ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

"Control mediante joystick de tarjeta AVR Butterfly (con microcontrolador ATMega169) mediante comunicación I2C con tarjeta LPCXpresso controladora de motor BLDC."

TESINA DE SEMINARIO

Previa la obtención del Título de:

INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES

Presentado por:

Lenin Fabian Toaza Oñate

John Jairo Caro Bermúdez

GUAYAQUIL – ECUADOR

AÑO 2013

AGRADECIMIENTO

Doy gracias a Dios en primer lugar por haberme ayudado a culminar de manera satisfactoria mis estudios superiores. A mis padres, amigos y seres queridos que con sus sabios consejos y apoyo incondicional supieron guiarme para alcanzar este gran objetivo.

Lenin Toaza

Agradezco a Dios por haberme ayudado en todo momento, a mis padres por apoyarme en mis estudios y a mis hermanos por aconsejarme y animarme en conseguir mis metas.

John Caro

DEDICATORIA

A mis padres y hermanas por estar siempre brindándome esa confianza que siempre necesite.

Lenin Toaza

A mis padres y hermanos porque siempre estuvieron inculcándome por el camino del saber.

John Caro

TRIBUNAL DE SUSTENTACIÓN

Ing. Carlos Valdivieso A.
DIRECTOR DE PROYECTO

Ignacio Marín García MSIS
PROFESOR DELEGADO POR LA UNIDAD ACADÉMICA

DECLARACIÓN EXPRESA

“La responsabilidad del contenido de esta tesina de graduación nos corresponde exclusivamente; y el patrimonio intelectual de la misma a la ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL”

(Reglamento de exámenes y títulos profesionales de la ESPOL)

_____

LENIN TOAZA OÑATE

_____

JOHN CARO BERMÚDEZ

RESUMEN

El proyecto realizado en la materia de graduación *Microcontroladores Avanzados*, consiste en manejar y controlar un motor BLDC a través de un joystick combinando tarjetas de desarrollo como la LPCXpresso 1769 y la AVR Butterfly, utilizando comunicación I2C (Inter-Integrated Circuit). Para la comunicación entre estas tarjetas la AVR Butterfly trabajó en modalidad MAESTRO, mientras que la tarjeta LPCXpresso 1769 trabajó en modalidad ESCLAVO. Para la implementación de este proyecto se usó lenguaje C mediante el entorno LPCXpresso 4 y AVR Studio 4 respectivamente.

ÍNDICE GENERAL

CAPÍTULO 2

SUSTENTACIÓN TEÓRICA

CAPÍTULO 3

PRUEBAS DE MÓDULOS I2C DE LAS TARJETAS AVR

BUTTERFLY Y LPCXPRESSO

GLOSARIO

Bit Bit es el acrónimo Binary digit. (Dígito binario). Un bit es un dígito del sistema de numeración binario

Buffer Es un espacio de memoria, en el que se almacenan datos para evitar que el programa o recurso que los requiere, ya sea hardware o software, se quede sin datos durante una transferencia.

Byte Un byte es la unidad fundamental de datos en los ordenadores personales, un byte son ocho bits contiguos. El byte es también la unidad de medida básica para memoria, almacenando el equivalente a un carácter.

Debugger Es un depurador, y una aplicación o herramienta que permite la ejecución controlada de un programa o un código, para seguir cada instrucción ejecutada y localizar así bugs o errores (proceso de depuración), códigos de protección.

EEPROM Son las siglas de Electrically Erasable Programmable Read-Only Memory (Memoria solo de lectura programable y borrable eléctricamente). Es un tipo de memoria ROM que puede ser programada, borrada y

	reprogramada eléctricamente. Son memorias no volátiles.
Half duplex	Es un tipo de transmisión en el cual el envío de información es bidireccional, pero no simultáneamente.
Hardware	Partes o componentes físicos que integran una herramienta; inclusive ella misma como una unidad.
LPCXpresso	Tarjeta de desarrollo evaluación con microcontroladores de la empresa Next eXPerience Semiconductors**.**
Microcontrolador	Es un circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria. Está compuesto de varios bloques funcionales, los cuales cumplen una tarea específica.
Protoboard	Es un tablero con orificios conectados eléctricamente entre sí, habitualmente siguiendo patrones de líneas, en el cual se pueden insertar componentes electrónicos y cables para el armado de circuitos electrónicos y sistemas similares.
Pull up	Es una configuración electrónica usada en circuitos lógicos para asegurar que la entrada a un sistema, tenga los niveles lógicos de voltaje adecuados. Generalmente consta de una resistencia conectada entre la entrada y el voltaje de alimentación.

Software Es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación.

ABREVIATURAS

ACK: Acknowledgement, Acuse de recibo. Es un mensaje que se usa para confirmar la recepción de un mensaje.

AVR Advanced Virtual RISC, RISC Virtual Avanzado. También significa Alf Vergard RISC, en honor a sus creadores Alf Egil Bogen y Vergard Wollan.

FEM: Fuerza Electro Motriz.

I2C: Inter-Integrated Circuit. Significa Circuitos Inter Integrados, es un protocolo de comunicaciones serial que utiliza dos líneas para transmitir la información: una para los datos y la otra genera la señal de reloj.

IDE: Integrated Development Environment. Es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica.

ISP: In System Programming. Característica que permite grabar o leer un dispositivo sin tener que extraerlo de su aplicación/sistema.

LED: De la sigla inglesa LED: Light-Emitting Diode: ("diodo emisor de luz", también "diodo luminoso") es un diodo semiconductor que emite luz. Se usan como indicadores en muchos dispositivos, y cada vez con mucha más frecuencia, en iluminación.

NACK: Negative Acknowledgement. Es un mensaje que se usa para informar que hubo un error en la recepción de una trama de datos.

NXP: Next eXPerience Semiconductors. Nombre de una compañía fundada por la empresa Koninklijke Philips Electronics N.V. en el 2006.

RISC: Reduced Instruction Set Computer. Es un modelo de diseño de CPU usado en microcontroladores y microprocesadores.

RPM: Revoluciones por minuto.

SCL: Serial Clock. Es la señal de reloj serial que se usa en la comunicación I2C para la sincronización entre los multiples dispositivos conectados al Bus.

SDA: Serial Data. Es la línea del Bus I2C por la cual se transmiten los datos.

TWI: Two wire interface. Significa Interface de dos cables, uno para trasmitir datos y el otro genera la señal de reloj, para la trasmisión serial de datos.

USI: Interface Universal de comunicación serial.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

El objetivo del trabajo fue el diseño e implementación de un control mediante Joystick de un motor BLDC, mediante comunicación I2C con tarjeta LPCXpresso. Teniendo en cuenta varios aspectos muy importantes como es la compatibilidad del LPCXpresso con otras tarjetas y combinarlas para tener un desarrollo en conjunto.

La tarjeta AVR Butterfly mediante su joystick es la que controla el sentido y la velocidad de giro del motor BLDC. Esta tarjeta envía señales a la LPCXpresso que es la tarjeta controladora del motor BLDC mediante comunicación I2C actuando la tarjeta AVR como Maestro y la LPCXpresso que recibirá esta comunicación actuará como Esclavo.

.

CAPÍTULO 1

DESCRIPCIÓN GENERAL DEL PROYECTO

El presente capítulo contiene los antecedentes que conlleva el proyecto, la motivación para hacerlo. También se indica e identifica el problema y su necesidad de crear aplicaciones prácticas para manejar los motores BLDC por medio del uso de tarjetas de desarrollo basadas en microcontroladores. El capítulo incluye además los objetivos principales y las limitaciones de alcance que tiene el proyecto.

1.1. Antecedentes

Desde la invención de los motores eléctricos ha existido la necesidad de controlar la velocidad, direccionamiento y conmutación de este tipo de motores. Con cada avance en la tecnología estos métodos de control se han vuelto más sofisticados usándose nuevas herramientas para desarrollar mejores soluciones.

La integración de los microcontroladores y sensores en los sistemas de control de motores eléctricos mejorado su rendimiento y eficiencia, puesto

que el sensor provee información más precisa acerca de la posición en la que se encuentra el motor.

1.2. Motivación

La comunicación serial I2C, para las tarjetas AVRbuttefly y LPCXpresso, se han implementado entre dispositivos de la misma familia. La motivación se formó en el hecho de aportar una misma comunicación I2C, pero entre las diferentes familias (AVRbuttefly con LPCXpresso), la cual puede controlar mediante señales la velocidad y giro del motor BLDC. Que aparte de ser algo novedoso, es económico y su implementación en hardware es sencilla.

1.3. Identificación del problema

En la actualidad, muchas son las industrias que requieren controlar la velocidad y el giro de motores eléctricos, para ello el operador debe controlar cada motor de forma independiente. Como solución a este problema se plantea la implementación del protocolo de comunicación serial I2C, que es un protocolo half-duplex basado en una estructura maestro-esclavo y que permite al operador manejar múltiples dispositivos desde un mismo lugar y desde una misma tarjeta controladora.

.

1.4. Objetivos

Los objetivos generales y específicos del presente proyecto, se detallan a continuación.

1.4.1. Objetivos Generales

El objetivo principal de este proyecto es implementar un sistema de control maestro-esclavo, para un motor BLDC mediante comunicación serial I2C que permita aumentar y disminuir la velocidad de giro e invertir el sentido del giro del motor mediante un joystick.

1.4.2. Objetivos Específicos

- Implementar un sistema de comunicación serial, utilizando el protocolo I2C.

- Desarrollar un código de fácil entendimiento que optimice los recursos de los procesos realizados por el software para la implementación del control del motor BLDC.

- Usar herramientas para el diseño de interfaces, basados en microcontroladores avanzados.

- Realizar una implementación en hardware que sea lo más simple posible, de tal manera que se puedan realizar las combinaciones Maestro - esclavo entre las tarjetas AVR Butterfly y LPCXpresso.

CAPÍTULO 2

SUSTENTACIÓN TEÓRICA

El presente capítulo incluye la base teórica y la explicación de conceptos necesarios para la realización del proyecto en curso. Asimismo incluye de manera básica el funcionamiento y las características de los motores BLDC, así como la teoría del funcionamiento del protocolo de comunicación serial I2C para así poder aplicarla en el proyecto y contribuir a la solución del problema planteado en el Capítulo 1.

2.1. Protocolo I2C

I2C es un protocolo que facilita la comunicación entre distintos dispositivos como microcontroladores, memorias, monitores de computadoras, y muchos otros dispositivos.

El protocolo I2C utiliza un bus que requiere de dos señales: Una de datos y una de reloj (además de la tierra común). Este protocolo fue diseñado por la

empresa Koninklijke Philips Electronics N.V. inicialmente con una tasa de transmisión de datos de 100Kbps, sin embargo actualmente se alcanzan tasas de hasta 3.4Mbps.

La señal de datos se llama SDA y la de reloj se llama SCL o SCK. Como son señales de colector abierto, es necesario agregar resistencias de pull-up que pueden tener valores entre 2KΩ y 50KΩ siendo 10KΩ su valor estándar.

El estado de reposo de ambas líneas en el bus I2C es el "1" lógico, de ahí el significado de las resistencias pull-up. El estándar I2C permite una comunicación bidireccional con los periféricos y además es posible crear entornos multi-maestros sin que entre ellos haya conflictos, tal como muestra la Figura 2.1.

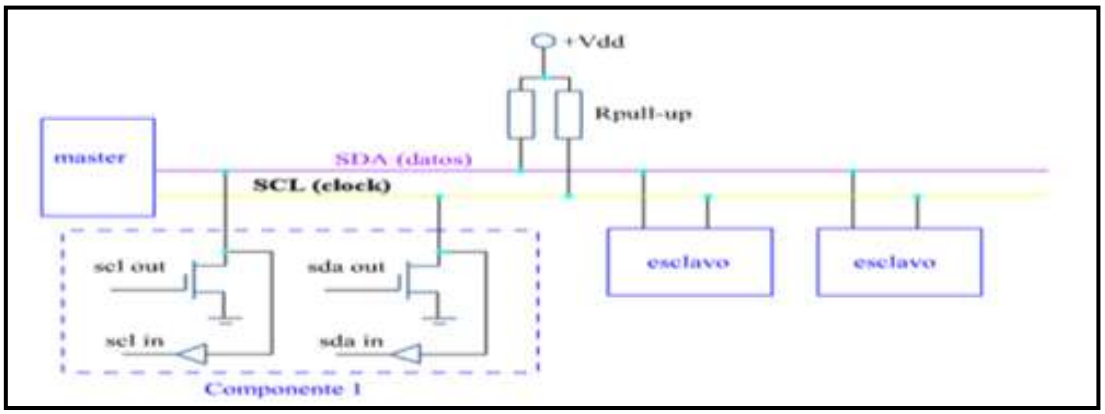

Figura 2.1 Bus I2C

El protocolo I2C también permite tener conectados al bus dispositivos que funcionen con voltajes de alimentación distintos, siempre que el voltaje de SDA y SCL sea menor que todos los voltajes de polarización de los dispositivos conectados al bus I2C y que los niveles lógicos del bus (alto y

bajo) puedan ser interpretados correctamente por cada uno de los dispositivos conectados.

2.1.1. Transferencia de datos

En la comunicación I2C existen algunas normas para que la información que se desea transmitir llegue sin problemas desde un maestro a un esclavo o a otro maestro que trabaje como esclavo. La secuencia a seguir por el maestro que quiere tomar el control del bus es inicialmente siempre la misma:

Primero envía la señal de INICIO, con lo que todos los demás componentes que no la hayan enviado se pondrán como esclavos (incluidos el resto de maestros), luego envía la dirección I2C (de 7 bits de longitud) del dispositivo al que se quiere dirigir (esta dirección será única en todo el bus), de tal forma que todos los componentes que no tengan esta dirección se desconectan. Éste dispositivo enviará al maestro una señal de reconocimiento (ACK) para que sepa que está a la espera, entonces el maestro enviará 8 bits del dato a transmitir, le responde con otro ACK, y el maestro seguirá enviando 8 bits de datos y recibiendo un ACK hasta que termine con la información que desea transmitir.

Una vez que ha terminado con la información pueden pasar dos cosas:

El maestro que tiene el bus, al terminar de establecer una comunicación con un esclavo quiera dirigirse a otro, con lo que enviará una nueva señal de INICIO y se siguen los pasos explicados anteriormente.

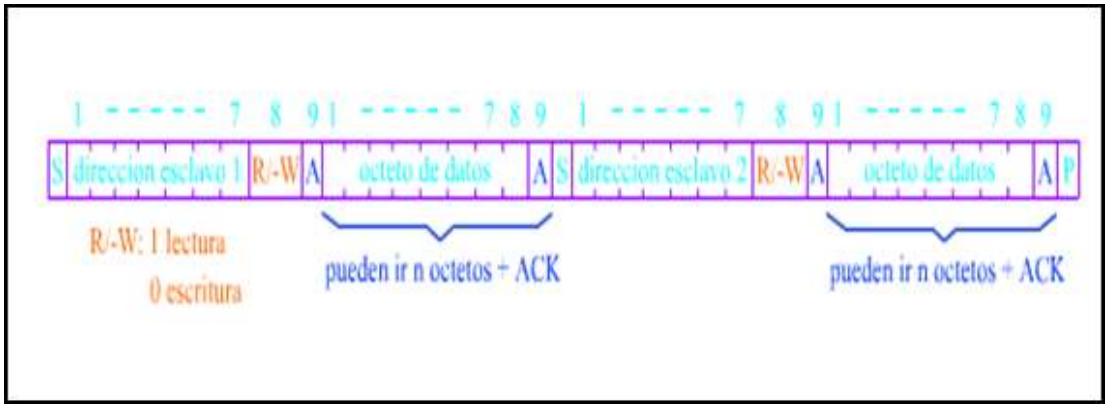

Figura 2.2 Formato I2C con inicio repetido

El maestro quiera terminar, con lo que envía la señal de PARADA y se detiene la transferencia tal como muestra la Figura 2.2.

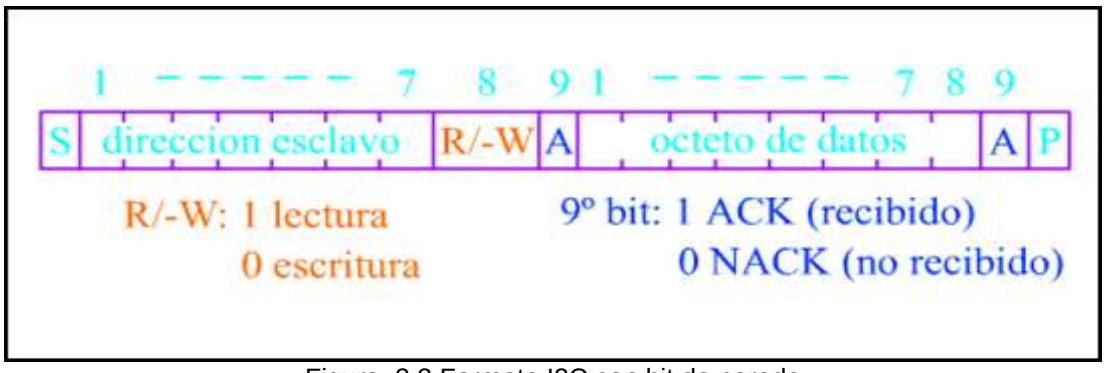

Figura 2.3 Formato I2C con bit de parada

En el caso que la información no llegue correctamente el esclavo enviará un NACK ("0") en vez de un ACK ("1"). Si esto sucede el maestro deberá enviar la señal de PARADA e intentará volver a

enviar la información repitiendo la secuencia de pasos desde el principio, tal como muestra la Figura 2.3.

2.1.2. Formato de mensaje

Las señales que se envían por el bus I2C tienen un formato estandarizado:

Señal de INICIO (S): siempre que partiendo de un estado de reposo (SDA y SCL en "1") se da la condición de que la SDA se ponga en "0" mientras la del reloj está en "1", generándose la señal de INICIO.

Señal de PARADA (P): si sucede lo contrario, es decir, que mientras la SCL está en "1" y SDA pasa de "0" a "1", se generara la señal de PARADA y la comunicación se termina liberándose el bus.

Datos: todas las demás señales se enviarán de igual manera, incluyéndose los bits de lectura o escritura (R/-W), ACK (A), NACK (N), datos o direcciones. Todos ellos se reconocen solo mientras SCL está en "1" y SDA debe mantener su tensión estable porque de lo contrario se generaría una señal de INICIO o PARADA en un lugar que no le corresponde.

La implementación de un sistema I2C se encargará de descifrar si el bit recibido o enviado es de datos o de dirección, en función de la posición en la que se encuentre.

Los datos se empiezan a enviar desde el bit más significativo y terminan con el menos significativo. [5]

2.2. Módulo I2C de la tarjeta AVR Butterfly

La tarjeta AVR Butterfly cuenta con una Interface Universal de comunicación serial que contiene los recursos de hardware necesarios para la comunicación serial. Esta interface cuenta con dos modos de funcionamiento, el modo de tres cables, ideal para la comunicación SPI (Interface serial periférica) y el modo de dos cables que es el que se va a utilizar para realizar la comunicación I2C.

La Figura 2.4 muestra dos unidades USI operando en modo de dos cables, una como maestro, y la otra como esclavo. La principal diferencia entre el modo de operación del maestro y del esclavo es que la señal de reloj es siempre generada por el maestro y solo el esclavo usa la unidad de control de reloj. La señal de reloj debe ser implementada por software, sin embargo la operación de desplazamiento de bits es realizada automáticamente por ambos dispositivos. [11]

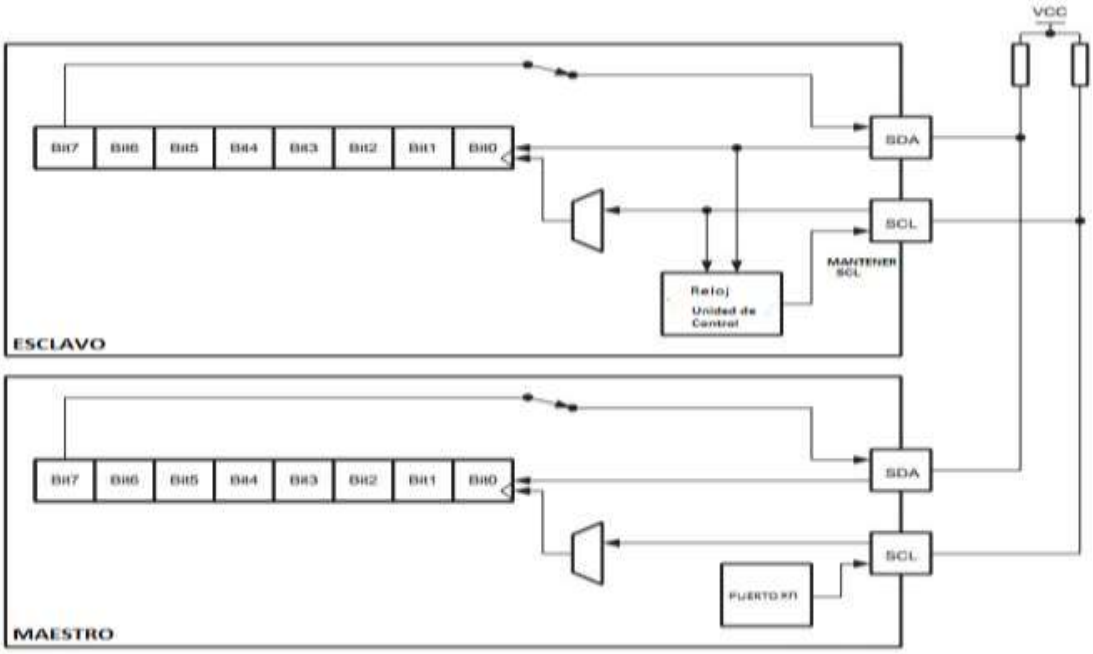

Figura 2.4 Unidades USI esclavo y maestro [11]

2.3. Módulo I2C de la tarjeta LPCXpresso 1769

La tarjeta LPCXpresso 1769 cuenta con tres módulos I2C llamados I2C0, I2C1 e I2C2, y cada módulo es orientado a bytes y tienen cuatro modos de operación: Maestro transmisor, maestro receptor, esclavo transmisor, esclavo receptor.

El módulo I2C0 tiene la habilidad de aislarse eléctricamente de los demás dispositivos conectados al bus I2C, cuando la tarjeta LPCXpresso 1769 se encuentra apagada, además soporta entornos multi-maestros y velocidades de hasta 1Mhz.

El modo de operación que usamos en este proyecto para la LPCXpresso 1769 es el modo esclavo receptor. Para inicializar este modo de operación

es necesario escribir en cualquiera de los cuatro registros de dirección de esclavo (I2ADR0-3) con los que cuenta la tarjeta LPCXpresso 1769, además se debe inicializar el registro de control I2CONSET como se muestra en la Tabla 2.1.

Bit	7	6	5	4	3	2	1	0
Symbol	-	I2EN	STA	STO	SI	AA	-	-
Value	-	1	0	0	0	1	-	-

Tabla 2.1 Registro de control I2CONSET [8]

El bit I2EN debe estar en alto para habilitar las funciones del I2C, asimismo el bit AA debe estar en alto para que el esclavo envíe ACK cuando recibe su dirección de esclavo. Y los bits STA, STO y SI bits deben estar en un nivel lógico bajo.

Después de que los registros I2ADR e I2CONSET se han inicializado, la interface I2C espera a que le llegue una dirección de esclavo que concuerde con alguna de las direcciones previamente grabadas en los registros de dirección (I2ADR0-3), junto con el bit de lectura o escritura. Si este bit de dirección es 0, significa escritura, por lo tanto el módulo entra en el modo de operación de esclavo receptor.

En cambio, si este bit es 1, significa lectura, y el módulo entra en el modo de operación de esclavo transmisor. Luego que el módulo ha recibido la dirección del esclavo junto con el bit de lectura o escritura, el bit SI del registro de control I2CONSET se cambia a alto, lo que permite al módulo leer

un código de estado del bus I2C, proveniente del registro de estados (I2CSTAT).[8]

2.4. Motores BLDC

Entre los motores eléctricos con mayor acogida debido a su alta eficiencia se encuentran los motores BRUSHLESS DC también llamados motores DC síncronos sin escobillas. Estos motores BLDC se emplean en sectores industriales tales como:

Automovilísticos, Aeroespaciales, Consumo, Médico, equipos de automatización e instrumentación.

Los bobinados de un motor sin escobillas (BLDC) están distribuidos a lo largo del estator en múltiples fases. Estos motores constan típicamente de tres fases con una separación entre sí de 120 grados.

2.4.1. Ventajas

Los motores BLDC tienen muchas ventajas sobre motores de corriente continua y motores de inducción. Como por ejemplo:

- Conmutan electrónicamente
- Respuesta dinámica de alta eficiencia
- Larga vida útil
- Funcionamiento silencioso

- Mayores rangos de velocidad

2.4.2. Desventajas

Por otra parte, los motores BLDC tienen dos desventajas, que son las siguientes:

- Tienen un mayor costo.
- Requieren un control mucho más complejo.

2.4.3. Principios básicos de operación

Los motores BLDC son un tipo de motor síncrono. Esto significa que el campo magnético generado por el estator y el campo magnético generado por el rotor gira a la misma frecuencia. Los motores BLDC no experimentan "Deslizamiento" que se observa normalmente en motores de inducción. Los motores BLDC vienen en una sola fase, bifásicos y en configuraciones trifásicas, correspondientes a su tipo, el estator tiene el mismo número de arrollamientos. De estos 3, los motores trifásicos son los más populares y ampliamente utilizados.

2.4.4. Partes del motor BLDC

El estator de un motor BLDC consta de aceros apilados, laminados con arrollamientos colocados en las ranuras que están a lo largo de la periferia interior. Tradicionalmente, el estator se asemeja al de un motor de inducción, sin embargo, los devanados se distribuyen de una manera diferente.

La mayoría de los motores BLDC tienen tres bobinados del estator conectado en estrella. Cada uno de estos devanados se construye con numerosas bobinas interconectadas para formar un arrollamiento. Una o más bobinas son colocadas en las ranuras y que están interconectados para hacer un devanado.

Cada uno de estos devanados se distribuye sobre la periferia del estator para formar un número par de polos. Hay dos tipos de bobinados del estator variantes: motores trapezoidales y sinusoidales. Esta diferenciación se hace sobre la base de la interconexión de las bobinas en los bobinados del estator para darles diferentes tipos de Fuerza electromotriz (FEM).

Además de la fuerza contra electromotriz, la fase actual también tiene variaciones trapezoidales y sinusoidales en los respectivos tipos de motor. Esto hace que el par de salida por un motor sinusoidal sea más suave que la de un motor trapezoidal.

Sin embargo, esto viene con un costo adicional, como los motores sinusoidales toman devanado extra e interconexiones debido a la distribución de bobinas en la periferia del estator, aumentan el consumo del cobre por los bobinados del estator. Dependiendo de la capacidad de control de la fuente de alimentación, el motor puede ser elegido con el voltaje correcto del estator.

El rotor está hecho de imanes permanentes y pueden variar de dos a ocho pares de polos con alternantes del Norte (N) y Sur (S).

En base a la densidad de campo magnético requerido en el rotor, el material magnético adecuado se elige para hacer el rotor. Los imanes de ferrita se utilizan tradicionalmente para hacer imanes permanentes.

Para girar el motor BLDC, los bobinados del estator tienen que ser energizados en una secuencia. Es importante conocer la posición del rotor con el fin de entender, que devanado se energizará siguiendo la secuencia depuesta en tensión. La posición de los rotores detecta el uso de sensores de efecto Hall integrados en el estator. La mayoría de los motores BLDC tienen tres sensores de efecto Hall integrado en el estator en el extremo no accionado del motor. Cada vez que los polos magnéticos del rotor pasan cerca de los sensores Hall, dan una señal de alto o bajo, lo que indica el

Norte o Sur pasando cerca de los sensores. Basado en la combinación de estas tres señales de los sensores Hall, la secuencia exacta de conmutación puede ser determinada. [3]

CAPÍTULO 3

PRUEBAS DE MÓDULOS I2C DE LAS TARJETAS AVR BUTTERFLY Y LPCXPRESSO

El presente capítulo contiene todo lo relacionado a los ejercicios que se han realizado previos a la implementación del proyecto. Antes de poder realizar la comunicación entre las dos tarjetas, nos vimos en la necesidad de probar las interfaces de comunicación de cada tarjeta con otros dispositivos que también soportan el protocolo I2C explicado en el capítulo previo.

Las pruebas que han sido realizadas son:

- Comunicación I2C de la LPC 1769 con la memoria EEPROM 24LC32A
- Comunicación I2C entre PIC16f887 y LPC 1769
- Comunicación I2C entre AVR Butterfly y EEPROM usando joystick.

En este capítulo se encuentra la descripción de cada prueba con su respectivo diagrama de bloques y de flujo.

3.1. Comunicación I2C de la LPC 1769 con EEPROM 24LC32A

En esta prueba se ha realizado la comunicación serial I2C entre la LPCXpresso 1769 y una EEPROM 24LC32A de 4 KB de memoria.

El microcontrolador LPC 1769 trabaja como maestro, mientras que la memoria EEPROM trabaja como esclavo.

La prueba consiste en enviar desde el microcontrolador 8 bytes, y escribirlos en la memoria EEPROM, en una cierta localidad de memoria, luego el microcontrolador lee estos datos, los almacena en un buffer y finalmente los envía al puerto 2 para ser visualizados por LEDS, tal como se ilustra en el diagrama de bloques de la Figura 3.1.

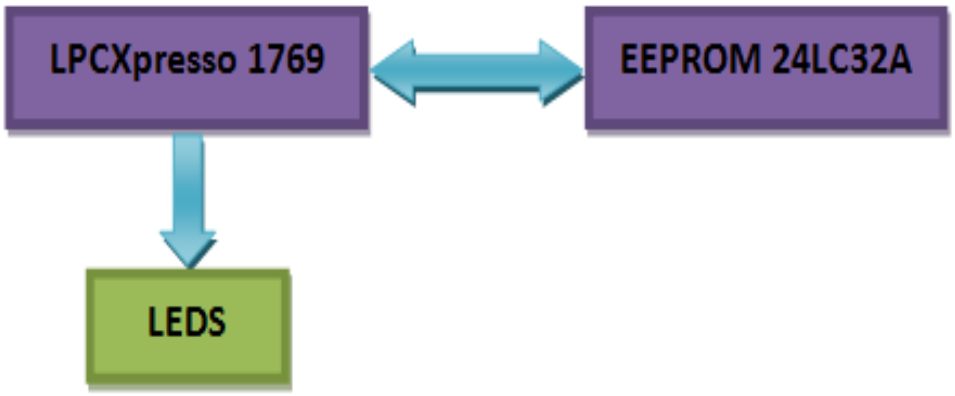

Figura 3.1 Diagrama de ejercicio 1

3.1.1. Descripción del algoritmo

Primero se envía la dirección de esclavo de la EEPROM, que en este caso es 0xA8, luego la posición de memoria, como es una

memoria EEPROM de 4KB se necesitan 2 bytes para acceder a una localidad de memoria, ya sea para leer como para escribir. Primero se envía el byte más significativo, luego el segundo byte. Cabe recalcar que el byte más significativo debe estar entre el rango de 0x00 y 0x0F, cualquier intento de escribir en direcciones superiores, es fallido, debido a que se ignoran los 4 bits más significativos del primer byte de dirección.

Después que se envían los bytes de memoria se envían los datos a escribir, que son “0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80” como se puede apreciar en el código en el ANEXO A

Luego que estos datos se escriben en la EEPROM a partir de la localidad de memoria especificada, se procede a leer los datos previamente escritos a partir de la misma localidad de memoria en donde se escribieron.

Para ello es necesario escribir nuevamente la dirección del esclavo 0xA8, seguido de los bytes de dirección de memoria, para posteriormente realizar un inicio repetido, que consiste en enviar nuevamente una señal de inicio, seguido de la dirección del esclavo junto con el bit de lectura, es decir 0xA9.

Una vez que se haya efectuado la lectura, los datos se almacenan en el buffer de lectura del LPCXpresso 1769, y luego son enviados al puerto 2, para ser visualizados en los LEDS en un lazo infinito tal como se ilustra en el diagrama de flujo de la Figura 3.2.

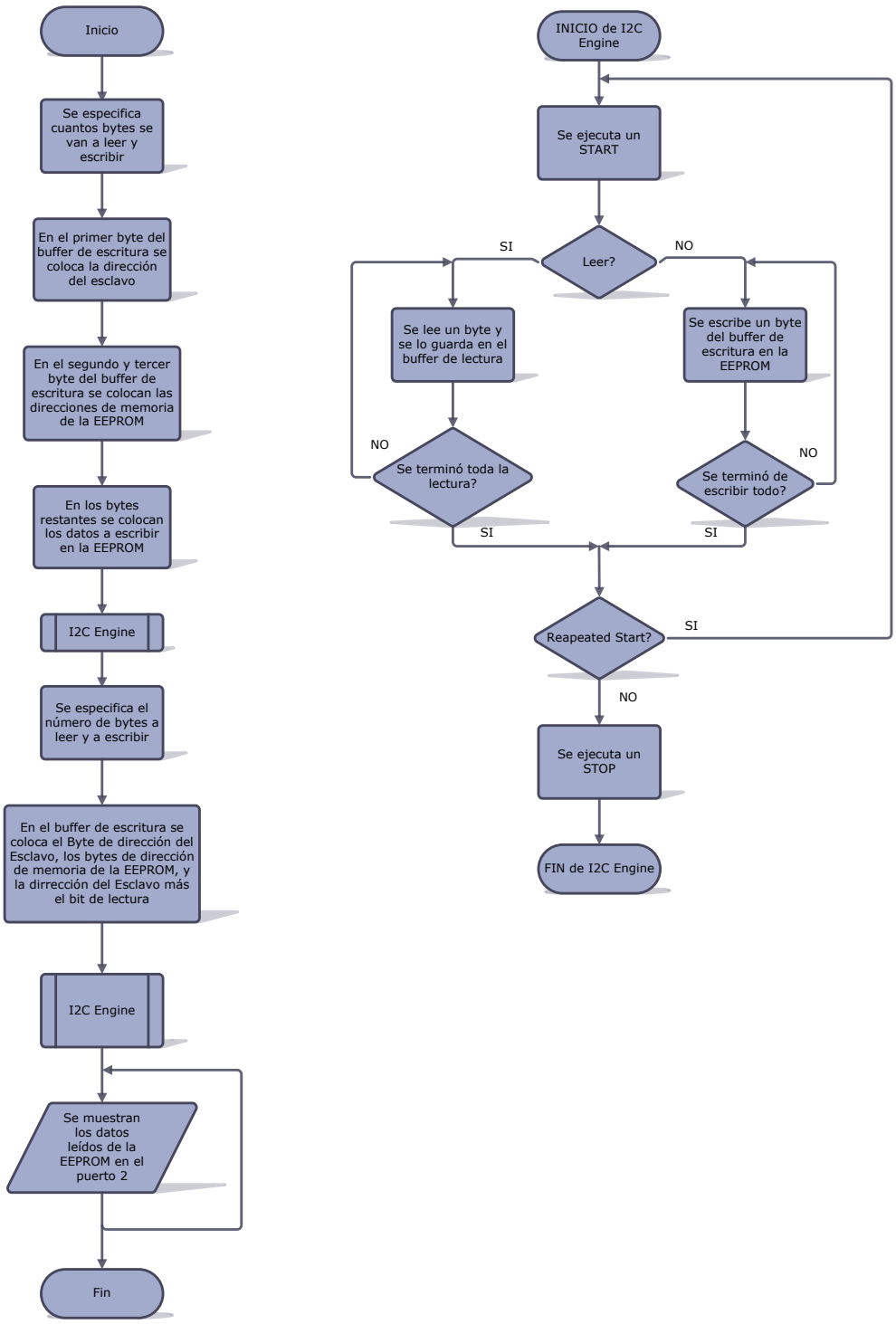

Figura 3.2 Diagrama de flujos del programa del LPCXpresso 1769

3.1.2. Implementación

Para la implementación en protoboard se utilizó una fuente de voltaje de 3 voltios, dos resistencias pull-up de 10KΩ para las lineas SDA y SCL, y resistencias de 220Ω para limitar la corriente en los LEDS.

En la Figura 3.3 se visualiza el resultado de los cuatro primeros bytes que se leen de la memoria EEPROM 24LC32A, la proyección del byte 0x01 en el puerto 2 da como resultado que el primer LED se encienda mientras los demás permanecen apagados, el byte 0x02 enciende el segundo LED y apaga los restantes, el byte 0x04 enciende el tercero, y el 0x08 enciende el cuarto LED mientras apaga los demás.

Cada byte es proyectado en el puerto 2 por 600ms, con el fin de que el resultado sea perceptible para el ojo humano.

Como resultado final debe observarse rotar los LEDS en el puerto 2.

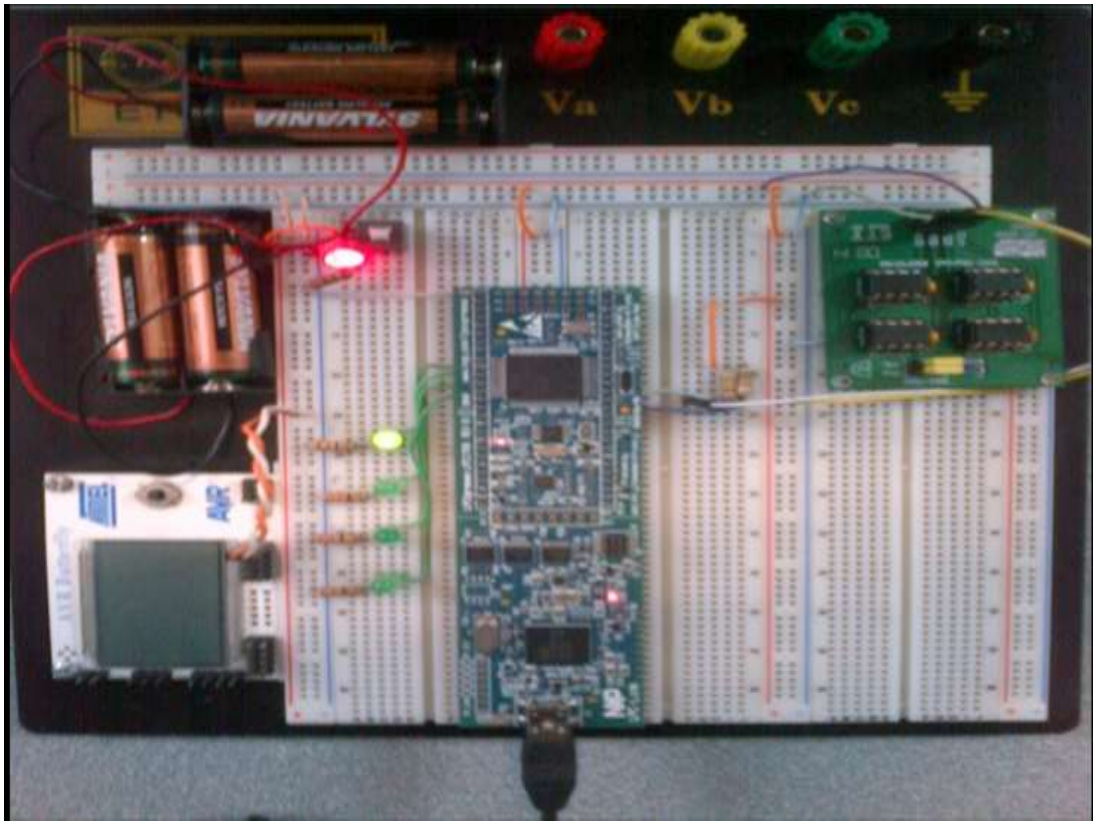

Figura 3.3 Implementacion fisica ejercicio 1

3.1.3. Conclusión de la Implementación

Al finalizar esta prueba se ha logrado comprender el funcionamiento del módulo de comunicación I2C de la LPCXpresso 1769, las librerías y funciones que programaron los desarrolladores del software en los ejemplos. Para entender dichos códigos, se ha procedido a analizar detalladamente los ejemplos de comunicación I2C como maestro, y posteriormente modificarlos según nuestras necesidades.

3.2. Comunicación I2C entre PIC16f887 y LPC 1769

En la presente prueba se ha implementado la comunicación I2C entre el PIC16F887 como maestro, y el LPCXpresso 1769 como esclavo.

Como el PIC funciona como maestro, es el encargado de enviar la señal de INICIO seguido de la dirección del esclavo, en este caso la dirección que se ha configurado en el LPC 1769, que es “0xA0”.

Luego que la dirección enviada por el PIC coincida con la dirección del esclavo del LPC, éste enviará un acuse de recibo ACK, y luego el PIC procederá a enviar los 8 datos que se almacenarán en el buffer de lectura del LPC, para posteriormente mostrarlos en el puerto 2 y visualizar la rotación en los LEDS, tal como se esquematiza en la figura 3.4.

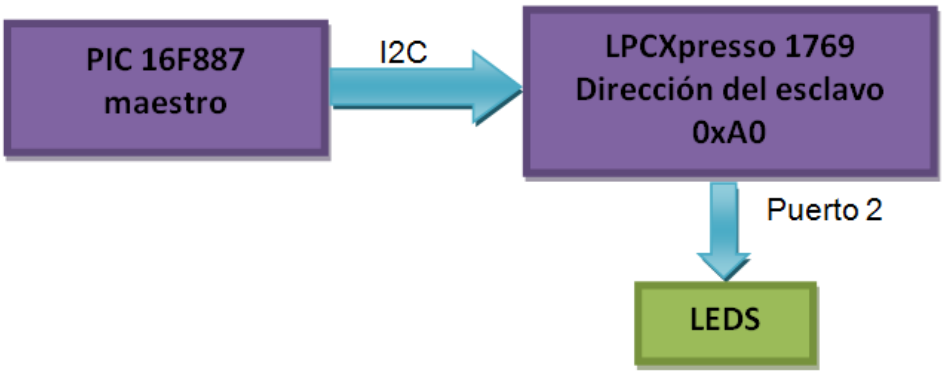

Figura 3.4 Diagrama de bloque ejercicio 2

3.2.1. Descripción del algoritmo del LPC 1769 como esclavo

Primero se procede a declarar e inicializar las variables que se van a usar en la programación, asimismo es necesario inicializar y

configurar al puerto 2 del LPC 1769 como salida ya que este puerto es el que muestra los resultados mediante LEDS. Luego se procede a encerar el buffer de lectura en donde se van a almacenar los datos leídos, y después se inicializa el módulo I2C como esclavo. Una vez inicializado el módulo I2C como esclavo el LPC 1769 está listo para la comunicación, y apenas reciba del maestro una dirección del esclavo que coincida con una de las direcciones previamente grabadas en sus registros de direcciones, el LPC envía un acuse de recibo ACK.

El bit de lectura o escritura que recibe el LPC 1769 es un 0, por lo tanto el LPC 1769 se prepara a leer los datos del maestro.

Cada vez que la LPCXpresso recibe un byte, envía su respectivo acuse de recibo ACK y almacena este byte en el buffer de lectura, y procede a mostrar cada byte en el puerto 2, con su respectivo retardo para facilitar la visualización tal como se detalla en el diagrama de flujo de la Figura 3.5.

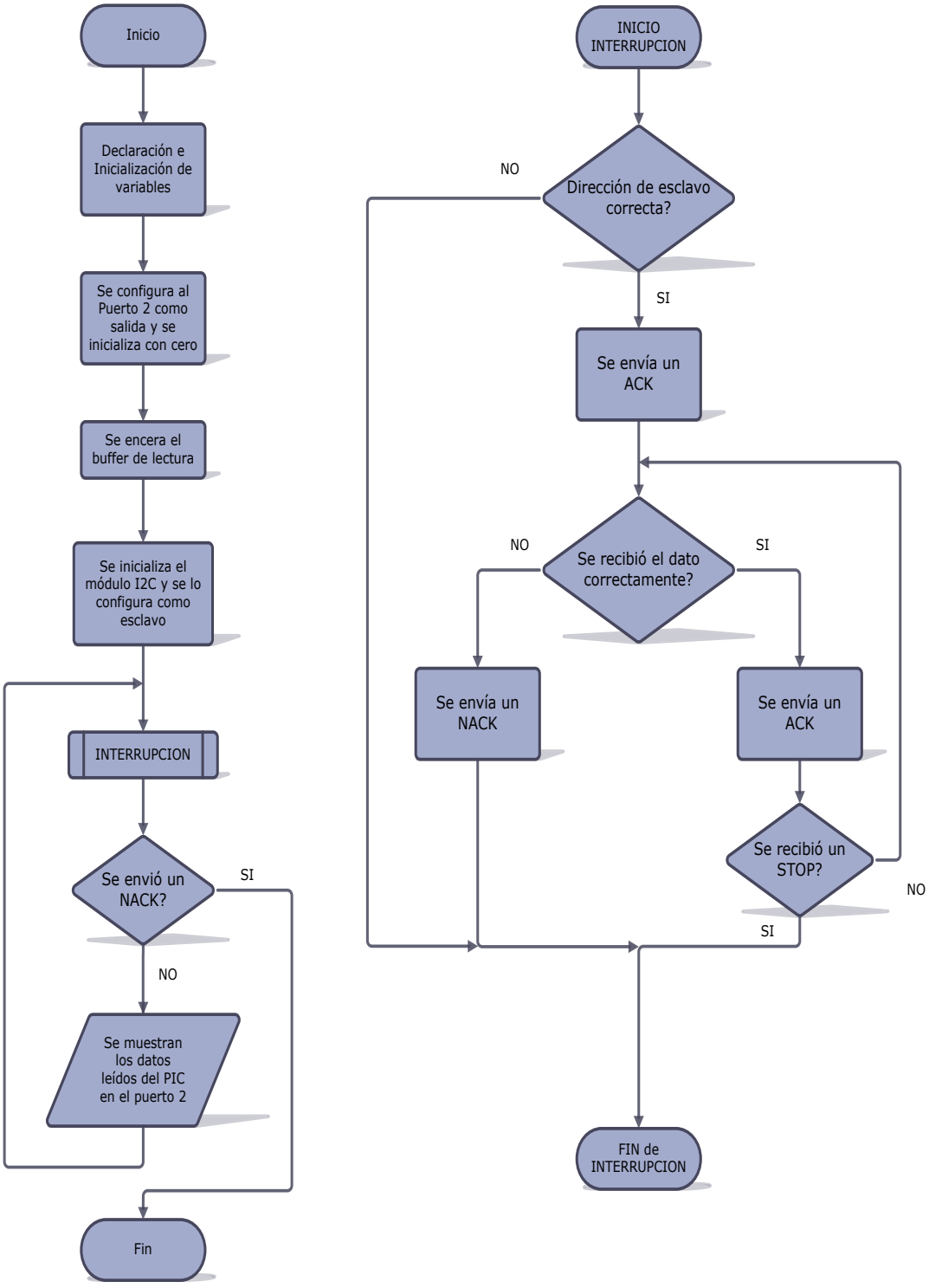

Figura 3.5 Diagrama de flujo del LPCXpresso 1769 como esclavo

3.2.2. Descripción del algoritmo del PIC 16F887

Primero se procede a inicializar los puertos y las variables que se van a utilizar, asimismo se inicializa el modulo I2C del PIC 16F887. Una vez terminadas las inicializaciones se envía la señal de inicio, seguida de la dirección del esclavo de la LPCXpresso 1769 que es “0xA0”, junto con el bit de escritura que es “0”, luego el PIC 16F887 espera por el acuse de recibo ACK, y cuando lo recibe, procede a enviar el primer byte de datos, asimismo cada vez que el PIC envía un byte tiene que esperar el acuse de recibo ACK para enviar el siguiente dato.

Cabe destacar que entre cada dato que se envía hay un retardo de 100 ms, con el fin de probar si el esclavo está listo para recibir un dato del maestro en cualquier momento y no sea necesario que los datos se envíen uno tras otro sin ninguna interrupción.
Finalmente, se envía la señal de parada una vez que el maestro haya enviado su último dato y haya recibido el ACK correspondiente.

Para indicar que la comunicación se ha establecido y ha finalizado con éxito se enciende un LED rojo en el puerto B del PIC 16F887 tal como se puede apreciar en el diagrama de flujo de la Figura 3.6.

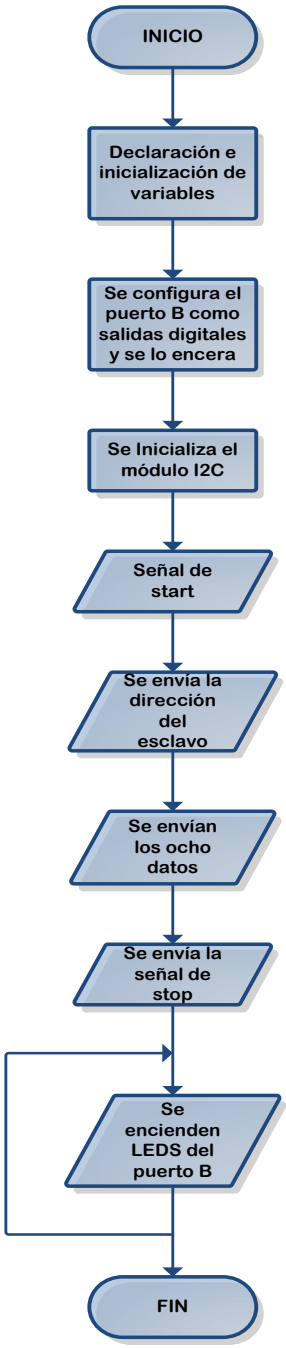

Figura 3.6 Diagrama de flujo del programa del PIC 16F887

3.2.3. Implementación

La imagen de la Figura 3.7 muestra al PIC16f887 y a la LPCXpresso 1769 estableciendo una comunicación I2C, siendo el PIC el maestro y el LPC 1769 el esclavo. El PIC16f887 se alimenta con una fuente de 5 voltios, mientras que el LPC se alimenta con una fuente de 3 voltios. Necesariamente los dos microcontroladores deben tener una tierra común, además el voltaje del bus I2C debe ser de 3 voltios que es el menor de los voltajes de alimentación de los dispositivos conectados al bus I2C.

Los bytes de datos que el PIC 16F887 envía al LPC 1769 son “0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80”, y son mostrados en el puerto 2 del LPC 1769 a través de LEDS, con su respectivo retardo, para percibir claramente una rotación en los LEDS tal como se explicó en el ejercicio previo.

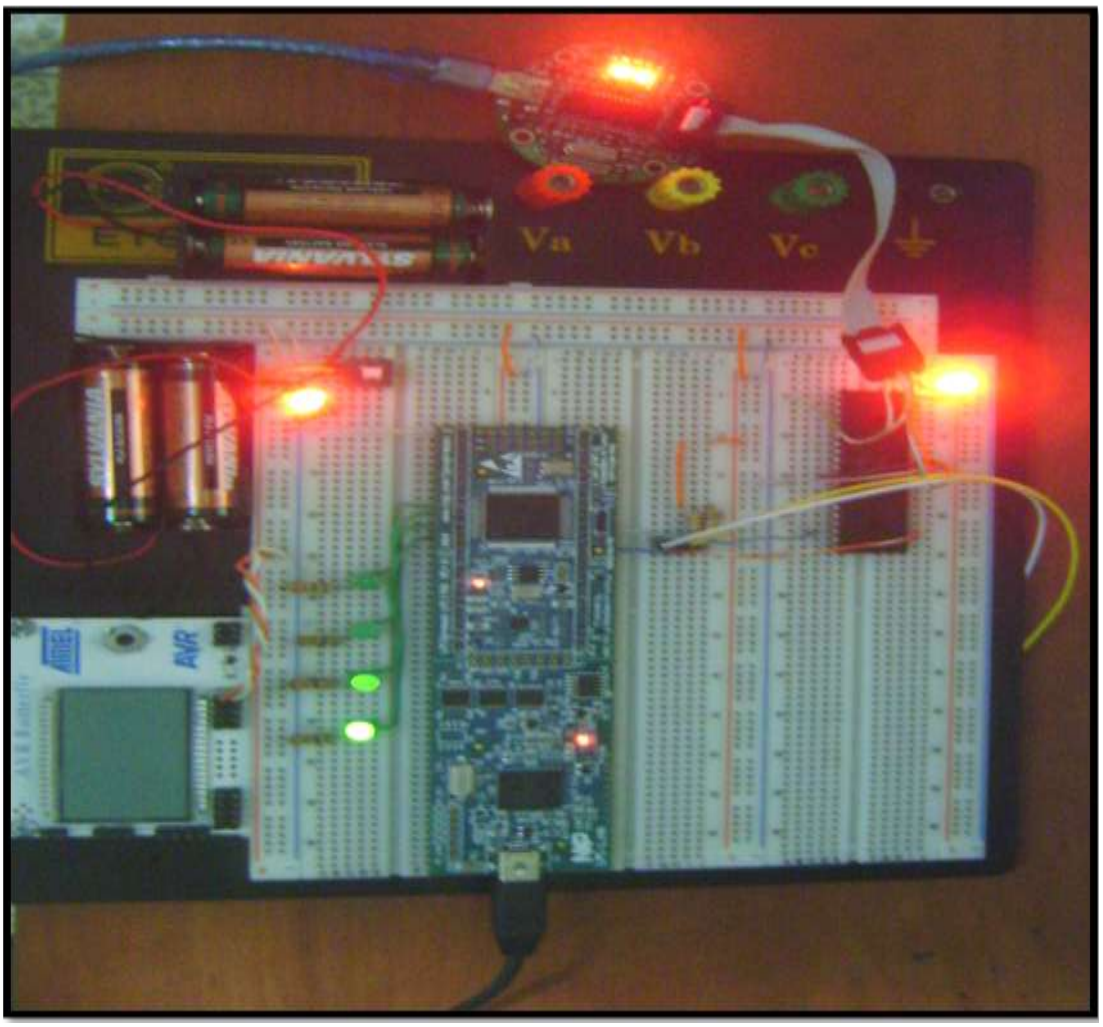

Figura 3.7 Implementacion fisica ejercicio 2

3.2.4. Conclusión de la Implementación

Luego de obtener resultados satisfactorios en esta prueba, se ha podido entender el funcionamiento de la LPCXpresso 1769 como esclavo, ya que una vez que se ejecuta el código en la LPCXpresso 1769, el módulo de comunicación I2C se encuentra listo para receptar y almacenar los datos que se reciban desde el

PIC 16F887 funcionando como maestro, inclusive cuando el maestro no envíe los datos de manera continua, sino con retardos variables entre datos.

3.3. Comunicación I2C entre AVR Butterfly y EEPROM usando joystick

En esta prueba se ha logrado establecer la comunicación I2C entre el AVR Butterfly y una memoria EEPROM 24LC32A.
El AVR Butterfly trabaja como maestro, y la memoria EEPROM trabaja como esclavo.

La prueba consiste en registrar en la memoria EEPROM cada movimiento que realiza el joystick de la tarjeta AVR Butterfly usando comunicación serial I2C.

Cada vez que se presione una posición en el joystick, se iniciará la comunicación I2C, es decir, el AVR Butterfly enviará una señal de INICIO, junto con la dirección del esclavo de la memoria EEPROM, luego los dos bytes de dirección de memoria de la EEPROM en donde se piensa escribir, y después el dato, que es un byte correspondiente a la posición del joystick que se haya elegido y finalmente envía la señal de PARADA.

3.3.1. Descripción del algoritmo

Primero se declaran e inicializan las variables que se van a utilizar durante la ejecución del programa, asimismo se inicializa el Joystick y el módulo de comunicación serial como maestro.

Antes de empezar a escribir en la memoria EEPROM es necesario enviar la dirección del esclavo correspondiente a la EEPROM en donde queremos escribir, junto con el bit de escritura. Además es necesario enviar dos bytes que especifican la dirección de memoria en donde se va a empezar a escribir los datos.

Cuando la memoria EEPROM haya en confirmado la recepción de los datos mediante un acuse de recibo ACK, ambos dispositivos estarán listos para comenzar la comunicación serial.

Cada vez que el joystick registre un cambio de posición el AVR Butterfly enviará un byte a la EEPROM, dependiendo de la posición del Joystick según se ilustra en el diagrama de flujo de la Figura 3.8.

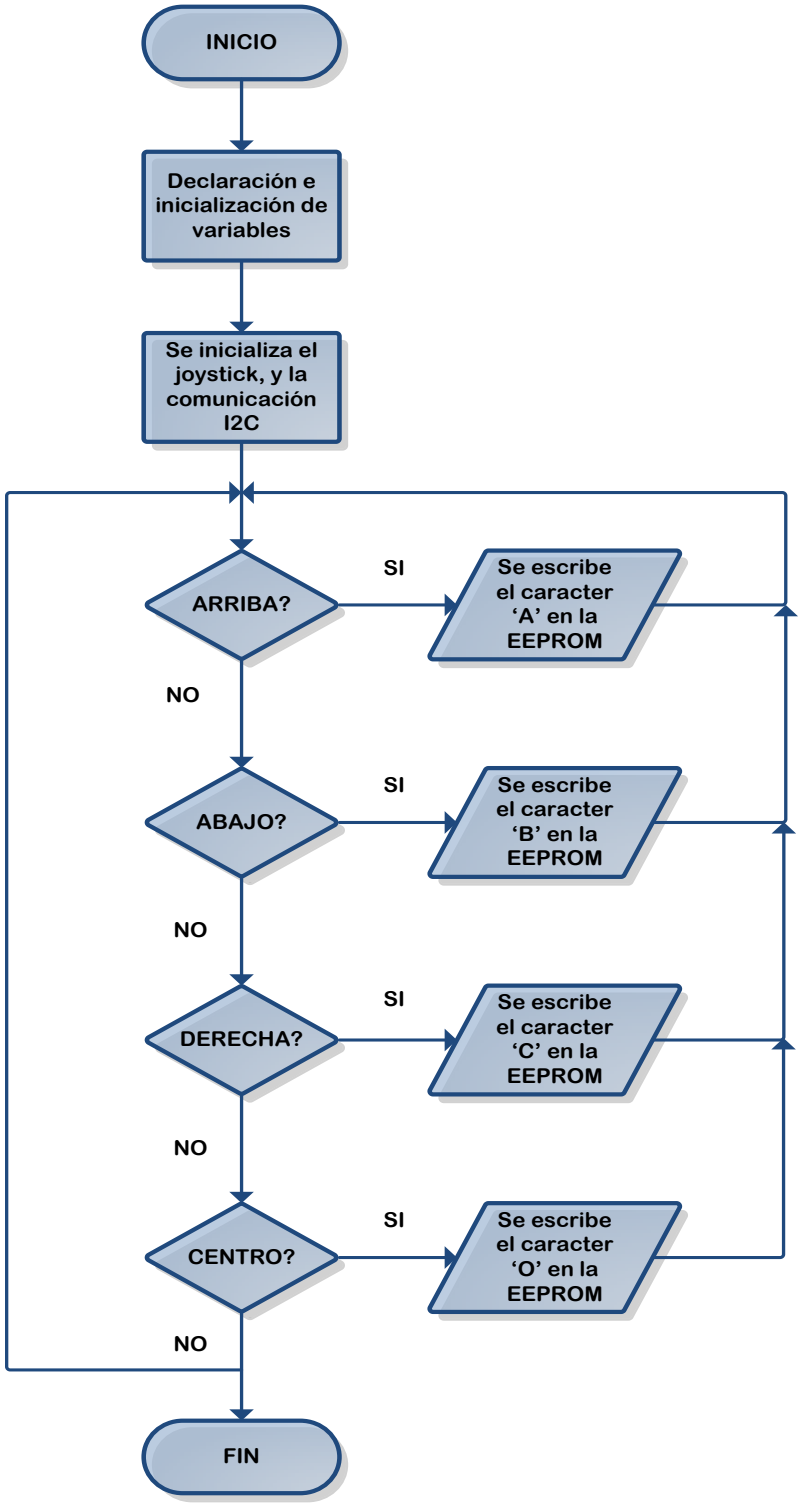

Figura 3.8 Diagrama de flujo del programa del AVR Butterfly

3.3.2. Implementación

La Figura 3.9 muestra la comunicación I2C entre el AVR Butterfly y una memoria EEPROM. Al presionar algun botón del joystick del AVR Butterfly se envía un dato correspondiente a lo que se haya presionado, si se presiona arriba se envía el codigo ASCII de la letra ‘A’, si se presiona abajo, ‘B’, si se presiona derecha, ‘C’, y si se presiona el centro se envía ‘O’. Para verificar que los resultados sean los esperados, se procede a pausar la simulación y verificar que en el contenido de memoria de la EEPROM a partir de la posición 0x0000 se encuentre el historial de todos los botones del joystick que se hayan presionado en dicha simulación.

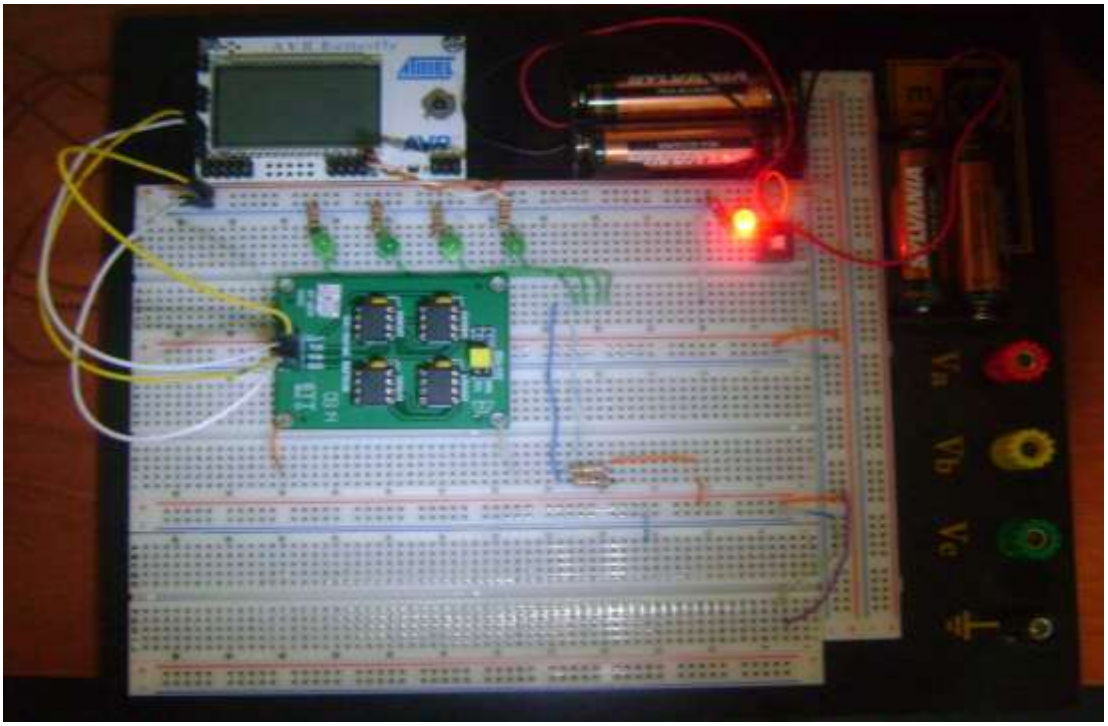

Figura 3.9 Implementacion fisica ejercicio 3

3.3.3. Conclusión de la implementación

Al finalizar este ejercio se ha podido comprender más acerca de la tarjeta de desarrollo AVR Butterfly y su módulo TWI, que es el usado para establecer la comunicación I2C con la memoria EEMPROM. La implementación de las funciones y librerías para el AVR studio es similar a las funciones del IDE de la LPCXpresso ya que usan un arreglo en el que se colocan los datos a escribir, y luego se procede a invocar a la función que realiza toda la gestión según el estandar I2C.

CAPÍTULO 4
IMPLEMENTACIÓN DEL PROYECTO FINAL

Este capítulo contiene la implementación física del proyecto final, incluyendo una breve descripción de los algoritmos que se usaron en la programación de las tarjetas AVR Butterfly y LPCXpresso 1769.

En el presente capítulo también se incluyen los diagramas de bloques y de flujo de los procedimientos que realizan la comunicación I2C entre las dos tarjetas, así como la conclusión de esta implementación.

4.1. Comunicación I2C entre la AVR Butterfly y la LPCXpresso 1769

Este proyecto consiste en controlar un motor BLDC mediante un joystick, por medio de la comunicación serial entre la tarjeta de desarrollo AVR Butterfly y la tarjeta LPCXpresso 1769, como se logra apreciar en el diagrama de bloques de la figura 4.1. La AVR Butterfly tiene incorporado un joystick y utiliza los pines 4, 6 y 7 del puerto B y los pines 2 y 3 del puerto E del microcontrolador ATMEGA 169 embebido en la tarjeta AVR Butterfly. La AVR Butterfly se encarga de enviar constantemente el estado del joystick a la LPCXpresso 1769 mediante la comunicación I2C, ya sea que el joystick esté

en una posición determinada o que simplemente se encuentre en reposo. Apenas la LPCXpresso 1769 reciba el byte correspondiente al estado del joystick, se lo coloca en el puerto 2, el cual está directamente conectado con la LPC 1114 y ésta a su vez con la tarjeta que controla el motor BLDC.

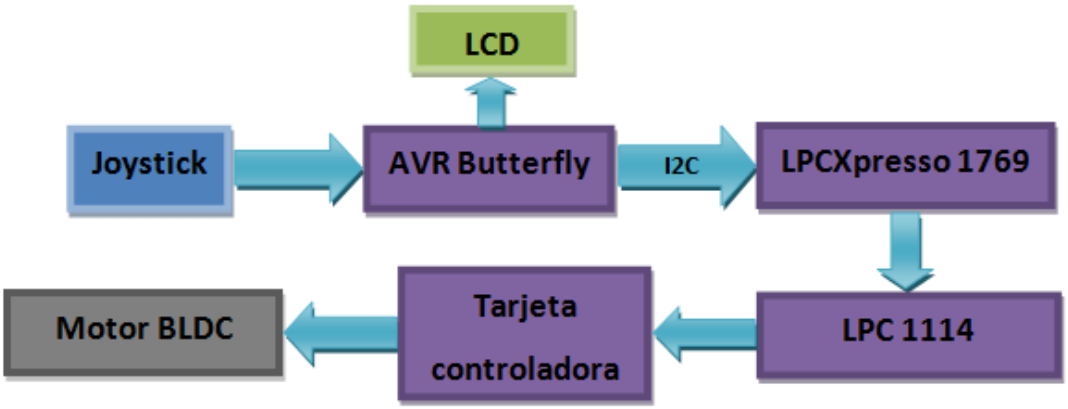

Figura 4.1 Diagrama de bloques proyecto final

4.1.1. Descripción del algoritmo de la AVR Butterfly

Primero se declaran e inicializan las variables que se van a usar durante la ejecución del programa, luego se inicializan el joystick, el módulo de comunicación I2C y el LCD, debido a que cada movimiento del joystick será mostrado por el LCD de la AVR Butterfly.

Luego de inicializados todos los módulos de la AVR Butterfly, se procede a iniciar la comunicación I2C mediante el envío de la

dirección del esclavo de la LPCXpresso 1769, junto con el bit de escritura 0. Una vez que el acuse de recibo que envía la LPCXpresso es recibida por la AVR Butterfly, el maestro AVR Butterfly está listo para enviar un byte dependiendo de la posición en la que se encuentre el joystick, y si éste se encuentra en estado de reposo se envía el valor predeterminado de 0x0F.

En el caso de que el joystick se encuentre presionado hacia arriba, la AVR Butterfly procede a enviar el byte 0x0E, y a mostrar en su pantalla LCD la palabra “UP”, en cambio, si el joystick se encuentra presionado hacia abajo, se envía el byte 0x0D, y se muestra en el LCD la palabra “DOWN”. Si se ha presionado el joystick hacia la derecha, la tarjeta AVR envía el byte 0x0B, y procede a mostrar en el LCD la palabra “REVERSE”, y si el joystick se encuentra presionado en el centro, se envía el byte 0x07, y se muestra en el LCD la palabra “START”.

Cabe recalcar que la función izquierda del joystick se encuentra deshabilitada, ya que solo eran necesarios cuatro de los cinco botones del joystick, si se llegara a presionar el joystick hacia la izquierda, el AVR Butterfly enviaría el valor predeterminado de 0x0F tal como se detalla en el diagrama de flujo de la Figura 4.2.

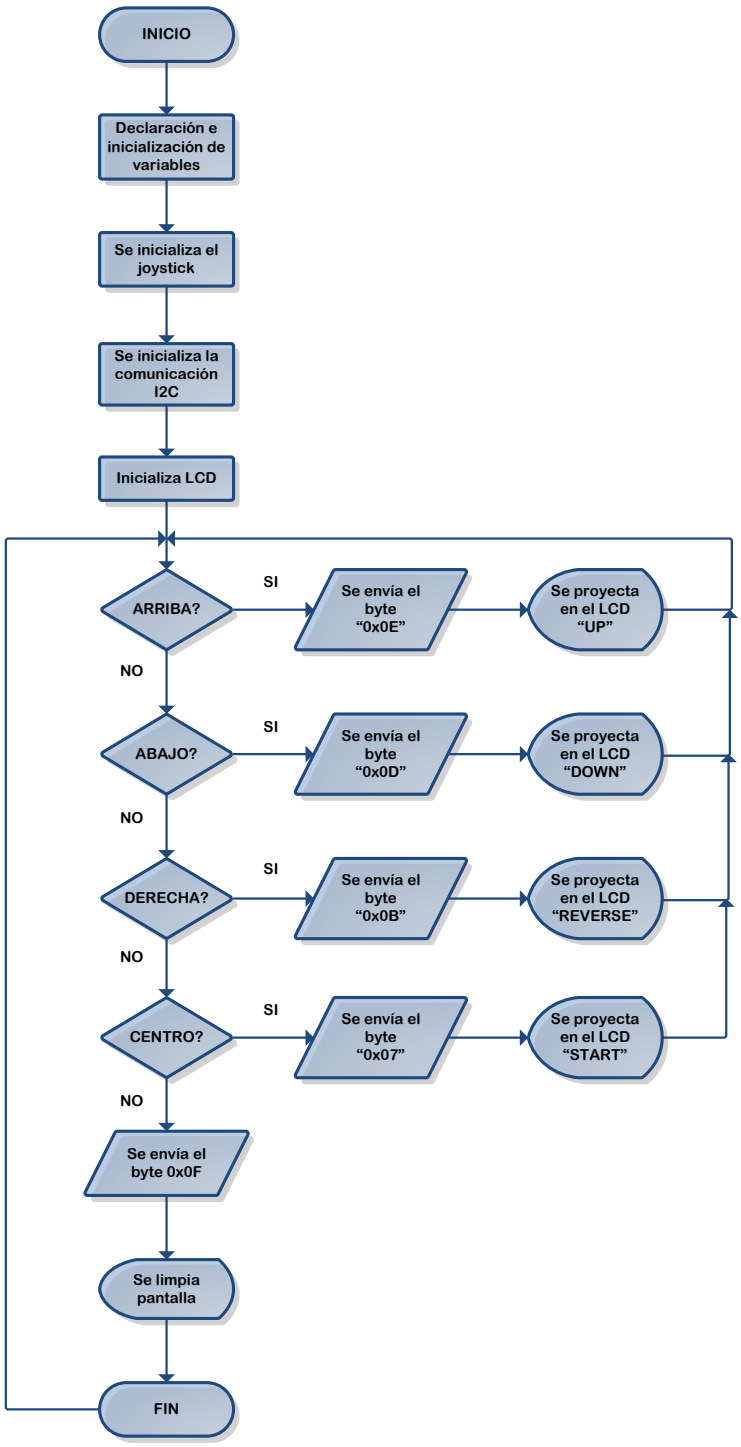

Figura 4.2 Diagrama de flujo del programa del AVR Butterfly

4.1.2. Descripción del algoritmo de la LPCXpresso 1769

En primer lugar se procede a declarar e inicializar las variables que serán utilizadas durante la ejecución del programa, luego se configura al puerto 2 como salida, debido a que éste será el encargado de proyectar los bytes que reciba de la tarjeta AVR Butterfly. Luego se procede a encerar el buffer de lectura y a inicializar el módulo de comunicación I2C y se lo configura como esclavo.

Una vez realizadas todas las inicializaciones, la LPCXpresso 1769, está lista para recibir la dirección de esclavo, para determinar si ésta coincide con la dirección del esclavo grabada en sus registros de direcciones. En el caso de coincidir, la LPCXpresso 1769 procede a enviar un acuse de recibo. Luego de esto cada vez que la LPCXpresso reciba un byte de datos, envía a la AVR Butterfly su respectiva confirmación ACK, y proyecta este byte en el puerto 2 como se puede apreciar en el diagrama de flujo de la Figura 4.3.

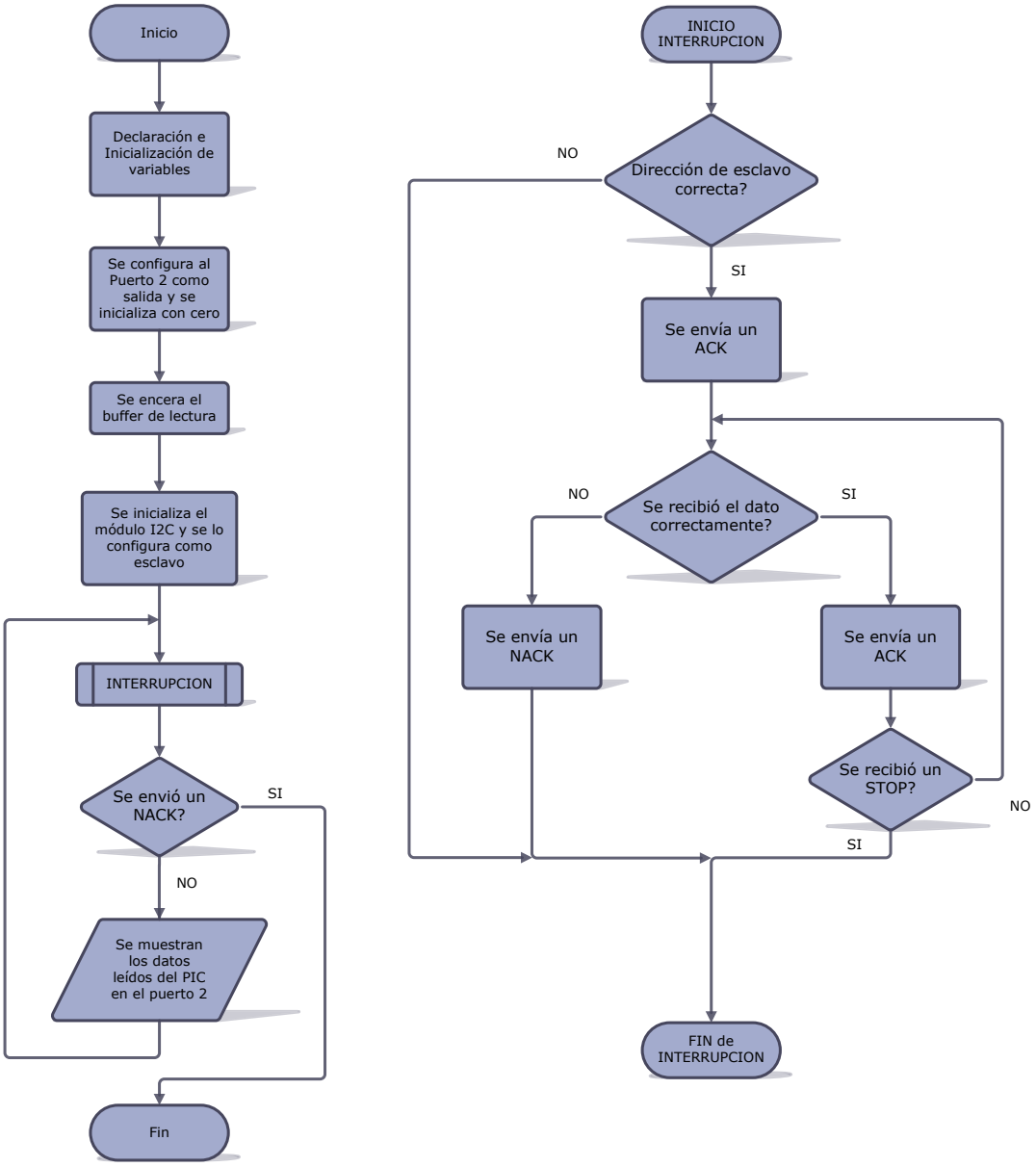

Figura 4.3 Diagrama de flujo del programa del LPCXpresso 1769

4.1.3. Implementación

En la Figura 4.4 se muestra el funcionamiento total de la comunicación I2C entre la AVR Butterfly como maestro y la LPCXpresso 1769 como esclavo controlando a otra tarjeta que es la LPC 1114 que a su vez es la encargada de enviar los comandos necesarios al driver (controlador) del motor BLDC.

El maestro es el encargado de dar la señal de inicio, seguido de la dirección del esclavo de la LPCXpresso 1769 que en este caso es 0xA0. Una vez que la LPC recibe esta dirección, envía un ACK y el AVR procede a enviar el dato, que es el byte correspondiente a la posición del joystick en ese momento.

Una vez que la LPCXpresso 1769 recibe el dato envía un acuse de recibo ACK y almacena el dato temporalmente en el buffer de lectura, e inmediatamente lo coloca en el puerto 2.

Los valores de los datos que se envían tienen un propósito, y es de simular unas botoneras en configuración pull-up. Como en la LPC 1114 solo se necesitan cuatro botoneras que realizan todas las funciones de mando, entonces, del puerto 2 de la LPCXpresso 1769 solo van a estar funcionando los 4 pines menos significativos, y por eso es que los cuatro bits más significativos de los datos son siempre bajos, mientras que tres de los cuatro bits menos

significativos son altos y uno de ellos es bajo, dependiendo de la posición del joystick, de tal manera que simule el hecho de que una de las cuatro botoneras se ha presionado, que representa el bit que está en bajo, mientras que las otras tres botoneras restantes que se encuentran sin presionar representan los tres altos.

En resúmen, cuando el joystick se encuentra en estado de reposo, se envía el valor predeterminado de “0x0F” lo que hace que los pines P2.0, P2.1, P2.2 y P2.3 permanezcan en alto, simulando los altos de las pull-up. Cuando el joystick se encuentra presionado en el centro se envía “0x07” lo que hace que el pin P2.3 sea un bajo, mientras que los demás pines permanezcan en alto. Cuando el joystick vuelva a su estado predeterminado el pin P2.3 vuelve a ser un alto, lo que simula una botonera que se ha presionado y soltado. Un bajo seguido de un alto en el pin P2.3 hace que el motor arranque, siempre y cuando no se encuentre actualmente funcionando, en ese caso, ocacionará que el motor se detenga.

Cuando el joystick se encuentra presionado hacia la derecha se envía “0x0B” y luego que el joystick deja de presionarse se envía el valor predeterminado “0x0F” lo que hace que el pin P2.2 se ponga en bajo y luego en alto, y todos los demas pines permanezcan en alto. Esto hace que el motor invierta su giro.

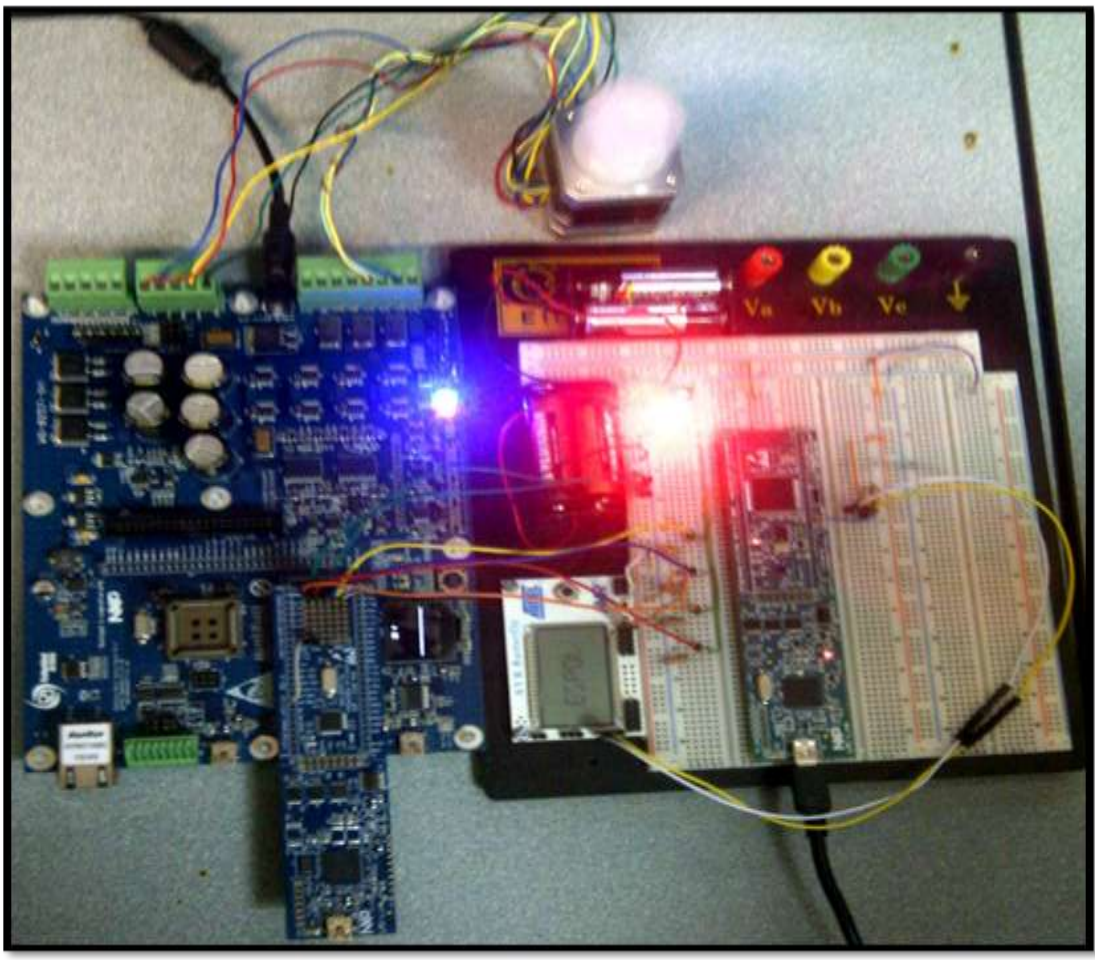

Figura 4.4 Implementacion fisica proyecto final

4.1.4. Descripción de los resultados

Si el joystick se encuentra presionado hacia la izquierda, esta acción es ignorada por el AVR Butterfly y se envía el valor predeterminado “0x0F”.

En cambio, si el joystick se encuentra presionado hacia abajo se envía “0x0D” y luego que el joystick deja de presionarse se envía

el valor predeterminado “0x0F” lo que hace que el pin P2.1 se ponga en bajo y luego en alto, y todos los demas pines permanezcan en alto. Cada vez que se realiza esta acción el motor pierde 50 rpm en su velocidad, hasta llegar a la velocidad mínima de 600 rpm. Finalmente si el joystick se encuentra presionado hacia arriba se envía “0x0E” y luego que el joystick deja de presionarse se envía el valor predeterminado “0x0F” lo que hace que el pin P2.0 se ponga en bajo y luego en alto, y todos los demas pines permanezcan en alto. Cada vez que esto suceda, el motor aumenta 50 rpm de velocidad, hasta llegar a la velocidad de 4100 rpm, una vez alcanzada esta velocidad, ya no podrá aumentar más.

4.1.5. Conclusión de implementación

Al finalizar la implementación de este proyecto, se ha podido fortalecer los conocimientos acerca del protocolo I2C y su implementación en las diferentes tarjetas de desarrollo, como son la AVR Butterfly que trabaja como maestro, como la LPCXpresso que funciona como esclavo. Se ha podido enviar satisfactoriamente comandos de control para el manejo del motor BLDC.

CONCLUSIONES

1. Se han presentado diversos ejemplos en donde se establece el protocolo de comunicación serial I2C. Por lo tanto concluimos que este protocolo es eficaz ya que requiere solo de dos señales, SDA y SCL para transmitir o recibir datos a velocidades relativamente altas (100Kbps como tasa de transmisión estándar).

2. Se ha experimentado con las tarjetas de desarrollo LPCXpresso y la AVR Butterfly. Estas tarjetas presentan grandes ventajas frente al uso de microcontroladores por si solos, ya que en sus tarjetas integran circuitos de protección, módulos, dispositivos terminales, programadores y debuggers. Esto permite ahorrar el uso de circuitería externa para el diseño de prototipos.

3. Se ha logrado establecer el control del motor BLDC, el cual posee mayor eficiencia que un motor DC con escobillas, ya que conmuta electrónicamente y no mecánicamente. Además, su ruido eléctrico es muy bajo y no tiene caída de tensión puesto que no posee escobillas. Estas ventajas han permitido que su uso se esté incrementando para diversas aplicaciones en la actualidad.

RECOMENDACIONES

1. Cuando se programe la LPC es recomendable evitar la manipulación de los circuitos con los dedos, debido a que se puedan ocasionar descargas electroestáticas que puedan dañar la tarjeta. En el caso de que el programa no pueda ser ejecutado mediante el debugger de la tarjeta LPCXpresso, verificar si el cable USB está bien conectado de lo contrario cambie de cable.

2. La programación del AVR Butterfly se la debe hacer con la tarjeta energizada y presionando un botón del joystick, teniendo en cuenta que si tiene puertos virtuales instalados en su computadora deberá desinstalarlos ya que el programa elige los puertos virtuales del uno al cuatro.

3. En el bus I2C es importante colocar una resistencia de pull-up a cada línea, tanto a la SDA como a la SCL, los rangos de valores de resistencias aceptables es muy amplio, desde 1.8 KΩ hasta 47 KΩ, sin embargo escoger valores bajos de resistencia incrementa el consumo de los integrados pero disminuye la sensibilidad al ruido y mejora el tiempo de los flancos de subida y bajada de las señales. Un valor recomendado es de 4.7 KΩ.

ANEXO A

CÓDIGOS FUENTE PARA CADA EJERCICIO Y PROYECTO FINAL

Ejercicio 1: Código fuente del programa del LPCXpresso 1769 para comunicación I2C con la EEPROM 24LC32A

```
 /**************** MICROCONTROLADORES AVANZADOS ******************
 *************************** Ejercicio # 1************************************/
#include <cr_section_macros.h>
#include <NXP/crp.h>


__CRP const unsigned int CRP_WORD = CRP_NO_CRP ;

#include "lpc17xx.h"
#include "type.h"
#include "i2c.h"

extern volatile uint8_t I2CMaestroBuffer[I2C_PORT_NUM][BUFSIZE]; //Buffer de escritura
extern volatile uint8_t I2CSlaveBuffer[I2C_PORT_NUM][BUFSIZE]; //Buffer de lectura
extern volatile uint32_t I2CReadLength[I2C_PORT_NUM]; //Caracteres que se van a leer
extern volatile uint32_t I2CWriteLength[I2C_PORT_NUM]; //Caracteres que se van a escribir

#define PORT_USED              2

//*************************  Main Function  main() ********

int main (void)
{
  uint32_t  i=0; // variable usada como iterador
  uint32_t  j=0; // variable usada como iterador
  SystemClockUpdate();
  LPC_GPIO2->FIODIR = 0xFFFFFFFF;          /* P2.xx se las define como salidas */
  LPC_GPIO2->FIOCLR = 0xFFFFFFFF;          /* Se inicializa con bajo al Puerto 2 */
  I2C2Init( );                             /* inicializa I2c2 */

  /* Escribe SLA(W), dirección de memoria y 8 datos */

  I2CWriteLength[PORT_USED] = 11;
  I2CReadLength[PORT_USED] = 0;
  I2CMaestroBuffer[PORT_USED][0] = PCF8594_ADDR;
```

```
I2CMaestroBuffer[PORT_USED][1] = 0xFF;  /* 2do byte de dirección LSB */
I2CMaestroBuffer[PORT_USED][2] = 0x06;  /* 1er byte de dirección MSB */
I2CMaestroBuffer[PORT_USED][3] = 0x01;  //Aquí comienzan los datos
I2CMaestroBuffer[PORT_USED][4] = 0x02;  //Se envían datos de tal forma
I2CMaestroBuffer[PORT_USED][5] = 0x04;  //que al recibirlos de vuelta
I2CMaestroBuffer[PORT_USED][6] = 0x08;  //y mostrarlos en el Puerto 2
I2CMaestroBuffer[PORT_USED][7] = 0x10;  //los LEDS roten un alto
I2CMaestroBuffer[PORT_USED][8] = 0x20;
I2CMaestroBuffer[PORT_USED][9] = 0x40;
I2CMaestroBuffer[PORT_USED][10] = 0x80;
I2CEngine( PORT_USED );

for ( i = 0; i < 0x200000; i++ );           /* Delay después de escribir */
for ( i = 0; i < BUFSIZE; i++ )
        I2CSlaveBuffer[PORT_USED][i] = 0x00;  //Se encera el buffer de lectura

/* Se escribe SLA(W), dirección, SLA(R), y se efectúa la lectura de los 8 bytes */

I2CWriteLength[PORT_USED] = 3;
I2CReadLength[PORT_USED] = 8;
I2CMaestroBuffer[PORT_USED][0] = PCF8594_ADDR;
I2CMaestroBuffer[PORT_USED][1] = 0xFF;              /* address */
I2CMaestroBuffer[PORT_USED][2] = 0x06;              /* address */
I2CMaestroBuffer[PORT_USED][3] = PCF8594_ADDR | RD_BIT;
I2CEngine( PORT_USED );

/* Se verifica el contenido del buffer de lectura */

while ( 1 ){

        for ( i =0; i < 8; i++ )
          {
                LPC_GPIO2->FIOPIN = I2CSlaveBuffer[PORT_USED][i];
                for(j = 6000000; j > 0; j--);
          }
 }
}
//*********************************FIN*************************************
```

Ejercicio 2: Código fuente del LPCXpresso 1769 como esclavo para Comunicación I2C entre PIC16f887

```
/***************** MICROCONTROLADORES AVANZADOS ********************
*************************** Ejercicio # 2************************************/


#include <cr_section_macros.h>
#include <NXP/crp.h>

__CRP const unsigned int CRP_WORD = CRP_NO_CRP ;

#include "LPC17xx.h"                        /* LPC11xx Peripheral Registers */
#include "type.h"
#include "i2cslave.h"

volatile uint32_t I2CMaestroState = I2C_IDLE; //Estado inicial del bus I2C
volatile uint32_t I2CSlaveState = I2C_IDLE; //Estado inicial del bus I2C

volatile uint32_t I2CMode;

volatile uint8_t I2CWrBuffer[BUFSIZE]; //Buffer de escritura
volatile uint8_t I2CRdBuffer[BUFSIZE]; //Buffer de lectura
volatile uint32_t I2CReadLength; //Longitud de lectura
volatile uint32_t I2CWriteLength; //Longitud de escritura

volatile uint32_t RdIndex = 0; //Índice del buffer de lectura
volatile uint32_t WrIndex = 0; //Índice del buffer de escritura

//*************************   Main Function  main()*********************//

int main (void)

{

  uint32_t i;
  uint32_t j=0;

  /* SystemClockUpdate() updates the SystemFrequency variable */
```

```
  SystemClockUpdate();
  LPC_GPIO2->FIODIR = 0xFFFFFFFF;          /* P2.xx se las define como salidas */
  LPC_GPIO2->FIOCLR = 0xFFFFFFFF;          /* Se inicializa con bajo al Puerto 2 */

  for ( i = 0; i < BUFSIZE; i++ )
  {
          I2CRdBuffer[i] = 0x00;  //Se encera el buffer de lectura
  }

  I2CSlave2Init();                                /* inicializa I2c */

/*Muestra en el Puerto 2 lo que acaba de leer en el buffer de lectura*/

  while ( I2CSlaveState != DATA_NACK ){
            for ( i = 0; i < 8; i++ )
            {
                      LPC_GPIO2->FIOPIN = I2CRdBuffer[i];
                      for(j = 6000000; j > 0; j--);
            }
  }
  return ( 0 );
}

//**********************************END**********************//
```

Código fuente del PIC 16F887 como maestro

```
 /**************** MICROCONTROLADORES AVANZADOS ******************
 *************************** Ejercicio # 2************************************/

unsigned short mask(unsigned short num) { //Función que simula la rotación de LEDS
        switch (num) {
                case 0 : return 0x00;
                case 1 : return 0x01;
                case 2 : return 0x02;
                case 3 : return 0x04;
                case 4 : return 0x08;
                case 5 : return 0x10;
                case 6 : return 0x20;
                case 7 : return 0x40;
                case 8 : return 0x80;
                default : return;
        }
}
unsigned short i=0;
void main() {
  ANSEL  = 0;            // Configura los pines AN como digitales
  ANSELH = 0;
  PORTB = 0;
  TRISB = 0;             // Configura PORTB  como salida

  I2C1_Init(100000);      // inicializa la comunicación I2C
  I2C1_Start();          // Se envía la señal de start
  I2C1_Wr(0xA0);    // Envía por medio del bus I2C un byte (dirección del esclavo + W)
       for(i=1 ; i<9 ; i++){
         I2C1_Wr(mask(i));         // Envía un dato
         delay_ms(100);           //retardo de 100 ms
       }
  I2C1_Stop();           // Se envía la señal de stop
  while(1){
       PORTB = 0xFF;     //Se encienden leds para indicar que la comunicación finalizó
  }
}
```

Ejercicio 3: Comunicación I2C entre AVR Butterfly y EEPROM usando joystick

```
/***************** MICROCONTROLADORES AVANZADOS ********************
**************************** Ejercicio # 3************************************/
  /***************** Código fuente de la tarjeta AVR Butterfly ********************

#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>
#include "USI_TWI_Maestro.h"
#define MASCARA_PINB ((1<<PINB7)|(1<<PINB6)|(1<<PINB4))
#define MASCARA_PINE ((1<<PINE3)|(1<<PINE2))
#define posicion_A 6 //ARRIBA
#define posicion_B 7 //ABAJO
#define posicion_C 2 //DERECHA
#define posicion_D 3 //IZQUIERDA
#define posicion_O 4 //CENTRO
unsigned char i=0,j=0;
unsigned char msg [4]={0xA0,0x00,0x00,0x00};
unsigned char msgSize = 4;
void inicializar(void);

//***************************** FUNCIONES ***************************

void inicializar(void)

{       CLKPR = (1<<CLKPCE);
        CLKPR = (1<<CLKPS3);
        while(CLKPR & (1<<CLKPCE));

        {       DDRB |= 0xD0;
                PORTB |= MASCARA_PINB;
                DDRE |= 0x0C;
                PORTE |= MASCARA_PINE;
                DDRB = 0;//entrada
                PORTB = MASCARA_PINB;//habilitar PULL-UPs
                DDRE = 0;//entrada
                PORTE = MASCARA_PINE;//habilitar PULL-UPs
                PCMSK1 |= MASCARA_PINB;
                PCMSK0 |= MASCARA_PINE;
                EIFR = ((1<<PCIF1)|(1<<PCIF0));
```

```
                EIMSK = ((1<<PCIE1)|(1<<PCIE0));
                DDRD = 0xFF;
                PORTD = 0x01;
        }
}

//********************** MAIN ************************

int main(void)

{
DDRB = 0x00;
DDRD = 0xFF;
unsigned char joystick;
inicializar(); // Inicializa el joystick
USI_TWI_Maestro_Initialise(); // Inicializa comunicacion I2C
_delay_us(100); // retardo para estabilizar

for(;;)

{
        joystick = ((~PINB) & MASCARA_PINB);
        joystick |= ((~PINE) & MASCARA_PINE);

        if((joystick & (1<<posicion_A))){ //ARRIBA
                        msg [0]=0xA0;
                        msg [1]=i;
                        msg [2]=j;
                        msg [3]='A';
                        USI_TWI_Start_Transceiver_With_Data(&msg, msgSize);
                        j++;
                        _delay_us(110);
        }
        else if((joystick & (1<<posicion_B))){ //ABAJO

                        PORTD = 0x02;
                        msg [0]=0xA0;
                        msg [1]=i;
                        msg [2]=j;
                        msg [3]='B';
                        USI_TWI_Start_Transceiver_With_Data(&msg, msgSize);
                        j++;
```

```
                                _delay_us(110);
    }

else if((joystick & (1<<posicion_C))){ // DERECHA
PORTD = 0x04;
msg [0]=0xA0;
msg [1]=i;
msg [2]=j;
msg [3]='C';
USI_TWI_Start_Transceiver_With_Data(&msg, msgSize);
j++;
_delay_us(110);

}
else if((joystick & (1<<posicion_O))){ //CENTRO
PORTD = 0x10;
msg [0]=0xA0;
msg [1]=i;
msg [2]=j;
msg [3]='O';
USI_TWI_Start_Transceiver_With_Data(&msg, msgSize);
j++;
_delay_us(110);

}

}

return 0;

}
```

Librería USI_TWI_Maestro.h

```
#include <avr/interrupt.h>
#define F_CPU 8000000UL            // Sets up the default speed for delay.h
#include <util/delay.h>
#include <avr/io.h>
#include "USI_TWI_Maestro.h"

unsigned char USI_TWI_Start_Transceiver_With_Data(unsigned char * , unsigned char);
unsigned char USI_TWI_Maestro_Transfer( unsigned char );
unsigned char USI_TWI_Maestro_Stop( void );
unsigned char USI_TWI_Maestro_Start( void );

union  USI_TWI_state
{
  unsigned char errorState;         // Can reuse the TWI_state for error states since it will not be
needed if there is an error.
  struct
  {
    unsigned char addressMode        : 1;
    unsigned char maestroWriteDataMode : 1;
    unsigned char memReadMode          : 1;
    unsigned char unused               : 5;
  };
}   USI_TWI_state;
void USI_TWI_Maestro_Initialise( void )
{
  PORT_USI |= (1<<PIN_USI_SDA);           // Enable pullup on SDA, to set high as released
state.
  PORT_USI |= (1<<PIN_USI_SCL);           // Enable pullup on SCL, to set high as released
state.

  DDR_USI  |= (1<<PIN_USI_SCL);          // Enable SCL as output.
  DDR_USI  |= (1<<PIN_USI_SDA);          // Enable SDA as output.

  USIDR    =  0xFF;                      // Preload dataregister with "released level" data.
  USICR    =  (0<<USISIE)|(0<<USIOIE)|                        // Disable Interrupts.
              (1<<USIWM1)|(0<<USIWM0)|                        // Set USI in Two-wire mode.
              (1<<USICS1)|(0<<USICS0)|(1<<USICLK)|      // Software stobe as counter clock
source
              (0<<USITC);
```

```
 USISR   =   (1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)|      // Clear flags,
          (0x0<<USICNT0);                                 // and reset counter.
}

unsigned char USI_TWI_Get_State_Info( void )
{
 return ( USI_TWI_state.errorState );                      // Return error state.
}
unsigned char USI_TWI_Start_Random_Read( unsigned char *msg, unsigned char msgSize)
{
 *(msg) &= ~(TRUE<<TWI_READ_BIT);                  // clear the read bit if it's set
 USI_TWI_state.errorState = 0;
 USI_TWI_state.memReadMode = TRUE;

 return (USI_TWI_Start_Transceiver_With_Data( msg, msgSize));
}
unsigned char USI_TWI_Start_Read_Write( unsigned char *msg, unsigned char msgSize)
{

          USI_TWI_state.errorState = 0;             // Clears all mode bits also

          return (USI_TWI_Start_Transceiver_With_Data( msg, msgSize));

}
unsigned char USI_TWI_Start_Transceiver_With_Data( unsigned char *msg, unsigned char
msgSize)
{
 unsigned char const tempUSISR_8bit =
(1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)|
                        (0x0<<USICNT0);  // set USI to shift 8 bits i.e. count 16 clock edges.
 unsigned                char                const                tempUSISR_1bit                =
(1<<USISIF)|(1<<USIOIF)|(1<<USIPF)|(1<<USIDC)|(0xE<<USICNT0);
          unsigned char *savedMsg;
          unsigned char savedMsgSize;
 USI_TWI_state.addressMode = TRUE;    // Always true for first byte

#ifdef PARAM_VERIFICATION
 if(msg > (unsigned char*)RAMEND)              // Test if address is outside SRAM space
 {
  USI_TWI_state.errorState = USI_TWI_DATA_OUT_OF_BOUND;
  return (FALSE);
 }
```

```
  if(msgSize <= 1)                    // Test if the transmission buffer is empty
  {
    USI_TWI_state.errorState = USI_TWI_NO_DATA;
    return (FALSE);
  }
#endif

#ifdef NOISE_TESTING                  // Test if any unexpected conditions have
arrived prior to this execution.
  if( USISR & (1<<USISIF) )
  {
    USI_TWI_state.errorState = USI_TWI_UE_START_CON;
    return (FALSE);
  }
  if( USISR & (1<<USIPF) )
  {
    USI_TWI_state.errorState = USI_TWI_UE_STOP_CON;
    return (FALSE);
  }
  if( USISR & (1<<USIDC) )
  {
    USI_TWI_state.errorState = USI_TWI_UE_DATA_COL;
    return (FALSE);
}
#endif

  if ( !(*msg & (1<<TWI_READ_BIT)) )  // The LSB in the address byte determines
if is a maestroRead or maestroWrite operation.
  {
          if ( !USI_TWI_Maestro_Start( ))
  {
          return (FALSE);             // Send a START condition on the TWI bus.
  }
  do
  {
    if (USI_TWI_state.addressMode || USI_TWI_state.maestroWriteDataMode)
    {
      /* Write a byte */
      PORT_USI &= ~(1<<PIN_USI_SCL);        // Pull SCL LOW.
      USIDR     = *(msg++);               // Setup data.
      USI_TWI_Maestro_Transfer( tempUSISR_8bit );    // Send 8 bits on bus.
```

```
    /* Clock and verify (N)ACK from slave */
    DDR_USI  &= ~(1<<PIN_USI_SDA);              // Enable SDA as input.
    if( USI_TWI_Maestro_Transfer( tempUSISR_1bit ) & (1<<TWI_NACK_BIT) )
    {
     if ( USI_TWI_state.addressMode )
       USI_TWI_state.errorState = USI_TWI_NO_ACK_ON_ADDRESS;
     else
       USI_TWI_state.errorState = USI_TWI_NO_ACK_ON_DATA;
     return (FALSE);
    }

          if ((!USI_TWI_state.addressMode) && USI_TWI_state.memReadMode)// means
memory start address has been written
          {
                msg = savedMsg;                                    // start at slave
address again
                *(msg) |= (TRUE<<TWI_READ_BIT);  // set the Read Bit on Slave
USI_TWI_state.errorState = 0;
                USI_TWI_state.addressMode = TRUE; // Now set up for the Read cycle
                msgSize = savedMsgSize;                            // Set byte count
correctly
                if ( !USI_TWI_Maestro_Start( ))
                {
                //        USI_TWI_state.errorState = USI_TWI_BAD_MEM_READ;
                        return (FALSE);                  // Send a START condition on the
TWI bus.
                }
          }
          else
          {
                USI_TWI_state.addressMode = FALSE;          // Only perform address
transmission once.
          }
   }
   else
   {
    /* Read a data byte */
    DDR_USI   &= ~(1<<PIN_USI_SDA);             // Enable SDA as input.
    *(msg++)  = USI_TWI_Maestro_Transfer( tempUSISR_8bit );

    /* Prepare to generate ACK (or NACK in case of End Of Transmission) */
    if( msgSize == 1)                   // If transmission of last byte was performed.
```

```
    {
      USIDR = 0xFF;                         // Load NACK to confirm End Of Transmission.
    }
    else
    {
      USIDR = 0x00;                         // Load ACK. Set data register bit 7 (output for SDA)
low.
    }
    USI_TWI_Maestro_Transfer( tempUSISR_1bit );   // Generate ACK/NACK.
   }
 }while( --msgSize) ;                       // Until all data sent/received.

 if (!USI_TWI_Maestro_Stop())
 {
          return (FALSE);                    // Send a STOP condition on the TWI bus.
          }


unsigned char USI_TWI_Maestro_Transfer( unsigned char temp )
{
 USISR = temp;                              // Set USISR according to temp.
                                    // Prepare clocking.
 temp  =  (0<<USISIE)|(0<<USIOIE)|          // Interrupts disabled
        (1<<USIWM1)|(0<<USIWM0)|            // Set USI in Two-wire mode.
        (1<<USICS1)|(0<<USICS0)|(1<<USICLK)|    // Software clock strobe as source.
        (1<<USITC);                         // Toggle Clock Port.
 do
 {
          _delay_us(T2_TWI);
  USICR = temp;                     // Generate positve SCL edge.
  while( !(PIN_USI & (1<<PIN_USI_SCL)) );// Wait for SCL to go high.
          _delay_us(T4_TWI);
  USICR = temp;                     // Generate negative SCL edge.
 }while( !(USISR & (1<<USIOIF)) );       // Check for transfer complete.

          _delay_us(T2_TWI);
 temp  = USIDR;                     // Read out data.
 USIDR = 0xFF;                      // Release SDA.
 DDR_USI |= (1<<PIN_USI_SDA);           // Enable SDA as output.

 return temp;                       // Return the data from the USIDR
}
```

```
unsigned char USI_TWI_Maestro_Start( void )
{
/* Release SCL to ensure that (repeated) Start can be performed */
  PORT_USI |= (1<<PIN_USI_SCL);                   // Release SCL.
 while( !(PORT_USI & (1<<PIN_USI_SCL)) );  // Verify that SCL becomes high
  _delay_us(T2_TWI);

/* Generate Start Condition */
  PORT_USI &= ~(1<<PIN_USI_SDA);                  // Force SDA LOW.
          _delay_us(T4_TWI);
  PORT_USI &= ~(1<<PIN_USI_SCL);                  // Pull SCL LOW.
  PORT_USI |= (1<<PIN_USI_SDA);                   // Release SDA.

#ifdef SIGNAL_VERIFY
  if( !(USISR & (1<<USISIF)) )


  {
    USI_TWI_state.errorState = USI_TWI_MISSING_START_CON;
    return (FALSE);
  }
#endif
  return (TRUE);
}
unsigned char USI_TWI_Maestro_Stop( void )
{
  PORT_USI &= ~(1<<PIN_USI_SDA);          // Pull SDA low.
  PORT_USI |= (1<<PIN_USI_SCL);           // Release SCL.
  while( !(PIN_USI & (1<<PIN_USI_SCL)) );  // Wait for SCL to go high.
          _delay_us(T4_TWI);
  PORT_USI |= (1<<PIN_USI_SDA);           // Release SDA.
          _delay_us(T2_TWI);

#ifdef SIGNAL_VERIFY
  if( !(USISR & (1<<USIPF)) )
  {
    USI_TWI_state.errorState = USI_TWI_MISSING_STOP_CON;
    return (FALSE);
  }
#endif
  return (TRUE);
}
```

Proyecto Final: Comunicación I2C entre la AVR Butterfly y la LPCXpresso 1769

```
/**************** MICROCONTROLADORES AVANZADOS ******************
************************* Proyecto Final*******************************/
/**************** Código fuente de la tarjeta AVR Butterfly ******************


#include <avr/io.h>
#define F_CPU 8000000UL
#include <util/delay.h>
#include "USI_TWI_Maestro.h"
#include <avr/pgmspace.h>
#include <inttypes.h>
#include "LCD_functions.h"
#include "LCD_driver.h"
#include "mydefs.h"
#include "button.h"

#define MASCARA_PINB ((1<<PINB7)|(1<<PINB6)|(1<<PINB4))
#define MASCARA_PINE ((1<<PINE3)|(1<<PINE2))
#define posicion_A 6 //ARRIBA
#define posicion_B 7 //ABAJO
#define posicion_C 2 //DERECHA
#define posicion_D 3 //IZQUIERDA
#define posicion_O 4 //CENTRO
unsigned char i=0,j=0;
unsigned char msg [2]={0xA0,0x00};
unsigned char msgSize = 2;
void inicializar(void);

//**********FUNCIONES **********************************

void inicializar(void)
{
        CLKPR = (1<<CLKPCE);
        CLKPR = (1<<CLKPS3);
        while(CLKPR & (1<<CLKPCE));
        {

                DDRB = 0;//entrada
```

```
                PORTB = MASCARA_PINB;//habilitar PULL-UPs
                DDRE = 0;//entrada
                PORTE = MASCARA_PINE;//habilitar PULL-UPs
                PCMSK1 |= MASCARA_PINB;
                PCMSK0 |= MASCARA_PINE;
                EIFR = ((1<<PCIF1)|(1<<PCIF0));
                EIMSK = ((1<<PCIE1)|(1<<PCIE0));

        }
}
//************************* MAIN**********************************

int main(void)
{
LCD_Init();
PGM_P statetext;

  uint8_t input;

unsigned char joystick;

inicializar(); // Inicializa el joystick
USI_TWI_Maestro_Initialise(); // Inicializa comunicacion I2C
_delay_us(100); // retardo para estabilizar


   while (1)
   {
      if (statetext){

       LCD_puts_f(statetext, 1);
       LCD_Colon(0);
       statetext = NULL;
                    }


     input = getkey();
          joystick = ((~PINB) & MASCARA_PINB);
          joystick |= ((~PINE) & MASCARA_PINE);
          msg [0]=0xA0;

          if((joystick & (1<<posicion_A))){ //ARRIBA
```

```
                        statetext = PSTR("UP");
                        msg [1]=0x0E;
        }
        else if((joystick & (1<<posicion_B))){ //ABAJO
                        statetext = PSTR("DOWN");
                        msg [1]=0x0D;
        }



else if((joystick & (1<<posicion_C))){ // Izquierda
                        statetext = PSTR("INVERT");
                        msg [1]=0x0B;
   }

        else if((joystick & (1<<posicion_O))){ //CENTRO
                        statetext = PSTR("START");
                        msg [1]=0x07;
        }

        else {
                        //statetext = PSTR("TOAZA-CARO");
                        statetext = PSTR("ESPOL");
                        msg [1]=0x0F;
                        }
        USI_TWI_Start_Transceiver_With_Data(&msg, msgSize);
        _delay_us(100);

   }

  return 0;



}
```

Código fuente del programa del LPCXpresso 1769

```
/**************** MICROCONTROLADORES AVANZADOS ******************
************************ Proyecto Final*******************************/
/**************** Código fuente de la tarjeta LPCXpresso 1769 ******************


#include <cr_section_macros.h>
#include <NXP/crp.h>

__CRP const unsigned int CRP_WORD = CRP_NO_CRP ;

#include "LPC17xx.h"                        /* LPC11xx Peripheral Registers */
#include "type.h"
#include "i2cslave.h"

volatile uint32_t I2CMaestroState = I2C_IDLE; //Estado inicial del bus I2C
volatile uint32_t I2CSlaveState = I2C_IDLE; //Estado inicial del bus I2C

volatile uint32_t I2CMode;

volatile uint8_t I2CWrBuffer[BUFSIZE]; //Buffer de escritura
volatile uint8_t I2CRdBuffer[BUFSIZE]; //Buffer de lectura
volatile uint32_t I2CReadLength; //Longitud de lectura
volatile uint32_t I2CWriteLength; //Longitud de escritura

volatile uint32_t RdIndex = 0; //Indice del buffer de lectura
volatile uint32_t WrIndex = 0; //Indice del buffer de escritura


//*****************************   Main Function  main()************************//
int main (void)
{
  uint32_t i;
  uint32_t j=0;

  /* SystemClockUpdate() updates the SystemFrequency variable */
  SystemClockUpdate();
  LPC_GPIO2->FIODIR = 0xFFFFFFFF;* P2.xx se las define como salidas */
  LPC_GPIO2->FIOCLR = 0xFFFFFFFF;/* Se inicializa con bajo al Puerto 2 */

  for ( i = 0; i < BUFSIZE; i++ )
```

```
  {
        I2CRdBuffer[i] = 0x00;  //Se encera el buffer de lectura
  }

  I2CSlave2Init();                          /* inicializa I2c */

/*Muestra en el Puerto 2 lo que acaba de leer en el buffer de lectura*/
  while ( I2CSlaveState != DATA_NACK ){

                LPC_GPIO2->FIOPIN = I2CRdBuffer[0];
                for(j = 6000000; j > 0; j--);

  }
  return ( 0 );
}

//*************************************END**********************//
```

ANEXO B

DIAGRAMAS ESQUEMATICOS DE EJERCICIOS Y PROEYECTO FINAL

Esquema Comunicación I2C de la LPC 1769 con EEPROM 24LC32A

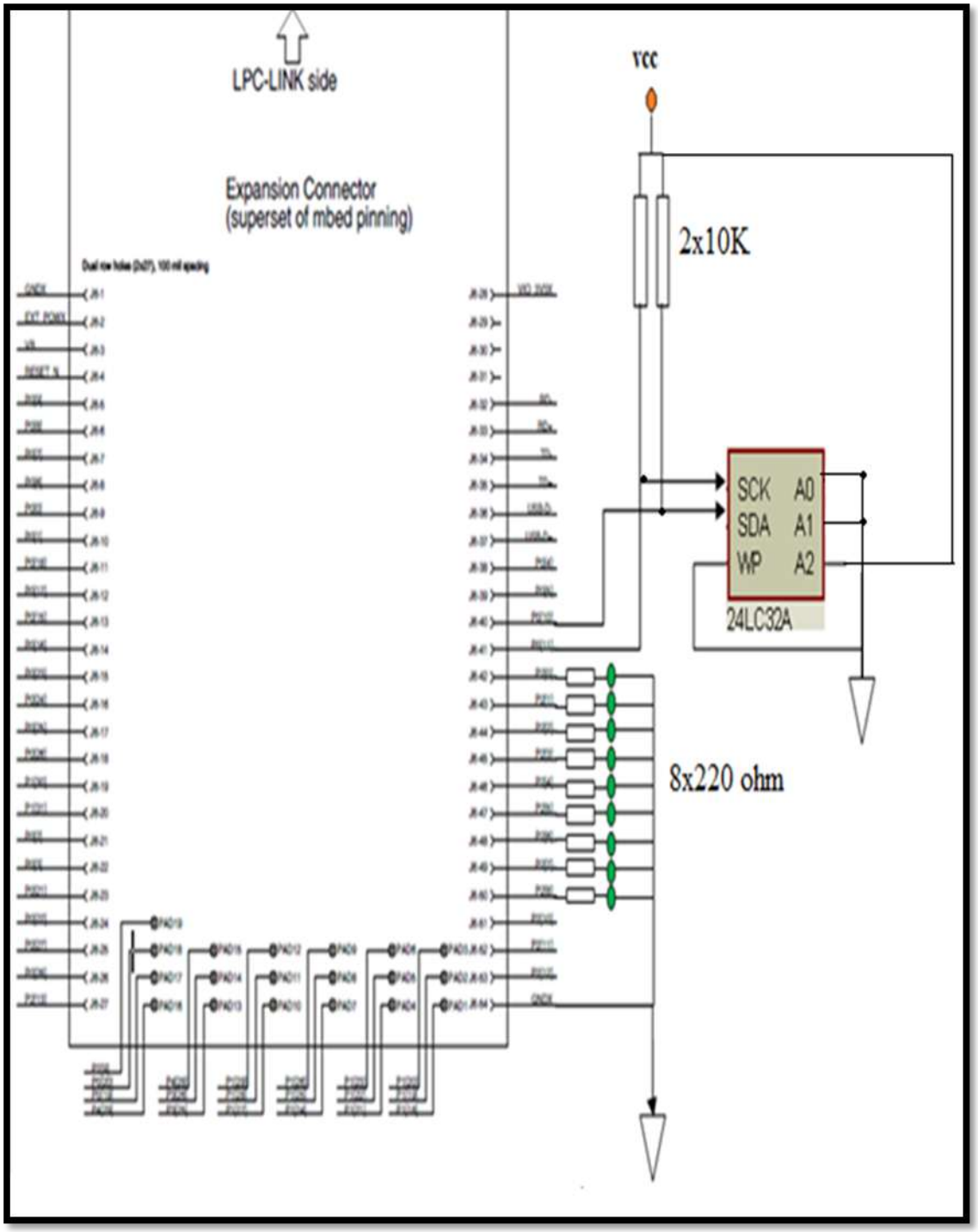

Figura B.1 Esquema ejercicio 1 Comunicación I2C de la LPC 1769 con EEPROM 24LC32A

Esquema Comunicación I2C entre PIC16f887 y LPC 1769

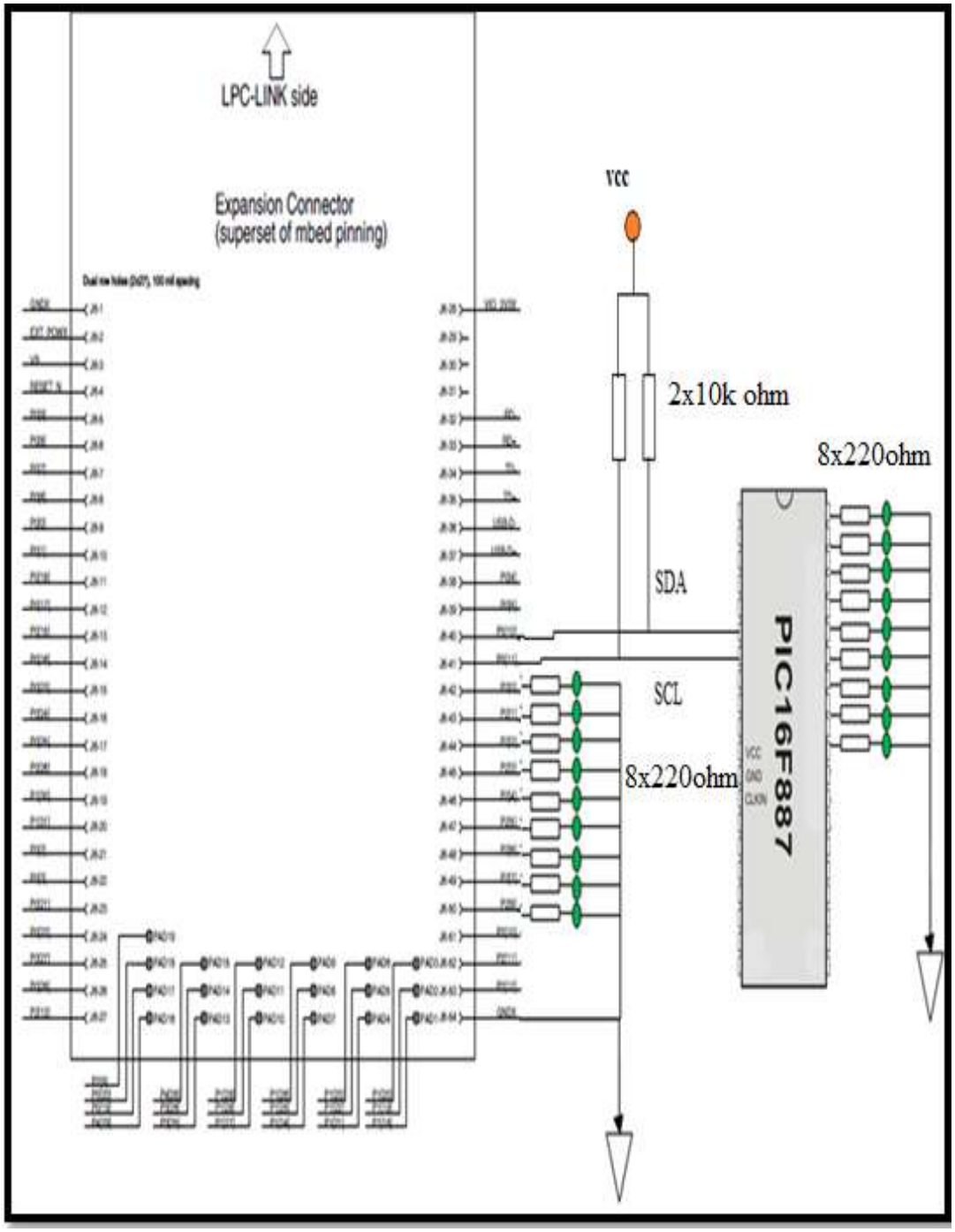

Figura B.2 Esquema ejercicio 2 Esquema Comunicación I2C entre PIC16f887 y LPC 1769

Esquema de Comunicación I2C entre AVR Butterfly y EEPROM 24LC32A

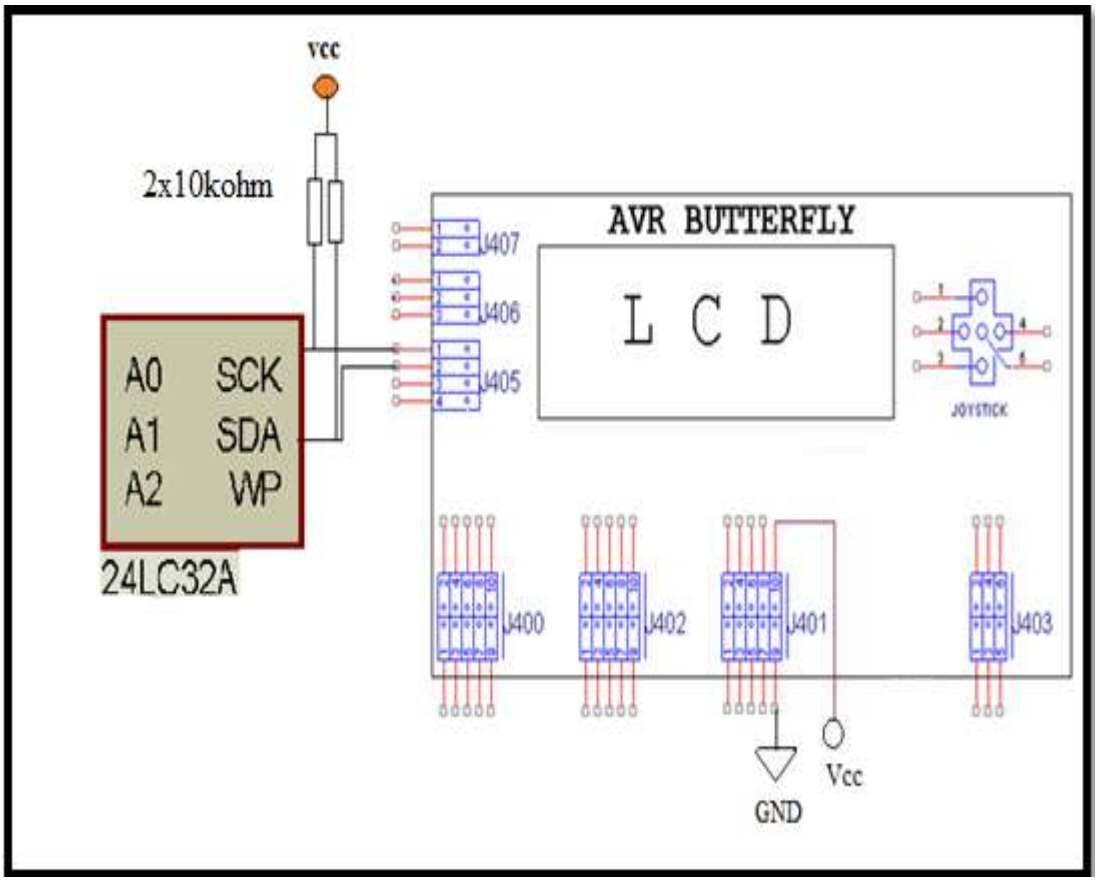

Figura B.3 Esquema ejercicio 3 Comunicación I2C entre AVR Butterfly y EEPROM 24LC32A

Proyecto Final: Esquema comunicación I2C entre la AVR Butterfly y la LPCXpresso 1769

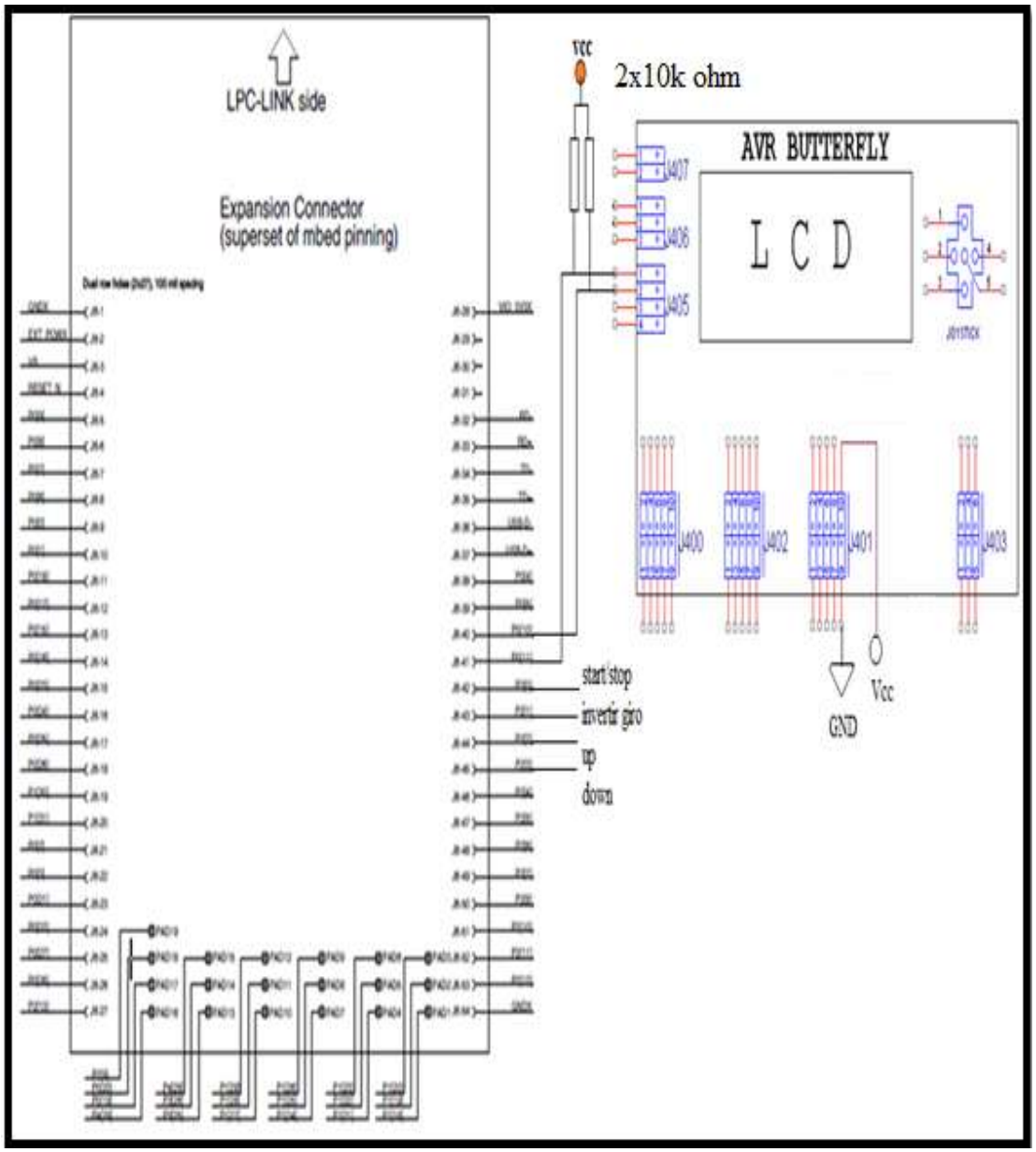

Figura B.4 Esquema Proyecto final Comunicación I2C entre la AVR Butterfly y la LPC1769

ANEXO C

GUÍA DE PROGRAMACÓN PARA TARJETA AVR BUTTERFLY

PROGRAMACIÓN MEDIANTE CONEXIÓN SERIAL (UART) CON LA PC

El AVR Butterfly tiene incluido un convertidor de nivel para la interfaz RS-232. Esto significa que no se necesita de hardware especial para reprogramar al AVR Butterfly utilizando la característica self-programming del ATmega169. A continuación se explica brevemente la distribución de los pines y como se debe realizar el cableado para la comunicación serial entre el AVR Butterfly y la PC.

Distribución de pines para la comunicación serial entre el AVR y la PC

La comunicación con la PC requiere de tres líneas: TXD, RXD y GND. TXD es la línea para transmitir datos desde la PC hacia el AVR Butterfly, RXD es la línea para recepción de datos enviados desde el AVR Butterfly hacia la PC y GND es la tierra común. En la Tabla 2 se observa la distribución de los pines para la comunicación serial, a la izquierda los pines del AVR Butterfly y a la derecha los pines del conector DB9 de la PC.

AVR Butterfly UART	COM2
Pin 1 (RXD)	Pin 3
Pin 2 (TXD)	Pin 2
Pin 3 (GND)	Pin 5

Tabla C.1 Distribución de pines, AVR Butterfly Vs PC

En la Figura A.2 se observa cómo se debe hacer el cableado para la comunicación, a través de la interfaz serial RS-232, entre el AVR Butterfly y la PC. A la izquierda se aprecia un conector DB9 hembra soldado a los cables que se conectan a la interfaz USART del AVR Butterfly (derecha).

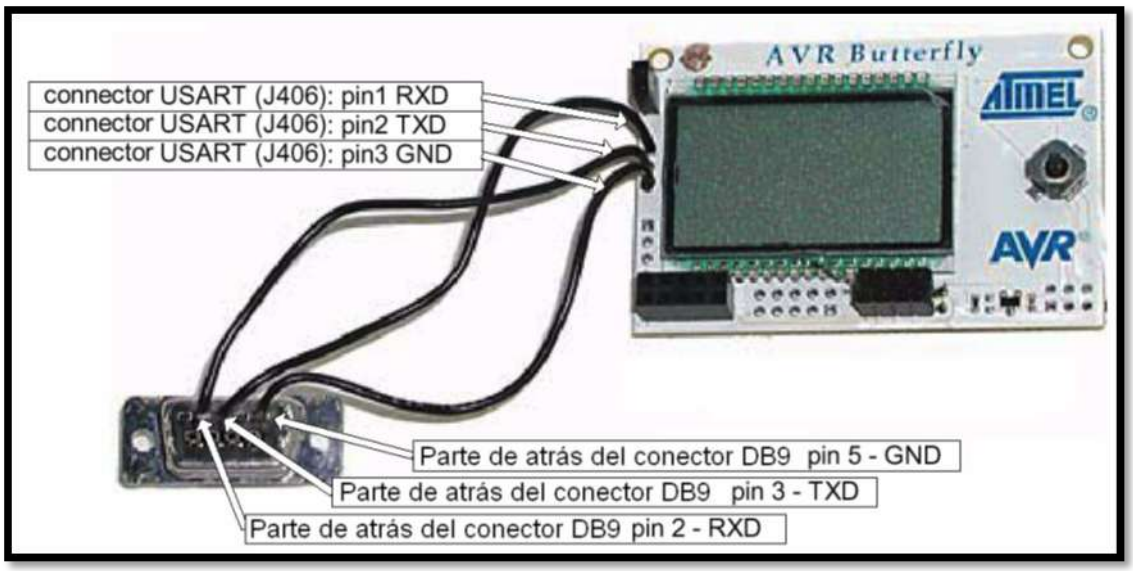

Fig. A.4 Conexiones para interfaz USART del AVR Butterfly

Programación del AVR Butterfly

Los pasos necesarios para la Programación del Kit AVR Butterfly son los siguientes:

En la PC, localizar y ejecutar el AVR Studio.

En el menú “File” del AVR Studio seleccionar “Open File“, luego seleccionar el archivo con el que se desea programar al AVR Butterfly, tal como indica la Figura; por ejemplo: …\nombre_del_programa.cof.

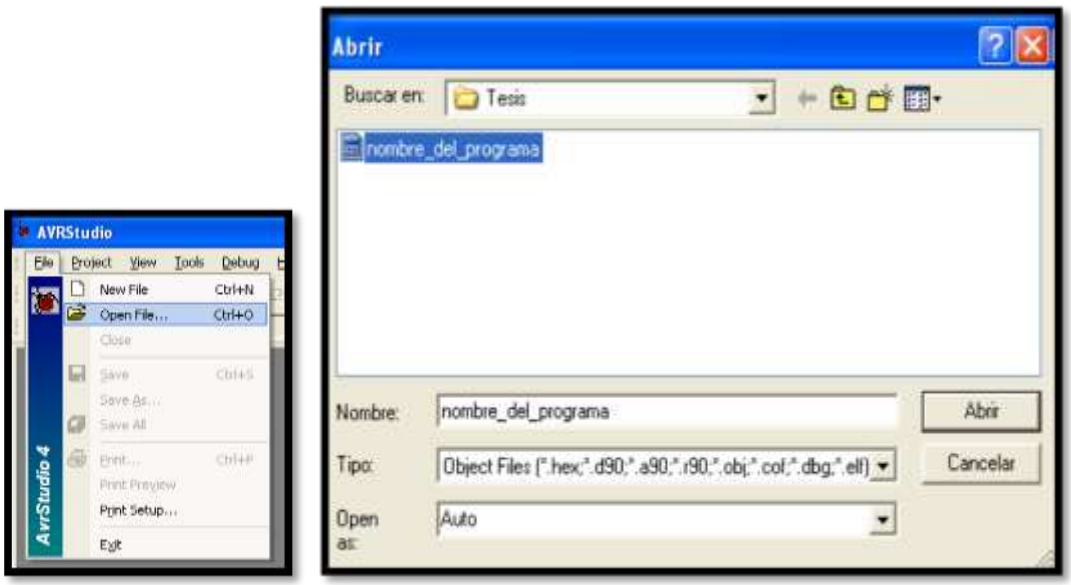

Fig. A.5 AVR Studio, selección del archivo COF para depuración

Seleccionar el AVR Simulator y luego el ATmega169 como en la Figura A.4

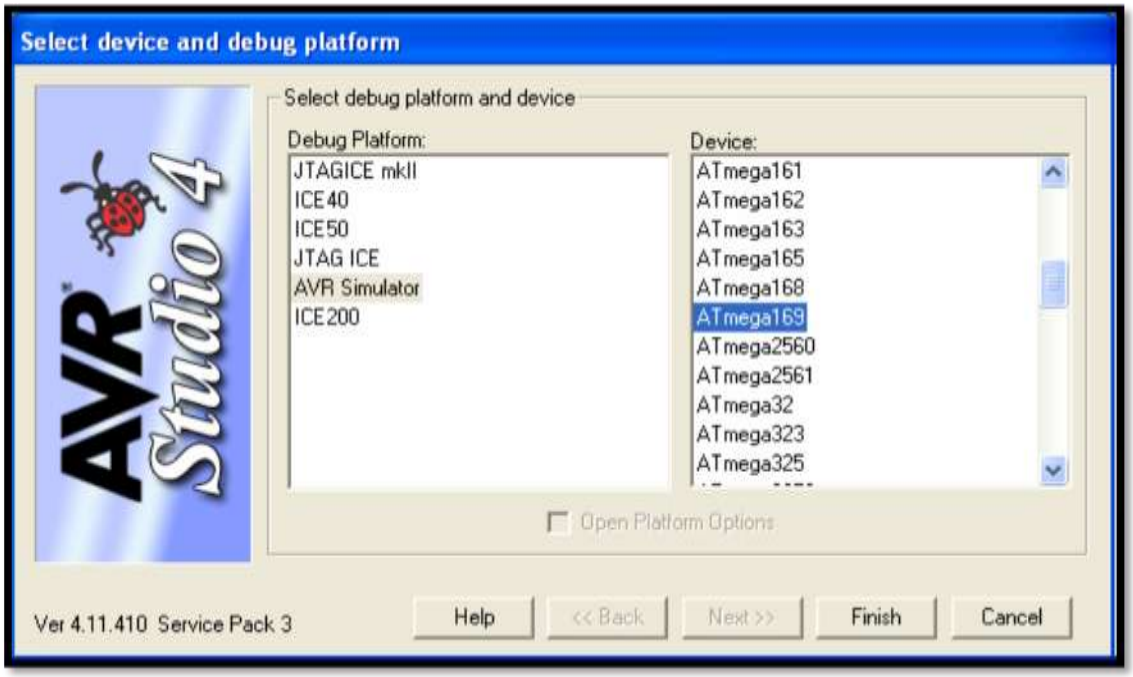

Fig. A.6 Butterfly Selección del AVR Simulator y Dispositivo ATmega169

- Presionar “Finish”.
- Conectar el cable serial entre la PC y el AVR Butterfly como se indica en la Figura A.2.
- Resetear el AVR Butterfly cortocircuitando los pines 5 y 6 en el conector J403, conector ISP, o quitando y aplicando nuevamente la fuente de alimentación.
- Luego de un reset, el microcontrolador ATmega169 comenzará desde la sección de Arranque. Nada se desplegará en el LCD mientras esté en la sección de Arranque. Entonces se deberá presionar ENTRAR en el joystick y mantener esa posición; mientras tanto desde la PC en el AVR Studio, iniciar el AVR Prog.
- Una vez que se haya iniciado el AVR Prog, soltar el joystick del AVR Butterfly. Desde el AVR Prog, utilizar el botón “Browse” para buscar el archivo con la extensión *.hex con el que desea actualizar al AVR Butterfly. Una vez localizado el archivo *.hex, presionar el botón “Program”. Se notará que “Erasing Device”, “Programming” y “Verifying” se ponen en “OK”, de manera automática. Luego de actualizar la aplicación, presionar el botón “Exit” en el AVR Prog para salir del modo de programación del ATmega169.
- Para que empiece a ejecutarse la nueva aplicación, resetear el AVR Butterfly cortocircuitando los pines 5 y 6 en el conector J403 y presionar el joystick hacia ARRIBA. [2]

ANEXO D

HERRAMIENTAS DE HARDWARE Y SOFTWARE

Tarjeta AVR Butterfly

La tarjeta AVR Butterfly se diseñó para demostrar los beneficios y las características importantes de los microcontroladores ATMEL.

El AVR Butterfly utiliza el microcontrolador AVR ATmega169V, que combina la Tecnología Flash con un versátil microcontrolador disponible. En la figura D.1 se puede apreciar la tarjeta AVR Butterfly.

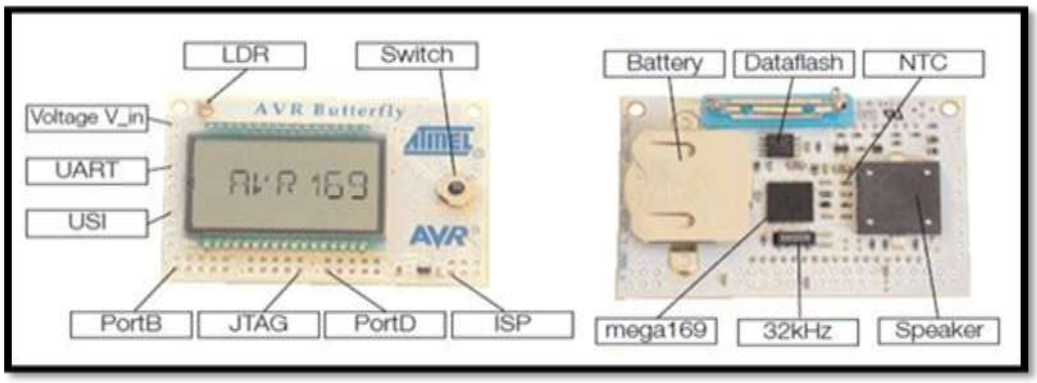

Figura D.7 Hardware disponible del kit de desarrollo AVR Butterfly

Características de la tarjeta AVR Butterfly

El AVR Butterfly expone las siguientes características principales:

- Un controlador LCD.
- Memorias:
- EEPROM, Flash, SRAM.
- DataFlash externa.
- Interfaces de comunicación:
- SPI, UART, USI.
- Métodos de programación
- Self-Programming/Bootloader, SPI, Paralelo, JTAG.
- Convertidor Analógico Digital (ADC).

- Temporizadores/Contadores:
- Contador de Tiempo Real (RTC).
- Modulación de Ancho de Pulso (PWM).

El AVR Butterfly está orientado para el desarrollo de diversas aplicaciones con el ATmega169 y además se puede usar como un módulo dentro de otros dispositivos.

Los siguientes recursos están disponibles:

- Microcontrolador ATmega169V (en encapsulado tipo MLF).
- Pantalla tipo vidrio LCD de 120 segmentos
- Joystick de cinco direcciones, incluida la presión en el centro.
- Altavoz piezoeléctrico, para reproducir sonidos.
- Cristal de 32 KHz para el RTC.
- Memoria DataFlash de 4 Mbit, para el almacenar datos.
- Convertidor de nivel RS-232 e interfaz USART, para comunicarse con unidades fuera del Kit sin la necesidad de hardware adicional.
- Termistor de Coeficiente de Temperatura Negativo (NTC), para sensar y medir temperatura.
- Resistencia Dependiente de Luz (LDR), para sensar y medir intensidad luminosa.
- Acceso externo al canal 1 del ADC del ATmega169, para lectura de voltaje en el rango de 0 a 5 V.
- Emulación JTAG, para depuración.
- Interfaz USI, para una interfaz adicional de comunicación.
- Terminales externas para conectores tipo Header, para el acceso a periféricos.

- Batería de 3 V tipo botón (600mAh), para proveer de energía y permitir el funcionamiento del AVR Butterfly.
- Bootloader, para programación mediante la PC sin hardware especial.
- Aplicación demostrativa pre programada.
- Compatibilidad con el Entorno de Desarrollo AVR Studio 4. [1]

Partes de la tarjeta AVR Butterfly

Son las siguientes:

Joystick

Para usar el AVR Butterfly se hace uso del joystick como una entrada que tiene acceso el usuario. Este funciona en cinco direcciones, incluyendo presión al centro; así como lo muestra la figura D.2.

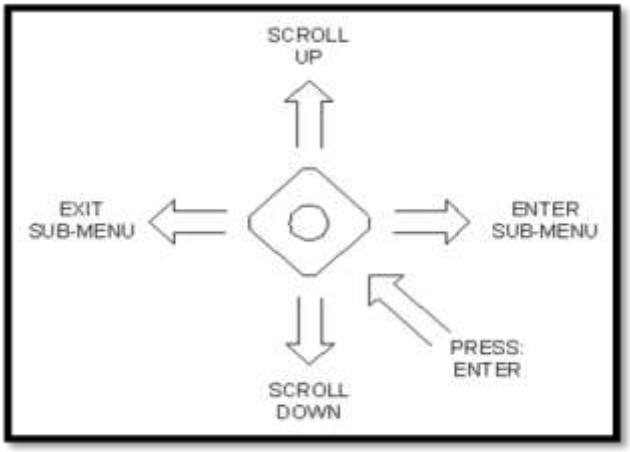

Figura D.8 Entrada tipo joystick

Conectores

Algunos de los pines de I/O del microcontrolador ATmega169 están disponibles en los conectores del AVR Butterfly. Estos conectores sirven para programación, comunicación y entrada al ADC del microcontrolador. Como es mostrado en la siguiente figura.

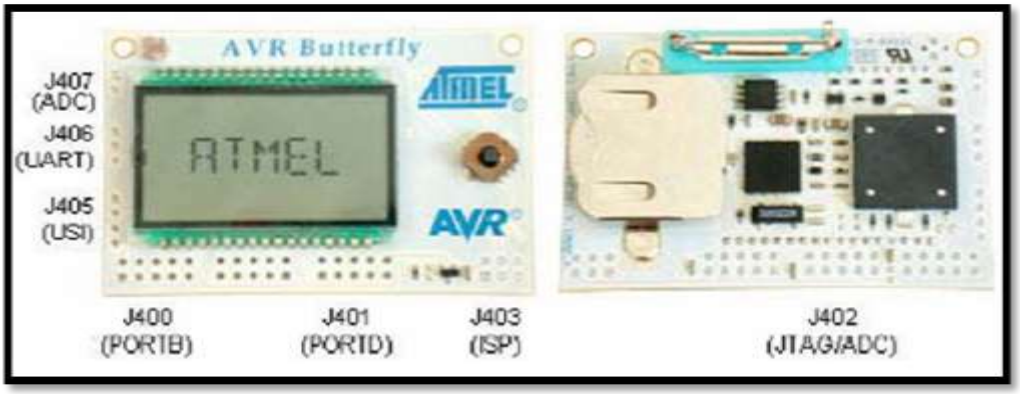

Figura. D.9 Conectores del AVR Butterfly para acceso a periféricos.

El LCD

En las aplicaciones donde se necesita interactuar con el usuario es de mucha utilidad mostrar la información para el usuario. Una interfaz muy simple para mostrar información podría ser usando LEDS; mientras que otra interacción un poco más compleja puede beneficiarse a través de una pantalla capaz de desplegar letras, números, palabras. Las Pantallas LCD se usan con frecuencia para desplegar mensajes. Los módulos del LCD pueden ser gráficos y se los puede usar para desplegar texto, o pueden ser alfanuméricos entre 10 y 80 caracteres.

Los módulos LCD alfanuméricos estándar son muy sencillos de conectar, pero son de alto costo monetario debido a que tienen incorporado drivers (controladores) que se ocupan de la generación de los caracteres sobre el LCD.

El microcontrolador ATmega169 tiene un controlador LCD (LCD Driver) integrado capaz de controlar hasta 100 segmentos. El núcleo altamente eficiente y el consumo de corriente muy bajo de este dispositivo lo hace ideal para aplicaciones energizadas por batería que requieren

de una interfaz humana.

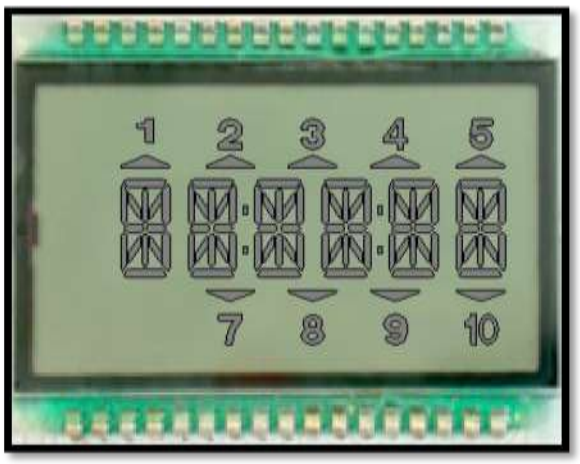

Figura D.10 Pantalla LCD

LPCXpresso

El LPCXpresso es una herramienta para evaluación y desarrollo con microcontroladores de NXP.

Contiene:

- LPCXpresso IDE y "development tools"
- IDE basado en Eclipse
- Compilador y linker GNU
- GDB depurador
- LPCXpresso target board (stick)
- BaseBoard o hardware adicional (opcional)

El target board es un microcontrolador con todo lo necesario para encender y también es una herramienta que incluye un depurador y un programador.

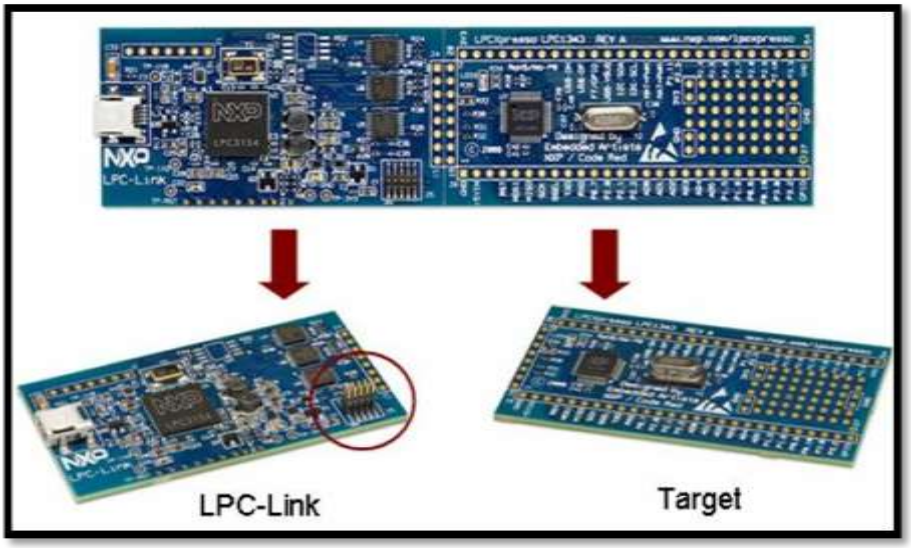

Figura D.5 Tarjeta de desarrollo LPCXpresso

Del lado del target están incluidos algunos periféricos básicos y se comercializan con diferentes tipos de microcontroladores. Pero en nuestro caso usaremos el siguiente:

- LPC1769: ARM Cortex-M3, 512KB flash, 64KB SRAM, Ethernet, USB On the go.

La placa contiene periféricos para desarrollo y experimentación:

Generales:

- Socket para LPCXpresso y un módulo mbed.
- Pequeña batería de alimentación en forma de moneda.
- Interface USB.
- Pulsador de Reset.

Analógicos:

- Potenciómetro trimmer para entrada analógica.
- Salida PWM y entrada analógica.

- Salida para speaker (salida PWM).
- Etapa de entrada para probador de osciloscopio.

Digitales:

- LED RGB (puede ser controlado con PWM)
- 5-key joysticks switch.
- 2 pulsadores, uno para activar el bootloader.
- Sensor de temperatura con salida PWM.
- Interruptor giratorio con la codificación de cuadratura.

Serial – SPI:

- Registro de desplazamiento conductor de LED de 7 segmentos.
- SD/MMC interface de tarjeta de memoria.
- DataFlash SPI-NOR.

Serial – I2C.

- Puerto expandido PCA9532 conectado a 16 LEDS.
- 8kbit E2PROM.
- Sensor de luz
- MMA7455L acelerómetro con interface I2C.[5]

AVR Studio 4

Es un entorno de desarrollo ensamblador y programador de software para el desarrollo de aplicaciones de Atmel AVR.

AVR Studio incorpora un depurador que permite el control de ejecución con fuente y nivel de instrucción, paso a paso y puntos de interrupción, el registro, la memoria y E/S puntos y configuración y gestión, y apoyo a la programación completa para los programadores independientes además permite crear archivos assembler (asm) y archivos .C.[1]

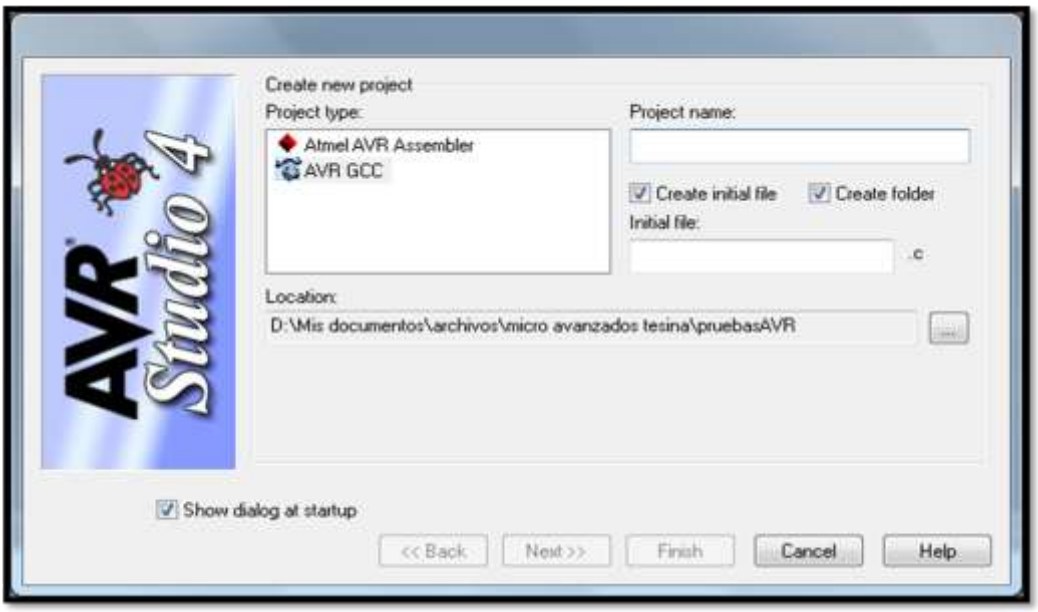

Figura D.6 Ventana de AVR studio 4

Características Principales

- Integrado ensamblador y simulador
- Se integra con el compilador GCC plug-in
- AVR RTOS plug-in de apoyo
- Soporta AT90PWM1 y ATtiny 40.
- Herramientas de CLI al día con el apoyo de TPI
- Ayuda en línea.

AVR Studio cuenta con algunas formas para poder programar los microcontroladores de la familia ATMEL.

Proteus

Es un entorno integrado diseñado para la realización completa de proyectos de construcción de equipos electrónicos en todas sus posibles etapas:

- diseño
- simulación
- depuración
- construcción.

Por su gran alcance actualmente es uno de los simuladores más populares

BIBLIOGRAFÍA

[1] ATMEL, AVR Butterfly Evaluation Kit – User Guide, http://www.atmel.com/dyn/resources/prod_documents/doc4271.pdf, fecha de consulta marzo 30 del 2012.

[2] Escuela Superior Politécnica del Ejército, Características del Kit AVR Butterfly en español, http://www.espe.edu.ec/repositorio/T-ESPE-014271.pdf, fecha de consulta marzo 30 del 2012.

[3] Microchip Technology Inc, Brushless DC (BLDC) Motor Fundamentals, http://electrathonoftampabay.org/www/Documents/Motors/Brushless DC (BLDC) Motor Fundamentals.pdf, fecha de consulta marzo 15 del 2012.

[4] Díaz García Antonio, Conceptos básicos del bus I2C, http://atc.ugr.es/~afdiaz/fich/bus_i2c.pdf, fecha de consulta abril 1 del 2012.

[5] Fernández Moreno Antonio, Conceptos básicos del bus I2C,http://www.uco.es/~el1mofer/Docs/IntPerif/Bus_I2C.pdf, fecha de consulta abril 1 del 2012.

[6] NXP Semiconductors, LPC17xx User manual, http://es.scribd.com /doc/37637774/User-manual-lpc17xx, fecha de consulta marzo 15 del 2012.

[7] Lecaro Andrés, González José, Valdivieso Carlos, Controlador PID de temperatura utilizando la tarjeta AVR Butterfly www.dspace.espol.edu .ec/bitstream/123456789/.../tesis%20final.doc, fecha de consulta marzo 16 del 2012.

[8] NXP Semiconductors, Lpc 1769/68/67/66/65/64/63, www.nxp. com/documents/data_sheet/LPC1769_68_67_66_65_64_63.pdf fecha de consulta marzo 17 del 2012.

[9] García V., Proteus Manual, http://www.hispavila.com/3ds_/chipspic /manualproteus.html, fecha de consulta marzo 30 del 2012.

[10] Burgess Mark, Programación en C, http://www.iu.hio.no/~mark /CTutorial/CTutorial.html, fecha de consulta marzo 12 del 2012.

[11] Atmel, Atmega169, http://www.atmel.com/images/doc2514.pdf fecha de consulta marzo 29 del 2012.