



**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 Y  
PRESENTACIÓN EN DISPLAY DE MENSAJE DE OPERACIÓN”

**TESINA DE SEMINARIO**

**Previa la obtención del Título de:**

**INGENIERO EN ELECTRÓNICA Y TELECOMUNICACIONES**

**Presentado por:**

**José David Jiménez Miranda**

**Luis Israel Pabón Orozco**

**GUAYAQUIL – ECUADOR**

**AÑO 2012**

## **AGRADECIMIENTO**

A Dios, por darnos la vida y la oportunidad de compartirla con todos los que conocemos. A mi familia por su apoyo incondicional. A mis amigos por su continuo aliento. A mi compañero que por nuestro esfuerzo conjunto logramos esta meta.

Luis Pabón

A Dios. A mi familia, en especial a mis padres por su comprensión y cariño que me han brindado durante todos los días de mi vida. A mis amigos y a todas las personas que apoyaron en el desarrollo de este trabajo.

José Jiménez

## DEDICATORIA

A mi familia, mis amigos por sobre todo a Manuel Villavicencio que en paz descanse. A todos aquellos que me brindaron su apoyo y amistad durante mi carrera en la ESPOL y en el proyecto. A Dios, y a la vida.

Luis Pabón

A mis padres y demás seres queridos, que gracias a sus consejos y apoyo incondicional me guiaron por el camino del bien y han sido mi mayor fortaleza para alcanzar este gran objetivo.

José Jiménez

## TRIBUNAL DE SUSTENTACIÓN



M. Sc. Carlos Valdivieso

Profesor Seminario de Graduación



Ing. Ronald Ponguillo

Profesor Delegado por la Unidad Académica

## DECLARACIÓN EXPRESA

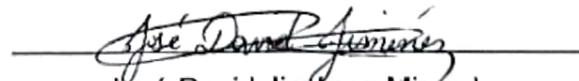
“La responsabilidad del contenido, hechos, ideas y doctrinas expuestos en esta Tesina, nos corresponde exclusivamente, y el patrimonio intelectual de la misma a la ESCUELA SUPERIOR POLITECNICA DEL LITORAL.”

(Reglamentos, exámenes y títulos profesionales de la ESPOL)



---

Luis Israel Pabón Orozco



---

José David Jiménez Miranda

## RESUMEN

El presente documento detalla análisis, investigación, implementación y desarrollo de conocimientos aprendidos en nuestra vida académica con respecto al uso de microcontroladores, centrándonos en el estudio de las características y funcionamiento de la tarjeta LPCXpresso LPC1769 basada en MCU ARM Cortex M3 y del AVR Butterfly con microcontrolador ATmega169; en la elaboración de un código que permita la comunicación entre dispositivos a través del protocolo I2C para la manipulación y control de motores BLDC.

El proyecto consiste en manipular el joystick del AVR Butterfly, enviar comandos por protocolo de transmisión I2C a la tarjeta LPCXpresso LPC1769 que a través de un módulo transmitirá las órdenes a los motores. Una vez ejecutada la orden dada, se procesa desde la LPCXpresso LPC1769 un código de confirmación que se presentará en el LCD del AVR Butterfly.

Para esto utilizaremos los modos de Lectura y Escritura de ambos módulos, ya sea como Esclavo o Maestro, según el caso, a través de un código en lenguaje C.

## ÍNDICE GENERAL

AGRADECIMIENTO .....	I
DEDICATORIA .....	II
TRIBUNAL DE SUSTENTACIÓN .....	III
DECLARACIÓN EXPRESA .....	IV
RESUMEN .....	V
ÍNDICE GENERAL.....	VI
ÍNDICE DE FIGURAS .....	XIII
ÍNDICE DE TABLAS .....	XVII
ÍNDICE DE ABREVIATURAS .....	XVIII
INTRODUCCIÓN .....	XXI

### CAPÍTULO 1: Descripción General del Proyecto

1.1 Antecedentes .....	1
1.2 Objetivos .....	3
1.3 Identificación del problema .....	4
1.4 Limitaciones .....	4

## CAPÍTULO 2: Fundamento Teórico

2.1 Introducción .....	6
2.2 Motores eléctricos .....	7
2.3 Motores Brushless.....	8
2.3.1 Principio de Funcionamiento del BLDC.....	12
2.3.2 Alimentación del BLDC .....	14
2.3.3 Conmutación del BLDC.....	15
2.3.4 Comparación entre motores con escobillas y sin escobillas ...	18
2.3.5 Sensores utilizados en motores BLDC.....	19
2.3.5.1 Encoder.....	19
2.3.5.2 Sensor de Efecto Hall .....	20
2.4 Comunicación Serial.....	21
2.5 Comunicación I2C .....	23
2.5.1 Características del bus I2C .....	23
2.5.2 Estados del Bus I2C.....	25
2.5.3 Formato de la transmisión.....	27
2.5.3.1 Gobierno de la señal de sincronía .....	28
2.5.3.2 Mecanismo de sincronización .....	29
2.5.3.3 Filosofía del reconocimiento .....	30
2.5.3.4 Caracteres de dirección .....	31
2.5.3.5 Llamado general .....	33

2.5.3.6 Caracter de inicio .....	33
2.6 Herramientas de diseño.....	34
2.6.1 AVR Studio.....	34
2.6.2 LPCXpresso 4 .....	36
2.6.3 Proteus.....	37
2.7 Descripción del Hardware.....	38
2.7.1 AVR Butterfly.....	38
2.7.1.1 Características .....	39
2.7.2 LPCXpresso 1769 .....	40
2.7.2.1 Características principales .....	41
2.7.3 LPCXpresso 1114 .....	42
2.7.4 LPCXpresso Motor Control Kit .....	44
2.8 Introducción a los microcontroladores .....	45
2.8.1 Estructura General de un microcontrolador.....	46
2.9 MCU AVR ATmega169.....	47
2.9.1 Características Generales .....	47
2.9.2 Distribución de pines .....	50
2.9.3 Revisión Global.....	51
2.9.4 Diagrama de bloques.....	51
2.9.5 CPU AVR CORE .....	53
2.9.6 Puertos de E/S.....	55

## CAPÍTULO 3: Diseño e implementación del proyecto

3.1 Introducción .....	57
3.2 Ejercicio 1 .....	58
3.2.1 Introducción.....	58
3.2.2 Diagrama de bloques .....	58
3.2.3 Diagrama de flujo .....	59
3.2.4 Descripción del algoritmo .....	60
3.2.5 Código Fuente .....	61
3.3. Ejercicio 2.....	63
3.3.1 Introducción.....	63
3.3.2 Diagrama de bloques .....	63
3.3.3 Diagrama de flujo .....	64
3.3.3.1 PIC16F887 .....	64
3.3.3.2 LPCXpresso 1769.....	65
3.3.4 Descripción del algoritmo .....	66
3.3.4.1 PIC16F887.....	66
3.3.4.2 LPCXpresso 1769.....	67
3.3.5 Código Fuente .....	68
3.3.5.1 PIC16F887.....	68
3.3.5.2 LPCXpresso 1769.....	68

3.4 Ejercicio 3 .....	70
3.4.1 Introducción.....	70
3.4.2 Diagrama de bloques .....	70
3.4.3 Diagrama de flujo .....	71
3.4.4 Descripción del algoritmo .....	72
3.4.5 Código Fuente .....	74
3.5 Proyecto Final de graduación .....	77
3.5.1 Introducción.....	77
3.5.2 Diagrama de bloques .....	78
3.5.3 Diagrama de flujo .....	78
3.5.3.1 AVR Butterfly .....	78
3.5.3.2 LPCXpresso 1769.....	80
3.5.4 Descripción del algoritmo .....	81
3.5.4.1 PIC16F887 .....	81
3.5.4.2 LPCXpresso 1769.....	83
3.5.5 Código Fuente .....	85
3.5.5.1 AVR Butterfly ATmega 169.....	85
3.5.5.2 LPCXpresso 1769.....	87

## CAPÍTULO 4: Desarrollo y Simulación del proyecto

4.1. Introducción .....	88
4.2. Elementos físicos.....	89
4.2.1. Plataforma del proyecto.....	89
4.2.1.1 Protoboard .....	89
4.2.1.2 Kit AVR Butterfly ATmega 169.....	91
4.2.1.3 Kit LPCXpresso LPC 1769.....	91
4.2.1.4 Kit de Control de Motores NXP .....	92
4.3 Imágenes y simulaciones.....	93
4.3.1. Ejercicio 1 .....	93
4.3.1.1 Conclusiones .....	95
4.3.2. Ejercicio 2.....	97
4.3.2.1 Conclusiones .....	99
4.3.3. Ejercicio 3.....	101
4.3.3.1 Conclusiones .....	104
4.3.4. Proyecto de graduación.....	105
4.3.4.1 Conclusiones .....	111

CONCLUSIONES

RECOMENDACIONES

ANEXO 1

ANEXO 2

BIBLIOGRAFIA

## ÍNDICE DE FIGURAS

Figura 1.1: Coolers de ordenadores como aplicación de motores BLDC .....	2
Figura 2.1: Partes fundamentales del motor eléctrico.....	8
Figura 2.2: Esquema de un motor BLDC .....	9
Figura 2.3: Esquema de motor Inrunner y Outrunner .....	10
Figura 2.4: Esquema general de un motor BLDC .....	13
Figura 2.5: Motor Inrunner con alimentación trifásica tipo estrella .....	14
Figura 2.6: motor de F.E.M. trapezoidal .....	16
Figura 2.7: Forma básica de intensidad y tensión de un BLDC .....	17
Figura 2.8: Niveles de Voltaje para cada fase y niveles lógicos en sensores Hall versus tiempo de manejo para un giro a favor del reloj .....	17
Figura 2.9: Piezas de un encoder óptico incremental de 500 pulsos por vuelta y tres canales.....	20
Figura 2.10: Condición del bus libre.....	25
Figura 2.11: Condición de Inicio.....	26
Figura 2.12: Condiciones de cambio y datos .....	26
Figura 2.13: Condición de parada.....	27
Figura 2.14: Presentación inicial del AVR Studio 4.....	34
Figura 2.15: Interfaz inicial del AVR Studio 4.....	35
Figura 2.16: Interfaz gráfica de LPCXpresso 4 .....	36

Figura 2.17: Interfaz gráfica del Proteus .....	37
Figura 2.18: Vista AVR Butterfly .....	38
Figura 2.19: Tarjeta LPC1769.....	40
Figura 2.20: Descripción de partes de la LPC 1769.....	42
Figura 2.21: Tarjeta LPC 1114.....	42
Figura 2.22: Vista de circuito LPC 1114.....	43
Figura 2.23: Vista LPCXpresso Motor Control Board.....	44
Figura 2.24: Distribución de tarjeta de Control de Motores de NXP.....	45
Figura 2.25: Estructura básica de un microcontrolador.....	47
Figura 2.26: Pines del MCU ATmega169 .....	50
Figura 2.27: Diagrama de Bloques del ATmega169 .....	51
Figura 2.28: Diagrama de Bloques de la Arquitectura AVR .....	54
Figura 2.29: Diagrama equivalente del GPIO .....	55
Figura 3.1: Diagrama de Bloques ejercicio 1 .....	58
Figura 3.2: Diagrama de Flujo ejercicio 1 .....	59
Figura 3.3: Diagrama de Bloques ejercicio 2 .....	63
Figura 3.4: Diagrama de Flujo ejercicio 2 – PIC 16F887 .....	64
Figura 3.5: Diagrama de Flujo ejercicio 2 – LPCXpresso 1769 .....	65
Figura 3.6: Diagrama de Bloques ejercicio 3 .....	70
Figura 3.7: Diagrama de Flujo ejercicio 3 .....	71
Figura 3.8: Diagrama de Bloques Proyecto Final.....	78

Figura 3.9: Diagrama de Flujo Proyecto Final - AVR Butterfly .....	79
Figura 3.10: Diagrama de Flujo Proyecto Final - LPCXpresso 1769.....	80
Figura 4.1: Protoboard .....	90
Figura 4.2: Protoboard con plataforma de ejercicios y proyecto final.....	90
Figura 4.3: AVR Butterfly implementado en ejercicios y proyecto conectado al protoboard .....	91
Figura 4.4: LPCXpresso 1769 implementada en el protoboard .....	92
Figura 4.5: Módulo de control de motores NXP y motor BLDC.....	93
Figura 4.6: Caracter "F" .....	94
Figura 4.7: Caracter "I" .....	94
Figura 4.8: Caracter "E" .....	95
Figura 4.9: Caracter "C" .....	95
Figura 4.10: Caracter "E" .....	97
Figura 4.11: Caracter "S" .....	98
Figura 4.12: Caracter "P" .....	98
Figura 4.13: Caracter "O" .....	99
Figura 4.14: Caracter "L" .....	99
Figura 4.15: Mensaje inicial de simulación .....	101
Figura 4.16: Presentación de mensaje al presionar ENTER.....	102
Figura 4.17: Presentación de mensaje al mover joystick hacia la izquierda	103
Figura 4.18: Presentación de mensaje al mover joystick hacia arriba .....	104

Figura 4.19: Presentación de mensaje al mover joystick hacia abajo.....	104
Figura 4.20: Presentación inicial de LCD de mensaje "PROYECT".....	105
Figura 4.21: Presentación en display de 7 segmentos de instrucción START .....	106
Figura 4.22: Funcionamiento después de presionar ENTER en el joystick	106
Figura 4.23: Presentación en display de 7 segmentos de instrucción REVERS .....	107
Figura 4.24: Funcionamiento después de mover a la izquierda el joystick	108
Figura 4.25: Presentación en display de 7 segmentos de instrucción INC .	108
Figura 4.26: Funcionamiento después de mover hacia arriba el joystick ...	109
Figura 4.27: Presentación en display de 7 segmentos de instrucción DEC	110
Figura 4.28: Funcionamiento después de mover hacia abajo el joystick ...	110
Figura 4.29: Presentación en display de 7 segmentos de estado de espera .....	111

## ANEXOS

Librería I2cslave.h para uso del modo esclavo de la LPCXpresso1769

Programa I2cslave2.c para uso del modo esclavo de la LPCXpresso1769

Programa USI\_TWI\_Master.c para el uso de I2C del AVR Butterfly como maestro

Figura A: Configuración de pines del módulo LPCXpresso 1769

## ÍNDICE DE TABLAS

Tabla 2.1: Comparación entre motores brushless y con escobillas .....	19
Tabla 2.2: Significado del primer bit de la transacción.....	32

## ÍNDICE DE ABREVIATURAS

### **AVR**

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

### **EEPROM**

Electrically Erasable Programmable Read Only Memory. Tipo de memoria de lectura solamente borrarle y programable eléctricamente.

### **HEX**

Tipo de archivo hexadecimal que contiene únicamente datos para la memoria de programa de los microcontroladores.

### **IDE**

Integrated Development Environment, Ambiente Integrado de Desarrollo. Software usado para el Desarrollo de Proyectos con Microcontroladores.

### **LCD**

Liquid Crystal Display, Pantalla de Cristal Líquido. Están formadas por dos filtros polarizantes con filas de cristales líquidos alineados perpendicularmente; aplicando una corriente eléctrica a los filtros se consigue que la luz pase o no dependiendo de que lo permita o no el segundo filtro.

### **LED**

Light Emitting Diode, Diodo Emisor de Luz. Una tensión aplicada al semiconductor da como resultado la emisión de energía luminosa.

**MCU**

Microcontroller Central Unit. Es la unidad de procesamiento o núcleo de los microcontroladores.

**Memoria FLASH**

Chip de memoria no volátil (su contenido permanece aunque se desconecte de la fuente) que se puede reescribir. En cierto sentido se considera una variante de la EEPROM; la diferencia está en que mientras esta última se borra y programa al nivel de byte, la memoria flash se puede borrar y reprogramar en unidades de memoria llamadas bloques, cuyo tamaño puede ir desde los 512 bytes hasta los 256 KB. Esto hace que la memoria flash sea muy útil para actualizar la BIOS de un ordenador o computadora, o para almacenar cantidades de información importantes, como una colección de imágenes o de documentos de texto, etc.

**MIPS**

Million Instructions Per Second. Medición de velocidad y el poder de un procesador tomando como referencia el número de instrucciones máquina que puede ejecutar en un segundo.

**MLF**

Micro Lead Frame. Tipo de Encapsulado de Chips.

**RISC**

Reduced Instruction Set Compiler. Procesadores que contienen una Colección Reducida de Instrucciones.

**SET**

Escribir o configurar un bit con el valor de uno lógico.

**SRAM**

Static Random Access Memory. Memoria Estática de solo lectura que conserva la información mientras esté conectada a la tensión de alimentación.

**TQFP**

Thin Quad Flat Pack. Tipo de Encapsulado de Chips.

## INTRODUCCIÓN

Este trabajo trata de una forma clara y amigable las principales características definidas por el protocolo que rige la comunicación I2C y la elaboración de un código para el control mediante el joystick de la tarjeta AVR-Butterfly con tarjeta LPCXpresso, de un motor BLDC. Se presentarán en el LCD de la AVR Butterfly los debidos mensajes de operación.

Todos los conceptos y características de los dispositivos serán ampliados en el presente proyecto y le daremos una aplicación. El proyecto se lo ha estructurado en 4 capítulos que los detallamos a continuación:

El capítulo 1, enfoca de manera general la descripción del proyecto, sus alcances, limitaciones y objetivos que se quiere lograr al finalizar el mismo. Se trata de dar un enfoque al estado actual de las tecnologías en el mundo sobre el potencial de los microcontroladores y en base a esto se propone una posible solución al planteamiento del problema.

El capítulo 2, comprende todo lo que es el marco teórico, microcontroladores Cortex M3, Tarjeta LPC1769, AVR Butterfly, ATmega169, Protocolo de comunicación I2C, y demás conceptos claves en base a los cuales se ha trabajado esta tesis.

El capítulo 3, trata el diseño e implementación del proyecto además de varios ejercicios de pruebas de comunicación serial I2C, que se ha orientado a un ambiente de laboratorio; se realiza un diagrama de bloques básico y general para luego ir explicando el funcionamiento detallado de cada uno de los componentes a través de un diagrama de flujo explicativo y la descripción del código fuente.

En el capítulo 4, se realizan las pruebas y simulaciones para el análisis de resultados en base a los parámetros establecidos, detallando paso a paso los procedimientos y realizando comentarios sobre las respuestas obtenidas.

También redactaremos las debidas conclusiones además de recomendaciones para casos de posibles fallas presentadas por problemas físicos o de configuración de los dispositivos, además de cierta información adicional de utilidad sobre el desarrollo del proyecto.

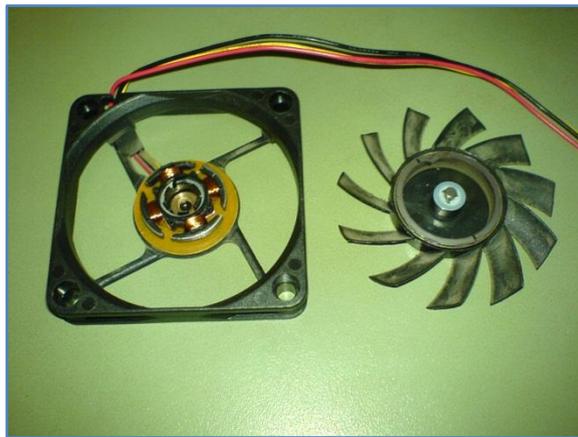
# **Capítulo 1**

## **Descripción General del Proyecto**

### **1.1. Antecedentes**

El presente proyecto es un ejemplo clásico de la aplicación de los protocolos de comunicación serial y fundamentos de control de sistemas de motores que utilizan microcontroladores. Una combinación como esta dará como resultado sistemas sumamente eficientes en consumo de energía, espacio utilizado y precisión, gracias al alto rendimiento de estos integrados.

Los principios por los cuales opera un controlador electrónico de motores sin escobillas son la base de la actual construcción de dispositivos muy utilizados en la vida diaria como lectores de CD-ROM, hard-drives, ventiladores de ordenador, cassettes, etc., y en pasatiempos como aeromodelismo.



**Figura 1.1: Coolers de ordenadores como aplicación de motores BLDC [1]**

Éstas no solo son aplicaciones sumamente interesantes de este tipo de sistemas, sino que también cuentan con un amplio campo de desarrollo y sirven como muestra para futuras innovaciones.

Podemos partir del hecho obvio de que nuestro entorno está constituido por componentes eléctricos y electrónicos para la comodidad de los seres humanos, donde existe una constante interacción en crecimiento con otros tipos de sistemas autónomos en prácticamente cualquier campo.

Desde el siglo pasado, el desarrollo de motores eléctricos ha marcado el avance de la tecnología actual. A finales del siglo XIX Joseph Henry construye el primer motor eléctrico en el desarrollo de tecnologías aplicadas al transporte, que compitió a principios del siglo XX con los motores de combustión interna, habiendo ganado estos últimos por la diferencia de almacenamiento de energía potencial para convertirla en movimiento dado las limitantes de almacenamiento de energía eléctrica en esas épocas. Ahora, se ha retomado el uso de motores eléctricos dado su excelente rendimiento (cercano al 90%), tamaño y potencia, no solo en la industria automotriz para evitar la contaminación ambiental, sino en el aspecto a pequeña y gran escala dentro de la industria por sus múltiples beneficios.

## **1.2. Objetivos**

Nuestro objetivo principal es lograr la comunicación exitosa y acoplamiento entre módulos de desarrollo de distintos fabricantes, tales como la AVR-Butterfly y LPCXpresso LPC1769.

La aplicación de los conocimientos adquiridos durante nuestra etapa estudiantil de tercer nivel, en el área electrónica, programación y de microcontroladores, con el afán de aportar en el desarrollo tecnológico de la sociedad ecuatoriana.

### **1.3. Identificación del problema**

Se desea realizar el control de motores BLDC mediante comandos ingresados por el movimiento del joystick en el AVR-Butterfly, que deberán ser enviados mediante el protocolo de comunicación I2C usando la interfaz serial universal USI (Universal Serial Interface) hacia la tarjeta LPCXpresso 1769, que transformará el dato recibido de manera serial en un código binario transmitido hacia la tarjeta de control de motores BLDC por medio de sus pines GPIO (General Purpose Input/Output). Una vez enviado el código desde la AVR-Butterfly, ésta presentará en su LCD un mensaje que dependerá del código enviado. Básicamente, es un sistema de control del motor BLDC utilizando la AVR-Butterfly como dispositivo externo. Los comandos recibidos por la LPCXpresso LPC1769 serán enviados a la tarjeta de control de motores para el movimiento pertinente del BLDC según la instrucción.

### **1.4. Limitaciones**

Para el desarrollo del proyecto una de las principales limitaciones es el trabajar por primera vez con la familia ARM CORTEX de microcontroladores de LPCXPRESSO lo que implica el manejo e implementación lógica de librerías, dispositivos, componentes, códigos y arquitectura que se encuentran en las tarjetas LPC1114 de control de motores y la LPC1769.

Las tarjetas LPCXpresso son un sistema embebido, ya que por programación tiene aplicaciones múltiples aunque sus características físicas nos permitan ejecutar tareas específicas, por ejemplo el número de pines de la tarjeta LPC1114 es limitada para la interacción con la tarjeta de control del “MOTOR CONTROL BOARD”, aún así es posible adaptarla a nuestras necesidades al momento de ejecutar acciones de movimiento de los motores BLDC.

Un motor BLDC contiene una limitación especial, dado que es un componente externo con necesidad de un control lógico por código, por lo tanto es necesaria una elaboración minuciosa para su correcto funcionamiento.

## **Capítulo 2**

### **Fundamento Teórico**

#### **2.1. Introducción**

En este capítulo se presentan los principales conceptos que vamos a utilizar en este proyecto para familiarizarnos con cada uno de ellos.

El estudio de los microcontroladores AVR-ATMEL y Cortex M3 así como el desarrollo de este trabajo, no consiste solo en dominar su arquitectura interna sino también en conocer programas auxiliares que facilitan el diseño de los sistemas donde intervienen.

A su vez se necesita conocer parte de la codificación de los microcontroladores utilizados en el proyecto y los ejercicios, además de la base teórica y práctica en electrónica analógica y digital para la implementación física de sistemas auxiliares a las tarjetas de desarrollo.

En la programación de los sistemas embebidos de las tarjetas de trabajo AVR-Butterfly y LPCXpresso LPC1769, utilizaremos programas específicos de ambos desarrolladores, para generación de códigos de trabajo basándonos en ejemplos modelos anexados por el fabricante.

## **2.2. Motores Eléctricos**

Un motor eléctrico, de cualquier tamaño o potencia, se basa en los principios de Ampere y de Faraday de electromagnetismo. La ley de Ampere nos describe que el paso de una corriente en una dirección determinada a través de un conductor genera un campo magnético estacionario alrededor de este, en sentido contrario de las agujas del reloj. La ley de Faraday describe que en presencia de un campo magnético estacionario se genera una corriente opuesta al sentido anti horario del campo magnético. Estas leyes nos explican como un motor eléctrico convierte la energía eléctrica en energía mecánica por medio de la interacción electromagnética de sus partes.

Un motor de corriente continua o DC por sus siglas en ingles (Direct Current) consiste en dos partes fundamentales, la parte móvil llamada rotor que se encuentra en el interior y la parte fija llamada estator ubicada en la parte

exterior del motor. En los motores podemos encontrar en el estator los imanes que producen un campo magnético estático que causa el reposicionamiento del motor, mientras que en el rotor se encuentra el devanado de las bobinas, los cuales al pasar una corriente a través de estas producen un campo magnético que se re alinea con los campos producidos en los polos del estator, causando el giro continuo del rotor. En los motores convencionales, la transferencia de corriente entre estator y rotor es llevada a cabo por escobillas (brushes en inglés) que están sujetas al estator y rozan con el rotor. Al observar la figura 2.1 podemos observar las partes principales de un motor eléctrico.



Figura 2.1: Partes fundamentales del motor eléctrico [3]

### 2.3. Motores brushless

Una evolución en los motores eléctricos es sin duda el motor sin escobillas o por sus siglas en inglés BLDC (*Brushless DC*), que en cierta manera invierte

en orden de un motor eléctrico tradicional, es decir, el rotor contiene los imanes mientras varias bobinas se encuentran ubicadas en el estator.

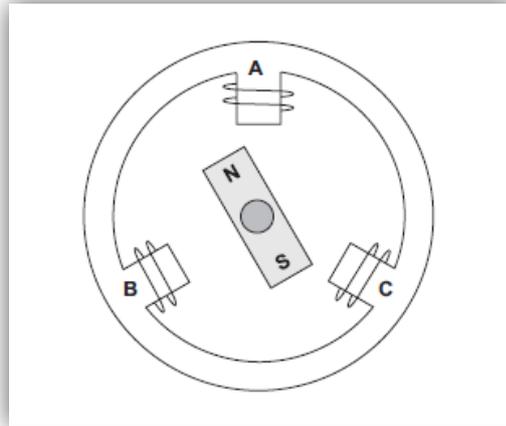


Figura 2.2: Esquema de un motor BLDC [5]

Al no existir las escobillas, la interacción entre rotor y estator es realizada a distancia entre los campos magnéticos generados en las bobinas del estator y la del imán del rotor. Para generar el giro del rotor, en el estator se ubican los devanados que generaran los campos magnéticos de alineación, los cuales no funcionarán todos a la vez sino siguiendo una secuencia óptima para un sentido del giro deseado. Esto hace que estos motores sean un poco más complicados que los convencionales al necesitar un sistema de control para la selección correcta de la bobina.

Según la disposición entre rotor y estator pueden ser clasificados como Inrunner, donde el rotor está en la parte interior, y Outrunner, donde el rotor se encuentra en la parte exterior como se muestra en la figura 2.3.

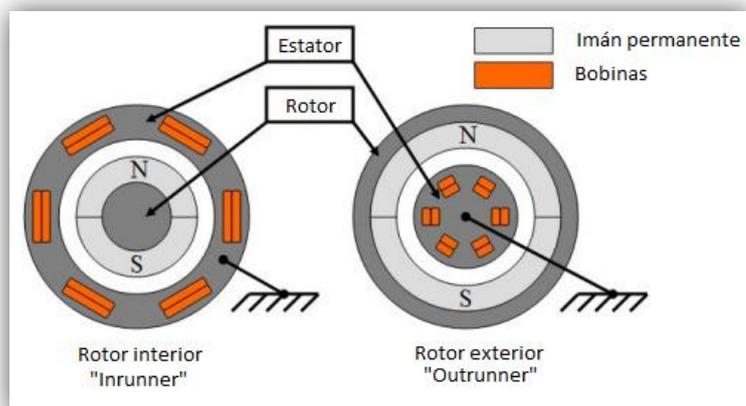


Figura 2.3: Esquema de motor Inrunner (derecha) y Outrunner (izquierda).

Los devanados del estator son alimentados con tensiones de manera que el imán permanente del rotor sigue los campos magnéticos creados por los devanados del estator.

Los Outrunner son construidos de forma que los imanes estén en la parte exterior y roten en torno al estator con las bobinas. De sus características destaca que tienen su torque máximo a baja velocidad, por lo que no necesitan reductoras y van directamente a la hélice y donde su conmutación se realiza con dispositivos externos de control. Producen mucho par motor y pueden ser utilizados con hélices grandes. Se usan en todo tipo de aplicaciones ya que se pueden construir para girar a cualquier velocidad.

Los Inrunner son construidos con imanes en la parte interior en el rotor y dispone de las bobinas en el estator ubicado en la parte exterior. No pueden tener una alimentación directa porque pueden producir un cortocircuito y

requiere un controlador externo para la conmutación. Por lo general produce par motor bajo y altas revoluciones, necesitando un reductor para sus distintas aplicaciones.

Los motores BLDC son más complicados que los motores tradicionales en su circuitería, ya que al eliminar las escobillas su conmutación es netamente electrónica y utiliza un sistema de control externo, generalmente un microcontrolador. Posee una prolongada vida útil al estar libre de desgaste mecánico, que es aproximadamente 20000 horas en carga máxima. Produce un giro suave, sin par-motor de retención y alcanza altas velocidades incluso a bajos voltajes donde fácilmente pueden alcanzar 50.000 rpm y 100.000 rpm en algunos casos dependiendo de varios factores. Aprovecha mejor la energía eléctrica al convertirla en energía mecánica dado que la potencia entregada tiene mejor eficiencia, aproximadamente un 90%, ya que al eliminar la interacción mecánica de rozamiento entre rotor y estator se pierde menos potencia en calor haciéndolo idóneo para aplicaciones alimentadas por baterías, o donde el consumo sea importante. También tiene una constante eléctrica de tiempo e inductancia reducida lo que presenta un mínimo de ruido eléctrico o interferencias eléctricas que podrían asumirse como inexistentes.

Los campos de aplicación industriales de los motores BLDC son en mayoría en entornos flamables, ya que al no poseer rozamiento de escobillas no

existe chispa; Salas limpias, ya que al no haber desgaste, no existen partículas en el aire. Además los BLDC se usan en cualquier otra aplicación que requiera velocidades de giro elevadas y larga vida en servicio.

### 2.3.1. Principio de funcionamiento del BLDC

En un motor de corriente continua con escobillas, se obtiene par motor gracias a la interacción del campo magnético inductor, estacionario, y la intensidad del arrollamiento inducido giratorio. Campo y corriente eléctrica se mantienen siempre en la misma posición relativa gracias al mecanismo de conmutación formado por el colector de delgas y las escobillas. En motores de pequeña potencia suele obtenerse la excitación mediante imanes permanentes. En este caso, solo se dispone de dos terminales para el control y la alimentación del motor. Las relaciones básicas electromecánicas son en este caso las siguientes:

$$T_m = K \cdot i \quad (2.1)$$

$$E = K \cdot \Omega \quad (2.2)$$

Siendo:

$T_m$ : Par motor;  $i$ : intensidad de inducido;  $E$ : tensión inducida;  $\Omega$ : velocidad angular.

El hecho de tener control directo sobre el par mediante la intensidad de inducido, y sobre la velocidad a través de la tensión, convierte a este motor en el modelo de referencia para la regulación de velocidad.

Este motor es similar al de corriente continua con escobillas, con las siguientes salvedades:

- a) La conmutación se realiza de forma electrónica en lugar de mecánica.
- b) Los imanes permanentes van alojados en el rotor en lugar de en el estator.
- c) Las bobinas van alojadas en el estator, constituyendo un devanado monofásico o polifásico.

Su funcionamiento se basa en la alimentación secuencial de cada una de las fases del estator de forma sincronizada con el movimiento del rotor. De esta forma, los imanes permanentes siguen el movimiento del campo magnético estático, cuyo desplazamiento depende a su vez del giro del rotor.

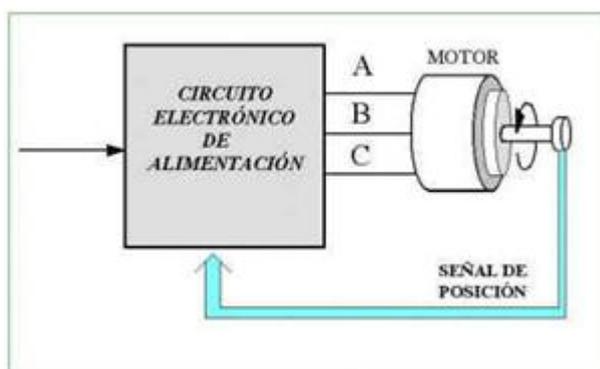


Figura 2.4: Esquema general de un motor BLDC [8]

La eliminación de las escobillas conlleva la necesidad de un circuito electrónico encargado de alimentar las distintas fases en función de la posición del eje y la de un sistema de sensores, tres por lo general, para detectar la posición del mismo. Estos sensores suelen ser del tipo Hall, sensibles al campo magnético, colocados en el devanado del estator y cerca de los imanes del rotor.

### 2.3.2. Alimentación del BLDC

La alimentación de los motores BLDC puede ser monofásica, bifásica y trifásica, siendo la última la más utilizada y de la cual se hablara en este trabajo. Al tener alimentación trifásica, cada bobina de las fases es separada en dos para producir la conmutación, que sigue un patrón determinado para efectuar un sentido de giro, sea este horario o anti horario. La figura 4 muestra un motor Inrunner donde la alimentación trifásica se encuentra en tipo estrella.

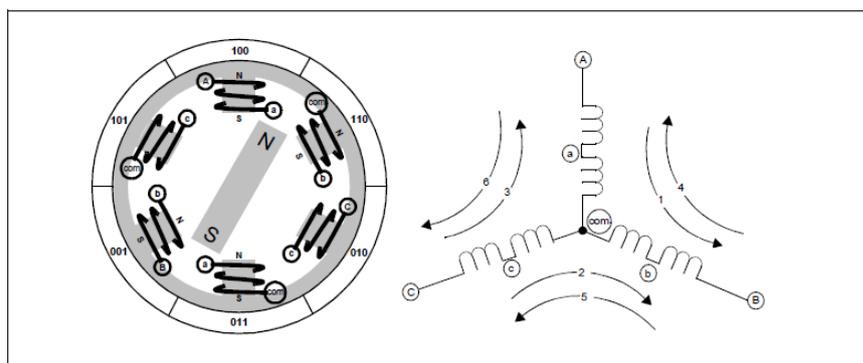


Figura 2.5: Motor Inrunner con alimentación trifásica tipo estrella [6]

Se puede apreciar en la parte derecha de la figura 2.5, que cada fase tiene una identificación las cuales son A, B y C, unidas por un punto común. Las fases son alimentadas de a dos a la vez con un voltaje determinado para el paso de corriente en un sentido, mientras la tercera es flotante. Los sentidos del recorrido de la corriente se expresan en modo de flechas indicando el sentido y el orden del 1 al 6 para un giro en sentido horario. Para el sentido anti horario, se genera una conmutación entre fases distinta, y obviamente las flechas del 1 al 6 cambiarían su sentido.

En la parte izquierda de la figura 2.5, se aprecia un motor BLDC con las bobinas en el estator y el imán permanente en el rotor, donde también se observan códigos binarios de tres dígitos, los cuales nos dicen cual es el estado del sensor de efecto Hall necesario para que el imán permanente se encuentre alineado con dicha bobina.

Dentro del estator se encuentran tres sensores Hall, uno por cada fase, que detecta si las bobinas de dicha fase se encuentran alimentadas por corriente, permitiéndonos saber la posición del rotor en todo momento y ayudándonos al giro en un sentido del rotor.

### **2.3.3. Conmutación del BLDC**

Para el control de conmutación del rotor existen dos tipos de señales, las de f.e.m. sinusoidal y las de f.e.m. trapezoidal. Los motores de f.e.m. sinusoidal han de ser alimentados con un sistema de tensiones e intensidades también

sinusoidales, y sincronizadas en todo momento con la f.e.m. inducida. El control de estos motores es complejo y se recurre a técnicas similares a las empleadas en los motores asíncronos, incluidas las técnicas de control vectorial. En cambio, los motores con f.e.m. trapezoidal envían señales donde dos fases tienen un voltaje determinado mientras la tercera es flotante. En la figura 2.6 se muestra la configuración más empleada de la etapa de potencia que compone de seis transistores de potencia MOSFETs o IGBTs, dependiendo de la tensión de alimentación. Para la regulación de velocidad se emplea la técnica PWM con portadora de alta frecuencia.

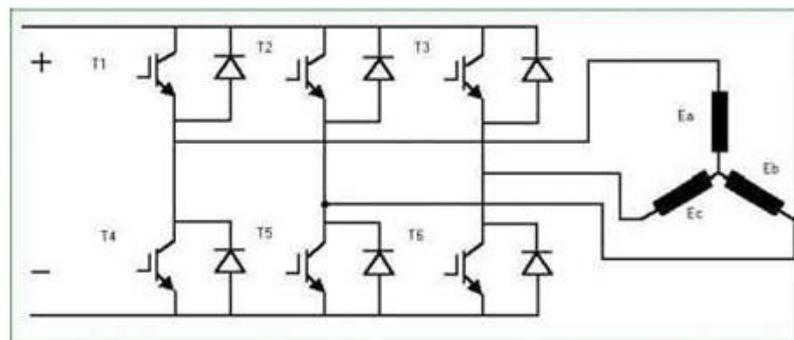


Figura 2.6: motor de F.E.M. trapezoidal [8]

La figura 2.7 muestra las ondas de tensión y de intensidad correspondiente a una fase para un motor de este tipo.

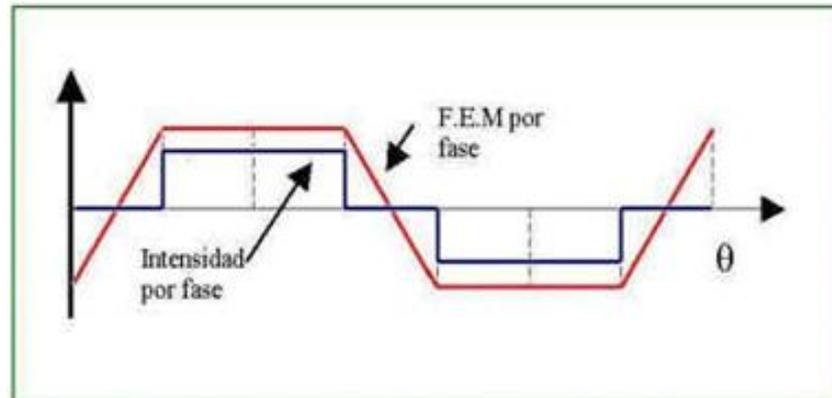


Figura 2.7: Forma básica de intensidad y tensión de un BLDC [8]

En la figura 2.8 se muestran los niveles de voltaje trapezoidales para cada fase en forma individual y las lecturas de los sensores de efecto Hall presentes en el estator para cada estado del rotor.

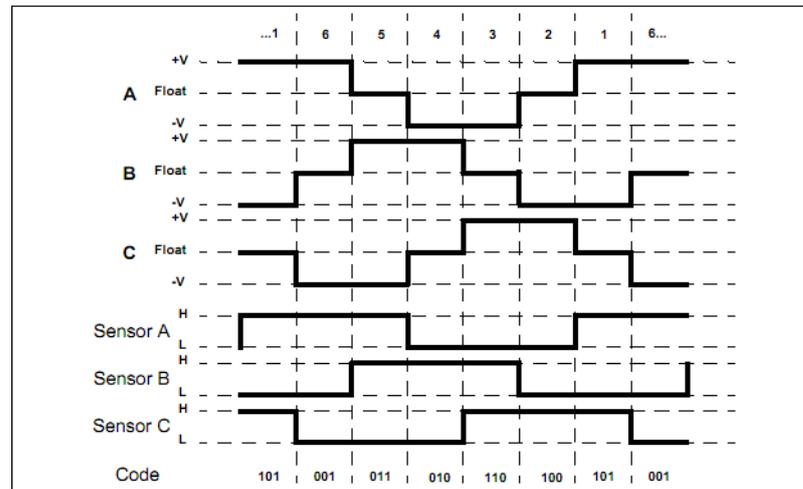


Figura 2.8: Niveles de Voltaje para cada fase y niveles lógicos de Sensores Hall versus tiempo de manejo para un giro a favor del reloj. [6]

Dependiendo de la lectura del efecto Hall, el microcontrolador sabrá cual es el siguiente estado de conmutación, previendo si el rotor ha cambiado o no

de posición, habilitando los voltajes y dejando flotante las fases del estado siguiente. Se puede observar también que existen seis diferentes estados de posición del rotor.

#### **2.3.4. Comparación entre motores con escobillas y sin escobillas**

Los motores BLDC tienen varias ventajas respecto a los motores con escobillas y motores de inducción. Algunas de estas son:

- Mejor relación velocidad-par motor
- Mayor respuesta dinámica
- Mayor eficiencia
- Mayor vida útil
- Menor ruido e interferencia eléctrica
- Mayor rango de velocidad

Además, la relación par motor- tamaño es mucho mayor, lo que implica que se pueden emplear en aplicaciones donde se trabaje en espacios reducidos.

Por otra parte, los motores BLDC tienen dos desventajas:

- Mayor coste de fabricación
- Sistema de control complejo.

En la tabla 2.1, se explica detenidamente las diferencias en distintos aspectos entre el motor sin escobillas y el motor con escobillas.

	<b>Motor BLDC</b>	<b>Motor con escobillas</b>
Commutación	Commutación electrónica basada en sensores de posición de efecto Hall	Commutación por escobillas
Mantenimiento	Mínimo	Periódico
Durabilidad	Mayor	Menor
Curva Velocidad / par	Plana. Operación a todas las velocidades con la carga definida	Moderada. A altas velocidades la fricción de las escobillas se incrementa, reduciendo el par.
Eficiencia	Alta. Sin caída de tensión por las escobillas.	Moderada
Potencia de salida / Tamaño	Alta. Menor tamaño debido a mejores características térmicas porque los bobinados están en el estator, que al estar en la carcasa tiene una mejor disipación de calor.	Baja. El calor producido en la armadura es disipado en el interior aumentando la temperatura y limitando las características.
Inercia del rotor	Baja. Debido a los imanes permanentes en el rotor	Alta. Limita las características dinámicas.
Rango de velocidad	Alto. Sin limitaciones mecánicas impuestas por escobillas/commutador.	Bajo. El límite lo imponen principalmente las escobillas.
Ruido eléctrico generado	Bajo	Arcos en las escobillas
Coste de construcción	Alto. Debido a los imanes permanentes	Bajo.
Control	Complejo y caro	Simple y barato.
Requisitos de control	Un controlador es requerido siempre para mantener el motor funcionando. El mismo puede usarse para variar la velocidad.	No se requiere control si no se requiere una variación de velocidad.

**Tabla 2.1: Comparación entre motores Brushless y de escobilla. [9]**

### 2.3.5. Sensores utilizados en motores BLDC

#### 2.3.5.1. Encoder

El encoder va acoplado al eje trasero del motor. En algunas ocasiones se usa un encoder adicional en la carga para posicionamiento muy preciso evitando las holguras en la transmisión entre el servomotor y la carga. Normalmente, la señal de salida del encoder es una onda cuadrada digital tipo TTL (0 V = 0 y 5 V = 1), la cual se procesa para la cuenta de pulsos (velocidad o posicionamiento de precisión).

El desfase de 90° entre las señales del canal A y B permite determinar el sentido de giro del servomotor. El canal Index se utiliza para tareas de búsqueda de cero (home, homing) en posicionamiento, al iniciar la máquina.

El encoder con line driver genera señales complementarias en cada canal para eliminar posibles interferencias eléctricas que reciban los cables. Dependiendo del entorno de las interferencias eléctricas, se pueden transmitir las señales a más de 30 metros sin cable apantallado. Hoy día, es el tipo más utilizado en la industria.

El encoder más usado habitualmente es el encoder tipo óptico incremental de 500 pulsos/vuelta, existiendo otras tecnologías como los encoder magnéticos. Se recomiendan los encoder magnéticos en lugares donde exista mucha polución ambiental.

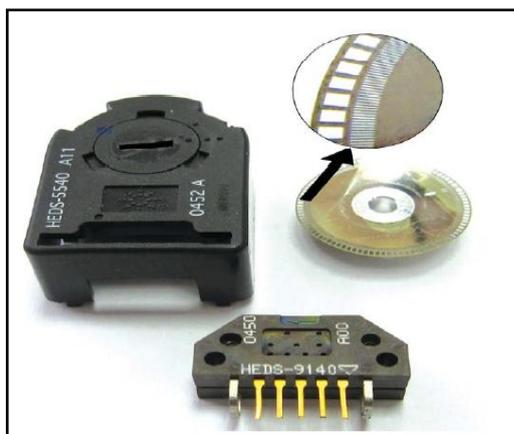


Figura 2.9: Piezas de un encoder óptico incremental de 500 pulsos por vuelta y tres canales. [10]

### 2.3.5.2. Sensor de Efecto Hall

*Efecto Hall:* Cuando fluye una corriente a través de un sensor Hall y este se aproxima a un campo magnético perpendicular, entonces se crea un voltaje

saliente proporcional al producto de la fuerza del campo magnético y de la corriente.

Gracias a este principio, mediante un disco magnético acoplado al eje del servomotor podemos sensor la posición del rotor.

Principalmente, estos sensores se usan para que la electrónica pueda conmutar las tres bobinas del motor de acuerdo a la posición de los polos del imán del rotor.

Así por ejemplo un motor brushless de dos polos con 3 sensores hall (a 120 °), tiene una resolución en posición de 6 pulsos por vuelta (60 ° de conmutación). En caso de los servomotores multipolares esta resolución aumenta.

Ocasionalmente, se pueden accionar los motores brushless sin sensores Hall para determinadas aplicaciones muy sencillas, como por ejemplo ventiladores y bombas.

Sin sensores Hall, el problema es que el arranque del servomotor es un poco brusco.

## **2.4. Comunicación Serial**

Existen dos formas de transmitir información binaria: Paralela y Serial.

La comunicación paralela transmite todos los bits de un dato de manera simultánea, por lo tanto la velocidad de transferencia es rápida, sin embargo tiene la desventaja de utilizar una gran cantidad de líneas, por lo tanto se vuelve más costoso y se atenúa a grandes distancias por la capacitancia entre conductores.

La comunicación serial consiste en el envío de un bit de información de manera secuencial, esto es, un bit a la vez.

Si se opta por una comunicación serie en lugar de paralela entonces se puede ahorrar un gran número de señales en el bus.

Existen dos tipos de comunicaciones seriales: la síncrona y asíncrona.

En la comunicación serial síncrona además de una línea sobre la cual se transmitirán los datos se necesita de una línea la cual contendrá los pulsos de reloj que indicarán cuando un dato es válido. Ejemplos de este tipo de comunicación son: I2C, SPI.

En la comunicación serial asíncrona, no son necesarios los pulsos de reloj. La duración de cada bit está determinada por la velocidad con la cual se realiza la transferencia de datos.

## 2.5. Comunicación I2C

La abreviatura I2C significa Inter Integrated Circuit, protocolo de transmisión cuyo campo de aplicación principal es la comunicación entre circuitos integrados.

El bus I2C fue desarrollado por Philips al inicio de la década de 1980, teniendo en mente todos estos aspectos, y ha encontrado una gran aceptación en el mercado.

### 2.5.1. Características del bus I2C:

Bus serie de dos hilos: reloj (SCL) y datos (SDA).

**SDA:** Es la línea de datos serie (Serial Data, en inglés). Eléctricamente se trata de una señal a colector o drenador abierto. Es gobernada por el emisor, sea éste un maestro o un esclavo.

**SCL:** Es la señal de sincronía (reloj serie, o Serial Clock en inglés). Eléctricamente se trata de una señal a colector o drenador abierto. En un esclavo se trata de una entrada, mientras que en un maestro es una salida. Un maestro, además de generar la señal de sincronía suele tener la capacidad de evaluar su estado. Esta señal es gobernada única y exclusivamente por el maestro; un esclavo sólo puede retenerla o pisarla para forzar al maestro a ralentizar su funcionamiento.

**Bus Half Duplex:** Es decir, existirá una única línea de datos que podrá utilizarse para el flujo en ambos sentidos, pero no simultáneamente. De esta manera se ahorra el número de señales del bus. Un mismo dispositivo podrá actuar como emisor o como receptor, en distintos momentos.

**Bus sincrónico:** El término síncrono deberá entenderse como que existirá, además de la línea de datos, una señal de sincronía explícita para validar los datos en el canal serie.

- Multimaestro.
- No necesita línea de selección.
- Tiene tres modos de velocidad compatibles en forma descendente:
  - Lenta, por debajo de los 100 Kbps
  - Rápida, 400 Kbps
  - Alta Velocidad, 3,4 Mbps

Deberá establecer un mecanismo de adaptación de velocidad, para el caso en que un esclavo no pueda seguir la velocidad impuesta por el maestro.

El bus tiene que ser controlado por un dispositivo amo que genere la señal de reloj SCL, controle el acceso al bus y genere las condiciones de inicio y parada del dispositivo esclavo.

El dispositivo que envía los datos al bus se define como transmisor y el dispositivo que recibe los datos como receptor. Ambos, amo y esclavo, pueden operar como transmisores o receptores pero el dispositivo maestro es quien determina el modo que esté activado.

Lo que diferenciará a un maestro de un esclavo es la capacidad de gobierno del bus (suministro de la señal de sincronía).

El Comité I2C es el encargado, entre otras cuestiones, de asignar las direcciones I2C a cada tipo de circuito. De esta manera cada función implementada, independientemente del fabricante, posee la misma dirección; es decir, circuitos que realicen funciones equivalentes deberán poseer la misma dirección oficial I2C independientemente del fabricante.

### 2.5.2. Estados del bus I2C

El bus I2C puede encontrarse en los siguientes estados:

Libre. Este estado se caracteriza por encontrarse las líneas SDA y SCL a 1.

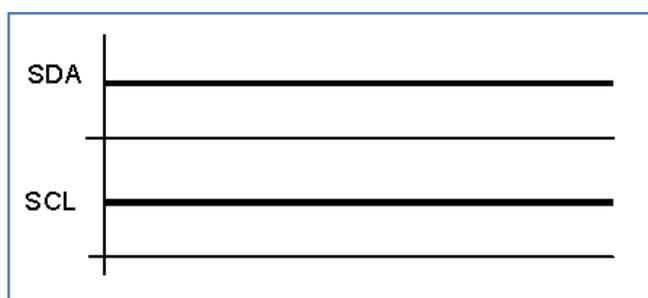


Figura 2.10: Condición del bus libre [11]

Inicio. Cuando un maestro inicia una transacción. Consiste en un cambio de alto a bajo en la línea SDA mientras SCL permanece a alta. Esto puede apreciarse en el en la figura 2.11. A partir de que se dé una condición de inicio se considerará que el bus está ocupado y ningún otro maestro deberá intentar generar su condición de inicio.

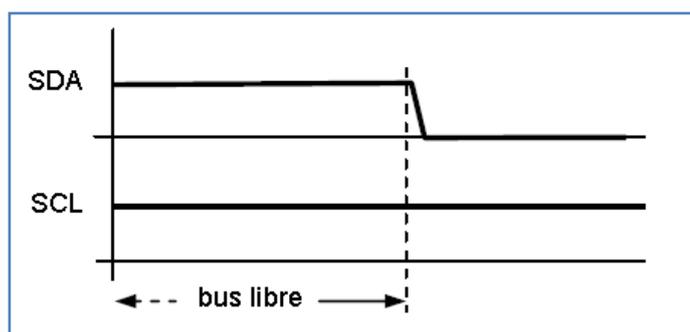


Figura 2.11: Condición de inicio [11]

Cambio. Cuando, la línea SCL estando en bajo, la línea SDA cambia de estado. En la transferencia de datos por un bus I2C éste es el único instante en el que el sistema emisor (que podrá ser tanto un maestro como un esclavo) podrá poner en la línea SDA cada bit a transmitir.

Dato. Este estado es el que, una vez iniciada una transacción, queda definido por la fase alta de la señal de sincronía SCL. En este estado se considera que el dato emitido es válido, y no se admite que pueda cambiar.

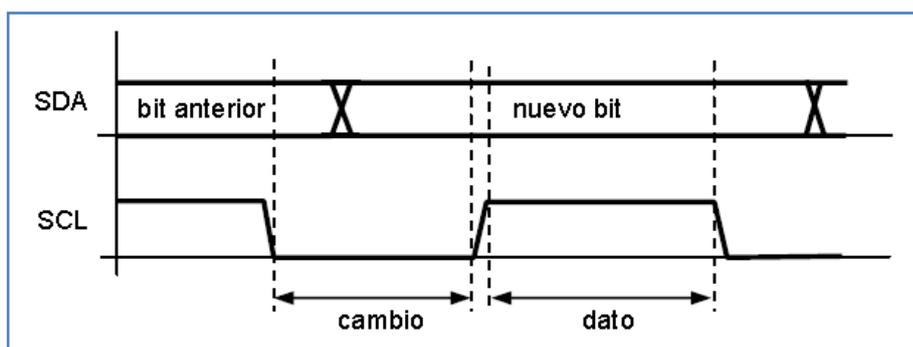


Figura 2.12: Condiciones de cambio y datos [11]

Parada. O condición de fin, cuando la línea SCL está en alto, se produce un cambio de bajo a alto en la línea SDA. Obsérvese que esto es una violación de la condición de dato, y es precisamente por esto por lo que se utiliza para que un maestro pueda indicar al esclavo que se finaliza la transferencia.

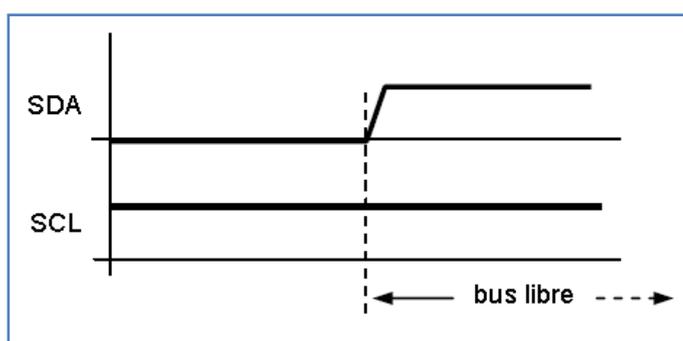


Figura 2.13: Condición de parada [11]

### 2.5.3. Formato de la transmisión

Luego de que se produzca el inicio de una transmisión se lleva a cabo la transferencia de uno o varios caracteres, en uno u otro sentido.

El carácter de ocho bits es la unidad de transferencia.

El orden de emisión de los bits de un carácter empieza por el más significativo, y termina por el menos significativo (contrario al que siguen las transmisiones UART).

El primer carácter lo transmite siempre el maestro; sus siete bits más significativos indican la dirección del esclavo al que se dirige, y el bit de menor peso indica el sentido de la transferencia de los subsiguientes

caracteres (0=escritura, 1=lectura, siempre desde el punto de vista del maestro).

Los circuitos EEPROM tienen asignada una dirección 1010XXX; 1010 es la parte fija de los siete bits, y XXX es la parte programable.

Ahora existen dos tipos direcciones: las normales de siete bits y las ampliadas de diez bits. En este último caso el direccionamiento de un esclavo implica la transferencia de dos caracteres; el primero deberá comenzar por 11110XX, siendo XX los dos bits de mayor peso de la dirección de diez bits, y el segundo caracter corresponderá a los ocho bits inferiores de la dirección del esclavo. El sentido de la transacción se sigue indicando por el bit menos significativo del primer caracter de dirección (11110-XX-L/E).

#### **2.5.3.1. Gobierno de la señal de sincronía**

Ya se ha indicado que la señal SCL sólo la puede generar el maestro. Si el maestro actúa como emisor entonces la señal de sincronía puede entenderse como una validación de los bits que va volcando en la línea de datos SDA, sean estos bits los de un caracter de dirección o los de un caracter propiamente de datos. Si actúa como receptor entonces SCL sirve no sólo para indicar los instantes en los que el maestro toma los bits que le envía el esclavo, sino también para – indirectamente – marcar la velocidad de envío del siguiente bit.

Sólo hay una situación en la que un esclavo puede interferir la señal SCL gobernada por el maestro. Se trata del mecanismo denominado sincronización, por medio del cual un esclavo puede ralentizar la velocidad de transferencia marcada por el maestro. En este caso el esclavo no gobierna el reloj pero interfiere con cierto criterio su estado para que el maestro advierta que se le está pidiendo que vaya más lento.

### **2.5.3.2. Mecanismo de sincronización**

Se trata de un procedimiento ingenioso que saca provecho de la naturaleza a colector o drenador abierto de la señal de reloj SCL.

Por defecto, un maestro gobernará la señal de reloj del bus, SCL, a la velocidad que considere oportuno.

El maestro marcha a su propia velocidad y supondrá que el esclavo con el que se comunique será capaz de seguir el ritmo. En muchas circunstancias esto es así, pero en otras puede resultar que no, bien porque el esclavo de ninguna manera pueda seguir jamás tal velocidad marcada por el maestro, o bien porque en ciertos instantes necesite realizar algunos procesos que le consuman un tiempo que no se pueda dedicar a la comunicación con el maestro.

El mecanismo consiste en lo siguiente: un maestro que admita la sincronización deberá comprobar el estado de la línea SCL del bus; si al

activarla ésta no se activa efectivamente en el bus, entonces querrá significar que el esclavo le está solicitando ralentizar su velocidad. De esta manera, el maestro esperará con su SCL local activada y no considerará que haya comenzado el estado de dato hasta que efectivamente la línea SCL del bus esté a alta. Por la otra parte, todo esclavo que admita sincronización, estando la línea SCL del bus a baja podrá pisarla (poner a baja su SCL local) durante el tiempo que estime oportuno, y sólo liberará la línea SCL del bus en el instante en que esté preparado para gestionar un nuevo bit o caracter.

### **2.5.3.3. Filosofía del reconocimiento**

El reconocimiento es obligatorio para todo esclavo receptor. Si uno no lo hace entonces el maestro emisor considerará que se ha producido un fallo en el bus y deberá establecer los mecanismos oportunos de respuesta a este fallo. Por ejemplo:

Si la ausencia de reconocimiento del esclavo receptor se produce en la fase de direccionamiento esto podrá significar varias cosas:

Que el circuito se encuentra ausente (removido del bus, o apagado).

Que el circuito está ocupado realizando algún tipo de tarea interna que le impide responder al direccionamiento del maestro.

Que el circuito está averiado o ha sucedido alguna anomalía en la línea.

Pero si esta ausencia se produce en medio de una transferencia de datos entonces esto normalmente será síntoma de una avería funcional (a menos que dé la casualidad de que en ese instante ha sido removido o apagado).

Aunque se ha dicho que el reconocimiento es obligatorio por parte del receptor, existe una excepción a esta regla: cuando el receptor es un maestro entonces es posible no dar el reconocimiento al último carácter, justo antes de generar la condición de parada. Para entender el motivo de esto es necesario recordar que es el maestro quien inicia una transacción, la gobierna y la finaliza. Por tanto, aunque un maestro actúe como receptor sabrá perfectamente cuándo no quedan más datos por recibir.

#### **2.5.3.4. Caracteres de dirección**

El primer carácter que se transmite tras una condición de inicio es la dirección del esclavo. La dirección ya se sabe que es un identificador de siete bits que representa un determinado circuito del bus I2C. El bit menos significativo del carácter de ocho bits, indica el sentido de la transferencia de los siguientes caracteres.

Comentaremos el significado de aquellas combinaciones que no son direcciones normales. En la siguiente tabla se resume lo explicado.

CARACTER		FUNCIÓN
DIRECCIÓN	L/E	
0000000	0	Llamada general ( <i>General Call</i> ) que va dirigida a todos los esclavos conectados al bus I2C.
0000000	1	Aviso de inicio. Se usa para que los esclavos que implementan la interfaz por programa lento puedan detectar la condición de inicio
0000001	X	Dirección CBUS. En topologías en las que se mezclen circuitos I2C y circuitos CBUS, los I2C deben ignorar este direccionamiento
0000010	X	Reservado para formatos diferentes de bus. Los circuitos I2C que no los admitan deberán ignorar este direccionamiento
0000011	X	Reservado para futuras ampliaciones
00001XX	X	Petición de transacción a alta velocidad ( <i>Hs</i> )
00010XXa111 0XX	X	Direcciones normales asignables a circuitos I2C
11110XX	X	Direccionamiento de diez bits
11111XX	X	Reservado para futuras ampliaciones

Tabla 2.2: Significado del primer bit de la transacción [11]

### **2.5.3.5. Llamado General**

La dirección de llamada general se ha implementado para permitir que ciertos mensajes que un maestro necesite enviar a todos los elementos conectados al bus se haga de manera colectiva en lugar de individual direccionándolos uno tras otro. Esta cuestión podrá tener sentido en algunos casos pero no en otros. Aquellos circuitos que no tienen nada que hacer ante una llamada general simplemente la ignorarán.

Una llamada general consta de una trama formada por el caracter de la llamada general, el reconocimiento, caracter de tipo de llamada general, y su reconocimiento. El segundo caracter indica qué tipo de llamada general se está realizando.

### **2.5.3.6. Caracter de Inicio**

El valor 00000001 (dirección 0, en modo lectura) es un caracter especial denominado de inicio que se utiliza al comenzarse una transacción en una topología I2C en la que se sabe que existen esclavos cuya interfaz I2C está implementada fundamentalmente por programa. Efectivamente, esto es muy frecuente cuando alguno de los dispositivos es un microcontrolador que no dispone de un controlador de bus I2C; en este caso se pueden utilizar líneas de E/S de propósito general para implantar las líneas del bus I2C y gestionar por programa todo el protocolo del bus. Si se observa bien, un esclavo de este tipo ha de estar sondeando constantemente por programa el estado de

las líneas SDA y SCL simplemente para poder detectar una condición de inicio y, en caso afirmativo, ver si se dirigen a él.

Pues bien, para permitir que este tipo de esclavos que implementan el protocolo I2C por programa puedan conseguir un rendimiento superior en otras tareas es por lo que la norma I2C contempla un modo especial de inicio de una transacción mediante el primer envío de un carácter de inicio.

El maestro que ha iniciado la transacción con el carácter de inicio deberá generar tras él el impulso SCL para el reconocimiento, pero no considerará como erróneo el que no se reciba; es más, ningún esclavo debe reconocer un carácter de inicio. A continuación generará una condición de reinicio y comenzará la transacción con el formato normal que ya se conoce.

## 2.6. Herramientas de Diseño

### 2.6.1. AVR Studio

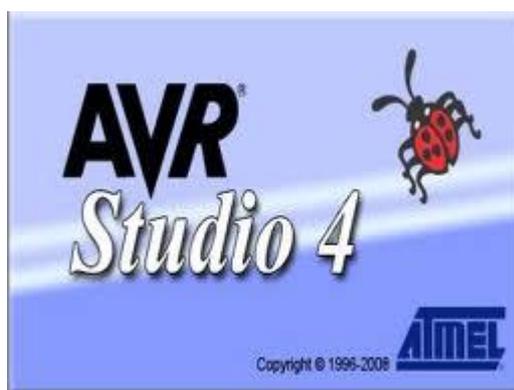
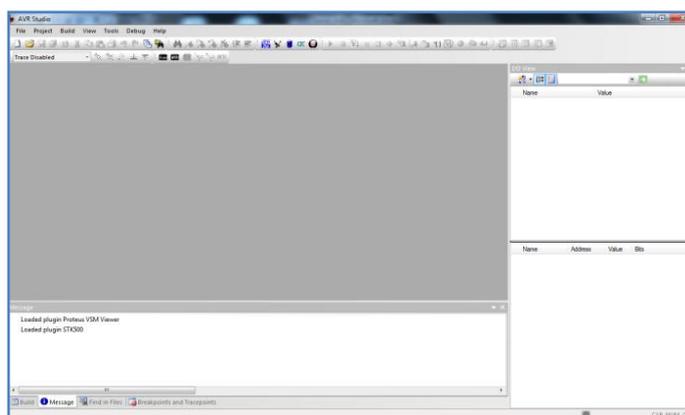


Figura 2.14: Presentación inicial del AVR Studio 4

AVR-Studio es un compilador desarrollado por ATMEL. Incluye herramientas de compilación en ensamblador, y trabaja en conjunto con WinAVR para proveer el compilador de C/C++, AVR-GCC. Sin embargo, la última versión incluye soporte directo del compilador AVR-GCC, sin necesidad de instalar adicionalmente WinAVR. AVR Studio soporta TODOS los microcontroladores de ATMEL de la familia AVR, así como las herramientas hardware que ellos ofrecen (programadores, depuradores, tarjetas de desarrollo, etc). AVR Studio es además completamente gratuito y no impone restricciones en cuanto a tamaño de código generado.



**Figura 2.15: Interfaz inicial del AVR Studio 4**

Al igual el AVR GCC nos ofrece las mismas características que el AVR ASSEMBLER en cuestión de selección de plataforma y de integrados pero la diferencia entre ambas son los archivos que se generan después de la compilación.

Los archivos que se generan en la creación de un código en assembler son (.asm) y en C el archivo que se crea en un (.c) los cuales son los que guardan la programación que se va a colocar en el microcontroladores.

## 2.6.2. LPCXpresso

LPCXpresso es una plataforma de la compañía NXP de bajo costo de desarrollo. El software consiste en un IDE basado en Eclipse, un compilador de lenguaje C de GNU, un enlazador, librerías con múltiples aplicaciones, y un depurador de GDB.

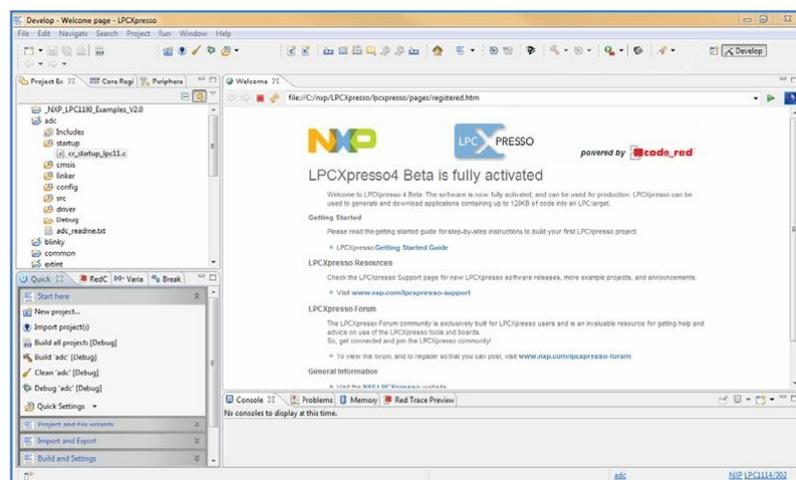


Figura 2.16: Interfaz gráfica de LPCXpresso 4

El software LPCXpresso permite al usuario desarrollar sus aplicaciones desde la evaluación inicial hasta la producción final. El IDE se basa en la plataforma de desarrollo Eclipse e incluye varias mejoras específicas de LPC. Se trata de un estándar de la industria con un conjunto

de herramientas GNU con una optimizada biblioteca de C que provee al usuario de todas las herramientas necesarias para desarrollar soluciones de alta calidad de software de forma rápida y rentable. El entorno de programación C incluye características de nivel profesional, como por ejemplo no tiene coloreado de sintaxis, formato de origen, función de plegado, la ayuda en línea o en el mismo entorno, y una amplia automatización de gestión de proyectos.

La placa de destino LPCXpresso que trabaja con el software, en nuestro caso específico LPCXpresso 1769, incluye un depurador JTAG integrado (LPC-Link), así que no hay necesidad de usar un depurador por separado.

### 2.6.3. Proteus

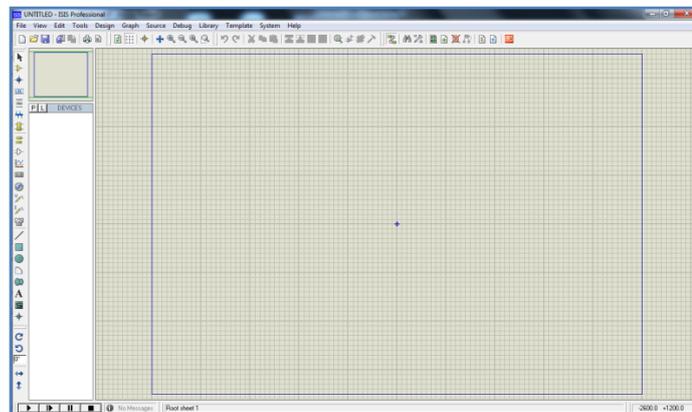


Figura 2.17: Interfaz gráfica del Proteus

PROTEUS es la herramienta de software de ISIS, que permite la simulación de circuitos electrónicos variados, entre ellos los que contienen

microcontroladores. Sus reconocidas prestaciones lo han llevado a ser el más popular simulador en software para microcontroladores.

Esta herramienta permite la simulación de circuitos electrónicos complejos, integrando inclusive desarrollos realizados con microcontroladores de varios tipos en una herramienta de alto desempeño con impresionantes capacidades gráficas.

Presenta una filosofía de trabajo semejante al SPICE de ORCAD, arrastrando y soltando componentes desde barras de herramientas e incrustándolos en el modelo a simular, siendo muy sencillo de manejar y además de brindar una interfaz gráfica amigable al usuario en manejo de las herramientas.

## 2.7. Descripción del Hardware

### 2.7.1. AVR Butterfly



Figura 2.18: Vista AVR Butterfly [14]

El módulo AVR está diseñado para demostrar beneficios y aplicaciones de los microcontroladores de AVR, por medio del uso de numerosas aplicaciones.

#### **2.7.1.1. Características**

- Tiene un diseño de bajo poder
- Usa un microcontrolador ATmega 169
- Sus periféricos son:
  - Controlador LCD
  - Memorias Flash, EEPROM, SRAM y capacidad de una memoria externa flash.
  - Modos de programación: Auto programación/Bootloader, SPI, Paralelo y JTAG.
  - Interfaces de comunicación: SPI, UART, USI.
  - Convertidor Analógico-Digital (ADC)
  - Timers/Counters:
    - Real Time Clock (RTC)
    - Pulse Width Modulation (PWM)
- Sus elementos:
  - ATmega 169 (encapsulado MLF)
  - LCD en vidrio con 120 segmentos.
  - Joystick de 4 direcciones y pulsado central

- Elementos piezoeléctricos para emitir sonidos.
- Un LDR para medir intensidades lumínicas.
- Emulación JTAG
- Interfaz USI para comunicación

### 2.7.2. LPCXpresso LPC1769

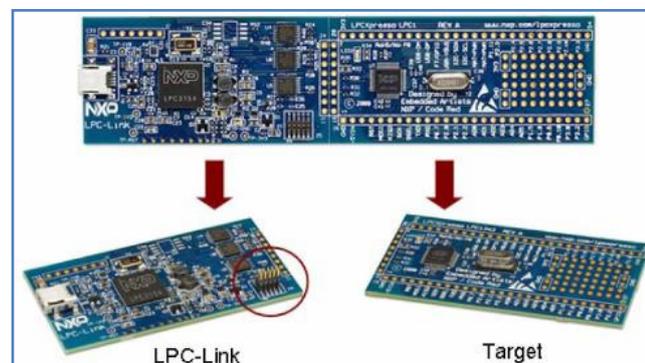


Figura 2.19: Tarjeta LPC1769 [12]

La tarjeta LPC1769 utiliza un ARM Cortex M3 basado en micro controladores para sistemas embebidos con un alto nivel de integración. El ARM Cortex M3 es la siguiente generación de procesadores usados en un sinnúmero de aplicaciones que soportan un alto nivel de integración por soporte de bloques.

Las versiones de alta velocidad como la LPC1769 usada en este proyecto operan a frecuencias superiores a 120 MHz mientras otras operan a 100 MHz. El Cortex-M3 CPU trae incorporado una línea de comandos de tres

estados y usa una arquitectura Harvard con separación de instrucciones locales y buses de comunicación internas y externas.

### **2.7.2.1. Características principales**

Los complementos periféricos de la LPC1769 incluyen:

- Memoria Flash de 512 kB
- Memoria de datos de 64 kB
- Interfaz de Ethernet
- Interfaz USB
- 4 interfaces UART
- 3 interfaces SPI
- 3 interfaces I2C
- 8 canales de ADC de 12 bits
- 6 GPIO para PWM
- 4 timers de propósito general.
- 70 GPIO para aplicaciones.
- Puertos de Vcc y GND.

Esta tarjeta cuenta con una división de hardware para las aplicaciones (Target) donde se encuentra la parte de programación embebida, y otra parte encargada de la comunicación entre la tarjeta y el software IDE para la programación de la tarjeta (LPC Link).

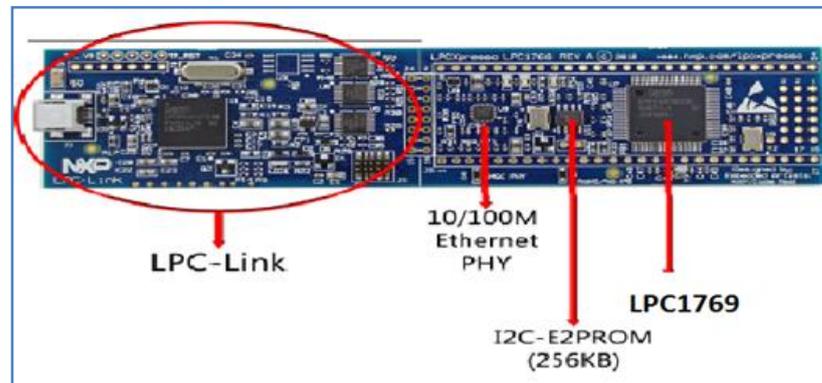


Figura 2.20: Descripción de partes de la LPC 1769 [12]

### 2.7.3. LPCXpresso LPC 1114



Figura 2.21: Tarjeta LPC 1114

La tarjeta de trabajo LPC1114 cuenta con un procesador basado en ARM Cortex-M0, de bajo costo de 32-bit, diseñado para aplicaciones de microcontroladores de 8/16-bit, que ofrece buen rendimiento a bajo consumo de potencia. Cuenta con un conjunto de instrucciones simples y fácil direccionamiento de memoria, junto con la facilidad de código RISC a

diferencia de otras con microcontroladores de la misma capacidad. Esta tarjeta opera en frecuencias de CPU de hasta 50MHz. Entre los complementos periféricos de laLPC1114 incluyen:

- Un máximo de 32kBde memoria flash de memoria.
- Hasta 8 Kb de memoria de datos.
- Modo de interfaz I2C
- Conexión para RS-485/EIA-485 UART
- 2 interfaces SPI con las características del SSP
- Cuatro contadores / temporizadores de propósito general
- ADC de 10 bits
- 42 entradas o salidas GIOP.



Figura 2.22: Vista de circuito LPC 1114 [15]

#### 2.7.4. LPCXpresso Motor control kit



Figura 2.23: Vista LPCXpresso Motor Control Board [15]

El Motor Control Board de NXP es una plataforma universal para el control de bajo voltaje de motores basados en microcontroladores de la misma compañía. Con esta plataforma es posible controlar motores BLDC, BLAC, motores de paso y motores de corriente continua para acoplarlos a sistemas embebidos programados en tarjetas de trabajo de NXP.

El control de motores de LPCXpresso tiene una estructura tal como se indica en la figura 2.24.

- El lado derecho contiene la electrónica de potencia para la conducción de las fases del motor.
- El lado izquierdo es el lado de control con zócalos para tarjetas de LPCXpresso diferentes, así como una toma de control del procesador

PLCC434. También contiene un OLED y una palanca de mando que puede servir como una interfaz de usuario para el sistema.

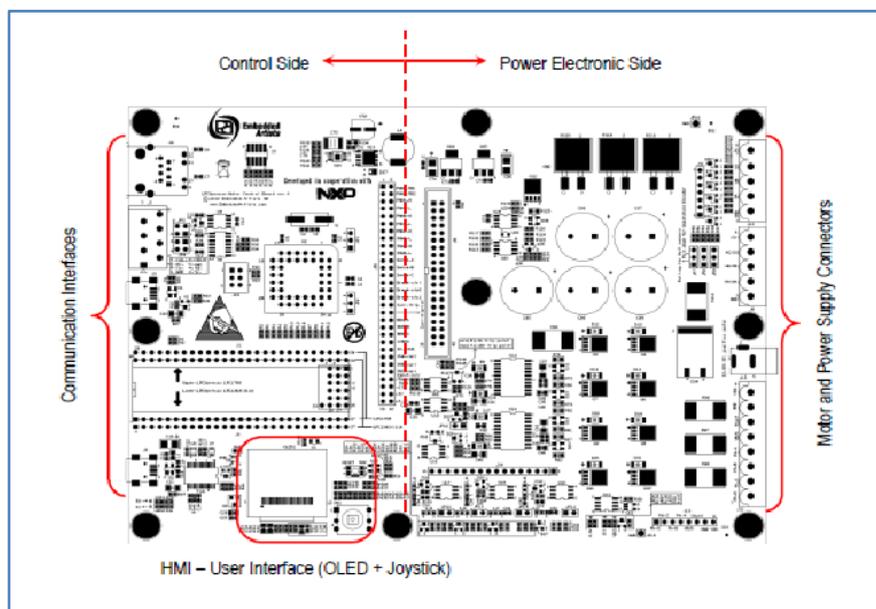


Figura 2.24: Distribución de tarjeta de Control de Motores de NXP [15]

## 2.8. Introducción a los Microcontroladores

Un microcontrolador es un chip o circuito integrado que contiene todos los elementos de una CPU (Procesador, RAM, ROM, E/S). Estos dispositivos nacieron a finales de la década del 70' para brindar una solución a los caros y complejos sistemas basados en lógica discreta.

En el presente trabajo se mencionan principalmente los dispositivos que pertenecen a la familia AVR de la empresa ATMEL, microcontroladores muy

utilizados dentro del ámbito aficionado y también dentro de la industria. Este tipo de dispositivos son del tipo RISC (conjunto de instrucciones reducido). Lo cual quiere decir que su conjunto de instrucciones es muy reducido en el orden de 30 a 200 instrucciones que puede ejecutar, salvo alguna de ellas, en el orden de 1 ciclo de máquina.

### **2.8.1. Estructura General de un microcontrolador**

Para definir la estructura general de un microcontrolador, es primero necesario definir que es un microcontrolador.

Básicamente un 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. Un microcontrolador incluye en su interior las tres principales unidades funcionales de una computadora: unidad central de procesamiento, memoria y periféricos de entrada/salida.

A continuación observamos los componentes básicos encontrados en cualquier microcontrolador por medio de un diagrama de bloques. Los componentes que conforman cada dispositivo suelen variar según el fabricante y la arquitectura que posea.

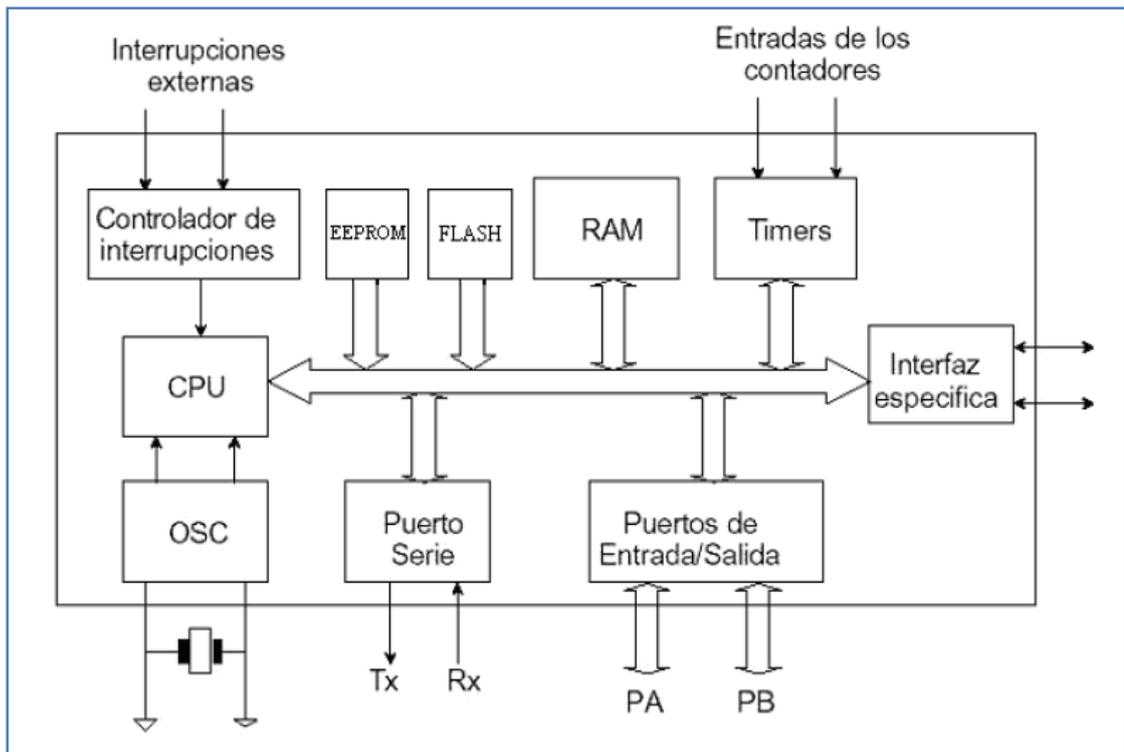


Figura 2.25: Estructura básica de un microcontrolador.

## 2.9. MCU AVR ATmega169

### 2.9.1. Características Generales

Este microcontrolador AVR cuenta con 8 bits de alto rendimiento y bajo consumo dentro de una arquitectura Avanzada RISC. Puede manejar 130 instrucciones, las cuales son de mayoría de un solo ciclo de reloj de ejecución. Además cuenta con:

- Contiene 32 registros de trabajo de 8 bits de propósito general con un funcionamiento estático total.

- Su capacidad de procesamiento de unos 16 MIPS a 16 MHz.
- Multiplicador por hardware de 2 ciclos y
- Segmentos de memoria y de datos no volátiles de alta duración.
- 16Kbytes de FLASH auto programable en sistema
- 512 bytes de EEPROM
- 1Kbyte de SRAM Interna
- Ciclos de escritura/borrado: 10.000 en Flash y 100.000 en EEPROM
- Retención de Datos: 20 años a 85°C / 100 años a 25°C
- Sección opcional de código Boot con bits de bloqueo independientes.
  - Programación en el sistema. A través del programa de Arranque presente en el chip.
  - Operación Real de lectura durante la escritura.
- Bloqueo programable para seguridad del software.
- Interfaz JTAG (conforme el Standard IEEE 1149.1)
- Exploración de acuerdo a Capacidad acorde al estándar JTAG
- Soporte Extendido para depuración dentro del chip
- Programación de FLASH, EEPROM, fusibles y bits de bloqueo a través de la interfaz JTAG.

### **Características de los periféricos**

- Driver LCD de 4 x 25

- Dos Temporizadores/Contadores de 8 bits con preescalamiento separado y modo de comparación.
- Un Temporizador/Contador de 16 bits con preescalamiento separado, modo de Comparación y modo de Captura.
- Contador en Tiempo Real con Oscilador separado.
- 4 Canales para PWM.
- 8 canales y 10 bits para ADC
- Puerto Serial USART Programable
- Interfaz Serie SPI Maestro/Esclavo
- Interfaz serial universal con detector de condición de inicio.
- Temporizador Watchdog Programable con oscilador independiente, presente en el mismo Chip.
- Comparador Analógico dentro del mismo Chip
- Interrupción y encendido por cambio de estado en pines

### **Encapsulados para Entradas/Salidas (E/S)**

- 54 líneas de E/S programables.
- 64-lead TQFP, 64-pad QFN/MLF and 64-pad DRQFN
- Grado de Velocidad: 0 - 4 MHz @ 1.8 - 5.5V, 0 - 8 MHz @ 2.7 - 5.5V
- Rango de Temperatura: -40°C a 85°C
- Bajo consumo de energía:
  - Modo Activo:

- 1 MHz, 1.8V: 330  $\mu$ A
- 32 kHz, 1.8V: 10  $\mu$ A (incluyendo el Oscilador)
- 32 kHz, 1.8V: 25  $\mu$ A (incluyendo el Oscilador y el LCD)
- Modo Power-down
  - 300  $\mu$ A a 1.8V
- Modo Power-save
  - 0.6  $\mu$ A a 1.8V (Incluyendo 32 kHz RTC)

## 2.9.2. Distribución de Pines

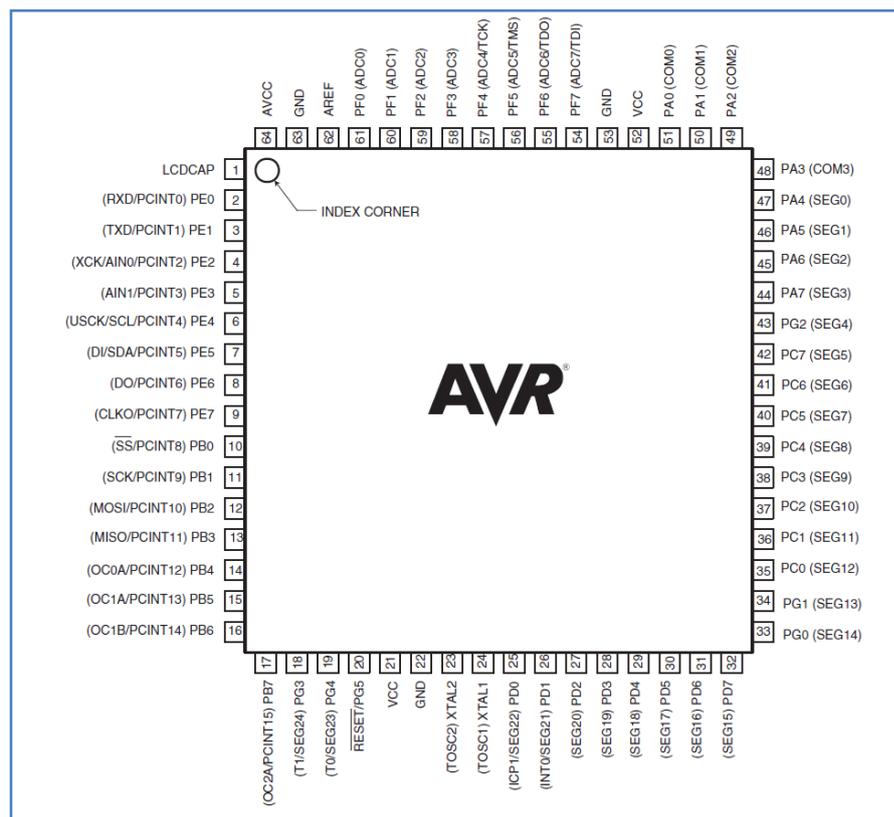


Figura 2.26: Pines del MCU ATmega169 [17]



El núcleo (Core) AVR combina un conjunto de instrucciones RISC con 32 registros para uso de propósito general. Estos 32 registros están directamente relacionados con la Unidad Aritmética Lógica (ALU), permitiendo dos registros independientes para ser accedido al ejecutarse una instrucción en un ciclo de máquina. El resultado de esto es una arquitectura más eficiente, donde se consigue un caudal de flujo y transferencia de inclusive hasta diez veces más rápido que los microcontroladores CISC convencionales.

El ATmega169 provee las siguientes características: 16K bytes en el sistema de Flash Programable con capacidad de lectura y escritura de 512bytes en la EEPROM, 1K bytes en la SRAM, 53 líneas de entrada/salida de propósitos generales, 32 registros de trabajo de propósito general, interfaz JTAG para exploración de límites, soporte de depuración y programación en el mismo chip, un controlador LCD completo en el mismo chip con voltaje interno discreto, tres Temporizadores/Contadores flexibles con modo de Comparación, interrupciones internas y externas, un USART serial programable, Interfaz Serial Universal con detector de condición de inicio, 8 canales ADC de 10 bits, Temporizador Watchdog programable con oscilador interno, un Puerto serial SPI, cinco modos de programación seleccionable para ahorro de energía. El modo Idle detiene al CPU mientras permite a la SRAM, Temporizador/Contador, Puerto SPI y un sistema de interrupción para funcionamiento continuo.

### **2.9.5. CPU AVR CORE**

La principal función del núcleo de la CPU AVR proveer un control para la ejecución correcta del programa. El CPU debe ser capaz de acceder a la memoria, para llevar a cabo cálculos, control de periféricos y atención de interrupciones.

Para maximizar el rendimiento y el paralelismo, el AVR usa una arquitectura de Hardware con memorias y buses separados para programa y datos.

Mientras una instrucción es ejecutada, la siguiente instrucción es ejecutada desde la memoria de programa. Este concepto permite que las instrucciones sean ejecutadas en cada ciclo de máquina ahorrando tiempo y recursos.

El Archivo del Registro (Register File) de rápido acceso es de 32 registros de propósito general de 8 bits trabajando en un simple ciclo de reloj. Esto permite una operación de ciclo simple en la ALU. En una operación típica de la ALU, dos operandos están fuera del Archivo de Registro, la operación es ejecutada, y el resultado es guardado en el Archivo de Registro en un ciclo de máquina.

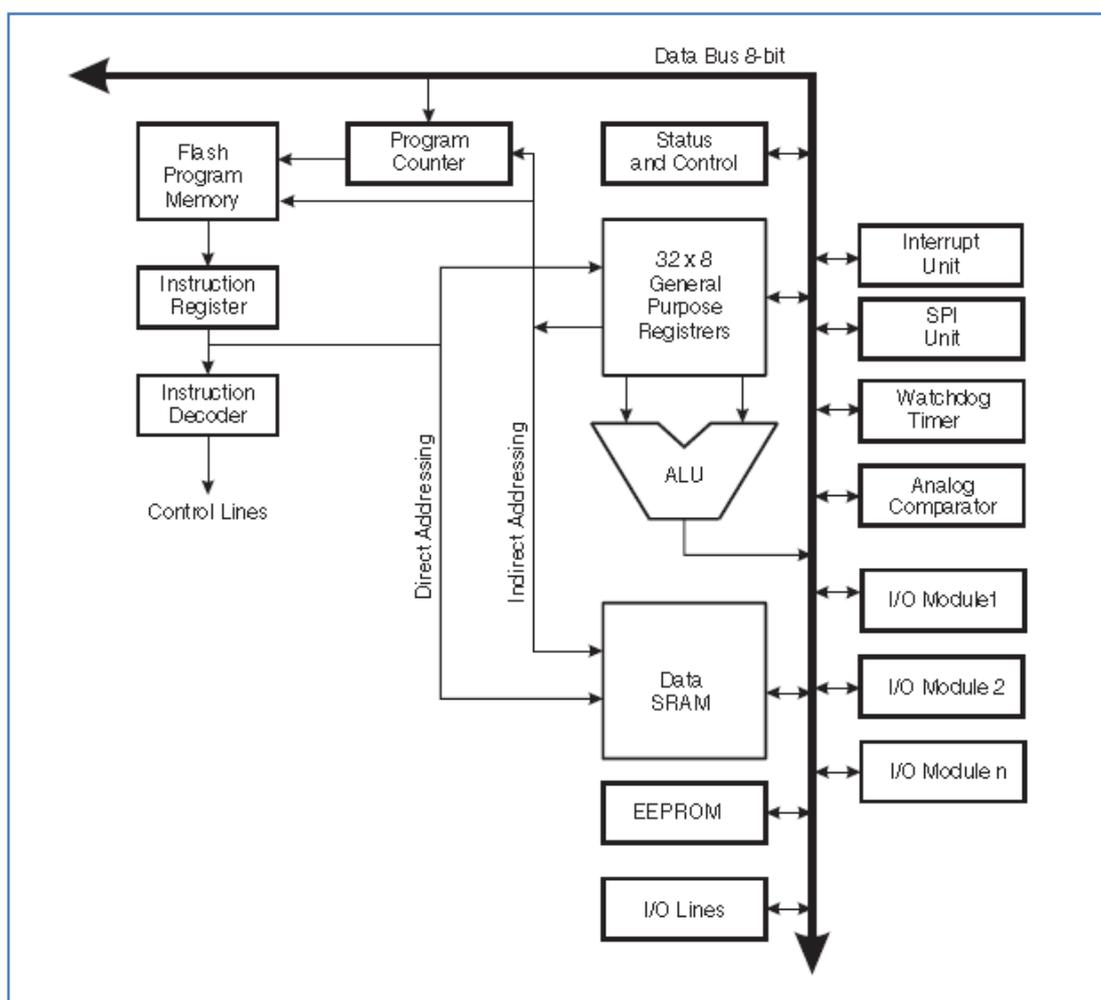


Figura 2.28: Diagrama de Bloques de la Arquitectura AVR [18]

La mayoría de instrucciones del AVR tienen un formato simple de una palabra de 16 bits. Toda dirección en memoria de programa contiene una instrucción de 16 o 32 bits. El espacio de la memoria de programa flash está dividido en dos secciones, la sección Baja del programa y la sección de aplicación de programa. Ambas secciones presentan bits de bloqueo de escritura y protección de lectura/escritura.

### 2.9.6. Puertos de E/S

Los puertos de E/S son un conjunto de líneas (pines) programables como entrada ó salida que dispone el microcontrolador para comunicarse con dispositivos externos.

Todos los pines de cada puerto son programables como entrada o salida de datos configurando el registro asociado respectivo.

Cuando se programa el funcionamiento de un puerto se debe habilitar o deshabilitar las resistencias pull-up internas. Cada pin del puerto tiene independiente su resistencia pull-up como una resistencia invariante hacia la fuente de voltaje, además presenta diodos de protección, uno conectado a Vcc y el otro conectado a GND.

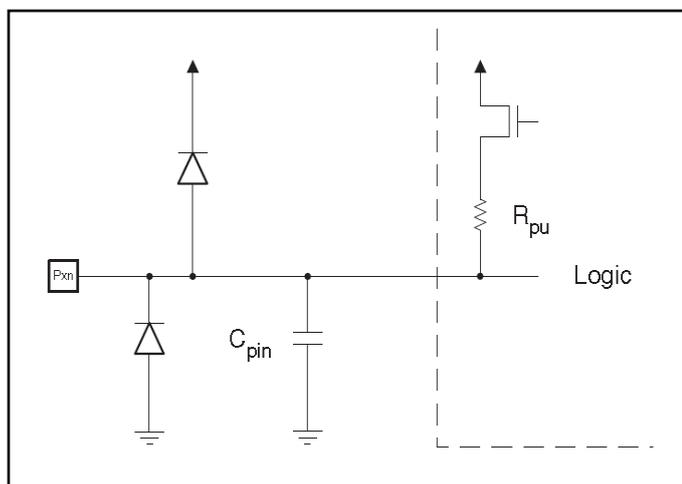


Figura 2.29: Diagrama equivalente del GPIO

En resumen, para cada puerto del microcontrolador (puertos B, C y D) existen tres registros de E/S que permiten configurar cada pin del puerto como entrada o salida, enviar datos a los pines configurados como salidas, y recibir datos de los pines configurados como entradas. Estos registros son: Registro de Direcciones de Datos – DDRx, Registro de Datos – PORTx, y el Puerto de Entrada de Pin – PINx,

En el párrafo anterior, “x” puede ser A, B, C ó D. Es decir, si nos referimos al puerto B, los registros son DDRB, PORTB y PINB. Los registros PINx son de sólo lectura, mientras que los registros PORTx y DDRx son de lectura/escritura. Adicionalmente, el bit “Pull-up Disable” – PUD, en el registro SFIOR inhabilita la función Pull-up para todos los pines de todos los puertos cuando es puesto a nivel alto.

## **Capítulo 3**

### **Diseño e implementación del proyecto**

#### **3.1. Introducción**

A continuación se detalla el proceso de diseño e implementación del proyecto que consiste en el envío de instrucciones desde la AVR Butterfly para el control de los motores BLDC, además de ejercicios básicos como modos de prueba para entablar la comunicación entre los dispositivos LPCXpresso 1769, LPCXpresso 1114, el módulo de control de motores y el módulo TWI AVR Butterfly.

Para explicar cada ejercicio y el proyecto, se hará una pequeña introducción seguida de un diagrama de bloques explicativo. Después se explicará gráficamente el algoritmo utilizado en un diagrama de flujo, y por último agregaremos los códigos de programación utilizados.

### 3.2. Ejercicio 1:

**LPCXpresso 1769 como Maestro usando una memoria EEPROM esclava.**

#### 3.2.1. Introducción

En este ejercicio tiene como objetivo familiarizarnos con las librerías y forma de transmisión usando la comunicación I2C de la tarjeta LPCXpresso 1769. Usaremos la tarjeta en modo maestro para escribir y leer datos desde una EEPROM 24LC32A ubicada en una tarjeta ET-MINI 10 que contiene cuatro EEPROMs direccionables.

#### 3.2.2. Diagrama de bloques

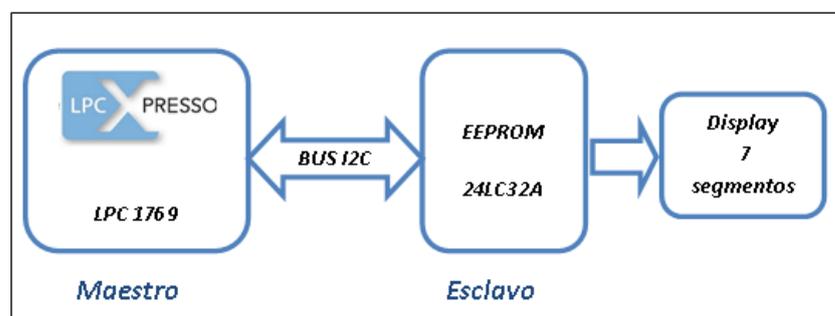


Figura 3.1: Diagrama de bloques ejercicio 1

### 3.2.3. Diagrama de flujo

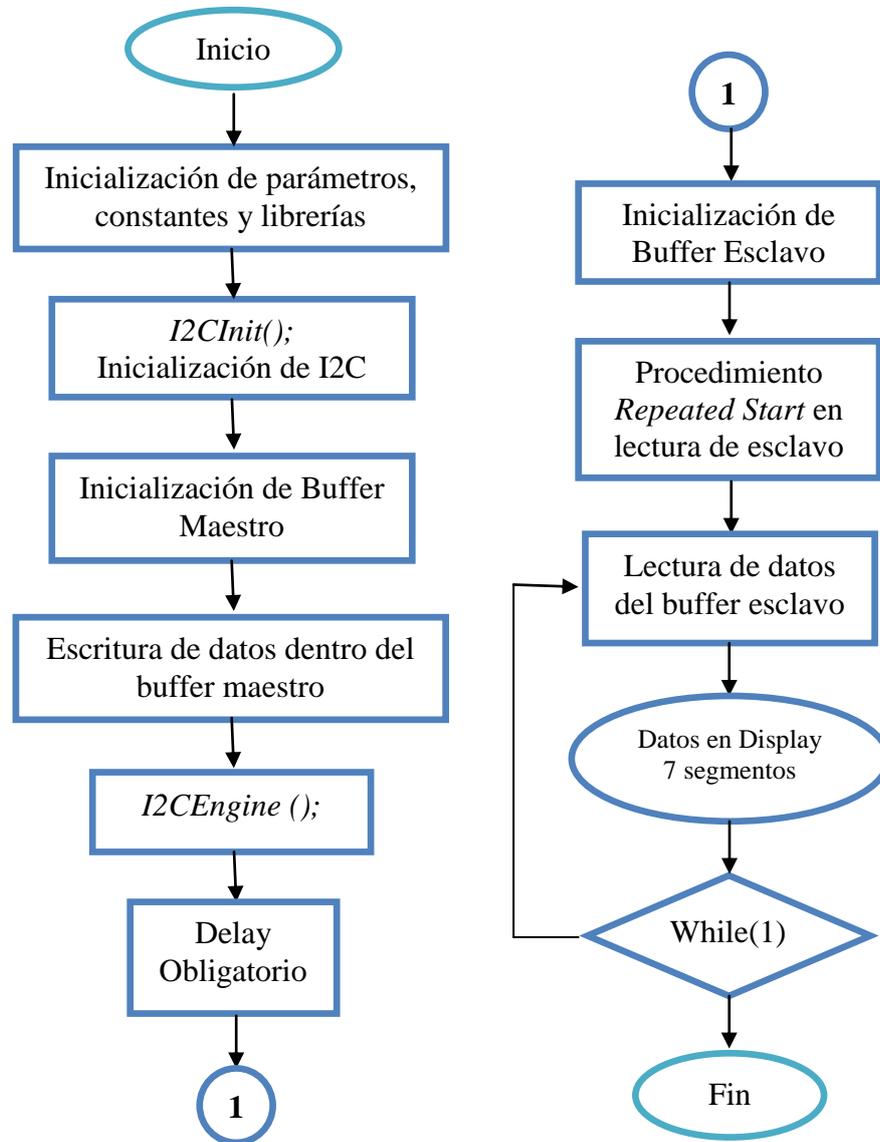


Figura 3.2: Diagrama de Flujo ejercicio 1

### 3.2.4. Descripción del algoritmo

1. Se declaran las variables, constantes y librerías a usarse.
2. Se inicia la función **MAIN()**
3. Se declaran puertos de entradas y salidas, para luego darles un valor de estado de inicio.
4. Se inicializa el **I2CInit()** que prepara los registros, banderas, activa o desactiva resistencias Pull up/down de los puertos I2C, inicializa puertos I2C e instala la interrupción por transmisión I2C (**I2C2\_IRQn**).
5. Define el tamaño del buffer de escritura según el dato a enviar.
6. Se coloca en el buffer de escritura del maestro la dirección del esclavo, la dirección de memoria dentro del esclavo donde se comenzara a escribir, y por último los datos a escribir. Los datos enviados a la EEPROM serán los códigos generados para poder presentar en un display de 7 segmentos ánodo común las letras de FIEC.
7. Se inicializa la transmisión por medio de la instrucción **I2CEngine(numero de puerto)**
8. Se ejecuta un Delay obligatorio después de escribir.
9. Se encera el buffer de lectura del maestro desde el esclavo **I2CSlaveBuffer**.
10. Se leerá los datos del esclavo usando el procedimiento de **Repeated Start**: Se envía como si fuera a realizar una escritura de esclavo, la

dirección de esclavo y la dirección de memoria donde se comenzaría a escribir, para luego enviar un segundo *start* modificando la instrucción de *escritura* por la de *lectura*, accediendo a los datos del esclavo. Todos los datos se cargan en el buffer de lectura de esclavo *I2CSlaveBuffer*.

11. Usando una instrucción *for*, direccionamos cada espacio del buffer para que se presente en un display de 7 segmentos. Cada dato es presentado utilizando un delay para poder apreciarlo.

### 3.2.5. Código fuente

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

// Variable to store CRP value in. Will be placed automatically
// by the linker when "Enable Code Read Protect" selected.
// See crp.h header for more information
__CRPconst unsigned int CRP_WORD=CRP_NO_CRP;

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

extern volatile uint8_t I2CMasterBuffer[I2C_PORT_NUM][BUFSIZE];
extern volatile uint8_t I2CSlaveBuffer[I2C_PORT_NUM][BUFSIZE];
extern volatile uint32_t I2CReadLength[I2C_PORT_NUM];
extern volatile uint32_t I2CWriteLength[I2C_PORT_NUM];

#define PORT_USED      2

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

    uint32_t j=0;

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

```

LPC_GPIO2->FIODIR=0xFFFFFFFF;      /* P2.xx defined as Outputs */
LPC_GPIO2->FIOSET=0xFFFFFFFF;      /* turn off all the LEDs */

I2C2Init();                          /* initialize I2c2 */

/* Write SLA(W), address and one data byte */
I2CWriteLength[PORT_USED]=7;
I2CReadLength[PORT_USED]=0;
I2CMasterBuffer[PORT_USED][0]=PCF8594_ADDR;
I2CMasterBuffer[PORT_USED][1]=0x30;      /* address */

I2CMasterBuffer[PORT_USED][2]=0x85;      /* address */
I2CMasterBuffer[PORT_USED][3]=0x8E;      /* Data0 */
I2CMasterBuffer[PORT_USED][4]=0xCF;      /* Data1 */
I2CMasterBuffer[PORT_USED][5]=0x86;      /* Data0 */
I2CMasterBuffer[PORT_USED][6]=0xC6;      /* Data1 */

I2CEngine(PORT_USED);

for(i=0;i<0x200000;i++);      /* Delay after write */

for(i=0;i<BUFSIZE;i++)
{
    I2CSlaveBuffer[PORT_USED][i]=0xFF;
}
/* Write SLA(W), address, SLA(R), and read one byte back. */
I2CWriteLength[PORT_USED]=3;
I2CReadLength[PORT_USED]=4;
I2CMasterBuffer[PORT_USED][0]=PCF8594_ADDR;
I2CMasterBuffer[PORT_USED][1]=0x30;      /* address */
I2CMasterBuffer[PORT_USED][2]=0x85;      /* address */
I2CMasterBuffer[PORT_USED][3]=PCF8594_ADDR|RD_BIT;
I2CEngine(PORT_USED);
/* Check the content of the Master and slave buffer */
while(1){
    for(i=0;i<4;i++)
    {
        LPC_GPIO2->FIOPIN=I2CSlaveBuffer[PORT_USED][i];
        for(j=6000000;j>0;j--);
    }
}

}

/**** End Of File ****/

```

### 3.3. Ejercicio 2:

#### PIC 16F887 como Maestro y LPCXpresso 1769 como esclavo

##### 3.3.1. Introducción

En el ejercicio anterior, utilizamos la LPC 1769 en modo maestro de lectura y escritura para familiarizarnos con los procesos de la comunicación I2C. Ahora, usaremos la LPC 1769 en modo de esclavo que recibirá instrucciones en su memoria desde un dispositivo maestro ya conocido y usado previamente por nosotros, el microcontrolador PIC 16F887.

El PIC enviara los datos, que serán las letras de la palabra ESPOL para ser presentados en un display de 7 segmentos, cada uno con un delay aleatorio para comprobar que la LPCXpresso 1769 siempre está disponible para recibir los datos como esclavo. Con este ejercicio, aseguramos parte del proyecto principal, dado que se usara la LPC como esclavo que recibe datos.

##### 3.3.2. Diagrama de bloques

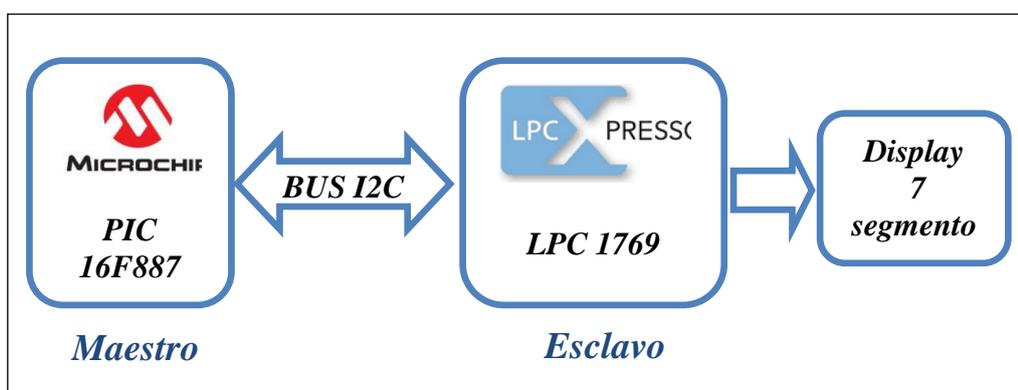


Figura 3.3: Diagrama de bloques ejercicio 2

### 3.3.3. Diagrama de flujo

#### 3.3.3.1. PIC 16F887

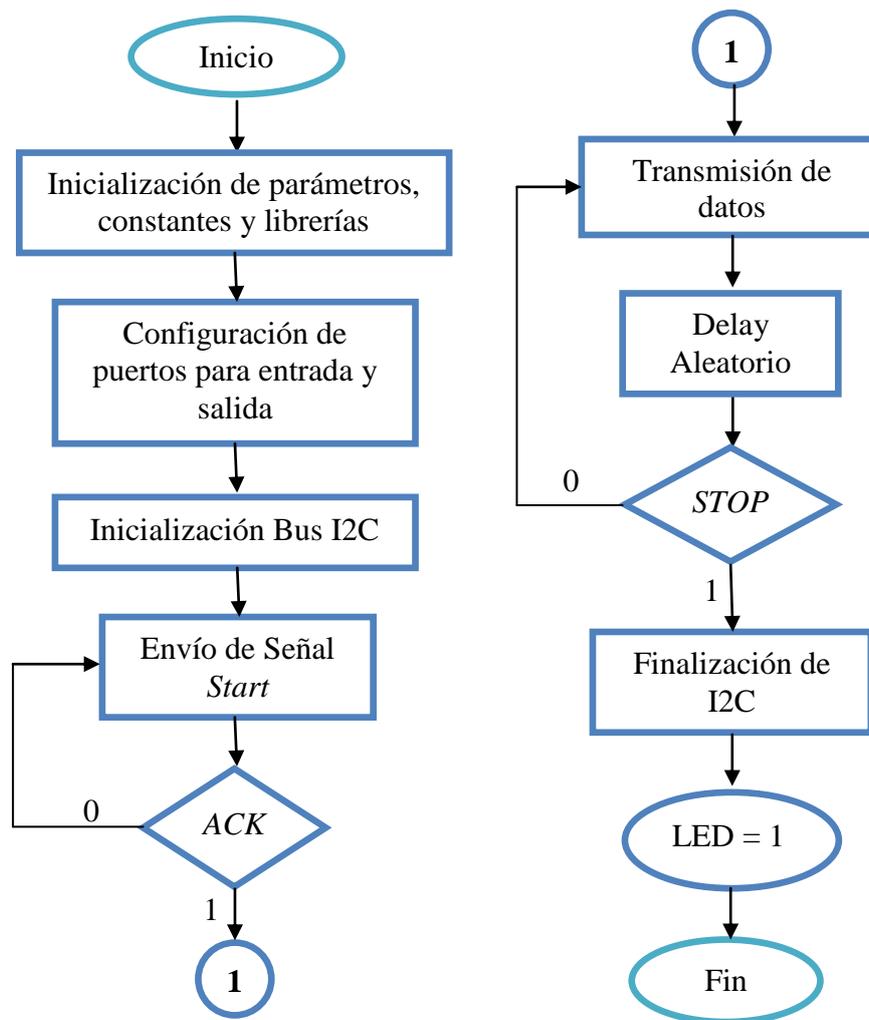


Figura 3.4: Diagrama de Flujo ejercicio 2 – PIC 16F887

### 3.3.3.2. LPCXpresso 1769

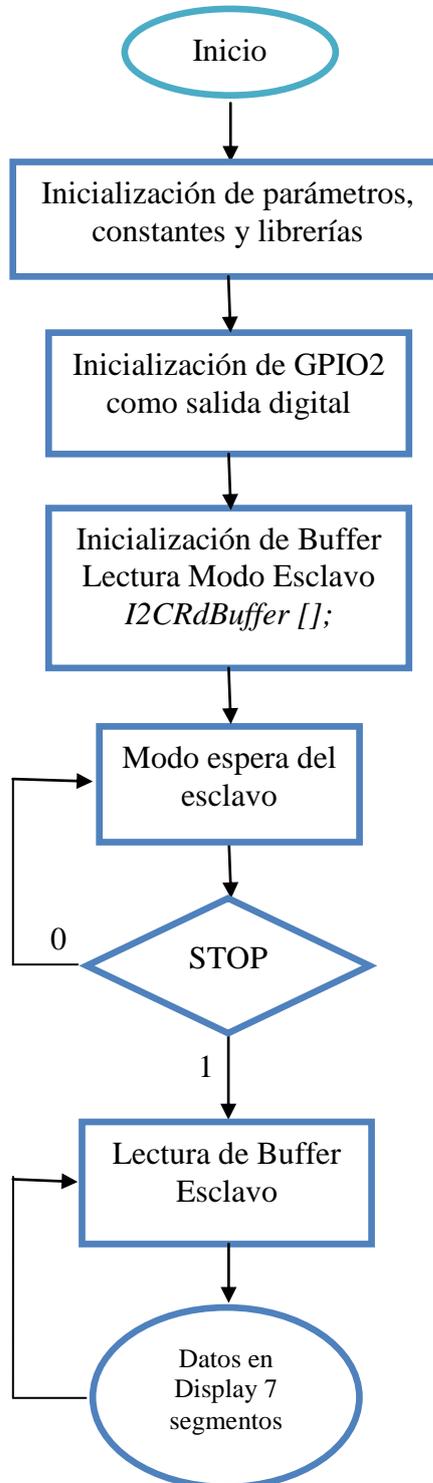


Figura 3.5: Diagrama de Flujo ejercicio 2 – LPCXpresso 1769

### 3.3.4. Descripción de algoritmo

#### 3.3.4.1. PIC 16F887

1. El PIC primero inicializa sus módulos de comunicación I2C.
2. Envía una señal de START seguido de la dirección del esclavo.
3. Recibe el ACK del esclavo notificando que hay conexión.
4. Enviar los datos. El esclavo recibe los datos enviados. La idea es que la tarjeta LPC 1769 proyecte en un display de 7 segmentos la palabra "ESPOL", por lo tanto se envía el código de cada letra para que se muestre en el lado del receptor. Se ha colocado a propósito retardos aleatorios entre cada byte que se envía, para probar que el esclavo está listo para recibir los datos en cualquier momento en que se lo solicite.
5. Se envía la señal de STOP para finalizar la conexión por protocolo I2C.
6. Se enciende un led para indicar que la transmisión se ha completado.

### 3.3.4.2. LPCXpresso 1769

1. Se declaran librerías, variables y constantes a utilizarse en el ejercicio.
2. Se declara un puerto como salida digital, en este caso el puerto GPIO 2, para mostrar los datos recibidos por I2C en un display de 7 segmentos.
3. Se inicializa y se encera el buffer de lectura de modo Esclavo. En este buffer se almacenarán los datos recibidos desde el microcontrolador 16F887.
4. Entra en modo espera. La LPC 1769 estará esperando el start proveniente desde el microcontrolador para iniciar la comunicación I2C. Como hemos puesto delays aleatorios entre datos, la tarjeta debe estar en constante espera de datos.
5. La LPC 1769 estará esperando y recibiendo datos mientras el Maestro no envíe la señal de Stop. Una vez recibida, la comunicación I2C llega a su fin.
6. Se procede a leer los datos del buffer de lectura de modo esclavo, para mostrar los datos uno por uno en el display de 7 segmentos de manera indefinida separados por un delay determinado. Los datos enviados corresponden a las letras de la palabra ESPOL.

### 3.3.5. Código fuente

#### 3.3.5.1. PIC 16F887

```

voidmain(){
  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
  delay_ms(50);//retardo de 50 ms
  I2C1_Start();// Se envía la señal de start
  I2C1_Wr(0xA0);// Envía por medio del bus I2C un byte (dirección del
  esclavo + bit de escritura que es 0)
  delay_ms(50);//retardo de 50 ms
  I2C1_Wr(0x86);// Envía la letra E
  delay_ms(100);//retardo de 100 ms
  I2C1_Wr(0x92);// Envía la letra S
  delay_ms(50);//retardo de 50 ms
  I2C1_Wr(0x8C);// Envía la letra P
  delay_ms(178);//retardo de 178 ms
  I2C1_Wr(0xC0);// Envía la letra O
  delay_ms(306);//retardo de 306 ms
  I2C1_Wr(0xC7);// Envía la letra L
  delay_ms(10);//retardo de 10 ms
  I2C1_Stop();// issue I2C stop signal
  while(1){
    PORTB=0xFF;//Se enciende un led para indicar que la comunicación
    finalizó
  }
}

```

#### 3.3.5.2. LPCXpresso 1769

```

#include <cr_section_macros.h>
#include <NXP/crp.h>
__CRPconstunsignedintCRP_WORD=CRP_NO_CRP;
#include "LPC17xx.h" /* LPC11xx Peripheral
Registers */
#include "type.h"
#include "i2cslave.h"

```

```

volatileuint32_tI2CMasterState=I2C_IDLE;//Estado inicial del bus I2C
volatileuint32_tI2CSlaveState=I2C_IDLE;//Estado inicial del bus I2C
volatileuint32_tI2CMode;
volatileuint8_tI2CWrBuffer[BUFSIZE];//Buffer de escritura
volatileuint8_tI2CRdBuffer[BUFSIZE];//Buffer de lectura
volatileuint32_tI2CReadLength;//Longitud de lectura
volatileuint32_tI2CWriteLength;//Longitud de escritura
volatileuint32_tRdIndex=0;//Indice del buffer de lectura
volatileuint32_tWrIndex=0;//Indice del buffer de escritura
/**** Main Function main()*****/
intmain(void)
{
uint32_tI;
uint32_tj=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++)//buff16
{
I2CRdBuffer[i]=0xFF;//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<5;i++)//buff16
{
LPC_GPIO2->FIOPIN=I2CRdBuffer[i];
for(j=5000000;j>0;j--);
}
}
return(0);
}

```

### 3.4. Ejercicio 3:

**Simulación AVR Butterfly ATmega169 como Maestro y EEPROM como Esclavo.**

#### 3.4.1. Introducción

En este ejercicio se realizara una simulación en Proteus de la comunicación entre la AVR Butterfly ATmega169 como maestro y una EEPROM como esclavo. Usando el Joystick se enviarán comandos de Start, Giro contrarreloj, incrementar y reducir velocidad del BLDC. Según el dato enviado, se mostrará en el LCD de la Butterfly un mensaje indicando el movimiento. Usando el visor de memoria de la EEPROM proporcionado por Proteus, se puede verificar el dato enviado y guardado de la EEPROM.

#### 3.4.2. Diagrama de bloques

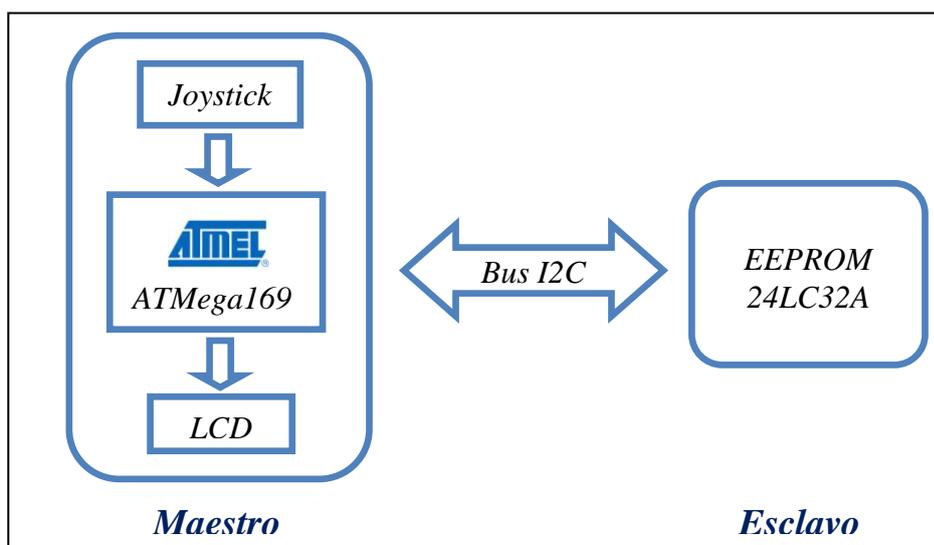


Figura 3.6: Diagrama de Bloques ejercicio 3

### 3.4.3. Diagrama de Flujo

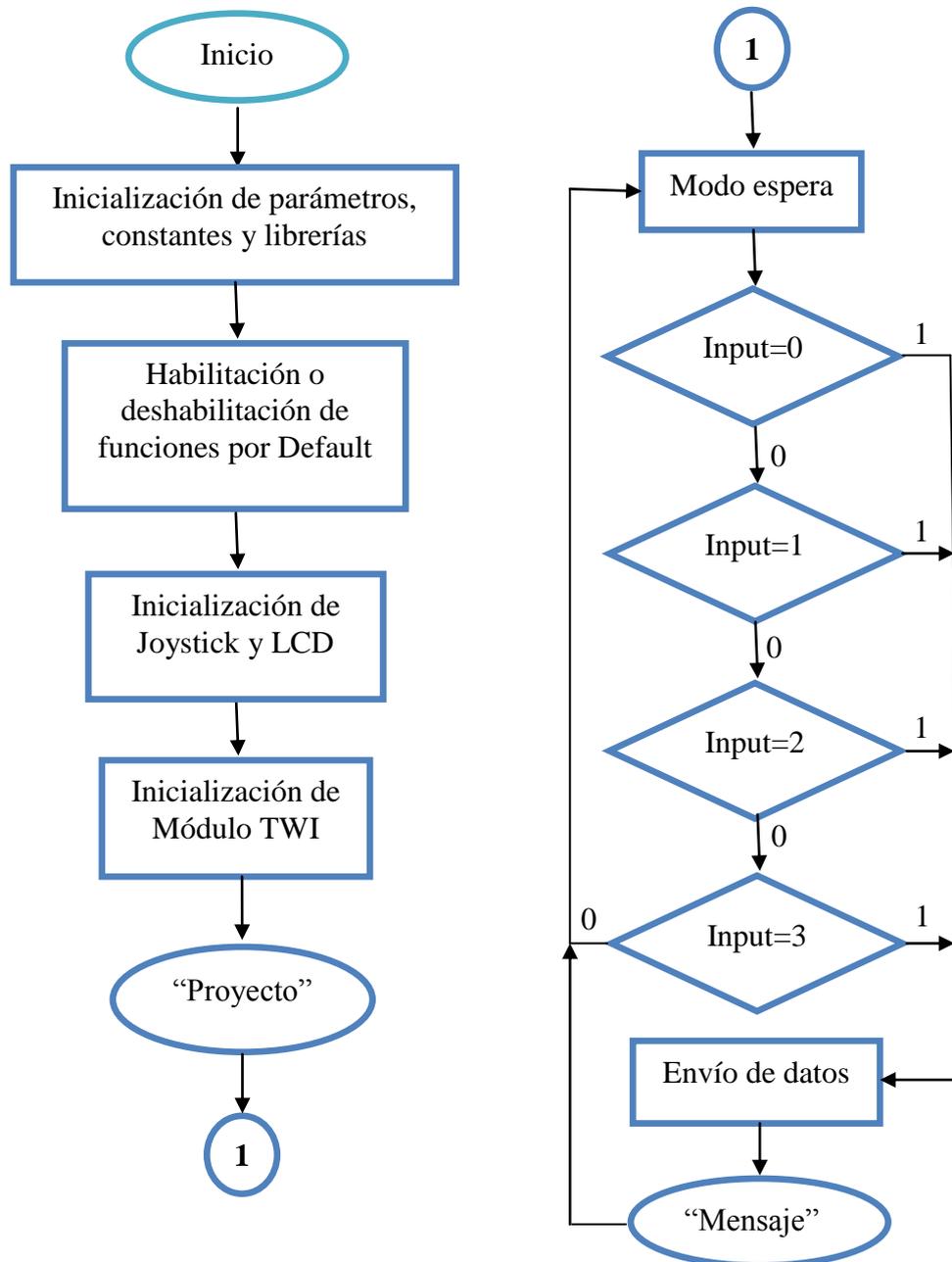


Figura 3.7: Diagrama de Flujo ejercicio 3

#### 3.4.4. Descripción del algoritmo

1. Se declaran librerías, variables y constantes para iniciar la programación del MAIN. Se declara entre otras variables, el mensaje de bienvenida al encender el módulo Butterfly.
2. Se deshabilitan el comparador analógico y la entrada digital PF0-2, dado que son características por default de la Butterfly y al no ser usadas consumen recursos innecesariamente. Se habilitan las resistencias Pull-up del PORTB y PORTE para utilizarlos el joystick.
3. Se inicializa el módulo programado TWI, que es el que permite la comunicación I2C en el módulo Butterfly, utilizando el procedimiento *USI\_TWI\_Master\_Initialise ()*. En este momento, el módulo se encuentra listo para comenzar la comunicación I2C.
4. Se presenta en el LCD el mensaje de bienvenida “Proyecto”, el cual indica que la Butterfly está lista para recibir instrucciones por medio del joystick.
5. Mientras se encuentra en espera de instrucciones, si se ejecuta un movimiento en el joystick, la variable *input* toma un valor específico del 0 al 3 para diferenciar el requerimiento del usuario.
6. El sistema entra en una función *Switch*, en la cual se escoge el procedimiento a realizar según la instrucción dada:
  - Caso KEY\_ENTER: Si se presiona el botón central del joystick. Se presentará en el LCD el mensaje “START” al salir del

*Switch*, y se prepara a enviar el carácter ASCII "C" a la EEPROM. Simboliza el comienzo del movimiento del motor BLDC en un sentido determinado.

- Caso KEY\_NEXT: Si se mueve el joystick a la derecha no se ejecutará acción alguna.
- Caso KEY\_PREV: Si se mueve el joystick a la izquierda. Se presentará en el LCD el mensaje "REVERS" al salir del *Switch*, y se prepara a enviar el carácter en ASCII "L" a la EEPROM. Simboliza el movimiento en sentido contrario al original del motor BLDC.
- Caso KEY\_PLUS: Si se mueve el joystick hacia arriba. Se presentará en el LCD el mensaje "INC" al salir del *Switch*, y se prepara a enviar el carácter en ASCII "U" a la EEPROM. Simboliza el incremento de velocidad del motor BLDC.
- Caso KEY\_MINUS: Si se mueve el joystick hacia abajo. Se presentará en el LCD el mensaje "DEC" al salir del *Switch*, y se prepara a enviar el carácter en ASCII "D" a la EEPROM. Simboliza la disminución de velocidad del motor BLDC.
- Caso DEFAULT: Si no se realiza algún movimiento en el joystick, continua mostrando el mensaje anterior en el LCD y no escribe en la EEPROM.

7. Después de esperar por instrucciones desde el joystick, si se realiza un movimiento, el módulo Butterfly envía el código correspondiente a la EEPROM y muestra en el LCD el mensaje respectivo. Si no se realiza movimiento, continua mostrando el mensaje anterior y no escribe en la EEPROM.
8. Regresa al modo de espera por una siguiente instrucción del joystick.

### 3.4.5. Código fuente

```

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <avr/delay.h>
#include <inttypes.h>
#define F_CPU 8000000UL
#include "mydefs.h"
#include "LCD_functions.h"
#include "LCD_driver.h"
#include "button.h"
#include "USI_TWI_Master.h"
unsigned char i=0,j=0;
unsigned char Comando[4]={0xA0,0x00,0x00,0x00};
unsigned char ComandoSize=4;
int main(void)
{
PGM_P statetext=PSTR("PROYECTO");
uint8_t input;
int i=0;
// Disable Analog Comparator (power save)
ACSR=(1<<ACD);
// Disable Digital input on PF0-2 (power save)
DIDR0=(7<<ADC0D);
// Enable pullups
PORTB=(15<<PB0);
PORTE=(15<<PE4);

```

```

Button_Init();// Initialize pin change interrupt on joystick

LCD_Init();// initialize the LCD

CLKPR=(1<<CLKPCE);// set Clock Prescaler Change Enable
// set prescaler = 8, Inter RC 8Mhz / 8 = 1Mhz
CLKPR=(0<<CLKPS1)|(1<<CLKPS0);
// USART_Init(25.04);

USI_TWI_Master_Initialise();

while(1)
{
if(statetext){

LCD_puts_f(statetext,1);

LCD_Colon(0);

statetext=NULL;
        }

input=getkey();// Read buttons
switch(input){
caseKEY_ENTER:

                statetext=PSTR("START");
                Comando[0]=0xA0;
                Comando[1]=0x00;
                Comando[2]=i;
                Comando[3]='C';//0x07;

                USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
                _delay_us(100);

                i++;

break;
caseKEY_NEXT:
        break;

```

```
case KEY_PREV:
statetext=PSTR("REVERS");
    Comando[0]=0xA0;
        Comando[1]=0x00;
        Comando[2]=i;
        Comando[3]='L';//0x0B;

    USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
        _delay_us(100);
        i++;

    break;
case KEY_PLUS:

statetext=PSTR("INC");
        Comando[0]=0xA0;
        Comando[1]=0x00;
        Comando[2]=i;
        Comando[3]='U';//0x0E;

    USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
        _delay_us(100);
        i++;

    break;
case KEY_MINUS:
statetext=PSTR("DEC");
        Comando[0]=0xA0;
        Comando[1]=0x00;
        Comando[2]=i;
        Comando[3]='D';//0x0D;
    USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
        _delay_us(100);
        i++;

break;
    default:
        break;
}}
return 0;
}
```

### **3.5. Proyecto final de graduación:**

**Control mediante joystick de AVR Butterfly ATmega169 por comunicación I2C con tarjeta LPCXpresso para control de motor BLDC y presentación en display de mensajes de operación.**

#### **3.5.1. Introducción**

En el proyecto final usaremos la AVR Butterfly ATmega169 en modo maestro como en el ejercicio 3, la LPCXpresso LPC 1769 como esclavo como en el ejercicio 2, además el módulo de control de motores de NXP para el movimiento de motores BLDC. El módulo AVR Butterfly receptorá instrucciones del usuario por medio del joystick, presentará mensajes en su LCD y enviará los datos hacia la LPCXpresso1769 por medio del protocolo de comunicación I2C, que procederá a transmitirlos por puerto GPIO al módulo de control de motores acoplado con la LPC 1114, generando la acción indicada en los motores BLDC.

### 3.5.2. Diagrama de bloques

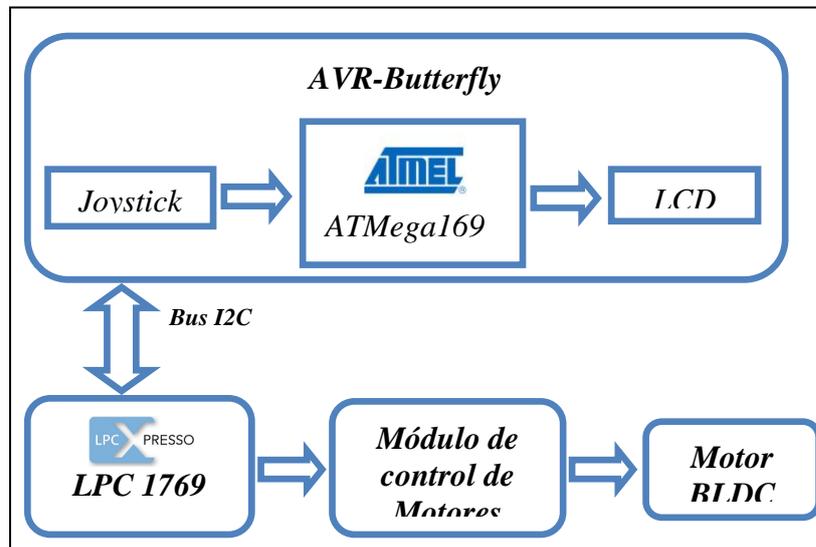


Figura 3.8: Diagrama de Bloques Proyecto Final

### 3.5.3. Diagrama de flujo

#### 3.5.3.1. AVR Butterfly ATMega169



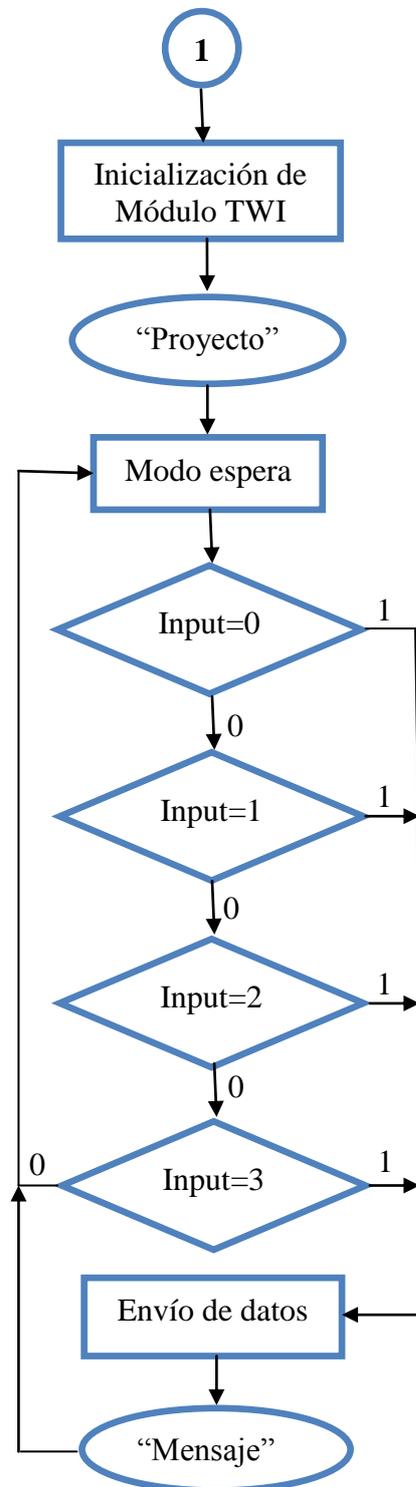


Figura 3.9: Diagrama de Flujo Proyecto Final-AVR Butterfly

### 3.5.3.2. LPCXpresso 1769

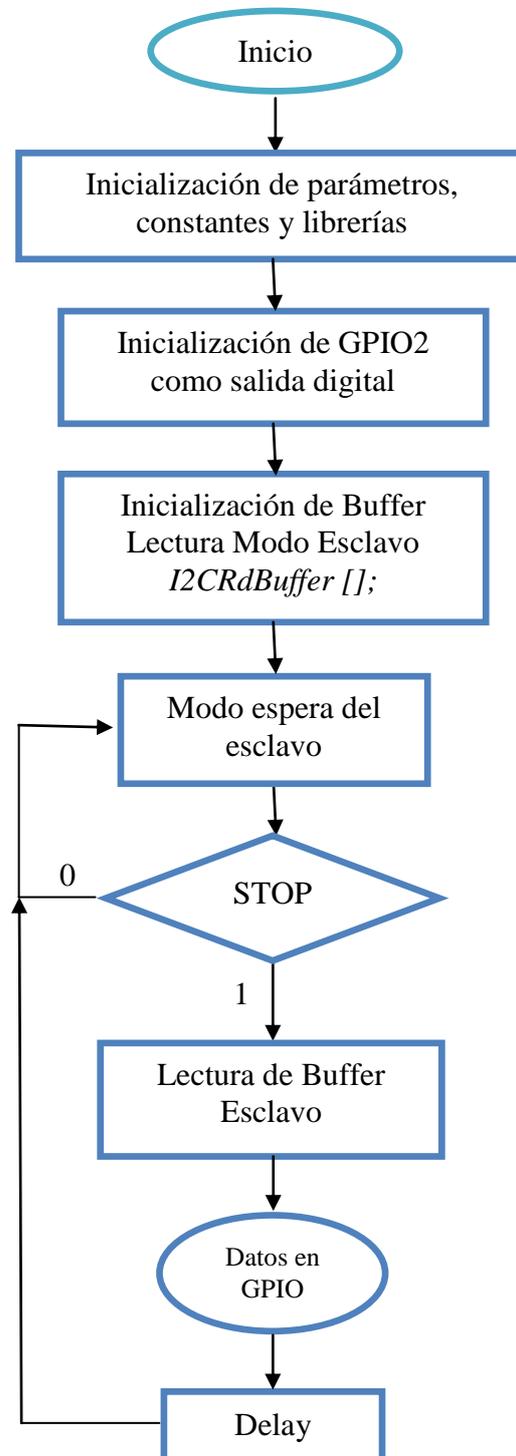


Figura 3.10: Diagrama de Flujo Proyecto Final - LPCXpresso 1769

### 3.5.4. Descripción del algoritmo

#### 3.5.4.1. AVR Butterfly

1. Se declaran librerías, variables y constantes para iniciar la programación del MAIN. Se declara entre otras variables, el mensaje de bienvenida al encender el módulo Butterfly.
2. Se deshabilitan el comparador analógico y la entrada digital PF0-2, dado que son características por default de la Butterfly y al no ser usadas consumen recursos innecesariamente. Se habilitan las resistencias Pull-up del PORTB y PORTE para utilizarlos el joystick.
3. Se inicializa el módulo programado TWI, que es el que permite la comunicación I2C en el módulo Butterfly, utilizando el procedimiento *USI\_TWI\_Master\_Initialise ()*. En este momento, el módulo se encuentra listo para comenzar la comunicación I2C.
4. Se presenta en el LCD el mensaje de bienvenida “Proyecto”, el cual indica que la Butterfly está lista para recibir instrucciones por medio del joystick.
5. Mientras se encuentra en espera de instrucciones, si se ejecuta un movimiento en el joystick, la variable *input* toma un valor específico del 0 al 3 para diferenciar el requerimiento del usuario.

6. El sistema entra en una función *Switch*, en la cual se escoge el procedimiento a realizar según la instrucción dada:
- Caso KEY\_ENTER: Si se presiona el botón central del joystick. Se presentará en el LCD el mensaje “START” al salir del *Switch*, y se prepara a enviar el carácter ASCII “C” a través del bus I2C. Simboliza el comienzo del movimiento del motor BLDC en un sentido determinado.
  - Caso KEY\_NEXT: Si se mueve el joystick a la derecha no se ejecutará acción alguna.
  - Caso KEY\_PREV: Si se mueve el joystick a la izquierda. Se presentará en el LCD el mensaje “REVERS” al salir del *Switch*, y se prepara a enviar el carácter en ASCII “L” a través del bus I2C. Simboliza el movimiento en sentido contrario al original del motor BLDC.
  - Caso KEY\_PLUS: Si se mueve el joystick hacia arriba. Se presentará en el LCD el mensaje “INC” al salir del *Switch*, y se prepara a enviar el carácter en ASCII “U” a través del bus I2C. Simboliza el incremento de velocidad del motor BLDC.
  - Caso KEY\_MINUS: Si se mueve el joystick hacia abajo. Se presentará en el LCD el mensaje “DEC” al salir del *Switch*, y

se prepara a enviar el carácter en ASCII "D" a través del bus I2C. Simboliza la disminución de velocidad del motor BLDC.

- Caso DEFAULT: Si no se realiza algún movimiento en el joystick, continua mostrando el mensaje anterior en el LCD y no envía datos a través del bus I2C.

7. Después de esperar por instrucciones desde el joystick, si se realizo un movimiento, el módulo Butterfly envía el código correspondiente a través del bus I2C y muestra en el LCD el mensaje respectivo. Si no se realiza movimiento, continua mostrando el mensaje anterior y no envía datos a través del bus I2C.
8. Regresa al modo de espera por una siguiente instrucción del joystick.

#### **3.5.4.2. LPCXpresso 1769**

1. Se declaran librerías, variables y constantes a utilizarse en el ejercicio.
2. Se declara un puerto como salida digital, en este caso el puerto GPIO 2. El puerto será usado para enviar instrucciones hacia el módulo de control de motores de NXP.

3. Se inicializa y se encera el buffer de lectura de modo Esclavo. En este buffer se almacenarán los datos recibidos desde el AVR Butterfly por comunicación I2C.
4. Se inicializa el módulo I2C de esclavo y entra en modo espera.
5. La LPC 1769 estará esperando el start proveniente desde la AVR Butterfly para iniciar la comunicación I2C.
6. Una vez iniciada la comunicación I2C, la LPC 1769 estará recibiendo datos y almacenándolos en el buffer de esclavo mientras el Maestro no envíe la señal de Stop. Una vez recibida, la comunicación I2C llega a su fin.
7. Se procede a leer los datos del buffer de lectura de modo esclavo, para poder procesar la instrucción originada en el joystick de la AVR Butterfly. Dependiendo del carácter ASCII recibido, se mostrará en el GPIO2 un código hexadecimal para la operación en el módulo de control de motores:
  - Si recibe "C": La LPCXpresso 1769 mostrará en el GPIO2 el código hexadecimal 0x07 (00000111).
  - Si recibe "L": La LPCXpresso 1769 mostrará en el GPIO2 el código hexadecimal 0x0B (00001011).
  - Si recibe "U": La LPCXpresso 1769 mostrará en el GPIO2 el código hexadecimal 0x0E (00001110).

- Si recibe "D": La LPCXpresso 1769 mostrará en el GPIO2 el código hexadecimal 0x0D (00001101).
  - Si no recibe un carácter, es decir, en el joystick del AVR Butterfly no se ejecutó acción alguna, se mostrará en el GPIO2 el código hexadecimal 0x0F (00001111).
8. Se realiza un delay obligatorio, y regresa al modo espera hasta que se inicie otra comunicación I2C por parte del maestro y repetir del paso 5 al 8 indefinidamente.

### 3.5.5. Código fuente

#### 3.5.5.1. AVR Butterfly ATmega 169

```

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

unsignedchar i=0,j=0;
unsignedchar Comando[2]={0xA0,0x00};
unsignedchar ComandoSize=2;

int main(void)
{
PGM_P statetext=PSTR("PROYECTO");
uint8_t input;
int i=0;

// Disable Analog Comparator (power save)
ACSR=(1<<ACD);
// Disable Digital input on PF0-2 (power save)
DIDR0=(7<<ADC0D);
// Enable pullups
PORTB=(15<<PB0);
PORTE=(15<<PE4);
Button_Init(); // Initialize pin change interrupt on joystick

```

```

LCD_Init(); // initialize the LCD
CLKPR=(1<<CLKPCE); // set Clock Prescaler Change Enable
// set prescaler = 8, Inter RC 8Mhz / 8 = 1Mhz
CLKPR=(0<<CLKPS1)|(1<<CLKPS0);
// USART_Init(25.04);
USI_TWI_Master_Initialise();
white(1)
{
if(statetext){
LCD_puts_f(statetext,1);
LCD_Colon(0);
statetext=NULL;
}
input=getkey(); // Read buttons

switch(input){
caseKEY_ENTER:
statetext=PSTR("START");
Comando[0]=0xA0;
Comando[1]='C'; //0x07;
USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
delay_us(1000);
break;

caseKEY_NEXT:
break;
caseKEY_PREV:
statetext=PSTR("REVERS");
Comando[0]=0xA0;
Comando[1]='L'; //0x0B;
USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
delay_us(1000);
break;
caseKEY_PLUS:
statetext=PSTR("INC");
Comando[0]=0xA0;
Comando[1]='U'; //0x0E;
USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
delay_us(1000);
break;
caseKEY_MINUS:
statetext=PSTR("DEC");
Comando[0]=0xA0;
Comando[1]='D'; //0x0D;
USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
delay_us(1000);
break;
default:
Comando[0]=0xA0;
Comando[1]=0x0F;
USI_TWI_Start_Transceiver_With_Data(&Comando,ComandoSize);
delay_us(100);
break;
}
}
return0;
}

```

### 3.5.5.2. LPCXpresso 1769

```

#include <cr_section_macros.h>
#include <NXP/crp.h>
__CRPconst unsigned int CRP_WORD=CRP_NO_CRP;
#include "LPC17xx.h" /* LPC11xx Peripheral registers */
#include "type.h"
#include "i2cslave.h"
volatile uint32_t I2CMasterState=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 I2CWrtBuffer[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();
LPC_GPIO2->FIODIR=0xFFFFFFFF; /* P2.xx como salidas */
LPC_GPIO2->FIOCLR=0xFFFFFFFF; /* Inicia con bajo GPIO2 */

for(i=0; i<BUFSIZE; i++) //buff16
{
I2CRdBuffer[i]=0xFF; //Se encera el buffer
}
I2CSlave2Init(); /* inicializa I2c */
/* Muestra en el Puerto 2 lo que acaba de leer en el buffer de
lectura */
while(I2CSlaveState!=DATA_NACK){
if(I2CRdBuffer[0]=='C'){
LPC_GPIO2->FIOPIN=0x07; //cod bin 0000111
}
elseif(I2CRdBuffer[0]=='L'){
LPC_GPIO2->FIOPIN=0x0B; //cod bin 00001011
}
elseif(I2CRdBuffer[0]=='U'){
LPC_GPIO2->FIOPIN=0x0E; //cod bin 00001110
}
elseif(I2CRdBuffer[0]=='D'){
LPC_GPIO2->FIOPIN=0x0D; //cod bin 00001101
}
else{
LPC_GPIO2->FIOPIN=0x0F; //cod bin 00001111
}
for(j=1000000; j>0; j--);
}
return(0);
}
//***** END *****/

```

## **Capítulo 4**

### **Desarrollo y Simulación del Proyecto**

#### **4.1. Introducción**

En el capítulo 4, se explicará la realización física del proyecto y los elementos utilizados. Además se mostrarán imágenes de la implementación de los ejercicios realizados y las simulaciones en Proteus del ejercicio 3.

## **4.2. Elementos físicos**

### **4.2.1. Plataforma del Proyecto**

El proyecto realizado esta enfocado a realizarse en etapa de prototipo, es decir, que no se implementará en un circuito impreso. Para la implementación del prototipo hemos utilizado los siguientes elementos:

#### **4.2.1.1. Protoboard**

El Protoboard es una placa de uso genérico reutilizable o semipermanente, usado para construir prototipos de circuitos electrónicos con o sin soldadura. Normalmente se utilizan para la realización de pruebas experimentales, como en nuestro caso.

Está compuesto por bloques de plástico perforados y numerosas láminas delgadas, de una aleación de cobre, estaño y fósforo, que unen dichas perforaciones, creando una serie de líneas de conducción paralelas.

Las líneas se cortan en la parte central del bloque de plástico para garantizar que dispositivos, en circuitos integrados tipo DIP (Dual Inline Packages), puedan ser insertados perpendicularmente a las líneas de conductores.



Figura 4.1: Protoboard

Debido a las características de capacitancia (de 2 a 30 [pF] por punto de contacto) y resistencia que suelen tener los protoboard, están confinados a trabajar a relativamente baja frecuencia (inferior a 10 ó 20 MHz, dependiendo del tipo y calidad de los componentes electrónicos utilizados).

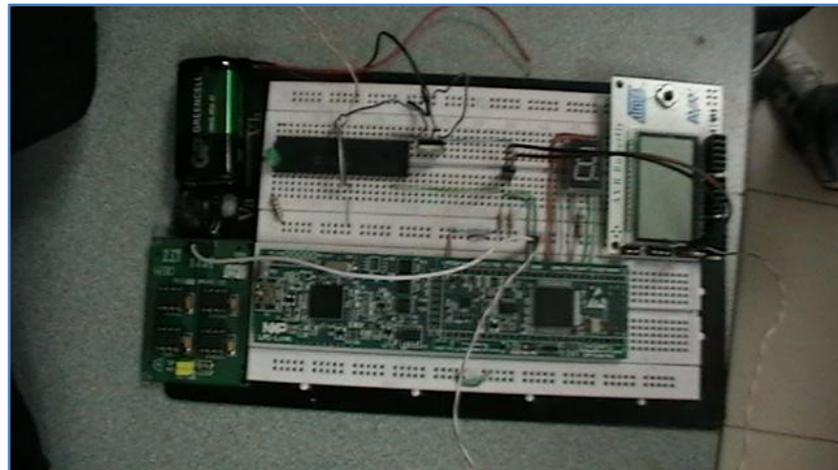


Figura 4.2: Protoboard con plataforma de ejercicios y proyecto final.

#### 4.2.1.2. Kit AVR Butterfly ATMega169

El Kit AVR Butterfly ATMega169 es un poderoso sistema embebido que usaremos como Maestro para enviar las instrucciones de movimiento del BLDC.

En el capítulo 2 se explicó a breves rasgos las características físicas y lógicas del kit AVR Butterfly ATMega169, y en el ejercicio 3 probamos sus funcionamiento como maestro que escribe, en una EEPROM esclava.

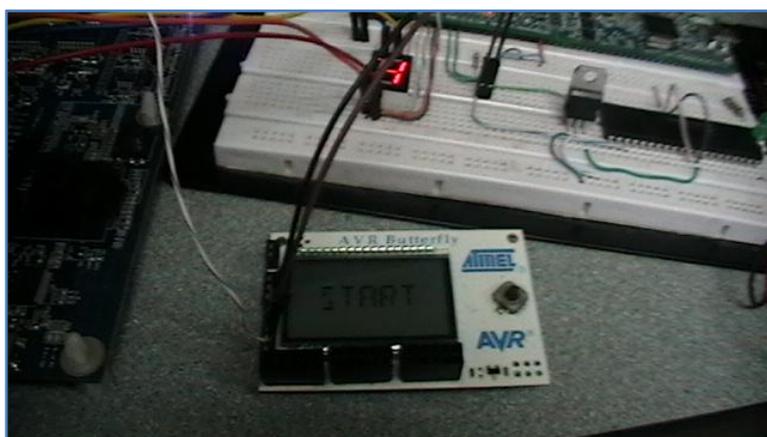


Figura 4.3: AVR Butterfly implementado en ejercicios y proyecto conectado al protoboard

#### 4.2.1.3. Kit LPCXpresso LPC 1769

El kit LPCXpresso LPC 1769 de NXP es usado en nuestro proyecto como esclavo que recibe instrucciones de la AVR Butterfly ATMega 169 por medio del bus I2C, para luego enviarlas por puerto paralelo GPIO al módulo de control de motores.

En el capítulo 2 se habla de sus características básicas brevemente. En el ejercicio 1 la utilizamos como maestro que escribe en una EEPROM para familiarizarnos con la comunicación I2C, mientras que en el ejemplo 2 es usada como un esclavo que recibe datos desde un PIC 16F887 para comprobar su uso en ese modo.

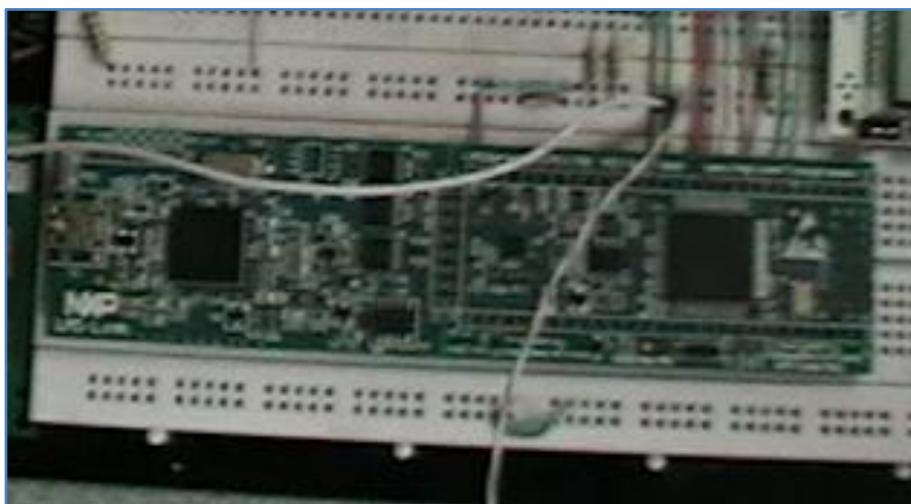


Figura 4.4: LPCXpresso 1769 implementada en el protoboard

#### 4.2.1.4. Kit de Control de Motores NXP

El kit de control de motores de NXP es una herramienta física para el acoplamiento de las tarjetas de NXP para prueba de aplicaciones para motores. Su uso en nuestro proyecto es el manejo de motores BLDC, cuyos movimientos son realizados por el usuario desde el joystick de la AVR Butterfly ATmega 169.

En el capítulo 2, se hace una breve descripción de sus características principales.



Figura 4.5: Módulo de control de motores NXP y motor BLDC

### 4.3. Imágenes y simulaciones

#### 4.3.1. Ejercicio 1: LPCXpresso 1769 como Maestro usando una EEPROM esclava

En el ejercicio 1, se deseaba establecer la comunicación entre la LPCXpresso 1769 en modo maestro que escribe y después lee desde una EEPROM esclava por medio de protocolo I2C. Las imágenes a continuación presentan los datos enviados correspondientes a los caracteres de la palabra "FIEC".

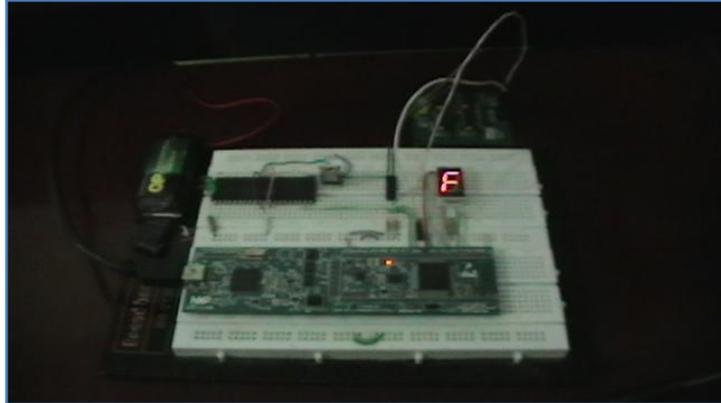


Figura 4.6: Carácter "F"

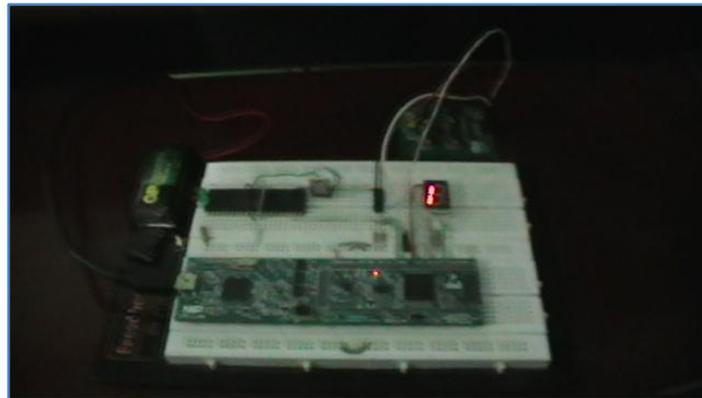


Figura 4.7: Carácter "I"

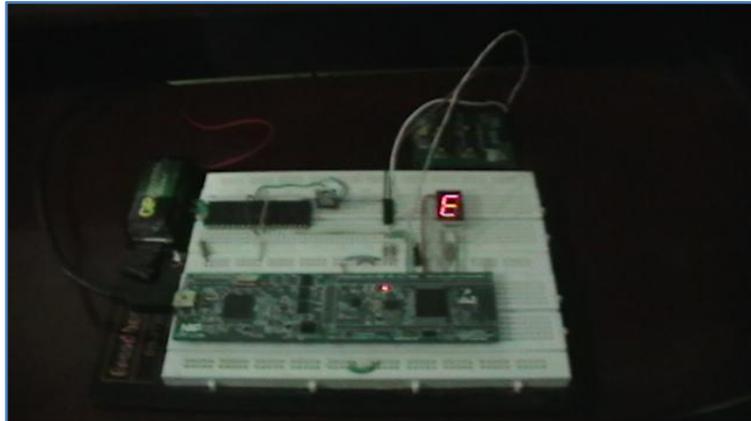


Figura 4.82: Caracter "E"

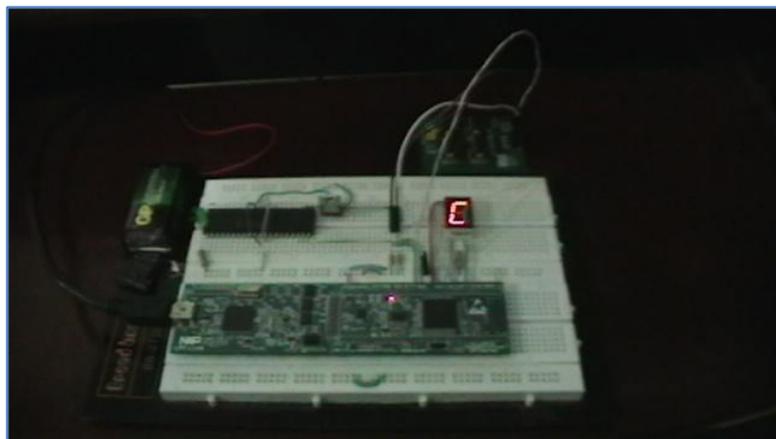


Figura 4.9: Caracter "C"

#### 4.3.1.1. Conclusiones

- Al utilizar el *Repeated Start* la LPCXpresso 1769 comienza la comunicación como si fuera a escribir de nuevo en el esclavo, es decir, envía la dirección del esclavo y la dirección de memoria donde debería escribir solo para localizar una especie de puntero. De ahí, cambia la instrucción de escritura por una de lectura por medio de una

instrucción OR. Luego de cualquier procedimiento de comunicación I2C, se debe colocar un Delay obligatorio para que la tarjeta LPCXpresso 1769 lo procese correctamente.

#### 4.3.2. Ejercicio 2: PIC 16F887 como Maestro y LPCXpresso 1769 como esclavo.

En el ejercicio 2, se utiliza un PIC 16F887 como maestro que escribe y la LPCXpresso 1769 como esclavo que recibe. Además muestra los caracteres recibidos de la palabra “ESPOL” en un display de 7 segmentos.

Las imágenes a continuación presentan los caracteres recibidos por la LPCXpresso 1769, teniendo en cuenta que al finalizar correctamente el envío de datos por comunicación I2C se enciende un led azul.

Las siguientes imágenes muestran la presentación en el display de los caracteres de la palabra “ESPOL”.

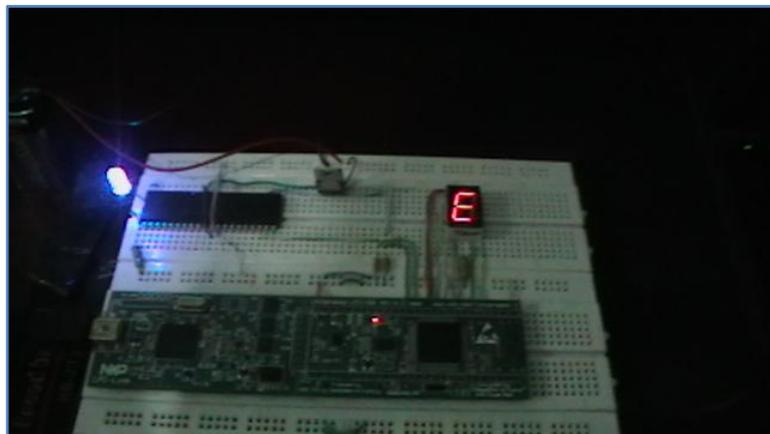


Figura 4.10: Caracter "E"

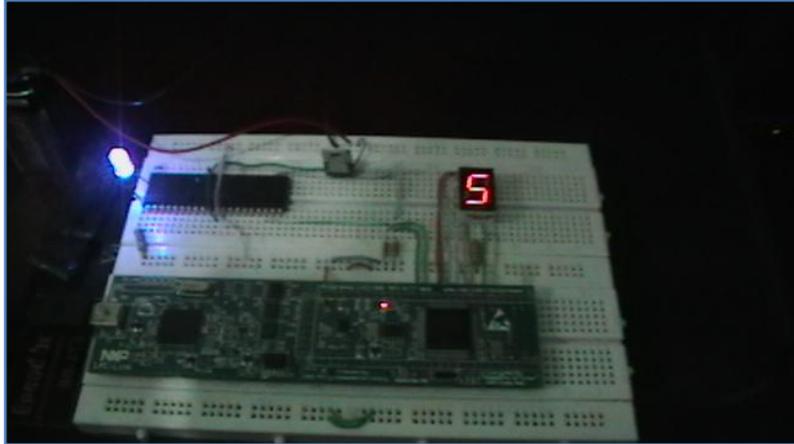


Figura 4.11: Carácter "S"

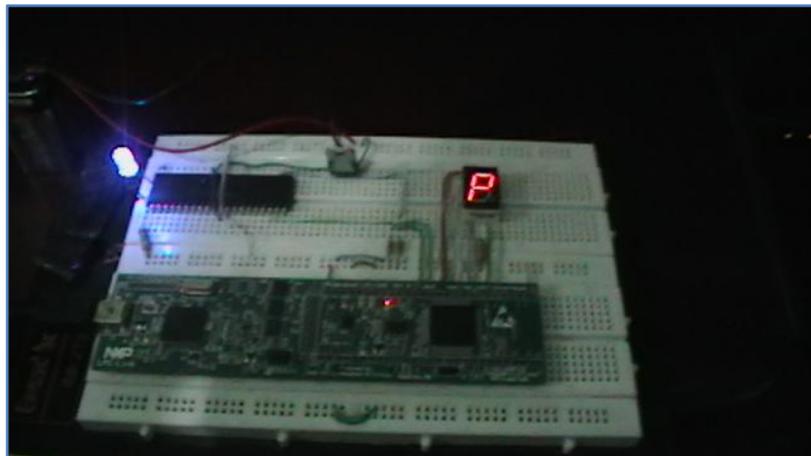


Figura 4.12: Carácter "P"

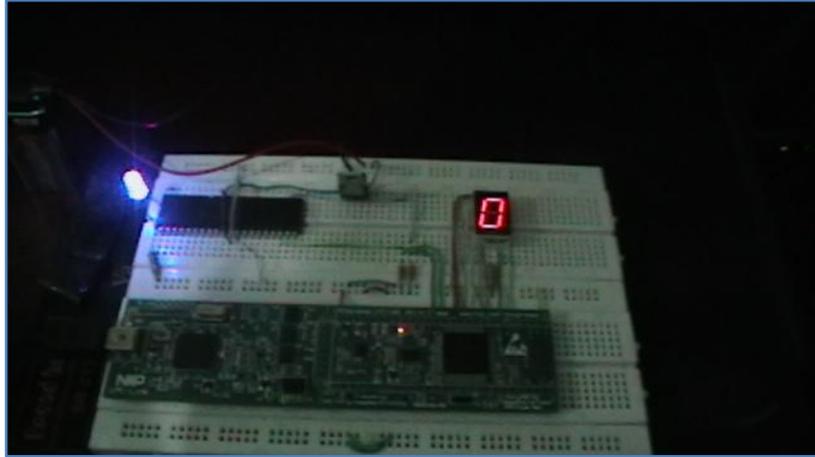


Figura 4.13: Caracter "0"

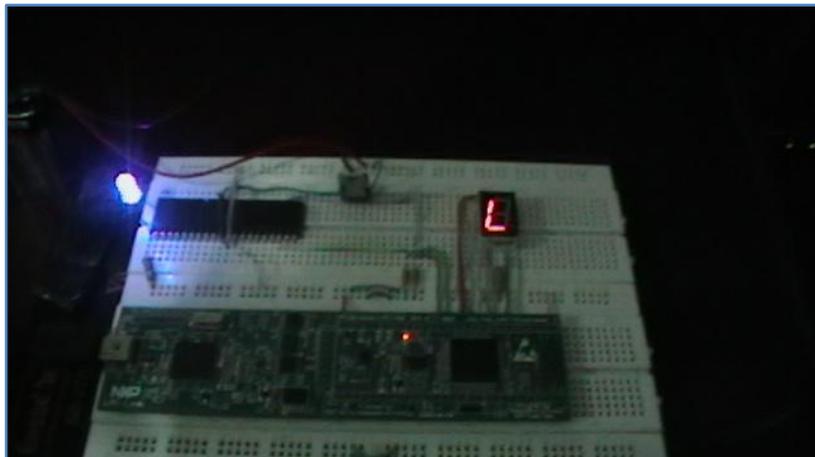


Figura 4.14: Caracter "L"

#### 4.3.2.1. Conclusiones

- En este ejercicio se han incluido distintos delays entre envío de datos por comunicación I2C para comprobar que en modo esclavo la LPCXpresso 1769 está disponible a recibir datos en cualquier momento hasta que el maestro envíe la señal de STOP, finalizando la comunicación.

- Se debe colocar los datos a enviar según el tipo de display de 7 segmentos a utilizar. En nuestro caso, utilizamos un ánodo común y hemos enviado datos que contengan niveles bajos específicos para mostrar los caracteres deseados. Si se desea utilizar un display de 7 segmentos de cátodo común, se deberán cambiar los datos enviados.

### 4.3.3. Ejercicio 3: Simulación AVR Butterfly ATmega169 como Maestro y EEPROM como Esclavo.

Al inicializar la simulación, el LCD del AVR Butterfly ATmega 169 muestra el mensaje de “Proyecto”.

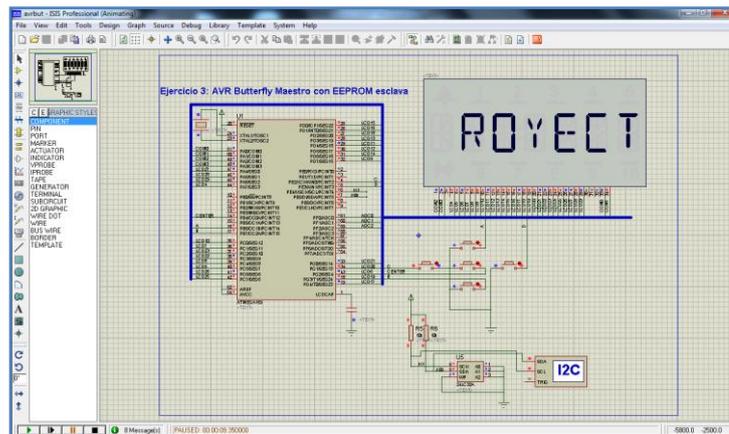


Figura 4.15: Mensaje Inicial de simulación

Cuando presionamos el ENTER en el joystick, se envía el caracter C a la EEPROM, que se puede observar en el visor de memoria. Se muestra en el LCD el mensaje “START” y el monitor de comunicación I2C donde se observa que el dato es enviado cada vez que se presiona ENTER.

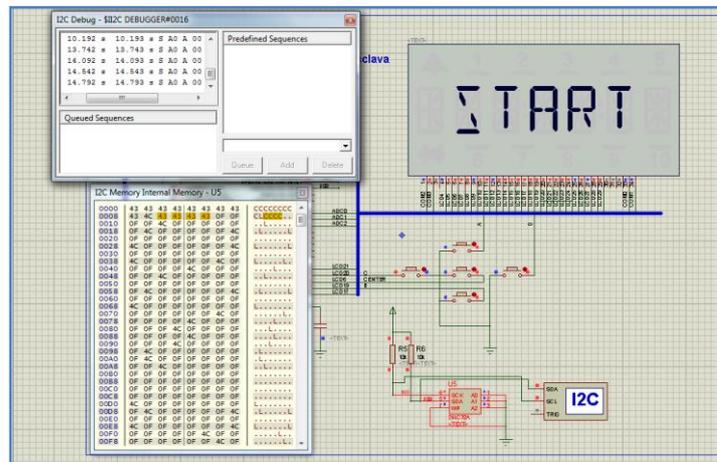


Figura 4.16: Presentación de mensaje al presionar ENTER

Al presionar el botón que representa el movimiento a la izquierda en el joystick, se presenta en el LCD el mensaje “REVERS”. Se puede observar en el monitor de memoria de la EEPROM que se ha escrito el carácter “L”.

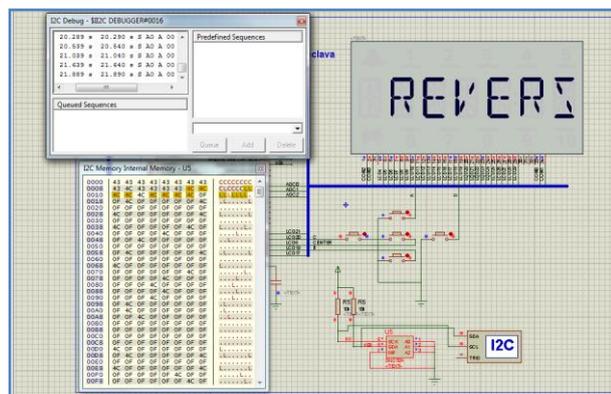


Figura 4.17: Presentación de mensaje al mover joystick a la izquierda

Al presionar el botón que representa el movimiento hacia arriba en el joystick, el LCD presenta el mensaje “INC” y se puede observar que se escribe en la EEPROM el carácter “U”.



#### 4.3.3.1. Conclusiones

- Este ejercicio fue solamente implementado en nivel de simulación, dado que así se aprovechan las herramientas del programa Proteus para visualizar los datos en los visores de memoria y bus I2C. Para verificar que el dato ha sido correctamente enviado, debemos visualizar la ventana de memoria de la EEPROM, donde aparecerá el carácter enviado resaltado con color amarillo. Implementar este ejercicio físicamente sería innecesario al no poder comprobar la transmisión de una manera sencilla.
- Cada vez que se modifique la codificación del ATmega169, se debe actualizar el archivo .HEX en la simulación para poder ver los cambios realizados. Para utilizar el joystick es necesario inicializar PORTB y PORTE como entradas. Estos recibirán los niveles altos y bajos correspondientes para su interpretación en el microcontrolador.

**4.3.4. Proyecto de Graduación: Control mediante joystick de AVR Butterfly ATmega169 por comunicación I2C con tarjeta LPCXpresso para control de motor BLDC y presentación en display de mensajes de operación.**

En el proyecto final, acoplamos parte de los ejercicios para realizar la comunicación I2C maestro esclavo entre la AVR Butterfly ATmega 169 y la LPCXpresso 1769. Por medio del joystick incorporado en la AVR Butterfly, se enviaron instrucciones para procesar movimientos en el motor BLDC y para comprobar que se envían los datos, presentamos en un display de 7 segmentos un mensaje característico según el dato enviado. Las siguientes imágenes muestran lo presentado en el display de 7 segmentos:

- Al inicio, sin presionar o mover el joystick, se debe presentar en el LCD del AVR Butterfly el mensaje "PROYECTO".

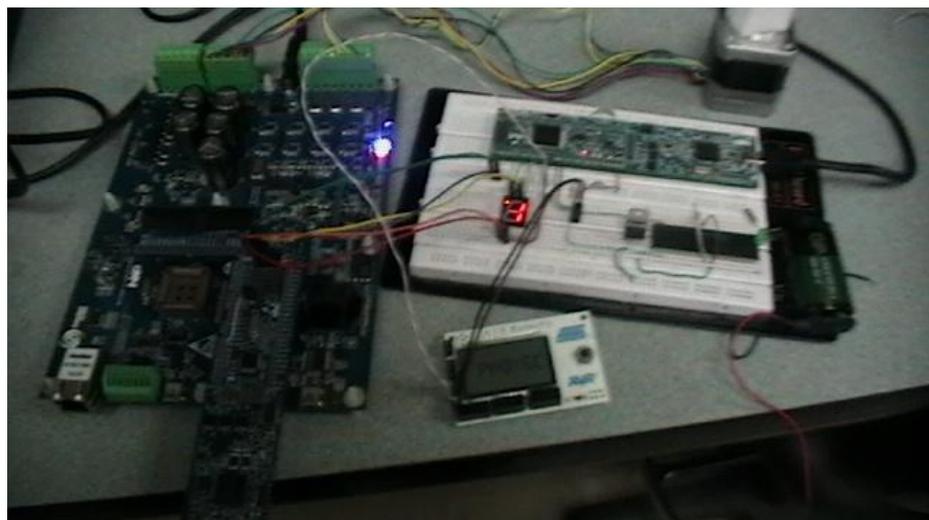


Figura 4.20: Presentación inicial de LCD de mensaje "PROYECTO"

- Al presionar ENTER en el joystick, se envía el dato hexadecimal 0x07 que representa el inicio o parada del motor BLDC.

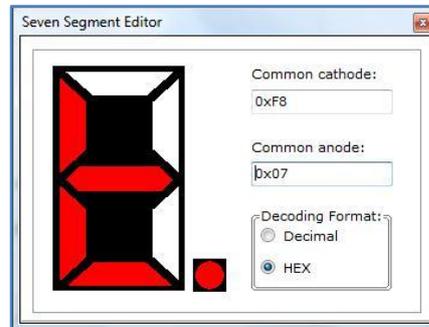


Figura 4.21: Presentación en display 7 segmentos de instrucción START

El motor BLDC comenzará o finalizará su funcionamiento. Al inicio por default, girará en sentido anti horario. En el LCD del AVR Butterfly se presentará el mensaje “START” al comenzar el movimiento.

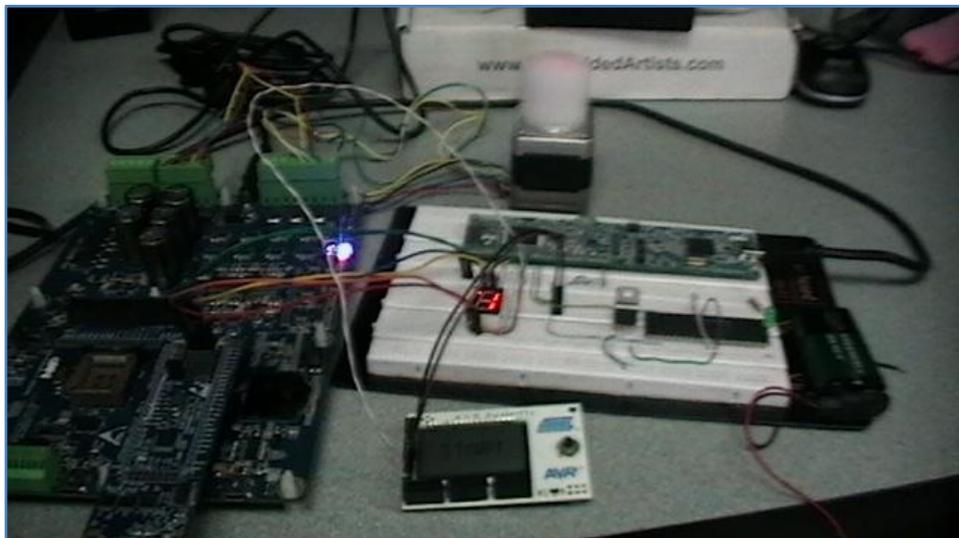
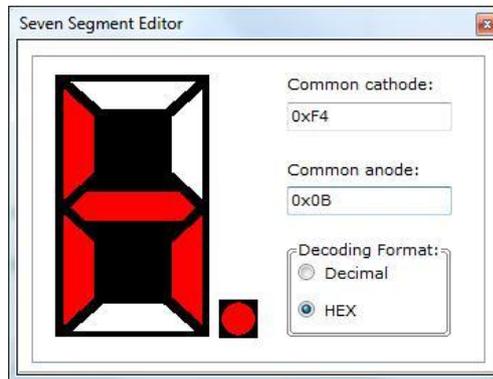


Figura 4.22: Funcionamiento después de presionar ENTER en el joystick

- Al mover el joystick a la izquierda, se envía el dato hexadecimal 0x0B que representa el cambio de sentido de giro en el motor BLDC.



**Figura 4.23: Presentación en display de 7 segmentos de instrucción REVERS**

El motor BLDC comenzará a girar en sentido contrario al que estaba funcionando, es decir, de horario pasará a anti horario y viceversa. En el LCD del AVR Butterfly se presentará el mensaje “REVERS” después de ejecutar la instrucción en el BLDC.

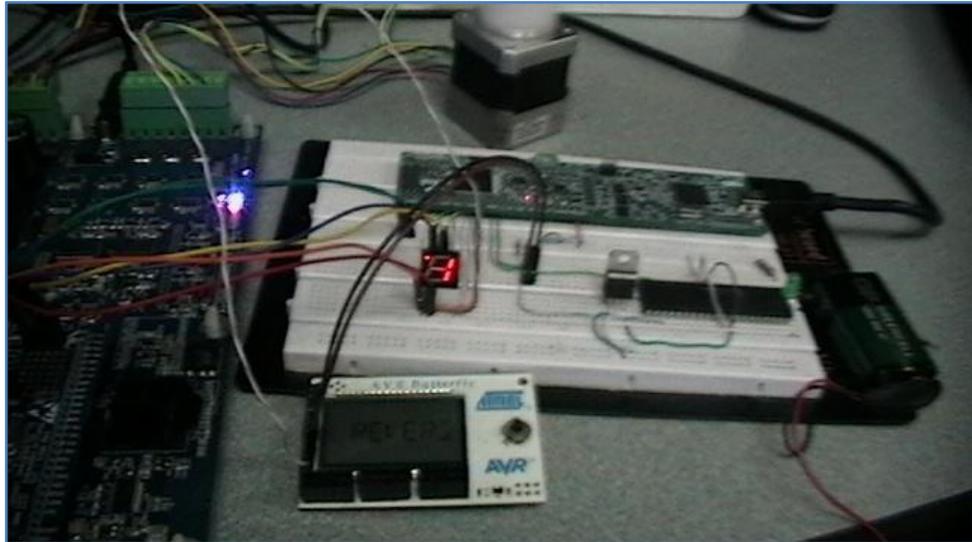


Figura 4.24: Funcionamiento después de mover a la izquierda el joystick

- Al mover el joystick hacia arriba, se envía el dato hexadecimal 0x0E que representa el incremento de velocidad de giro en el motor BLDC. En el proyecto implementado, se puede escuchar cada vez un sonido más fino al incrementar la velocidad del motor BLDC.

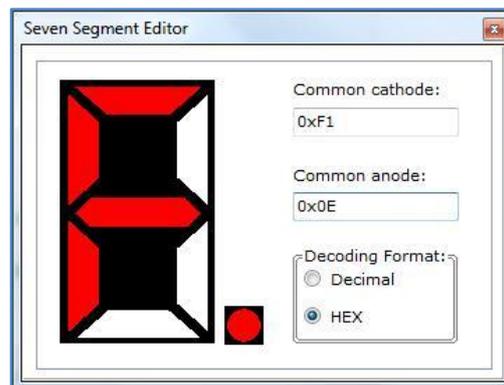


Figura 4.25: Presentación en display de 7 segmentos de instrucción INC

El motor BLDC comenzará a girar a mayor velocidad con cada instrucción del joystick. En el LCD del AVR Butterfly se presentará el mensaje “INC” después de ejecutar la instrucción en el BLDC.

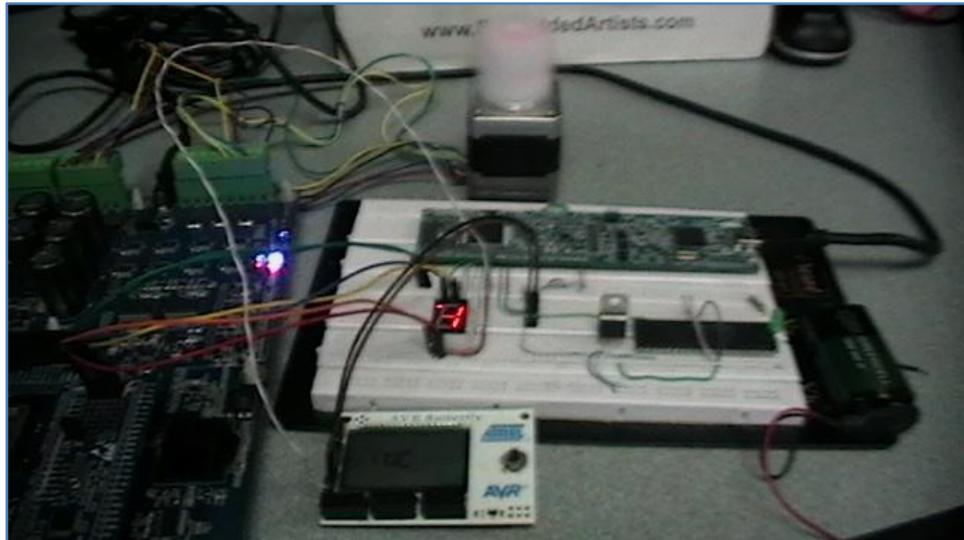


Figura 4.26: Funcionamiento después de mover hacia arriba el joystick

- Al mover el joystick hacia abajo, se envía el dato hexadecimal 0x0D que representa la dismunición de velocidad de giro en el motor BLDC. En el proyecto implementado, se puede escuchar cada vez un sonido más grueso al disminuir la velocidad del motor BLDC.

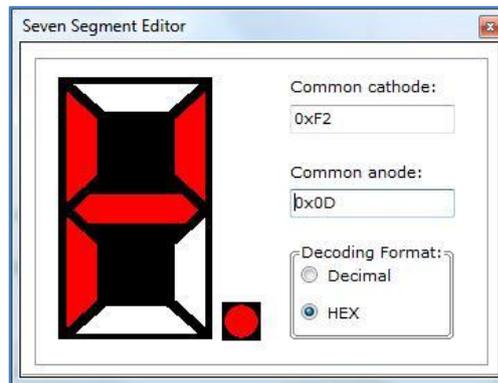


Figura 4.27: Presentación en display de 7 segmentos de instrucción DEC

El motor BLDC comenzará a girar a menor velocidad con cada instrucción del joystick. En el LCD del AVR Butterfly se presentará el mensaje “DEC” después de ejecutar la instrucción en el BLDC.

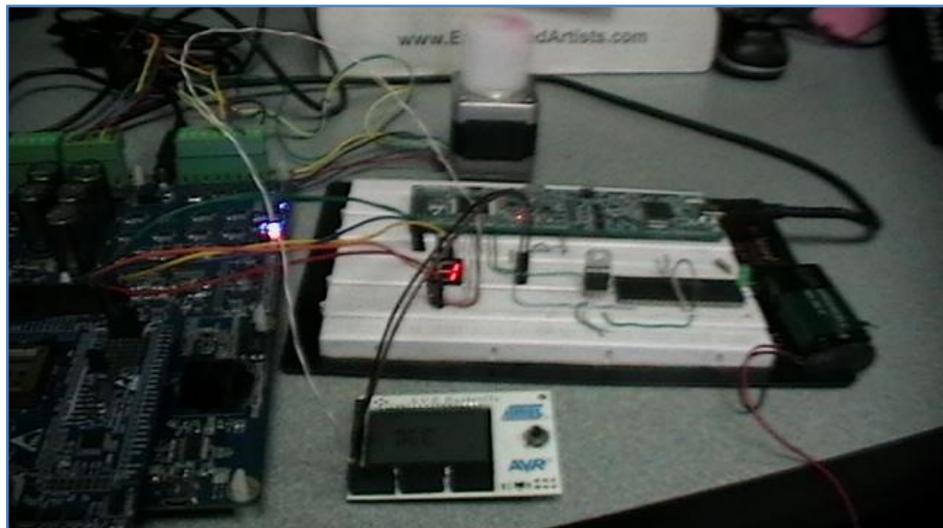


Figura 4.28: Funcionamiento después de mover hacia abajo el joystick

- Si no se realiza movimiento alguno en el joystick, se enviará el dato hexadecimal 0x0F. Dado que el control de motores necesita un bajo

especifico para realizar una acción, si todos los datos son altos, el control de motores estará esperando por una siguiente instrucción del joystick. Se puede decir que este mensaje esta activo cuando la LPCXpresso 1769 está en modo espera.

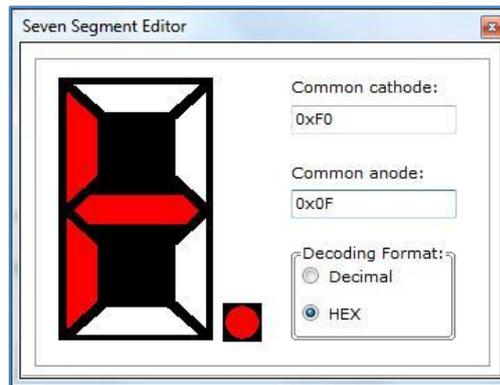


Figura 4.29: Presentación en display de 7 segmentos de estado de espera

#### 4.3.4.1. Conclusiones

- Para visualizar que la instrucción generada en el joystick ha sido enviada correctamente a la LPCXpresso 1769, se ha colocado un display de 7 segmentos que presentará un encendido de un led distinto para cada instrucción enviada. En general para que el proceso de comunicación I2C no tenga problemas debemos activar las resistencias Pull-Up para evitar daños en los puertos destinados para la comunicación, dado que están en configuración de colector abierto. Así mismo, debemos colocar los cables entre dispositivo maestro y esclavo de la manera correcta.

- Para que las acciones generadas por el usuario mediante el joystick funcionen en el motor BLDC, se debe colocar correctamente los cables de comunicación entre la LPCXpresso 1769 y la LPCXpresso 1114. Si se colocan de modo incorrecto no ocasionaran daño físico a las tarjetas, pero el motor podría ejecutar una acción distinta a la requerida.

## **Conclusiones**

1. Con el desarrollo de los ejercicios y su implementación práctica y simulada pudimos darnos cuenta del proceso de comunicación I2C. Esta comunicación permite la implementación de varios esclavos con un maestro determinado, de manera que pueden ser seleccionados utilizando su dirección de esclavo mientras los otros se mantienen inactivos. Además se garantiza la comunicación entre maestro y esclavo por medio de mensajes de confirmación, es decir, que la comunicación I2C no se realiza si el esclavo no responde con un mensaje de confirmación ACK después de haberle enviado un START.

2. La programación de los ejercicios de prueba se pudo simplificar gracias a que los comandos o instrucciones utilizados para operar el protocolo I2C son relativamente simples. Solo fue necesario entender el funcionamiento de los ejemplos prototipo dados por el desarrollador del software de programación y modificarlos en ciertas partes para utilizarlos en la realización del proyecto.
3. El módulo de control de motores procesa instrucciones para el movimiento de los motores BLDC de manera paralela, donde el dato debe contener un valor lógico bajo para habilitar la acción a realizar mientras las demás están en alto. Si el módulo recibe todos los valores lógicos alto, no realizara acción alguna, continuando con la acción del motor que se encuentra activa en el momento.

## Recomendaciones

1. Se debe tener en cuenta que la LPC 1769 trabaja de manera distinta a los anteriores microcontroladores usados durante proyectos pasados, es decir, que para realizar la configuración de un GPIO se deben colocar 1 en los pines que serán salidas y 0 en los de entrada.
2. Para un aprovechamiento óptimo de la AVR Butterfly se deben desactivar ciertas opciones por default, que consumen recursos innecesarios en ciertas aplicaciones.
3. Se debe tener especial cuidado al manipular los módulos de trabajo, ya que cualquier contacto innecesario con los componentes electrónicos

podría causar que se dañen por estática, dado que los elementos están diseñados para trabajar a corrientes muy bajas.

4. Para evitar confusiones al colocar los cables en los puertos correctos después de una desconexión accidental, se deben colocar etiquetas en el cableado, por sobretodo en los que comunican pines de comunicación y de voltaje de alimentación, dado que por posible error se podrían dañar seriamente los elementos de las tarjetas de trabajo.

## Anexo 1

### Librerías empleadas para uso del modo esclavo de la LPCXpresso

1769

#### I2cslave.h

```
/* *****  
 * $Id:: i2cslave.h 5745 2010-11-30 23:27:22Z usb00423  
$  
 * Project: NXP LPC17xx I2C Slave example  
 * Description:  
 * This file contains I2C Slave code header definition.  
 *****  
 * products. This software is supplied "AS IS" without any  
 warranties.  
 * NXP Semiconductors assumes no responsibility or liability for  
 the  
 * use of the software, conveys no license or title under any  
 patent,  
 * copyright, or mask work right to the product. NXP  
 Semiconductors  
 * reserves the right to make changes in the software without  
 * notification. NXP Semiconductors also make no representation  
 or  
 * warranty that such application will be suitable for the  
 specified  
 * use without further testing or modification.  
 *****/  
#ifndef __I2CSLAVE_H  
#define __I2CSLAVE_H
```

```

#define FAST_MODE_PLUS          1

#define BUFSIZE                 16
#define MAX_TIMEOUT             0x00FFFFFF

#define PCF8594_ADDR           0xA0
#define READ_WRITE              0x01

#define RD_BIT                  0x01
#define I2C_IDLE                0
#define I2C_STARTED            1
#define I2C_RESTARTED          2
#define I2C_REPEATED_START     3
#define DATA_ACK               4
#define DATA_NACK              5
#define I2C_WR_STARTED          6
#define I2C_RD_STARTED          7
#define I2CONSET_I2EN          0x00000040 /* I2C Control Set Register
*/
#define I2CONSET_AA             0x00000004
#define I2CONSET_SI             0x00000008
#define I2CONSET_STO            0x00000010
#define I2CONSET_STA            0x00000020
#define I2CONCLR_AAC            0x00000004 // I2C Control clear Register
#define I2CONCLR_SIC            0x00000008
#define I2CONCLR_STAC           0x00000020
#define I2CONCLR_I2ENC          0x00000040

#define I2DAT_I2C               0x00000000 /* I2C Data Reg */
#define I2ADR_I2C               0x00000000 /* I2C Slave Address Reg */
#define I2SCLH_SCLH             0x00000180 // I2C SCL Duty Cycle High
Reg
#define I2SCLL_SCLL             0x00000180 /* I2C SCL Duty Cycle Low Reg
#define I2SCLH_HS_SCLH          0x00000008 /* Fast Plus I2C SCL Duty
Cycle High Reg */
#define I2SCLL_HS_SCLL          0x00000008 /* Fast Plus I2C SCL Duty
Cycle Low Reg */

extern void I2C0_IRQHandler(void );
extern void I2C1_IRQHandler(void );
extern void I2C2_IRQHandler(void );
extern void I2CSlave0Init(void );
extern void I2CSlave1Init(void );
extern void I2CSlave2Init(void );

#endif /* end __I2CSLAVE_H */
/*****
*
*                               End Of File
*****/

```

## 12cslave2.c

```
/*
 * $Id:: i2cslave2.c 5866 2010-12-08 21:44:48Z usb00423
 *
 * Project: NXP LPC17xx I2C Slave example
 *
 * Description:
 * This file contains I2C slave code example which include
 * I2C slave
 * initialization, I2C interrupt handler, and APIs for I2C
 * slave access.
 *
 * Software that is described herein is for illustrative
 * purposes only
 * which provides customers with programming information
 * regarding the
 * products. This software is supplied "AS IS" without any
 * warranties.
 * NXP Semiconductors assumes no responsibility or liability for
 * the
 * use of the software, conveys no license or title under any
 * patent,
 * reserves the right to make changes in the software without
 * notification. NXP Semiconductors also make no representation
 * or
 * warranty that such application will be suitable for the
 * specified
 * use without further testing or modification.
 */
#include "lpc17xx.h"
#include "type.h"
#include "i2cslave.h"
extern volatile uint8_t I2CWrBuffer[BUFSIZE];
extern volatile uint8_t I2CRdBuffer[BUFSIZE];
extern volatile uint32_t I2CSlaveState;
extern volatile uint32_t I2CReadLength, I2CWriteLength;
extern volatile uint32_t RdIndex, WrIndex;
/*
 * From device to device, the I2C communication protocol may vary,
 * in the example below, the protocol uses repeated start to read
 * data from or
 * write to the device:
 * For master read: the sequence is: STA, Addr(W), offset, RE-
 * STA, Addr(r), data...STO
 * for master write: the sequence is: STA, Addr(W), offset, RE-
 * STA, Addr(w), data...STO
 * Thus, in state 8, the address is always WRITE. in state 10, the
 * address could
 * be READ or WRITE depending on the I2C command.
 */
```

```

/*****
** Function name:          I2C_IRQHandler
**
** Descriptions:         I2C interrupt handler, deal with master
mode only.
** parameters:           None
** Returned value:       None
*****/
voidI2C2_IRQHandler(void)
{
uint8_tStatValue;
// this handler deals with master read and master write only
StatValue = LPC_I2C2->STAT;
switch ( StatValue )
    {
        case 0x60:          /* An own SLA_W has been received. */
        case 0x68:
            RdIndex = 0;
            LPC_I2C2->CONSET = I2CONSET_AA;      /* assert ACK after
SLV_W is received */
            LPC_I2C2->CONCLR = I2CONCLR_SIC;
            I2CSlaveState = I2C_WR_STARTED;
            break;
        case 0x80:          /* data receive */
        case 0x90:
            if ( I2CSlaveState == I2C_WR_STARTED )
            {
                I2CRdBuffer[RdIndex++] = LPC_I2C2->DAT;
                LPC_I2C2->CONSET = I2CONSET_AA; /* assert ACK after
data is received */
            }
            else
            {
                LPC_I2C2->CONCLR = I2CONCLR_AAC; /* assert NACK */
            }
            LPC_I2C2->CONCLR = I2CONCLR_SIC;
            break;}

        case 0xA8:          /* An own SLA_R has been received. */
        case 0xB0:
            RdIndex = 0;
            LPC_I2C2->CONSET = I2CONSET_AA;      /* assert ACK after
SLV_R is received */
            LPC_I2C2->CONCLR = I2CONCLR_SIC;
            I2CSlaveState = I2C_RD_STARTED;
            WrIndex = I2CRdBuffer[0];          /* The 1st byte is the
index. */
            break;
    }
}

```

```

case 0xB8:                /* Data byte has been transmitted */
    case 0xC8:
        if ( I2CSlaveState == I2C_RD_STARTED )
        {
            LPC_I2C2->DAT = I2CRdBuffer[WriIndex+1];/* write the same
data back to master */
            WriIndex++;                                /* Need to
skip the index byte in RdBuffer */
            LPC_I2C2->CONSET = I2CONSET_AA;            /* assert ACK */
        }
        else
        {
            LPC_I2C2->CONCLR = I2CONCLR_AAC;          /* assert NACK
        }
        LPC_I2C2->CONCLR = I2CONCLR_SIC;
        break;

        case 0xC0:          // Data byte has been transmitted, NACK
        LPC_I2C2->CONCLR = I2CONCLR_AAC;    /* assert NACK */
        LPC_I2C2->CONCLR = I2CONCLR_SIC;
        I2CSlaveState = DATA_NACK;
        break;

        case 0xA0:          // Stop condition or repeated start has
        LPC_I2C2->CONSET = I2CONSET_AA;    /* been received,
assert ACK. */
        LPC_I2C2->CONCLR = I2CONCLR_SIC;
        I2CSlaveState = I2C_IDLE;
        break;

        default:
        LPC_I2C2->CONCLR = I2CONCLR_SIC;
        LPC_I2C2->CONSET = I2CONSET_I2EN | I2CONSET_SI;
        break;
    }
return;
}
/*****
** Function name:          I2CSlaveInit
**
** Descriptions:          Initialize I2C controller
**
** parameters:            I2c mode is either MASTER or SLAVE
** Returned value:        true or false, return false if the I2C
                           interrupt handler was not installed correctly
*****/

```

```

voidI2CSlave2Init(void )
{
    LPC_SC->PCONP |= (1 << 26);

    /* set PIO0.10 and PIO0.11 to I2C2 SDA and SCL */
    /* function to 10 on both SDA and SCL. */
    LPC_PINCON->PINSEL0&= ~( (0x03<<20) | (0x03<<22) );
    LPC_PINCON->PINSEL0 |= ( (0x02<<20) | (0x02<<22) );
    LPC_PINCON->PINMODE0&= ~( (0x03<<20) | (0x03<<22) );
    LPC_PINCON->PINMODE0 |= ( (0x02<<20) | (0x2<<22) ); /* No pull-up no
pull-down */
    LPC_PINCON->PINMODE_OD0 |= ( (0x01<<10) | (0x1<<11) );

    /*--- Clear flags ---*/
    LPC_I2C2->CONCLR = I2CONCLR_AAC | I2CONCLR_SIC | I2CONCLR_STAC
| I2CONCLR_I2ENC;

    LPC_I2C2->SCLL = I2SCLL_SCLL;
    LPC_I2C2->SCLH = I2SCLH_SCLH;

    LPC_I2C2->ADR0 = PCF8594_ADDR;
    I2CSlaveState = I2C_IDLE;

    /* Enable the I2C Interrupt */
    NVIC_EnableIRQ(I2C2_IRQn);

    LPC_I2C2->CONSET = I2CONSET_I2EN | I2CONSET_SI;
return;
}

/*****
End Of File
*****/

```

## Programa empleado para el uso del módulo I2C del AVR Butterfly como maestro

### USI\_TWI\_Master.c

```
/*
 *
 * File           USI_TWI_Master.c compiled with gcc
 * Date          Friday, 10/31/08           Boo!
 * Updated by    jkl
 * AppNote      : AVR310 - Using the USI module as a TWI
 * Master
 *
 * Extensively modified to provide complete I2C driver.
 *Notes:
 * - T4_TWI and T2_TWI delays are modified to work with
1MHz default clock
 * and now use hard code values. They would need
to change
 * for other clock rates. Refer to the Apps Note.
 * 12/17/08      Added USI_TWI_Start_Memory_Read Routine
 * -jkl
 * Note msg buffer will have slave adrs( with write bit
set) and memory adrs;
 * length should be these two bytes plus the
number of bytes to read.
 *****/
#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_Master.h"

unsignedcharUSI_TWI_Start_Transceiver_With_Data(unsignedchar*,un
signedchar);
unsignedcharUSI_TWI_Master_Transfer(unsignedchar);
unsignedcharUSI_TWI_Master_Stop(void);
unsignedcharUSI_TWI_Master_Start(void);

unionUSI_TWI_state
{
  unsignedcharerrorState;// Can reuse the TWI_state for error
states since it will not be needed if there is an error.
  struct
  {
    unsignedcharaddressMode:1;
    unsignedcharmasterWriteDataMode:1;
    unsignedcharmemReadMode :1;
    unsignedcharunused:5;
  };
}USI_TWI_state;
```

```

/*-----
  USI TWI single master initialization function
-----*/
void USI_TWI_Master_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 data register 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.
}

/*-----
  Use this function to get hold of the error message from the last
  transmission
-----*/
unsigned char USI_TWI_Get_State_Info(void)
{
  return(USI_TWI_state.errorState); // Return error state.
}

/*-----
  USI Random (memory) Read function. This function sets up for
  call
  to USI_TWI_Start_Transceiver_With_Data which does the work.
  Doesn't matter if read/write bit is set or cleared, it'll be
  set
  correctly in this function.

  The msgSize is passed to USI_TWI_Start_Transceiver_With_Data.

  Success or error code is returned. Error codes are defined in
  USI_TWI_Master.h
-----*/
unsigned char USI_TWI_Start_Random_Read(unsigned char*msg, unsigned c
har 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));
}
-----
*/

```

```

/*-----
  USI Normal Read / Write Function
  Transmit and receive function. LSB of first byte in buffer
  indicates if a read or write cycles is performed. If set a read
  operation is performed.

  Function generates (Repeated) Start Condition, sends address
  and
  R/W, Reads/Writes Data, and verifies/sends ACK.

  Success or error code is returned. Error codes are defined in
  USI_TWI_Master.h
  -----*/
unsignedcharUSI_TWI_Start_Read_Write(unsignedchar*msg,unsignedchar
msgSize)
{
    USI_TWI_state.errorState=0;           // Clears
    all mode bits also

    return(USI_TWI_Start_Transceiver_With_Data(msg,msgSize));
}
/*-----
  USI Transmit and receive function. LSB of first byte in buffer
  indicates if a read or write cycles is performed. If set a read
  operation is performed.

  Function generates (Repeated) Start Condition, sends address
  and
  R/W, Reads/Writes Data, and verifies/sends ACK.

  This function also handles Random Read function if the
  memReadMode
  bit is set. In that case, the function will:
  The address in memory will be the second
  byte and is written *without* sending a STOP.
  Then the Read bit is set (lsb of first byte), the byte count is
  adjusted (if needed), and the function function starts over by
  sending
  the slave address again and reading the data.

  Success or error code is returned. Error codes are defined in
  USI_TWI_Master.h
  -----*/
unsignedcharUSI_TWI_Start_Transceiver_With_Data(unsignedchar*msg
,unsignedcharmsgSize)
{
    unsignedcharconsttempUSISR_8bit=(1<<USISIF)|(1<<USIOIF)|(1<<USIP
E)|(1<<USIDC)// Prepare register value to: Clear flags, and
(0x0<<USICNT0);// set USI to shift 8 bits i.e. count 16 clock
edges.
    unsignedcharconsttempUSISR_1bit=(1<<USISIF)|(1<<USIOIF)|(1<<USIP
E)|(1<<USIDC)// Prepare register value to: Clear flags, and
(0xE<<USICNT0);
    // set USI to shift 1 bit i.e. count 2 clock edges.

```

```

        unsignedchar*savedMsg;
        unsignedcharsavedMsgSize;
//This clear must be done before calling this function so that
memReadMode can be specified.
// USI_TWI_state.errorState = 0;           // Clears
all mode bits also
USI_TWI_state.addressMode=TRUE;           // Always true
for first byte
#ifdef PARAM_VERIFICATION
if(msg>(unsignedchar*)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 masterRead or masterWrite operation.
{
USI_TWI_state.masterWriteDataMode=TRUE;
}
//   if (USI_TWI_state.memReadMode)
//   {
        savedMsg=msg;
        savedMsgSize=msgSize;
//   }
if(!USI_TWI_Master_Start())
{
return(FALSE);// Send a START condition on the TWI bus.
}
}

```

```

/*Write address and Read/write data */
do
{
/* If masterwrite cycle (or initial address transmission)*/
if(USI_TWI_state.addressMode||USI_TWI_state.masterWriteDataMode)
{
/* Write a byte */
PORT_USI&=~(1<<PIN_USI_SCL); // Pull SCL LOW.
USIDR=*(msg++); // Setup data.
USI_TWI_Master_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_Master_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 address
USI_TWI_state.errorState=0;
USI_TWI_state.addressMode=TRUE; // Now set up for
the Read cycle
msgSize=savedMsgSize; // Set byte count correctly

/*NOTE that the length should be slave adrs byte + #
bytes to read + 1 (gets decremented below)*/

if(!USI_TWI_Master_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.
}
}
}

```

```

/* ElsemasterRead cycle*/
else
{
/* Read a data byte */
DDR_USI&=~(1<<PIN_USI_SDA); // Enable SDA as input.
*(msg++)=USI_TWI_Master_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_Master_Transfer(tempUSISR_1bit); // Generate ACK/NACK.
}
}while(--msgSize); // Until all data sent/received.

if(!USI_TWI_Master_Stop())
{
return(FALSE); // Send a STOP condition on the TWI bus.
}
/* Transmission successfully completed*/
return(TRUE);
}
/*-----
Core function for shifting data in and out from the USI.
Data to be sent has to be placed into the USIDR prior to
calling
this function. Data read, will be return'ed from the function.
-----*/
unsignedchar USI_TWI_Master_Transfer(unsignedchar 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.
(1<<USICS1)|(0<<USICS0)|(1<<USICLK) | // Software
(1<<USITC); // Toggle Clock Port.
do
{
delay_us(T2_TWI);
USICR=temp; // Generate positive 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
}
/*-----
Function for generating a TWI Start Condition.
-----*/
unsigned char USI_TWI_Master_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);
}
/*-----
Function for generating a TWI Stop Condition. Used to release
the TWI bus.
-----*/
unsigned char USI_TWI_Master_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);
}

```

## Anexo 2

### Diagrama de configuración de pines del módulo LPCXpresso 1769

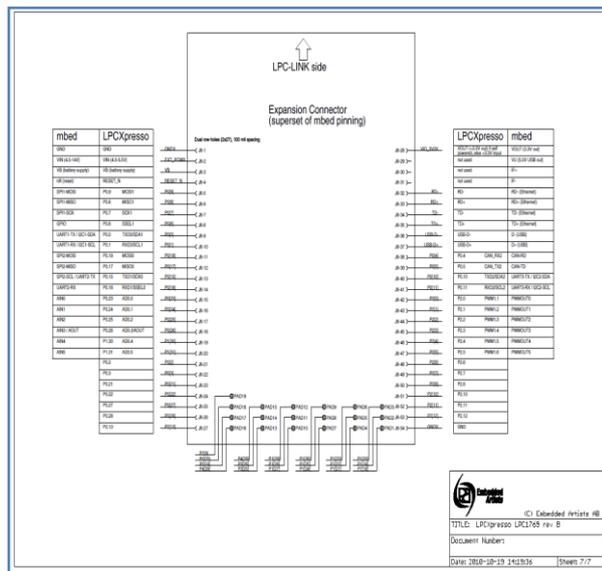


Figura A: Diagrama de pines LPCXpresso 1769 [16]

## BIBLIOGRAFÍA

[1] Storey, J. , How real electric motors work

[http://www.phys.unsw.edu.au/hsc/hsc/electric\\_motors6.html](http://www.phys.unsw.edu.au/hsc/hsc/electric_motors6.html)

Fecha de consulta febrero 2012.

[2] ATMEL Corporation, Fully Integrated BLDC Motor Control from the Signal Generation to the Full BLDC Motor Control Chain

[www.atmel.com/Images/doc4987.pdf](http://www.atmel.com/Images/doc4987.pdf)

Fecha de consulta febrero 2012.

[3] Blogger, Motores eléctricos

<http://eltekno.blogspot.com/2009/11/el-motor-electrico.html>

Fecha de consulta febrero 2012.

**[4] Varsani, A., Low Cost Brushless DC Motor Controller**

<http://innovexpo.itee.uq.edu.au/2003/exhibits/s363281/thesis.pdf>

Fecha de consulta febrero 2012.

**[5] Wiberg, J. , Controlling a BLDC motor in a Shift-by-wire System**

<http://es.scribd.com/doc/5154320/brushless-DC-motor>

Fecha de consulta febrero 2012.

**[6] Microchip, Brushless DC Motor Control Made Easy - Application**

**Note 857**

<http://ww1.microchip.com/downloads/en/AppNotes/00857B.pdf>

Fecha de consulta febrero 2012.

**[7] Escobar C., Martínez J., Téllez G., Control de un motor brushless DC con frenado regenerativo**

[www.javeriana.edu.co/biblos/tesis/ingenieria/tesis89.pdf](http://www.javeriana.edu.co/biblos/tesis/ingenieria/tesis89.pdf)

Fecha de consulta febrero 2012.

**[8] Oliveiros, J., Motores Brushless trifásicos**

<http://tegmei.blogspot.com/2011/08/brushless-son-motores-trifasicos-de.html>

Fecha de consulta febrero 2012

**[9] Máster Ingenieros S.A., Motores brushless**

[www.masteringenieros.com/master/Ficheros/File/motor.pdf](http://www.masteringenieros.com/master/Ficheros/File/motor.pdf)

Fecha de consulta febrero 2012

**[10] Motor Continua, Tutorial del motor DC**

<http://www.motorbrushless.es/documentos/tutorial-motor-dc.pdf>

Fecha de consulta febrero 2012

**[11] Fernández, A., El bus I2C**

<http://www.uco.es/~el1mofer/Docs/IntPerif/Bus%20I2C.pdf>

Fecha de consulta febrero 2012

**[12] NXP, LPCXpresso 1769 Getting Started**

<http://ics.nxp.com/support/documents/microcontrollers/pdf/lpcxpresso.getting.started.pdf>

Fecha de consulta febrero 2012.

**[13] Shao, J., Direct Back EMF Detection Method for Sensorless Brushless DC**

<http://scholar.lib.vt.edu/theses/available/etd-09152003-171904/unrestricted/T.pdf>

Fecha de consulta febrero 2012.

**[14] ATMEL Corporation, AVRProg User Guide.**

[www.atmel.com](http://www.atmel.com)

Fecha de consulta marzo 2012.

**[15] NXP, NXP about LPCXpresso,**

<http://ics.nxp.com/lpcxpresso/>

Fecha de consulta marzo 2012.

**[16] NXP, User Manual LPCXpresso 1769,**

[http://www.nxp.com/documents/user\\_manual/UM10360.pdf](http://www.nxp.com/documents/user_manual/UM10360.pdf)

Fecha de consulta marzo 2012.

**[17] ATMEL Corporation, ATmega169 Data Sheet Revision,**

<http://pdf1.alldatasheet.com/datasheet-pdf/view/313155/ATMEL/ATmega169PV.html>

Fecha de consulta marzo 2012.

**[18] ATMEL Corporation, AVR MCU Architecture,**

<http://www.atmel.com/Images/doc8155.pdf>

Fecha de consulta marzo 2012.

**[19] Roth, T., AVR, LibC Reference Manual,**

<http://www.nongnu.org/avr-libc/>

Fecha de consulta abril 2012.

**[20] Phillips Semiconductors, THE I2C-BUS SPECIFICATION V2.1,**

<http://i2c2p.twibright.com/spec/i2c.pdf>

Fecha de consulta abril 2012.

**[21] Fornaguera, A. , Portal, A. ,Trabajo de diploma - Automatización del pluviógrafo P-2 - intercambio de señales I2C,**

<http://es.scribd.com/doc/72744703/37/Intercambio%C2%A0de%C2%A0senal%20I2C>

Fecha de consulta abril 2012.

**[22] O'Flynn, Colin, Downloading Installing and Configuring WinAVR,**

<http://winavr.sourceforge.net/>

Fecha de consulta abril 2012.

**[23] AVRBeginners, The AVR TWI (I<sup>2</sup>C) Interface,**

<http://www.avrbeginners.net/architecture/twi/twi.html>

Fecha de consulta abril 2012.

**[24] AVR Projects, AVR Programation**

<http://winavr.scienceprog.com/example-avr-Projects>

Fecha de consulta abril 2012.