

Universidad Nacional de Cuyo  
Facultad de Ingeniería  
Ingeniería en Mecatrónica



## Microcontroladores y Electrónica de Potencia

Proyecto:

# Control de Eje con Microcontrolador y Memoria de Posiciones Aprendidas

Alumno da Costa Neto, Julio Joel

Profesor Ing. Iriarte, Eduardo

Mendoza, 27 de julio de 2017

# Índice

<b>1. Objetivo</b>	<b>1</b>
<b>2. Introducción</b>	<b>1</b>
2.1. Microcontrolador Atmel ATmega328 . . . . .	2
2.2. EEPROM Microchip 24LC256 . . . . .	3
2.3. Protocolo I <sup>2</sup> C . . . . .	4
2.3.1. Hardware . . . . .	4
2.3.2. Software . . . . .	5
<b>3. Diagrama de bloques</b>	<b>6</b>
<b>4. Diagrama de flujo</b>	<b>7</b>
4.1. Descripción del funcionamiento . . . . .	7
<b>5. Simulación</b>	<b>8</b>
<b>6. Montaje</b>	<b>10</b>
<b>7. Conclusiones y mejoras</b>	<b>11</b>
<b>Appendices</b>	<b>13</b>
<b>Apéndice A. Código del programa</b>	<b>13</b>

## **1. Objetivo**

En este informe se presentará el desarrollo de un control de eje que recibe por puerto serie las posiciones y guardalos en memoria para posterior ejecución. Para esto se ha hecho una integración de los contenidos aprendidos en la cátedra de Microcontroladores y Electrónica de Potencia, como programación de un microcontrolador, comunicación con periféricos, simulación y montaje de prototipos.

## **2. Introducción**

La popularización de los microcontroladores, junto con la caída de los costos de componentes electrónicos, baterías y motores, ha posibilitado el desarrollo de proyectos electrónicos con costo accesible.

La implementación de un control de eje implica conceptos de electrónica, computación y mecánica que, integrados de forma sincronizada, posibilitan la construcción de un sistema embarcado que percibe el ambiente a través de sus sensores e interactúa con el medio a través de actuadores.

Se utiliza un microcontrolador para gestionar el funcionamiento de todos los dispositivos conectados a él, y garantizar la sincronización necesaria para el funcionamiento de todos los elementos dentro de, por ejemplo, un controlador de eje.

Así, este proyecto presenta el desarrollo de un controlador de eje con memoria de posiciones aprendidas utilizando el microcontrolador ATmega 328, a través de un Arduino Uno, y la memoria Microchip 24LC256. La elección de este microcontrolador se debe principalmente a las siguientes características: bajo costo, facilidad de programación, confiabilidad y amplia gama de informaciones sobre su uso diseminadas en internet.

En este documento, vamos a presentar todos los detalles involucrados en la construcción del controlador, desde el microcontrolador elegido hasta las acciones implementadas. A continuación se describen los materiales y la metodología utilizada en este proyecto mostrando los conceptos base del controlador. El capítulo 3 describe el diagrama de bloques, en el capítulo 4 se detalla la implementación del software de control, en los capítulos 5 y 6 se presentan los resultados obtenidos a través de simulación y montaje del prototipo y por fin el capítulo 7 trae las conclusiones y mejoras propuestas. El apéndice A contiene el código fuente del programa desarrollado.

## 2.1. Microcontrolador Atmel ATmega328

El ATmega328 pertenece a la familia AVR de Atmel. Todos los modelos de esta familia comparten una arquitectura y un conjunto de instrucciones básicas, particularmente los grupos tinyAVR (microcontroladores ATtiny), megaAVR (los ATmega) y XMEGA (los Atxmega).

Los primeros modelos de Arduino usaban el ATmega8 con 8K de memoria Flash, que posteriormente fue sustituido por el ATMega168 con 16K de Flash y mayores capacidades de entrada y salida, y finalmente por el ATMega328 con 32K de Flash. La versión DIP de estos tres modelos comparten la misma pinza, pero el ATMega168 y ATMega328 permiten algunos usos diferentes de los pines. El Arduino Mega 2560 utiliza el ATMega2560 con 256K de Flash y una capacidad mucho mayor de entrada y salida.

La figura 1 muestra los principales bloques del microcontrolador.

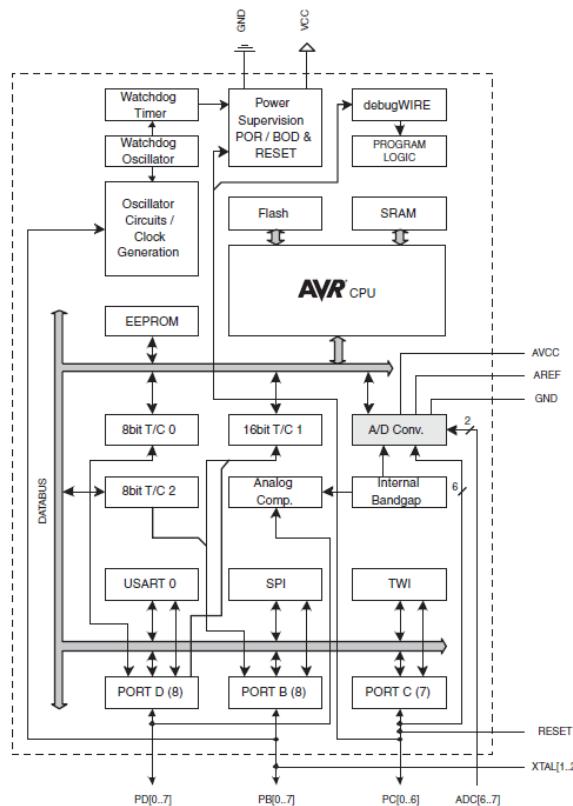


Figura 1: Bloques del microcontrolador ATmega328

Se pueden observar las conexiones separadas entre la CPU y las memorias Flash y SRAM. El uso de vías de datos separadas para el programa y los

datos es una característica de la arquitectura de Harvard. En la familia AVR las dos vías tienen un ancho de 8 bits y la memoria Flash se puede utilizar para almacenar datos constantes. Sin embargo, sólo se pueden ejecutar instrucciones almacenadas en Flash, no se puede ejecutar código que está en SRAM.

Al igual que otros microcontroladores AVR, el ATmega328 posee una memoria del tipo EEPROM, pero esta memoria está conectada a la conexión de los periféricos y por lo tanto no es accedida por las instrucciones normales de acceso a la memoria.

Además de la EEPROM se pueden ver los tres puertos de E/S digital (PORTs B, C y D), los tres temporizadores (TCx, dos de 8 bits y uno de 16 bits), el convertidor A/D, el comparador analógico y las interfaces seriales SPI, TWI (compatible con I<sup>2</sup>C) y USART.

## 2.2. EEPROM Microchip 24LC256

La memoria 24LC256 es una PROM eliminable eléctricamente en serie de 32K x 8 (256Kbit), capaz de funcionar a través de un amplio rango de voltaje (1,7V a 5,5V). Se ha desarrollado para aplicaciones avanzadas de baja potencia, como comunicaciones personales o adquisición de datos. Este dispositivo también tiene una capacidad de escritura de página de hasta 64 bytes de datos. Este dispositivo es capaz de lecturas aleatorias y secuenciales hasta el límite de 256K. Las líneas de direcciones funcionales permiten hasta ocho dispositivos en el mismo bus, para un espacio de direcciones de hasta 2Mbit.

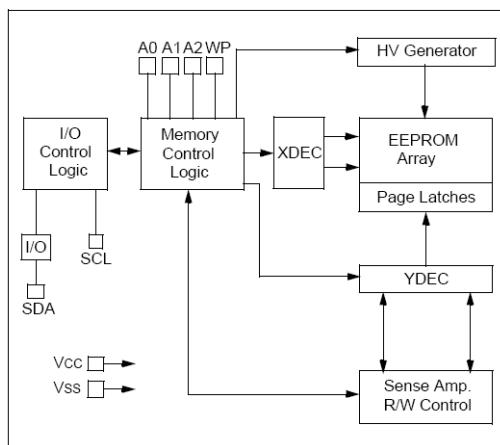


Figura 2: Bloques de la EEPROM 24LC256

## 2.3. Protocolo I<sup>2</sup>C

El protocolo I<sup>2</sup>C fue creado por Phillips en los años 80. Su objetivo era simplificar y estandarizar la forma de comunicación entre diferentes circuitos integrados, su nombre I<sup>2</sup>C significa "Inter-Integrated Circuits". El I<sup>2</sup>C es un protocolo para cortas distancias, típicamente dentro de la misma placa de circuito impreso, media velocidad, y necesita sólo dos hilos para comunicación (SDA-datos, y SCL-reloj).

Actualmente existen miles de componentes electrónicos que disponen de interfaz I<sup>2</sup>C, tales como relojes de tiempo real, circuitos de radio, sensores diversos, brújulas electrónicas, memorias, pantallas LCD, etc. En este protocolo todos los dispositivos pueden compartir el mismo bus SDA y SCL. El límite de número de dispositivos que se pueden integrar en un mismo bus es de 1008 dispositivos para direccionamiento de 10 bits y 112 para direccionamiento de 7 bits, prácticamente infinito para las aplicaciones típicas, donde son necesarios 3 o 4 circuitos conectados al bus. Las versiones "modernas" del protocolo se expandieron la velocidad para hasta 5MHz, pero en el caso de los microcontroladores ATmega328, nuestro foco principal, podemos utilizar como máximo la velocidad de 400KHz.

### 2.3.1. Hardware

Un bus I<sup>2</sup>C tiene dos señales, reloj y datos. El reloj es siempre generado por el maestro, normalmente el microcontrolador. Algunos esclavos pueden conectar a tierra la señal de reloj para solicitar más tiempo al maestro mientras procesa internamente la información. Es más común la situación donde los esclavos siguen fielmente el reloj generado por el maestro.

En un bus I<sup>2</sup>C, todos los controladores son "dreno abierto". Es decir, un controlador no puede colocar la línea en señal alta, sólo puede conectar la línea a tierra. Para que la línea asume el nivel lógico uno si es necesario una resistencia de "pull-up". Un resistor para la línea SCL y otro para la línea SDA. Estas resistencias se seleccionan de acuerdo con el número de dispositivos en el bus, y también la función de la distancia cubierta por el bus. El valor predeterminado utilizado es de  $4700\Omega$ . A medida que aumenta la longitud del bus, o el número de dispositivos, debe disminuir este valor.

La figura 3 muestra cómo se interconectan los dispositivos I<sup>2</sup>C.

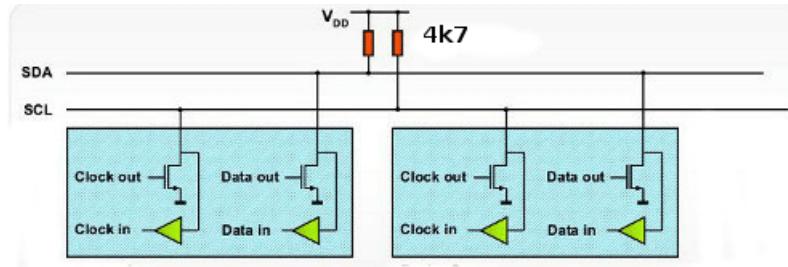


Figura 3: Conexión de dispositivos I<sup>2</sup>C

En el caso del ATmega328, los pines asociados al I<sup>2</sup>C son 27(SDA) y 28(SCL).

### 2.3.2. Software

El protocolo I<sup>2</sup>C tiene dos tipos de dispositivos: maestro y esclavo, el maestro es la unidad de control responsable de coordinar todos los periféricos. La línea SCL es responsable del reloj del bus, y la línea SDA por la transmisión de datos. En el estado neutro del bus I<sup>2</sup>C se mantienen el valor digital alto en ambas líneas de comunicación, para iniciar la comunicación, SDA es traído al valor digital bajo por el maestro. Para escribir datos en el bus, SCL pulsa, y cada pulso, el valor en SDA se lee como un bit, comenzando desde el MSB. Después de que SDA se traiga abajo, el maestro escribe la dirección del dispositivo que desea comunicarse, por ejemplo 0xC0, si el dispositivo existe, responderá como un ACK, un pulso en la línea SCL. Entonces comienza la transferencia de datos, el maestro escribe la dirección del registrador en el esclavo que él desea leer o escribir (R/W) y opera entonces, en secuencia, pudiendo leer/escribir uno o más registrador.

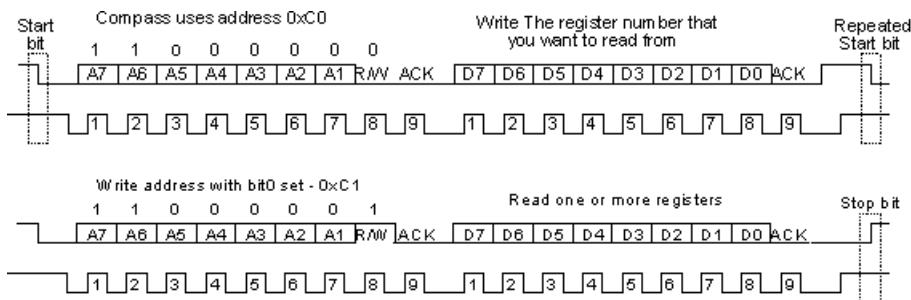


Figura 4: Comunicación de dispositivos I<sup>2</sup>C

### 3. Diagrama de bloques

En la figura 5 se puede ver los componentes más importantes del proyecto y a la vez la interacción entre ellos. El dispositivo principal es el Microcontrolador ATmega328, que comunicase con la PC e interactua con la memoria EEPROM a través del protocolo I<sup>2</sup>C, el sensor de fin de carrera, las salidas auxiliares, y también con el driver que comandará al motor PaP.

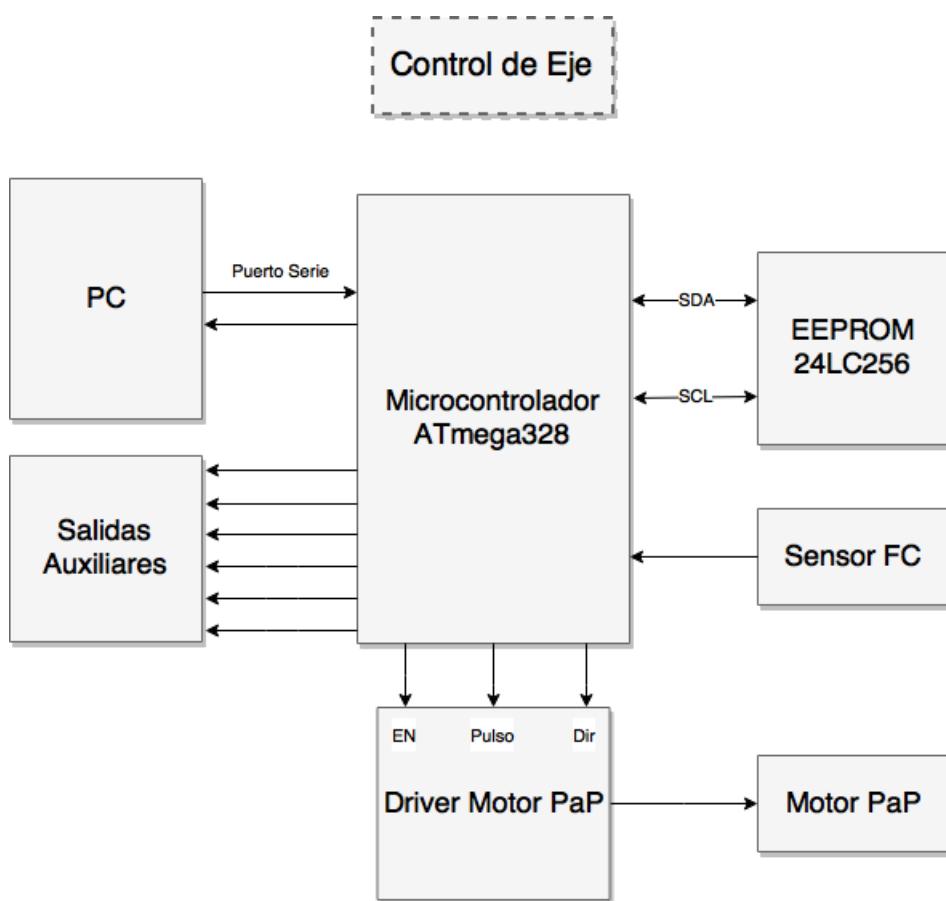


Figura 5: Diagrama de bloques

## 4. Diagrama de flujo

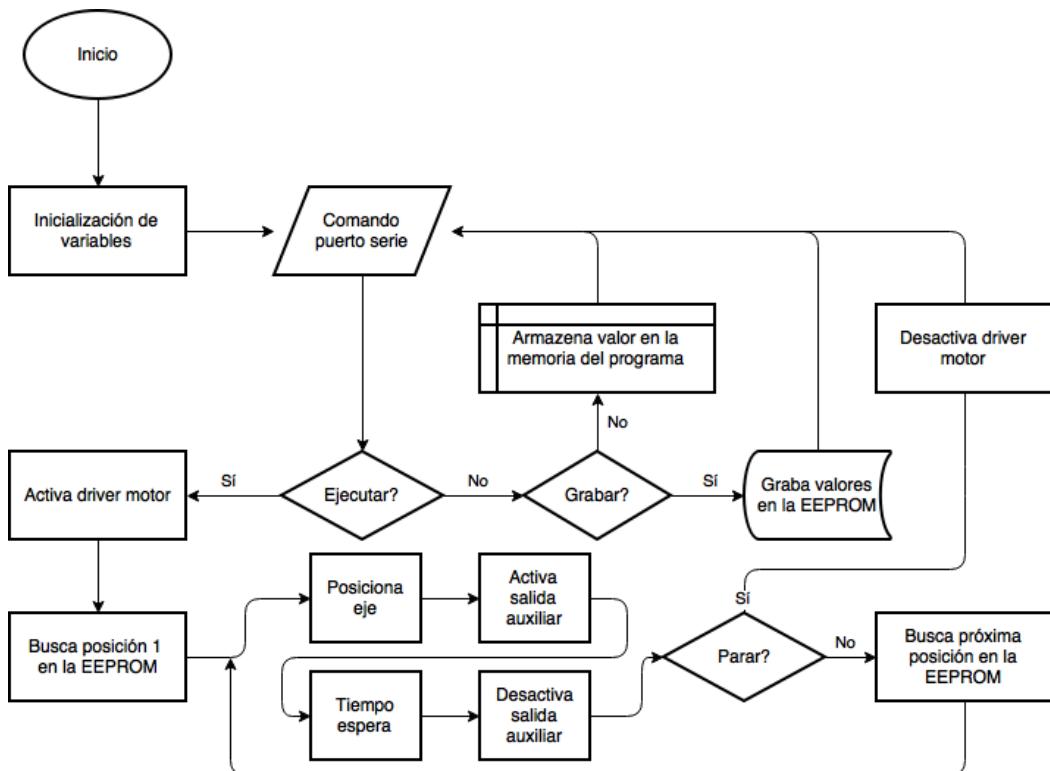


Figura 6: Diagrama de flujo

### 4.1. Descripción del funcionamiento

El programa se inicia con la declaración de variables y luego comienza un loop donde ejecuta diferentes tareas de acuerdo con el estado actual. En este paso el programa se interrumpe siempre cuando recibe un comando a través del puerto serie y lo interpreta, la lista de comandos interpretables y sus descripciones se puede ver en la tabla 1.

El estado inicial del programa es "desactivado", donde están inactivas todas las salidas del controlador. En este estado se pueden recibir e interpretar los comandos para borrar una o todas las etapas de la memoria EEPROM, crear nueva etapa en la memoria temporal del programa, asignar valores para las variables de la etapa en la memoria temporal, grabar la etapa y sus variables recién creadas en la memoria EEPROM e iniciar ejecución de movimientos.

Una vez que se recibe el comando para iniciar la ejecución de los movimientos, el estado del sistema cambia a "activado", se activa el driver del motor, y luego pasa al estado de "homing".

En el estado "homing", se envían pulsos al motor, con intervalo y dirección previamente determinados, hasta que el sensor de final de carrera sea accionado, y luego la dirección es invertida y se sigue enviando pulsos con un intervalo mayor hasta que el sensor deje de ser accionado. Al finalizar la maniobra de homing, el estado se cambia a "etapa".

El estado "etapa" es donde el controlador realiza la comunicación con la memoria EEPROM a través del protocolo I<sup>2</sup>C y busca las variables del paso actual, y luego cambia al estado "posicionamiento". En este estado el controlador inicia el envío de pulsos hasta que la posición deseada sea alcanzada, activa la salida auxiliar durante el tiempo de espera programado y luego regresa al estado "etapa" para leer las variables del paso siguiente.

A partir de ese momento el programa se ejecuta cíclicamente hasta que se interrumpe a través del comando "R" o encuentre un error en la lectura o interpretación de los datos en la memoria. Sólo en este estado se puede enviar el comando "E" para imprimir la etapa actual en ejecución.

Cuadro 1: Comandos aceptados por puerto serie

Inicio	Comando	Rango	Final	Descripción
	BT	-		Borra toda la memoria
	B	1-255		Borra la etapa especificada
	N	1-255		Crea nueva etapa
	P	0-9999		- Posición
	TP	0-9999		- Tiempo de pulso
:	TE	0-9999	:	- Tiempo de espera
	A	0-5		- Accionamiento auxiliar
	S	1-255		- Etapa siguiente
	G	-		Graba datos en la memoria
	R	-		Inicia/interrumpe ejecución
	E	-		Imprime etapa actual

## 5. Simulación

Se realizó una simulación en la computadora antes de montar los diferentes componentes, para poder analizar el programa antes de llevarlo a la práctica. Las figuras 7 y 8 ilustran la montaje en el simulador.

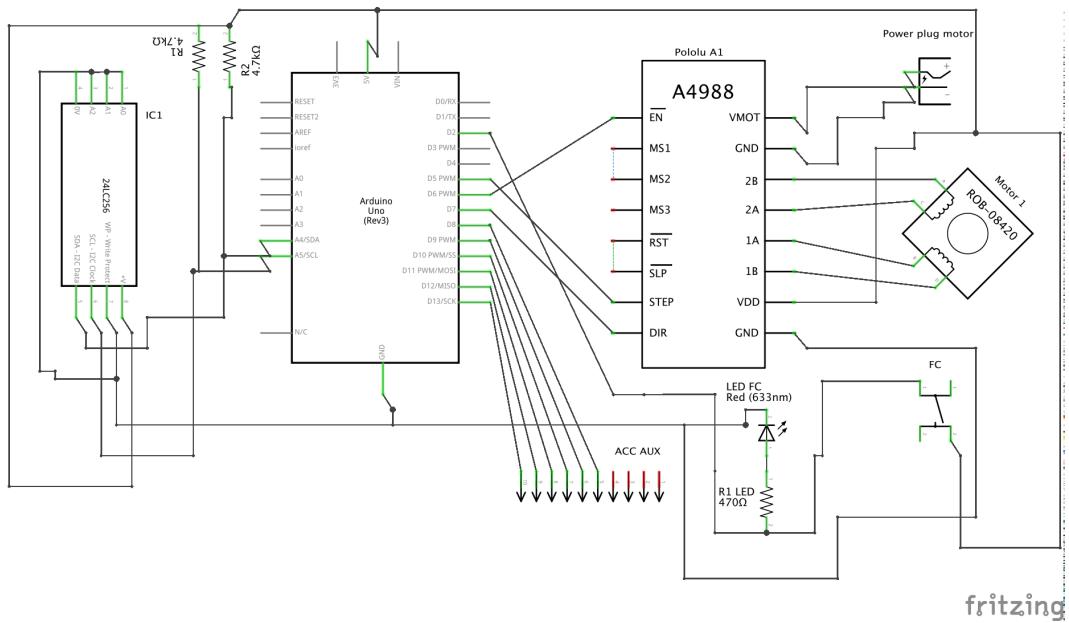


Figura 7: Simulación - Vista de esquemático

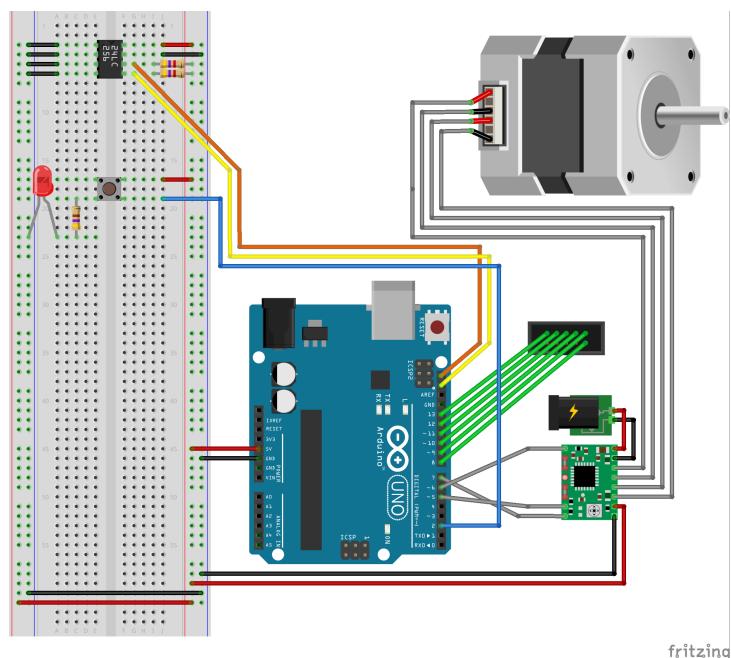


Figura 8: Simulación - Vista de protoboard

## 6. Montaje

Después de la simulación, se realizó el montaje de un prototipo funcional utilizando un Arduino Uno y una memoria EEPROM 24LC256. Para simular el sensor de fin de carrera y las salidas para el driver del motor, se utilizó un pushbutton y varios leds.

Como no se utilizó el driver y el motor para el montaje, el circuito pudo ser alimentado solamente a través del puerto USB de la computadora.

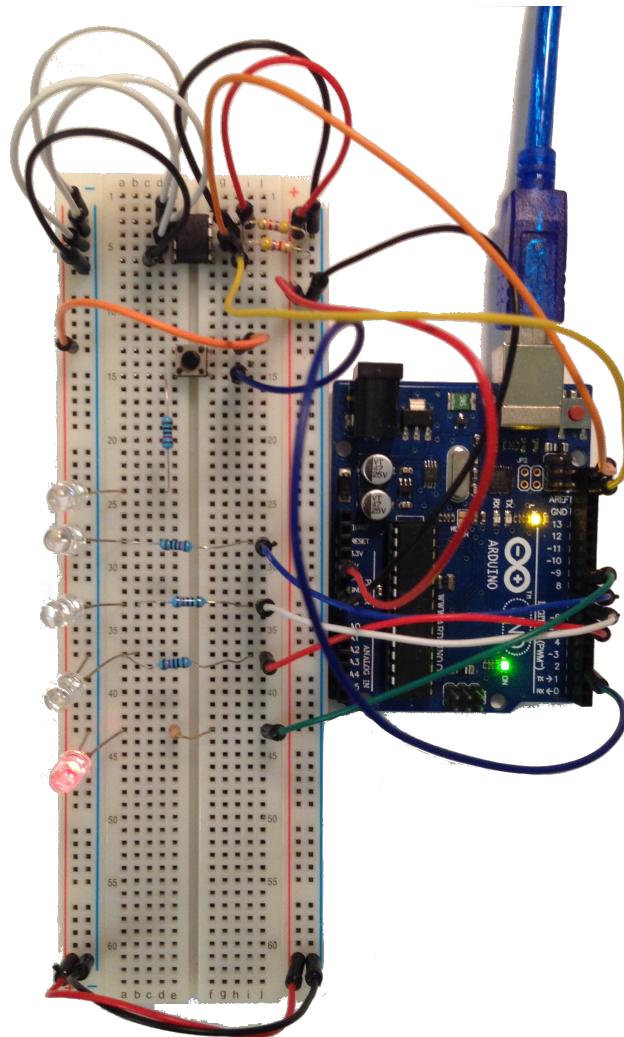


Figura 9: Montaje de componentes

## **7. Conclusiones y mejoras**

Al final del proyecto se pudo construir un prototipo funcional del controlador de eje. El programa desarrollado logra realizar todas las actividades propuestas y con un tiempo de respuesta satisfactorio. Sin embargo, permanece como actividades futuras realizar mejoras tanto en el software y en el hardware presentado.

Un ejemplo de mejora a desarrollar sería permitir el control de varios ejes en paralelo utilizando el mismo controlador y memoria para almacenar las etapas aprendidas. También sería interesante realizar el montaje y pruebas utilizando solamente el microcontrolador en lugar del Arduino, quedando así el proyecto más compacto y aún menos costoso.

El proyecto desarrollado permitió integrar diversos conceptos aprendidos durante las clases, como programación de microcontroladores utilizando interrupciones, UART, I<sup>2</sup>C, entre otros. También se aplicaron conceptos adquiridos sobre accionamiento de motores, sensores, lazos de control y máquinas de estados. Con eso, se logró consolidar los conocimientos repasados durante las clases y perfeccionar las líneas de raciocinio a través de las pruebas y ensayos realizados.

## Referencias

- [1] Ing. Iriarte, Eduardo. *Apuntes Cátedra “Microcontroladores y Electrónica de potencia”*. UNCuyo. 2017.
- [2] Atmel. Datasheet ATmega328P, 2016. [http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P\\_Datasheet.pdf](http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf) [Acceso en jul 2017].
- [3] Microchip. Datasheet 24LC256, 2013. <http://ww1.microchip.com/downloads/en/DeviceDoc/20001203U.pdf> [Acceso en jul 2017].

# Appendices

## A. Código del programa

```
1  /*
2   Proyecto Microcontroladores y Electronica de Potencia
3   Julio J. da Costa Neto
4   */
5
6 #define F_CPU 16000000
7 #define brate0 9600
8
9 #include <avr/interrupt.h>
10 #include <avr/io.h>
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <util/delay.h>
14 #include <util/twi.h>
15
16 //Variables posicion, posicion anterior, periodos y activado
17 volatile int8_t Act=-1;
18 volatile uint8_t Accion=0, Etapa=0, EtapaSeg=0;
19 volatile int16_t Pos=0, Pos_ant=0;
20 volatile uint16_t T=0, TP=0, TE=0;
21
22 //Define estados e registro
23 enum tEstado{D, A, h, E, p, P} estado;
24
25 //Define rango de las variables
26 #define T_MIN 1
27 #define P_MIN 0
28 #define E_MIN 1
29 #define B_MIN 0
30 #define T_MAX 9999
31 #define P_MAX 9999
32 #define E_MAX 255
33 #define B_MAX 5
34
35 #define LC256_DIR 0xA0
36 #define BYTES_ETAPA 10
37
38 #define I2C_READ 0x01
```

```

39 #define I2C_WRITE 0x00
40
41 #define F_SCL 100000 // SCL frequency
42 #define Prescaler 1
43 #define TWBR_val (((F_CPU / F_SCL) / Prescaler) - 16 ) / 2)
44
45 //Define variables para buffer
46 uint8_t indcom, dato[BYTES_ETAPA];
47 char comando[30];
48
49 void i2c_init(void)
50 {
51     TWBR = (uint8_t)TWBR_val;
52 }
53
54 uint8_t i2c_start(uint8_t address)
55 {
56     // reset TWI control register
57     TWCR = 0;
58     // transmit START condition
59     TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
60
61     // wait for end of transmission
62     while( !(TWCR & (1<<TWINT)) );
63
64     // check if the start condition was successfully transmitted
65     if((TWSR & 0xF8) != TW_START){ return 1; }
66
67     // load slave address into data register
68     TWDR = address;
69     // start transmission of address
70     TWCR = (1<<TWINT) | (1<<TWEN);
71     // wait for end of transmission
72     while( !(TWCR & (1<<TWINT)) );
73
74     // check if the device has acknowledged the READ / WRITE mode
75     uint8_t twst = TW_STATUS & 0xF8;
76
77     if ( (twst != TW_MT_SLA_ACK) && (twst != TW_MR_SLA_ACK) )
78         return 1;
79     return 0;
80 }
```

```

81 void i2c_stop(void)
82 {
83     // transmit STOP condition
84     TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
85     _delay_ms(5);
86 }
87
88 uint8_t i2c_write(uint8_t data)
89 {
90     // load data into data register
91     TWDR = data;
92     // start transmission of data
93     TWCR = (1<<TWINT) | (1<<TWEN);
94     // wait for end of transmission
95     while( !(TWCR & (1<<TWINT)) );
96
97     if( (TWSR & 0xF8) != TW_MT_DATA_ACK ){ return 1; }
98
99     return 0;
100 }
101
102 uint8_t i2c_read_ack(void)
103 {
104
105     // start TWI module and acknowledge data after reception
106     TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
107     // wait for end of transmission
108     while( !(TWCR & (1<<TWINT)) );
109     // return received data from TWDR
110     return TWDR;
111 }
112
113 uint8_t i2c_read_nack(void)
114 {
115
116     // start receiving without acknowledging reception
117     TWCR = (1<<TWINT) | (1<<TWEN);
118     // wait for end of transmission
119     while( !(TWCR & (1<<TWINT)) );
120     // return received data from TWDR
121     return TWDR;
122 }
123

```

```

124     uint8_t i2c_writeReg(uint8_t devaddr, uint16_t regaddr, uint8_t*
125                           data, uint16_t length)
126     {
127         if (i2c_start(devaddr | I2C_WRITE)) return 1;
128
129         i2c_write((regaddr >> 8) & 0xff);
130         i2c_write(regaddr & 0xff);
131
132         for (uint16_t i = 0; i < length; i++)
133         {
134             if (i2c_write(data[i])) return 1;
135         }
136
137         i2c_stop();
138
139         return 0;
140     }
141
142     uint8_t i2c_readReg(uint8_t devaddr, uint16_t regaddr, uint8_t*
143                           data, uint16_t length)
144     {
145         if (i2c_start(devaddr | I2C_WRITE)) return 1;
146
147         i2c_write((regaddr >> 8) & 0xff);
148         i2c_write(regaddr & 0xff);
149
150         if (i2c_start(devaddr | I2C_READ)) return 1;
151
152         for (uint16_t i = 0; i < (length-1); i++)
153         {
154             data[i] = i2c_read_ack();
155         }
156         data[(length-1)] = i2c_read_nack();
157
158         i2c_stop();
159
160         return 0;
161     }
162 //-----//
163 //      Definiciones siguientes son para usar getc y putc para E/S
164 //      de caracteres por UART
165 int mi_putc(char, FILE *stream), mi_getc(FILE *stream);

```

```

164 #define getc() mi_getc(&uart_io) // redefine la
165     primitiva de entrada como funcion recibir por UART
166 #define putc(x) mi_putc(x,&uart_io) // redefine la
167     primitiva de salida como funcion transmitir por UART
168 //      Declara un tipo stream de E/S
169 FILE uart_io = FDEV_SETUP_STREAM(mi_putc, mi_getc, _FDEV_SETUP_RW);
170 //          Inicializa UART
171 void mi_UART_Init(unsigned int ubrr)
172 {
173     UBRR0 = F_CPU/16/ubrr-1; // Configura baudrate.
174     UCSROB = (1<<RXENO)|(1<<TXENO); // Habilita
175         bits TXENO y RXENO
176     UCSROC = (1<<USBS0)|(3<<UCSZ00); // USBS0=1
177         2 bits stop, UCSZxx=3 8 bits
178 }
179 //          Pone caracter en UART. *stream es un parametro
180 // usado solo para igualar los parametros en stdio
181 int mi_putc(char c, FILE *stream)
182 {
183     while(!(UCSROA & (1<<UDRE0))); // Espera mientras el bit
184         UDRE0=0 (buffer de transmision ocupado)
185     UDR0 = c; // Cuando
186         se desocupa, UDR0 puede recibir el nuevo dato c a trasmisir
187     return 0;
188 }
189 //          Recibe caracter de la UART. *stream es un parametro
190 // usado solo para igualar los parametros en stdio
191 int mi_getc(FILE *stream)
192 {
193     while (!(UCSROA & (1<<RXCO))); // Espera mientras el bit
194         RXCO=0 (reception incompleta)
195     return UDR0; // Cuando
196         se completa, se lee UDR0
197 }
198 //-----
199 //          Funcion delay
200 void mi_delay(int ms) // Delay con argumento
201     variable
202 {
203     int t;
204     for(t=0;t<ms;t++)
205         _delay_ms(1);

```

```

195 }
196
197 uint8_t borrarTodo() {
198     printf(":BT...");
199     for (int i=0; i<500; i+=64) {
200         if (i2c_start(LC256_DIR) || i2c_write((i >> 8) & 0xff) ||
201             i2c_write(i & 0xff)) return 1;
202         for (int j; j<64; j++)
203             if (i2c_write(0)) return 1;
204         i2c_stop();
205     }
206     printf("ok\r\n");
207     return 0;
208 }
209 uint8_t borrarEtapa(uint8_t num) {
210     printf(":B%d...", num);
211     if (i2c_start(LC256_DIR) || i2c_write(((num-1)*BYTES_ETAPA)
212 >> 8) & 0xff) ||
213         i2c_write(((num-1)*BYTES_ETAPA) & 0xff)) return 1;
214     for (int j = 0; j < BYTES_ETAPA; j++) {
215         if (i2c_write(0)) return 1;
216     }
217     i2c_stop();
218     printf("ok\r\n");
219     return 0;
220 }
221 uint8_t grabar() {
222     if (Etapa==0 || TP==0 || TE==0 || EtapaSeg==0) return 1;
223
224     dato[0] = Etapa;
225     dato[1] = (Pos >> 8) & 0xff;
226     dato[2] = Pos & 0xff;
227     dato[3] = (TP >> 8) & 0xff;
228     dato[4] = TP & 0xff;
229     dato[5] = Accion;
230     dato[6] = (TE >> 8) & 0xff;
231     dato[7] = TE & 0xff;
232     dato[8] = EtapaSeg;
233     dato[9] = 0;
234
235     if (i2c_start(LC256_DIR) || i2c_write(((Etapa-1)*BYTES_ETAPA)
236 >> 8) & 0xff) ||
237         i2c_write(((Etapa-1)*BYTES_ETAPA) & 0xff)) return 1;

```

```

235     for (int j=0; j<BYTES_ETAPA; j++)
236         if (i2c_write(dato[j])) return 1;
237     i2c_stop();
238     _delay_ms(10);
239     return 0;
240 }
//           Analiza los elementos del buffer de recepcion
241 void InterpretaComando()
242 {
243     int aux=0;
244     switch(comando[0]) {                                // Analiza primer Byte
245         case 'T':                                     // T modifica tiempos
246             etapa
247             if (Etapa!=0) {
248                 switch (comando[1]) {
249                     case 'P':
250                         aux = atoi(&comando[2]);          //
251                         Convierte cadena decimal en entero
252                         if (aux >= T_MIN && aux <= T_MAX)
253                             TP = aux;                  //
254                         evalua rango, asigna a TP
255                         printf(":TP%d\r\n", TP);      // imprime
256                         tiempo informado
257                         break;
258                     case 'E':
259                         aux = atoi(&comando[2]);
260                         if (aux >= T_MIN && aux <= T_MAX)
261                             TE = aux;                  //
262                         evalua rango, asigna a TE
263                         printf(":TE%d\r\n", TE);      // imprime
264                         tiempo informado
265                         break;
266                     default:
267                         break;
268                 }
269             } else
270                 printf("Error, etapa=0.\r\n");
271             break;
272         case 'P':                                     // P
273             modifica posicion etapa
274             if (Etapa!=0) {
275                 aux = atoi(&comando[1]);

```

```

270             if (aux >= P_MIN && aux <= P_MAX)
271                 Pos = aux;                                     // 
272             evalua rango, asigna a P
273                 printf(":P%d\r\n", Pos);
274             } else
275                 printf("Error, etapa=0.\r\n");
276             break;
277             case 'S':                                         // S
278             modifica etapa siguiente
279                 if (Etapa!=0) {
280                     aux = atoi(&comando[1]);
281                     if (aux >= E_MIN && aux <= E_MAX)
282                         EtapaSeg = aux;
283                     // evalua rango, asigna a etapaSeg
284                         printf(":S%d\r\n", EtapaSeg);
285                     } else
286                         printf("Error, etapa=0.\r\n");
287                     break;
288                     case 'R':                                         // R
289                     modifica status sistema
290                     if (atoi(&comando[1]) == 1 && estado == D) {           //
291                     activa sistema
292                         Act = 1;
293                         printf(":R%d\r\n", Act);
294                         estado = A;
295                     } else if (atoi(&comando[1]) == -1) {                   //
296                     desactiva sistema
297                         Etapa = 0;
298                         Act = -1;
299                         printf(":R%d\r\n", Act);
300                         Pos = Pos_ant;
301                         estado = D;
302                     } else
303                         printf(":R%d\r\n", Act);
304                     break;
305                     case 'E':
306                         if (estado != D)
307                             printf(":E%d\r\n", Etapa);           // imprime etapa
308                         actual, si en ejecucion
309                         break;
310                         case 'B':                                // B borrar etapas
311                             if (estado == D) {
312                                 switch (comando[1]) {

```

```

306                         case 'T': // borrar todas
307                         etapas
308                             if (borrarTodo()) printf("Error BT\r\n");
309                             break;
310                         default: // borrar etapa X
311                             aux = atoi(&comando[1]); // 
312                             Convierte cadena decimal en entero
313                             if (aux >= E_MIN && aux <= E_MAX)
314                                 if (borrarEtapa(aux)) printf("Error B%d\
r\n", aux); // evalua rango, borra etapa informada
315                             break;
316                         } else
317                             printf("Error, en ejecucion.\r\n");
318                         break;
319                         case 'N': // N nueva etapa
320                             if (estado == D) {
321                                 aux = atoi(&comando[1]); // Convierte
322                                 cadena decimal en entero
323                                 if (aux >= E_MIN && aux <= E_MAX)
324                                     Etapa = aux; // evalua
325                                     rango, asigna a Etapa
326                                     printf(":N%d\r\n", Etapa);
327                                     Pos = 0; // borra
328                                     variables temporales
329                                     TP = 0;
330                                     Accion = 0;
331                                     TE = 0;
332                                     EtapaSeg = 0;
333                             } else
334                                 printf("Error, en ejecucion.\r\n");
335                             break;
336                         case 'A': // A accionamiento
337                         puerta X(solo en PORTB)
338                             if (Etapa!=0) {
339                                 aux = atoi(&comando[1]); // Convierte
340                                 cadena decimal en entero
341                                 if (aux >= B_MIN && aux <= B_MAX) {
342                                     Accion = aux; // evalua
343                                     rango, asigna a Accion
344                                     }
345                                     printf(":AB%d\r\n", Accion);
346                             } else

```

```

340         printf("Error, etapa=0.\r\n");
341         break;
342     case 'G': // G graba variables en
343         if (Etapa!=0)
344             if (grabar()) printf("Error grabando\r\n");
345             else printf(":G\r\n");
346         else
347             printf("Error, etapa=0.\r\n");
348             break;
349     default:
350         break;
351     }
352 }
353 // Programa main
354 int main()
355 {
356     int dif;
357
358     i2c_init();
359     mi_UART_Init(brate0); // Inicializa UART
360     stdout = stdin = &uart_io;
361
362     DDRD &= ~(1<<2);
363     DDRD |= (1<<5)|(1<<6)|(1<<7);
364     DDRB = 63; // Puertas 0 a 5 salida
365
366     // PORTD2 -> INTO FINAL DE CARRERA
367     // PORTD5 -> PULSO
368     // PORTD6 -> ENABLE
369     // PORTD7 -> DIRECCION
370     // PORTBX -> ACCIONAMIENTOS
371
372     EIMSK=(1<<INT0); // INTO interrupcion
373     EICRA=(3<<ISC00); // Flanco de subida
374     UCSROB |= (1<<RXCIE0); // Interrupcion Rx UART0
375
376     indcom=0;
377     estado=D; // Estado inicial
378     Pos=-1; // Eje no referenciado
379
380     sei(); // Interrupcion

```

```

global
382
383     while(1) {
384         switch (estado) {
385             case D:
386                 PORTD |= (1<<6); // desabilita
387                 driver
388                     PORTD &= ~(1<<7);
389                     PORTD &= ~(1<<5);
390                     break;
391             case A:
392                 PORTD &= ~(1<<6); // habilita
393                 driver
394                     estado=h;
395                     break;
396             case h:
397                 printf(":h\r\n");
398
399                 T=50;
400                 PORTD &= ~(1<<7); // cambio
401             direccion motor
402                 while (Pos!=0) { // pulso
403                     mientras no sea activada la int0 final de carrera
404                         if (estado!=h) break;
405                         PORTD |= (1<<5);
406                         mi_delay(T);
407                         PORTD &= ~(1<<5);
408                         mi_delay(T);
409                     }
410                     if (estado!=h) break;
411
412                     _delay_ms(100); // delay fijo
413                     T=100; // tiempo de
414             avance lento fijo
415                     PORTD |= (1<<7); // cambio
416             direccion motor
417                 while (PIND&(1<<PD2)) { // pulso
418                     mientras salga del fin de carrera (tension baja)
419                         if (estado!=h) break;
420                         PORTD |= (1<<5);
421                         mi_delay(T);
422                         PORTD &= ~(1<<5);
423                         mi_delay(T);

```

```

417     }
418     if (estado!=h) break;
419
420     printf(":H\r\n");
421     Etapa = 1;
422     estado=E;
423     break;
424 case E:
425     //busca info da etapa
426     i2c_readReg(LC256_DIR, (Etapa-1)*BYTES_ETAPA, dato,
427     BYTES_ETAPA);
428
429     if (Etapa!=dato[0]) {
430         printf("Erro etapa: %d %d\r\n", Etapa, dato[0])
431         ;
432         estado = D;
433     } else {
434
435         Etapa = dato[0];
436         Pos = ((uint16_t)dato[1] << 8) | dato[2];
437         TP = ((uint16_t)dato[3] << 8) | dato[4];
438         Accion = dato[5];
439         TE = ((uint16_t)dato[6] << 8) | dato[7];
440         EtapaSeg = dato[8];
441
442         estado = P; //inicia movimiento
443     }
444     break;
445 case P:
446     if (Pos!=Pos_ant)           // si posicion
447     nueva, cambia estado para 'realizando posicionamiento'
448     estado=p;
449     else {
450         PORTB |=(1<<Accion);      // inicia
451         accionamiento auxiliar
452         mi_delay(TE);            // delay tiempo
453         espera
454         PORTB &= ~(1<<Accion);    // termina
455         accionamiento auxiliar
456         Etapa = EtapaSeg;
457         estado = E;              // si ya esta
458         en la posicion, cambia busca proxima etapa
459     }

```

```

453         break;
454     case p:
455         T=TP;
456
457         if (Pos > Pos_ant) {
458             dif = 1;
459             PORTD |= (1<<7); // cambio
460         } else if (Pos < Pos_ant) {
461             dif = -1;
462             PORTD &= ~(1<<7); // cambio
463         }
464         direccion motor
465     } else
466         dif = 0;
467
468         while (Pos_ant != Pos) // pulso
469             si la posicion actual es distinta de la deseada
470                 PORTD |=(1<<5);
471                 mi_delay(T);
472                 PORTD &= ~(1<<5);
473                 mi_delay(T);
474
475                 if (estado!=p) break;
476                 Pos_ant += dif;
477             }
478             if (estado!=p) break;
479             estado=P;
480             break;
481         default:
482             break;
483     }
484 }
485 //-----//
486 //      Rutina de Servicio de Interrupcion de UART
487 ISR(USART_RX_vect)
488 {
489     char dato;
490     dato=getc();
491     switch (dato) {
492         case ':': // Delimitadores de inicio
493         case '/':
494             comando[0] = 0;

```

```

493     indcom = 0;           // Inicializa indice de buffer
494     de recepcion
495     break;
496     case ';':
497     case '\r':
498         comando[indcom] = 0; // coloca \0 luego del ultimo
        caracter recibido antes de \r
        InterpretaComando(); // Llama a funcion interprete
        de comandos
        break;
499     default:
500         comando[indcom++] = dato; // Guarda en elemento del
        buffer, si la direccion es correcta
        break;
501     }
502 }
503 //      Rutina de Servicio de Interrupcion de INTO(fin de carrera)
504 ISR(INT0_vect)
505 {
506     Pos_ant = 0;
507     Pos = 0;
508 }
509 //-----
510
511

```