Control de un Motor PAP con Arduino Mega usando AtmelStudio

Proyecto Integrador

MICROCONTROLADORES Y ELECTRÓNICA DE POTENCIA

Febrero, 2017

Jonathan Matías Obredor

Tabla de contenidos

Introducción

En el presente trabajo se aborda el control de un motor PAP a través de un Arduino Mega 2560, programando en C a través de Atmel Studio.

El objetivo principal es tener buena precisión en el control, manteniendo una buena sincronización entre paso y paso del motor, para lograr un funcionamiento suave y sin retrasos.

El funcionamiento general del software se explica a través de diagramas. El código está escrito de forma que al leerlo se vaya explicando a sí mismo, agregando comentarios donde se cree pertinente. El mismo se puede consultar en los anexos.

Esquema

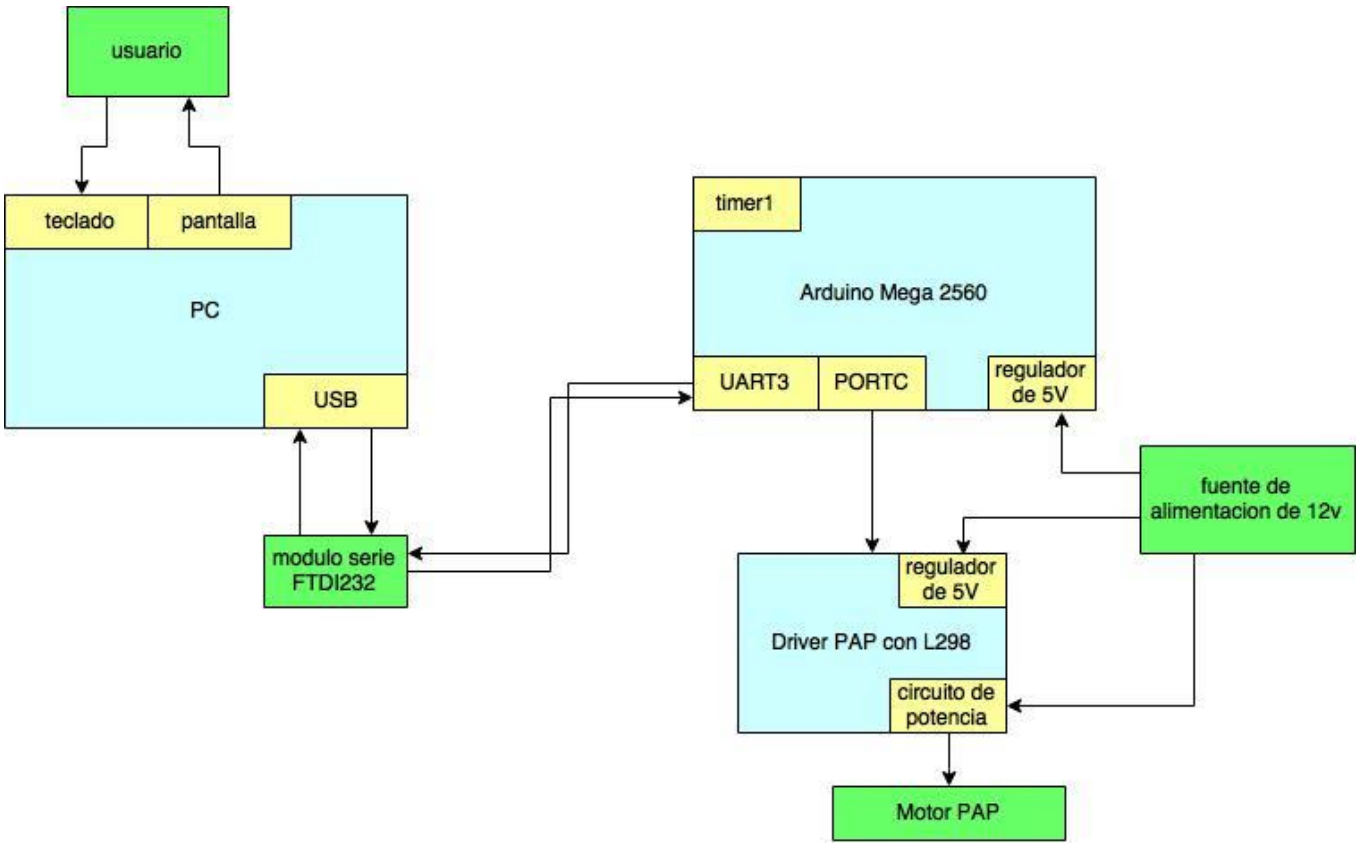
Descripción general

Este proyecto busca implementar el control de posición de un motor PAP Bipolar, con opción a cambiar a Unipolar. Para ello, el controlador debe ser capaz de recibir instrucciones del operador, en este caso una PC, que se comunica a través del puerto serie.

Lo que se plantea como objetivo es mantener el procesador libre, es decir sin actividad la mayor cantidad de tiempo posible, y de esta manera hacer que el motor reciba consignas de movimiento con buena sincronización, sin retrasos entre un paso y el siguiente para una velocidad dada.

Para no bloquear el procesador se plantea el uso de interrupciones, a saber, tres: llegada de dato por el puerto serie, dato enviado por el puerto serie, y el timer iguala al valor del registro de comparación.

Diagrama de Bloques



El Atmega 2560

Como se puede ver en la hoja de datos [10] del controlador:

AVR 8-Bit Microcontroller

- Advanced RISC Architecture

– 135 Powerful Instructions – Most Single Clock Cycle Execution
– 32 × 8 General Purpose Working Registers
– Fully Static Operation
– Up to 16 MIPS Throughput at 16MHz
– On-Chip 2-cycle Multiplier

- High Endurance Non-volatile Memory Segments

– 64K/128K/256KBytes of In-System Self-Programmable Flash
– 4Kbytes EEPROM
– 8Kbytes Internal SRAM
– Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
– Data retention: 20 years at 85°C/ 100 years at 25°C
– Optional Boot Code Section with Independent Lock Bits

- In-System Programming by On-chip Boot Program
- True Read-While-Write Operation

– Programming Lock for Software Security

- Endurance: Up to 64Kbytes Optional External Memory Space

- Atmel® QTouch® library support

– Capacitive touch buttons, sliders and wheels
– QTouch and QMatrix acquisition
– Up to 64 sense channels

- JTAG (IEEE®std. 1149.1 compliant) Interface

– Boundary-scan Capabilities According to the JTAG Standard
– Extensive On-chip Debug Support
– Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface

- Speed Grade:

– ATmega640V/ATmega1280V/ATmega1281V:

- 0 - 4MHz @ 1.8V - 5.5V, 0 - 8MHz @ 2.7V - 5.5V

– ATmega2560V/ATmega2561V:

- 0 - 2MHz @ 1.8V - 5.5V, 0 - 8MHz @ 2.7V - 5.5V

– ATmega640/ATmega1280/ATmega1281:

- 0 - 8MHz @ 2.7V - 5.5V, 0 - 16MHz @ 4.5V - 5.5V

– ATmega2560/ATmega2561:

- 0 - 16MHz @ 4.5V - 5.5V

- Peripheral Features

– Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
– Four 16-bit Timer/Counter with Separate Prescaler Compare- and Capture Mode
– Real Time Counter with Separate Oscillator
– Four 8-bit PWM Channels
– Six/Twelve PWM Channels with Programmable Resolution from 2 to 16 Bits (ATmega1281/2561, ATmega640/1280/2560)
– Output Compare Modulator
– 16-channel, 10-bit ADC
– Four Programmable Serial USART
– Master/Slave SPI Serial Interface
– Byte Oriented 2-wire Serial Interface
– Programmable Watchdog Timer with Separate On-chip Oscillator
– On-chip Analog Comparator
– Interrupt and Wake-up on Pin Change

- Special Microcontroller Features

– Power-on Reset and Programmable Brown-out Detection
– Internal Calibrated Oscillator
– External and Internal Interrupt Sources
– Six Sleep Modes: Idle, ADC Noise Reduction, Power-save, Power-down, Standby, and Extended Standby

- I/O and Packages

– 86 Programmable I/O Lines
– 64-pad QFN/MLF, 64-lead TQFP (ATmega1281/2561)
– 100-lead TQFP, 100-ball CBGA (ATmega640/1280/2560)
– RoHS/Fully Green

- Temperature Range:

–-40°C to 85°C Industrial

- Ultra-Low Power Consumption

– Active Mode: 1MHz, 1.8V: 500µA
– Power-down Mode: 0.1µA at 1.8V

Comunicaciones

Las comunicaciones en este proyecto se basan en el hardware de comunicaciones del Atmega, que recibe y envía datos por el puerto serie 3. Tanto para enviar como para recibir información se usan buffers, que van almacenando los bytes correspondientes. Luego, a medida que llegan las interrupciones, se van cargando los bytes almacenados en los buffers.

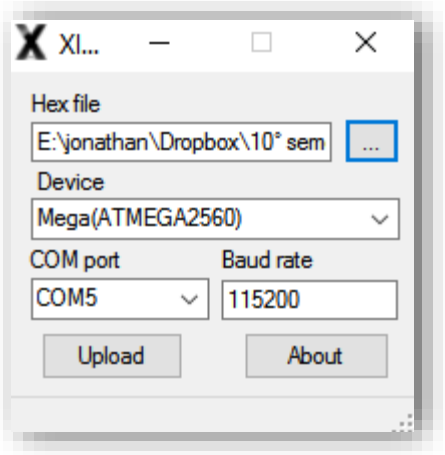
Se usa un pequeño protocolo al enviar datos que consiste en escribir ":" al comienzo de la trama y "\n" para terminarla.

Hay una función que "interpreta" los comandos recibidos, es decir analiza la trama y según los caracteres recibidos llama a otra función según corresponda.

Para acceder desde el USB de la pc al puerto serie del Arduino se propone usar un módulo ftdi232 (de bajo costo). Para que la pc reconozca dicho componente es necesario instalar los drivers correspondientes [5], cosa sencilla desde Windows 10. Es importante ejecutar el instalador como Administrador y seguir el manual de instalación [6].

Para grabar el programa compilado en el controlador se utiliza el programa *XLoader 1.0* [11], que permite utilizar el grabador incorporado en el Arduino habiendo seleccionado previamente el archivo hexadecimal.

Finalmente, para establecer una comunicación con el controlador se usa *PUTTY* [7], una herramienta de código libre que permite emular una terminal por el puerto serie que uno elija. Para saber a qué puerto COM está conectado el Arduino se puede usar el "administrador de dispositivos" de Windows.



Interrupciones para comunicación serie

La función handler para una interrupción en los AVR se llama "ISR()" y recibe como argumento un elemento vectorial, que es el nombre de la interrupción correspondiente

- ISR(USART3_TX_vect) {}
- ISR(USART3_RX_vect) {}

Para enviar datos, la interrupción que se utiliza es la que aparece cuando el buffer de envío por el puerto TX esta vacío. El registro para enviar y recibir datos se llama UDR3 (son dos registros diferentes con el mismo nombre), y la transmisión comienza luego de guardar nuevos datos en él.

- UDR3 = serialBuffer[serialReadPos]; //comenzar transmisión de un dato
- char dato=UDR3; //almacenar el dato recibido

Para inicializar la comunicación se escribió la siguiente función, en la que los registros UBRR3H y UBRR3L guardan una constante que se calcula en función de los siguientes parámetros:

- #define BUAD 9600 //baudios
- #define BRC ((F_CPU/16/BUAD) - 1)

```
void iniciarSerial(){
 UBRR3H = (BRC >> 8);
 UBRR3L =  BRC;
 //habilitar recepcion (1 << RXEN3)  ,  habilitar flag de recepción completa (1 << RXCIE3) ; transmisión y  transmisión completa
 UCSR3B = (1 << RXEN3)  | (1 << RXCIE3) | (1 << TXEN3)  | (1 << TXCIE3);
 //usart character size,  (1 << UCSZ31) | (1 << UCSZ30) para 8 bit ;
 //(0 <<UMSEL31) (0 <<UMSEL30) usart asincrona; (0 <<UPM31) |  (0 <<UPM30) sin paridad;    (0 <<USBS3) 1 bit de stop
 UCSR3C = (1 << UCSZ31) | (1 << UCSZ30) | (0 <<UMSEL31) | (0 <<UMSEL30) | (0 <<UPM31) |  (0 <<UPM30) | (0 <<USBS3);
}
```

Control

Para pasar la señal de control a la tensión y corriente adecuada para el motor se plantea el uso de un puente H basado en el integrado L298. El mismo tiene 4 pines que son las señales de control, y se usarán para alimentar las bobinas del PAP; y tiene 2 pines de *enable*, que activan las salidas de potencia. Desde el teclado, el usuario puede enviar comandos para modificar estos pines y así conectar o desconectar la alimentación a las bobinas.

El puerto C del microcontrolador está configurado como salida [4] y 4 de los pines son las señales que llegan al puente H. Se ha generado explícitamente la secuencia de pasos dentro del programa, y para mover al motor es necesario recorrer esta secuencia de manera ordenada [3].

El sistema puede recibir instrucciones de posición y velocidad en cualquier momento es decir, mientras el motor está en movimiento se pueden cambiar las consignas e instantáneamente se ajustan los parámetros.

Funcionamiento

Diagrama de Estados

A continuación se propone el siguiente diagrama general para facilitar la comprensión del código.

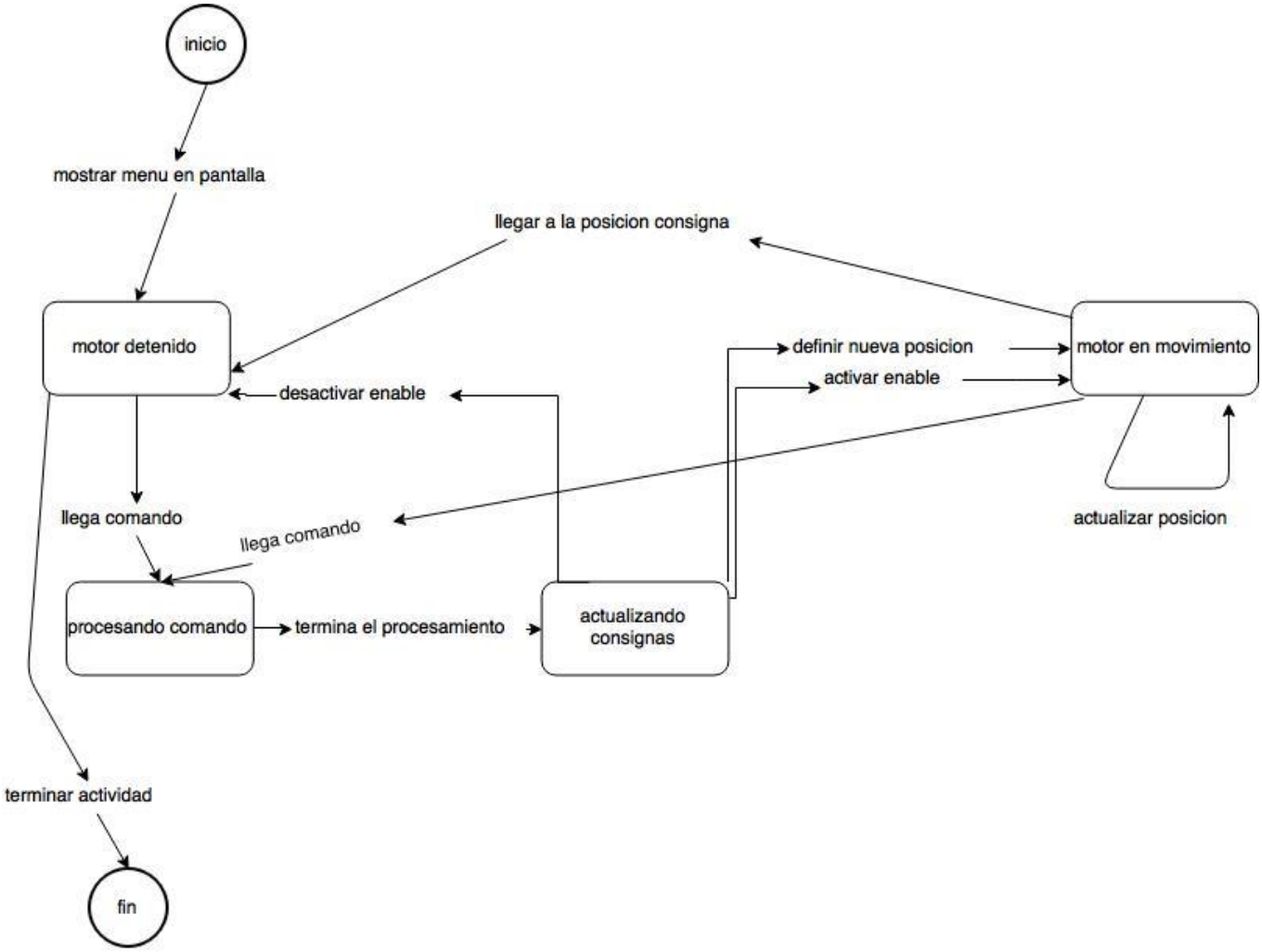
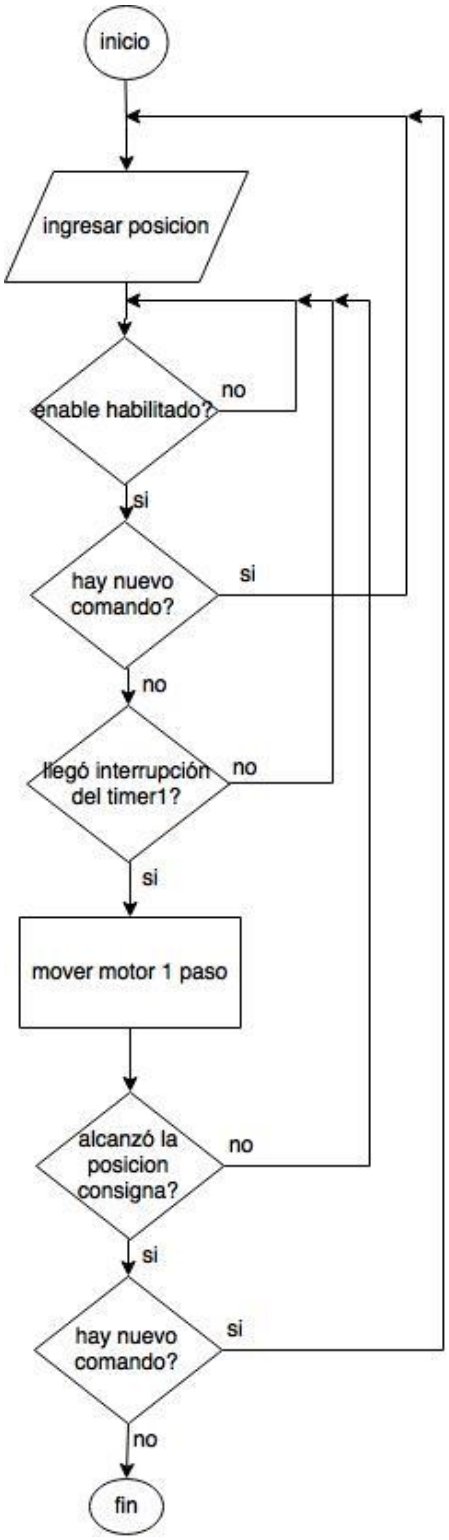


Diagrama de flujo

Aquí se puede ver una breve demostración de cómo funciona el mecanismo de movimiento del motor. Lo que se desea es mantener el procesador lo más desocupado posible, por esto, el motor avanza un paso cuando llega una interrupción del timer1. El resto del tiempo el procesador está en un bucle *while* vacío.



Software

Detalles del programa

El proyecto está escrito en C bajo el IDE Atmel Studio 7.0.1006.

El código fuente está dividido en archivos [9] para facilitar la lectura y manipulación:

- main.c
- configuracion.h
- comunicacion.h
- comunicacion.c
- motorPAP.h
- motorPAP.c

En el archivo main se encuentra la inicialización de los componentes y el infaltable bucle infinito. Por otra parte, dentro de configuracion.h hay algunos `#include` y definiciones de constantes solamente. En lo que refiere a los archivos de comunicación, ahí se encuentra todo lo relacionado a la transmisión y recepción de datos por puerto serie. Finalmente dentro de los fuentes motorPAP se pueden leer las rutinas de control para motores PAP, están relacionadas muchas de ellas a pines de salida del controlador.

Conclusión

Se consiguió de manera exitosa el movimiento del motor PAP con el programa desarrollado.

Dado que el microcontrolador que se usa tiene buena capacidad de procesamiento, se puede lograr que el motor no tenga retrasos en la secuencia de pasos y mantenga un movimiento suave y continuo.

El Arduino Mega facilita mucho la tarea del desarrollador, porque permite llevar adelante este tipo de prototipos muy rápidamente sin necesidad de una gran cantidad de accesorios, además de ser una herramienta robusta, debido a varias de sus características, como por ejemplo, protecciones contra cortocircuitos, etc.

Es importante mencionar que hoy en día existe una gran variedad de drivers para motores PAP, y la secuencia de paso o micro paso se puede resolver con estos drivers externos (por ejemplo A4988, DRV8825, drivers industriales, etc.). Algunos de estos permiten obtener un torque constante al usar micro pasos. Además, usar estos drivers mejora en algunos casos el rendimiento de los motores, dado que tienen control de corriente por PWM.

Se puede decir que el menú que se muestra en la pantalla es algo extenso, como así también algunos mensajes de advertencias y errores que el controlador envía a la pc. Esto debería replantearse para ser mínimo o casi inexistente porque el envío de mensajes es una carga importante para el procesador. Además, esta función es un accesorio a la función principal, por lo que podría prescindirse de ella.

Se espera en el futuro poder ampliar este proyecto para controlar 3 o 4 motores, para manipular una máquina CNC, estableciendo en la pc el intérprete de código G y que el Atmega 2560 solamente se ocupe de los motores y algunos sensores fines de carrera, etc.

Anexo: Código Fuente

Main

```
#include "configuracion.h"

//////////////////////////////////////////timer/////////////////////////////////////////////

//habilitar y configurar el timer 1
void iniciarTimer1(void)
{

//poner el timer en modo CTC: clear timer on compare
TCCR1B = (1 << WGM12);
//registro de comparacion, guarda el numero de ticks en los que se produce el match
//usar script para calcular!!
OCR1A = 19500;
//TIMSK es el registro de interrupciones; (1 << OCIE1A) activar interrupcion cuando se produzca match con
OCR1A //activar sei()!!!!!
TIMSK1 = (1 << OCIE1A);

//configurar el prescaler en 1024
//al configurar el prescaler, el timer empieza la cuenta
TCCR1B |= (1 << CS12) | (1 << CS10);
}

//interrupcion por CTC del timer 1
//cuando se produce la senal del timer1 indica que es el momento de mover el motor 1 paso
ISR(TIMER1_COMPA_vect)
{
        moverMotorX();
        //led que parpadea
        PORTB ^= (1 << PB0);
}


int main(void)
{
        //led testigo que parpadea
        DDRB = (1 << PB0);

        iniciarMotor();

        iniciarSerial();

        iniciarTimer1();



        //habilitar interrupciones
        sei();



        serialWrite("\n\rHOLA!\n\r");
        _delay_ms(1000);

        mostrarAyuda();

    while (1)
    {

    }
}
```

configuracion.h

```
#ifndef CONFIGURACION_H_
#define CONFIGURACION_H_

#include <avr/io.h>
#include <avr/interrupt.h>
#include <string.h>
#include <stdlib.h>

#define F_CPU 16000000

#include <util/delay.h>


#include "motorPAP.h"
#include "comunicacion.h"



#endif /* CONFIGURACION_H_ */
```

comunicacion.h

```
#ifndef COMUNICACION_H_
#define COMUNICACION_H_


#include "configuracion.h"

#define BUAD  9600 //baudios
#define BRC          ((F_CPU/16/BUAD) - 1)
#define  RX_BUFFER_SIZE     128
#define  TX_BUFFER_SIZE     128
#define  LONGI_BUF   128    // Constante para definir longitud del buffer de recepción

void iniciarSerial();


void appendSerial(char c);
void serialWrite(char c[]);
void interpretaComando();
unsigned int StringADecimal(int k);
void EnviarIntComoString();
void mostrarAyuda();


#endif /* COMUNICACION_H_ */
```

comunicacion.c

```
#include "comunicacion.h"

char rxBuffer[RX_BUFFER_SIZE];
uint8_t rxReadPos = 0;
uint8_t rxWritePos = 0;
char serialBuffer[TX_BUFFER_SIZE];
uint8_t serialReadPos = 0;
uint8_t serialWritePos = 0;
char comando[LONGI_BUF];         // array para recibir las cadenas de caracteres de los comandos
unsigned int indCom;     //indice para apuntar a los elementos dentro del array comando[]
unsigned int HayComando; // -Flag para indicar que se hay recepción de mensaje en curso (para evitar
interpretar tramas incompletas



 //rutina que prepara el puerto serie numero 3 para enviar y recibir datos
void iniciarSerial()
{
        //guardar el valor de ubrr en los registros adecuados
        UBRR3H = (BRC >> 8);
        UBRR3L =  BRC;
        //habilitar recepcion (1 << RXEN3)  ,  habilitar flag de recepcion completa (1 << RXCIE3) ;
transmision y  transmision completa
        UCSR3B = (1 << RXEN3)  | (1 << RXCIE3) | (1 << TXEN3)  | (1 << TXCIE3);
        //usart character size,  (1 << UCSZ31) | (1 << UCSZ30) para 8 bit ;(0 <<UMSEL31) (0 <<UMSEL30)
usart asincrona; (0 <<UPM31) |  (0 <<UPM30) sin paridad;    (0 <<USBS3) 1 bit de stop
        UCSR3C = (1 << UCSZ31) | (1 << UCSZ30) | (0 <<UMSEL31) | (0 <<UMSEL30) | (0 <<UPM31) |  (0
<<UPM30) | (0 <<USBS3) ;
}
//////////////////////////////recepcion serial//////////////////////////////////////////////////////////////

// Toma el caracter recibido y lo evalúa para llenar el buffer comando[]
//cuando se termina de recibir un comando llama a la funcion interpretaComando
ISR(USART3_RX_vect)
{
        char dato=UDR3; //almacenar el dato recibido
        switch(dato)
        {
                case ':':      //si es delimitador de inicio
                indCom=0;        //inicializa contador i
                HayComando=1;      //y activa bandera que indica que hay comando en curso
                break;
                case 13:       //si es delimitador de final
                if (HayComando==1)    // si habia comando en curso
                {
                        comando[indCom]=0;  //termina cadena de comando con caracter null
                        interpretaComando();  //va a interpretar el comando
                        HayComando=0;         //desactiva bandera de comando en curso
                        serialWrite("ok\n\r");
                }
                break;
                default:
                if (indCom<LONGI_BUF)    //si contador menor que longitud del buffer
                {
                        comando[indCom]=dato;    //pone caracter en cadena
                        indCom++;                //incrementa contador i
                }
                else
                {
                        serialWrite("err:buffer lleno, reiniciar\n\r");
                }
                break;
        }
}
```

```
//////////////////////////////transmision
serial////////////////////////////////////////////////////////////


//funcion que sirve para enviar datos por el puerto serie
//coloca todos los bytes que se desean enviar en un buffer: serialBuffer
void serialWrite(char c[])
{
        for(uint8_t i = 0; i < strlen(c); i++)
        {
                appendSerial(c[i]);
        }

        if(UCSR3A & (1 << UDRE3)) //verificar que el buffer de transmision este vacio
        {
                UDR3 = 0;
        }
}

//guarda un byte en el buffer y mantiene actualizado el indice para recorrerlo
void appendSerial(char c)
{
        serialBuffer[serialWritePos] = c;
        serialWritePos++;

        if(serialWritePos >= TX_BUFFER_SIZE)
        {
                serialWritePos = 0;
        }
}

//interrupcion por buffer de envio vacio
//envia el proximo dato que esta en el buffer por el puerto serie y actualiza el indice para recorrerlo
ISR(USART3_TX_vect)
{
        if(serialReadPos != serialWritePos)
        {
                UDR3 = serialBuffer[serialReadPos]; //no verifica si el buffer esta vacio porque esta
funcion la llama la interrupcion de transmision completa
                serialReadPos++;

                if(serialReadPos >= TX_BUFFER_SIZE)
                {
                        serialReadPos=0;
                }
        }
}

//analiza la instruccion guardada en el arreglo comando[] y llama a una funcion, segun corresponda
void interpretaComando()
{
        unsigned int auxInterpreta=2;
        switch(comando[0])
        {
                case 'x':
                case 'X':
                if (comando[1]=='?')
                {
                        serialWrite("x:");
                        EnviarIntComoString(getPosicionActualX());
                        serialWrite("\n\r");
                }
                else
                {
                        auxInterpreta=StringADecimal(1);
                        setConsignaX(auxInterpreta);
                }
```

```
                break;
                case 'v':
                case 'V':
                if (comando[1]=='?')
                {
                        serialWrite("v:");
                        EnviarIntComoString(getVelocidadX());
                        serialWrite("\n\r");
                }
                else
                {
                        auxInterpreta=StringADecimal(1);
                        if(auxInterpreta<60000) setVelocidadX(auxInterpreta);
                }
                break;
                case 'E':
                case 'e':
                setEnableMotor(StringADecimal(1));
                break;
                case 'D':
                case 'd':
                verEjemplo();
                break;
                default:
                serialWrite("err com") ;
                break;
        }
}

//convierte los valores de enteros a string y luego los envia por el puerto serie
void EnviarIntComoString(int dato){
        char buffer[20];
        itoa(dato,buffer,10);
        serialWrite(buffer);
}


//Función auxiliar que devuelve el valor numérico de una cadena decimal a partir del elemento k hasta
encontrar el caracter null (similar a atoi o atol)
unsigned int StringADecimal(int k)
{
        unsigned int aux16=0;
        while(comando[k]!=0)  // después de la cadena decimal hay un caracter null (ASCII 0)
        {
                aux16=aux16*10+comando[k]-'0'; //'0' es 48 en decimal o 0x30 en hexa
                k++;
        }
        return aux16;
}

//muestra un pequeno menu de opciones en la pantalla
void mostrarAyuda(){
        serialWrite("proyecto de control para motor PAP Jonathan Obredor 2017\n\r");
        _delay_ms(100);
        serialWrite("iniciar mensaje con ':'\n\rpara:                    enviar:\n\r");
        _delay_ms(100);
        serialWrite("consultar v___________________:v?\n\rasignar v_____________:v[valor]\n\r");
        _delay_ms(100);
        serialWrite("consultar x___________________:x?\n\rasignar x_____________:x[valor]\n\r");
        _delay_ms(100);
        serialWrite("activar motor(enable)_________:e1\n\rdesactivar motor_____________:e0\n\r");
        serialWrite("ver ejemplo                   :d\n\r");
}
```

motorPAP.h

```
#ifndef MOTORPAP_H_
#define MOTORPAP_H_

#include "configuracion.h"


#define enableX PB1



void iniciarMotor();
int setEnableMotor(int enable);
int getEnableMotor();
int actualizarBobinas();
void avanzaPaso();
void retrocedePaso();
void moverMotorX();
void setVelocidadX(int velo);
int getVelocidadX();
void setConsignaX(int consig);
int getPosicionActualX();
void verEjemplo();


#endif /* MOTORPAP_H_ */
```

motorPAP.c

```
#include "motorPAP.h"

volatile int ConsignaX;                 // variable consigna de posición (se actualizará a través de
mensaje por UART
int PosicionActualX;     // variable posición actual. Refleja la posición actual del motor, se incrementa
o decrementa) en cada pulso enviado al driver.
volatile float rpmX;             // tiempo de retardo entre pulsos enviados al driver.

int DireccionX;
int IndicePasoX;
int PosicionOkX;
int Xmax;
int HalfStep;
int MotorBipolar;
float PasosPorVuelta;
//establecer valores iniciales para algunas constantes y adoptar caracteristicas del motor PAP
void iniciarMotor(){
        //establecer pines del motor como salida
        //usar pines 37, 36,35,34
        DDRC = 0b11111111;
        //poner como salida el pin enable
        DDRB ^=(1 << PB1);

        setEnableMotor(1);
        PasosPorVuelta=48;
        setVelocidadX(100);

        MotorBipolar=1;
        HalfStep=1;
        IndicePasoX=0;

        Xmax=10000;
        PosicionActualX=0;                  //Supone posactual=0
        ConsignaX=0;                        //arranca en reposo

        //TODO:escribir funcion de homing
}

//colocar el pin de enable en el valor que recibe como parametro
int setEnableMotor(int enable){
        if (enable==1)
        {
                PORTB= (1<<enableX);
                return 0;
        }
        else if(enable==0)
        {
                PORTB= (0<<enableX);
                return 0;
        }
        else{
                serialWrite("err enableMotor");
                return -1;
        }
}

//devolver el valor del pin de enable
int getEnableMotor(){
        if(PINB & (1<<PINB1)){
                return 1;
        }
        return 0;
}

//configura los pines de salida para establecer la secuencia de pasos
//tiene 8 pasos para permitir half step
//si se desea full step deberia recorrerse de dos en dos
```

```
int actualizarBobinas(){

        switch (IndicePasoX){
                case 1:
                PORTC = 0b00000001;// PASO 1
                break;
                case 2:
                PORTC = 0b00000101;// PASO 2
                break;
                case 3:
                PORTC = 0b00000100;// PASO 3
                break;
                case 4:
                PORTC = 0b00000110;// PASO 4
                break;
                case 5:
                PORTC = 0b00000010;// PASO 5
                break;
                case 6:
                PORTC = 0b00001010;// PASO 6
                break;
                case 7:
                PORTC = 0b00001000;// PASO 7
                break;
                case 8:
                PORTC = 0b00001001;// PASO 8
                break;
                default:
                serialWrite("err bobina");
                return 0;
        }
        return 1;
}
//actualizar el indice del paso del motor y efectuar el movimiento
void avanzaPaso(){
        if(HalfStep){
                if (IndicePasoX==8){
                        IndicePasoX=1;
                }
                else{
                        IndicePasoX++;
                }

        }
        else{ //fullstep
                //TODO:
        }
        actualizarBobinas();
}
//funcion identica que avanzaPaso pero en sentido contrario
void retrocedePaso(){
        if(HalfStep){
                if (IndicePasoX==1){
                        IndicePasoX=8;
                }
                else{
                        IndicePasoX--;
                }

        }
        else{ //fullstep
                //TODO:
        }
        actualizarBobinas();
}

// llevar el motor hasta la posicion consigna
//esta funcion la llama la interrupcion del timer
//cuando llega al destino pone una variable en alto
```

```
void moverMotorX(){
	if (getEnableMotor())
	{
		if(PosicionActualX==ConsignaX)
		{
			PosicionOkX=1;
		}
		else if (PosicionActualX<ConsignaX && ConsignaX<Xmax)
		{
			PosicionOkX=0;
			DireccionX=1;
			avanzaPaso();
			PosicionActualX++;
//			serialWrite("x:");
//			EnviarIntComoString(PosicionActualX);
//			serialWrite("\n\r");
		}
		else if(ConsignaX>=0)
		{
			PosicionOkX=0;
			DireccionX=0;		// incrementa o decrementa en 1
			retrocedePaso();
			PosicionActualX--;
//			serialWrite("x:");
//			EnviarIntComoString(PosicionActualX);
//			serialWrite("\n\r");
		}
	}

}
//establecer la velocidad del eje del motor en rpm
void setVelocidadX(int velo){
	rpmX=velo;
	float SegundosPorPasoX=60/(PasosPorVuelta*rpmX); //verificar!!!
	float ticksNecesarios=(F_CPU/1024)*SegundosPorPasoX;
	OCR1A=ticksNecesarios;
	//serialWrite("ticks necesarios:");
	//EnviarIntComoString(ticksNecesarios);
}
int getVelocidadX(){
	return rpmX;
}

void setConsignaX(int consig){
	if (0<=consig && consig<Xmax)
		ConsignaX=consig;
	else
		serialWrite("err setConsigna excede lim\n\r");
}
int getPosicionActualX(){
	return PosicionActualX;
}
void verEjemplo(){
//	serialWrite("EJEMPLO");
//	setVelocidadX(200);
//	_delay_ms(200);
//	for (int i=0;i<10;i++)
//	{
//		setConsignaX(2000);
//		while(PosicionActualX!=ConsignaX){}
//
//		setConsignaX(0);
//		while(PosicionActualX!=ConsignaX){}
//	}
//	serialWrite("listo");
}
```

Anexo: Pinout y Mapeo del Arduino

A continuación se puede ver el Pinout o diagrama de pines del Arduino [1].

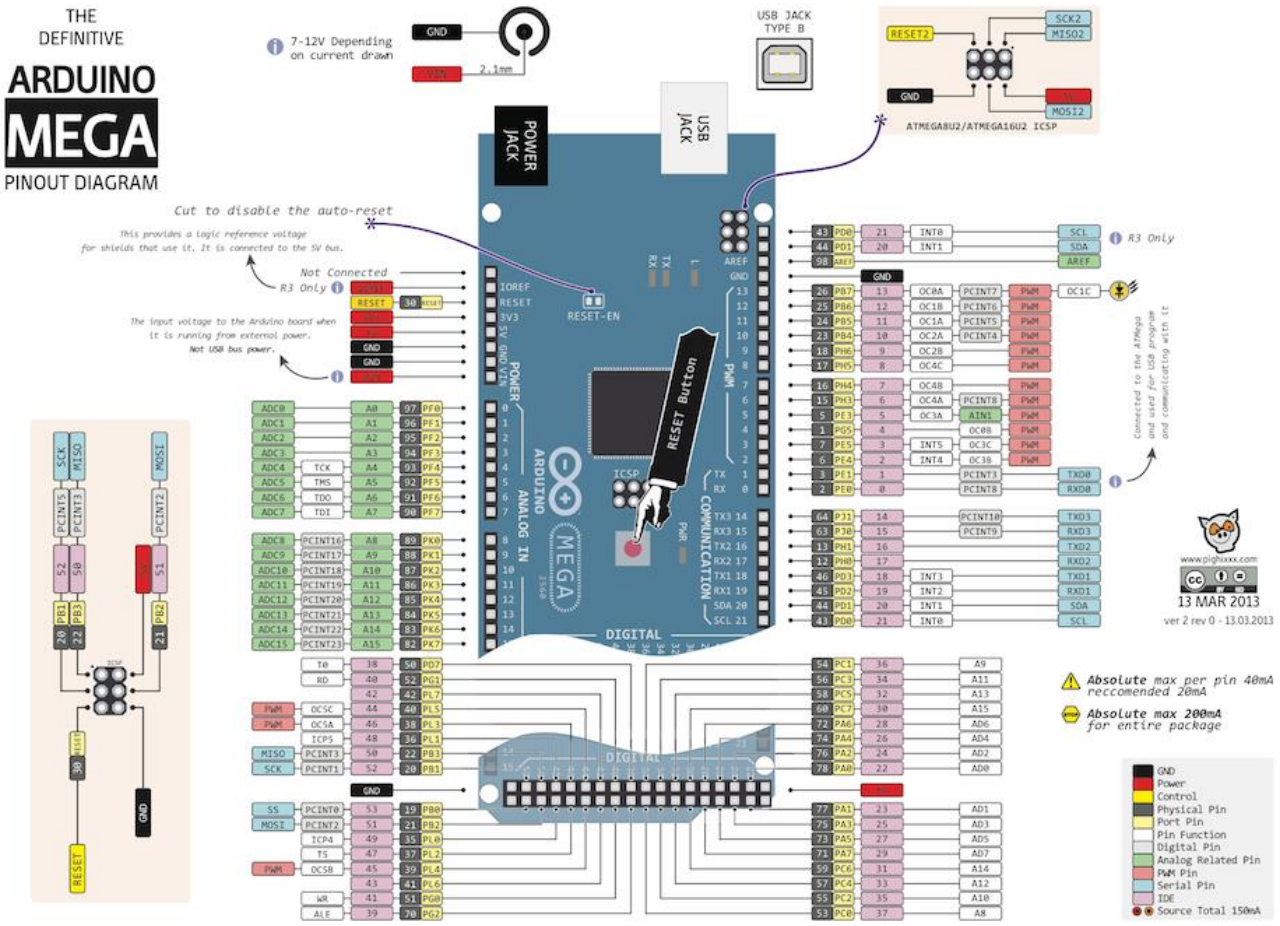
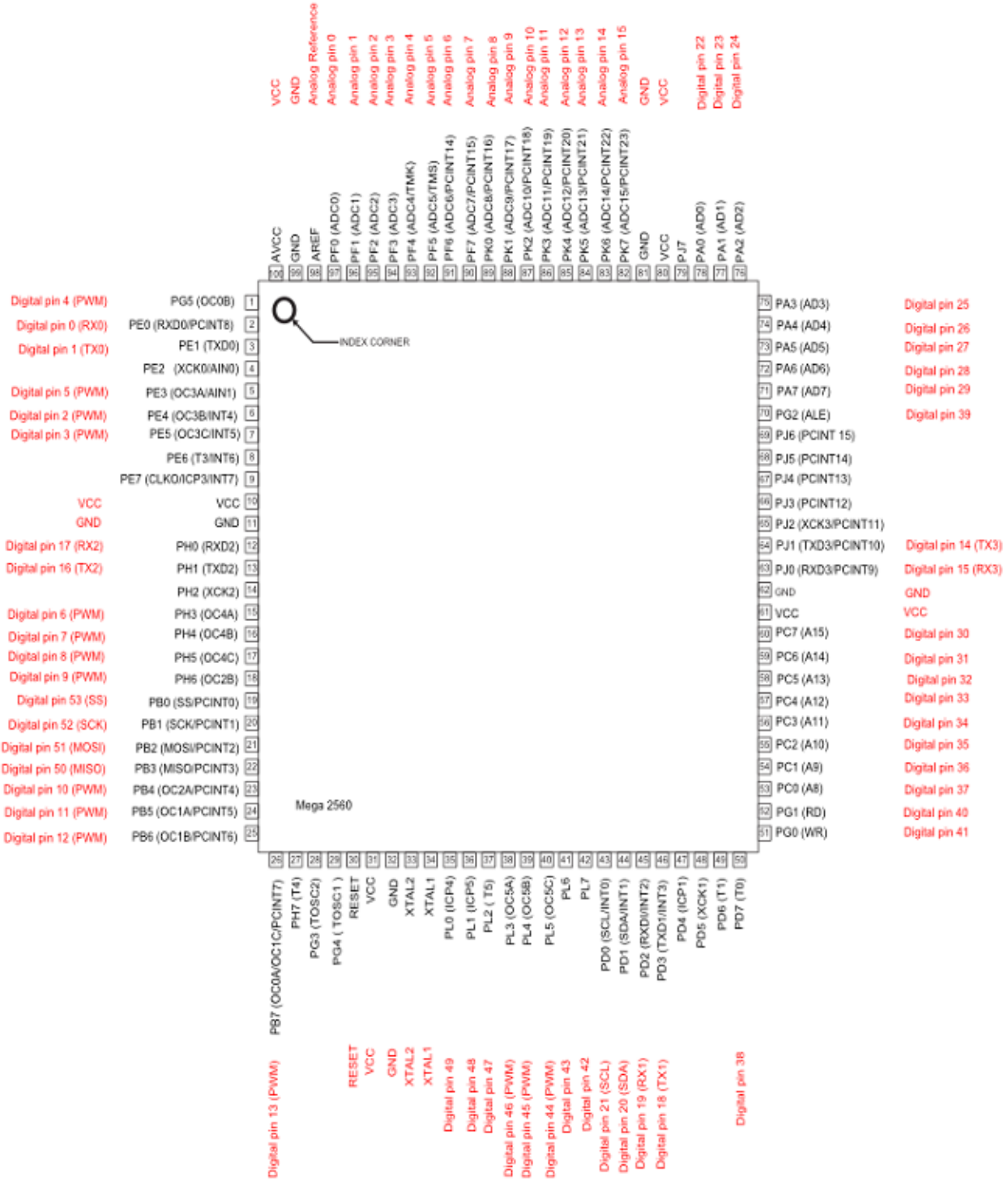


Diagrama de pines del Arduino Mega 2560



El archivo SVG se puede consultar [2] en Internet.

Arduino Mega 2560 PIN mapping table

Pin Number	Pin Name	Mapped Pin Name
1	PG5 (OC0B)	Digital pin 4 (PWM)
2	PE0 (RXD0/PCINT8)	Digital pin 0 (RX0)
3	PE1 (TXD0)	Digital pin 1 (TX0)
4	PE2 (XCK0/AIN0)	
5	PE3 (OC3A/AIN1)	Digital pin 5 (PWM)
6	PE4 (OC3B/INT4)	Digital pin 2 (PWM)
7	PE5 (OC3C/INT5)	Digital pin 3 (PWM)
8	PE6 (T3/INT6)	
9	PE7 (CLKO/ICP3/INT7)	
10	VCC	VCC
11	GND	GND
12	PH0 (RXD2)	Digital pin 17 (RX2)
13	PH1 (TXD2)	Digital pin 16 (TX2)
14	PH2 (XCK2)	
15	PH3 (OC4A)	Digital pin 6 (PWM)
16	PH4 (OC4B)	Digital pin 7 (PWM)
17	PH5 (OC4C)	Digital pin 8 (PWM)
18	PH6 (OC2B)	Digital pin 9 (PWM)
19	PB0 (SS/PCINT0)	Digital pin 53 (SS)
20	PB1 (SCK/PCINT1)	Digital pin 52 (SCK)
21	PB2 (MOSI/PCINT2)	Digital pin 51 (MOSI)
22	PB3 (MISO/PCINT3)	Digital pin 50 (MISO)
23	PB4 (OC2A/PCINT4)	Digital pin 10 (PWM)
24	PB5 (OC1A/PCINT5)	Digital pin 11 (PWM)
25	PB6 (OC1B/PCINT6)	Digital pin 12 (PWM)
26	PB7 (OC0A/OC1C/PCINT7)	Digital pin 13 (PWM)
27	PH7 (T4)	
28	PG3 (TOSC2)	
29	PG4 (TOSC1)	
30	RESET	RESET
31	VCC	VCC

32	GND	GND
33	XTAL2	XTAL2
34	XTAL1	XTAL1
35	PL0 (ICP4)	Digital pin 49
36	PL1 (ICP5)	Digital pin 48
37	PL2 (T5)	Digital pin 47
38	PL3 (OC5A)	Digital pin 46 (PWM)
39	PL4 (OC5B)	Digital pin 45 (PWM)
40	PL5 (OC5C)	Digital pin 44 (PWM)
41	PL6	Digital pin 43
42	PL7	Digital pin 42
43	PD0 (SCL/INT0)	Digital pin 21 (SCL)
44	PD1 (SDA/INT1)	Digital pin 20 (SDA)
45	PD2 (RXDI/INT2)	Digital pin 19 (RX1)
46	PD3 (TXD1/INT3)	Digital pin 18 (TX1)
47	PD4 (ICP1)	
48	PD5 (XCK1)	
49	PD6 (T1)	
50	PD7 (T0)	Digital pin 38
51	PG0 (WR)	Digital pin 41
52	PG1 (RD)	Digital pin 40
53	PC0 (A8)	Digital pin 37
54	PC1 (A9)	Digital pin 36
55	PC2 (A10)	Digital pin 35
56	PC3 (A11)	Digital pin 34
57	PC4 (A12)	Digital pin 33
58	PC5 (A13)	Digital pin 32
59	PC6 (A14)	Digital pin 31
60	PC7 (A15)	Digital pin 30
61	VCC	VCC
62	GND	GND
63	PJ0 (RXD3/PCINT9)	Digital pin 15 (RX3)
64	PJ1 (TXD3/PCINT10)	Digital pin 14 (TX3)
65	PJ2 (XCK3/PCINT11)	
66	PJ3 (PCINT12)	

67	PJ4 (PCINT13)	
68	PJ5 (PCINT14)	
69	PJ6 (PCINT 15)	
70	PG2 (ALE)	Digital pin 39
71	PA7 (AD7)	Digital pin 29
72	PA6 (AD6)	Digital pin 28
73	PA5 (AD5)	Digital pin 27
74	PA4 (AD4)	Digital pin 26
75	PA3 (AD3)	Digital pin 25
76	PA2 (AD2)	Digital pin 24
77	PA1 (AD1)	Digital pin 23
78	PA0 (AD0)	Digital pin 22
79	PJ7	
80	VCC	VCC
81	GND	GND
82	PK7 (ADC15/PCINT23)	Analog pin 15
83	PK6 (ADC14/PCINT22)	Analog pin 14
84	PK5 (ADC13/PCINT21)	Analog pin 13
85	PK4 (ADC12/PCINT20)	Analog pin 12
86	PK3 (ADC11/PCINT19)	Analog pin 11
87	PK2 (ADC10/PCINT18)	Analog pin 10
88	PK1 (ADC9/PCINT17)	Analog pin 9
89	PK0 (ADC8/PCINT16)	Analog pin 8
90	PF7 (ADC7)	Analog pin 7
91	PF6 (ADC6)	Analog pin 6
92	PF5 (ADC5/TMS)	Analog pin 5
93	PF4 (ADC4/TMK)	Analog pin 4
94	PF3 (ADC3)	Analog pin 3
95	PF2 (ADC2)	Analog pin 2
96	PF1 (ADC1)	Analog pin 1
97	PF0 (ADC0)	Analog pin 0
98	AREF	Analog Reference
99	GND	GND
100	AVCC	VCC

Referencias

[1] https://s-media-cache-ak0.pinimg.com/originals/aa/c6/d1/aac6d1487157283d2b0905c91d6775c1.png

[2] https://www.arduino.cc/en/Hacking/PinMapping2560

[3] https://es.wikipedia.org/wiki/Motor_paso_a_paso#Motores_paso_a_paso_Bipolares

[4] http://www.elecrom.com/avr-tutorial-2-avr-input-output/

[5] http://www.ftdichip.com/Drivers/VCP.htm

[6] http://www.ftdichip.com/Support/Documents/InstallGuides/AN_396%20FTDI%20Drivers%20Installation%20Guide%20for%20Windows%2010.pdf

[7] www.**putty**.org

[8] interrupciones en AVR, lista detallada y cómo escribir handers: http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#gad28590624d422cdf30d626e0a506255f

[9] cómo agregar archivos al proyecto en AtmelStudio: https://www.youtube.com/watch?v=n_IvIQeRaCo

[10] datasheet Atmega 2560 http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_datasheet.pdf

[11] www.russemotto.com/xloader