y a los bytes comando se les agrega bytes de paridad, para convertir el bloque de comandos PC a la estructura de un bloque de comandos que pueda ser enviado al esclavo respectivo.

La variable 'BUS2C' del programa principal del microcontrolador 1 permite determinar si se ha de reenviar el bloque de comandos al esclavo con el que se está manteniendo comunicación. El valor en esta variable es modificado cuando se recibe información por el *bus* 2, ya que dependiendo de cómo transita esta línea es el valor que toma. Así si hubo error 'BUS2C = F' y si no hubo error 'BUS2C=V'. De esta manera cuando ocurren errores se reenvía el bloque de comandos.

La variable 'BUS2I' permite al programa principal del microcontrolador 1 decidir entre, volver a esperar el reenvío de un bloque de información (BUS2I=F) porque hubo errores, o esperar a que el microcontrolador 2 termine de mostrar en el display la información procesada del bloque de información debido a que no se presentaron errores (BUS2I=V).

La variable 'TMT' permite al programa principal del microcontrolador 1 determinar, si se ha sobrepasado el tiempo (TMT=F) en que debe de responder el esclavo con el que se mantiene comunicación, o si el esclavo respondió antes de que se cumpliera el tiempo límite de espera (TMT=V). Cuando el tiempo se ha sobrepasado la comunicación finaliza, para volver a intentar otra comunicación con el esclavo o revisar si existe un problema con el esclavo.

El proceso que se encarga de enviar el bloque de comandos al esclavo respectivo es mostrado en el diagrama de la figura 41, en él se observa que dependiendo del valor del contador (índice) es la dirección y el comando que conforma al bloque de comandos que es enviado al esclavo respectivo, así como el tipo de comunicación que se establecerá (simple com=X o compuesta com=W). En la figura 41, los valores del índice que van entre 1 y 4 corresponden a los asociados con el esclavo tipo 1 de este trabajo y los valores del índice 5 y 6 están asociados con el esclavo tipo 2 revisado más adelante.

En el caso de la comunicación con el esclavo tipo 1, los bloques de comandos (siguiendo la estructura de la figura 28) que le son enviados dependiendo del valor del contador son los siguientes:

Contador=1: primer byte-ASCII '1', segundo byte-ASCII '1', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

Contador=2: primer byte-ASCII '1', segundo byte-ASCII '2', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

Contador=3: primer byte-ASCII '1', segundo byte-ASCII '3', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

Contador=4: primer byte-ASCII '1', segundo byte-ASCII '4', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

De esto, se puede observar que el primer byte el cual corresponde a la dirección no cambia mientras el contador se encuentre entre 1 y 4. El segundo byte es el que corresponde al comando. El tercer byte es el byte de paridad de los segundos bytes y el cuarto byte corresponde al retorno de carro.

Cuando la comunicación se realiza con el esclavo tipo 2, los bloques que se envían al esclavo dependiendo del valor del contador son los siguientes:

Contador=5: primer byte-ASCII '2', segundo byte-ASCII '5', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

Contador=6: primer byte-ASCII '2', segundo byte-ASCII '6', tercer byte-paridad del segundo byte, cuarto byte-retorno de carro.

Aquí se puede observar que la dirección del esclavo tipo 2 es el ASCII '2'.

En la figura 41 también se muestra que para los valores 7 y 8 del contador, los bloques de comandos serían enviados a un esclavo tipo 3 con dirección ASCII '3' y para los valores 9 y 10 del contador se establecería comunicación con un esclavo tipo 4 con dirección ASCII '4'. Siguiendo este esquema para que el dispositivo maestro se pudiera comunicar con los 32 esclavos que soporta la interfaz RS-485, el valor al que debería de llegar el contador dependería del número de comandos que estuvieran asociados a cada esclavo. Ya que en el esclavo 1 estuvieron asociados 4 comandos con el contador, y en el caso del esclavo 2, 2 comandos.

Cada vez que llega un bloque de datos al microcontrolador 2 por interrupción (figura 39), que puede ser un bloque de información o un bloque de error, el microcontrolador 2 primero revisa el primer byte, y si su valor (dirección) corresponde con la de algún esclavo que este programado, procede a revisar el segundo byte, si detecta un ASCII 'F' o 'V' se trata de un bloque de error, en caso contrario revisa el tercer byte, el cual si corresponde a un valor binario 0 ó 1, indica que se trata de un bloque de información. Estas operaciones son realizadas para filtrar bloques de datos que no sean válidos.

En el microcontrolador 2 cada esclavo debe tener asociado un bloque de código similar como el que se encuentra en el recuadro punteado de la figura 39, debido a que para cada esclavo la comunicación puede ser simple o compuesta. Además de que el proceso que se realiza sobre el bloque de información en el caso de una comunicación compuesta, puede ser diferente para cada uno de los esclavos. En la figura 39 es mostrado el proceso seguido con el esclavo tipo 1 (dirección = ASCII '1'), que se revisa en este trabajo, el cual es un esclavo que realiza una comunicación compuesta con el dispositivo maestro, lo cual quiere decir que este esclavo envía al maestro tanto bloques de error como bloques de información.

En la figura 40 se muestra el proceso que se sigue con el esclavo tipo 2 descrito más adelante, el cual tiene asignado la dirección ASCII '2'. La comunicación que el dispositivo maestro mantiene con el esclavo tipo 2 es simple, por lo tanto, lo único que envía al maestro son bloques de error. Cuando un bloque de error es recibido por el microcontrolador 2 (figura 39), en respuesta a un bloque de comandos que le fue enviado al esclavo tipo 2, el microcontrolador 2 revisa la dirección contenida en el bloque de error, si corresponde a la del esclavo tipo 2,

entonces el microcontrolador 2 informa al microcontrolador 1 por el *bus* 2 el carácter ASCII contenido en el bloque de error, de modo que el microcontrolador 1 reenviaría el bloque de comandos cuando hubiesen ocurrido errores. Por su parte el microcontrolador 2 imprime en el display el mensaje “BLOQUE ERRONEO” cuando el bloque de error contiene el ASCII 'F' y el mensaje “COMANDO EJECUTADO” cuando se trata del ASCII 'V'. Si el mensaje mostrado fue este último la comunicación finaliza después de terminar de desplegarlo.

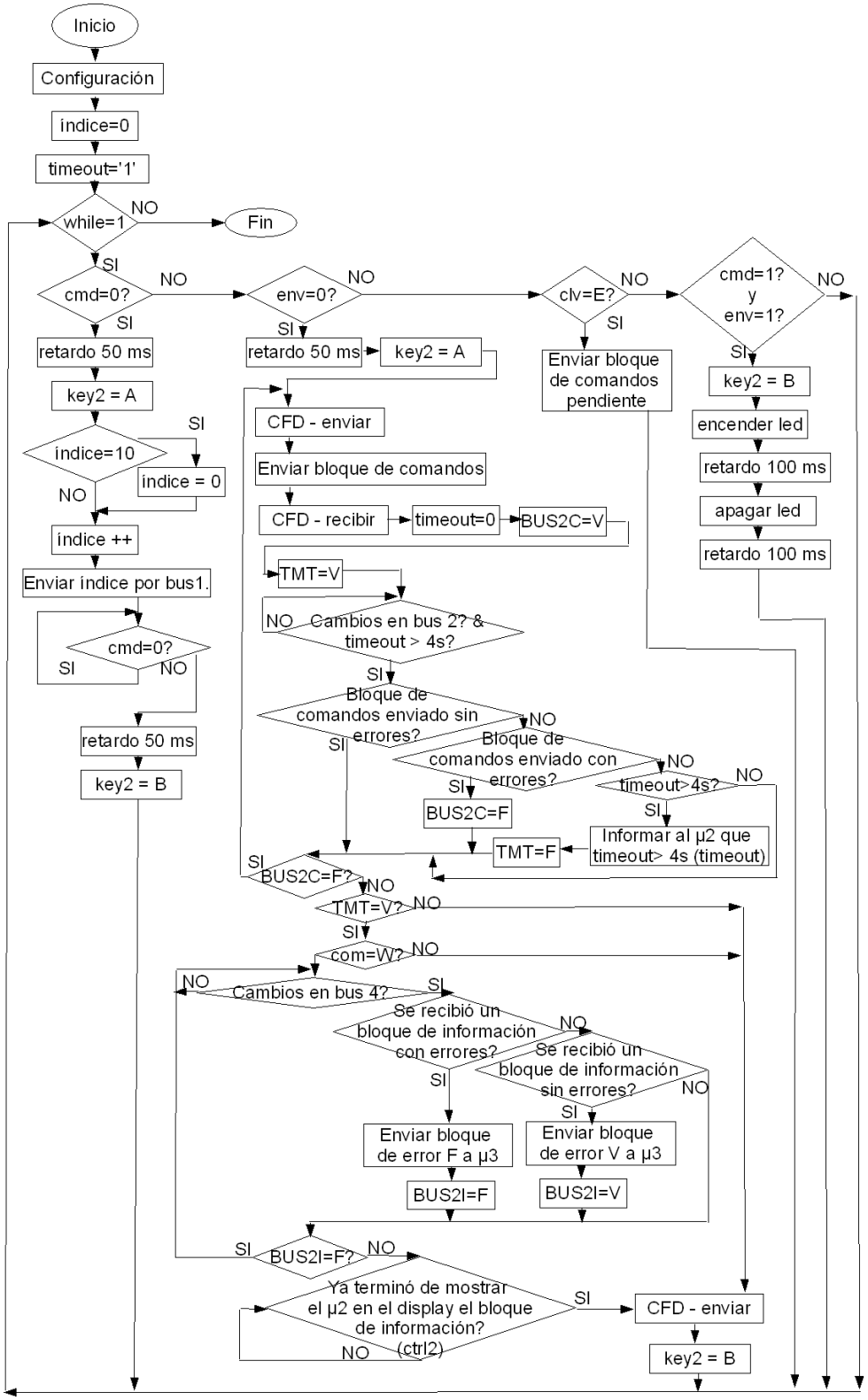


Figura 36: Programa principal del microcontrolador 1

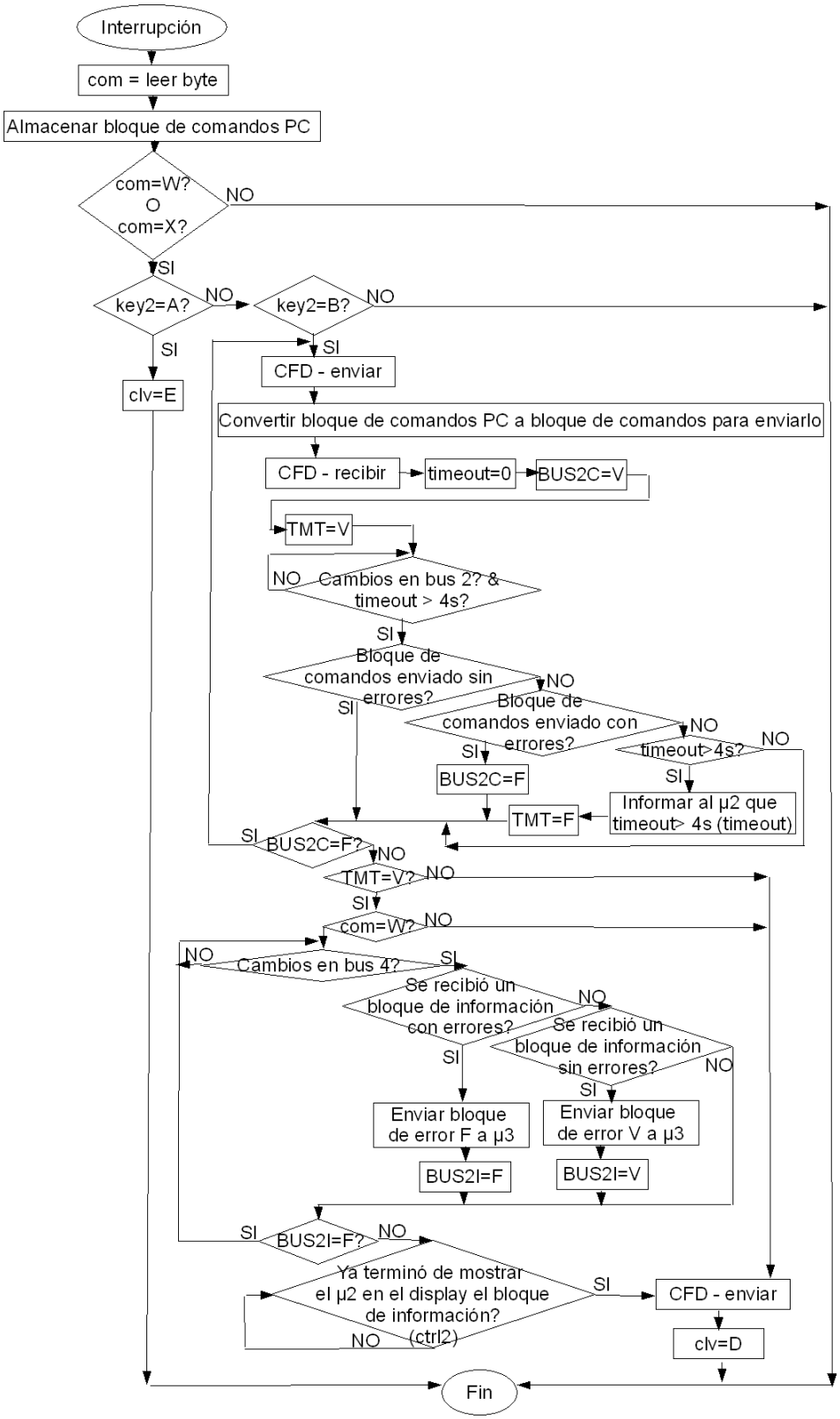


Figura 37: Rutina de interrupción del microcontrolador 1

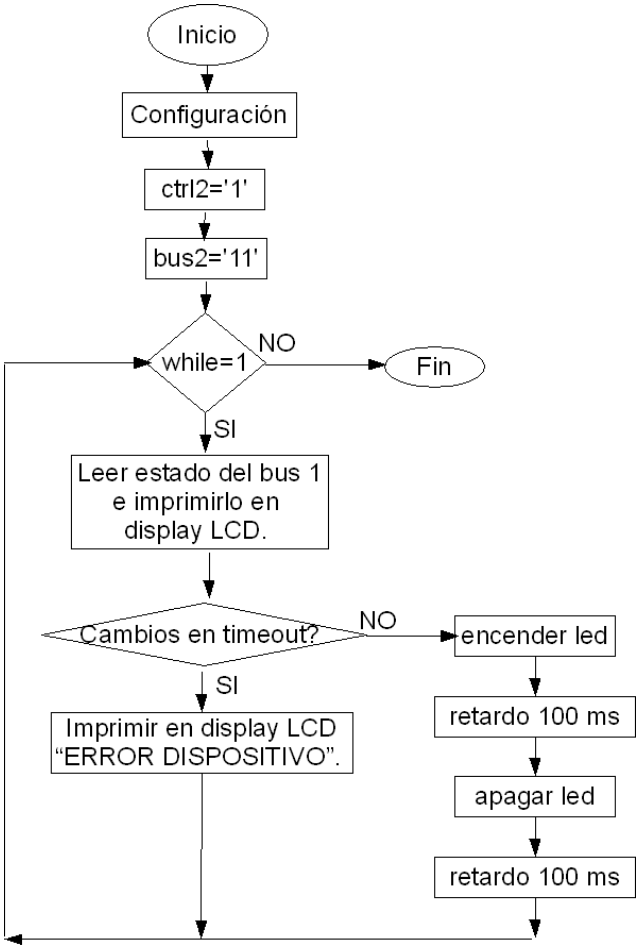


Figura 38: Programa principal del microcontrolador 2

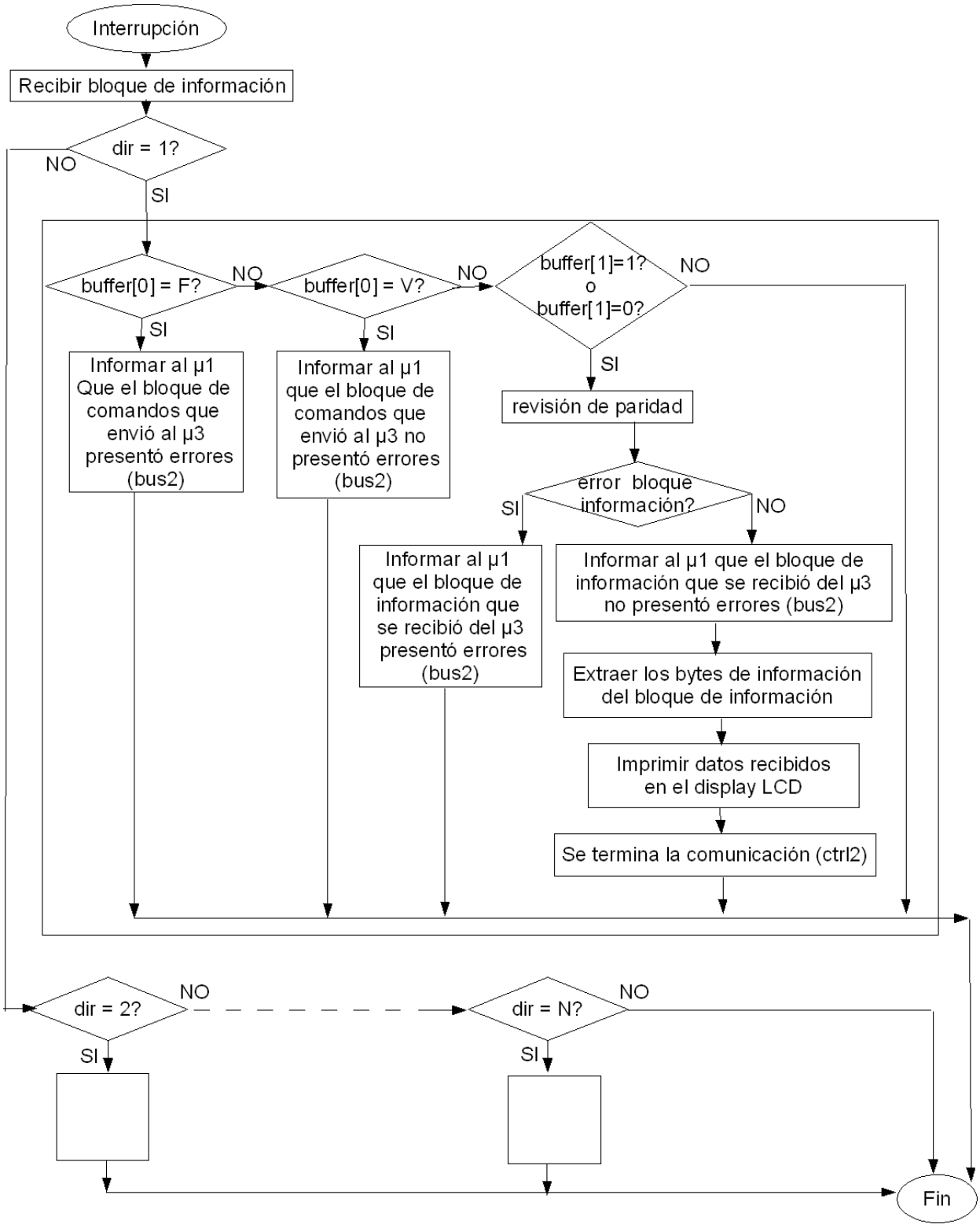


Figura 39: Rutina de interrupción del microcontrolador 2

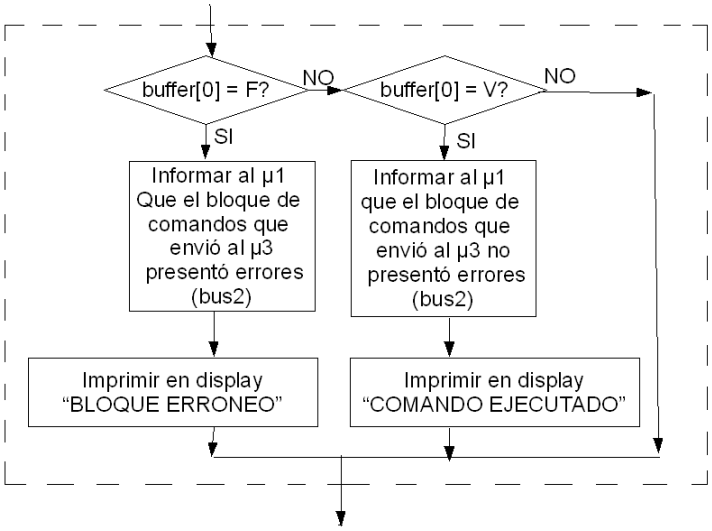


Figura 40: Rutina del esclavo tipo 2 (rutina de interrupción del microcontrolador 2)

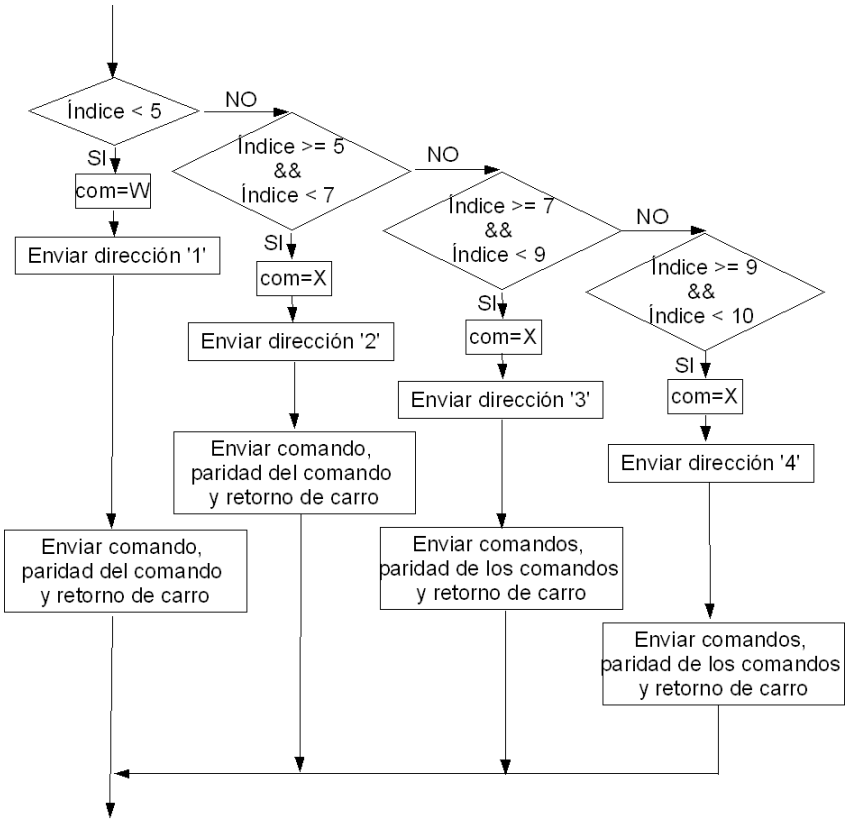


Figura 41: Rutina de envío de bloque de comandos.

3.3 Diseño del esclavo tipo 1

3.3.1 Arquitectura del esclavo tipo 1

En esta parte se describirá la arquitectura del dispositivo esclavo tipo 1. El esclavo tipo 1 es un dispositivo que tiene la función de comunicarse en forma serial utilizando el protocolo RS-232 con un medidor de vacío XGS-600 (VARIAN), de modo que este esclavo tiene que comunicarse, tanto con el dispositivo maestro como con el medidor mencionado, por lo que la arquitectura es similar a la del dispositivo maestro.

La comunicación entre el esclavo tipo 1 y el medidor de vacío utiliza tres líneas del protocolo RS-232, una para transmitir datos, una para recibir datos y la señal de tierra.

En la figura 42 se muestran a los elementos físicos que constituyen al dispositivo esclavo tipo 1, el cual tiene por dirección el carácter ASCII '1'.

A continuación se describirá la función que posee cada uno de sus elementos.

MAX487. Este circuito integrado es un transmisor-receptor 485 que se encarga de realizar la conversión de señales referidas a tierra a señales diferenciales y viceversa.

MAX232. Realiza la conversión de voltajes entre señales RS-232 que provienen del medidor de vacío a señales TTL que son manejadas por los microcontroladores PIC´s.

Microcontrolador 3 (μC 3). Se encarga, de recibir los bloques de comandos o bloques de error que proviene del dispositivo maestro a través de la terminal Rx de su USART, de enviar el comando que indique el dispositivo maestro al medidor de vacío por medio de la terminal Tx de su USART, de controlar el flujo de datos del circuito integrado MAX487 y de enviar información al microcontrolador 4 a través del *bus* 4.

Microcontrolador 4 (μC 4). Tiene la función, de recibir los bytes de información que provienen del medidor de vacío a través de la terminal Rx de su USART, de transmitir bloques de información y bloques de error al dispositivo maestro mediante la terminal Tx de su USART, y de enviar información al μC 3 por medio de las líneas ctrl3 y ctrl4.

Bus 4.Conjunto de 2 líneas que utiliza el μC 3 para informar al μC 4, si el bloque de información que el μC 4 envió al dispositivo maestro presentó o no errores. También se utiliza para informar al μC 4 si el bloque de comandos que se recibió el μC 3 del dispositivo maestro tuvo o no errores.

Ctrl 3. Es una línea por medio de la cual se le indica al μC 3, que el μC 4 ha enviado al dispositivo maestro un bloque de error, esto sirve para que el μC 3 espere un tiempo, antes de que ejecute el comando recibido en el bloque de comandos, en caso de que no hubiesen ocurrido errores en el bloque de comandos.

Ctrl 4. Es una línea que usa el μC 4 para informar al μC 3, que ha enviado o reenviado un bloque de información al dispositivo maestro. Esto sirve para que el μC 3 cambie a modo recibir (antes enviar) la línea de control de flujo de datos, debido a que siempre que se envían bloques de información posteriormente se reciben bloques de error.

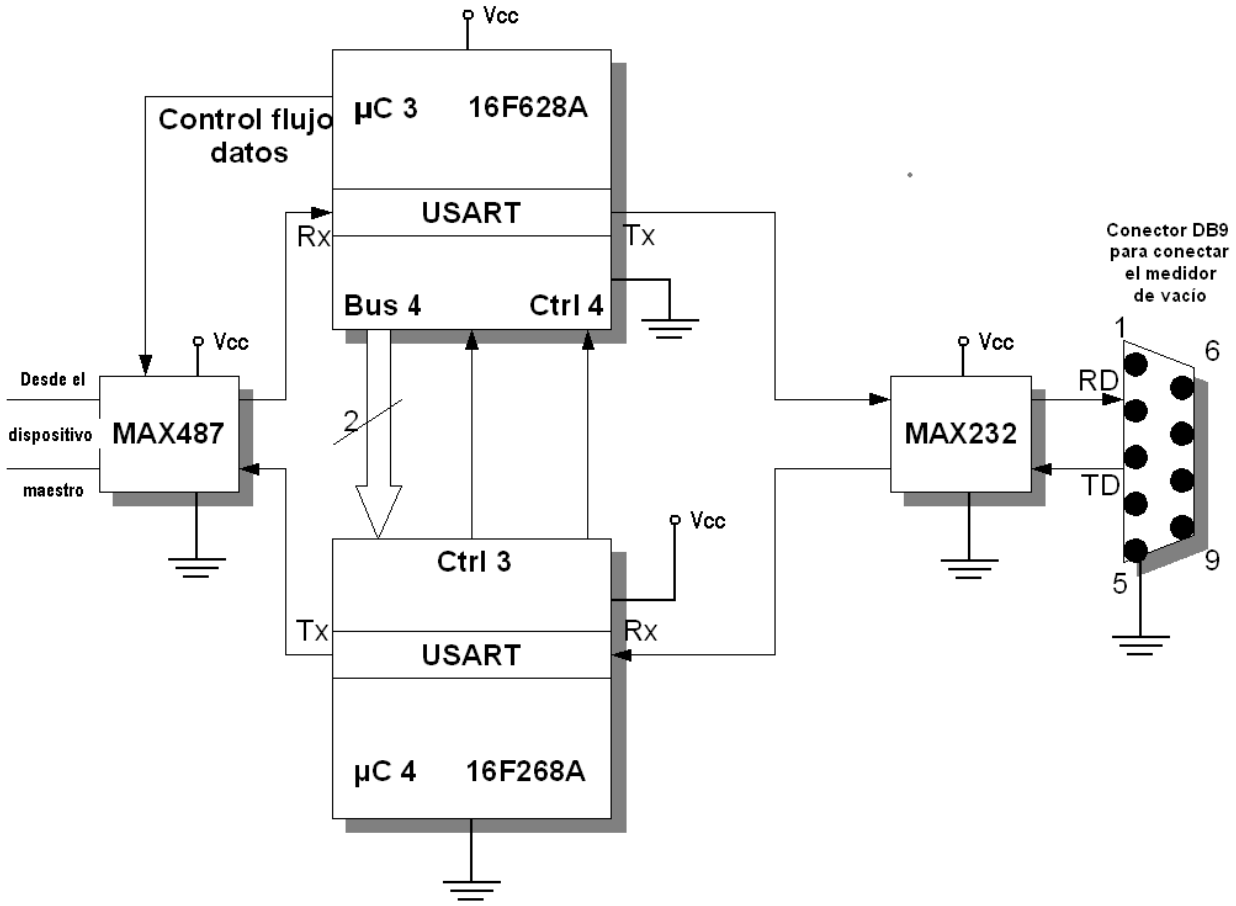


Figura 42: Hardware del esclavo 1

3.3.2 Programación del esclavo tipo 1

Dado que este esclavo está compuesto por dos microcontroladores el proceso de comunicación se distribuye tal y como lo muestran los diagramas de las figuras 43, 44, 45 y 46. Tanto el microcontrolador 3 como el microcontrolador 4 cuentan con un programa principal y una rutina de interrupción.

En el caso del microcontrolador 3 la rutina de interrupción es la que se encarga de llevar a cabo la comunicación, ya que el programa principal (figura 43) además de configurar las terminales de entrada y salida, de habilitar la interrupción por llegada de datos en la USART y de definir las variables a utilizar, sólo se encarga de mantener a la línea de control de flujo de datos (CFD) en modo recibir, para que el esclavo siempre esté dispuesto a recibir bloques de datos, y de sostener al *bus* 4 en el estado binario '11'.

Cada vez que llega un bloque de datos al microcontrolador 3 (figura 44), que puede ser un bloque de comandos o un bloque de error, el microcontrolador se encarga de filtrarlo ya que este bloque llega a todos los esclavos que estén conectados al sistema.

El filtrado consiste en revisar el primer byte que debe contener una dirección, si la dirección coincide con la del esclavo, se procede a revisar el segundo byte, si este corresponde a un ASCII 'F' o un ASCII 'V' se trata de un bloque de error. Cuando lo anterior no sucede se procede a revisar el tercer byte, el cual, si contiene el valor binario 0 o 1 quiere decir que se trata de un bloque de comandos. Una vez que se determina si el bloque es de error o de comandos, el microcontrolador procede a tratar el bloque.

Cuando una comunicación es iniciada con el esclavo tipo 1, una vez que el microcontrolador 3 ha recibido un bloque de comandos, cambia la línea de control de flujo de datos (CFD) a modo enviar para transmitir al dispositivo maestro un bloque de error, después de que se haya realizado la revisión de paridad en el bloque de comandos. Tanto, si encuentra errores, como si no, en el bloque de comandos, el microcontrolador 3 lo debe de informar al microcontrolador 4 por medio del *bus* 4. El *bus* 4 está compuesto por dos líneas y se encuentra por defecto en el estado binario '11', por lo que un cambio al estado '10' y después de regreso al estado '11' indica que hubo errores en el bloque de comandos, mientras que el cambio al estado '01' y luego al estado '11' indica que no hubo errores.

El microcontrolador 4 a través del programa principal (figura 45) recibe por el *bus* 4 la información del estado con que llegó el bloque de comandos, y procede a enviar un bloque de error al dispositivo maestro (microcontrolador 2), con el ASCII 'F' cuando halla algún error y con el ASCII 'V' cuando no. Si hubiesen ocurrido errores en el bloque de comandos, entonces el dispositivo maestro (microcontrolador 1) procedería a reenviar el bloque tantas veces como hubiese errores.

En el momento en que ya no hubiesen errores con el bloque de comandos, lo cual quiere decir que el microcontrolador 4 ha enviado un bloque de error con el ASCII 'V', el microcontrolador 4 lo informa al microcontrolador 3, por medio de la línea ctrl3 con los cambios binarios '1', '0', '1', para que el microcontrolador 3 espere algunos milisegundos a que el microcontrolador 1 se informe del contenido del bloque de error, antes de ejecutar el comando contenido en el bloque de comandos que lo lleve a decidir que cadena de bytes se enviará al medidor de vacío. Si el comando corresponde a una ASCII ' 1' la cadena enviada son los ASCII ´s, '#','0','0','0','1', para cuando el ASCII es un '2' la cadena es, '#','0','0','0','5', para el ASCII '3' la cadena es, '#','0','0','0','5' y para cuando el ASCII es un '4' la cadena enviada es, '#','0','0','0','2','I','1'. Una vez que el medidor de vacío procesa la cadena recibida, procede a responder con una cadena de bytes, la cual es recibida por el microcontrolador 4 a través de su rutina de interrupción (figura 46). Una vez que la cadena es recibida, el microcontrolador 4 la procesa para ajustarla a la estructura que posee un bloque de información (figura 29), que será enviado al dispositivo maestro (microcontrolador 2).

Después de que es enviado el bloque de información, el microcontrolador 4 lo informa al microcontrolador 3 por la línea ctrl4 con los cambios binarios '1' ,'0', '1', esto se realiza para que el microcontrolador 3 cambie a modo recibir la línea de control de flujo de datos y finalice la rutina de interrupción actual, para que el microcontrolador 3 vuelva a interrumpirse con la llegada de un bloque de error (figura 44) que le indique el estado en que llegó el bloque de información. Cuando el bloque de error llega al microcontrolador 3, este informa el contenido al microcontrolador 4 por medio del *bus* 4. Si el carácter ASCII que contiene el bloque de error es una 'F' ocurrieron errores y si es el carácter 'V' no los hubo. Por tanto, en el *bus* 4 se presenta los

cambios binarios '11', '10', '11', cuando hubo errores y los cambios binarios '11', '01', '11', cuando no los hubo. Cuando hubiesen ocurrido errores, el microcontrolador 4 reenvía el bloque de información y procede a informarse por medio del *bus* 4, del estado de este reenvió, de modo que este proceso se repetiría mientras ocurriesen errores. En el momento en que ya no ocurriesen errores el proceso de comunicación finalizaría.

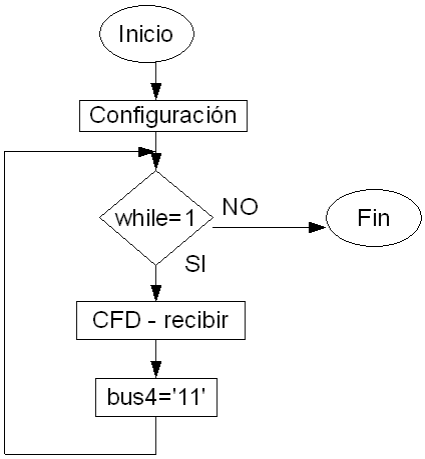


Figura 43: Programa principal del microcontrolador 3

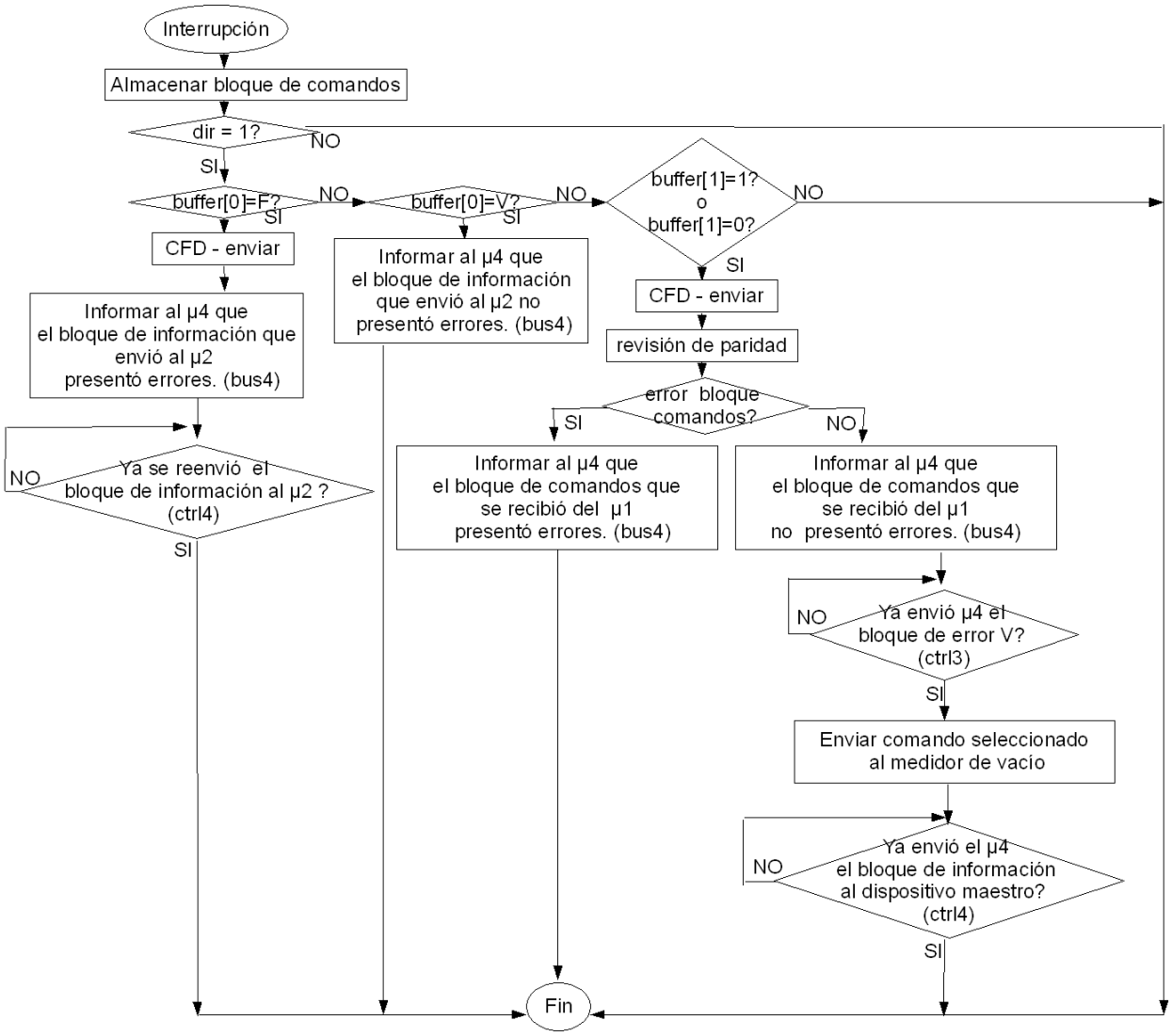


Figura 44: Rutina de interrupción del microcontrolador 3

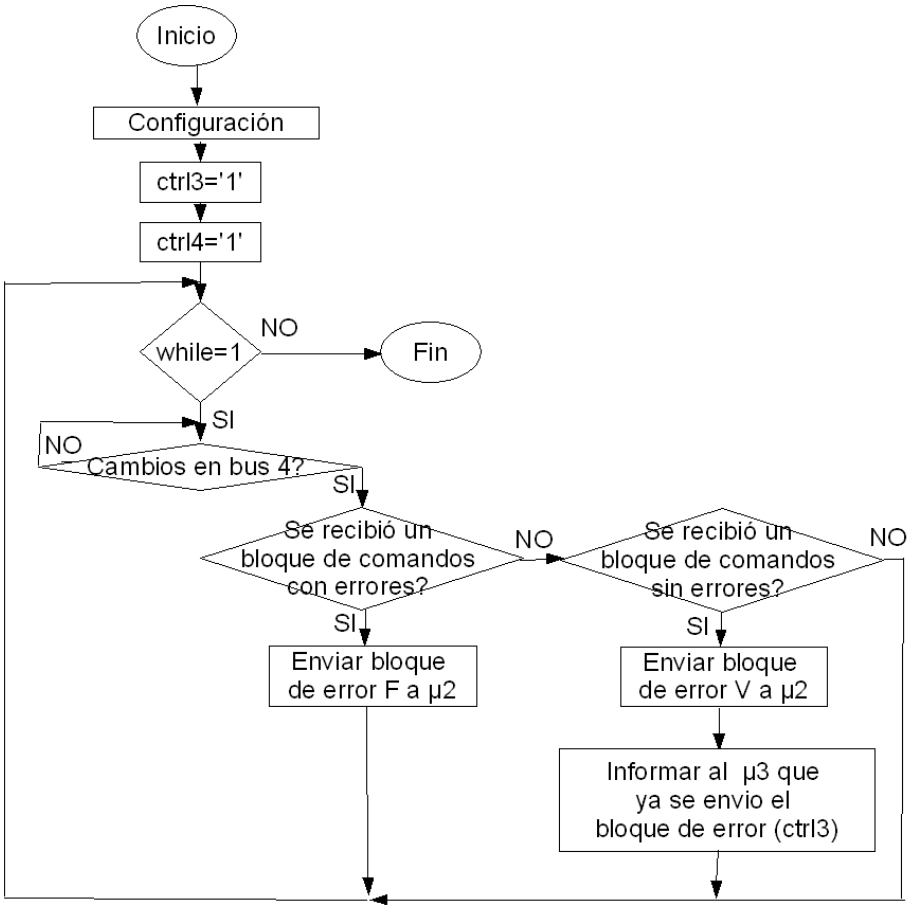


Figura 45: Programa principal del microcontrolador 4

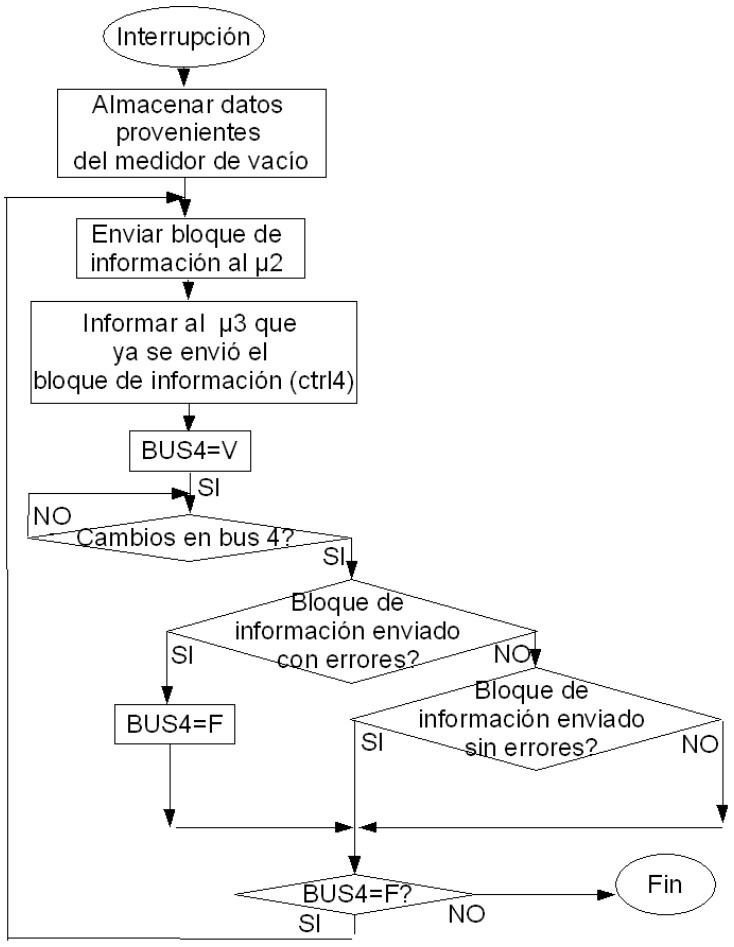


Figura 46: Rutina de interrupción del microcontrolador 4

3.3 Diseño del esclavo tipo 2

3.3.1 Arquitectura del esclavo tipo 2

En esta parte se tratará el diseño de un esclavo (figura 47) que mantiene una comunicación simple con el dispositivo maestro, el cual tiene la función de encender o apagar un LED, lo cual puede ser fácilmente aplicable a la apertura o cierre de una válvula de dos estados.

Este esclavo se constituye de los siguientes elementos:

MAX487. Este circuito integrado es un transmisor-receptor 485 que realiza la conversión de señales referidas a tierra a señales diferenciales y viceversa.

Microcontrolador 5 (µC 5). Se encarga de recibir los bloques de comandos que provienen del dispositivo maestro a través de la terminal Rx de la USART, de enviar bloques de error al

dispositivo maestro por la terminal Tx de la USART, de controlar el flujo de datos (CFD) del circuito integrado MAX487 y de ejecutar los comandos recibidos.

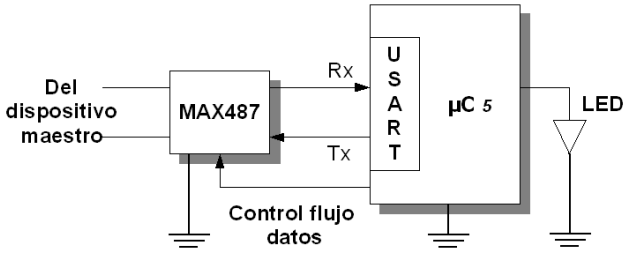


Figura 47: Arquitectura del esclavo tipo 2

3.3.2 Programación del esclavo tipo 2

El control de la comunicación en este esclavo es llevado a cabo por un programa principal y por una rutina de interrupción como se muestran en la figuras 48 y 49.

El programa principal (figura 48) se encarga, de configurar las terminales del microcontrolador 5 que van utilizarse como entradas y como salidas, de activar la interrupción por recepción de datos en la USART, de definir las variables a utilizar y de mantener a la línea de control de flujo de datos (CFD) en modo recibir para que el esclavo pueda recibir bloques de comandos en cualquier momento.

La rutina de interrupción (figura 49) se encarga de recibir los bloques de comandos que provienen del dispositivo maestro. Una vez que se ha recibido un bloque de comandos, el microcontrolador 5 verifica que el primer byte corresponda a la dirección que tiene asignada este esclavo, que es el ASCII '2', si la dirección corresponde, procede a revisar si en el tercer byte (byte de paridad) hay un valor binario '1' ó '0', si esto ocurre, entonces el microcontrolador 5 procede a cambiar la línea de control de flujo de datos (CFD) a modo enviar para poder transmitir al dispositivo maestro un bloque de error. La revisión del tercer byte, que siempre corresponde a un byte de paridad, se realiza para comprobar que efectivamente se trata de un bloque de comandos válido. El contenido del bloque de error es producto de la revisión de paridad que se realiza en el bloque de comandos, si hubo errores entonces el bloque de error enviado al dispositivo maestro contiene un carácter ASCII 'F', esto ocasionará que el dispositivo maestro reenvíe el bloque de comandos al esclavo, tantas veces como este presentase errores. Cuando el bloque de error enviado al dispositivo maestro contiene el ASCII 'V', entonces el microcontrolador 5 procede a ejecutar el comando contenido en el bloque de comandos el cual corresponde al segundo byte. Si el comando corresponde al carácter ASCII '5' entonces el microcontrolador 5 enciende el LED que tiene conectado y si es el carácter ASCII '6' entonces apaga el LED, finalizando con esto la rutina de interrupción. Cuando el programa principal toma el control de las tareas, nuevamente coloca a la línea de control de flujo de datos en modo recibir, para que posteriormente se pueda recibir por interrupción otro bloque de comandos.

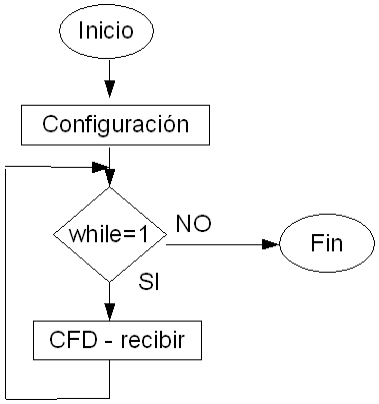


Figura 48: Programa principal del esclavo tipo 2

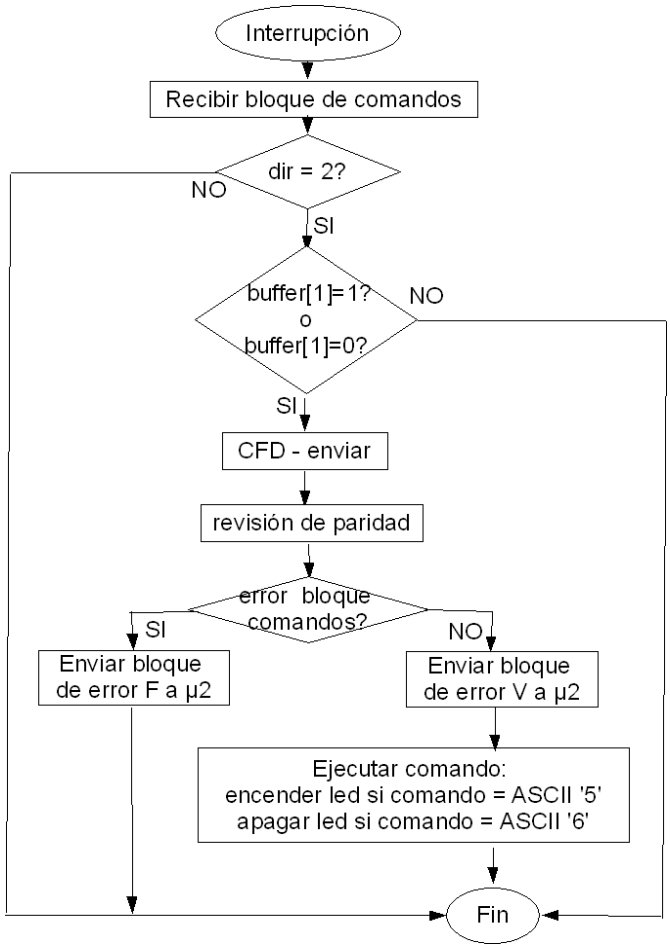


Figura 49: Rutina de interrupción del esclavo tipo 2

4. LabVIEW

4.1 Introducción

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) es un entorno de programación gráfico, que sirve para realizar pruebas y mediciones experimentales, automatización de procesos, control instrumental, adquisición de datos y análisis de datos.

De acuerdo a las funcionalidades que ofrece este lenguaje de programación se eligió como la interfaz gráfica de usuario que permitiera la interacción con los dispositivos esclavos del sistema. De este modo la PC funciona como un dispositivo de entrada-salida que tiene comunicación con el dispositivo maestro.

Al igual que cualquier lenguaje de programación, éste posee las siguientes estructuras para controlar el flujo del programa: *while*, *if*, *for*, *case*, además de algunas otras, propias de LabVIEW para controlar el orden en que se ejecutan las instrucciones. A este lenguaje por ser de modo gráfico se le ha denominado lenguaje G.

4.2 VI´s

A los elementos básicos de programación en LabVIEW se les llama VI´s (de *Virtual Instruments*), los cuales pueden contener desde tipos de datos primarios y funciones, hasta otros VI´s. Un VI se compone de un panel frontal y un diagrama de bloques. En el panel frontal se encuentra el entorno gráfico, que es en donde se realiza la interacción con el usuario ya que se pueden introducir datos o visualizar información de algún proceso en curso. En el diagrama de bloques se encuentra el código fuente del VI.

En las figura 50 y 51 se muestra un ejemplo de un VI llamado suma, en donde se puede observar el panel frontal y el diagrama de bloques respectivamente. La función de este programa

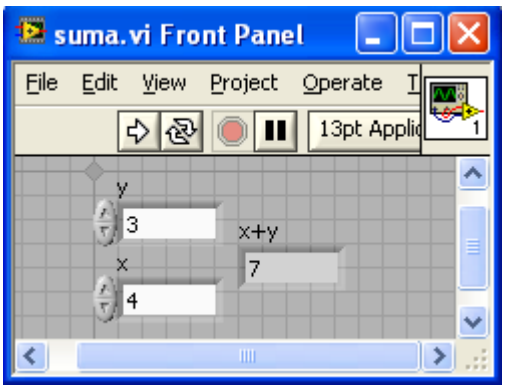


Figura 50: Panel frontal de un VI

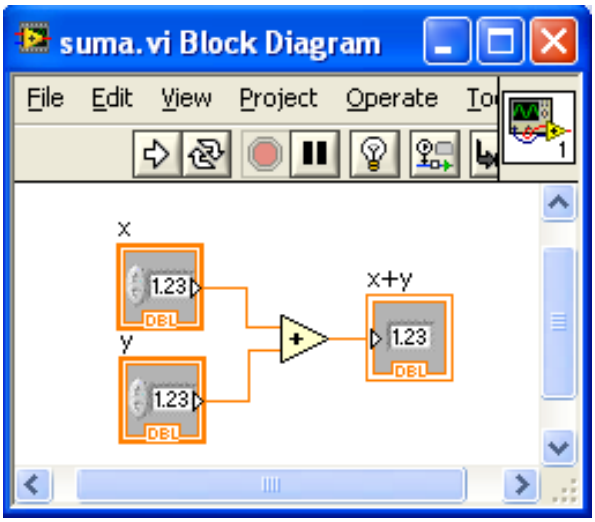


Figura 51: Diagrama de bloques de un VI

es realizar la suma numérica de las variables *x* y *y*, y mostrar el resultado en la variable *x+ y*. Se puede ver que las variables *x* y *y* son las que el usuario manipula en el panel frontal y que el código fuente se encuentra en el diagrama de bloques, como se mencionó antes.

4.3 Manejo del puerto serial

Para el manejo del puerto serial, LabVIEW ya cuenta con un conjunto de VI´s que permiten manipular la entrada o salida de información serial a través de un puerto serial RS-232. Los VIs que se utilizan son los siguientes: VISA Configure Serial Port VI, VISA Write, VISA Read y VISA Close. A continuación se describe la función de cada uno de ellos.

El VI VISA Configure Serial Port VI se encarga de la configuración del puerto serial RS-232 específico. En la figura 52 se muestran las entradas y salidas de este VI de las cuales, las que se utilizan para este proyecto son las siguientes:

- Baud rate.- Es la velocidad con la que se transmiten los datos, la cual debe de coincidir con la velocidad configurada en el microcontrolador. En este proyecto la velocidad elegida es de 9600 bps.
- Data bits.- Es el número de bits que se transmiten por bloques de datos, el cual puede variar de 5 a 8 bits. El valor utilizado en este trabajo es de 8 bits ya que es el tamaño que puede procesar el módulo USART de los PIC´s.
- Flow control.- Se encarga de controlar el flujo de datos. Debido a que en este caso no hay líneas de control de datos, la opción asignado a esta entrada es "*none"*.
- Parity.- Especifica la paridad utilizada en los bloques de datos. Dado que la distancia entre entre la PC y el dispositivo maestro es muy corta, en este caso no se implementó la verificación de la paridad ya que es muy improbable que se produzcan errores en los datos. Debido a esto, la opción seleccionada en esta entrada es "*none"*.
- Stop bits.- Corresponde al número de bits utilizados al terminar la transmisión de cada bloque de datos. En este caso solamente se utiliza un bit de stop, por lo que la opción elegida en esta entrada es "1.0" que equivale a 1 bit de stop.
- VISA resource name.- Especifica el puerto serial (puertos COM) con el que se va a trabajar, en donde este puede ser físico o virtual.
- VISA resource name out.- Es una copia de la entrada VISA resource name, la cual pone a disposición del resto del programa el puerto serial elegido.
- Error in.- Esta entrada recibe los errores que ocurrieron antes de que se ejecutara este VI.
- Error out.- Es una salida que contiene los posibles errores que se hayan presentado en el VI.

El VI VISA Write mostrado en la figura 53 se encarga de escribir en forma serial, los datos que se van a transmitir al buffer del puerto serial, mediante la entrada *write buffer.* La entrada *error in* se encarga de verificar que no haya errores antes de que se ejecute este VI y las entradas *VISA resource name* y *error in,* y las salidas *VISA resource name out* y *error out,*

realizan la misma función que en el VI de configuración. La salida *return count* muestra el número de bytes que se han escrito en el buffer del puerto serial.

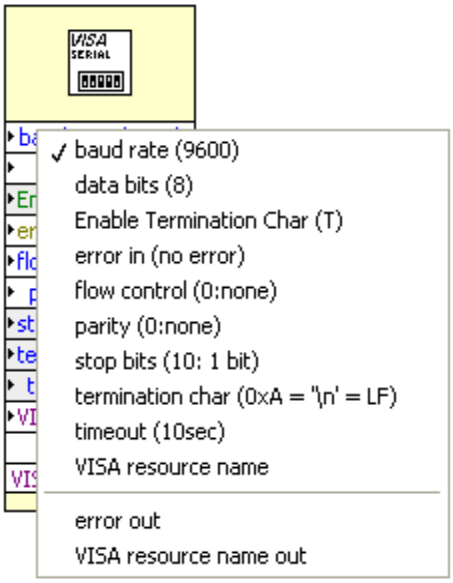


Figura 52: VI para la configuración del puerto serial

El VI VISA Read mostrado también en la figura 53 se encarga de leer los bytes provenientes de la interfaz serial, y mostrarlos como cadenas ASCII a través de la salida *read buffer*. La entrada *byte count* corresponde a un número que determina la cantidad de bytes máxima que se van a leer del buffer de entrada. La salida *return count* muestra la cantidad de bytes que han sido leídos. Las demás variables de entrada y salida tienen la misma funcionalidad que en el VI de escritura.

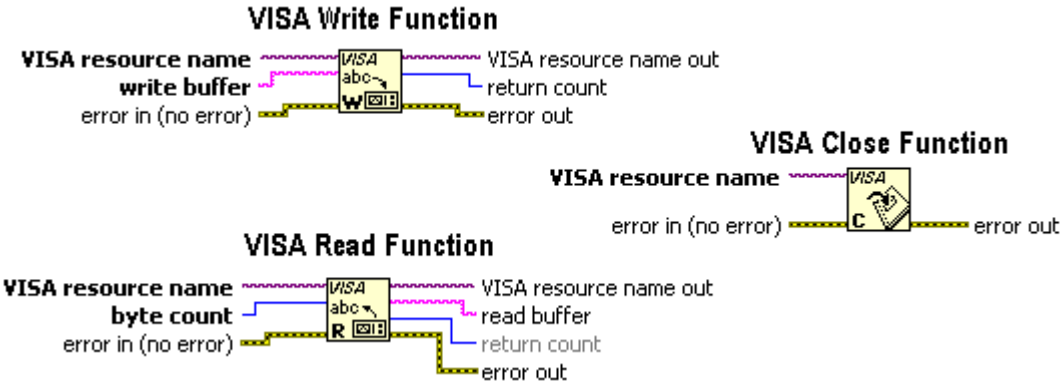


Figura 53: VI write, VI read y VI close

El VI VISA Close mostrado en la figura 53 se encarga de terminar la sesión con la interfaz serial que fue habilitada.

4.4 Comunicación con el dispositivo maestro

Una vez que se han descrito los VI´s correspondientes al manejo del puerto serial. Ahora corresponde revisar a los VI´s que se encargan de procesar los bloques de datos que viajan entre el dispositivo maestro y la PC. Como se definió en el protocolo de comunicación los bloques de datos que son enviados y recibidos entre la PC y el dispositivo maestro son como los que se muestran en la figura 54. Cada esclavo con el que se establece comunicación, tiene asociado uno o dos VI, dependiendo del tipo de comunicación que se establezca (simple o compuesta). Estos VI´s se encargan de procesar los dos tipos de bloques de datos de la figura 54.

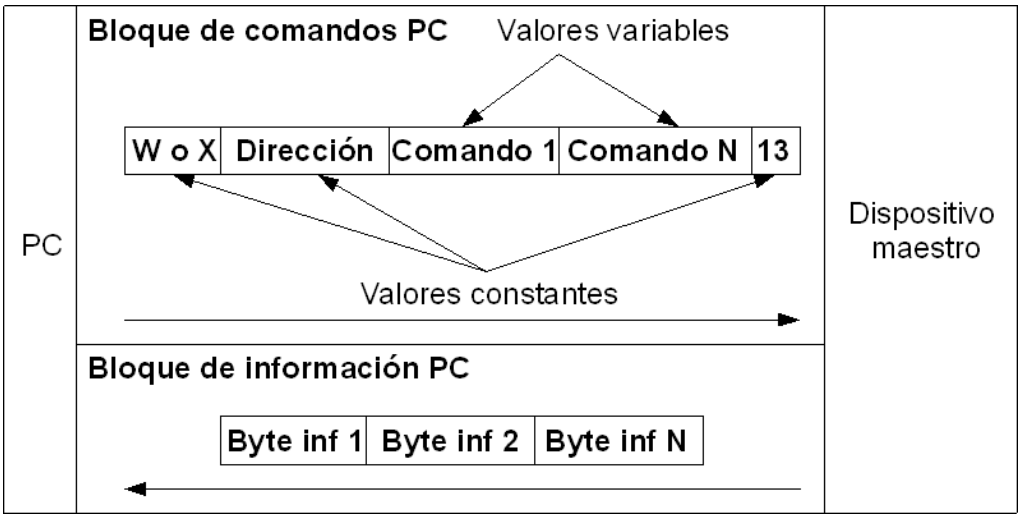


Figura 54: Bloques de datos que transitan entre la PC y el dispositivo maestro

En las figuras 55 y 56 se muestran los dos VI´s que se encargan de procesar los bloques de comandos PC y los bloques de información PC.

El VI de la figura 55 es el que se encarga de formar el bloque de datos PC que se enviará al dispositivo maestro. Se puede observar que posee cinco entradas (W o X, actualizar proceso, dirección, comando 1 y comando 2) y una salida (cadena). Los valores de las entradas *W o X* y *dirección* son definidos en el momento en que se elabora el diagrama de bloques (código fuente). Así, la entrada *W o X* es el primer byte, que define el tipo de comunicación que se realizará con algún esclavo, el cual puede tomar dos valores; el ASCII 'X' cuando se trata de una comunicación simple o el ASCII 'W' cuando corresponde a una comunicación compuesta. La entrada *dirección* (segundo byte) puede tomar un valor numérico entre 1 y 32, cuyo valor concuerda con la dirección del esclavo con el que se quiere establecer comunicación. El último elemento de la cadena que se envía al dispositivo maestro corresponde a un byte que contiene el carácter ASCII '13' (retorno de carro), el cual es agregado internamente por el VI de la figura 55. De este modo, una vez que estas entradas toman un valor en el momento en que se elabora el código fuente, no pueden cambiar su valor en tiempo de ejecución, por lo que se consideran entradas de valores constantes.

Las entradas *comando 1* y *comando 2* (fig 55) son entradas que pueden cambiar su valor en tiempo de ejecución ya que están ligadas a botones que son manipulados por el usuario. El VI de la figura 55 puede variar en el número de entradas *comando*, ya que la comunicación con un esclavo puede requerir que se le envíen más de dos comandos como ocurre con este VI. Las entradas *comando* son del tamaño de un byte, por lo que pueden tomar valores que van desde 0 hasta 255.

La entrada *actualizar proceso* está unida a un botón binario, el cual cada vez que es oprimido por el usuario, genera un bloque de comandos PC en forma serial en la variable de salida *cadena* con el formato de la figura 54 y es enviada por el puerto RS-232 hacia el dispositivo maestro.

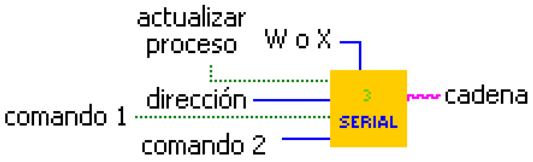


Figura 55: VI Formador de cadenas de datos

El VI de la figura 56 es utilizado junto con el VI que forma los bloques de comandos PC que se envían al maestro, cuando la comunicación es compuesta (cuando se solicita información de los dispositivos esclavos). La función que realiza este VI es tomar los bytes (bloque de información PC) que se encuentran en el buffer de la interfaz serial RS-232, que proceden del dispositivo maestro, para entregarlos en forma serial a través de la variable de salida *datos*. El formato en que son entregados los datos (bytes) en la variable de salida *datos* es siempre como código ASCII, por lo que si no se desea desplegarlos de esta forma en el panel frontal de LabVIEW, es necesario procesarlos, por ejemplo, cuando se requiera visualizarlos en una gráfica.

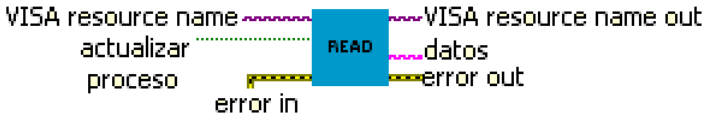


Figura 56: VI Lector de cadenas de datos

4.5 Diagrama de flujo de la interfaz gráfica

En la figura 57 se muestra el diagrama de flujo de la interfaz gráfica elaborada en LabVIEW, en él se muestra que después de que se ha configurado el puerto serial por el usuario, el programa entra en un bucle en donde se revisa continuamente cuál proceso es verdadero para ejecutarlo. Cada proceso está asociado con un sólo dispositivo esclavo. Como ya se había

mencionado, las variables que modifica el usuario en el panel frontal corresponden al valor deseado en cada uno de los comandos asociados con cada esclavo. En el momento que el usuario oprime el botón actualizar en el panel frontal mostrado en la figura 52, un bloque de datos PC es enviado al dispositivo maestro tal y como se indica en el diagrama de flujo. Cuando la comunicación es simple (se requiere actualizar el estado de algún dispositivo esclavo) solamente se envía el bloque de datos PC y el proceso termina, en cambio, si se trata de una comunicación compuesta (se solicita información a algún dispositivo esclavo), el bloque de información PC que provienen del dispositivo maestro es almacenado en el buffer de la interfaz serial RS-232 para posteriormente procesarlo, y finalizar con esto el proceso actual, para volver a revisar que otro proceso se presenta como verdadero.

En la figura 58 se muestra el panel frontal para comunicarse con tres esclavos diferentes. En la parte izquierda se pueden observar los botones mediante los que se configura la interfaz serial. En la parte derecha se encuentran los módulos asociados a cada esclavo a los que se les ha denominado procesos. Los proceso 1 y 2 se tratan de comunicaciones compuestas ya que se envían y reciben bloques de datos, y el proceso 3 se trata de una comunicación simple ya que solamente envía bloques de datos. En cada proceso pueden observarse los botones que son utilizados, para modificar el valor de los comandos (*comando*, *on/off*) o para comenzar el envío de información (*actualizar*). Así también pueden observarse cajas llamadas *datos* en las que se despliega la información que es solicitada a los dispositivos esclavo.

De los tres procesos que pueden visualizarse en la figura 58, el proceso 2 es el que está asociado al dispositivo esclavo 1 que se desarrolló en el capítulo 3 y el proceso 3 el que está asociado al esclavo tipo 2. En cambio, el proceso 1 sólo muestra un ejemplo de como un tercer esclavo podría realizar dos procesos a la vez, entregar la información que le sea solicitada como el proceso 1 y encender y apagar un LED como en el proceso 3.

Por último en la figura 59 se muestra el diagrama de bloques de la interfaz gráfica, el cual corresponde al código fuente del programa en LabVIEW.

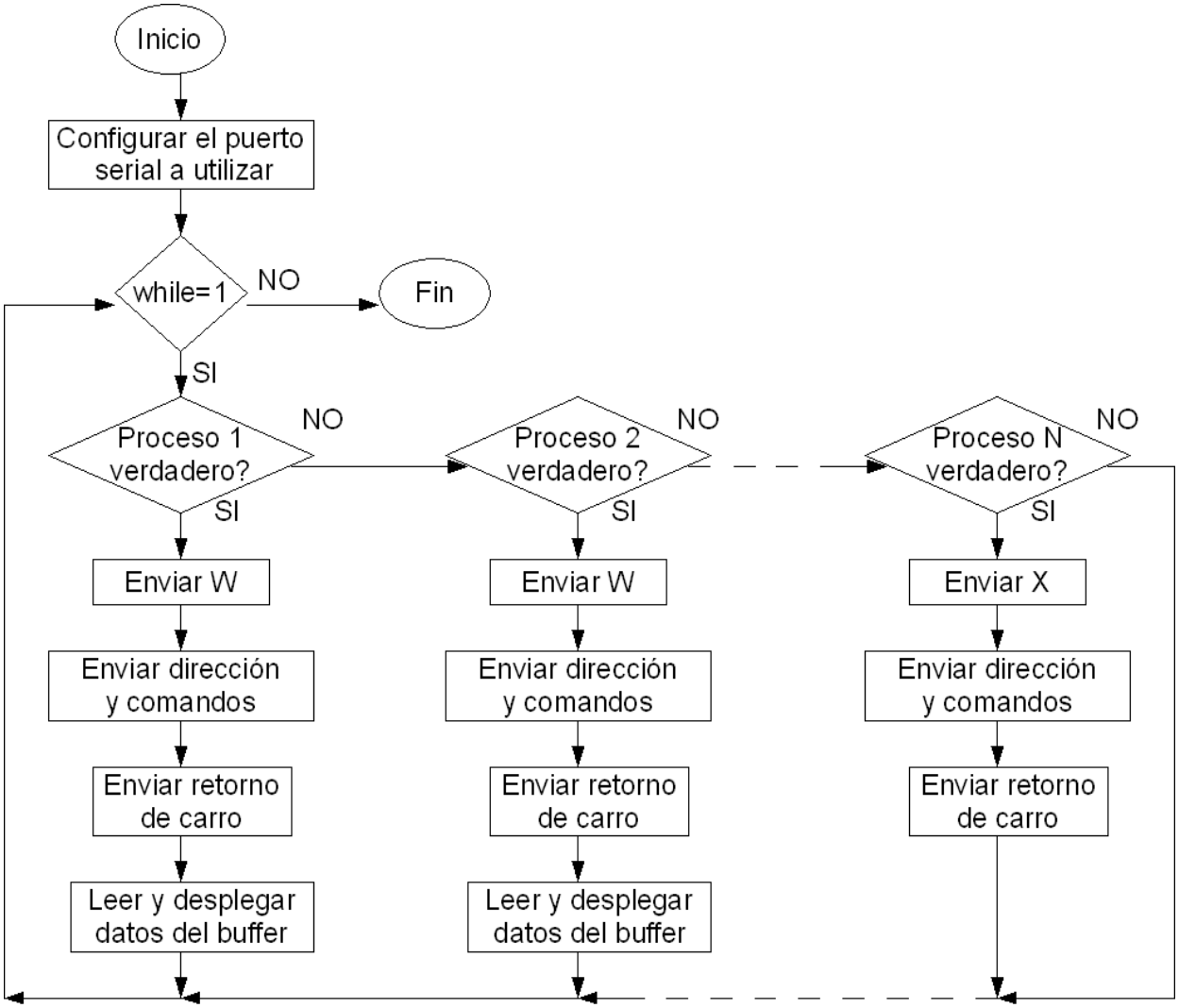


Figura 57: Diagrama de la interfaz gráfica en LabVIEW

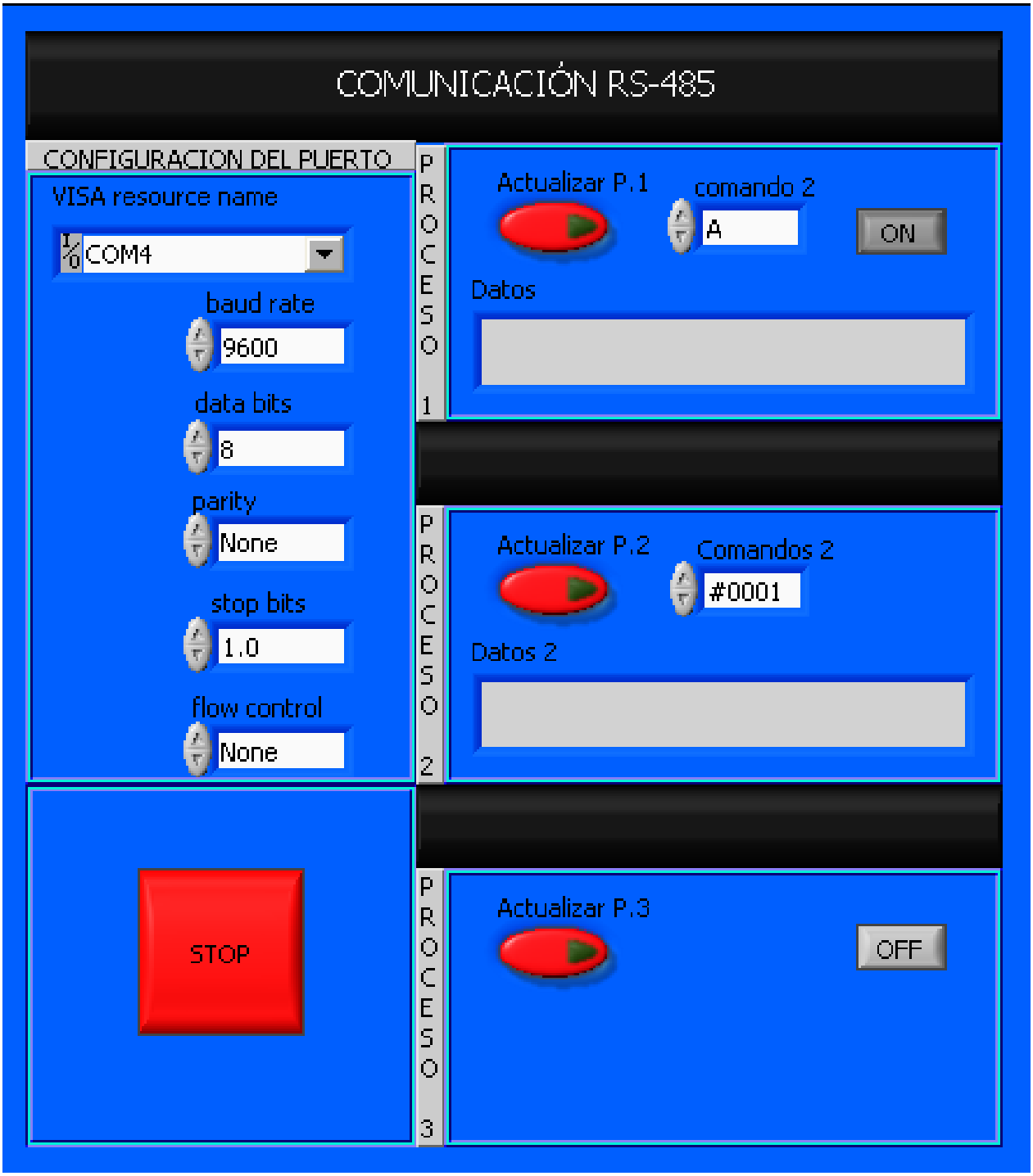


Figura 58: Panel frontal de la interfaz gráfica

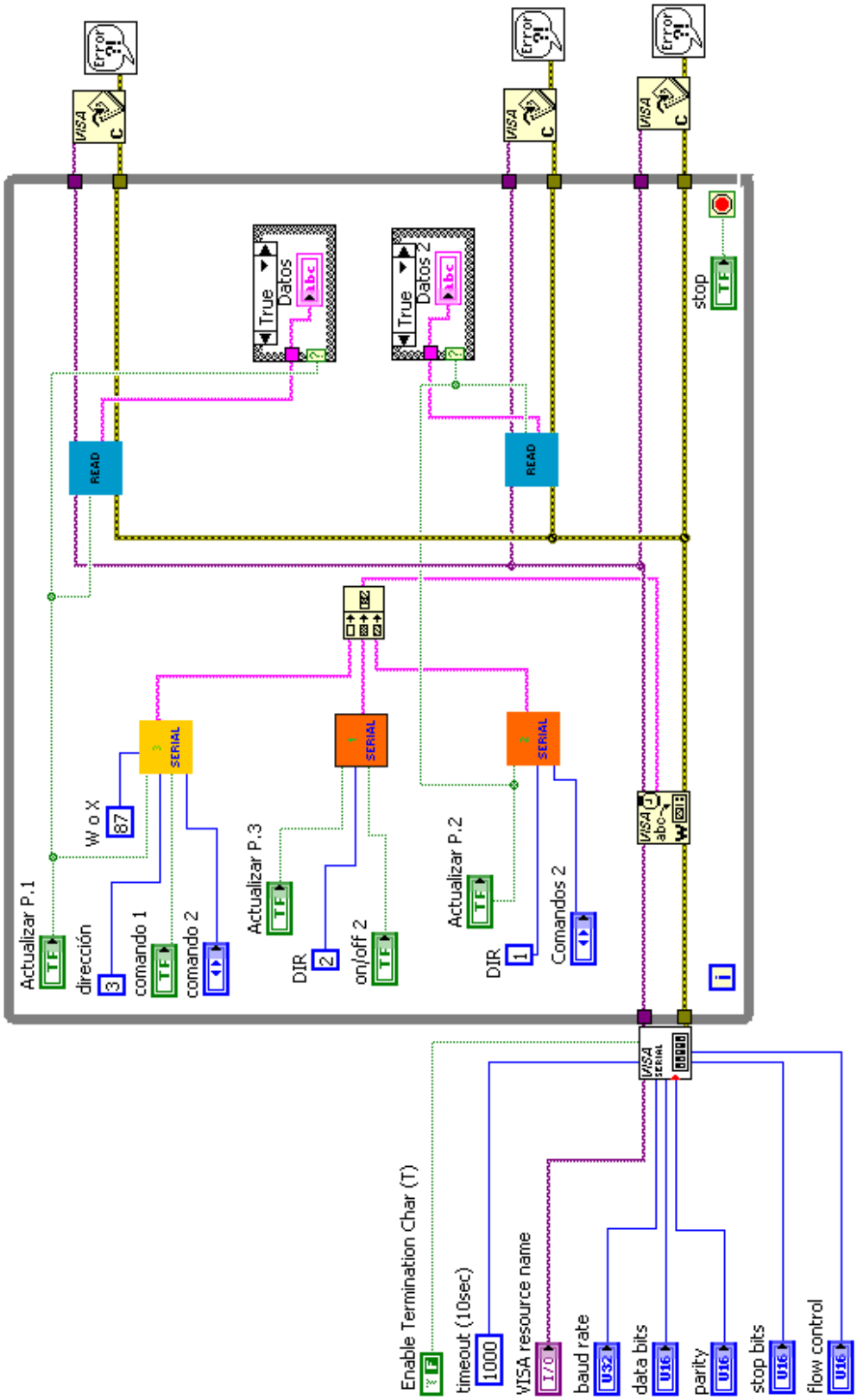


Figura 59: Diagrama de bloques de la interfaz gráfica

5. PRUEBAS Y RESULTADOS

En este capítulo se presentarán las pruebas y resultados que se llevaron a cabo con el sistema de comunicación. Este proceso consistió de dos etapas, en la primera se realizaron pruebas mediante el simulador PROTEUS de Labcenter Electronics con el dispositivo maestro y los esclavos tipo 1 y tipo 2, una vez que se hubo comprobado que el sistema respondía de manera correcta en la simulación, se procedió a ejecutar la segunda etapa, que consistió en implementar de forma física al dispositivo maestro y al esclavo tipo 1 para realizar pruebas con ellos.

SIMULACIÓN

Prueba 1

El esclavo tipo 1 es un dispositivo al que se le conecta un medidor de vacío, este medidor en su manual de operación especifica tres comandos, que se le deben de enviar para comprobar que la comunicación que se realiza con él se lleva a cabo de manera exitosa. Las respuestas del medidor de vacío a cada uno de estos comandos se muestran en la tabla 2.

Comando seleccionado en el dispositivo maestro	Comandos programado en el esclavo tipo 1 (ASCII)	Respuestas del medidor de vacío (ASCII)
Comando 1	#0001	>FE10FE10FE40
Comando 2	#0005	>0201,0150,0150,0150
Comando 3	#000F	>0.000E+00,0.000E+00,OPEN ,OPEN

Tabla 2: Comandos de prueba para el medidor de vacío

Cada uno de los comandos que se envían al medidor de vacío, así como sus respuestas se programaron en el esclavo tipo 1 para simular una conexión con el medidor. Una vez tenido el diagrama del sistema en PROTEUS con las respectivas programaciones del dispositivo maestro, el esclavo tipo 1 y el esclavo tipo 2, se procedió a ejecutar la simulación y a solicitar información al esclavo tipo 1 con el botón de envío, una vez que se hubo seleccionado este esclavo y el comando 1, 2 ó 3 con el botón de comandos (figura 60). Las respuestas obtenidas (para cada uno de los tres comandos) mostradas en la figura 60 comprueban que no hubo ningún error al establecer comunicación con este esclavo.

Una prueba similar se realizó con el esclavo tipo 2. A este esclavo están asociados dos comandos, mostrados en la tabla 3. En simulación, cuando en el botón de comandos es seleccionado el comando 5 el esclavo 2 debe de encender el LED, además de mostrar en el display el mensaje “BLOQUE EJECUTADO” para comprobar que la comunicación se ha llevado de manera correcta. La respuesta obtenida se muestra en la figura 61, en ella se muestra que en el display aparece el mensaje mencionado anteriormente, comprobándose nuevamente con esto; el buen funcionamiento del sistema. Cabe mencionar que, el mismo mensaje aparece en el display cuando se ejecuta el comando 6.

5. PRUEBAS Y RESULTADOS

Comando seleccionado en el dispositivo maestro	Acción del esclavo tipo 2
Comando 5	Encender un LED
Comando 6	Apagar un LED

Tabla 3: Comandos asociados al esclavo tipo 2

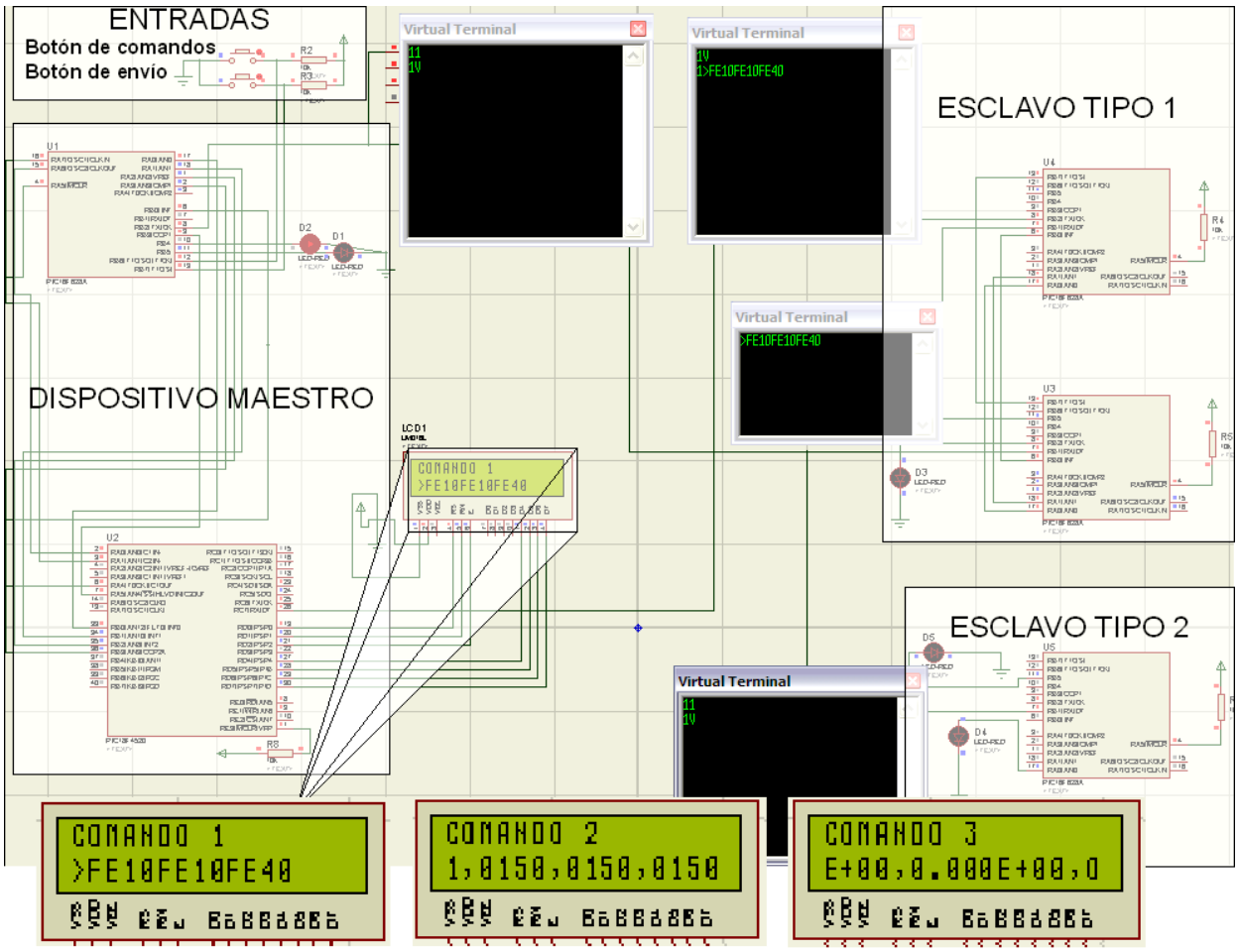


Figura 60: Prueba 1 con el esclavo tipo 1

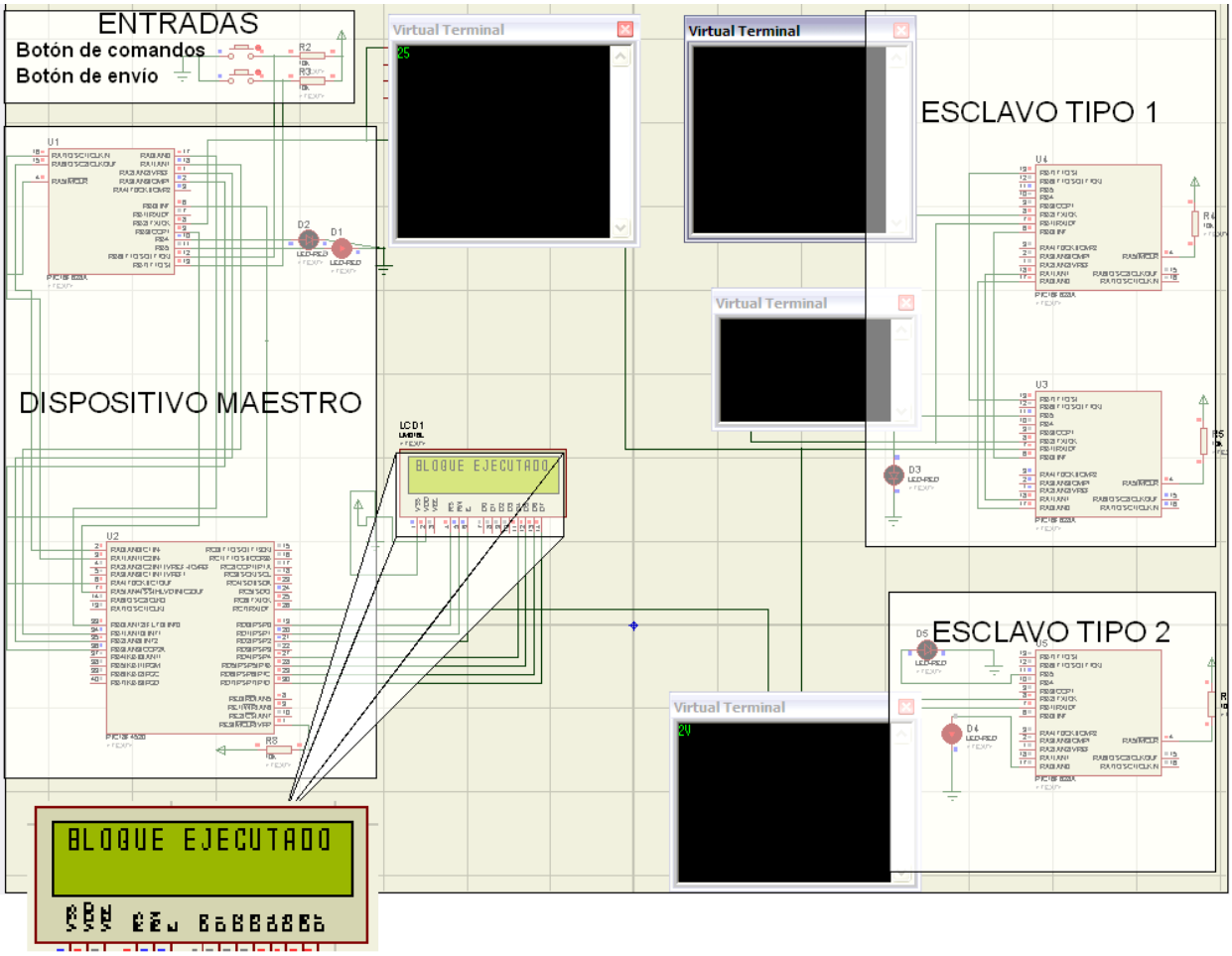


Figura 61: Prueba 1 con el esclavo tipo 2

Prueba 2

En estas pruebas se comprobó lo que se planteó en el capítulo 3, en donde se estableció que si ocurría un error durante la transmisión de un bloque de comandos, el maestro tendría que reenviar este bloque tantas veces como se presentaran errores. También, si un error ocurría en el bloque de información que el esclavo tipo 1 transmite al maestro, éste tendría que reenviar este bloque tantas veces como los errores se presentarán.

Para el esclavo tipo 1 estos errores pueden ocurrir tanto cuando recibe un bloque de comandos como cuando le envía un bloque de información al dispositivo maestro. En las figuras 62 y 63 se muestran los resultados al simular errores en estos bloques.

En la figura 62 puede verse que para cada bloque de comandos con el formato “ASCII '1', ASCII '2', byte de paridad, retorno de carro” que es enviado al esclavo tipo 1, éste realiza la revisión de paridad. Una vez que el esclavo comprueba que ha ocurrido un error, éste le envía al maestro un bloque de error con el formato “ASCII '1', ASCII 'F', retorno de carro” para indicarle que reenvíe el bloque de comandos. Como en esta simulación el error permanece siempre, el reenvío del bloque de comandos ocurre indefinidamente, pero en un caso real, una vez que ya no

hubiese errores el esclavo tipo 1 procedería a ejecutar su rutina siguiente que es, enviarle un bloque de información al maestro.

Cuando el error ocurre en el bloque de información que el esclavo envía al maestro, es el maestro el que solicita el reenvío de este bloque mediante un bloque de error con el mismo formato que en el caso anterior. Esto es mostrado en la simulación de la figura 63, en donde el reenvío del bloque de información también ocurre de manera indefinida debido a que el error siempre está presente.

En la figura 64 se muestra la simulación para cuando el bloque de comando que se le envía al esclavo tipo 2 tiene un error. Se observa entonces que el maestro, tiene que reenviar el bloque de comandos “ASCII '2', ASCII '5', byte de paridad, retorno de carro” al esclavo, porque el error se mantiene.

Con esto se comprueba el buen funcionamiento que tiene la revisión de paridad en el sistema ya que se si presentase un error en un bloque de comandos o un bloque de información, éstos serían reenviados.

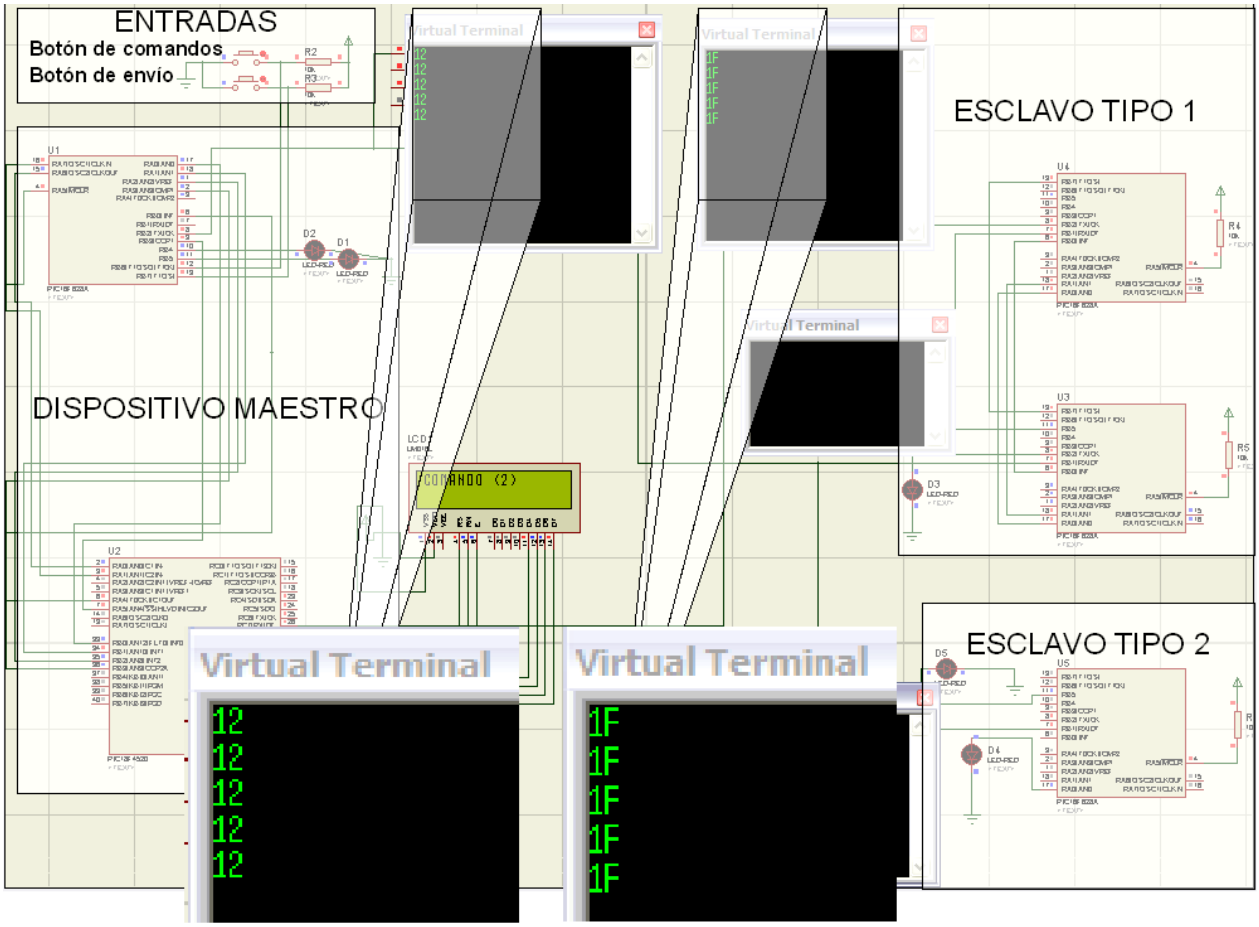


Figura 62: Prueba 2, error en bloque de comandos enviado al esclavo tipo 1

5. PRUEBAS Y RESULTADOS

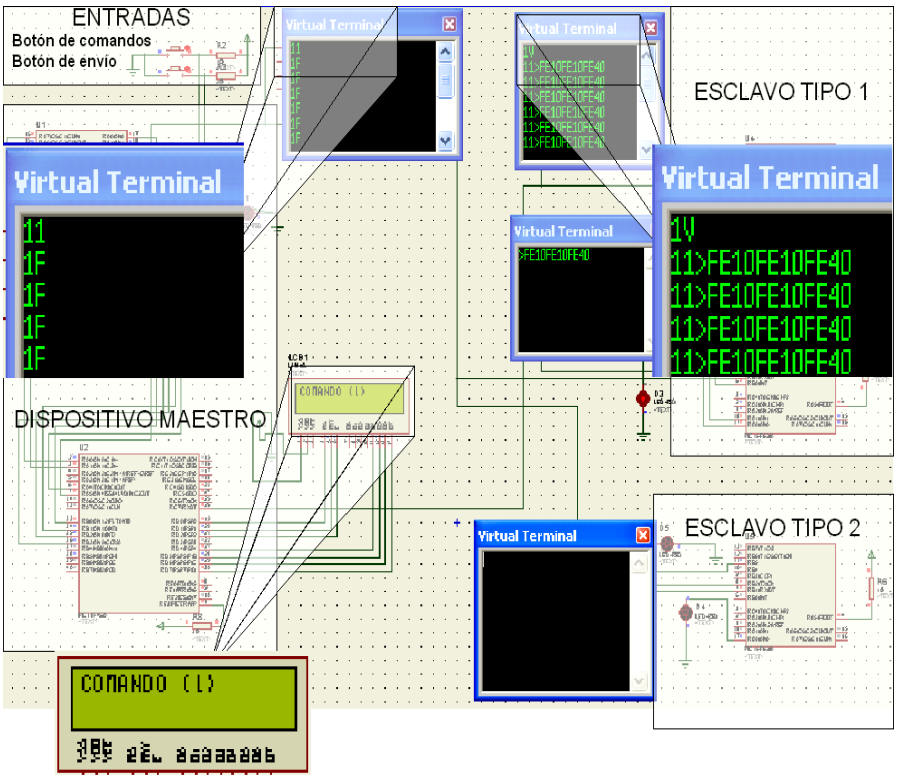


Figura 63: Prueba 2, error en bloque de información enviado por el esclavo tipo 1 al maestro

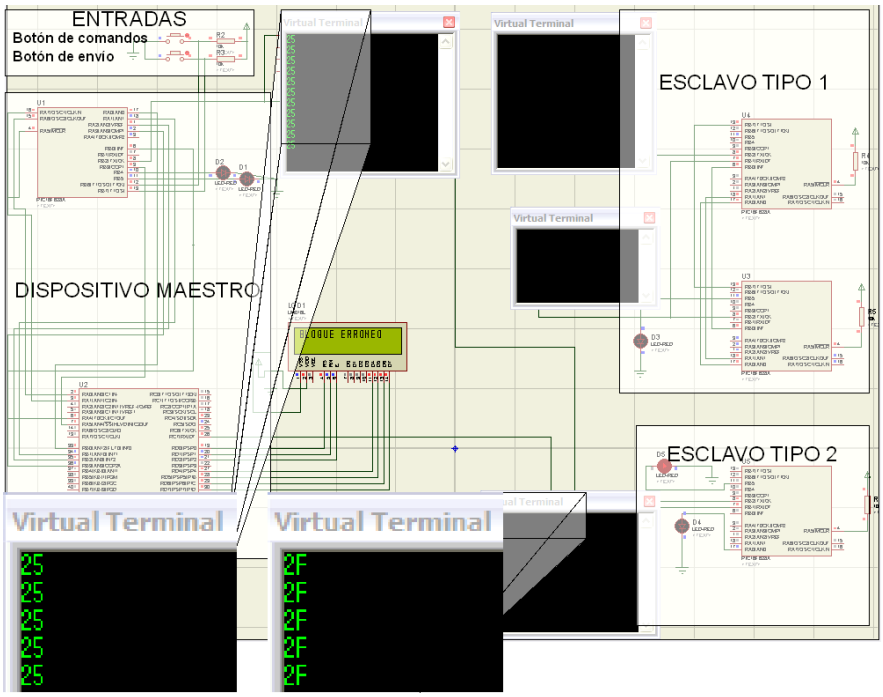


Figura 64: Prueba 2, error en el bloque de comandos enviado al esclavo tipo 2

5. PRUEBAS Y RESULTADOS

Prueba 3

En estas simulaciones (figura 65 y 66) se comprueba el buen funcionamiento del control del *time out*, ya que al desconectar tanto al esclavo tipo 1 como al esclavo tipo 2, se mostró en el display, después de transcurrir los 4 segundos programados, el mensaje "ERROR DISPOSITIVO". Una vez que termina de mostrarse el mensaje, la comunicación se aborta.

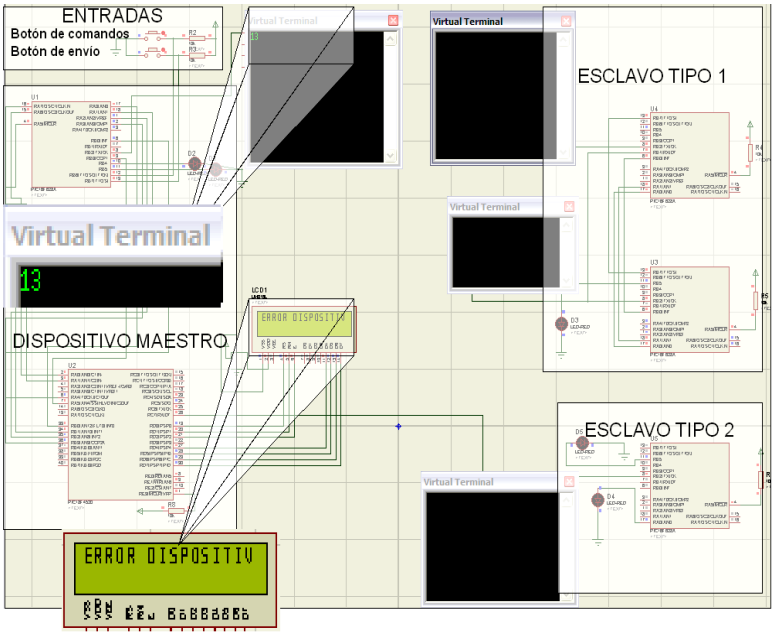


Figura 65: Prueba 3, funcionamiento del time out con el esclavo tipo 1

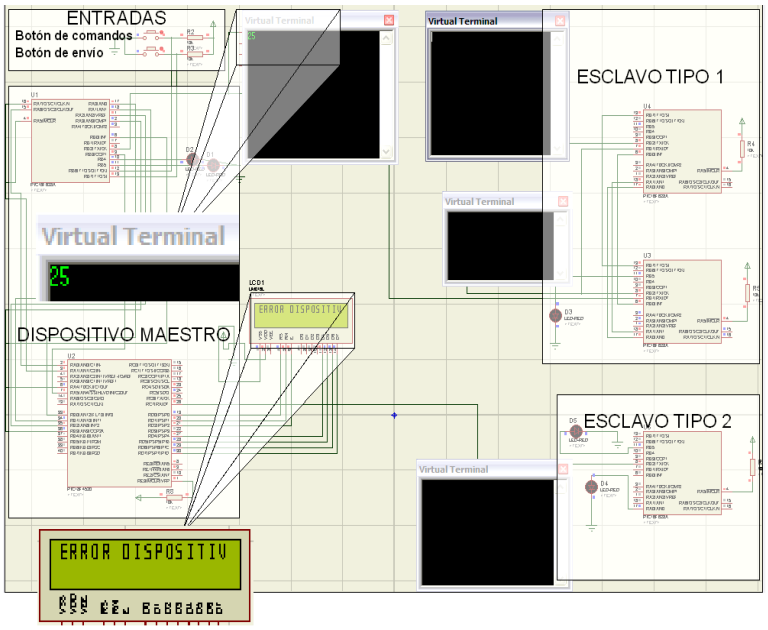


Figura 66: Prueba 3, funcionamiento del time out con el esclavo tipo 2

5. PRUEBAS Y RESULTADOS

IMPLEMENTACIÓN

Una vez realizadas las simulaciones se procedió a implementar y probar el circuito en *protoboard,* para comprobar los resultados de simulación. Posteriormente se hicieron los PCB´s (*Printed Circuit Boards*) del dispositivo maestro y del esclavo tipo 1, para realizar las últimas pruebas del sistema que son las que se presentan aquí.

En las figuras 67 y 68 se muestran al dispositivo maestro y al dispositivo esclavo tipo 1 respectivamente. En las figuras también son señalados cada uno de los elementos que los conforman.

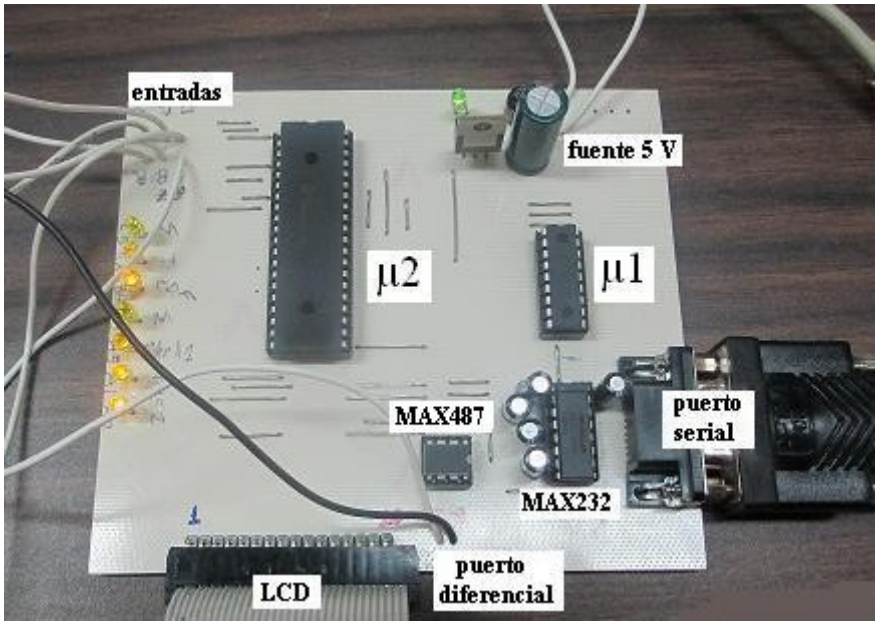


Figura 67: Dispositivo maestro

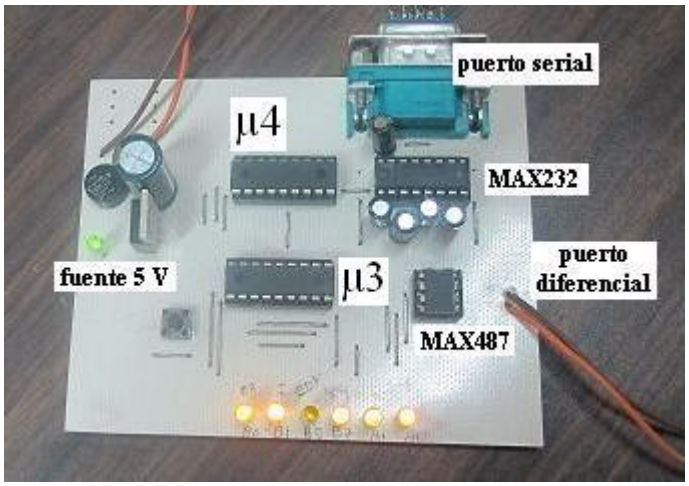


Figura 68: Dispositivo esclavo tipo 1

5. PRUEBAS Y RESULTADOS

En la figura 69 se muestra al medidor de vacío (XGS-600 VARIAN) que se conecta al dispositivo esclavo tipo 1 y que mantiene una comunicación serial RS-232 con el esclavo. Este instrumento responde a una lista de comandos que le son enviados en forma serial (cadenas de bytes en formato ASCII) por el esclavo 1. Sus respuestas también son entregadas en forma serial (cadenas de bytes en formato ASCII).

Figura 69: Medidor de vacío que se conecta al esclavo tipo 1

En la figura 70 se muestra la puesta en marcha del dispositivo maestro y el medidor de vacío conectado al esclavo tipo 1. Para simular la distancia entre el maestro y el esclavo se utilizó un cable dúplex con una longitud de 30 metros para comprobar que la transmisión y recepción de datos a esta distancia se realiza de manera correcta. Se realizaron dos bloques de pruebas una utilizando el display LCD y otra utilizando la PC.

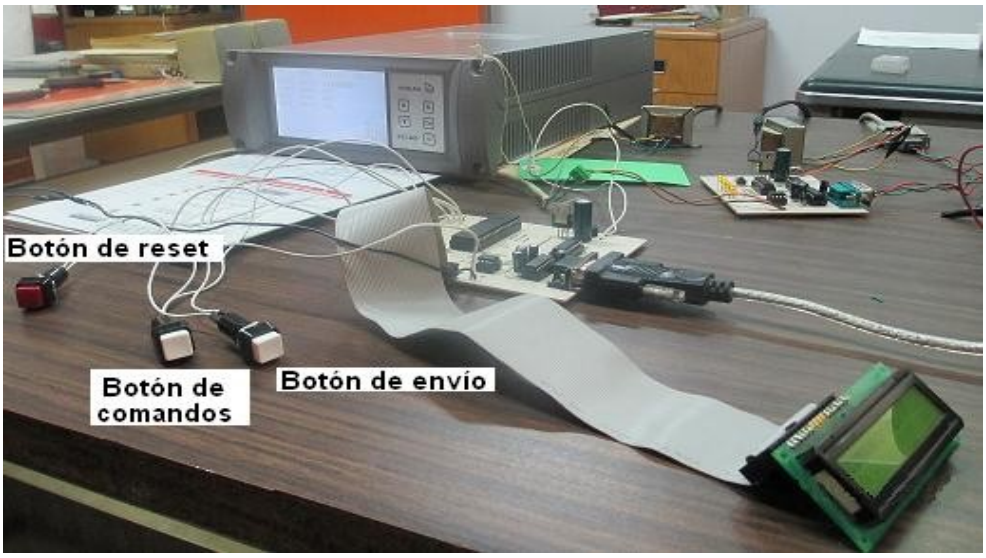


Figura 70: Medidor adjunto al dispositivo esclavo

Nuevamente para probar la comunicación con el medidor de vacío se utilizaron los tres comandos que se indican en su manual de operación y que fueron mostrados en la tabla 2.

Una vez conectados el dispositivo maestro con el esclavo tipo 1 y a éste conectado el medidor de vacío, se procedió a seleccionar el comando que se le enviaría al esclavo mediante el botón de comandos en el dispositivo maestro. Seleccionado el comando mediante el botón de envío, éste se envió dentro de un bloque de comandos al esclavo. Al no haberse presentado errores en la comunicación el dispositivo maestro recibió del esclavo la información que este último le solicitó al medidor de vacío. Las respuestas a los comandos 1, 2 y 3 son presentadas en las figuras, 71, 72 y 73 respectivamente.

Para comprobar que siempre se obtuviera esta respuesta se mantuvo presionado el botón de envío durante 3 horas para cada uno de los tres comandos, obteniendo siempre el mismo resultado.

Es importante señalar que siempre que se ponga en marcha el sistema de comunicación, tanto el dispositivo maestro como el esclavo o esclavos que estén conectados al sistema deben de reiniciarse para asegurarse de que una interrupción en el esclavo o en el maestro no bloquee el sistema, lo mismo debe hacerse con todos los dispositivos del sistema cuando se requiera conectar un esclavo más mientras el sistema se encuentre en marcha. Una vez realizado esto, el sistema operará sin presentar bloqueos.

La razón por la que se bloquea el sistema es porque el dispositivo maestro o los esclavos reciben información que no tiene la estructura de un bloque de comandos, un bloque de error o un bloque de información ya que estos bloques contienen en su último byte el ASCII retorno de carro que permite que una interrupción termine.

Las pruebas con la aplicación de LabVIEW se realizaron de igual forma con el esclavo tipo 1 y se utilizaron los mismos tres comandos de prueba del medidor de vacío (tabla 2) para comparar los resultados. La respuesta a cada uno de los comandos se visualiza en el “proceso 2” de la figura 74. Con esto se comprobó que la aplicación responde de manera correcta, además de poder demostrar una de las ventajas que tiene utilizar la aplicación en comparación con el display del sistema, ya que aquí no se tuvieron que desplazar los datos que se le solicitaron al medidor de vacío para visualizarlos.

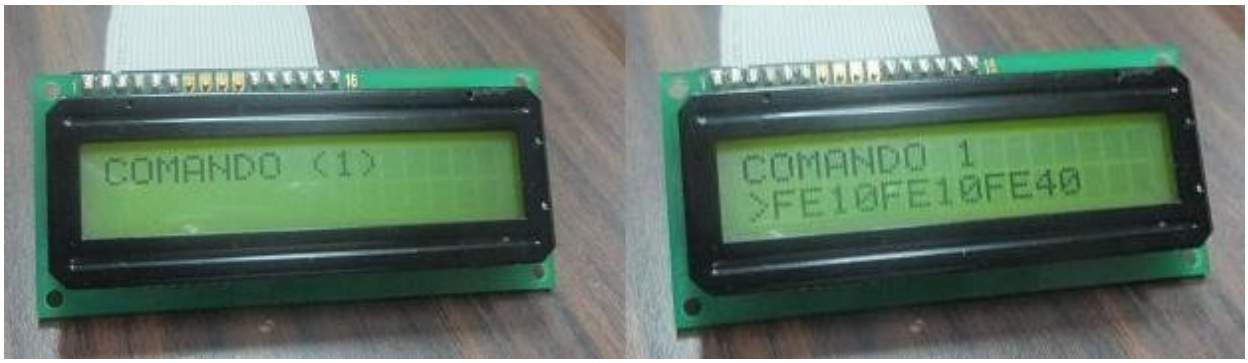


Figura 71: Respuesta al comando 1 enviado al medidor de vacío

5. PRUEBAS Y RESULTADOS

Figura 72: Respuesta al comando 2 enviado al medidor de vacío

Figura 73: Respuesta al comando 3 enviado al medidor de vacío

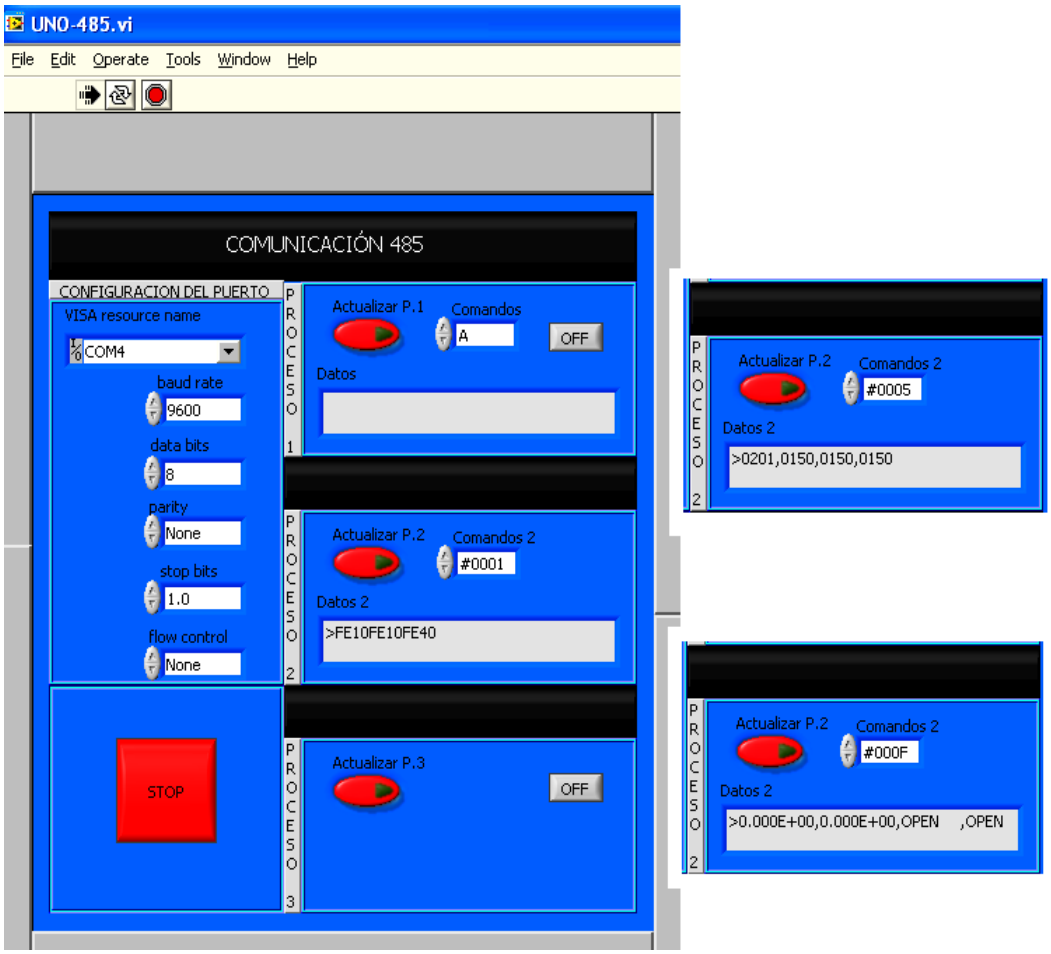


Figura 74: Prueba con la interfaz gráfica LabVIEW

CONCLUSIONES

Este trabajo cumplió su objetivo al diseñar e implementar un sistema de comunicación que permite el enlace entre un dispositivo maestro y varios dispositivos esclavos, en donde el elemento que controla la comunicación es el dispositivo maestro y la solicitud de comunicación puede ser realizada a través de los dos entradas (botones) que posee el dispositivo maestro o a través de la PC mediante una aplicación elaborada en LabVIEW.

Mediante el protocolo desarrollado y los ejemplos de las arquitecturas de los esclavos tipo 1 y tipo 2 se pueden programar y realizar físicamente más esclavos que tengan conectados, equipos de medición, sensores y actuadores. El número máximo de esclavos que se pueden conectar a la red es de 32.

A pesar de que este sistema se elaboró con microcontroladores PIC de Microchip, no se limita a éstos, ya que existen diversos fabricantes que también tienen microcontroladores con el módulo USART, que es el elemento principal que lleva a cabo la recepción y la transmisión de los bloques de datos.

Este sistema de comunicación puede ser utilizado en entornos en donde se requiera una comunicación multipunto y en donde exista una distancia de hasta 1200 metros entre el panel de control o monitoreo y los esclavos.

De las pruebas realizadas con el maestro y el esclavo tipo 1 al que se conectó un medidor de vacío, se comprobó el buen funcionamiento del sistema al obtener las mismas respuestas a los comandos que señala el manual de operación del medidor de vacío.

También en las pruebas al desconectar el esclavo tipo 1 del dispositivo maestro se pudo comprobar el buen funcionamiento de la línea t*ime out,* ya que en el display apareció el mensaje "ERROR DISPOSITIVO" después de transcurrir 4 segundos que es el tiempo que se programó.

Entre las mejoras que se pueden realizar a este trabajo se tienen:

Este sistema podría ser ampliado para tener la capacidad de manejar hasta 256 esclavos utilizando transmisores-receptores de alta impedancia o repetidores de señal en la interfaz RS-485.

En el caso del dispositivo maestro, se puede agregar un teclado matricial en lugar de utilizar botones y se puede agregar un display con mayor número de líneas, o una pantalla gráfica.

En el caso de la aplicación de LabVIEW, mediante los VI´s realizados se pueden extender el código para manejar un mayor número de esclavos. También se pueden crear otros VI´s para mostrar gráficas, almacenar y procesar la información que provenga de los esclavos.

REFERENCIAS

[1] R. Deon y W. Edwin, *Practical TCP/IP and Ethernet Networking.* Ed. Newnes, 2003, pp. 1-14.

[2] A.M. Muhammad, D. M. Rolin y C. Danny, *PIC microcontroller and embedded systems,* Ed. Pearson Education, 2008, pp. 7-8, 14-17, 26-27, 84-87, 387-394, 424.

[3] J. T. Ronald, *Sistemas Digitales Principios y Aplicaciones*, Ed. Pearson Educación, 1996, pp. 36-38.

[4] V. P. Fernando y P. A. Ramón, *Microcontroladores: fundamentos y aplicaciones con PIC*, Ed. Alfaomega, 2007, pp 247-249.

[5] Dallas Semiconductor, *Fundamentals of RS-232 Serial Communications,* Appl. Note 83.

[6] A. Jan, *Serial Port Complete,* 2ed*,* Ed. Lakeview Research, 2007, pp. 79-81, 87.

[7] B&Belectronics, *RS-422 and RS-485* Appl. Note, 1992.

[8] H. Han-Way, *PIC Microcontroller: An Introduction to Software and Hardware Interfacing*, Ed. Thomson, 2005, pp 6-7.

BIBLIOGRAFÍA

B. Rick, T. M. Taqui y N. Matt, *LabVIEW Advanced Programming Techniques*, Ed. CRC Press, 2007.

G. B. Eduardo, *Compilador C CCS y simulador Proteus para microcontroladores PIC,* Ed. Alfaomega, 2008.

Microchip Technology Inc, *Asynchronous communications with the PICmicro USART,* G. Mike, Appl. Note AN647, 2003

Microchip Technology Inc, *Ethernet Theory of Operation*, M. Simmons, Appl. Note AN1120, 2008.

Microchip Technology Inc, *PICmicro Mid-Range MCU Family Reference Manual,* 1997.

Microchip Technology Inc, *SPI Overview and Use of the PICmicro Serial Peripheral Interface*, Appl. Note AN647.

National Instruments, *BridgeVIEW and LabVIEW G Programming Reference Manual*, 1998.

Texas Instruments, *Comparing Bus Solutions*, Application Report, 2000.

Texas Instruments, *Introduction to the Controller Area Network (CAN)*, C. Steve, Application Report, 2002.

APÉNDICES

Compilador CCS

El compilador C de CCS es un compilador desarrollado para todas las familias de microcontroladores PIC (10F, 12F, 16F, 18F y 24F). Además de contar con una gama amplia de librerías de funciones predefinidas.

Un programa está compuesto de los siguientes cuatro elementos:

Directivas de preprocesador.- Se encargan de controlar la conversión del programa a código máquina por parte del compilador. Ejemplos: #include , #fuses, #use, etc.

Definición de datos.- Son los datos constantes o variables con los que se trabaja a lo largo del programa.

Definición de funciones.- Permiten agrupar conjuntos instrucciones con fines o tareas comunes.

Comentarios.- Permiten describir la función que tienen cada una de las líneas del código. Se pueden insertar en cualquier parte del programa iniciando con dos diagonales invertidas "//" para cada línea, o dentro de los signos "/*" y "*/", cuando un comentario abarque más de una línea.

Cada programa en C debe de contener una función principal, la cual indica el comienzo de ejecución de un programa. Un programa puede ser dividido en múltiples funciones, para elaborar un programa más modular; estas funciones deben de ser llamadas desde la función principal u otras sub-funciones. En proyectos extensos las funciones pueden ser colocadas en diferentes archivos C o en archivos encabezados que pueden ser agrupados junto con el programa principal.

Operadores

Operadores aritméticos

Operador	Descripción
+	suma
-	resta
*	multiplicación
/	división
%	resto de una división entera (módulo)
++	incremento
--	decremento

APÉNDICES

Operadores de comparación

Operador	Descripción
==	igual
!=	distinto
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que

Operadores lógicos

Operador	Descripción
&&	AND
||	OR
!	NOT

Operadores bitwise

Operador	Descripción
&	And
|	Or
^	Xor
~	complemento
<<	rotar izquierda
>>	rotar derecha

Tipos de datos

Tipo	Tamaño	Mínimo	Máximo
int1	1 bit	0	1
int8	8 bits	0	255
char	8 bits	0	255

APÉNDICES

signed int8	8 bits	-128	127
int16	16 bits	0	65536
signed int16	16 bits	-32768	32767
int32	32 bits	0	4 294 967295
signed int32	32 bits	-2 147 483 648	2 147 483 647
float	32 bits	±1.175*10^-38	±3.102*10^38

Constantes

123	decimal
123	octal
0x123	hexadecimal
0b01011010	binario
'x'	carácter
'\010'	carácter octal
'\0xA5'	carácter hexadecimal
“abcdef”	cadena
'\n'	fin de línea (ASCII 10)
'\r'	retorno de carro (ASCII 13)
'\b'	backspace (ASCII 8)

Estructuras de control

Sentencia	Ejemplo

if (expresión){
sentencias;
} else{
sentencia;
}
	if(y==10){
	x++;
	} else{
	x--;
	}

while(expresión){
sentencias;
}
	i=0;
	c[i]=getchar();
	while(c[i] != 13){
	i++;
	c [i]= getchar();
	}

do {
sentencias;
}while (expresión);
	do{
	i++;
	}while(i<10);

for (inicialización;condición de finalización;
incremento){
sentencias;
}
	for(i=0;i<=10;i++){
	lcd_gotoxy(1,1);
	printf(LCD_PUTC,"%d",i);
	}

switch(expresión){
case constante 1: sentencias;
break;
case constante N: sentencias;
break;
default: sentencias;
break;
}
	switch(cmd){
	case 0: putchar(1);
	break;
	case 1: putchar(2);
	break;
	default:putchar(3);
	break;
	}

return(expresión);	return (5); //Devuelve datos en las funciones
break;	break; /*Permite salir de un bucle(while, for, do, switch)*/

Definición de variables

Tipo_de_dato Nombre = Valor_inicial

Ejemplo:

int8 temperatura = 0;

Definición de funciones

Tipo_de_dato Nombre (Tipo_dato Parámetro1, Tipo_dato Parámetro2,...){

Sentencias;

}

Ejemplo:

```
int8 suma (int8 a, int8 b){
int8 c;
c=a+b;
return c;
}
```

APÉNDICES

Funciones predefinidas

RS-232	getchar(); Lee un byte de el pin serial de la USART.
	gets(); Lee una cadena de bytes del pin serial de la USART.
	putchar(); Envía un byte por el pin serial de la USART.
	puts(); Envía una cadena de bytes por el pin serial de la USART.
	kbhit(); Retorna un uno lógico cuando un byte se ha recibido por la USART.
Retardos	delay_us(time); time es una constante entera de 16 bits que indica un tiempo de retardo en microsegundos.
	delay_ms(time); time es una constante entera de 16 bits que indica un tiempo de retardo en milisegundos.

Manejo de puertos

Configuración de los puertos como entradas y salidas.

set_tris_X(valor); //Configura los pines del puerto X (donde X puede ser A,B,C,D,E,F)como entradas y salidas mediante un número (valor) de 0 a 255.

Ejemplo:

set_tris_b(0b11110000); // Los pines b0,...,b3 están configurados como salidas y los pines // b4,…, b7 como entradas.

output_X(valor); //Establece un valor desde 0 hasta 255 por el puerto correspondiente X.

Ejemplo:

output_X(0b00001111); // Los pines b0,...,b3 son puestos a 1 lógico mientras que los pines // b4,...,b7 son puestos a 0 lógico.

output_low(pin_B4); //Establece el valor de 0 en el pin b4 del puerto B.

output_high(pin_A0); // Establece el valor de 1 en el pin a0 del puerto A.

indice = input_X(); //Asigna a la variable indice el estado del puerto X.

clv = input(pin_A6); //Asigna el estado del pin A6 del puerto A a la variable clv.

APÉNDICES

Código fuente

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Programa del microcontrolador 1

#include <16f628A.h>
#FUSES INTRC_IO, NOWDT, MCLR
#use delay (clock =4000000)
#use rs232(baud=9600, xmit=pin_b2, rcv=pin_b1, bits=8, parity=N)

char key;//clave para recibir datos
unsigned int i;
char buffer[10];//receptor de datos
char key2;
unsigned int indice;
char cte;
char clv;
char bus2_C;
char bus2_I;
unsigned int m;
unsigned int n;
void enviar_datos(void);
unsigned long time_out;
char tmt;

//FUNCIONES
unsigned int paridad(unsigned int cte);
void buffer_datos(void);
void send_dir(void);
void send_cmd(void);
//void bus2FV_t(void);
void time_out_b0_a4(void);
void ctrl2_a5_b3(void);
void send_blockF(void);
void send_blockV(void);
void send_dat_PC(void);
void send_bus1(void);
void CFD_b4_send(void);
void CFD_b4_rece(void);

// Rutina de Interrupción
#int_rda
void serial_isr(){
            key=getchar();
             buffer_datos(); // Se reciben bloques de comandos provenientes de la PC
                        if(key=='W' || key== 'X'){
                                    if(key2=='A'){ //Si está ocupado
                                                clv='E'; //En espera para enviar //estado de espera
                                             }else if(key2=='B'){ //si esta desocupado
                              enviar_datos(); //funcion que envia los datos recibidos de la PC a los dispositivos   esclavos por la USART
                                             } //Fin de if key2 == B
                                  } //Fin de if key == X || W
                      } //Fin de interrupcion
//Principal
void main(){
                       //ENTRADA = 1  SALIDAS = 0
   set_tris_a(0b11100000);//Se define al pin A4 como entrada señal para comprobar que se realizado un envio a la PC
   set_tris_b(0b11001110); //Se defineN a los pines B6, B7, como entradas pines para los controles pulsadores
            key2='B';
```

```
enable_interrupts(global);
enable_interrupts(int_rda);
        indice=0; //se inicializa el contador de comandos
        output_A(0);
        bus2_C='V';
        bus2_I='V';
CFD_b4_send(); //CONTROL de flujo de datos
        output_high(pin_B0); //LINEA TIMEOUT
        m=input(pin_A6); //BUS 2 A6, A7
        n=input(pin_A7);

        while(TRUE){
                if(input(pin_B6)==0){  //B6 CONTADOR DE COMANDOS
                        delay_ms(50);
                        key2='A'; //estado ocupado
                        if(indice==10){
                                indice=0;
                                }
                        indice++;
                        m=input(pin_A6);
                        n=input(pin_A7);
                                        send_bus1();
                        m=input(pin_A6);
                        n=input(pin_A7);
                while(input(pin_B6)==0);
                        delay_ms(50);
                        key2='B'; //estado desocupado
                }else if(input(pin_B7)==0){ // B7 ENTRADA QU ENVIA LOS DATOS POR LA USART
                        delay_ms(50);
                        key2='A'; //estado ocupado
                        do{
                CFD_b4_send(); //CONTROL de flujo de datos //Es el pin que controla al max487 ctrl==1 para enviar
                                  send_dir(); //Se envia una direccion dependiendo dek valor del indice
                                  send_cmd(); //Se envian los comandos
                        CFD_b4_rece();//ctrl==0 para poder recibir
                                bus2_C='V';//valor por default
                                time_out=0;
                                tmt='V';
        while(input(pin_A6)==1 && input(pin_A7)==1 && (++time_out<1000000));  //BUS2 = 3 y timeout
                                if (input(pin_A6)==0 && input(pin_A7)==1){  //     BUS2 = 2
                                          while(input(pin_A6)==0);
                                                bus2_C='F'; //Se presentaron errores en el bloque de comandos enviado
                                    }else if (input(pin_A6)==1 && input(pin_A7)==0){//   BUS2 = 1
                                            while(input(pin_A6)==1 && input(pin_A7)==0);
                                              bus2_C='V'; //No se presentaron errores en el bloque de comandos enviado
                                                                break;
                                   }else if(time_out>64000){ // El contador timeout ha sobrepasado el tiempo de espera
                                          time_out_b0_a4();
                                        tmt='F';  //Hace finalizar la comunicación debido a que no hay esclavo conectado
                                          break;
                                                }
                                }while(bus2_C=='F');
                                                  CFD_b4_rece(); //ctrl==0 para poder recibir
                                if(tmt=='V'){
                                    if(key=='W'){ //Selecciona entre una comunicacion simple y una compuesta
                                                do{
                                                        while(input(pin_A6)==1 && input(pin_A7)==1); // BUS2 = 3
                                                          if (input(pin_A6)==0 && input(pin_A7)==1){// BUS2 = 2
                                                                while(input(pin_A6)==0);
                                                        output_high(pin_B4); //Es el pin que controla al max487 ctrl==1 para enviar
```

```
                                                send_blockF();
                                        bus2_I='F'; //Se presentaron errores en el bloque de informacion recibido por m2
                                                }else if (input(pin_A6)==1 && input(pin_A7)==0){//BUS2 = 1
                                                        while(input(pin_A6)==1 && input(pin_A7)==0);
                                                output_high(pin_B4); //Es el pin que controla al max487 ctrl==1 para enviar
                                                        send_blockV();
                                                bus2_I='V';//No se presentaron errores en el bloque de informacion recibido por m2
                                                                }
                                                        }while(bus2_I=='F');
                                        ctrl2_a5_b3(); //Fin de la comunicación a través de la línea ctrl2
                                           }
                                        }
                                        CFD_b4_send(); //Control para poder enviar
                                        delay_ms(50);
                                        key2='B'; //estado desocupado
                                        }else if(clv=='E'){ //si hay algo en espera
                                        enviar_datos();
                                        clv='D'; //buffer desocupado
                                        }else if(input(pin_B6)==1 && input(pin_B7)==1){
                                        key2='B'; //estado desocupado
                output_high(pin_B5);  //LED
                delay_ms(100);
                output_low(pin_B5);
                delay_ms(100);
                                        key2='B'; //estado desocupado
                                        }
                }
}
//  Rutina de Paridad
unsigned int paridad (unsigned int cte){
        char rot1;
        char var2;
        char rot2;
        char var3;
        char rot3;
        char var4;
        char var5;
        char p;
        p=0x01;
        rot1=(cte>>1);
        var2=cte^rot1;
        rot2=(var2>>2);
        var3=var2^rot2;
        rot3=(var3>>4);
        var4=(var3^rot3);
        var5=var4&p;
        return var5;
        }
//  Rutina Enviar Datos a la PC
void enviar_datos(void){
        do{
                CFD_b4_send(); //Es el pin que controla al max487 ctrl==1 para enviar
                  send_dat_PC();
                CFD_b4_rece();  //ctrl==0 para poder recibir

                        bus2_C='V';//valor por default
                        time_out=0;
                        tmt='V';

        while(input(pin_A6)==1 && input(pin_A7)==1 && (++time_out<1000000));//BUS2 = 3 y timeout
```

```
            if (input(pin_A6)==0 && input(pin_A7)==1){//BUS2 = 2
                  while(input(pin_A6)==0);
                        bus2_C='F';
              }else if (input(pin_A6)==1 && input(pin_A7)==0){//BUS2 = 1
                    while(input(pin_A6)==1 && input(pin_A7)==0);
                          bus2_C='V';
                          break;
                  }
        else if(time_out>64000){
                    time_out_b0_a4();
                    tmt='F';
                    break;
                  }
        }while(bus2_C=='F');
                          CFD_b4_rece();  //ctrl==0 para poder recibir
                  if(tmt=='V'){
                          if(key=='W'){ //selecciona entre una comunicacion simple y una compuesta
                          do{
                                  while(input(pin_A6)==1 && input(pin_A7)==1); //BUS2 = 3
                                    if (input(pin_A6)==0 && input(pin_A7)==1){ //BUS2 = 2
                                          while(input(pin_A6)==0);
                                             output_high(pin_B4); //Es el pin que controla al max487 ctrl==1 para enviar
                                                        send_blockF();
                                                        bus2_I='F';
                                       }else if (input(pin_A6)==1 && input(pin_A7)==0){//BUS2 = 1
                                          while(input(pin_A6)==1 && input(pin_A7)==0);
                                             output_high(pin_B4); //Es el pin que controla al max487 ctrl==1 para enviar
                                                        send_blockV();
                                                        bus2_I='V';
                                                        }
                          }while(bus2_I=='F');
                   ctrl2_a5_b3(); //Fin de la comunicación a través de la línea ctrl2
                          }
                  }
                 CFD_b4_send(); //Control para poder enviar
}
void buffer_datos(void){

        i=0;
                  buffer[i]=getchar();
                  while(buffer[i] != 13){
                  i++;
                  buffer[i]=getchar();
                  }
}
void send_dir(void){
if(indice< 5){
        key='W';
        putchar('1'); // Se envia la direccion
        }else if(indice>= 5 && indice< 7){
        key='X';
        putchar('2');
         }else if(indice>= 7 && indice< 9){
        key='X';
        putchar('3');
         }else if(indice>= 9 && indice< 10){
        key='X';
        putchar('4');
}
}
```

```
void send_cmd(void){
        cte=indice+48;
        putchar(cte); //Se envian los comandos
        putchar(paridad(cte));
        //putchar(cte+1);
        //putchar(paridad(cte+1));
        //putchar(cte+2);
        //putchar(paridad(cte+2));
        //putchar(cte+3);
        //putchar(paridad(cte+3));
        putchar(13);
        delay_ms(250);
}
void time_out_b0_a4(void){
                                output_high(pin_B0);  //TIMEOUT 1 ERROR DISPOSITIVO
                                output_low(pin_B0);   //TIMEOUT 0
                                delay_ms(2000);
                                output_high(pin_B0);  //TIMEOUT 1
}
void ctrl2_a5_b3(void){
                                 while(input(pin_B3)==1); //CONTROL 2 =1
                                 while(input(pin_B3)==0); //CONTROL 2 =0
}
void send_blockF(void){
                                 delay_ms(100);
                                 putchar('1');
                                putchar('F');
                                 putchar(13);
}
void send_blockV(void){
                                 delay_ms(100);
                                 putchar('1');
                                 putchar('V');
                                 putchar(13);
}
void send_dat_PC(void){
                        i=0;
                        putchar(buffer[i]);
                        i++;
                        while(buffer[i] != 13){
                        putchar(buffer[i]);
                        putchar(paridad(buffer[i]));
                        i++;
                        clv='D'; //bufer desocupado
                        }
                        putchar(buffer[i]);
                        delay_ms(10);
}
void send_bus1(void){
                         output_A(indice);
}
void CFD_b4_send(void){
                       output_high(pin_B4); //CONTROL DE DATOS
}
void CFD_b4_rece(void){
                       output_low(pin_B4); //CONTROL DE DATOS
}
```

APÉNDICES

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Programa del microcontrolador 2

#include <18f4520.h>
#FUSES INTRC_IO, NOWDT, MCLR
#use delay (internal = 4Mhz)
#include "flex_lcd.c"
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7, bits=8, parity=N)

//Variables
char buffer[160];//receptor de datos
char buffer3[80];
char buffer2;
char dir;
char par;
unsigned int i;
unsigned int x;
//Funciones
unsigned int paridad(unsigned int cte);
void imprimir_datos(void);
void buffer_datos(void);
void blo_error_f(void);
void blo_error_v(void);
void rev_paridad(void);
void ext_datos(void);
void end_ctrl2(void);
void time_out(void);

// Servicio de Interrupción
#int_rda
            void serial_isr(){
            dir=getchar(); //Se recibe una dirección
             buffer_datos(); // Se reciben bloques de error o de información provenientes del esclavo 1
             if(dir=='1'){
               if (buffer[0]=='F'){
                            blo_error_f(); //Se informa al pic 1 que el bloque de comandos presento errores
                }else if(buffer[0]=='V'){
                             blo_error_v();//Se informa al pic 1 que el bloque de comandos no presento errores
                }else{
                             rev_paridad(); //Se revisa que no haya errores en la información que ha enviado el esclavo 1
                        if(par=='F'){
                             blo_error_f(); //Se informa al pic 1 que el bloque de información presento errores

                          }else{
                                blo_error_v();//Se informa al pic 1 que el bloque de información no presento errores
                                ext_datos(); //Se eliminan los bytes de paridad del bloque de información
              printf(LCD_PUTC, "\f");
                                 if(buffer2 == 1){
                                     lcd_gotoxy(1,1);
                                     printf(LCD_PUTC, "comando %d", buffer2);
                                 }else if(buffer2 == 2){
                                     lcd_gotoxy(1,1);
                                     printf(LCD_PUTC, "cmd %d", buffer2);
                                 }else if(buffer2 == 3){
                                     lcd_gotoxy(1,1);
                                     printf(LCD_PUTC, "cty %d", buffer2);
                                 }else if(buffer == 4){
                                     lcd_gotoxy(1,1);
                                     printf(LCD_PUTC, "hfig %d", buffer2);
```

```
                                }else{
                                    lcd_gotoxy(1,1);
                                    printf(LCD_PUTC, "hfig %d", buffer2);
                                }
                                    imprimir_datos();
                                    end_ctrl2(); //Libera el bus
                                                }
                            }
                           }
   else if(dir=='2'){
                    if (buffer[0]=='F'){
                                blo_error_f(); //Se informa al pic 1 que el bloque de comandos presento errores
                                            printf(LCD_PUTC, "\f");
                                            lcd_gotoxy(1,1);
                                            printf(LCD_PUTC, "BLOQUE ERRONEO");
                    }else if(buffer[0]=='V'){
                                blo_error_v();//Se informa al pic 1 que el bloque de comandos no presento errores
                                            printf(LCD_PUTC, "\f");
                                            lcd_gotoxy(1,1);
                                            printf(LCD_PUTC, "BLOQUE EJECUTADO");
                        }
            }
}

void main(){

             char indice;
             set_tris_b(0x0F); // Se defineN a los pines B0, B1, B2, B3 como entradas
             set_tris_e(0b11111100);

            lcd_init();
            delay_ms(100);
            enable_interrupts(global);
            enable_interrupts(int_rda);

            output_high(pin_E0);
            output_high(pin_E1);
            output_high(pin_A5);
            output_high(pin_A0);
            output_high(pin_A1);
            output_low(pin_C5);
            output_high(pin_C4);
            output_low(pin_B4);
            output_low(pin_B5);
            output_low(pin_B6);
            output_low(pin_B7);

            indice=0; //se inicializa el contador de comandos

while(TRUE){

        indice= input_B();
        buffer2=indice&0x0F;
        printf(LCD_PUTC, "\f");
        lcd_gotoxy(1,1);
        printf(LCD_PUTC, "INSTRUCCION (%d)",buffer2);

        output_high(pin_C5);
        delay_ms(100);
        output_low(pin_C5);
```

```
        delay_ms(100);
                                time_out();
             }
}
//Función Imprimir Datos
void imprimir_datos(void){
            char c;
            char k;
            signed int j;

            i=0;
            while(buffer3[i] != 13){
                        i++;
                        }
            c=i;
            if(c>16){
                        for(i=0;i<=15;i++){
                                    buffer3[i+c]=32;
                                    }
                        //printf(LCD_PUTC, "\f");

            for(k=1;k<=3;k++){
                        for(j=-1;j<=c-1;j++){
                                     for(i=1;i<=16;i++){
                        lcd_gotoxy(i,2);
                        printf(LCD_PUTC, "%c", buffer3[i+j]);
                                                }
                                                delay_ms(250);
                                    }
                        }
            for(i=1;i<=16;i++){
       lcd_gotoxy(i,2);
       printf(LCD_PUTC, "%c", buffer3[i+j]);
   }
            }else{

            i=1;
            while(buffer3[i-1] != 13){
            lcd_gotoxy(i,2);
            printf(LCD_PUTC, "%c",buffer3[i-1]);
            //delay_ms(100);
            i++;
            }
            delay_ms(1000);
            delay_ms(1000);
            delay_ms(1000);
            }
}
void buffer_datos(void){
                        i=0;
                        buffer[i]=getchar();
                        while(buffer[i] != 13){
                        i++;
                        buffer[i]=getchar();
                        }
}
void blo_error_f(void){

                         output_high(pin_A0);
                        output_high(pin_A1);
```

```
                        delay_ms(100);
                        output_low(pin_A0);
                        output_high(pin_A1);
                        delay_ms(200);
                        output_high(pin_A0);
                        output_high(pin_A1);
}
void blo_error_v(void){
                       output_high(pin_A0);
                        output_high(pin_A1);
                        delay_ms(100);
                        output_high(pin_A0);
                        output_low(pin_A1);
                        delay_ms(200);
                        output_high(pin_A0);
                        output_high(pin_A1);
}
void rev_paridad(void){
                        i=0;
                        while(buffer[i]!=13){
                                if(buffer[i+1]==paridad(buffer[i])){
                                i++;
                                i++;
                                par='V';
                        }else{
                        par='F';
                        output_low(pin_B5);//Volver a recibir los comandos otra vez porque se presento un error.

                        break;
                        }
            }
}
void ext_datos(void){
                                 i=0;
                                x=0;
                                while(buffer[i]!=13){
                                        buffer3[x]=buffer[i];
                                        i++;
                                        i++;
                                        x++;
                                        }
                                buffer3[x]=buffer[i];

}
void end_ctrl2(void){
                                output_high(pin_A5);
                                output_low(pin_A5);
                                delay_ms(200);
                                output_high(pin_A5);
}
void time_out(void){

      while(input(pin_A4)==0){
      printf(LCD_PUTC, "\f");
      lcd_gotoxy(1,1);
      printf(LCD_PUTC, "ERROR DISPOSITIVO");
     }
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Programa del microcontrolador 3

#include <16f628A.h>
#FUSES INTRC_IO, NOWDT
#use delay (clock =4000000)
#use rs232(baud=9600, xmit=pin_b2, rcv=pin_b1, bits=8, parity=N)

//SCH ESCLAVO RS485 IC U2, PIC16F628A, @@@ UTILIZA RB5 @@@
char dir; //Dirección del dispositivo
char par; //Toma el valor de falso o verdadero dependiendo de si el envio de comandos fue de manera correcta..
char i;
char buffer[10];
char cmd1[5]={'#','0','0','0','1'};
char cmd2[5]={'#','0','0','0','5'};
char cmd3[5]={'#','0','0','0','F'};
//_________Comando de 7 bytes_________________
char cmd4[7]={'#','0','0','0','2','I','1'};

//Funciones
char paridad(char);
void buffer_datos(void);
void CFD_b5_send(void);
void CFD_b5_rece(void);
void bus4_F(void);
void bus4_V(void);
void rev_paridad(void);
void ctrl3_b0(void);
void ctrl4_b7(void);

#int_rda
        void serial_isr(){
        dir=getchar();
        buffer_datos(); // Se reciben bloques de error o de comandos provenientes del dispositivo maestro

                        if(dir=='1'){ //Si la direccion corresponde

                         if (buffer[0]=='F'){
                                //comunicacion via interrupcion con pic 4
                                CFD_b5_send();//ctrl 2 a 1 para enviar datos.
                    bus4_F();//Se informa al m4 que el bloque de información enviado por el m4 al maestro presento errores
                                ctrl4_b7(); //.....................
                                CFD_b5_rece();//ctrl 2 a 1 para recibir datos.

                        }else if(buffer[0]=='V'){
                                CFD_b5_rece();//ctrl 2 a 1 para recibir datos.
                bus4_V();//Se informa al m4 que el bloque de información enviado por el m4 al maestro no presento errores
                                break;
                        }else{
                           CFD_b5_send();//ctrl 2 a 1 para enviar datos.
                                delay_ms(250);
                                rev_paridad(); //Se revisa la paridad del bloque de comandos

                     if(par=='F'){
                                CFD_b5_send();
                                  //debe ser sensado por el programa principal del pic 4
                            bus4_F();//Se informa al m4 que el bloque de comandos recibido del maestro presento errores
                              //aqui el pic 4 debe de enviar una F al maestro
            ctrl3_b0();//Se envia un bloque de error F
```

```
                                        //output_low(pin_B5);//modo recepcion de datos
            }else{
                                        CFD_b5_send();//modo enviar
                                      bus4_V();//Se informa al m4 que el bloque de comandos recibido del maestro no presento
errores
                                        ctrl3_b0();//Se envia un bloque de error V
                                     //aqui el pic 4 debe de enviar una V  al maestro
                                        delay_ms(10);

    switch (buffer[0]) {
                                case 0x31: //numero 1 ascii
                                                for(i=0; i<5; i++){
                                                        putchar(cmd1[i]);
                                                        }
                                                        putchar(13);
                                break;
                                case 0x32: //numero 2 ascii
                                                for(i=0; i<5; i++){
                                                        putchar(cmd2[i]);
                                                        }
                                                        putchar(13);
                                break;
                                case 0x33: //numero 3 ascii
                                                for(i=0; i<5; i++){
                                                        putchar(cmd3[i]);
                                                        }
                                                        putchar(13);
                                break;
                                 case 0x34: //numero 4 ascii
                                                for(i=0; i<7; i++){
                                                        putchar(cmd4[i]);
                                                        }
                                                        putchar(13);
                                break;
                                default:
                                CFD_b5_send();
                                break;
                                }

                                ctrl4_b7(); //..... //fin de la interrupcion por comandos
        CFD_b5_rece();//modo recepcion de datos

                                }
                        }
            }
}
void main(){

            enable_interrupts(global);
            enable_interrupts(int_rda);

            CFD_b5_rece();
            output_A(3);
            par='V';

            while(TRUE){
                                CFD_b5_rece(); //ctrl 2 a 0 para poder recibir en cualquier momento
                                }
            } //FIN void main
```

```
void buffer_datos(void){

                i=0;
                        buffer[i]=getchar();
                        while(buffer[i] != 13){
                        i++;
                        buffer[i]=getchar();
                        }

}
void CFD_b5_send(void){
output_high(pin_B5);
}
void CFD_b5_rece(void){
output_low(pin_B5);
}
void bus4_F(void){
                                                output_A(3);
                                                delay_ms(10);
                                                output_A(2);
                                                delay_ms(200);
                                                output_A(3);
}
void bus4_V(void){
                output_A(3);
                                                delay_ms(10);
                                                output_A(1);
                                                delay_ms(200);
                                                output_A(3);
}
void rev_paridad(void){
                i=0;
                while(buffer[i]!=13){ //...
                                                //revisión de la paridad de los datos provenientes del dispositivo maestro
                                                if(buffer[i+1]==paridad(buffer[i])){
                                                        i++;
                                                        i++;
                                                        par='V';
                                                }else{
                                                   par='F';
                                                //Volver a los comandos otra vez porque se presento un error.
                                                   CFD_b5_send();
                                                //putchar('B');
                                                   break;
                                                    }
                                        }//...
}

}
void ctrl3_b0(void){
                  while (input(pin_B0)==1);//Se espera a que se haya enviado una F
                                                while (input(pin_B0)==0);
}
void ctrl4_b7(void){
                  while(input(pin_B7)==1);
                                                        while(input(pin_B7)==0);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Programa del microcontrolador 4

//PIC4_RS485 SLAVE
#include <16f628A.h>
#FUSES INTRC_IO, NOWDT
#use delay (clock =4000000)
#use rs232(baud=9600, xmit=pin_b2, rcv=pin_b1, bits=8, parity=N)

char buffer[60];//BUFFER PARA RECIBIR DATOS DEL MEDIDOR
unsigned int i;
char bus4;

//Funciones
unsigned int paridad(unsigned int cte);
void buffer_datos(void);
void end_ctrl4(void);
void end_ctrl3(void);
void send_dir_inf(void);
void send_blockF(void);
void send_blockV(void);

// Rutina de Interrupción
#int_rda
        void serial_isr(){
                buffer_datos();//Se recibe la información del medidor de vacio
                do{
       send_dir_inf(); //Se envia la direccion y la informacion proveniente del medidor de vacio al dispositivo maestro con los bytes de paridad
        end_ctrl4(); //b7 1, 0, 1

       bus4='V';

          while(input(pin_A0)==1 && input(pin_A1)==1); //BUS4 = 3
                                        if (input(pin_A0)==0 && input(pin_A1)==1){ // BUS4 = 2
                                                while(input(pin_A0)==0);
                                                        bus4='F';
                                        }else if (input(pin_A0)==1 && input(pin_A1)==0){// BUS4 = 1
                                                while(input(pin_A0)==1 && input(pin_A1)==0);
                                                        //bus4='V';
                                                break;
                                                }
                                        //end_ctrl3();
                                }while(bus4=='F');
        }// Fin     void serial_isr


//  Principal
void main(){

        enable_interrupts(global);
        enable_interrupts(int_rda);

        bus4='V';
        output_low(pin_B5);
        output_high(pin_B0);
        output_high(pin_B7);
delay_ms(1000);
        while(TRUE){
                while(input(pin_A0)==1 && input(pin_A1)==1); //BUS4 = 3
```

```
                                if (input(pin_A0)==1 && input(pin_A1)==0){//BUS4 = 1
                                        while(input(pin_A1)==0);
                                        send_blockV(); //Se envia un bloque de error V al maestro
        end_ctrl3();
                                }else if (input(pin_A0)==0 && input(pin_A1)==1){//BUS4 = 2
                                        while(input(pin_A0)==0 && input(pin_A1)==1);
                                        send_blockF(); //Se envia un bloque de error F al maestro
        } // Fin else if

                                } // Fin while(TRUE)
        } // Fin void main
void end_ctrl4(void){
                                output_high(pin_B7);
                                output_low(pin_B7);
                                delay_ms(400);
                                output_high(pin_B7);
}
void end_ctrl3(void){
                        output_high(pin_B0);
                        output_low(pin_B0);
                        delay_ms(400);
                        output_high(pin_B0);
}
void send_dir_inf(void){
                                delay_ms(250); //antes 250
                                putchar('1');
                                delay_ms(100);  //antes 100
                                i=0;
                                while(buffer[i]!=13){
                                        putchar(buffer[i]);
                                        putchar(paridad(buffer[i]));
                                        i++;
                                        }
                                putchar(buffer[i]);

}
void buffer_datos(void){
                        i=0;
                        buffer[i]=getchar();
                        while(buffer[i] != 13){
                        i++;
                        buffer[i]=getchar();
                                }


}
void send_blockF(void){
        putchar('1');
        putchar('F');
        putchar(13);
}
void send_blockV(void){
                putchar('1');
                                        delay_ms(200);
                                        putchar('V');
                                        delay_ms(200);
                                        putchar(13);
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Programa del microcontrolador 5

#include <16f628A.h>
#FUSES INTRC_IO, NOWDT
#use delay (clock =4000000)
#use rs232(baud=9600, xmit=pin_b2, rcv=pin_b1, bits=8, parity=N)

char dir;
char buffer[10];
unsigned int i;
char par; //Toma el valor de falso o verdadero dependiendo de si el envio de comandos fue de manera correcta..

//Funciones
unsigned int paridad(unsigned int cte);
void buffer_datos(void);
void rev_paridad(void);
void CFD_b5_send(void);
void CFD_b5_rece(void);

#int_rda
void serial_isr(){
      dir=getchar();
      buffer_datos(); // Se reciben bloques de comandos provenientes del dispositivo maestro

       if(dir=='2'){
           if(buffer[1]==1 || buffer[1]==0){

             CFD_b5_send();//ctrl 2 a 1 para enviar datos.
             delay_ms(100);
              rev_paridad(); //Se revisa la paridad del bloque de comandos

              if(par=='F'){
                          putchar('2');
                          putchar('F');
                          putchar(13);
                          break;

                }else{
                          putchar('2');
                           delay_ms(100);
                          putchar('V');
                          putchar(13);

                  if(buffer[0]=='5'){
                     output_high(pin_A0); //Encender led
                  }else if(buffer[0]=='6'){
                     output_low(pin_A0); //Apagar led
                  }
           }
       } // fin de buffer[1]==1 ||  buffer[1]==0
    }//fin de dir==2
}//fin de interrupcion

void main(){

 enable_interrupts(global);
 enable_interrupts(int_rda);
```

```
CFD_b5_rece();//ctrl 2 a 1 para recibir datos.
output_low(pin_A0);

while(TRUE){

CFD_b5_rece();//ctrl 2 a 1 para recibir datos.
    }
}
void buffer_datos(void){

            i=0;
                    buffer[i]=getchar();
                    while(buffer[i] != 13){
                    i++;
                    buffer[i]=getchar();
                    }

}
void rev_paridad(void){

          i=0;
          while(buffer[i]!=13){ //...
                                                //revisión de la paridad de los datos provenientes del dispositivo maestro
                                                if(buffer[i+1]==paridad(buffer[i])){
                                                        i++;
                                                        i++;
                                                        par='V';
                                                }else{
                                                   par='F';
                                                   break;
                                                    }
                                }//...
}
void CFD_b5_send(void){
output_high(pin_B5);

}
void CFD_b5_rece(void){
output_low(pin_B5);
}
```

Esquemas de los dispositivos esclavo y maestro

Microcontrolador 1 (16F628A - U1)

Entradas: RB6, RB7, entradas digitales

Control de flujo de datos: RB4, salida digital

RX: RB1, entrada serial

TX: RB2, salida serial

Bus 1: RA0, RA1, RA2, RA3, salidas digitales

Bus 2: RA6, RA7, entradas digitales

Ctrl 2: RB3, entrada digital

Time out: RB0, salida digital

Microcontrolador 2 (16F887- U2)

RX: RC7, entrada serial

TX: RC6, salida serial

Bus 1: RB0, RB1, RB2, RB3, entradas digitales

Bus 2: RA0, RA1, salidas digitales

Bus 3: RD0, RD1, RD2, RD4, RD5, RD6, RD7, salidas digitales

Ctrl 2: RA5, salida digital

Time out: RA4, entrada digital

Microcontrolador 3 (16F628A - U3)

Control de flujo de datos: RB5, salida digital

RX: RB1, entrada serial

TX: RB2, salida serial

Ctrl 3: RB0, entrada digital

Ctrl 4: RB7, entrada digital

Bus 4: RA0, RA1, salidas digitales

Microcontrolador 4 (16F628A - U4)

RX: RB1, entrada serial

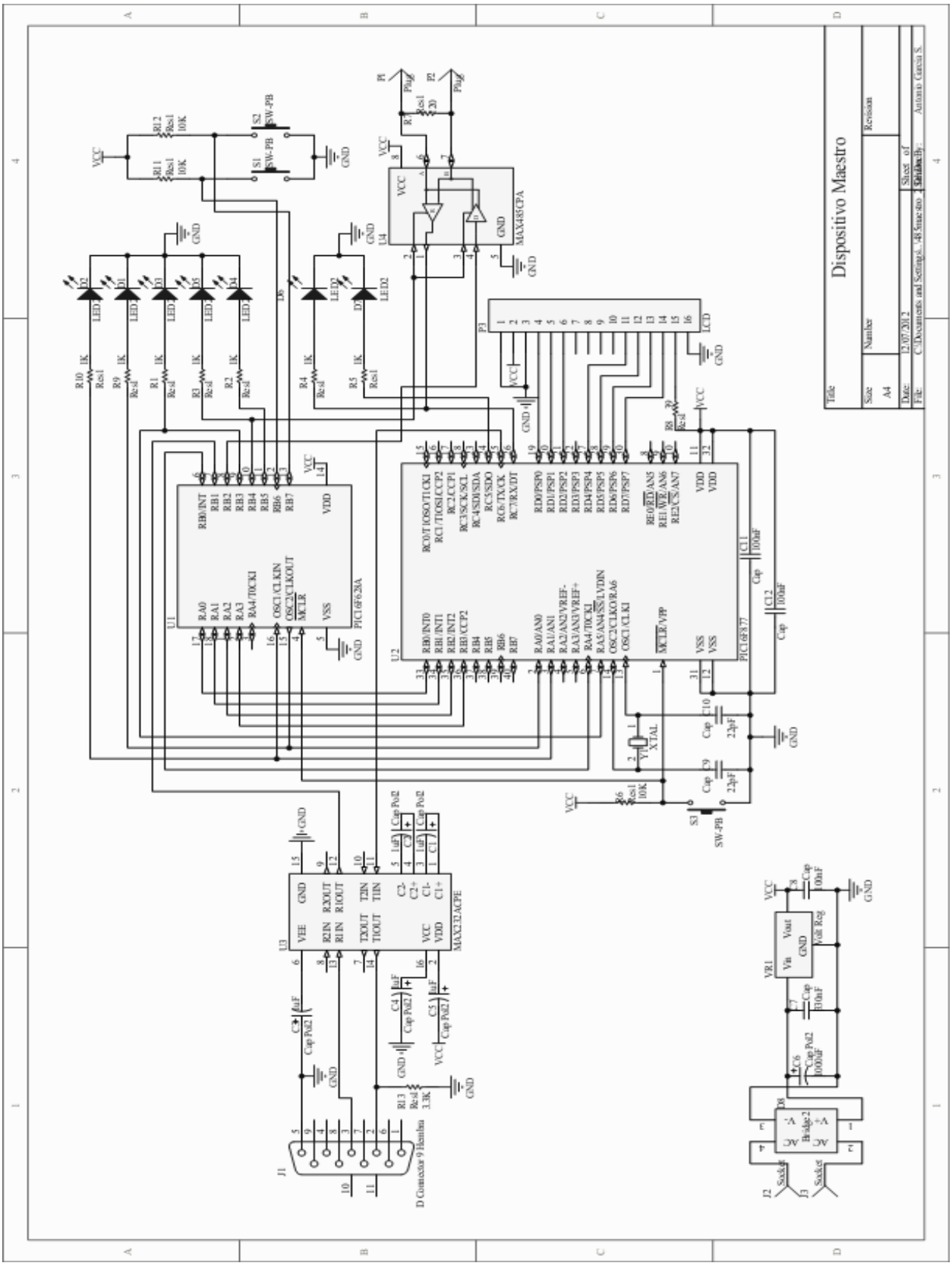
TX: RB2, salida serial

Ctrl 3: RB0, salida digital

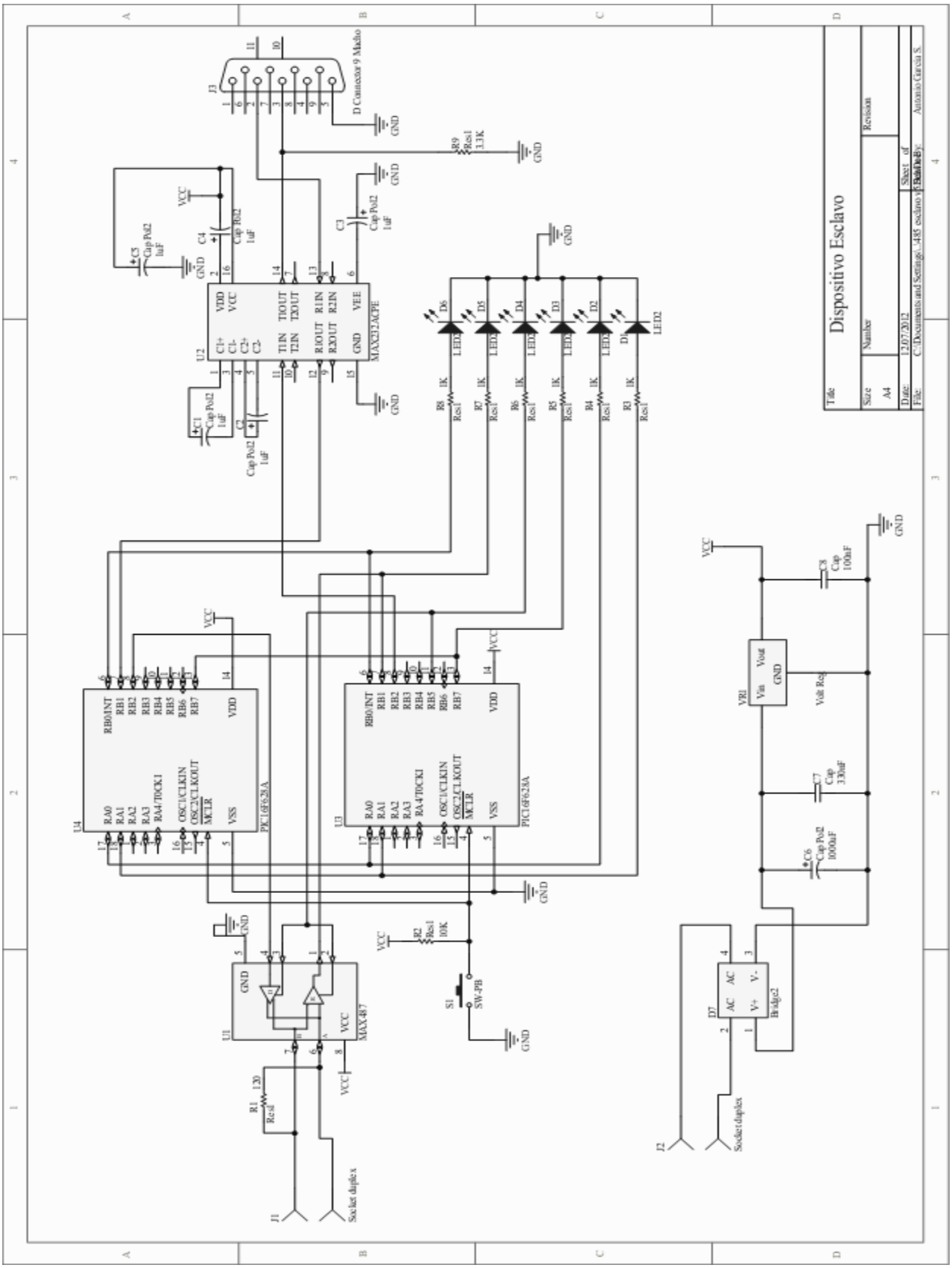
Ctrl 4: RB7, salida digital

Bus 4: RA0, RA1, entradas digitales

APÉNDICES


Dispositivo Maestro
Title
Size
A4
Number
Revision
Date:
12/07/2012
File:
C:\Documents and Settings\...\Maestro
Sheet of
Drawn By:
Antonio Garcia S.
PIC16F628A
PIC16F877
MAX485CPA
MAX232ACPE
LCD
XTAL
SW-PB
Volt Reg
Bridge 2
D Connector 9 Hembra

APÉNDICES


Dispositivo Esclavo
Title
Size
A4
Number
Revision
Date:
12/07/2012
Sheet of
File:
C:\Documents and Settings\...\485 esclavo 3.
Drawn By:
Antonio Garcia S.
D Connector 9 Macho
MAX232ACPE
PIC16F628A
MAX487
Bridge2
Socket duplex
SW-PB
LED1
LED2
LED3
LED4
LED5
LED6
VCC
GND