# CNC PRINTED CIRCUIT BOARD DRILLING MACHINE

ALPER YILDIRIM

HACETTEPE UNIVERSITY
Department of Electrical and Electronics Engineering
ELE 401 – ELE 402 Final Project Report
Instructor: Dr. Uğur BAYSAL

Spring 2003

# ABSTRACT

This report explains the final project "CNC Printed Circuit Board (PCB) Drilling Machine" which is implemented in Fall 2002 and Spring 2003 semesters at Hacettepe University Department of Electrical and Electronics Engineering. This report gives information about the necessary parts of the system and their interconnections in the implementation of this project.

# TABLE OF CONTENTS

ABSTRACT ........ ii
TABLE OF CONTENTS ........ iii
LIST OF FIGURES ........ v
LIST OF TABLES ........ vi
LIST OF TABLES ........ vi

1. INTRODUCTION ........ 1

2. SYSTEM OVERVIEW ........ 2

3. MECHANICS ........ 3

4. STEPPER MOTOR BASICS ........ 5
4.1 Technical Description ........ 5
4.2 Stepper Motor Types ........ 6
4.2.1 Variable Reluctance (VR) ........ 6
4.2.2 Permanent Magnet (PM) ........ 7
4.2.3 Hybrid (HB) ........ 7
4.3 Coil Excitation Types ........ 8
4.3.1 Unipolar Stepper Motors ........ 8
4.3.2 Bipolar Stepper Motors ........ 9
4.4 Stepper Motor Drive Sequences ........ 11

5. STEPPER MOTOR DRIVING ........ 12
5.1 Winding Resistance and Inductance ........ 12
5.2 Drive Circuit Schemes ........ 14
5.2.1 Unipolar Drive ........ 15
5.2.2 Bipolar Drive ........ 16
5.3 Current Paths ........ 17
5.4 Current Control ........ 18
5.4.1 The L/R Drive ........ 18
5.4.2 The Bilevel L/R Drive ........ 19
5.4.3 Chopper Control ........ 20

6. STEPPER MOTOR DRIVER IMPLEMENTATION ........ 23
6.1 L297 – Stepper Motor Controller IC ........ 23
6.2 L298 – Dual Full-Bridge Driver IC ........ 24

7. CONTROL BOARD ........ 26
7.1 Microcontrollers ........ 26
7.2 Microcontroller Selection ........ 26
7.3 PIC16F877 Features ........ 26
7.4 Control Board Implementation ........ 28
7.4.1 Communication with PC ........ 28
7.4.2 Connection of Control Signals for Stepper Motor Drivers ........ 28
7.4.3 Limit Switch Inputs ........ 29
7.5 Embedded Programming of PIC16F877 ........ 29
7.5.1 Block Diagram ........ 30

7.5.2 Operation of PIC Firmware ........................................ 31

8. PC SOFTWARE ........................................ 35
8.1 Excellon Drill Format ........................................ 35
8.2 Processing the Drill File ........................................ 36
8.3 Jog Movements ........................................ 36
8.4 Drill Movements ........................................ 36

9. RESULTS AND CONCLUSION ........................................ 38

10. REFERENCES ........................................ 39

11. APPENDICES ........................................ 40
APPENDIX 1 - PIC Source Code (CCS PCWH PIC C Compiler) ........................................ 40
APPENDIX 2 - Software Source Code (Borland C++ Builder 5.0) ........................................ 50
Maxdriller.cpp ........................................ 50
Mainform.cpp ........................................ 50
Aboutform.cpp ........................................ 67
Jogform.cpp ........................................ 68
Settingsform.cpp ........................................ 76
Toolform.cpp ........................................ 79
APPENDIX 3 – Software Screenshots ........................................ 81
Main Screen ........................................ 81
Jog Screen ........................................ 82
Machine Settings Screen ........................................ 82
APPENDIX 4 – L297 STEPPER MOTOR CONTROLLER IC ........................................ 83
APPENDIX 5 – L298 DUAL FULL-BRIDGE DRIVER IC ........................................ 91
APPENDIX 6 – STEPPER MOTOR DRIVER SCHEMATIC ........................................ 99
APPENDIX 7 – CONTROL BOARD SCHEMATIC ........................................ 100
APPENDIX 8 – 3-D SOLID MODELS OF THE MECHANICS ........................................ 101

# LIST OF FIGURES

Figure 2-1: System Overview ........ 2

Figure 3-1 Linear Motion by using a leadscrew ........ 3
Figure 3-2 Leadscrew with ACME Nut ........ 3

Figure 4-1 Construction of a stepper motor ........ 5
Figure 4-2 Variable Reluctance Motor........ 7
Figure 4-3 Permanent Magnet Motor........ 7
Figure 4-4 Hybrid Motor........ 8
Figure 4-5 6-wire Unipolar Stepper Motor ........ 8
Figure 4-6 Reversal of Current in one coil of a Unipolar Stepper Motor ........ 9
Figure 4-7 Unipolar Drive Sequence ........ 9
Figure 4-8 4-wire Bipolar Stepper Motor ........ 10
Figure 4-9 Bipolar Drive Sequence........ 10
Figure 4-10 Conseptional H-Bridge Circuit........ 10

Figure 5-1 Current waveform in an RL circuit ........ 13
Figure 5-2 Effect of Inductance at high stepping rates ........ 13
Figure 5-3 Unipolar Drive........ 15
Figure 5-4 Bipolar Drive........ 16
Figure 5-5 Current Paths on Bipolar Driver........ 17
Figure 5-6 Current Paths on Unipolar Drivers........ 18
Figure 5-7 Current Waveforms on L/R Drive........ 19
Figure 5-8 Current Waveforms on Bilevel L/R Drive ........ 20
Figure 5-9 Simplified Chopper Circuit ........ 21
Figure 5-10 Current Waveforms in a Chopper Circuit ........ 22
Figure 5-11 H-Bridge Connected as Constant Current Chopper ........ 22

Figure 6-1 L297 IC Block Diagram........ 23
Figure 6-2 L298 IC Block Diagram........ 24
Figure 6-3 Two Phase Bipolar Stepper Motor Circuit........ 25

Figure 7-1 Block Diagram of PIC Firmware ........ 30
Figure 7-2 Structure of Jog Data Packet ........ 31
Figure 7-3 Structure of Initialize Drill Mode Packet ........ 32
Figure 7-4 Structure of Drill Data Packet ........ 33

# LIST OF TABLES

Table 4-1 Comparison of Stepper Motor Drive Sequences ................................................ 11

Table 7-1 Dsub-9 Female Connector Pin Diagram............................................................ 28
Table 7-2 Stepper Motor Drivers Input Connector Pin Diagram....................................... 28
Table 7-3 Driver Connections to PIC MCU........................................................................ 29

Table 8-1 Example of an Excellon Drill File ...................................................................... 35

# 1. INTRODUCTION

The goal of this project is to design and implement a computer controlled PCB drilling machine. All the mechanical and electronics design are done from scratch to realize the project. There is also a computer program which communicates with the machine electronics.

Next chapter reveals the main blocks of the designed project. It gives introductory information about the whole system. Following chapters explain the main blocks of the system seperately in detail. The conclusions chapter includes the results of the project and future decisions. References are also added at the end of the project report. Finally, appendices list the source codes and circuit diagrams that are used in this project.

# 2. SYSTEM OVERVIEW

In electronics industry, Printed Circuit Boards (PCB) are designed using Computer Aided Design (CAD) programs. These programs generate a standardized file which is known as Excellon Drill File (Refer to [1] for details). Excellon files define the position of hole locations on the designed PCB. This information is used in Computer Numerical Control (CNC) machines to drill the necessary holes on the PCB.

In this project, the developed software takes the Excellon drill file of the PCB. Than it calculates the necessary parameters and sends the coordinate information to PIC16F877 microcontroller unit (MCU) over RS-232 line. When MCU takes the necessary information, it immediately indexes the stepper motor drivers. Stepper motor drivers turn the stepper motors according to the index pulses applied to them. Stepper motors move the mechanism to accomplish the drilling of the PCBs.

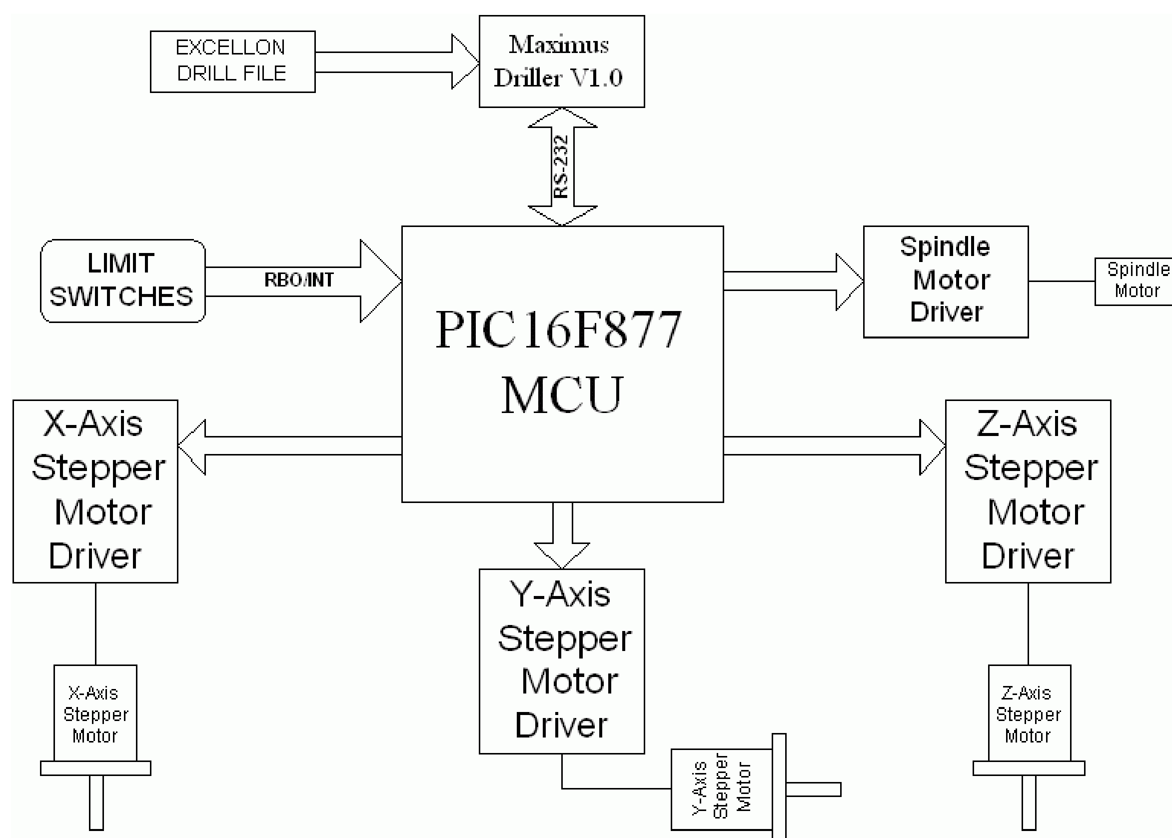



**Figure 2-1: System Overview**

Limit switches are used on the drill mechanism to realize axis overrange protection. That is, when an axis tries to pass it’s limits, the switch is pressed and MCU stops the axis movement immediately.

# 3. MECHANICS

Mechanical system is the realization of 3-dimensional motion control. Motion Control, in electronic terms, means to accurately control the movement of an object based on speed, distance, load, inertia or a combination of all these factors. There are numerous types of motion control systems, including; stepper motor, linear stepper motor, DC brush, brushless DC, servo, brushless servo and more.

The stepper motors are chosen in this study, for the following reasons that they are generally preferred to use with computer controls because they are essentially digital devices and ideal for low cost, open loop control schemes where high torque and rotation speed values are not required. In theory, a Stepper motor is a marvel in simplicity.

A stepper motor-lead screw combination is used on all three axis to make axis movements. The shaft of the motors are coupled to lead screws. The screws are beared at two ends by ball bearings. With this mechanism the rotational movement of the stepper motors are converted to linear movements to drive the axises.

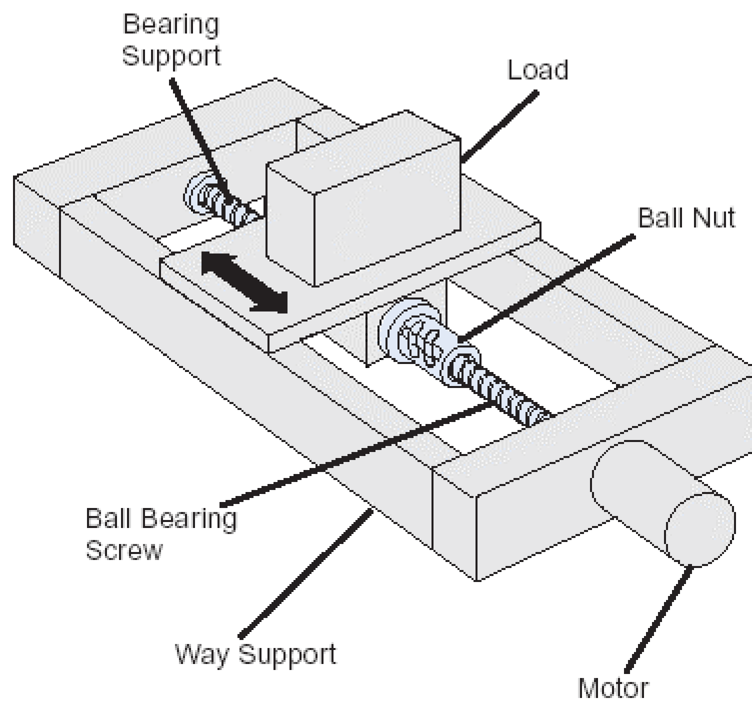



**Figure 3-1 Linear Motion by using a leadscrew**

**Figure 3-2 Leadscrew with ACME Nut**

A 3-D solid model of the mechanical system is designed with AutoCAD 2000 before making the mechanical parts. The design is investigated for various working parameters. These parameters include lightness in weight, low friction, low mechanical cost and low inertia. The system is designed to be driven with NEMA 23 sized stepper motors. (i.e. Motors with 50-100 Ncm holding torque)
By considering these parameters, the mechanical system is constructed as follows:

- 3 axis stepper motor – lead screw linear motion mechanism
- 12mm diameter Inox stainless steel lead screws
- 12mm Inox Nuts
- Kestamid and Polietheleyn nut holders
- 12mm mercury steel shafts for axis support
- 12mm delrin parts for shaft sliders
- Polietheleyn parts for main body
- Kestamid couplings
- Aluminium motor holders

All parts are done in mechanical workshops by using lathes and milling machines.
Appendix 8 shows 3-D solid model drawings of the mechanics.

# 4. STEPPER MOTOR BASICS

## 4.1 Technical Description

Stepper motors are electromechanical equipments converting electrical energy into rotation movement. Pulses of electricity drive rotor and connected shaft. They are connected to stepper motor drivers which have high switching capability. This driver gets pulses from a digital controller and each pulse drives the shaft of the motor for a determined angle. This little angle is called step angle and fixed for each motor. The speed and direction of the movement depends on pulse sequence and pulse frequency. A basic shape of a stepper motor is shown in **Figure 4.1**.

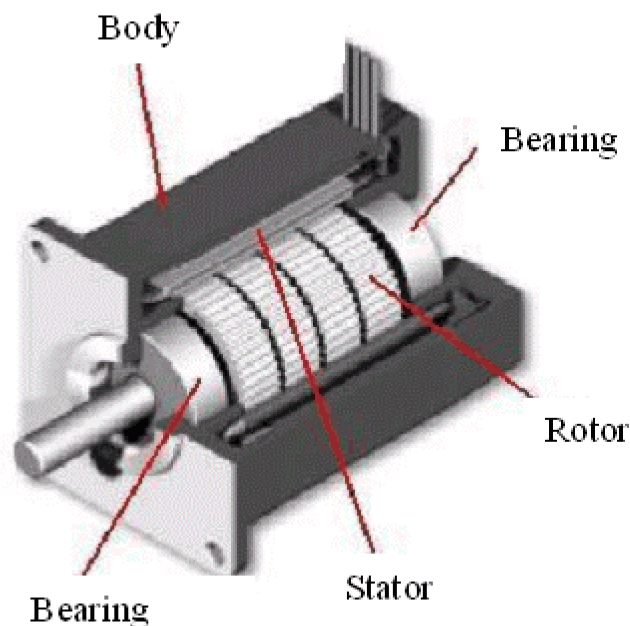



**Figure 4-1 Construction of a stepper motor**

The rotation has not only a direct relation to the number of input pulses, but its speed is also related to the frequency of the pulses. Stepper motors vary in the amount of rotation that the shaft turns each time when a winding is energized. The amount of rotation is called step angle as mentioned before and vary from 0.9° degrees (1.8° degrees is more common) to 90° degrees. Step angle determines the number of steps per revolution. A stepper with a 1.8° degrees step angle must be pulsed 200 times (1.8° x 200 = 360°) for the shaft to turn one complete revolution. Sensitivity of a stepper motor increases with the number of steps in one revolution like its cost.

Obviously, a smaller step angle increase the accuracy of a motor. But stepper motors have an upper limit to the number of pulses they can accept per second. Heavy-duty steppers usually have a maximum pulse rate (or step rate) of 200 or 300 steps per second, so they

have an effective high speed of one to three revolution per second (60 to 180 rpm). Some smaller steppers can accept a thousand or more pulses per second, but they don't provide very torque and are not suitable as driving or steering motors.

The stepper motor coils are typically rated for a particular voltage. The coils act as inductors when voltage is supplied to them. As such they don't instantly draw their full current and in fact may never reach full current at high stepping frequencies. The electromagnetic field produced by the coils is directly related to the amount of current they draw. The larger the electromagnetic field the more torque the motors have the potential of producing. The solution to increasing the torque is to ensure that the coils reach full current draw during each step.

Stepper motors can be viewed as electric motors without commutators. Typically, all windings in the motor are part of the stator, and the rotor is either a permanent magnet or, in the case of variable reluctance motors, a toothed block of some magnetically soft material. All of the commutation must be handled externally by the motor controller, and typically, the motors and controllers are designed so that the motor may be held in any fixed position as well as being rotated one way or the other.

It should be noted that stepper motors couldn't be motivated to run at their top speeds immediately from a dead stop. Applying too many pulses right off the bat simply causes the motor to freeze up. To achieve top speeds, the motor must be gradually accelerated. The acceleration can be quite swift in human terms. The speed can be 1/3 for the first few milliseconds, 2/3 for the next 50 or 75 milliseconds, then full blast after that.

Actuation of one of the windings in a stepper motor advances the shaft. Continue to apply the current to the winding and the motor wont turn any more. In fact, the shaft will be locked, as if brakes are applied. As a result of this interesting locking effect, you never need to add a braking circuit to a stepper motor, because it has its own brakes built in. The amount of breaking power of a stepper motor is expressed as holding torque.

## 4.2 Stepper Motor Types

### 4.2.1 Variable Reluctance (VR)

VR motors are characterized as having a soft iron multiple rotor and a wound stator. They generally operate with step angles from 5 degrees to 15 degrees at relatively high step rates, and have no detent torque (detent torque is the the holding torque when no current is flowing in the motor).

In **Figure 4.2** , when phase A is energized, four rotor teeth line up with the four stator teeth of phase A by magnetic attraction. The next step is taken when A is turned off and phase B is energized, rotating the rotor clockwise 15 degrees; Continuing the sequence, C is turned on next and then A again. Counter clockwise rotation is achieved when the phase order is reversed.

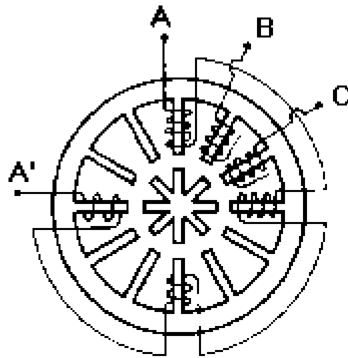



**Figure 4-2 Variable Reluctance Motor**

### 4.2.2 Permanent Magnet (PM)

PM motors differ from VR's by having permanent magnet rotors with no teeth, and are magnetized perpendicular to the axis. In energizing the four phases in sequence, the rotor rotates as it is attracted to the magnetic poles. The motor shown in **Figure 4.3** will take 90 degree steps as the windings are energized in sequence ABCD. PM's generally have step angles of 45 or 90 degrees and step at relatively low rates, but they exhibit high torque and good damping characteristics.

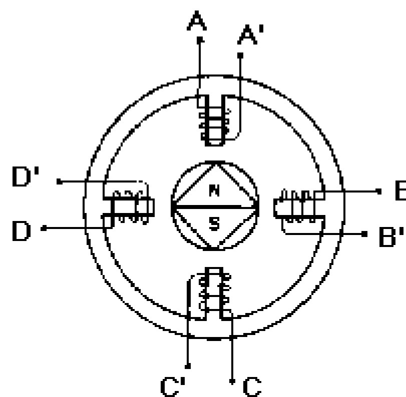



**Figure 4-3 Permanent Magnet Motor**

### 4.2.3 Hybrid (HB)

Combining the qualities of the VR and the PM, the hybrid motor has some of the desirable features of each. They have high detent torque and excellent holding and dynamic torque, and they can operate at high stepping speeds. Normally, they exhibit step angles of 0.9 to 5

degrees. Bi-filar windings are generally supplied (as depicted in **Figure 4.4**), so that a single-source power supply can be used . If the phases are energized one at a time, in the order indicated, the rotor would rotate in increments of 1.8 degrees. This motor can also be driven two phases at a time to yield more torque, or alternately one then two then one phase, to produce half steps or 0.9 degree increments.

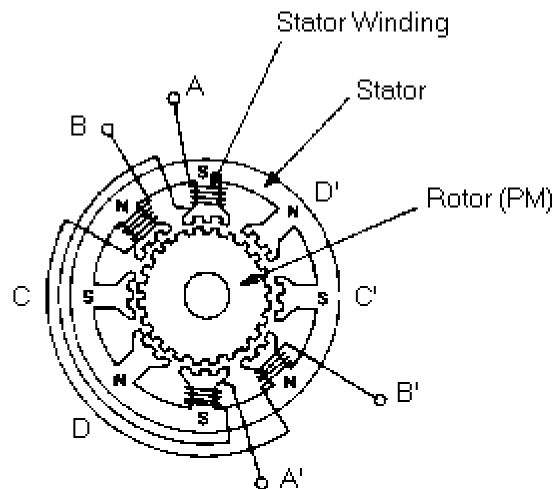



**Figure 4-4 Hybrid Motor**

## 4.3 Coil Excitation Types

### 4.3.1 Unipolar Stepper Motors

Unipolar motors are relatively easy to control. A simple 1-of-'n' counter circuit can generate the proper stepper sequence, and drivers as simple as 1 transistor per winding are possible with unipolar motors. Unipolar stepper motors are characterized by their center-tapped windings. A common wiring scheme is to take all the taps of the center-tapped windings and feed them +Vm (Motor voltage). The driver circuit would then ground each winding to energize it.

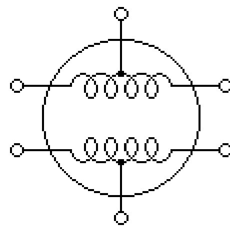


**Figure 4-5 6-wire Unipolar Stepper Motor**

Unipolar stepper motors, both Permanent magnet and hybrid stepper motors with 5 or 6 wires are usually wired as shown in the schematic in **Figure 4.5**, with a center tap on each of two windings. In use, the center taps of the windings are typically wired to the positive

supply, and the two ends of each winding are alternately grounded to reverse the direction of the field provided by that winding. (**Figure 4.6**)

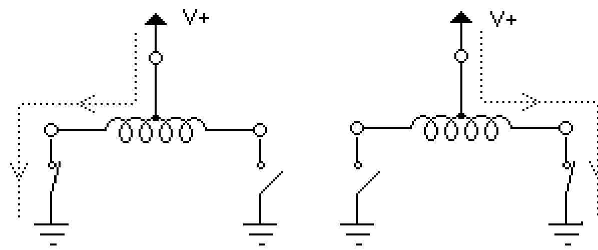
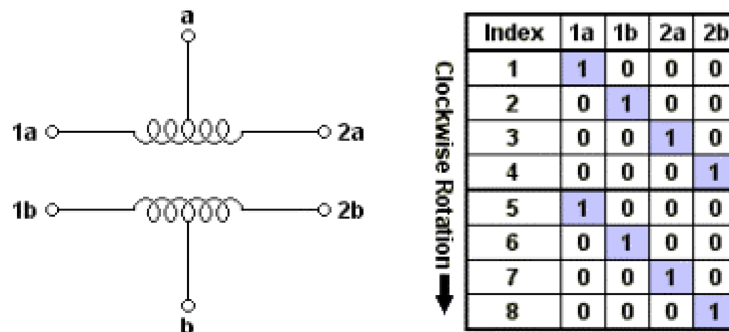



**Figure 4-6 Reversal of Current in one coil of a Unipolar Stepper Motor**

In unipolar stepper motors the number of phases is twice the number of coils, since each coil is divided in two. The diagram below which has two center-tapped coils, represents the connection of a 4-phase unipolar stepper motor.




| Index | 1a | 1b | 2a | 2b |
|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 0 | 0 |
| 7 | 0 | 0 | 1 | 0 |
| 8 | 0 | 0 | 0 | 1 |

**Figure 4-7 Unipolar Drive Sequence**

In addition to the standard drive sequence, high-torque and half-step drive sequences are also possible. In the high-torque sequence, two windings are active at a time for each motor step. This two-winding combination yields around 1.4 times more torque than the standard sequence, but it draws twice the current. Half-stepping is achieved by combining the two sequences. First, one of the windings is activated, then two, then one, etc. This effectively doubles the number of steps the motor will advance for each revolution of the shaft, and it cuts the number of degrees per step in half.

## 4.3.2 Bipolar Stepper Motors

Bipolar permanent magnet and hybrid motors are constructed with exactly the same mechanism as is used on unipolar motors, but the two windings are wired more simply, with no center taps. Thus, the motor itself is simpler but the drive circuitry needed to reverse the polarity of each pair of motor poles is more complex. The schematic in **Figure 4.8** shows how such a motor is wired.

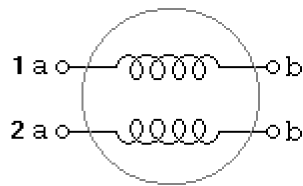
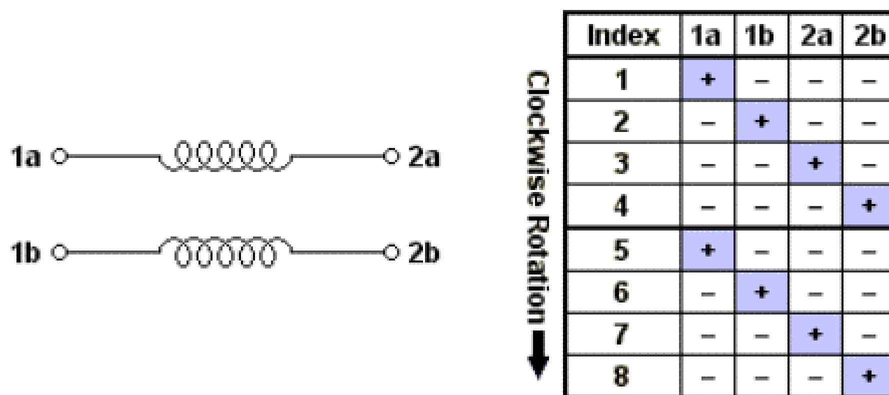



**Figure 4-8 4-wire Bipolar Stepper Motor**

Unlike unipolar stepper motors, bipolar units require more complex driver circuitry. Bipolar motors are known for their excellent size/torque ratio, and provide more torque for their size than unipolar motors. Bipolar motors are designed with *separate* coils that need to be driven in either direction (the polarity needs to be reversed during operation) for proper stepping to occur. This presents a driver challenge. Bipolar stepper motors use the same binary drive pattern as a unipolar motor, only the '0' and '1' signals correspond to the polarity of the voltage applied to the coils, not simply 'on-off' signals. **Figure 4.9** shows a basic 4-phase bipolar motor's coil setup and drive sequence.




| Index | 1a | 1b | 2a | 2b |
|---|---|---|---|---|
| 1 | + | – | – | – |
| 2 | – | + | – | – |
| 3 | – | – | + | – |
| 4 | – | – | – | + |
| 5 | + | – | – | – |
| 6 | – | + | – | – |
| 7 | – | – | + | – |
| 8 | – | – | – | + |

**Figure 4-9 Bipolar Drive Sequence**

The Bipolar Controller must be able to reverse the polarity of the voltage across either coil, so current can flow in both directions. And, it must be able to energize these coils in sequence. The mechanism for reversing the voltage across one of the coils is called an H-Bridge, because it resemble a letter "H" (**Figure 4.10**).

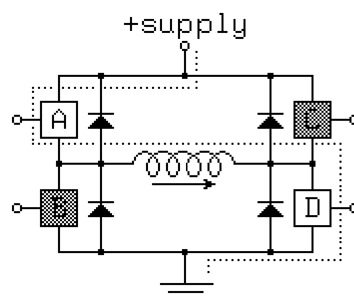



**Figure 4-10 Conseptional H-Bridge Circuit**

## 4.4 Stepper Motor Drive Sequences

The following table describes 3 useful stepping sequences and their relative merits. The polarity of terminals is indicated with +/-. After the last step in each sequence the sequence repeats. Stepping backwards through the sequence reverses the direction of the motor. Note that these sequences are identical for unipolar and bipolar stepper motors.

| Name | Sequence | Polarity | Description |
|---|---|---|---|
| Wave Drive,<br>One-Phase | 0001<br>0010<br>0100<br>1000 | ---+<br>--+-<br>-+--<br>+--- | Consumes the least power. Only one phase is energized at a time. Assures positional accuracy regardless of any winding imbalance in the motor. |
| Hi-Torque,<br>Two-Phase | 0011<br>0110<br>1100<br>1001 | --++<br>-++-<br>++--<br>+--+ | This sequence energizes two adjacent phases, which offers an improved torque-speed product and greater holding torque. |
| Half-Step | 0001<br>0011<br>0010<br>0110<br>0100<br>1100<br>1000<br>1001 | ---+<br>--++<br>--+-<br>-++-<br>-+--<br>++--<br>+---<br>+--+ | Effectively doubles the stepping resolution of the motor, but the torque is not uniform for each step. (Since switching occurs between Wave Drive and Hi-Torque with each step, torque alternates each step.) This sequence reduces motor resonance which can sometimes cause a motor to stall at a particular resonant frequency. Note that this sequence is 8 steps. |

**Table 4-1 Comparison of Stepper Motor Drive Sequences**

# 5. STEPPER MOTOR DRIVING

## 5.1 Winding Resistance and Inductance

Resistance and inductance are two inherent physical properties of a winding, or any coil. These two basic factors also limit the possible performance of the motor.

The resistance of the windings is responsible for the major share of the power loss and heat up of the motor. Size and thermal characteristics of the winding and the motor limit the maximum allowable power dissipated in the winding. The power loss is given by:

$$Pr = R * Im^2$$

where R is the winding resistance and Im is the winding current.

It is important to note that a motor should be used at its maximum power dissipation to be efficient. If a motor is running below its power dissipation limit, it means that it could be replaced by a smaller size motor, which most probably is less expensive.

Inductance makes the motor winding oppose current changes, and therefore limits high speed operation. Figure 5.1 shows the electrical characteristics of an inductive-resistive circuit. When a voltage is connected to the winding the current rises according to the equation

$$I(t) = ( V / R ) * ( 1 - e^{- t \cdot R/L} )$$

Initially the current increases at a rate of

$$\frac{dI}{dt}(0) = V / L$$

The rise rate decreases as the current approaches the final level:

$$I_{max} = V / R$$

The value of $\tau = L / R$ is defined as the electrical time constant of the circuit. $\tau$ is the time until the current reaches 63 % ( 1 – 1/e ) of its final value.

When the inductive–resistive circuit is disconnected and shorted at the instant $t = t_1$ , the current starts to decrease:

$$I(t) = ( V / R ) * e^{- (t-t_1) * R / L}$$

at an initial rate of

$$I(t) = - V / L$$

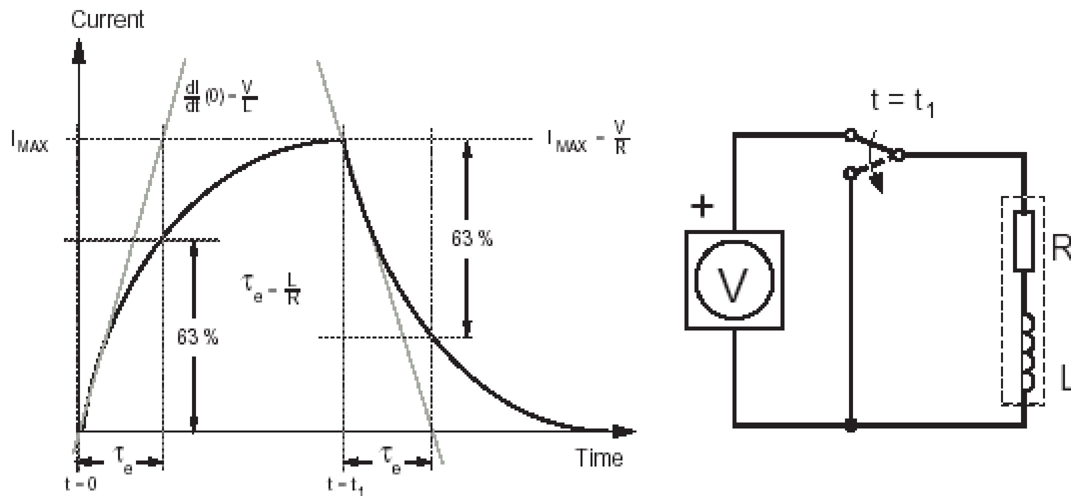



**Figure 5-1 Current waveform in an RL circuit**

When a square wave voltage is applied to the winding, which is the case when fullstepping a stepper motor, the current waveform will be smoothed. Figure 5.2 shows the current at three different frequencies. Above a certain frequency (B) the current never reaches its maximum value (C). As the torque of the motor is approximately proportional to the current, the maximum torque will be reduced as the stepping frequency increases.

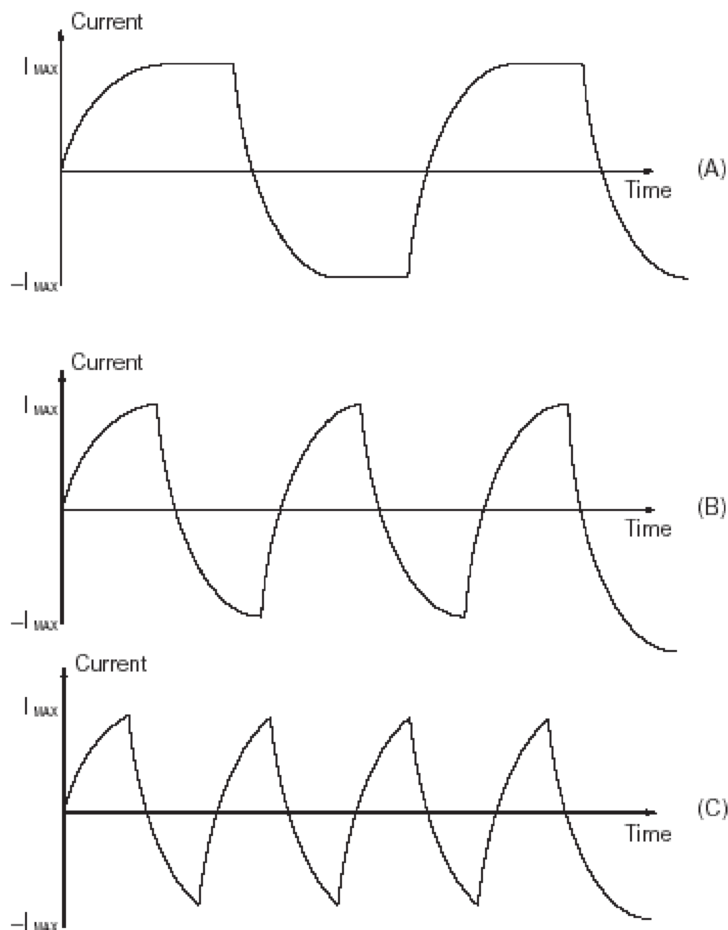



**Figure 5-2 Effect of  Inductance at high stepping rates**

To overcome the inductance and gain high speed performance of the motor two possibilities exist: Increase the current rise rate and/or decrease the time constant. As an increased resistance always results in an increased power loss, it is preferably the ratio V/L that should be increased to gain high speed performance.

To drive current through the winding, we should:

- use as high voltage as possible (without exceeding the motor limits)
- keep the inductance low.

Accordingly, a low inductance  resistance motor has a higher current rating. As the maximum current is limited by the driver, we find that high performance is highly dependant on the choice of driver.

The limiting factor of the motor is the power dissipation, and not the current itself. To utilize the motor efficiently, power dissipation should be at the maximum allowed level.

## 5.2 Drive Circuit Schemes

The stepper motor driver circuit has two major tasks:

• To change the current and flux direction in the phase windings

• To drive a controllable amount of current through the windings, and enabling current rise and fall times as short as possible for good high speed performance.

Stepping of the stepper motor requires a change of the flux direction, independently in each phase. The direction change is done by changing the current direction, and may be done in two different ways, using a unipolar or bipolar drive.

### 5.2.1 Unipolar Drive

The unipolar drive principle requires a winding with a center-tap, or two separate windings per phase (Figure 5.3). Flux direction is reversed by moving the current from one half of the winding to the other half. This method requires only two switches per phase. On the other hand, the unipolar drive utilizes only half the available copper volume of the winding. The power loss in the winding is therefore twice the loss of a bipolar drive at the same output power.

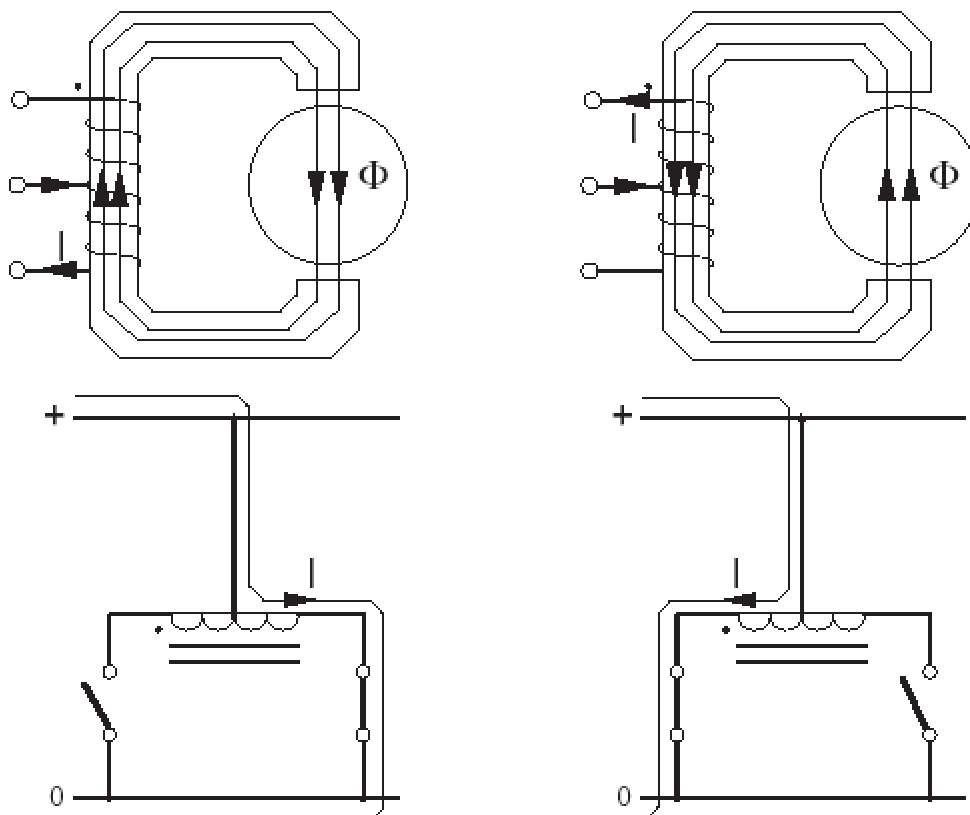



**Figure 5-3 Unipolar Drive**

### 5.2.2 Bipolar Drive

Bipolar drive refers to the principle where the current direction in one winding is changed by shifting the voltage polarity across the winding terminals. To change polarity a total of four switches are needed, forming an “H” letter. Thus bipolar drivers are known as H-Bridges

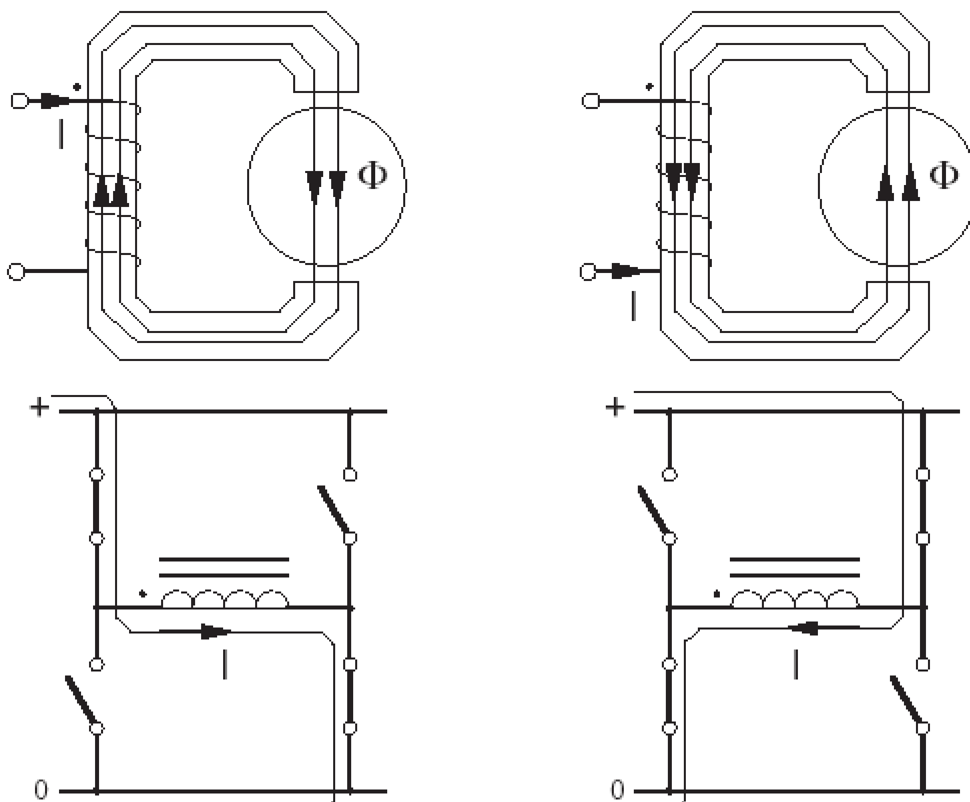



**Figure 5-4 Bipolar Drive**

## 5.3 Current Paths

Another very important consideration is current paths at turn-off and at phase shift. The inductive nature of the winding demands that a current path always exists. When using transistors as switches, diodes have to be added to enable current flow in both directions across the switch. For the bipolar driver four diodes, one for each switch, provide current paths according to figure 5.5.

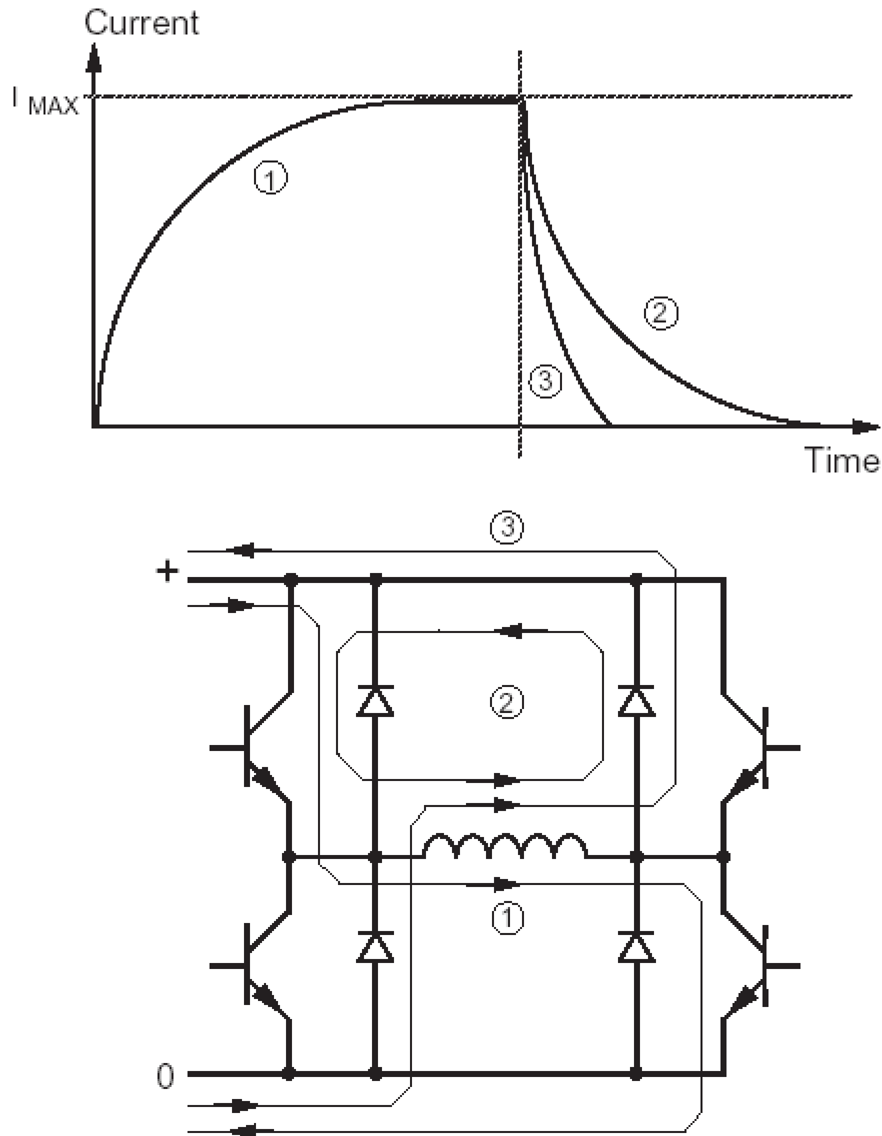



**Figure 5-5 Current Paths on Bipolar Driver**

There are two ways to turn the current off, either by turning all transistors off (path 3), or turn just one of the two conducting transistors off (path 2). First methode gives a fast current decay as the energy stored in the winding inductance is discharged at a high voltage, $V_{supply}$. Second methode gives a slow current decay as the counter voltage is only two diode voltage drops and the resistive voltage drop across the winding resistance. At

phase shift the current will decay rapidly as both conducting transistors are turned off. For high speed halfstepping a rapid decay to zero in the half step position is important.
Figure 5.6 shows some possible schemes for current paths on unipolar drivers.

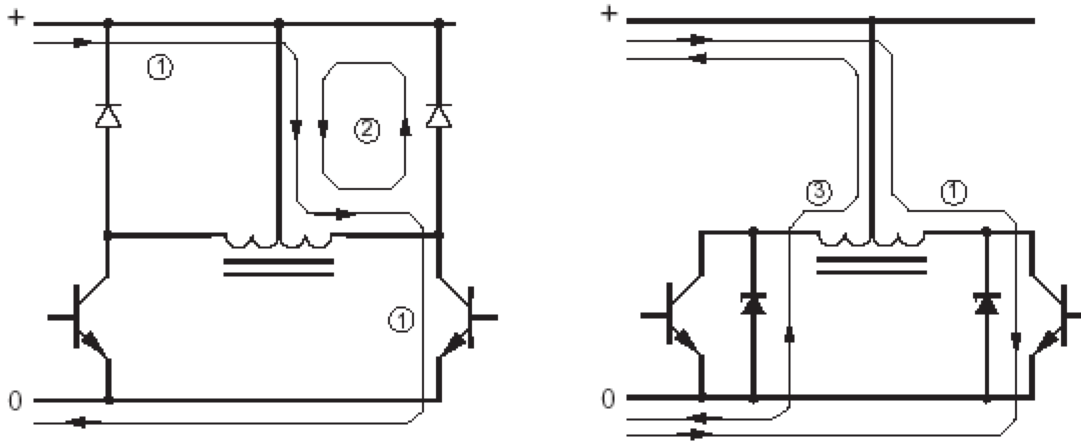



**Figure 5-6 Current Paths on Unipolar Drivers**

## 5.4 Current Control

To control the torque as well as to limit the power dissipation in the winding resistance, the current must be controlled or limited. Furthermore, when half stepping, a zero current level is needed, while microstepping requires a continuously variable current.
Principles to limit the current are described below. Any of the methods may be realized as a bipolar or unipolar driver.

### 5.4.1 The L/R Drive

In this basic method the current is limited by supply voltage and the resistance of the winding, and if necessary, an additional external resistance (dropping resistor):

$$I_m = \frac{V_{supply}}{R_{coil} + R_{ext}}$$

If the nominal motor voltage is the same as the supply voltage, $R_{ext}$ is excluded.

For a given motor, high speed performance is increased by increasing the supply voltage. An increased supply voltage in the resistance limited drive must be accompanied by an additional resistor $R_{ext}$ in series with the winding to limit the current to the previous level.

The time constant:

$$\tau = \frac{L}{R_{coil} + R_{ext}}$$

decreases, which shortens the current rise time (Figure 5.7).

The penalty using this method is the power loss in the additional external resistors which is given by:

$$P_{loss} = I_m^2 * R_{ext}$$

Usually several watts has to be dissipated - and supplied. Spacious power resistors, heat removal considerations and a space consuming power supply reduce cost effectivenes and limits L/R drive scheme to small motors, rated around 1- 2 Watts.

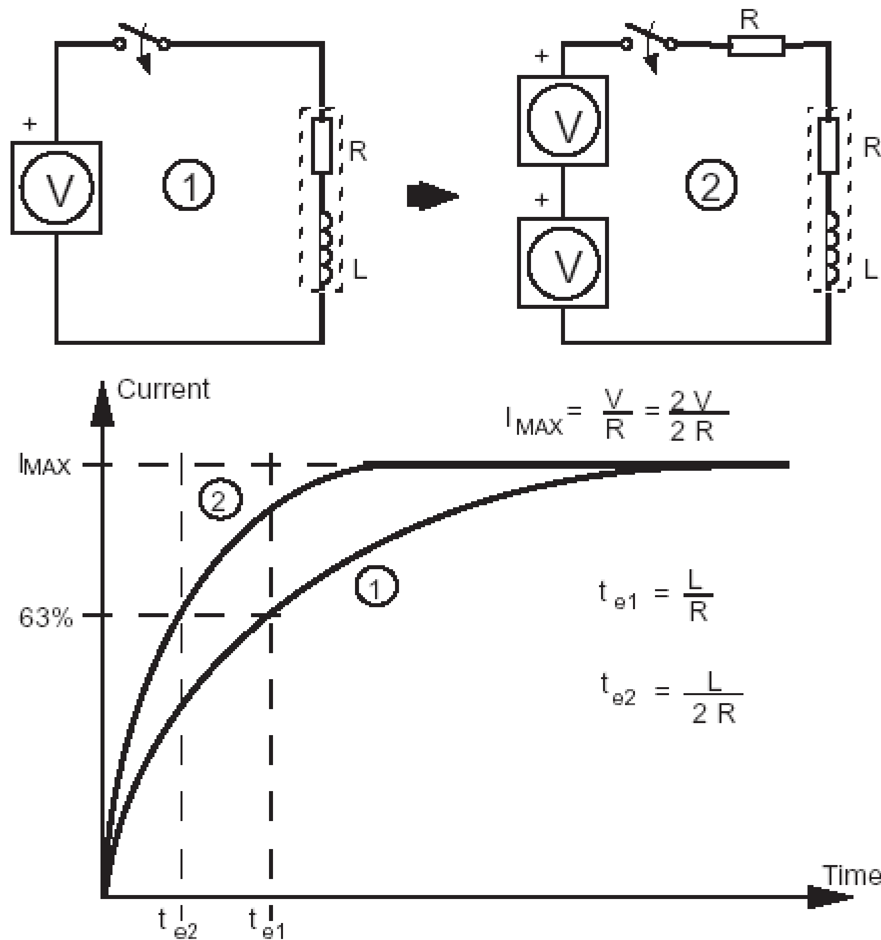



**Figure 5-7 Current Waveforms on L/R Drive**

### 5.4.2 The Bilevel L/R Drive

The bilevel L/R drive provides a solution to the power waste using dropping resistors. In the beginning of the current build-up period, the winding is connected to a secondary high voltage supply. After a short time, when the current has reached its nominal level, the second level supply is disconnected (Figure 5.8). The disadvantage of bilevel drive is the need of a second level power supply. In some applications where 5V, 12V and 24V are available, it may be a cost effective solution, but, if not available, it is a costly method. It is possible to use voltage doubling techniques as well.

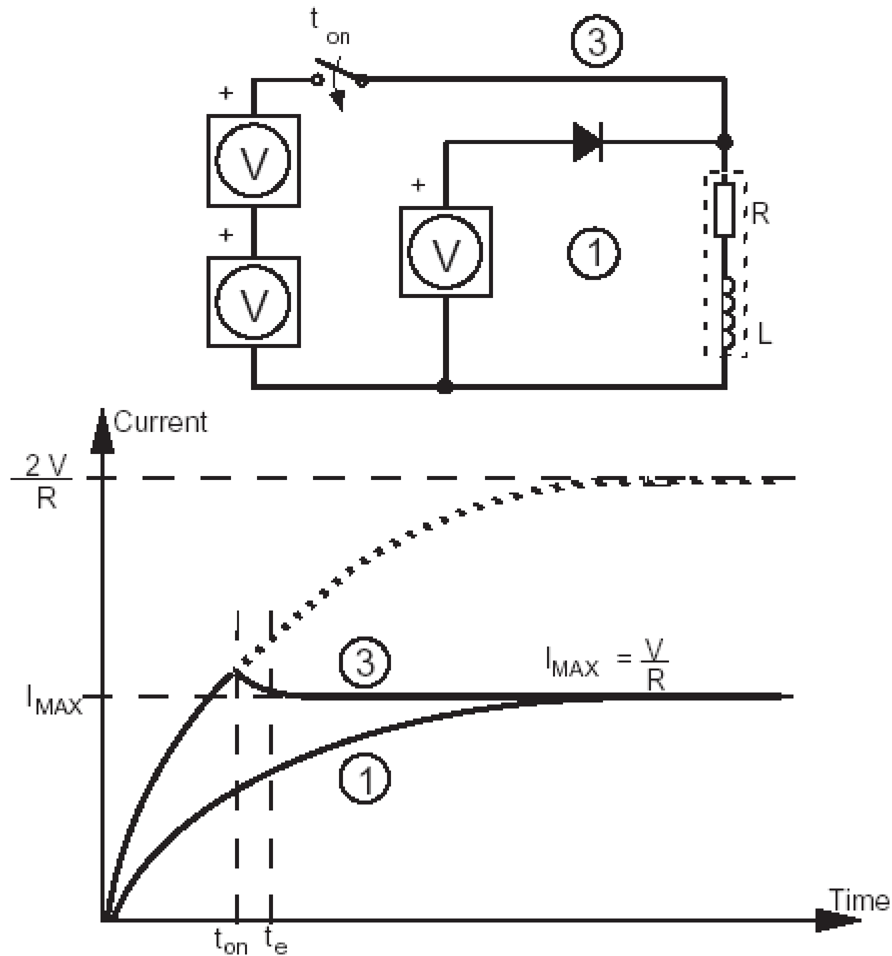



**Figure 5-8 Current Waveforms on Bilevel L/R Drive**

### 5.4.3 Chopper Control

The chopper driver provides an optimal solution both to current control and fast current build-up and reversal. The basic idea is to use a supply voltage which is several times higher than the nominal voltage of the motor. The current rise rate, which initially is V / L, is thereby possible to increase substantially. The ratio $V_M$ / $V_{supply}$ is called the overdrive ratio. By controlling the duty cycle of the chopper, an average voltage and an average current equal to the nominal motor voltage and current are created.

Constant current regulation is achieved by switching the output current to the windings. This is done by sensing the peak current through the winding via a current-sensing resistor, effectively connected in series with the motor winding. As the current increases, a voltage develops across the sensing resistor, which is fed back to the comparator. At the predetermined level, defined by the voltage at the reference input, the comparator resets the flipflop, which turns off the output transistor. The current decreases until the clock oscillator triggers the flip-flops, which turns on the output transistor again, and the cycle is repeated (Figure 5.9).

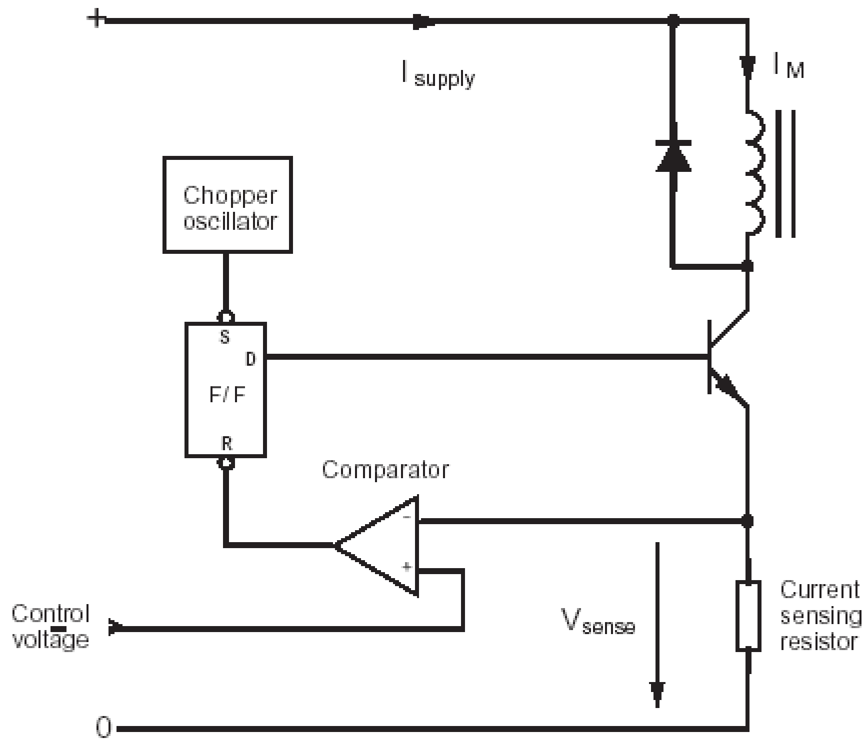



**Figure 5-9 Simplified Chopper Circuit**

The advantage of the constant current control is a precise control of the developed torque, regardless of power supply voltage variations. It also gives the shortest possible current build-up and reversal time. Power dissipation is minimized, as well as supply current. Figure 5.10 shows the current waveforms in a chopper circuit. Supply current is not the same as the motor current in a chopper drive. It is the motor current multiplied by the duty cycle. At standstill typically

$$I_{supply} = I_m \frac{V_m}{V_{supply}}$$

Figure 5.11 shows an H-bridge configured as a constant current chopper. Depending on how the H-bridge is switched during the turn-off period, the current will either recirculate through one transistor and one diode (path 2), giving the slow current decay, or recirculate back through the power supply (path 3). The advantage of feeding the power back to the power supply is the fast current decay and the ability to quickly reduce to a lower current level.

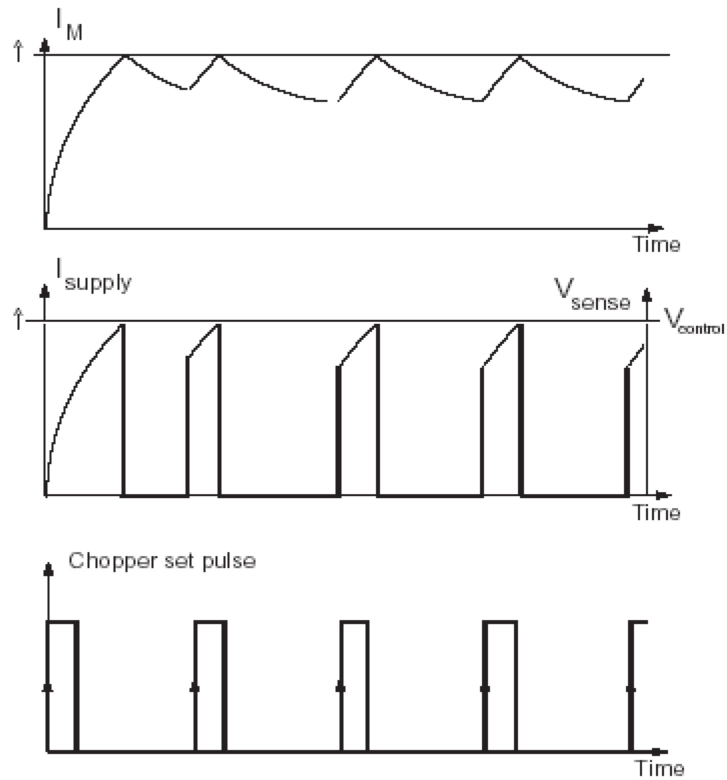



**Figure 5-10 Current Waveforms in a Chopper Circuit**

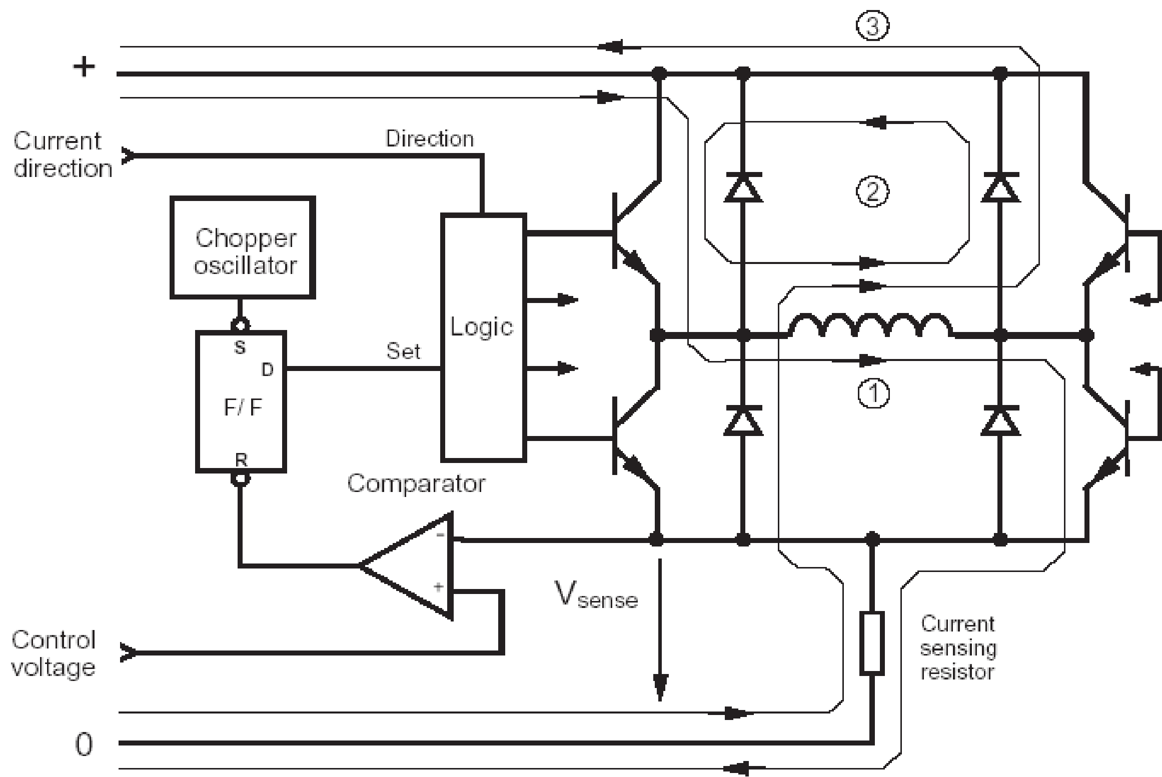



**Figure 5-11 H-Bridge Connected as Constant Current Chopper**

# 6. STEPPER MOTOR DRIVER IMPLEMENTATION

According to the theoretical background given in previous chapters, 3 stepper motor drivers are implemented, one for each axis. Drivers are based on ST's L297 Stepper Motor Controllers and L298 Dual Full-Bridge Drivers.

## 6.1 L297 – Stepper Motor Controller IC

The L297 IC integrates all the control circuitry required to control bipolar and unipolar stepper motors. It receives control signals from the system's controller, usually a microcomputer chip, and provides all the necessary drive signals for the power stage. Additionally, it includes two Pulse Width Modulation (PWM) chopper circuits to regulate the current in the motor windings.

With a suitable power actuator the L297 drives two phase bipolar permanent magnet motors, four phase unipolar permanent magnet motors and four phase variable reluctance motors.Moreover, it handles normal, wave drive and half step drive modes.

A feature of this device is that it requires only clock, direction and mode input signals to drive stepper motors. Since the phases are generated internally the burden on the microprocessor, and the programmer, is greatly reduced.

**Figure 6.1** shows the block diagram of the L297 integrated circuit. See Appendix 4 for detailed information about L297.

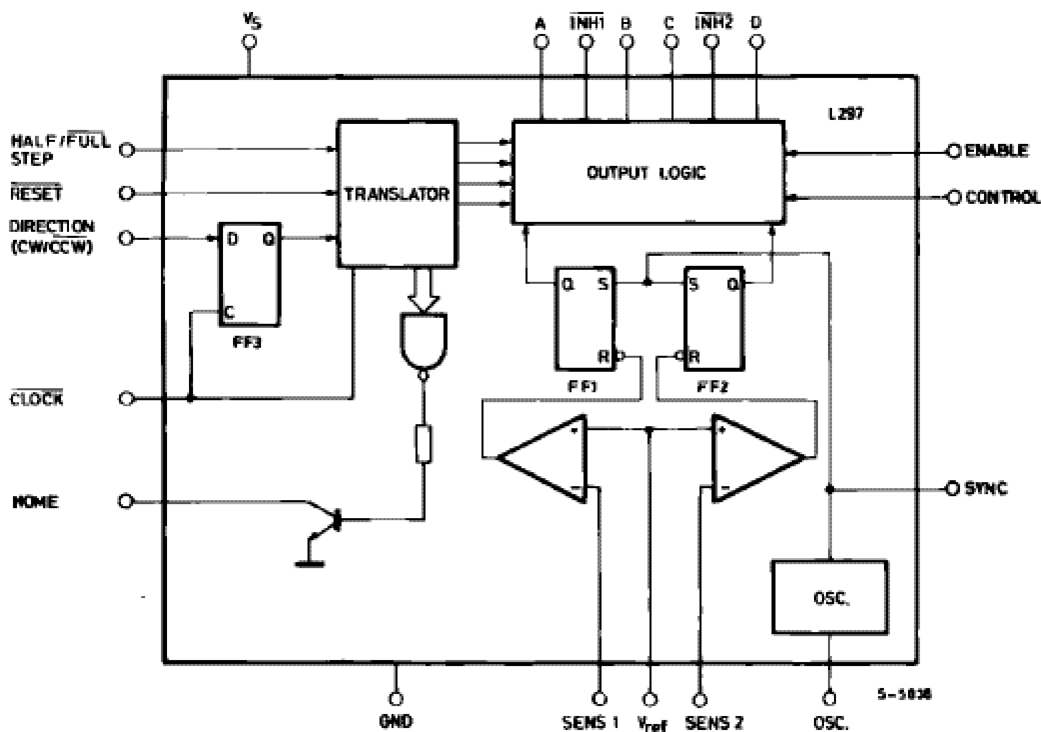



**Figure 6-1 L297 IC Block Diagram**

## 6.2 L298 – Dual Full-Bridge Driver IC

The L298 is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepper motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external terminal can be used for the connection of an external current sensing resistor. An additional supply input is provided so that the logic works at a lower voltage.

The L298 has an operating supply voltage up to 46V which is suitable for most stepper motors. (i.e. High operating voltage means higher rotation speeds in a stepper motor.) It can handle 2A/Phase motor currents with a total DC current up to 4A (two phases).

**Figure 6.2** shows the block diagram of the L298 integrated circuit. See Appendix 5 for detailed information about L298 IC.

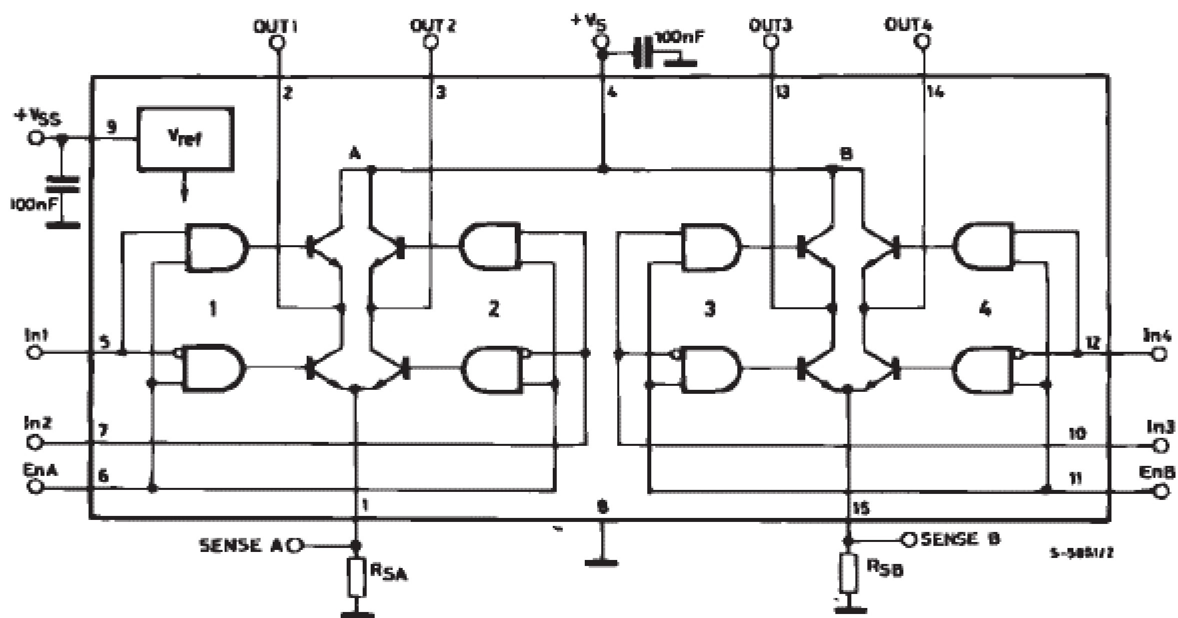
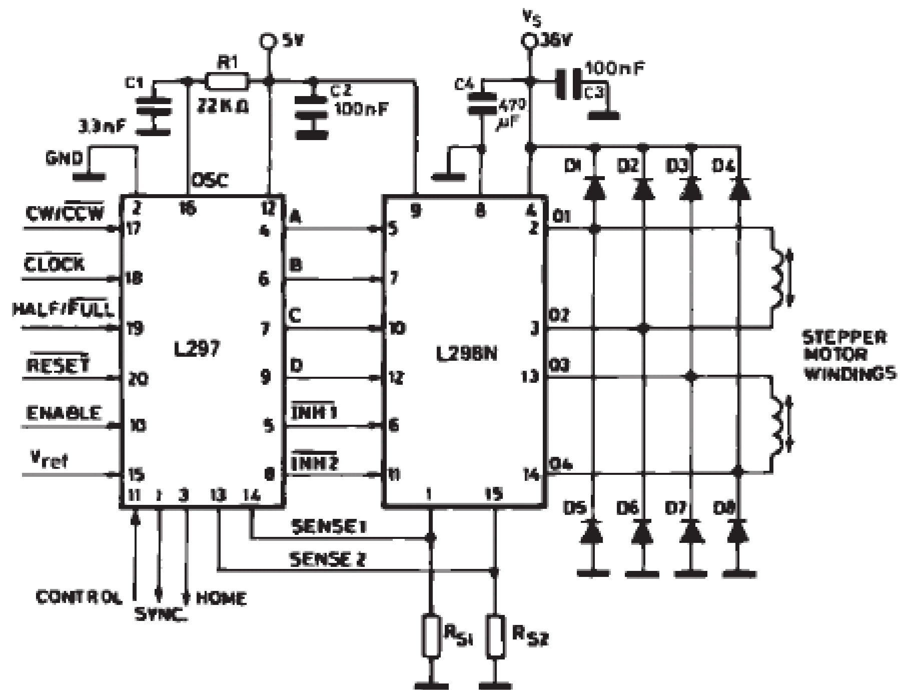



**Figure 6-2 L298 IC Block Diagram**

The outputs (Out1-Out4) must be connected to stepper motor coils by using external recirculating diodes to enable current paths for the motor windings. These diodes must be “Fast Recovery Diodes” with reverse recovery times smaller than 200ns to reduce switching noises during step changes.

A typical stepper motor driver designed with L297 and L298 ICs can be seen in Figure 6.3




$R_{S1} = R_{S2} = 0.5\,\Omega$

D1 to D8 = 2 A Fast diodes $\left\{ \begin{array}{l} V_F \leq 1.2\text{ V @ I} = 2\text{ A} \\ trr \leq 200\text{ ns} \end{array} \right.$

**Figure 6-3 Two Phase Bipolar Stepper Motor Circuit**

The stepper motor drivers used in this project are implemented according to the basic circuit given in **Figure 6.3**. Driver schematic is shown in Appendix 6.

# 7. CONTROL BOARD

## 7.1 Microcontrollers

Microcontroller is basicly a small computer which has in-built central processing unit (CPU), read only memory (ROM), random access memory (RAM) and peripheral input/output (I/O) ports. With the advance in VLSI technology, todays microcontrollers also can have A/D (Analog/Digital) - D/A (Digital/Analog) convertors, USART (Universal Synchronous Asynchronous Receiver Transmitter) and I2C (Inter Integrated Circuit) interfaces, comparators, timers, PWM (Pulse Width Modulation) modules packaged in a standard IC socket.

## 7.2 Microcontroller Selection

In this project, main tasks of the microcontroller are:

1. Communication with PC at reliable speeds
2. Generation of control signals for 3 stepper motor drivers
3. Limit switch inputs for axis overrange protection
4. Spindle speed control (As a future work)

To accomplish these tasks PIC16F877 microcontroller from Microchip is selected. This MCU comes with an in-built serial UART and 33 I/O pins which is suitable for above tasks.

## 7.3 PIC16F877 Features

**Microcontroller Core Features:**

- High-performance RISC CPU
- Only 35 single word instructions to learn
- All single cycle instructions except for program branches which are two cycle
- Operating speed: DC - 20 MHz clock input DC - 200 ns instruction cycle
- 8K x 14 words of FLASH Program Memory
- 368 x 8 bytes of Data Memory (RAM)
- 256 x 8 bytes of EEPROM data memory
- Interrupt capability (up to 14 sources)
- Eight level deep hardware stack
- Direct, indirect and relative addressing modes
- Power-on Reset (POR)

- Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation
- Programmable code-protection
- Power saving SLEEP mode
- Selectable oscillator options
- Low-power, high-speed CMOS FLASH/EEPROM technology
- Fully static design
- In-Circuit Serial Programming (ICSP) via two pins
- Single 5V In-Circuit Serial Programming capability
- In-Circuit Debugging via two pins
- Processor read/write access to program memory
- Wide operating voltage range: 2.0V to 5.5V
- High Sink/Source Current: 25 mA
- Commercial and Industrial temperature ranges
- Low-power consumption:
  < 2 mA typical @ 5V, 4 MHz
  20 uA typical @ 3V, 32 kHz
  < 1 uA typical standby current

**Peripheral Features:**

- Timer0: 8-bit timer/counter with 8-bit prescaler
- Timer1: 16-bit timer/counter with prescaler, can be incremented during sleep via external crystal/clock
- Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler
- Two Capture, Compare, PWM modules
  -Capture is 16-bit, max. resolution is 12.5 ns
  -Compare is 16-bit, max. resolution is 200 ns
  -PWM max. resolution is 10-bit
- 10-bit multi-channel Analog-to-Digital converter
- Synchronous Serial Port (SSP) with SPI (Master Mode) and I$^2$C (Master/Slave)
- Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) with 9-bit address detection
- Parallel Slave Port (PSP) 8-bits wide, with external RD, WR and CS controls
- Brown-out detection circuitry for Brown-out Reset (BOR)

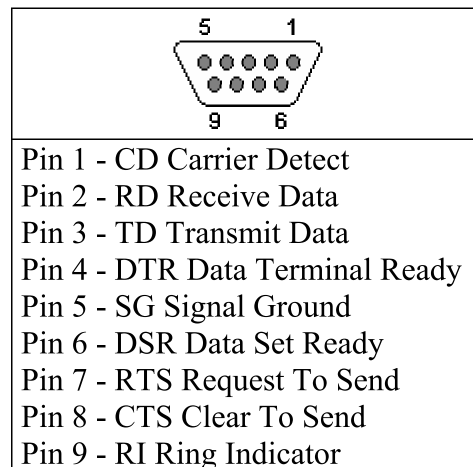
## 7.4 Control Board Implementation

### 7.4.1 Communication with PC

For the bidirectional data transmission with PC, hardware UART of PIC16F877 is used. For the 40-pin PDIP package pin 25 is UART TX (Transmit) and pin 26 is UART RX (Receive). When properly configured these pins are used for RS-232 formatted data at TTL signal levels. (i.e. 0V for low logic and 5V for high logic) This level has to be converted to proper RS-232 signal levels. (i.e. +12V for low logic and –12V for high logic)

This conversion can be done simply with a MAX232 Level Convertor IC from Maxim and four additional capacitors. Refer to [2] for details.

After level shifting transmit and receive lines are connected to Dsub-9 Female connector. **(Table 7.1)**




Pin 1 - CD Carrier Detect
Pin 2 - RD Receive Data
Pin 3 - TD Transmit Data
Pin 4 - DTR Data Terminal Ready
Pin 5 - SG Signal Ground
Pin 6 - DSR Data Set Ready
Pin 7 - RTS Request To Send
Pin 8 - CTS Clear To Send
Pin 9 - RI Ring Indicator

**Table 7-1 Dsub-9 Female Connector Pin Diagram**

### 7.4.2 Connection of Control Signals for Stepper Motor Drivers

Each stepper motor driver, implemented in Section 6 has a 10-pin IDC Male connector for input signal and power connections. (See **Table 7.2**)

| Pin No | Function |
|---|---|
| 1 | Enable |
| 2 | Vcc |
| 3 | Sync |
| 4 | Gnd |
| 5 | Dir |
| 6 | Gnd |
| 7 | Step |
| 8 | Gnd |
| 9 | Half/Full |
| 10 | Vcc |

**Table 7-2 Stepper Motor Drivers Input Connector Pin Diagram**

These pins are connected to PIC16F877 MCU over a 10-way ribbon cable and a 10-pin IDC Male connector on the control board. Since there are three stepper motor driver boards, there ribbon cables and connectors are used to make connection with PIC. **Table 7.3** shows the connection of each driver to PIC Mcu pins.

| PIC Pin No | Driver Axis | Driver Input Signal |
|---|---|---|
| 19 – RD0 | X | ModeX |
| 20 – RD1 | | StepX |
| 21 – RD2 | | DirX |
| 22 – RD3 | | EnableX |
| 27 – RD4 | Y | ModeY |
| 28 – RD5 | | StepY |
| 29 – RD6 | | DirY |
| 30 – RD7 | | EnableY |
| 37 – RB4 | Z | ModeZ |
| 38 – RB5 | | StepZ |
| 39 – RB6 | | DirZ |
| 40 – RB7 | | EnableZ |

**Table 7-3 Driver Connections to PIC MCU**

### 7.4.3 Limit Switch Inputs

Limit switches are used for axis overrange protection in CNC systems. When an axis reaches it’s limit, switch is pressed and the controller is informed that the axis is going to pass it’s limits, causing the controller to immediately stop the axis movement.

In this project, six limit switch inputs -two for each axis are used for this purpose. This is done by connecting each switch to a different I/O pin on the PIC MCU. This connection enables the microcontroller to know machine position when a limit switch is pressed. Also, in order to generate an external interrupt all switch inputs are connected to a 6-input AND gate. Output of this AND gate is tied to pin 33 of PIC MCU which is the external interrupt pin for the microcontroller.

According to the information given in 7.4.1 through 7.4.3, the circuit shown in Appendix 7 is designed and implemented.

## 7.5 Embedded Programming of PIC16F877

Embedded programming is the process of writing programs that will work on an integrated circuit such as a microcontroller, DSP (Digital Signal Processor) or an FPGA (Field Programmable Gate Array). It can be done by using assembly language or any available high level language. For PICMicro’s, assembly language or some C and BASIC cross

compilers are available. Inorder to take the advantage of C language, a cross compiler from Custom Computer Services (http://www.ccsinfo.com) is used for this project.

### 7.5.1 Block Diagram

Block diagram of the source code can be seen in Figure 7.1

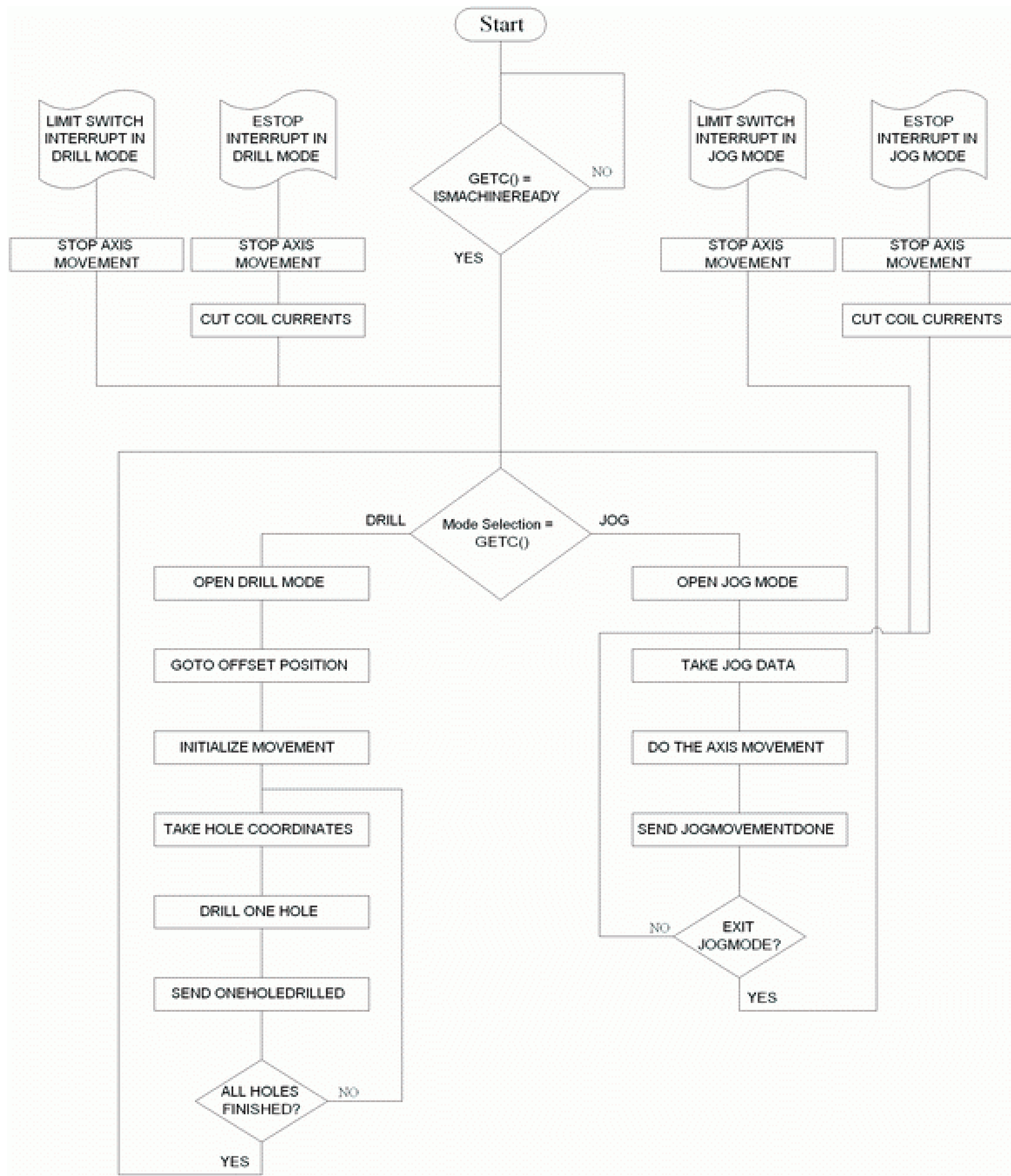



**Figure 7-1 Block Diagram of PIC Firmware**

### 7.5.2 Operation of PIC Firmware

When power is first applied to the control board PIC MCU performs some initilizations on I/O pins and starts waiting for an ISMACHINEREADY byte to come from RS-232 line. When ISMACHINEREADY byte comes from PC, MCU sends a MACHINEREADY byte as an answer. This process enables the full synchronization of PC software and PIC firmware.

There are two modes of operation for the program. One is “Jog Mode” which allows manual axis movements with predefined step numbers at a given direction. The other is “Drill Mode” which enables drilling of all the holes on a board working in a continuous loop. In both modes all the movement information is sent to PIC from PC software. This information transfer is manually held in Jog mode and automatically generated in Drill mode.

Next step in PIC firmware is mode selection. User can select Jog or Drill mode from the PC program. If Jog mode is selected (i.e. Right branch in Figure 7.1) by getting OPENJOGMODE byte from PC, MCU responses with a JOGMODEON byte (again for synchronization). Next, PIC waits for the 4-bytes long “Jog Data Packet”. (Figure 7.2)

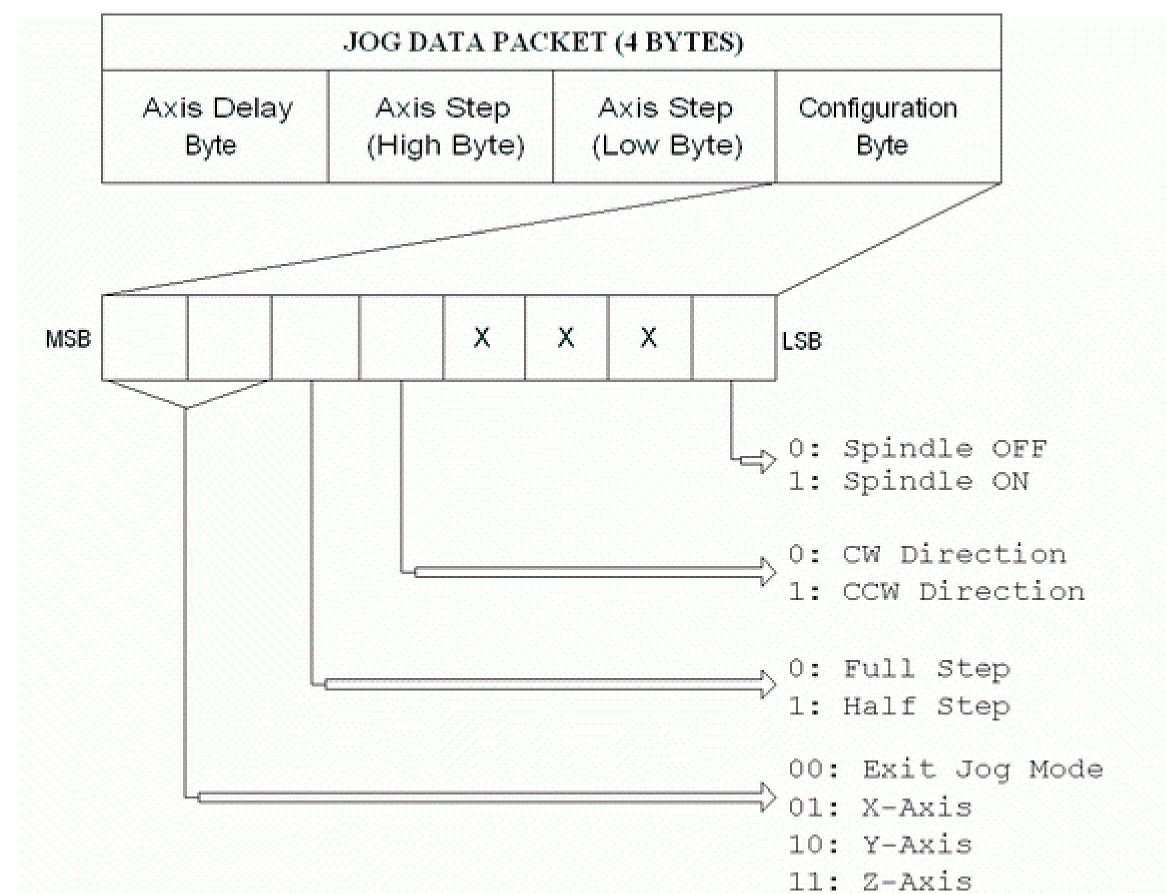



**Figure 7-2 Structure of Jog Data Packet**

Upon receiving this data, packet is internally processed in PIC for generation of axis movement with the given parameters. After this, PIC controls the stepper motor driver pins. (i.e. sends exact step pulses at given axis and direction) When movement is completed MCU sends JOGMODEDONE byte to PC to inform it for the transfer of new data packet. This cycle continues until an EXITJOGMODE command comes from PC. (i.e. 00 for most significant 2 bits of configuration byte)

If Drill mode is selected (i.e. Left branch in **Figure 7.1**) by getting OPENDRILLMODE1 byte from PC, MCU responses with a DRILLMODE1ON byte (again for synchronization). Next, PIC waits for a 10-bytes long data packet to goto offset location which is defined in the user program. Upon receiving this data PIC moves the necessary axises and brings drill head to offset position. It sends an ATOFFSETPOS byte at the end.

After this, PIC waits for the 9-bytes long "Initialize Drill Mode Packet". (**Figure 7.3**)

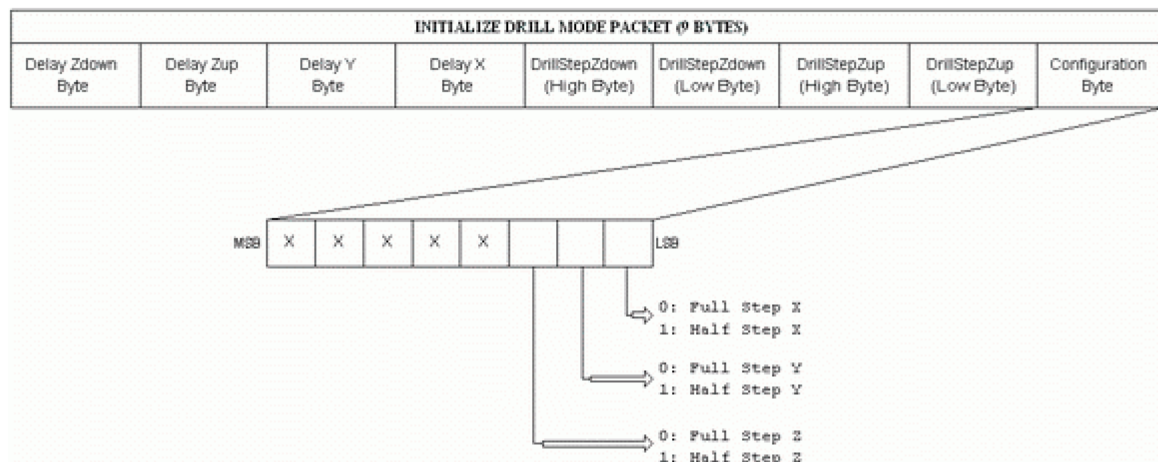



**Figure 7-3 Structure of Initialize Drill Mode Packet**

This packet is transfered only one time. It sends the following parameters to PIC:

- Driving modes of all axises (Half/Full)
- Position of drill head from offset location when moving from one hole to another (DrillStepZup)
- Drill depth position from offset location (DrillStepZdown)
- Axis Speeds of X,Y, Zdown and Zup (Drill head down and up speeds can be different to speed up drilling process)

When MCU receives this data packet, it assigns necessary variables and sets/resets proper control pins of stepper motor drivers. It sends a DRILLMODE1INITIALIZED byte at the end.

Next, PIC waits for the 5-bytes long "Drill Data Packet". (Figure 7.4)

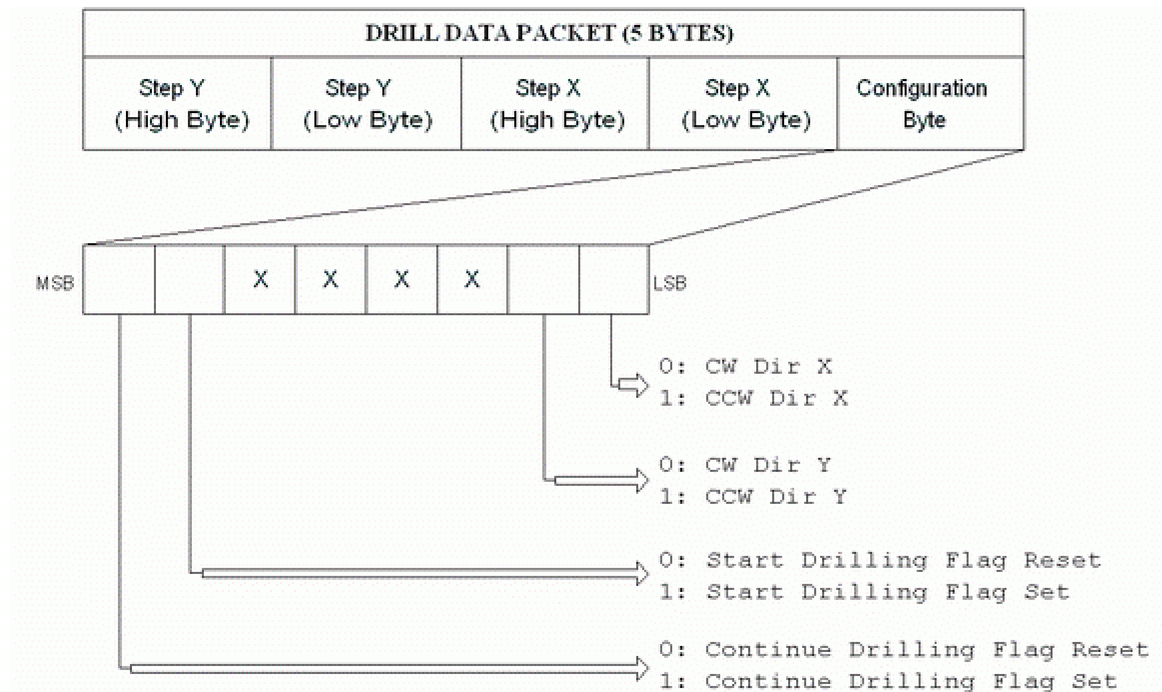



**Figure 7-4 Structure of Drill Data Packet**

When PIC receives drill data packet hence the drill coordinates, it drives the necessary pins according to initialize drill mode and drill data packets. This causes the drill head to go to drill coordinates, drill down with DelayZdown and drill up with DelayZup. Then PIC sends a ONEHOLEDRILLED byte and starts waiting for new drill data packet. This cycle continues until Bit7 of configuration byte is reset. (i.e. until all holes on the PCB are drilled)

In the PIC firmware both modes (Jog and Drill) exits to mode selection point (Refer to **Figure 7.1**). So main routine never finishes, allowing user to easily change the operating mode of the machine.

In addition to the main routine the firmware includes 2 interrupt service routines.

- External Interrupt
- RS-232 Receive Buffer Full Interrupt

External Interrupt: This interrupt is used for detecting the pressing of a limit switch for axis overrange protection. PIC16F877 has one external interrupt pin. It is pin 33 or RB0. To interrupt the CPU all limit switches are ANDed and output is fed to the interrupt pin. By doing this any state change on a limit switch will interrupt the CPU.

The external interrupt service routine waits for 300ms to debounce switch and then inputs all limit switch conditions via Port A. A simple comparision algorithm then finds which switch was pressed.

RS-232 Receive Buffer Full Interrupt: This interrupt is used for emergency stopping of the machine. Normally, interrupts are disabled. Thus, any data packet transfer does not cause an RS-232 interrupt. Before moving an axis either in Jog or Drill mode, interrupts are enabled till the end of the movement. This enables RS-232 and external interrupts to be served during a motion. Emergency stop command is an EMERGENCYSTOP byte. If this byte comes from RS-232 port during a motion, PIC MCU will immediately stop the axis movements and cut motor currents by disabling stepper motor drivers.

Refer to Appendix 1 for the firmware source code.

# 8. PC SOFTWARE

To control the PIC MCU, hence the machine movements, a computer software is developed. Software is written in Borland C++ Builder 5.0 environment. (Refer to Appendix 3 for software screenshots)

Main parts of the developed software are:

- A Graphical User Interface (GUI) to interact with the user
- Excellon Drill File Import
- Conversion of string hole coordinates to proper variables
- Jog movement to set offset coordinates
- Drill routine to drill all holes from the imported file
- Machine settings to change machine resolution, axis speeds, driving modes, etc.
- Com port settings to change RS-232 transmission parameters
- Tool index to show different tool diameters
- An emergency stop button to immediately stop machine movements

## 8.1 Excellon Drill Format

In PCB fabrication industry, the standard for drilling PCBs is known as Excellon Drill File Standard. This ASCII (American Standard Code for Information Interchange) file includes information about the location of each hole relative to the board offset. It also contains some other parameters like tool diameters, spindle speeds, feed rates, etc. Refer to [] for the details of Excellon file format.

Table 8.1 shows a sample Excellon drill file which is exported from OrCAD 9.0 Layout Plus PCB design program.

| **Lines** | **Description** |
|---|---|
| % | End of header (i.e. No header file for this file) |
| T1C0.034F200S100 | Tool No = 1, Diameter = 34 mils,<br>Feedrate = 200 IPM, Spindle = 100000 RPM |
| X003000Y003000 | X = 300 mil, Y = 300 mil (From Offset Position) |
| X004000Y003000 | X = 400 mil, Y = 300 mil (From Offset Position) |
| ... | .... |
| ... | ... |
| ... | ... |
| X009000Y003000 | X = 900 mil, Y = 300 mil (From Offset Position) |
| X009000Y006000 | X = 900 mil, Y = 600 mil (From Offset Position) |
| M30 | End of drill file |

**Table 8-1 Example of an Excellon Drill File**

## 8.2 Processing the Drill File

The coordinates in the drill file are in ASCII format. It must be converted to proper variables before being processed by the computer. For this purpose a structure is created to hold the drill information.

```
struct holeFile
{
  char T[3];
  char X[7];
  char Y[7];
};
holeFile holes[1000];
```

In this structure, T holds the 2-digit tool number, X holds the 6-digit X-coordinate and Y holds the 6-digit Y-coordinate. By using the holes[i], $0 < i < 999$ structure array, any expression can reach to hole informations where "i" represents the hole number. (i.e. A PCB having 20 holes will use holes[0] to holes[19] variables.)

## 8.3 Jog Movements

When user clicks Jog button on PC, the following steps are executed:

1. PC sends an OPENJOGMODE byte to PIC.
2. PIC responds with a JOGMODEON byte.
3. Jog form is opened on PC.
4. When user presses on an axis movement button, a 4-bytes long "Jog Data Packet" is constructed and sent to PIC according to the user parameters.
5. Upon completion of the movement PIC responds with a JOGMODEDONE byte.
6. PIC starts waiting for new Jog Data Packets.
7. If user closes Jog form, 4-bytes long 0's (zeros) are sent to PIC to close Jog mode.
8. PIC responds with a JOGMODEOFF byte and goes to mode selection.

## 8.4 Drill Movements

When user clicks Drill button on PC, the following steps are executed:

1. PC sends an OPENDRILLMODE1 byte to PIC
2. PIC responds with a DRILLMODE1ON byte.
3. PC calculates the axis displacements from the offset position. It constructs and sends a 10-bytes long data packet to PIC according to displacement values.
4. Upon completion of the offset movement, PIC responds with an ATOFFSETPOS byte and starts waiting for Initialize Drill Mode Packet.

5. PC constructs a 9-bytes long Initialize Drill Mode Packet according to machine settings and sends it to PIC.
6. PIC takes this packet and assigns necessary variables. It returns with a DRILLMODE1INITIALIZED byte.
7. When PC receives this byte, it converts holes[i] array elements to exact step numbers according to machine settings. Then it constructs a 5-bytes long Drill Data Packet and sends it to PIC.
8. PIC makes the axis movements (i.e. Drill one hole) and responds with a ONEHOLEDRILLED byte. Then it starts waiting for new Drill Data Packet.
9. PC counts the number of drilled holes inorder to know whether all holes are drilled. If not, it constructs and sends a new Drill Data Packet to PIC until all the holes on the imported drill file has been finished.
10. When all holes are drilled, PC sends a 0 bit in MSB of the configuration byte. (See Figure 7.4)
11. This causes PIC to exit from drill mode by responding with a JOGMODE1DONE byte and go to mode selection.

As a general summary, software controls the microcontroller and the machine movements with proper data packets. With the help of control bytes such as JOGMODEON, DRILLMODE1DONE, etc software is in full synchronization with the PIC firmware.

# 9. RESULTS AND CONCLUSION

In this project, specifications of a simple prototyping machine is achieved by drilling one hole in 5 seconds with a theoretical resolution of 4.375 microns. A maximum rotation speed of 187 rpm is achieved for surplus stepper motors. This corresponds to a 32.8 cm/minute feedrate.

Errors that might occur in manual drilling are totally eliminated with the 3-axis precise control of the drill head movements. So, drill bit positioning on a pad or breaking of tools is no more a problem.

In addition, PCBs can be drilled with this machine before the etching process. This provides smooth drill faces causing better solderability on the final PCBs. Also with the help of a chemical process, this system can be used to produce two sided and through hole plated printed circuit boards.

Project started at July 2002 and finished at the end of May 2003. Thus total development time for this project is 11 months.

In the future,

- Current system can be improved to reach higher axis speeds;
- Variable spindle speed control mechanism can be easily incorporated to the system;
- The developed system can be built up for milling PCBs.

# 10. REFERENCES

[1] Excellon Part Programming Commands
http://www.excellon.com
[2] Maxim Semiconductor MAX232 Data Sheet
http://www.maxim-ic.com
[3] ST Application Note – Stepper Motor Driving
http://www.st.com
[4] Geckodrive Inc. – Stepper Motor White Papers
http://www.geckodrive.com
[5] New Japan Radio Co. (JRC) – Stepper Motor Basics
http://www.njr.co.jp
[6] Anaheim Automation – Introduction to Stepper Motor Systems
http://www.anaheimautomation.com
[7] Douglas W. Jones – Control of Stepping Motors
http://www.cs.uiowa.edu/~jones/step
[8] Chris Reinders – The Zoltar Machine
http://www.rcmodels.net/cnc
[9] Srecko Lavric – Home-made CNC Drilling Machine
http://194.249.207.126/users/slavric/cnc_eng/cnc_eng.html
[10] Dejan Crnila - ComPort Library version 2.63
http://www2.arnes.si/~sopecrni

# 11. APPENDICES

## APPENDIX 1 - PIC Source Code (CCS PCWH PIC C Compiler)

```
//**********************************************************************
//*** MAXIMUS V1.0                                                   ***
//*** Hacettepe University 2003 Graduation Project                   ***
//*** Written by Alper YILDIRIM                                      ***
//*** 29.05.2003                                                     ***
//**********************************************************************

#include <16f877.h>     // Header definitions
#include <stdio.h>

#ORG 0x1F00,0x1FFF {}   // For the 8k 16F876/7 protect bootloader code

#fuses HS,NOWDT,NOPROTECT     // Configuration bits
#use delay(clock=20000000)    // Operating frequency
#use rs232(baud=115200, xmit=PIN_C6, rcv=PIN_C7)   // Enable RS-232
communication

#define MODE_X       PIN_D0   // PIC pin definitions
#define STEP_X       PIN_D1
#define DIR_X        PIN_D2
#define ENABLE_X     PIN_D3

#define MODE_Y       PIN_D4
#define STEP_Y       PIN_D5
#define DIR_Y        PIN_D6
#define ENABLE_Y     PIN_D7

#define MODE_Z       PIN_B4
#define STEP_Z       PIN_B5
#define DIR_Z        PIN_B6
#define ENABLE_Z     PIN_B7

#define SPINDLE_BIT  PIN_B2

#define PACKETLENGTH 10       // Define the length of Drill Data Packet
#define STEPSBACKATLIMIT 700  // Move 700 steps in limit switch interrupt

#define ISMACHINEREADY        0x61  // Byte definitions
#define MACHINEREADY          0x62
#define OPENJOGMODE           0x63
#define JOGMODEON             0x64
#define JOGMODEOFF            0x65
#define JOGMODEDONE           0x66
#define OPENDRILLMODE1        0x67
#define DRILLMODE1ON          0x68
#define ONEHOLEDRILLED        0x69
#define DRILLMODE1INITIALIZED 0x6A
#define DRILLMODE1DONE        0x70
#define OPENDRILLMODE2        0x71
#define DRILLMODE2ON          0x72
#define DRILLMODE2OFF         0x73
#define DRILLMODE2DONE        0x74
#define ATOFFSETPOS           0x75
#define EMERGENCYSTOP         0x80
#define MACHINESTOPPED        0x81
#define ERRORLIMITSWITCH      0x90
```

```
#define HOME_SWITCH_X   0x01     // Define outgoing bytes for
#define LIMIT_SWITCH_X  0x02     // Home and Limit Switches
#define HOME_SWITCH_Y   0x04
#define LIMIT_SWITCH_Y  0x08
#define HOME_SWITCH_Z   0x10
#define LIMIT_SWITCH_Z  0x20

// Function prototypes
void constspeed(int16 conststep, int8 delay, int8);
void constspeed2(int16 conststep, int8 delay, int8);  // For int_ext
routine
void constspeed_xy(int16 StepX, int16 StepY, int8 DelayX, int8 DelayY);
void dly_100us(int8 n);

// Global variables
int8  rs232_buffer;           // Stores the return data from getc()
function
int8  dataIn[PACKETLENGTH];   // Stores the incoming data packet from Com
Port
int16 jogDataInputLocation;   // Stores the address of the location where
interrupts will return
int16 modeSelectionLocation;  // Stores the address of the location where
interrupts will return
int1  currentMachineMode;     // MachineMode Flag 0=Jog 1=Drill
int8  axisDelay;              // used in int_ext routine & main()

#int_ext // RB0 External Interrupt Service Routine
void isr_ext()
{
// INTF Flag of INTCON register must be cleared if goto is used in isr
int8 switchConditions;
disable_interrupts(GLOBAL);
delay_ms(300); // debounce button
switchConditions=input_a();
switch (switchConditions)
   {
   case 0x3E:  // RA0=0
      output_bit(DIR_X,1);
      constspeed2(STEPSBACKATLIMIT,axisDelay,1);
      putc(HOME_SWITCH_X);
   break;
   case 0x3D:  // RA1=0
      output_bit(DIR_X,0);
      constspeed2(STEPSBACKATLIMIT,axisDelay,1);
      putc(LIMIT_SWITCH_X);
   break;
   case 0x3B:  // RA2=0
      output_bit(DIR_Y,1);
      constspeed2(STEPSBACKATLIMIT,axisDelay,2);
      putc(HOME_SWITCH_Y);
   break;
   case 0x37:  // RA3=0
      output_bit(DIR_Y,0);
      constspeed2(STEPSBACKATLIMIT,axisDelay,2);
      putc(LIMIT_SWITCH_Y);
   break;
   case 0x2F:  // RA4=0
      putc(HOME_SWITCH_Z);
   break;
   case 0x1F:  // RA5=0
      putc(LIMIT_SWITCH_Z);
```

```
   break;
   case 0x3F:  // Unpressed switch Interrupt
      return;
   break;
   default:
      //putc(ERRORLIMITSWITCH);    // On Error send ERRORLIMITSWITCH byte
      return;
   break;
   }
switch (currentMachineMode)
   {
   case 0:  // Jog Mode
      goto_address(jogDataInputLocation);
   break;
   case 1:  // Drill Mode
      goto_address(modeSelectionLocation);
   break;
   }
}  // End of RB0 External Interrupt Service Routine

#int_rda // UART Receive Data Interrupt Service Routine
void serial_isr()
{
rs232_buffer=getc();
switch (rs232_buffer)
   {
   case (EMERGENCYSTOP):
      // Disable all drivers,Cut motor coil currents
      output_bit(ENABLE_X,0);
      output_bit(ENABLE_Y,0);
      output_bit(ENABLE_Z,0);
      putc(MACHINESTOPPED);
      switch (currentMachineMode)
         {
         case 0:  // Jog Mode
            goto_address(jogDataInputLocation);
         break;
         case 1:  // Drill Mode
            goto_address(modeSelectionLocation);
         break;
         }
   break;
   }
}  // End of UART Receive Data Interrupt Service Routine

void main()
{
int16 stepX,stepY,stepZ,drillStepZup,drillStepZdown,axisStep;
int8  delayX,delayY,delayZdown,delayZup,axisSelect,config,jogconfig;
int1  modeX,modeY,modeZ,axisMode,dirX,dirY,dirZ,axisDir,spindle;

output_b(0xF0);   // Set all Driver Pins
output_d(0xFF);
set_tris_b(0x01); // RB0 is input for external interrupts
printf("MAXIMUS_V1.0");
jogDataInputLocation = label_address(JOG_DATA_INPUT);
modeSelectionLocation = label_address(MODE_SELECTION);

disable_interrupts(GLOBAL);   // Disable all Interrputs

while (getc()!=ISMACHINEREADY);   // getc() until ISMACHINEREADY has come
```

```
putc(MACHINEREADY);

MODE_SELECTION:
rs232_buffer=getc();

switch (rs232_buffer)
   {
   case (ISMACHINEREADY):
      putc(MACHINEREADY);
      goto MODE_SELECTION;
   break;
   case (OPENJOGMODE):
      putc(JOGMODEON);
      goto JOG_DATA_INPUT;
   break;
   case (OPENDRILLMODE1):
      putc(DRILLMODE1ON);
      goto DRILL_MODE1_INITIALIZE;
   break;
   case (OPENDRILLMODE2):  // !Not Completed!
      putc(DRILLMODE2ON);
      goto JOG_DATA_INPUT; // open jog mode to take 2 hole positions
   break;
   }

JOG_DATA_INPUT:

currentMachineMode = 0; // Machine is in Jog Mode

dataIn[0] = getc(); // Axis Configuration Byte
dataIn[1] = getc(); // Axis Step Low Byte
dataIn[2] = getc(); // Axis Step High Byte
dataIn[3] = getc(); // Axis Delay Byte

bit_clear(*0x0B,1);  // Clear INTF Flag of INTCON register

// if dataIn bit7 = 0 exit jog mode
if ((dataIn[0] >> 6) == 0)
   {
   putc(JOGMODEOFF);
   disable_interrupts(GLOBAL);
   goto MODE_SELECTION;
   }

// else, make the jog movement
axisStep = make16(dataIn[2],dataIn[1]);   // Construct Axis Step Data
axisDelay = dataIn[3];
jogconfig = dataIn[0];

// Construct control bits
axisSelect = (jogconfig & 0xC0) >> 6;     // 01=X, 10=Y, 11=Z
axisMode = jogconfig & 0x20;              // 0=FULL, 1=HALF
axisDir = jogconfig & 0x10;               // 0=CW, 1=CCW
Spindle = jogconfig & 0x01;               // 0=SpindleOFF, 1=SpindleON

// Before movement, enable RS232 and RB0 interrupts for
// Emergency Stop and Axis Limit Switches, respectively
ext_int_edge(H_TO_L);         // RB0 interrupts on falling edge
enable_interrupts(INT_EXT);
enable_interrupts(INT_RDA);
enable_interrupts(GLOBAL);
```

```
// Enable all drivers, i.e Energize motor coils
output_bit(ENABLE_X,1);
output_bit(ENABLE_Y,1);
output_bit(ENABLE_Z,1);

output_bit(SPINDLE_BIT,Spindle); // Initialize Spindle Pin

// Do the axis movement according to given parameters
switch (axisSelect)
   {
   case 1:  // 1=01b -> X-Axis
      output_bit(MODE_X,axisMode);
      output_bit(DIR_X,axisDir);
      constspeed(axisStep,axisDelay,1); // (Stepnumber,delay,pin)
   break;
   case 2:  // 2=10b -> Y-Axis
      output_bit(MODE_Y,axisMode);
      output_bit(DIR_Y,axisDir);
      constspeed(axisStep,axisDelay,2); // (Stepnumber,delay,pin)
   break;
   case 3:  // 3=11b -> Z-Axis
      output_bit(MODE_Z,axisMode);
      output_bit(DIR_Z,axisDir);
      constspeed(axisStep,axisDelay,3); // (Stepnumber,delay,pin)
   break;
   }

putc(JOGMODEDONE);      // Inform PC that Movement has been completed
disable_interrupts(GLOBAL);
goto JOG_DATA_INPUT;    // Wait for new coordinates at JOG_DATA_INPUT

DRILL_MODE1_INITIALIZE: // Drill using offset position of the machine

currentMachineMode = 1; // Machine is in Drill Mode

dataIn[0] = getc();  // Movement Configuration Byte
dataIn[1] = getc();  // Step X Low Byte
dataIn[2] = getc();  // Step X High Byte
dataIn[3] = getc();  // Step Y Low Byte
dataIn[4] = getc();  // Step Y High Byte
dataIn[5] = getc();  // Step Z Low Byte
dataIn[6] = getc();  // Step Z High Byte
dataIn[7] = getc();  // Delay X Byte
dataIn[8] = getc();  // Delay Y Byte
dataIn[9] = getc();  // Delay Z Byte

bit_clear(*0x0B,1);  // Clear INTF Flag of INTCON register

// Construct Step and Delay bytes
config = dataIn[0];
StepX = make16(dataIn[2],dataIn[1]);
StepY = make16(dataIn[4],dataIn[3]);
StepZ = make16(dataIn[6],dataIn[5]);
DelayX = dataIn[7];
DelayY = dataIn[8];
DelayZup = dataIn[9];

// Construct Mode and Direction bits
ModeX = config & 0x08;
ModeY = config & 0x10;
```

```
ModeZ = config & 0x20;
DirX = config & 0x01;
DirY = config & 0x02;
DirZ = config & 0x04;

// Before movement, enable RS232 and RB0 interrupts for
// Emergency Stop and Axis Limit Switches, respectively
ext_int_edge(H_TO_L);         // RB0 interrupts on falling edge
enable_interrupts(INT_EXT);
enable_interrupts(INT_RDA);
enable_interrupts(GLOBAL);

// Enable all drivers, i.e Energize motor coils
output_bit(ENABLE_X,1);
output_bit(ENABLE_Y,1);
output_bit(ENABLE_Z,1);

// Initialize Mode and Direction pins
output_bit(MODE_X,ModeX);
output_bit(MODE_Y,ModeY);
output_bit(MODE_Z,ModeZ);
output_bit(DIR_X,DirX);
output_bit(DIR_Y,DirY);
output_bit(DIR_Z,DirZ);

// Do the axis movement
constspeed(StepZ,DelayZup,3);    // Z
constspeed(StepX,DelayX,1);      // X
constspeed(StepY,DelayY,2);      // Y

putc(ATOFFSETPOS);   // Inform PC that Offset Movement has been completed

disable_interrupts(GLOBAL);   // Disable Interrupts before taking data
packet

// Initialize Drill Mode
dataIn[0] = getc();  // Movement Configuration Byte (ModeX,ModeY,ModeZ)
dataIn[1] = getc();  // drillStepZup Low Byte
dataIn[2] = getc();  // drillStepZup High Byte
dataIn[3] = getc();  // drillStepZdown Low Byte
dataIn[4] = getc();  // drillStepZdown High Byte
dataIn[5] = getc();  // delayX Byte
dataIn[6] = getc();  // delayY Byte
dataIn[7] = getc();  // delayZup Byte
dataIn[8] = getc();  // delayZdown Byte

config = dataIn[0];
drillStepZup = make16(dataIn[2],dataIn[1]);
drillStepZdown = make16(dataIn[4],dataIn[3]);
DelayX = dataIn[5];
DelayY = dataIn[6];
DelayZup = dataIn[7];
DelayZdown = dataIn[8];

ModeX = config & 0x01;
ModeY = config & 0x02;
ModeZ = config & 0x04;

output_bit(MODE_X,ModeX);
output_bit(MODE_Y,ModeY);
output_bit(MODE_Z,ModeZ);
```

```
output_bit(DIR_Z,1);                        // Z down
constspeed(drillStepZup,DelayZup,3);        // Z

putc(DRILLMODE1INITIALIZED);  // Inform PC that DrillMode1 has been
initialized
// End of Initialize Drill Mode

DRILL_MODE1_INPUT:

dataIn[0] = getc(); // Movement Configuration Byte (Drill?,DirX,DirY)
dataIn[1] = getc(); // drillStepZup Low Byte
dataIn[2] = getc(); // drillStepZup High Byte
dataIn[3] = getc(); // drillStepZdown Low Byte
dataIn[4] = getc(); // drillStepZdown High Byte

if (!(dataIn[0] >> 7))  // If Continue Drilling Flag Reset exit Drill
Mode
   {
   putc(DRILLMODE1DONE);   // inform PC that drilling has been completed
   disable_interrupts(GLOBAL);
   goto MODE_SELECTION;    // Goto and Wait at MODE_SELECTION
   }

if ((dataIn[0] & 0x40) >> 6)  // If Start Drilling Flag Set return
without drilling
   {
   putc(ONEHOLEDRILLED);   // Machine is ready for taking next hole
coordinates
   disable_interrupts(GLOBAL);   // Disable Interrupts before taking data
packet
   goto DRILL_MODE1_INPUT; // Wait for new hole coordinates
   }

// else, continue drilling
config = dataIn[0];
StepX = make16(dataIn[2],dataIn[1]);
StepY = make16(dataIn[4],dataIn[3]);

DirX = config & 0x01;
DirY = config & 0x02;

output_bit(DIR_X,DirX);
output_bit(DIR_Y,DirY);

// Before movement, enable RS232 and RB0 interrupts for
// Emergency Stop and Axis Limit Switches, respectively
ext_int_edge(H_TO_L);         // RB0 interrupts on falling edge
enable_interrupts(INT_EXT);
enable_interrupts(INT_RDA);
enable_interrupts(GLOBAL);

// Do the Dual Axis movement
constspeed_xy(StepX,StepY,DelayX,DelayY);

output_bit(DIR_Z,1);                        // Z down
constspeed(drillStepZdown,DelayZdown,3);    // Z
output_bit(DIR_Z,0);                        // Z up
constspeed(drillStepZdown,DelayZup,3);      // Z
```

```
putc(ONEHOLEDRILLED);   // Machine is ready for taking next hole
coordinates
disable_interrupts(GLOBAL);   // Disable Interrupts before taking data
packet

goto DRILL_MODE1_INPUT; // Wait for new hole coordinates

}  // End of main()

void constspeed_xy(int16 StepX, int16 StepY, int8 DelayX, int8 DelayY)
{
int1  axis;
int8  DelayXY;
int16 StepXY,StepMore,i;

if (DelayX < DelayY)    // DelayXY is the speed of slow axis
   DelayXY = DelayY;
else
   DelayXY = DelayX;

if (StepX > StepY)      // StepXY is the step number of the smallest
movement
   {
   StepXY = StepY;
   axis = 0;            // axis = X axis
   StepMore = StepX - StepY;
   }
else
   {
   StepXY = StepX;
   axis = 1;            // axis = Y axis
   StepMore = StepY - StepX;
   }

// Do the dual axis movement
for (i=0; i<StepXY; i++)
   {
   output_bit(STEP_X,1);   // StepX pin HIGH
   output_bit(STEP_Y,1);   // StepY pin HIGH
   dly_100us(DelayXY);
   output_bit(STEP_X,0);   // StepX pin LOW
   output_bit(STEP_Y,0);   // StepY pin LOW
   dly_100us(DelayXY);
   }
output_bit(STEP_X,1);      // Add the last step on X axis
output_bit(STEP_Y,1);      // Add the last step on Y axis

// Do the remaining single axis movement
switch (axis)
   {
   case 0:  // X Axis
      for (i=0; i<StepMore; i++)
      {
      output_bit(STEP_X,1);   // Step pin HIGH
      dly_100us(DelayX);
      output_bit(STEP_X,0);   // Step pin LOW
      dly_100us(DelayX);
      }
      output_bit(STEP_X,1);   // Add last step
   break;
   case 1:  // Y Axis
```

```
      for (i=0;i<StepMore;i++)
      {
      output_bit(STEP_Y,1);   // Step pin HIGH
      dly_100us(DelayY);
      output_bit(STEP_Y,0);   // Step pin LOW
      dly_100us(DelayY);
      }
      output_bit(STEP_Y,1);   // Add last step
   break;
   }
}  // End of constspeed_xy()

void constspeed(int16 conststep, int8 delay, int8 pin)
// Build conststep number of step pulses
// Frequency = 1/(2*delay*100us)
{
int16 i;

switch (pin) {
   case 1:
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_X,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_X,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_X,1);   // Add last step
   break;
   case 2:
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_Y,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_Y,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_Y,1);   // Add last step
   break;
   case 3:
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_Z,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_Z,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_Z,1);   // Add last step
   break;
   }
}  // End of constspeed()

// Same as constspeed() funtion (used for int_ext routine)
void constspeed2(int16 conststep, int8 delay, int8 pin)
// Build conststep number of step pulses
// Frequency = 1/(2*delay*100us)
{
int16 i;

switch (pin) {
   case 1:
```

```
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_X,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_X,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_X,1);   // Add last step
   break;
   case 2:
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_Y,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_Y,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_Y,1);   // Add last step
   break;
   case 3:
      for (i=0;i<conststep;i++)
      {
      output_bit(STEP_Z,1);   // Step pin HIGH
      dly_100us(delay);
      output_bit(STEP_Z,0);   // Step pin LOW
      dly_100us(delay);
      }
      output_bit(STEP_Z,1);   // Add last step
   break;
   }
}  // End of constspeed2()

void dly_100us(int8 n)  // 3us+(1.2us+d)*n
{
// Delay for n * 100us
for(;n!=0;n--)
delay_us(98); // d is exactly 98.8 us
delay_cycles(4); // 0.8 us for 20Mhz oscillator
}
```

# APPENDIX 2 - Software Source Code (Borland C++ Builder 5.0)

## Maxdriller.cpp

```
//---------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
USERES("maxdriller.res");
USEFORM("mainform.cpp", FormMain);
USEFORM("aboutform.cpp", AboutBox);
USEFORM("jogform.cpp", FormJog);
USEFORM("settingsform.cpp", FormMachineSettings);
USEFORM("toolform.cpp", FormToolIndex);
//---------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
         Application->Initialize();
         Application->CreateForm(__classid(TFormMain), &FormMain);
       Application->CreateForm(__classid(TAboutBox), &AboutBox);
       Application->CreateForm(__classid(TFormJog), &FormJog);
       Application->CreateForm(__classid(TFormMachineSettings),
&FormMachineSettings);
       Application->CreateForm(__classid(TFormToolIndex),
&FormToolIndex);
       Application->Run();
    }
    catch (Exception &exception)
    {
         Application->ShowException(&exception);
    }
    return 0;
}
//-------------------------------------------------------------------------
```

## Mainform.cpp

```
//-------------------------------------------------------------------------
#include <vcl.h>
#include <stdlib.h>
#include <math.h>
#include "stdio.h"  // For File Input routines
#pragma hdrstop

#include "mainform.h"
#include "aboutform.h"
#include "jogform.h"
#include "settingsform.h"
#include "toolform.h"
//-------------------------------------------------------------------------
#pragma package(smart_init)
#pragma link "dfsToolBar"
#pragma link "CPort"
#pragma link "AdvToolBtn"
#pragma resource "*.dfm"
TFormMain *FormMain;
//-------------------------------------------------------------------------
// Global variables
int   i,j;
int   lineCounter = 0;
```

```
int   toolCounter = 0;
float scaleFactorX,scaleFactorY;
char  comPortFlag;                   // Used to understand if data read
from Com Port
bool  drillCompleteFlag=0;           // Used to understand if drill
completed (ComPortRxChar)
char  currentMachineMode;            // 0 = Jog , 1 = Drill Mode1 , 2 =
Drill Mode2
bool  fileOpenedFlag;                // 0 = No file opened , 1 = File
opened
bool  machineOnFlag;                 // 0 = MachineOFF , 1 = MachineON
bool  warnForToolChange;             // 0 = Drill all holes without
warning , 1 = Warn when tool needs to be changed

double   stepsPerMicronX,stepsPerMicronY,stepsPerMicronZ;
double   pointOneMilStepsX,pointOneMilStepsY,pointOneMilStepsZ;
char
machineDelayX,machineDelayY,machineDelayZdown,machineDelayZup,whichAxis;
int   drillPositionX,drillPositionY,positionX,positionY,positionZ;  //
distances from offset position
bool  machineModeX,machineModeY,machineModeZ;
bool  machineInvertDirX,machineInvertDirZ,machineInvertDirY;
bool  dirX,dirY,dirZ;

// Variables for Drill Mode2
int drillX,drillY;
double differenceStepX,differenceStepY;
double angleTeta,angleAlpha;

int Xpixel,Ypixel,holeRadius=DRILLHOLERADIUS;
int previousARow,previousACol;      // Coordinates of lastly selected
hole (Used in StringGrid OnSelectCell)
int holeNumber=0;                   // Used in drillOneHole()

char jogData[JOGPACKETLENGTH];      // 1 byte Axis Configuration
                                    // 2 byte Axis Step
char dataOut[PACKETLENGTH];

unsigned short int axisStep;        // 16 bit Step Data variable
unsigned short int drillStepX,drillStepY,drillStepZup,drillStepZdown;

AnsiString FileName;                // For OpenDialog Box
unsigned char outputBuffer[10];     // Serial Port Output Data Buffer
unsigned char inputBuffer[10];      // Serial Port Input Data Buffer
//-------------------------------------------------------------------------
// Create a structure to hold drill data
struct holeFile
{
  char T[3];
  char X[7];
  char Y[7];
};

holeFile holes[1000];
//-------------------------------------------------------------------------
// Create a structure to hold tool diameters
struct toolDiameter
{
   char TI[3];
   char D[6];
};
```

```
toolDiameter tools[99];
//---------------------------------------------------------------------------
__fastcall TFormMain::TFormMain(TComponent* Owner)
    : TForm(Owner)
{
StringGrid1->Cells[0][0]="Hole";
StringGrid1->Cells[1][0]="Tool";
StringGrid1->Cells[2][0]="X(.1 mil)";
StringGrid1->Cells[3][0]="Y(.1 mil)";
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::drillOneHole(int holeNumber)
{
drillStepX = (machineModeX + 1) * (StrToInt(holes[holeNumber].X)) *
pointOneMilStepsX;
drillStepY = (machineModeY + 1) * (StrToInt(holes[holeNumber].Y)) *
pointOneMilStepsY;
drillPositionX = drillStepX - positionX;
drillPositionY = drillStepY - positionY;
positionX += drillPositionX;  // data dönmeden pozisyonları update etmek
zorunda kaldım!
positionY += drillPositionY;

for(i=0;i<5;i++)
   dataOut[i] = 0;

if (drillPositionX > 0)
   {
   dirX = (machineInvertDirX ^ 1);                     // X right
   dataOut[1] = drillPositionX;                        // drillStepX LOW
Byte
   dataOut[2] = drillPositionX >> 8;                   // drillStepX HIGH
Byte
   }
   else if (drillPositionX < 0)
   {
   dirX = (machineInvertDirX ^ 0);                     // X left
   dataOut[1] = (drillPositionX * -1);                 // drillStepX LOW
Byte
   dataOut[2] = (drillPositionX * -1) >> 8;            // drillStepX HIGH
Byte
   }

if (drillPositionY > 0)
   {
   dirY = (machineInvertDirY ^ 1);                     // Y up
   dataOut[3] = drillPositionY;                        // drillStepY LOW
Byte
   dataOut[4] = drillPositionY >> 8;                   // drillStepY HIGH
Byte
   }
else if (drillPositionY < 0)
   {
   dirY = (machineInvertDirY ^ 0);                     // Y down
   dataOut[3] = (drillPositionY * -1);                 // drillStepY LOW
Byte
   dataOut[4] = (drillPositionY * -1) >> 8;            // drillStepY HIGH
Byte
   }
```

```
dataOut[0] = 0x80 + (dirY << 1) + dirX;   // Continue Drilling Flag Set

ComPort1->Write(dataOut,5); // Send drill coordinates until all holes
drilled

}
//---------------------------------------------------------------------------
void __fastcall TFormMain::drillOneHole2(int holeNumber)
{
double tempX,tempY;

angleAlpha = atan(double(StrToInt(holes[holeNumber].Y)-2000) /
double(StrToInt(holes[holeNumber].X)-2000));
tempX = sqrt(pow(StrToInt(holes[holeNumber].X)-2000,2) +
pow(StrToInt(holes[holeNumber].Y)-2000,2)) * cos(angleAlpha + angleTeta);
tempY = sqrt(pow(StrToInt(holes[holeNumber].X)-2000,2) +
pow(StrToInt(holes[holeNumber].Y)-2000,2)) * sin(angleAlpha + angleTeta);
/* Edit3->Text = tempX;
   Edit4->Text = tempY;
   Edit5->Text = angleAlpha; */
drillStepX = (machineModeX + 1) * (tempX) * pointOneMilStepsX;
drillStepY = (machineModeY + 1) * (tempY) * pointOneMilStepsY;
drillPositionX = drillStepX - positionX;
drillPositionY = drillStepY - positionY;
positionX += drillPositionX;  // data dönmeden pozisyonları update etmek
zorunda kaldım!
positionY += drillPositionY;

for(i=0;i<5;i++)
   dataOut[i] = 0;

if (drillPositionX > 0)
   {
   dirX = (machineInvertDirX ^ 1);                      // X right
   dataOut[1] = drillPositionX;                         // drillStepX LOW
Byte
   dataOut[2] = drillPositionX >> 8;                    // drillStepX HIGH
Byte
   }
   else if (drillPositionX < 0)
   {
   dirX = (machineInvertDirX ^ 0);                      // X left
   dataOut[1] = (drillPositionX * -1);                  // drillStepX LOW
Byte
   dataOut[2] = (drillPositionX * -1) >> 8;             // drillStepX HIGH
Byte
   }

if (drillPositionY > 0)
   {
   dirY = (machineInvertDirY ^ 1);                      // Y up
   dataOut[3] = drillPositionY;                         // drillStepY LOW
Byte
   dataOut[4] = drillPositionY >> 8;                    // drillStepY HIGH
Byte
   }
else if (drillPositionY < 0)
   {
   dirY = (machineInvertDirY ^ 0);                      // Y down
   dataOut[3] = (drillPositionY * -1);                  // drillStepY LOW
Byte
```

```
   dataOut[4] = (drillPositionY * -1) >> 8;           // drillStepY HIGH
Byte
   }

dataOut[0] = 0x80 + (dirY << 1) + dirX;   // Continue Drilling Flag Set

ComPort1->Write(dataOut,5); // Send drill coordinates until all holes
drilled
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::limitPositionOnDrillMode()
{
GoJog->Enabled = true;
GoEStop->Enabled = false;
GoCheckMachine->Enabled = true;
holeNumber = 0;   // Reset holeNumber inorder to start from first hole on
next drill
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::initializeDrillMode()
{
// This function constructs DrillMode initialization data and sends it to
PIC
dataOut[0] = (machineModeX) + (machineModeY << 1) + (machineModeZ << 2);
dataOut[1] = drillStepZup;
dataOut[2] = drillStepZup >> 8;
dataOut[3] = (drillStepZdown - drillStepZup);      // Can be done in PIC
dataOut[4] = (drillStepZdown - drillStepZup) >> 8;
dataOut[5] = machineDelayX;
dataOut[6] = machineDelayY;
dataOut[7] = machineDelayZup;
dataOut[8] = machineDelayZdown;
ComPort1->Write(dataOut,9);
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::drawCircle(int X, int Y, int Radius,TColor
InnerColor)
{
// This function draws a circle on a TImage Object with given parameters
Image1->Canvas->Pen->Color = clBlack;
Image1->Canvas->Brush->Color = InnerColor;

Image1->Canvas->Ellipse(X-Radius, Y-Radius, X+Radius, Y+Radius);
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::markHole(int X, int Y, int Radius,TColor
BorderColor)
{
// This function draws an arc on a TImage Object with given parameters
Image1->Canvas->Pen->Color = BorderColor;
Image1->Canvas->Brush->Style = bsClear;
Image1->Canvas->Ellipse(X-Radius, Y-Radius, X+Radius, Y+Radius);
//Image1->Canvas->Pen->Color = clBlack;
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::exitJogMode()
{
jogData[0] = 0; // Send 00xxxxxxb to exit from jog mode
jogData[1] = 0;
jogData[2] = 0;
jogData[3] = 0;
```

```
ComPort1->Write(jogData,JOGPACKETLENGTH); // Exit from Jog mode
comPortFlag = ComPort1->Read(inputBuffer,1);

if (comPortFlag == 0)
   {
   Application->MessageBox("Machine is not responding. Check connections
and reset PIC","Jog Mode Close Error", MB_OK);
   GoJog->Enabled = false;
   }
else if (comPortFlag == 1){
   if (inputBuffer[0] == JOGMODEOFF)
      {
      if (!currentMachineMode)
         MemoMachineInfo->Lines->Add(">> Jog mode successfully closed");
      }
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Jog Mode Close Error", MB_OK);
      GoJog->Enabled = false;
      }
   }
else
   {
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Jog Mode Close Error", MB_OK);
   GoJog->Enabled = false;
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::updatePositionDisplays()
{
//LblPositionX->Caption = IntToStr(int(positionX / (pointOneMilStepsX *
10* (machineModeX+1)))+1);
LblPositionX->Caption = IntToStr(positionX) + " steps";
LblPositionY->Caption = IntToStr(positionY) + " steps";
LblPositionZ->Caption = IntToStr(positionZ) + " steps";
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::gotoOffsetXInJogMode()
{
outputBuffer[0] = OPENJOGMODE;
ComPort1->Write(outputBuffer,1);
comPortFlag = ComPort1->Read(inputBuffer,1);

if (comPortFlag == 0)
   {
   Application->MessageBox("Machine is not responding. Check connections
and try again","Jog Mode Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized");
   }
else if (comPortFlag == 1)
   {
   if (inputBuffer[0] == JOGMODEON)
      {
      for (i=0;i<JOGPACKETLENGTH;i++)
         dataOut[i]=0;

      if (positionX > 0)
         {
         dirX = (machineInvertDirX ^ 0);       // X left
```

```
         dataOut[1] = positionX;                              // StepX LOW
Byte
         dataOut[2] = positionX >> 8;                         // StepX HIGH
Byte
         }
      else if (positionX < 0)
         {
         dirX = (machineInvertDirX ^ 1);       // X right
         dataOut[1] = (positionX * -1);                       // StepX LOW
Byte
         dataOut[2] = (positionX * -1) >> 8;                  // StepX HIGH
Byte
         }

      dataOut[0] = 0x41 + (machineModeX << 5) + (dirX << 4);
      dataOut[3] = machineDelayX;
      ComPort1->Write(dataOut,JOGPACKETLENGTH);
      whichAxis = 7; // OffsetX in Jog Mode
      }
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
      MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized");
      }
   }
else
   {
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::gotoOffsetYInJogMode()
{
for (i=0;i<JOGPACKETLENGTH;i++)
   dataOut[i]=0;

if (positionY > 0)
   {
   dirY = (machineInvertDirY ^ 0);       // Y down
   dataOut[1] = positionY;                              // StepY LOW Byte
   dataOut[2] = positionY >> 8;                         // StepY HIGH Byte
   }
else if (positionY < 0)
   {
   dirY = (machineInvertDirY ^ 1);       // Y up
   dataOut[1] = (positionY * -1);                       // StepY LOW Byte
   dataOut[2] = (positionY * -1) >> 8;                  // StepY HIGH Byte
   }

dataOut[0] = 0x81 + (machineModeY << 5) + (dirY << 4);
dataOut[3] = machineDelayY;
ComPort1->Write(dataOut,JOGPACKETLENGTH);
whichAxis = 8; // OffsetY in Jog Mode
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::gotoOffsetZInJogMode()
{
for (i=0;i<JOGPACKETLENGTH;i++)
```

```
   dataOut[i]=0;

if (positionZ > 0)
   {
   dirZ = (machineInvertDirZ ^ 0);       // Z up
   dataOut[1] = positionZ;                            // StepZ LOW Byte
   dataOut[2] = positionZ >> 8;                       // StepZ HIGH Byte
   }
else if (positionZ < 0)
   {
   dirZ = (machineInvertDirZ ^ 1);       // Z down
   dataOut[1] = (positionZ * -1);                     // StepZ LOW Byte
   dataOut[2] = (positionZ * -1) >> 8;                // StepZ HIGH Byte
   }

dataOut[0] = 0xC1 + (machineModeZ << 5) + (dirZ << 4);
dataOut[3] = machineDelayZup;
ComPort1->Write(dataOut,JOGPACKETLENGTH);
whichAxis = 9; // OffsetZ in Jog Mode
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::gotoOffsetPosition()
{
ComPort1->Open();
outputBuffer[0] = OPENDRILLMODE1;
ComPort1->Write(outputBuffer,1);
comPortFlag = ComPort1->Read(inputBuffer,1);
if (comPortFlag == 0)
   {
   Application->MessageBox("Machine is not responding. Check connections
and try again","Drill Mode Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Drill Mode can not be initialized.");
   }
else if (comPortFlag == 1)
   {
   if (inputBuffer[0] == DRILLMODE1ON)
      MemoMachineInfo->Lines->Add(">> Drill Mode1 On.");
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Drill Mode Error", MB_OK);
      MemoMachineInfo->Lines->Add(">> Drill Mode can not be
initialized.");
      }
   }
else
   {
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Drill Mode Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
   }

for (i=0;i<PACKETLENGTH;i++)  // Reset all bits
   dataOut[i]=0;

if (positionX > 0)
   {
   dirX = (machineInvertDirX ^ 0);                    // X left
   dataOut[1] = positionX;                            // StepX LOW Byte
   dataOut[2] = positionX >> 8;                       // StepX HIGH Byte
   }
```

```
else if (positionX < 0)
   {
   dirX = (machineInvertDirX ^ 1);                       // X right
   dataOut[1] = (positionX * -1);                        // StepX LOW Byte
   dataOut[2] = (positionX * -1) >> 8;                   // StepX HIGH Byte
   }

if (positionY > 0)
   {
   dirY = (machineInvertDirY ^ 0);                       // Y down
   dataOut[3] = positionY;                               // StepY LOW Byte
   dataOut[4] = positionY >> 8;                          // StepY HIGH Byte
   }
else if (positionY < 0)
   {
   dirY = (machineInvertDirY ^ 1);                       // Y up
   dataOut[3] = (positionY * -1);                        // StepY LOW Byte
   dataOut[4] = (positionY * -1) >> 8;                   // StepY HIGH Byte
   }

if (positionZ > 0)
   {
   dirZ = (machineInvertDirZ ^ 0);                       // Z up
   dataOut[5] = positionZ;                               // StepZ LOW Byte
   dataOut[6] = positionZ >> 8;                          // StepZ HIGH Byte
   }
else if (positionZ < 0)
   {
   dirZ = (machineInvertDirZ ^ 1);                       // Z down
   dataOut[5] = (positionZ * -1);                        // StepZ LOW Byte
   dataOut[6] = (positionZ * -1) >> 8;                   // StepZ HIGH Byte
   }

dataOut[0] = (machineModeZ<<5) +(machineModeY << 4)+(machineModeX <<
3)+(dirZ << 2)+(dirY << 1)+(dirX);
dataOut[7] = machineDelayX;
dataOut[8] = machineDelayY;
dataOut[9] = machineDelayZup;
ComPort1->Write(dataOut,PACKETLENGTH); // Send dataOut Packet to PIC
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::clearImage()
{
// This function clears the screen of TImage Object
Image1->Canvas->Pen->Color = clWhite;
Image1->Canvas->Brush->Color = clWhite;
Image1->Canvas->Rectangle(0,0,Image1->Width,Image1->Height);
}
//---------------------------------------------------------------------------
int __fastcall TFormMain::loadDrillFile(const char *FileName)
{
// This function Loads Drill Coordinates with Tool Numbers
// and saves thE data in holeFile structure
   FILE *drillFile;  // File Pointer
   char ch;
   char lineBuffer[20];  // LineBuffer to hold one line data
   bool  fileFinished = 0;
   char toolNumber[2];

   lineCounter = 0;
```

```
   drillFile = fopen(FileName, "r");    // Open file again to reset File
Pointer

   while (!fileFinished && !feof(drillFile))
      {
      fgets(lineBuffer, 20, drillFile);
      switch (lineBuffer[0])
         {
         case '%':
         break;
         case 'T':
         if (lineBuffer[2] == 'C')
            {
            toolNumber[0]=lineBuffer[1];
            toolNumber[1]=0;
            for (i=0;i<5;i++)
               tools[toolCounter].D[i] = lineBuffer[i+3];
            }
         else if (lineBuffer[3] == 'C')
            {
         toolNumber[0]=lineBuffer[1];
         toolNumber[1]=lineBuffer[2];
            for (i=0;i<5;i++)
               tools[toolCounter].D[i] = lineBuffer[i+4];
            }
         for (i=0;i<2;i++)
            tools[toolCounter].TI[i] = toolNumber[i];
         toolCounter++;
         break;
         case 'X':
         for (i=0; i<6; i++)
            {
            holes[lineCounter].X[i] = lineBuffer[i+1];
            holes[lineCounter].Y[i] = lineBuffer[i+8];
            }
         for (i=0; i<2; i++)
            holes[lineCounter].T[i] = toolNumber[i];
         lineCounter++;
         break;
         case 'M':
         fileFinished = 1;
         break;
         default:
         break;
         } // End of switch (lineBuffer[0])
      } // End of while (!fileFinished && !feof(drillFile))
    fclose(drillFile);
    return lineCounter;
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FileOpenExecute(TObject *Sender)
{
char *filePath;
int maxX,maxY;
double diameterMm,diameterMil;

toolCounter = 0;
fileOpenedFlag = OpenDialog1->Execute();
if (fileOpenedFlag)
   {
        FileName = OpenDialog1->FileName;
```

```
        StatusBar1->Panels->Items[0]->Text = FileName;
   filePath = FileName.c_str();           // Change AnsiString to char
   lineCounter=loadDrillFile(filePath);   // Put Drill Coordinates into
   maxX = StrToInt(holes[0].X);           // hole[] structure
   maxY = StrToInt(holes[0].Y);
   clearImage();  // Clear Image1
   for (j=1;j<lineCounter;j++)
      {
      if (StrToInt(holes[j].X) > maxX)
         maxX = StrToInt(holes[j].X);     // Find Max X Coordinate
      if (StrToInt(holes[j].Y) > maxY)
         maxY = StrToInt(holes[j].Y);     // Find Max Y Coordinate
      }
   scaleFactorX = 1 / ((float(Image1->Width)-100) / maxX);
   scaleFactorY = 1 / ((float(Image1->Height)-100) / maxY);
   StringGrid1->RowCount = 2;
   for (i=1;i<=lineCounter;i++)
      {
      StringGrid1->Cells[0][i]=i;
      StringGrid1->Cells[1][i]=StrToInt(holes[i-1].T);
      StringGrid1->Cells[2][i]=StrToInt(holes[i-1].X);
      StringGrid1->Cells[3][i]=StrToInt(holes[i-1].Y);
      StringGrid1->RowCount += 1;
      Xpixel = (StrToInt(holes[i-1].X))/scaleFactorX;
      Ypixel = (StrToInt(holes[i-1].Y))/scaleFactorY;
      drawCircle(Xpixel+20,Ypixel+20,holeRadius,clYellow);
      }
   StringGrid1->RowCount = lineCounter+1; // Delete the empty line at the
end of string grid
   FormToolIndex->StringGrid1->RowCount = 2;
   for (i=1;i<=toolCounter;i++)
      {
      FormToolIndex->StringGrid1->Cells[0][i]=tools[i-1].TI;
      diameterMil = (StrToFloat(tools[i-1].D) * 1000);
      FormToolIndex->StringGrid1->Cells[1][i]=diameterMil;
      diameterMm = (StrToFloat(tools[i-1].D) * 25.4);
      FormToolIndex->StringGrid1->Cells[2][i]=diameterMm;
      FormToolIndex->StringGrid1->RowCount += 1;
      }
   GoToolIndex->Enabled = true;
   if (machineOnFlag)
      GoDrillBoard->Enabled = true;
   MemoMachineInfo->Lines->Add(">> Drill File Opened. (" +
IntToStr(lineCounter) + " holes, " + IntToStr(toolCounter) + " tools)");
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::FileExitExecute(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::HelpAboutExecute(TObject *Sender)
{
AboutBox->ShowModal();
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::SetComExecute(TObject *Sender)
{
ComPort1->ShowSetupDialog();
}
```

```
//---------------------------------------------------------------------------
void __fastcall TFormMain::GoJogExecute(TObject *Sender)
{
ComPort1->Open();
outputBuffer[0] = OPENJOGMODE;
ComPort1->Write(outputBuffer,1);
comPortFlag = ComPort1->Read(inputBuffer,1);
if (comPortFlag == 0)
   {
   Application->MessageBox("Machine is not responding. Check connections
and try again","Jog Mode Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
   }
else if (comPortFlag == 1)
   {
   if (inputBuffer[0] == JOGMODEON)
      {
      MemoMachineInfo->Lines->Add(">> Jog Mode On.");
      currentMachineMode = 0; // Machine is in Jog Mode
      FormJog->ShowModal();
      }
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
      MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
      }
   }
else
   {
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
   MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
   }
}
//---------------------------------------------------------------------------


void __fastcall TFormMain::SetMachineExecute(TObject *Sender)
{
FormMachineSettings->ShowModal();
}
//---------------------------------------------------------------------------

void __fastcall TFormMain::GoToolIndexExecute(TObject *Sender)
{
FormToolIndex->Show();  // NonModal & Always on Top
}
//---------------------------------------------------------------------------

void __fastcall TFormMain::GoCheckMachineExecute(TObject *Sender)
{
ComPort1->Open();
outputBuffer[0] = ISMACHINEREADY;
ComPort1->Write(outputBuffer,1);
comPortFlag = ComPort1->Read(inputBuffer,1);    // If no data can be read
and timeout elapses
                                                // Read function returns
0.
if (comPortFlag == 0)
   {
```

```
   Application->MessageBox("Machine is not responding. Check connections
and try again","Machine Error", MB_OK);
   ComPort1->Close();
   }
else if (comPortFlag == 1){
   if (inputBuffer[0] == MACHINEREADY)
      {
      machineOnFlag = 1;
      MemoMachineInfo->Lines->Add(">> Machine is ready. Use Jog or Drill
mode.");
      GoJog->Enabled = true;
//      GoCheckMachine->Enabled = false;
      if (fileOpenedFlag)
         GoDrillBoard->Enabled = true;
      }
   else
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
   }
else
Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
}
//-------------------------------------------------------------------------
void __fastcall TFormMain::ComPort1RxChar(TObject *Sender, int Count)
{
byte dataIn[100];
ComPort1->Read(dataIn, Count);
   switch (dataIn[0])
      {
      case JOGMODEDONE:
      FormJog->enableAxisButtons();
      // Update Axis Positions
      switch (whichAxis)
         {
         case 1:
         positionX += axisStep;
         break;
         case 2:
         positionX -= axisStep;
         break;
         case 3:
         positionY += axisStep;
         break;
         case 4:
         positionY -= axisStep;
         break;
         case 5:
         positionZ -= axisStep;
         break;
         case 6:
         positionZ += axisStep;
         break;
         case 7:  // OffsetX in Jog Mode completed
         positionX = 0;
         gotoOffsetYInJogMode();
         break;
         case 8:  // OffsetY in Jog Mode completed
         positionY = 0;
         gotoOffsetZInJogMode();
         break;
```

```
         case 9:  // OffsetZ in Jog Mode completed
         positionZ = 0;
         //Application->MessageBox("Board Drilled Successfully.","Drill
Complete", MB_OK);
         MemoMachineInfo->Lines->Add(">> Board Drilled Successfully.");
         drillCompleteFlag = 0;  // Reset drillCompleteFlag inorder to
drill a new board
         holeNumber = 0;         // Reset holeNumber inorder to drill a
new board
         exitJogMode();
         GoDrillBoard->Enabled = true;
         GoJog->Enabled = true;
         GoEStop->Enabled = false;
         break;
         }
      updatePositionDisplays();
      FormJog->LblPositionX->Caption = IntToStr(positionX) + " steps";
      FormJog->LblPositionY->Caption = IntToStr(positionY) + " steps";
      FormJog->LblPositionZ->Caption = IntToStr(positionZ) + " steps";
      if (!currentMachineMode)       // Machine is in Jog Mode
         MemoMachineInfo->Lines->Add(">> Jog movement completed.");
      break;
      case HOME_SWITCH_X:
      MemoMachineInfo->Lines->Add(">> Machine is at HOME position on X
axis.");
      if (currentMachineMode)        // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                           // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case LIMIT_SWITCH_X:
      MemoMachineInfo->Lines->Add(">> Machine is at END position on X
axis.");
      if (currentMachineMode)        // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                           // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case HOME_SWITCH_Y:
      MemoMachineInfo->Lines->Add(">> Machine is at HOME position on Y
axis.");
      if (currentMachineMode)        // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                           // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case LIMIT_SWITCH_Y:
      MemoMachineInfo->Lines->Add(">> Machine is at END position on Y
axis.");
      if (currentMachineMode)        // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                           // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case HOME_SWITCH_Z:
      MemoMachineInfo->Lines->Add(">> Machine is at HOME position on Z
axis.");
      if (currentMachineMode)        // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                           // Machine is in Jog Mode
         FormJog->enableAxisButtons();
```

```
      break;
      case LIMIT_SWITCH_Z:
      MemoMachineInfo->Lines->Add(">> Machine is at END position on Z 
axis.");
      if (currentMachineMode)       // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                          // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case ERRORLIMITSWITCH:
      MemoMachineInfo->Lines->Add(">> Machine is at UNDEFINED limit 
switch position.");
      if (currentMachineMode)       // Machine is in Drill Mode
         limitPositionOnDrillMode();
      else                          // Machine is in Jog Mode
         FormJog->enableAxisButtons();
      break;
      case ATOFFSETPOS:
      MemoMachineInfo->Lines->Add(">> Machine is at Offset Position.");
      positionX = 0;    // Reset position variables
      positionY = 0;
      positionZ = 0;
      updatePositionDisplays();
      initializeDrillMode(); // Send drill mode initialization data (9 
bytes)
      break;
      case DRILLMODE1INITIALIZED:
      positionZ += drillStepZup;
      dataOut[0] = 0xC0;   // Continue & Start Drilling Flag Set
      dataOut[1] = 0;
      dataOut[2] = 0;
      dataOut[3] = 0;
      dataOut[4] = 0;
      ComPort1->Write(dataOut,5);
      break;
      case ONEHOLEDRILLED:
      //*********Start of Drill Code**************
      updatePositionDisplays();        // Show current machine 
coordinates
      if (drillCompleteFlag == 0)
         {
         // Start of "Circle drilled holes with red"
         if (holeNumber != 0)
            {
            // mark all minus last hole
            Xpixel = (StrToInt(holes[holeNumber-1].X))/scaleFactorX;
            Ypixel = (StrToInt(holes[holeNumber-1].Y))/scaleFactorY;
            markHole(Xpixel+20,Ypixel+20,MARKHOLERADIUS,clRed);
            }
         // End of "Circle drilled holes with red"
         // Start of "Warn for tool change"
         if (warnForToolChange)
            {
            if (holeNumber != 0)
               {
               if (StrToInt(holes[holeNumber].T) != 
StrToInt(holes[holeNumber-1].T))
                  {
                  Application->MessageBox(("Change tool and press OK.
\nTool Number = " + IntToStr(StrToInt(holes[holeNumber].T))).c_str()
,"Change Tool", MB_OK);
```

```
                }
            }
        }
        // End of "Warn for tool change"
        // Drill hole according to Selected Drill mode
        if (currentMachineMode == 1)       // Drill Mode1
            drillOneHole(holeNumber);        // Drill coordinates goes to
PIC from here
        else if (currentMachineMode == 2)  // Drill Mode2
            drillOneHole2(holeNumber);       // Drill coordinates goes to
PIC from here
        MemoMachineInfo->Lines->Add(">> Drilling hole [" +
IntToStr(holeNumber+1) + " of " + IntToStr(lineCounter) +"]");
        }
      else
        {
        dataOut[0] = 0x00;   // Continue Drilling Flag Reset (Exit Drill
Mode)
        dataOut[1] = 0;
        dataOut[2] = 0;
        dataOut[3] = 0;
        dataOut[4] = 0;
        ComPort1->Write(dataOut,5);
        }
      if (holeNumber < lineCounter-1)
        {
        holeNumber++;
        drillCompleteFlag = 0;
        }
      else
        drillCompleteFlag = 1;
      //*********End of Drill Code*************
      break;
      case DRILLMODE1DONE:
      // mark last hole here
      Xpixel = (StrToInt(holes[holeNumber].X))/scaleFactorX;
      Ypixel = (StrToInt(holes[holeNumber].Y))/scaleFactorY;
      markHole(Xpixel+20,Ypixel+20,6,clRed);
      gotoOffsetXInJogMode(); // Goto offset position on X than Y than Z
      break;
      default:
      MemoMachineInfo->Lines->Add(">> ERROR - Unrecognized data came from
Com Port.(E1001)");
      break;
      }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::GoDrillBoardExecute(TObject *Sender)
{
GoCheckMachine->Enabled = false;
GoDrillBoard->Enabled = false;
GoJog->Enabled = false;
GoEStop->Enabled = true;
clearImage();  // Clear Image1
for (i=0;i<lineCounter;i++)
   {
   Xpixel = (StrToInt(holes[i].X))/scaleFactorX;
   Ypixel = (StrToInt(holes[i].Y))/scaleFactorY;
   drawCircle(Xpixel+20,Ypixel+20,holeRadius,clYellow);
   }
switch (FormMachineSettings->RadioGroupDrillMode->ItemIndex)
```

```
   {
   case 0:  // Drill using offset position
   currentMachineMode = 1; // Machine is in Drill Mode1
   gotoOffsetPosition();
   break;
   case 1:  // Drill using placement matrix
   currentMachineMode = 2; // Machine is in Drill Mode2
   // Start of Drill Mode2
   //Application->MessageBox("Jog to holes[96]","Select first hole",
MB_OK);
   // Modify Jog Form for Drill Mode2
   FormJog->GroupBox8->Top = 290;
   FormJog->GroupBox8->Visible = true;
   outputBuffer[0] = OPENJOGMODE;
   ComPort1->Write(outputBuffer,1);
   comPortFlag = ComPort1->Read(inputBuffer,1);
   if (comPortFlag == 0)
      {
      Application->MessageBox("Machine is not responding. Check
connections and try again","Jog Mode Error", MB_OK);
      MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
      }
   else if (comPortFlag == 1)
      {
      if (inputBuffer[0] == JOGMODEON)
         {
         MemoMachineInfo->Lines->Add(">> Jog to first hole and press
Button1.");
         FormJog->ShowModal();     // PIC is ready to take data in Jog
Mode
         }
      else
         {
         Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
         MemoMachineInfo->Lines->Add(">> Jog Mode can not be
initialized.");
         }
      }
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Machine Error", MB_OK);
      MemoMachineInfo->Lines->Add(">> Jog Mode can not be initialized.");
      }
   gotoOffsetPosition();
   // End of Drill Mode2
   break;
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::GoEStopExecute(TObject *Sender)
{
char estopData = EMERGENCYSTOP;
ComPort1->Write(&estopData,1);
comPortFlag = ComPort1->Read(inputBuffer,1);    // If no data can be read
and timeout elapses
                                                // Read function returns
0.
if (comPortFlag == 0)
```

```
   Application->MessageBox("Machine is not responding. Check connections
and try again","Emergency Stop Error", MB_OK);
else if (comPortFlag == 1){
   if (inputBuffer[0] == MACHINESTOPPED)
      {
      positionX = 0;    // Reset position variables
      positionY = 0;
      positionZ = 0;
      updatePositionDisplays();
      GoJog->Enabled = true;
      GoEStop->Enabled = false;
      GoDrillBoard->Enabled = false;
      GoCheckMachine->Enabled = true;
      MemoMachineInfo->Lines->Add(">> Machine stopped successfully.
Offset Coordinates lost! Current position is set as offset(0,0,0)");
      holeNumber = 0;   // Reset holeNumber to start from first hole on
next drill
      }
   else
   Application->MessageBox("Machine can not be stopped!","Emergency Stop
Error", MB_OK);
   }
else
Application->MessageBox("Machine can not be stopped!","Emergency Stop
Error", MB_OK);
}
//---------------------------------------------------------------------------
void __fastcall TFormMain::StringGrid1SelectCell(TObject *Sender, int
ACol,
      int ARow, bool &CanSelect)
{
// Draw a new yellow circle on previosly selected hole
if (previousARow != 0)
   {
   Xpixel = (StrToInt(StringGrid1->Cells[2][previousARow]))/scaleFactorX;
   Ypixel = (StrToInt(StringGrid1->Cells[3][previousARow]))/scaleFactorY;
   drawCircle(Xpixel+20,Ypixel+20,holeRadius,clYellow);
   }
// Draw a red circle on yellow circle to mark selected hole
Xpixel = (StrToInt(StringGrid1->Cells[2][ARow]))/scaleFactorX;
Ypixel = (StrToInt(StringGrid1->Cells[3][ARow]))/scaleFactorY;
drawCircle(Xpixel+20,Ypixel+20,holeRadius,clRed);

previousARow = ARow;
previousACol = ACol;
}
//---------------------------------------------------------------------------
```

## Aboutform.cpp

```
//---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "aboutform.h"
//---------------------------------------------------------------------------
#pragma resource "*.dfm"
TAboutBox *AboutBox;
//---------------------------------------------------------------------------
__fastcall TAboutBox::TAboutBox(TComponent* AOwner)
        : TForm(AOwner)
```

```
{
}
//---------------------------------------------------------------------
```

**Jogform.cpp**

```
//-----------------------------------------------------------------------

#include <vcl.h>
#include <math.h>
#pragma hdrstop

#include "jogform.h"
#include "mainform.h"
#include "settingsform.h"
//-----------------------------------------------------------------------
----
#pragma package(smart_init)
#pragma link "AdvToolBtn"
#pragma resource "*.dfm"
TFormJog *FormJog;

unsigned char outputBuffer2[10]; // Serial Port Output Data Buffer
unsigned char inputBuffer2[10];  // Serial Port Input Data Buffer
                                 // AxisStep is defined in mainform.cpp
char axisDelay;                  // Define 8 bit Delay Data variable


//-----------------------------------------------------------------------
__fastcall TFormJog::TFormJog(TComponent* Owner)
   : TForm(Owner)
{
}
//-----------------------------------------------------------------------
void __fastcall TFormJog::disableAxisButtons()
{
ToolBtnXleft->Enabled = false;
ToolBtnXright->Enabled = false;
ToolBtnYup->Enabled = false;
ToolBtnYdown->Enabled = false;
ToolBtnZup->Enabled = false;
ToolBtnZdown->Enabled = false;
ToolBtnEstop->Enabled = true;
}
//-----------------------------------------------------------------------
void __fastcall TFormJog::enableAxisButtons()
{
ToolBtnXleft->Enabled = true;
ToolBtnXright->Enabled = true;
ToolBtnYup->Enabled = true;
ToolBtnYdown->Enabled = true;
ToolBtnZup->Enabled = true;
ToolBtnZdown->Enabled = true;
ToolBtnEstop->Enabled = false;
}
//-----------------------------------------------------------------------
void __fastcall TFormJog::ToolBtnSpindleClick(TObject *Sender)
{
if (ToolBtnSpindle->Down==true)
    ToolBtnSpindle->Caption="SPINDLE ON";
```

```
else
    ToolBtnSpindle->Caption="SPINDLE OFF";
}
//---------------------------------------------------------------------------
void __fastcall TFormJog::BtnSetOffsetPositionClick(TObject *Sender)
{
// Offset Code
positionX = 0;
positionY = 0;
positionZ = 0;
LblPositionX->Caption = positionX;
LblPositionY->Caption = positionY;
LblPositionZ->Caption = positionZ;
FormMain->updatePositionDisplays();
if (fileOpenedFlag)
   FormMain->GoDrillBoard->Enabled = true;
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnXrightClick(TObject *Sender)
{
switch (ComboUnitX->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeX +1) * StrToInt(EditMovementX->Text) * 100 /
stepsPerMicronX;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeX +1) * StrToInt(EditMovementX->Text) * 1000 /
stepsPerMicronX;
   break;
   case 2:  // 2.54 mm
   axisStep = (machineModeX +1) * StrToInt(EditMovementX->Text) * 2540 /
stepsPerMicronX;
   break;
   case 3:  // Full Steps
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text);
   break;
   default: // 100um as default value
   axisStep = (machineModeX +1) * StrToInt(EditMovementX->Text) * 100 /
stepsPerMicronX;
   break;
   }
Edit1->Text=axisStep;

jogData[0] = 0x40 + (machineModeX << 5) + ((machineInvertDirX ^ 1) << 4)
+ (ToolBtnSpindle->Down);
jogData[1] = axisStep;                    // axisStep LOW Byte
jogData[2] = axisStep >> 8;               // axisStep HIGH Byte
jogData[3] = TrackBarDelayX->Position;    // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 1; // X Axis Right Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
disableAxisButtons();

}
//---------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnYupClick(TObject *Sender)
```

```
{
switch (ComboUnitY->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 100 /
stepsPerMicronY;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 1000 /
stepsPerMicronY;
   break;
   case 2:  // 2.54 mm
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 2540 /
stepsPerMicronY;
   break;
   default: // 100um as default value
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 100 /
stepsPerMicronY;
   break;
   }
Edit1->Text=axisStep;
jogData[0] = 0x80 + (machineModeY << 5) + ((machineInvertDirY ^ 1) << 4)
+ (ToolBtnSpindle->Down);
jogData[1] = axisStep;                    // axisStep LOW Byte
jogData[2] = axisStep >> 8;               // axisStep HIGH Byte
jogData[3] = TrackBarDelayY->Position;    // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 3; // Y Axis Up Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
disableAxisButtons();

}
//---------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnYdownClick(TObject *Sender)
{
switch (ComboUnitY->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 100 /
stepsPerMicronY;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 1000 /
stepsPerMicronY;
   break;
   case 2:  // 2.54 mm
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 2540 /
stepsPerMicronY;
   break;
   default: // 100um as default value
   axisStep = (machineModeY + 1) * StrToInt(EditMovementY->Text) * 100 /
stepsPerMicronY;
   break;
   }
Edit1->Text=axisStep;

jogData[0] = 0x80 + (machineModeY << 5) + ((machineInvertDirY ^ 0) << 4)
+ (ToolBtnSpindle->Down);
```

```
jogData[1] = axisStep;                  // axisStep LOW Byte
jogData[2] = axisStep >> 8;             // axisStep HIGH Byte
jogData[3] = TrackBarDelayY->Position;    // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 4; // Y Axis Down Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
disableAxisButtons();
}
//--------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnZupClick(TObject *Sender)
{
switch (ComboUnitZ->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 100 /
stepsPerMicronZ;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 1000 /
stepsPerMicronZ;
   break;
   case 2:  // 2.54 mm
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 2540 /
stepsPerMicronZ;
   break;
   default: // 100um as default value
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 100 /
stepsPerMicronZ;
   break;
   }
Edit1->Text=axisStep;

jogData[0] = 0xC0 + (machineModeZ << 5) + ((machineInvertDirZ ^ 0) << 4)
+ (ToolBtnSpindle->Down);
jogData[1] = axisStep;                  // axisStep LOW Byte
jogData[2] = axisStep >> 8;             // axisStep HIGH Byte
jogData[3] = TrackBarDelayZ->Position;    // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 5; // Z Axis Up Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
disableAxisButtons();
}
//--------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnZdownClick(TObject *Sender)
{
switch (ComboUnitZ->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 100 /
stepsPerMicronZ;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 1000 /
stepsPerMicronZ;
   break;
```

```
   case 2:  // 2.54 mm
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 2540 /
stepsPerMicronZ;
   break;
   default: // 100um as default value
   axisStep = (machineModeZ + 1) * StrToInt(EditMovementZ->Text) * 100 /
stepsPerMicronZ;
   break;
   }
Edit1->Text=axisStep;

jogData[0] = 0xC0 + (machineModeZ << 5) + ((machineInvertDirZ ^ 1) << 4)
+ (ToolBtnSpindle->Down);
jogData[1] = axisStep;                    // axisStep LOW Byte
jogData[2] = axisStep >> 8;               // axisStep HIGH Byte
jogData[3] = TrackBarDelayZ->Position;    // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 6; // Z Axis Down Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
disableAxisButtons();
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::Button2Click(TObject *Sender)
{
enableAxisButtons();
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::TrackBarDelayXChange(TObject *Sender)
{
EditDelayX->Text=TrackBarDelayX->Position;
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::TrackBarDelayYChange(TObject *Sender)
{
EditDelayY->Text=TrackBarDelayY->Position;
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::TrackBarDelayZChange(TObject *Sender)
{
EditDelayZ->Text=TrackBarDelayZ->Position;
}
//---------------------------------------------------------------------------


void __fastcall TFormJog::ToolBtnEstopClick(TObject *Sender)
{
byte estopData = EMERGENCYSTOP;
FormMain->ComPort1->Write(&estopData,1);
comPortFlag = FormMain->ComPort1->Read(inputBuffer2,1);   // If no data
can be read and timeout elapses
                                                          // Read function
returns 0.
if (comPortFlag == 0)
   Application->MessageBox("Machine is not responding. Check connections
and try again","Emergency Stop Error", MB_OK);
```

```
else if (comPortFlag == 1){
   if (inputBuffer2[0] == MACHINESTOPPED)
      {
      positionX = 0;    // Reset position variables
      positionY = 0;
      positionZ = 0;
      FormMain->updatePositionDisplays();    // Main Form Displays
      LblPositionX->Caption = positionX;     // Jog Form Displays
      LblPositionY->Caption = positionY;
      LblPositionZ->Caption = positionZ;
      enableAxisButtons();
      FormMain->MemoMachineInfo->Lines->Add(">> Machine stopped
successfully. Offset Coordinates lost! Current position is set as
offset(0,0,0)");
      }
   else
   Application->MessageBox("Machine can not be stopped!","Emergency Stop
Error", MB_OK);
   }
else
Application->MessageBox("Machine can not be stopped!","Emergency Stop
Error", MB_OK);
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::ToolBtnXleftClick(TObject *Sender)
{
switch (ComboUnitX->ItemIndex)
   {
   case 0:  // 100 um
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text) * 100 /
stepsPerMicronX;
   break;
   case 1:  // 1 mm
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text) * 1000 /
stepsPerMicronX;
   break;
   case 2:  // 2.54 mm
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text) * 2540 /
stepsPerMicronX;
   break;
   case 3:  // Full Steps
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text);
   break;
   default: // 100um as default value
   axisStep = (machineModeX + 1) * StrToInt(EditMovementX->Text) * 100 /
stepsPerMicronX;
   break;
   }
Edit1->Text=axisStep;

jogData[0] = 0x40 + (machineModeX << 5) + ((machineInvertDirX ^ 0) << 4)
+ (ToolBtnSpindle->Down);
jogData[1] = axisStep;                     // axisStep LOW Byte
jogData[2] = axisStep >> 8;                // axisStep HIGH Byte
jogData[3] = TrackBarDelayX->Position;     // axisDelay Byte

FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Send Jog Data
Packet to PIC
whichAxis = 2; // X Axis Left Move
FormMain->MemoMachineInfo->Lines->Add(">> Jog movement started.");
```

```
disableAxisButtons();
}
//---------------------------------------------------------------------------



void __fastcall TFormJog::BtnExitJogModeClick(TObject *Sender)
{
Close();
}
//---------------------------------------------------------------------------

void __fastcall TFormJog::FormClose(TObject *Sender, TCloseAction
&Action)
{
jogData[0] = 0; // Send 00xxxxxxb to exit from jog mode
jogData[1] = 0;
jogData[2] = 0;
jogData[3] = 0;
FormMain->ComPort1->Write(jogData,JOGPACKETLENGTH); // Exit from Jog mode
comPortFlag = FormMain->ComPort1->Read(inputBuffer2,1);

if (comPortFlag == 0)
   {
   Application->MessageBox("Machine is not responding. Check connections
and reset PIC","Jog Mode Close Error", MB_OK);
   FormMain->GoJog->Enabled = false;
   }
else if (comPortFlag == 1){
   if (inputBuffer2[0] == JOGMODEOFF)
      FormMain->MemoMachineInfo->Lines->Add(">> Jog Mode Off.");
   else
      {
      Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Jog Mode Close Error", MB_OK);
      FormMain->GoJog->Enabled = false;
      }
   }
else
   {
   Application->MessageBox("Machine is not ready. Reset PIC and try
again.","Jog Mode Close Error", MB_OK);
   FormMain->GoJog->Enabled = false;
   }
}
//---------------------------------------------------------------------------


void __fastcall TFormJog::FormActivate(TObject *Sender)
{
TrackBarDelayX->Position = machineDelayX; // Initial Jog Delays are
Machine Delays
TrackBarDelayY->Position = machineDelayY;
TrackBarDelayZ->Position = machineDelayZup;

ComboUnitX->ItemIndex = 1; // Default Jog unit is mm
ComboUnitY->ItemIndex = 1;
ComboUnitZ->ItemIndex = 1;

EditDelayX->Text=TrackBarDelayX->Position;
EditDelayY->Text=TrackBarDelayY->Position;
```

```
EditDelayZ->Text=TrackBarDelayZ->Position;

LblPositionX->Caption = positionX;
LblPositionY->Caption = positionY;
LblPositionZ->Caption = positionZ;

}
//-------------------------------------------------------------------------
void __fastcall TFormJog::BtnSetZDrillUpPositionClick(TObject *Sender)
{
   if (positionZ > 0)
      {
      drillStepZup = positionZ;
      FormMachineSettings->EditDrillStepZup->Text = drillStepZup;
      }
   else
      Application->MessageBox("Drill Up position can not be higher than
offset position.","Set Drill Up Position Error", MB_OK);
}
//-------------------------------------------------------------------------
void __fastcall TFormJog::BtnSetZDrillDownPositionClick(TObject *Sender)
{
   if (positionZ > drillStepZup)
      {
      drillStepZdown = positionZ;
      FormMachineSettings->EditDrillStepZdown->Text = drillStepZdown;
      }
   else
      Application->MessageBox("Drill Down position can not be higher than
Offset or Drill Up position.","Set Drill Down Position Error", MB_OK);
}
//-------------------------------------------------------------------------



void __fastcall TFormJog::Button1Click(TObject *Sender)
{
positionX = 0;
positionY = 0;
LblPositionX->Caption = positionX;
LblPositionY->Caption = positionY;
FormMain->updatePositionDisplays();
}
//-------------------------------------------------------------------------

void __fastcall TFormJog::Button3Click(TObject *Sender)
{
drillX = positionX;
drillY = positionY;

differenceStepX = (45000 - 2000) * 0.290285 * 2;
differenceStepY = (19000 - 2000) * 0.290285 * 2;

angleTeta = atan(double(drillY) / double(drillX)) - atan(differenceStepY
/ differenceStepX);
//angleTeta = atan(double(drillY) / double(drillX));
//angleAlpha = atan(differenceStepY / differenceStepX);

Close();

}
```

```
//---------------------------------------------------------------------------
```

## Settingsform.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "settingsform.h"
#include "mainform.h"
//-----------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormMachineSettings *FormMachineSettings;
//---------------------------------------------------------------------------
__fastcall TFormMachineSettings::TFormMachineSettings(TComponent* Owner)
   : TForm(Owner)
{
// Load all data from INI file
TIniFile *ini;
   ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   MaskEditOneStepX->Text = ini->ReadString( "Steps per Unit",
"OneStepX", "008.7500" );
   MaskEditOneStepY->Text = ini->ReadString( "Steps per Unit",
"OneStepY", "008.7500" );
   MaskEditOneStepZ->Text = ini->ReadString( "Steps per Unit",
"OneStepZ", "008.7500" );
   ComboOneStepUnitX->ItemIndex = ini->ReadInteger( "Steps per Unit",
"OneStepUnitX", 0 );
   ComboOneStepUnitY->ItemIndex = ini->ReadInteger( "Steps per Unit",
"OneStepUnitY", 0 );
   ComboOneStepUnitZ->ItemIndex = ini->ReadInteger( "Steps per Unit",
"OneStepUnitZ", 0 );
   TrackBarDelayX->Position = ini->ReadInteger( "Axis Speeds",
"TrackBarDelayX",10 );
   TrackBarDelayY->Position = ini->ReadInteger( "Axis Speeds",
"TrackBarDelayY",10 );
   TrackBarDelayZdown->Position = ini->ReadInteger( "Axis Speeds",
"TrackBarDelayZdown",10 );
   TrackBarDelayZup->Position = ini->ReadInteger( "Axis Speeds",
"TrackBarDelayZup",10 );
   RadioXMode->ItemIndex = ini->ReadInteger( "Driving Modes",
"RadioXMode",1);
   RadioYMode->ItemIndex = ini->ReadInteger( "Driving Modes",
"RadioYMode",1);
   RadioZMode->ItemIndex = ini->ReadInteger( "Driving Modes",
"RadioZMode",1);
   EditDrillStepZup->Text = ini->ReadString( "Head Movement",
"EditDrillStepZup",4000);
   EditDrillStepZdown->Text = ini->ReadString( "Head Movement",
"EditDrillStepZdown",5000);
   CheckXInvert->Checked = ini->ReadBool( "Invert
Direction","CheckXInvert",0);
   CheckYInvert->Checked = ini->ReadBool( "Invert
Direction","CheckYInvert",0);
   CheckZInvert->Checked = ini->ReadBool( "Invert
Direction","CheckZInvert",0);
   CheckWarnForToolChange->Checked = ini->ReadBool( "Tool
Changing","CheckWarnForToolChange",1);
delete ini;
```

```
   setStepsPerUnit();      // set stepsPerMicron(X,Y,Z) global variables
on load
   setPointOneMilSteps();  // set pointOneMilSteps(X,Y,Z) global
variables on load
   setDelays();            // set machineDelay(X,Y,Zdown,Zup) global
variables on load
   setDrivingModes();      // set machineMode(X,Y,Z) global variables on
load
   setInvertDirs();        // set machineInvertDir(X,Y,Z) global
variables on load
   warnForToolChange = CheckWarnForToolChange->Checked;  // initialize
warnForToolChange Flag
   drillStepZup = StrToInt(EditDrillStepZup->Text);
   drillStepZdown = StrToInt(EditDrillStepZdown->Text);
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::setDrivingModes()
{
machineModeX = RadioXMode->ItemIndex;
machineModeY = RadioYMode->ItemIndex;
machineModeZ = RadioZMode->ItemIndex;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::setInvertDirs()
{
machineInvertDirX = CheckXInvert->Checked;
machineInvertDirY = CheckYInvert->Checked;
machineInvertDirZ = CheckZInvert->Checked;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::setDelays()
{
machineDelayX = TrackBarDelayX->Position;
machineDelayY = TrackBarDelayY->Position;
machineDelayZdown = TrackBarDelayZdown->Position;
machineDelayZup = TrackBarDelayZup->Position;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::setStepsPerUnit()
{
switch (ComboOneStepUnitX->ItemIndex)
   {
   case 0:  // um
   stepsPerMicronX = StrToFloat(MaskEditOneStepX->Text);
   break;
   case 1:  // mm
   stepsPerMicronX = StrToFloat(MaskEditOneStepX->Text)*1000;
   break;
   default: // um is default
   stepsPerMicronX = StrToFloat(MaskEditOneStepX->Text);
   break;
   }

switch (ComboOneStepUnitY->ItemIndex)
   {
   case 0:  // um
   stepsPerMicronY = StrToFloat(MaskEditOneStepY->Text);
   break;
   case 1:  // mm
   stepsPerMicronY = StrToFloat(MaskEditOneStepY->Text)*1000;
   break;
```

```
   default: // um is default
   stepsPerMicronY = StrToFloat(MaskEditOneStepY->Text);
   break;
   }

switch (ComboOneStepUnitZ->ItemIndex)
   {
   case 0:  // um
   stepsPerMicronZ = StrToFloat(MaskEditOneStepZ->Text);
   break;
   case 1:  // mm
   stepsPerMicronZ = StrToFloat(MaskEditOneStepZ->Text)*1000;
   break;
   default: // um is default
   stepsPerMicronZ = StrToFloat(MaskEditOneStepZ->Text);
   break;
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::setPointOneMilSteps()
{
pointOneMilStepsX = 2.54 / stepsPerMicronX;
pointOneMilStepsY = 2.54 / stepsPerMicronY;
pointOneMilStepsZ = 2.54 / stepsPerMicronZ;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::FormClose(TObject *Sender,
      TCloseAction &Action)
{
// Save all data to INI file before closing
TIniFile *ini;
   ini = new TIniFile(ChangeFileExt( Application->ExeName, ".INI" ) );
   ini->WriteString("Steps per Unit","OneStepX",MaskEditOneStepX->Text);
   ini->WriteString("Steps per Unit","OneStepY",MaskEditOneStepY->Text);
   ini->WriteString("Steps per Unit","OneStepZ",MaskEditOneStepZ->Text);
   ini->WriteInteger("Steps per Unit","OneStepUnitX",ComboOneStepUnitX-
>ItemIndex);
   ini->WriteInteger("Steps per Unit","OneStepUnitY",ComboOneStepUnitY-
>ItemIndex);
   ini->WriteInteger("Steps per Unit","OneStepUnitZ",ComboOneStepUnitZ-
>ItemIndex);
   ini->WriteInteger("Axis Speeds","TrackBarDelayX",TrackBarDelayX-
>Position);
   ini->WriteInteger("Axis Speeds","TrackBarDelayY",TrackBarDelayY-
>Position);
   ini->WriteInteger("Axis
Speeds","TrackBarDelayZdown",TrackBarDelayZdown->Position);
   ini->WriteInteger("Axis Speeds","TrackBarDelayZup",TrackBarDelayZup-
>Position);
   ini->WriteInteger("Driving Modes","RadioXMode",RadioXMode->ItemIndex);
   ini->WriteInteger("Driving Modes","RadioYMode",RadioYMode->ItemIndex);
   ini->WriteInteger("Driving Modes","RadioZMode",RadioZMode->ItemIndex);
   ini->WriteString("Head Movement","EditDrillStepZup",EditDrillStepZup-
>Text);
   ini->WriteString("Head
Movement","EditDrillStepZdown",EditDrillStepZdown->Text);
   ini->WriteBool("Invert Direction","CheckXInvert",CheckXInvert-
>Checked);
   ini->WriteBool("Invert Direction","CheckYInvert",CheckYInvert-
>Checked);
```

```
   ini->WriteBool("Invert Direction","CheckZInvert",CheckZInvert-
>Checked);
   ini->WriteBool("Tool
Changing","CheckWarnForToolChange",CheckWarnForToolChange->Checked);
delete ini;

setStepsPerUnit();      // update StepsPerUnit(X,Y,Z) global variables on
close
setPointOneMilSteps();  // update pointOneMilSteps(X,Y,Z) global
variables on close
setDrivingModes();      // update machineMode(X,Y,Z) global variables on
close
setInvertDirs();        // update machineInvertDir(X,Y,Z) global
variables on close
warnForToolChange = CheckWarnForToolChange->Checked;  // update
warnForToolChange global Flag
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::TrackBarDelayXChange(TObject
*Sender)
{
EditDelayX->Text=TrackBarDelayX->Position;
machineDelayX = TrackBarDelayX->Position;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::TrackBarDelayYChange(TObject
*Sender)
{
EditDelayY->Text=TrackBarDelayY->Position;
machineDelayY = TrackBarDelayY->Position;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::TrackBarDelayZdownChange(
      TObject *Sender)
{
EditDelayZdown->Text=TrackBarDelayZdown->Position;
machineDelayZdown = TrackBarDelayZdown->Position;
}
//---------------------------------------------------------------------------
void __fastcall TFormMachineSettings::TrackBarDelayZupChange(
      TObject *Sender)
{
EditDelayZup->Text=TrackBarDelayZup->Position;
machineDelayZup = TrackBarDelayZup->Position;
}
//---------------------------------------------------------------------------
```

## Toolform.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "toolform.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormToolIndex *FormToolIndex;
//---------------------------------------------------------------------------
__fastcall TFormToolIndex::TFormToolIndex(TComponent* Owner)
```

```
   : TForm(Owner)
{
StringGrid1->Cells[0][0]="Tool";
StringGrid1->Cells[1][0]="Diameter (mil)";
StringGrid1->Cells[2][0]="Diameter (mm)";
}
//---------------------------------------------------------------------------
void __fastcall TFormToolIndex::FormKeyDown(TObject *Sender, WORD &Key,
      TShiftState Shift)
{
if (Key==VK_F8)
   Close();
}
//---------------------------------------------------------------------------
```

# APPENDIX 3 – Software Screenshots

## Main Screen

Maximus Driller V1.0

File Settings Go Help

Open | Check Machine | Jog | Drill Board | Emergency Stop | Machine Settings | Com Port | Tool Index

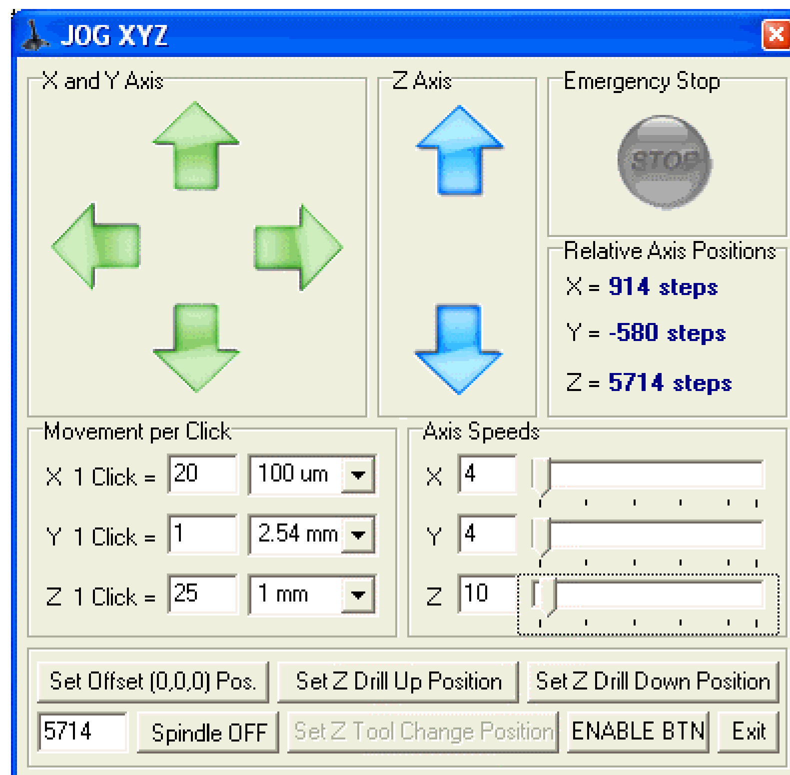
| Hole | Tool | X(.1 mil) | Y(.1 mil) |
|---|---|---|---|
| 40 | 2 | 4000 | 22000 |
| 41 | 2 | 4000 | 23000 |
| 42 | 2 | 4000 | 24000 |
| 43 | 2 | 4000 | 25000 |
| 44 | 2 | 4000 | 26000 |
| 45 | 2 | 8000 | 32000 |
| 46 | 2 | 8000 | 34000 |
| 47 | 2 | 9000 | 38000 |
| 48 | 2 | 10000 | 38000 |
| 49 | 2 | 11000 | 32000 |
| 50 | 2 | 11000 | 33000 |
| 51 | 2 | 11000 | 34000 |
| 52 | 2 | 11000 | 40000 |
| 53 | 2 | 12000 | 38000 |
| 54 | 2 | 13000 | 38000 |
| 55 | 2 | 14000 | 32000 |
| 56 | 2 | 14000 | 34000 |
| 57 | 2 | 19000 | 3000 |
| 58 | 2 | 19000 | 4000 |
| 59 | 2 | 19000 | 5000 |
| 60 | 2 | 19000 | 10000 |
| 61 | 2 | 19000 | 22000 |
| 62 | 2 | 19000 | 34000 |
| 63 | 2 | 19000 | 38000 |
| 64 | 2 | 20000 | 25000 |
| 65 | 2 | 20000 | 26000 |
| 66 | 2 | 20000 | 29000 |

CurrentX = 11030 steps
CurrentY = 5805 steps
CurrentZ = 4343 steps

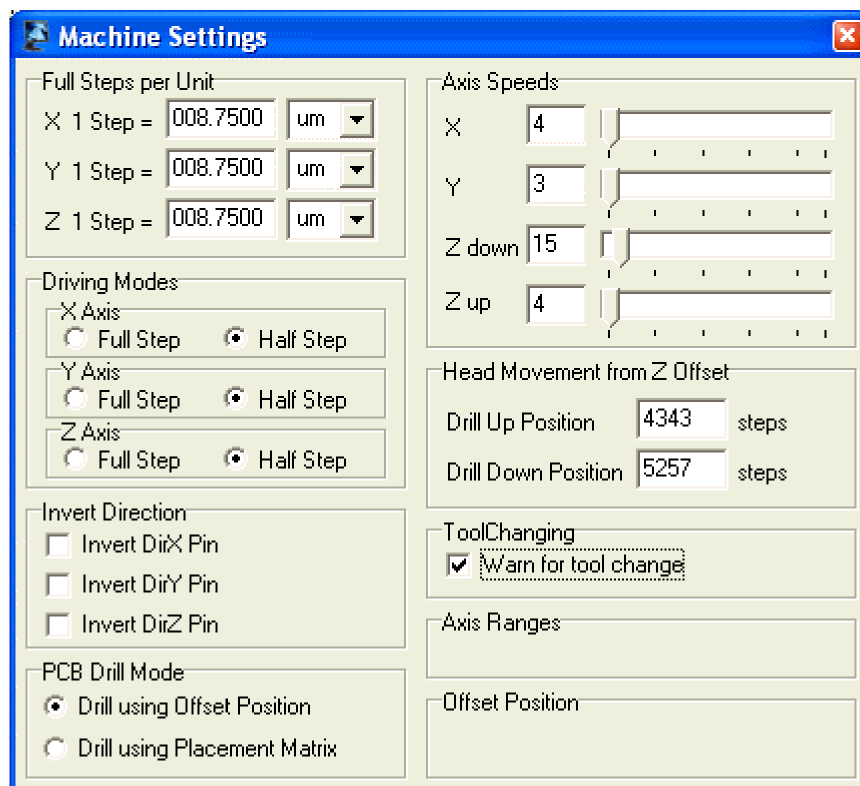
>> Drilling hole [59 of 362]
>> Drilling hole [60 of 362]
>> Drilling hole [61 of 362]

C:\Documents and Settings\Alper YILDIRIM\Desktop\C++ Builder Dosyalarım\örneklerim\Maximus Driller V0.1\drill files\LPTtester.t

## Jog Screen




## Machine Settings Screen

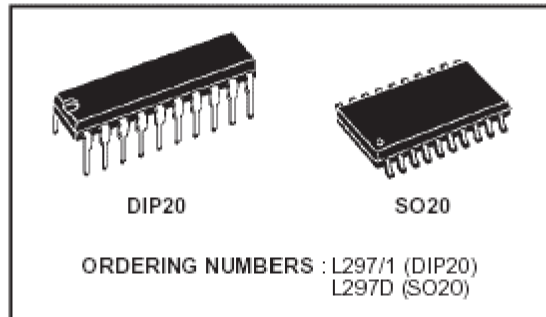


# APPENDIX 4 – L297 STEPPER MOTOR CONTROLLER IC

## L297

## STEPPER MOTOR CONTROLLERS

- NORMAL/WAVE DRIVE
- HALF/FULL STEP MODES
- CLOCKWISE/ANTICLOCKWISE DIRECTION
- SWITCHMODE LOAD CURRENT REGULATION
- PROGRAMMABLE LOAD CURRENT
- FEW EXTERNAL COMPONENTS
- RESET INPUT & HOME OUTPUT
- ENABLE INPUT



DIP20 SO20

**ORDERING NUMBERS :** L297/1 (DIP20)
L297D (SO20)

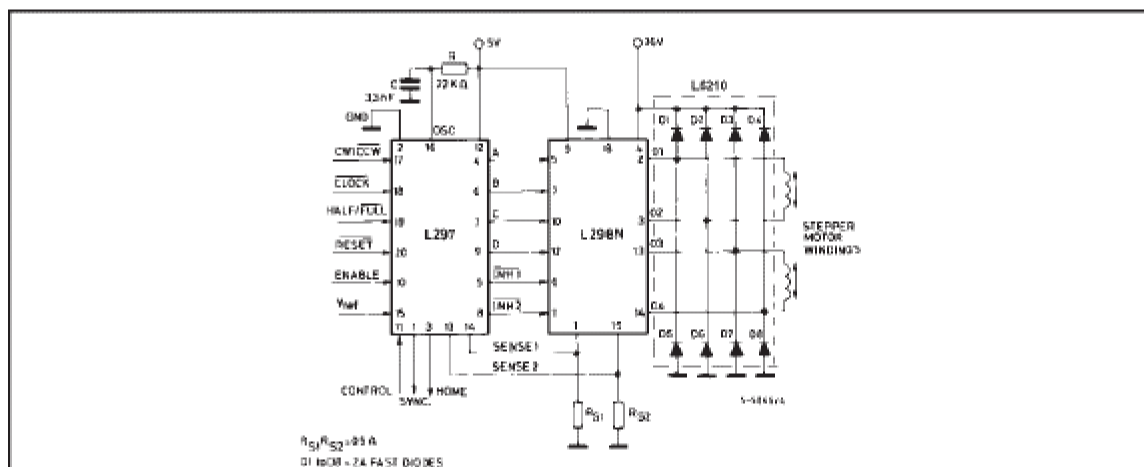
### DESCRIPTION

The L297 Stepper Motor Controller IC generates four phase drive signals for two phase bipolar and four phase unipolar step motors in microcomputer-controlled applications. The motor can be driven in half step, normal and wawe drive modes and on-chip PWM chopper circuits permit switch-mode control of the current in the windings. A feature of this device is that it requires only clock, direction and mode input signals. Since the phase are generated internally the burden on the microprocessor, and the programmer, is greatly reduced. Mounted in DIP20 and SO20 packages, the L297 can be used with monolithic bridge drives such as the L298N or L293E, or with discrete transistors and darlingtons.

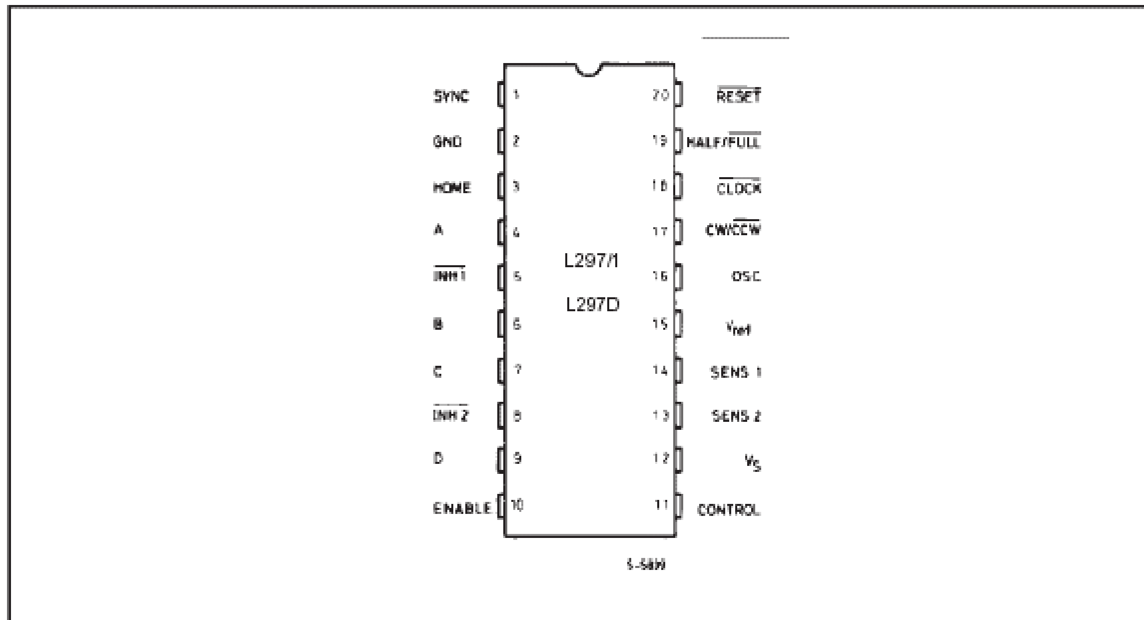
### ABSOLUTE MAXIMUM RATINGS

| Symbol | Parameter | Value | Unit |
|---|---|---|---|
| $V_s$ | Supply voltage | 10 | V |
| $V_i$ | Input signals | 7 | V |
| $P_{tot}$ | Total power dissipation ($T_{amb}$ = 70°C) | 1 | W |
| $T_{stg}$, $T_j$ | Storage and junction temperature | -40 to + 150 | °C |

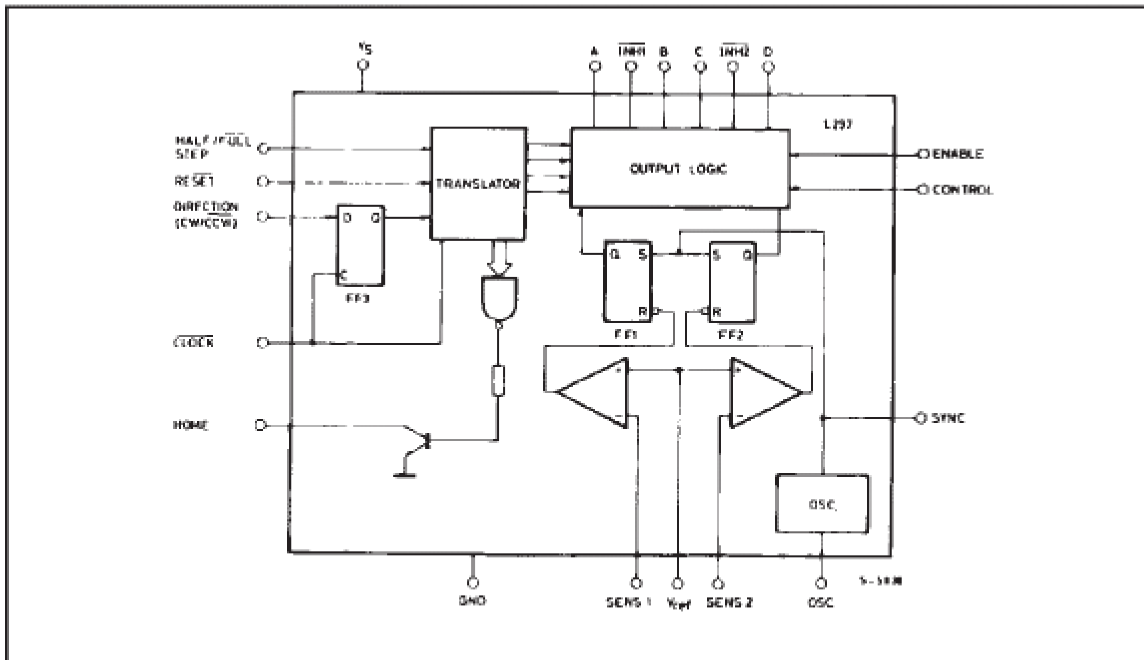
### TWO PHASE BIPOLAR STEPPER MOTOR CONTROL CIRCUIT



## PIN CONNECTION (Top view)




## BLOCK DIAGRAM (L297/1 - L297D)



**PIN FUNCTIONS - L297/1 - L297D**

| N° | NAME | FUNCTION |
|---|---|---|
| 1 | SYNC | Output of the on-chip chopper oscillator. The SYNC connections The SYNC connections of all L297s to be synchronized are connected together and the oscillator components are omitted on all but one. If an external clock source is used it is injected at this terminal. |
| 2 | GND | Ground connection. |
| 3 | HOME | Open collector output that indicates when the L297 is in its initial state (ABCD = 0101). The transistor is open when this signal is active. |
| 4 | A | Motor phase A drive signal for power stage. |
| 5 | $\overline{INH1}$ | Active low inhibit control for driver stage of A and B phases. When a bipolar bridge is used this signal can be used to ensure fast decay of load current when a winding is de-energized. Also used by chopper to regulate load current if CONTROL input is low. |
| 6 | B | Motor phase B drive signal for power stage. |
| 7 | C | Motor phase C drive signal for power stage. |
| 8 | $\overline{INH2}$ | Active low inhibit control for drive stages of C and D phases. Same functions as INH1. |
| 9 | D | Motor phase D drive signal for power stage. |
| 10 | ENABLE | Chip enable input. When low (inactive) INH1, INH2, A, B, C and D are brought low. |
| 11 | CONTROL | Control input that defines action of chopper. When low chopper acts on INH1 and INH2; when high chopper acts on phase lines ABCD. |
| 12 | $V_s$ | 5V supply input. |
| 13 | $SENS_2$ | Input for load current sense voltage from power stages of phases C and D. |
| 14 | $SENS_1$ | Input for load current sense voltage from power stages of phases A and B. |
| 15 | $V_{ref}$ | Reference voltage for chopper circuit. A voltage applied to this pin determines the peak load current. |
| 16 | OSC | An RC network (R to $V_{CC}$, C to ground) connected to this terminal determines the chopper rate. This terminal is connected to ground on all but one device in synchronized multi - L297 configurations. f $\cong$ 1/0.69 RC |
| 17 | CW/$\overline{CCW}$ | Clockwise/counterclockwise direction control input. Physical direction of motor rotation also depends on connection of windings. Synchronized internally therefore direction can be changed at any time. |
| 18 | $\overline{CLOCK}$ | Step clock. An active low pulse on this input advances the motor one increment. The step occurs on the rising edge of this signal. |

**PIN FUNCTIONS - L297/1 - L297D** (continued)

| N° | NAME | FUNCTION |
|---|---|---|
| 19 | HALF/$\overline{FULL}$ | Half/full step select input. When high selects half step operation, when low selects full step operation. One-phase-on full step mode is obtained by selecting FULL when the L297's translator is at an even-numbered state.<br>Two-phase-on full step mode is set by selecting FULL when the translator is at an odd numbered position. (The home position is designate state 1). |
| 20 | $\overline{RESET}$ | Reset input. An active low pulse on this input restores the translator to the home position (state 1, ABCD = 0101). |

**THERMAL DATA**

| Symbol | Parameter | | DIP20 | SO20 | Unit |
|---|---|---|---|---|---|
| $R_{th-j-amb}$ | Thermal resistance junction-ambient | max | 80 | 100 | °C/W |

**CIRCUIT OPERATION**

The L297 is intended for use with a dual bridge driver, quad darlington array or discrete power devices in step motor driving applications. It receives step clock, direction and mode signals from the systems controller (usually a microcomputer chip) and generates control signals for the power stage.

The principal functions are a translator, which generates the motor phase sequences, and a dual PWM chopper circuit which regulates the current in the motor windings. The translator generates three different sequences, selected by the HALF/FULL input. These are normal (two phases energised), wave drive (one phase energised) and half-step (alternately one phase energised/two phases energised). Two inhibit signals are also generated by the L297 in half step and wave drive modes. These signals, which connect directly to the L298's enable inputs, are intended to speed current decay when a winding is de-energised. When the L297 is used to drive a unipolar motor the chopper acts on these lines.

An input called CONTROL determines whether the chopper will act on the phase lines ABCD or the inhibit lines INH1 and INH2. When the phase lines are chopped the non-active phase line of each pair (AB or CD) is activated (rather than interrupting the line then active). In L297 + L298 configurations this technique reduces dissipation in the load current sense resistors.

A common on-chip oscillator drives the dual chopper. It supplies pulses at the chopper rate which set the two flip-flops FF1 and FF2. When the current in a winding reaches the programmed peak value the voltage across the sense resistor (connected to one of the sense inputs $SENS_1$ or $SENS_2$) equals $V_{ref}$ and the corresponding comparator resets its flip flop, interrupting the drive current until the next oscillator pulse arrives. The peak current for both windings is programmed by a voltage divider on the $V_{ref}$ input.

Ground noise problems in multiple configurations can be avoided by synchronising the chopper oscillators. This is done by connecting all the SYNC pins together, mounting the oscillator RC network on one device only and grounding the OSC pin on all other devices.

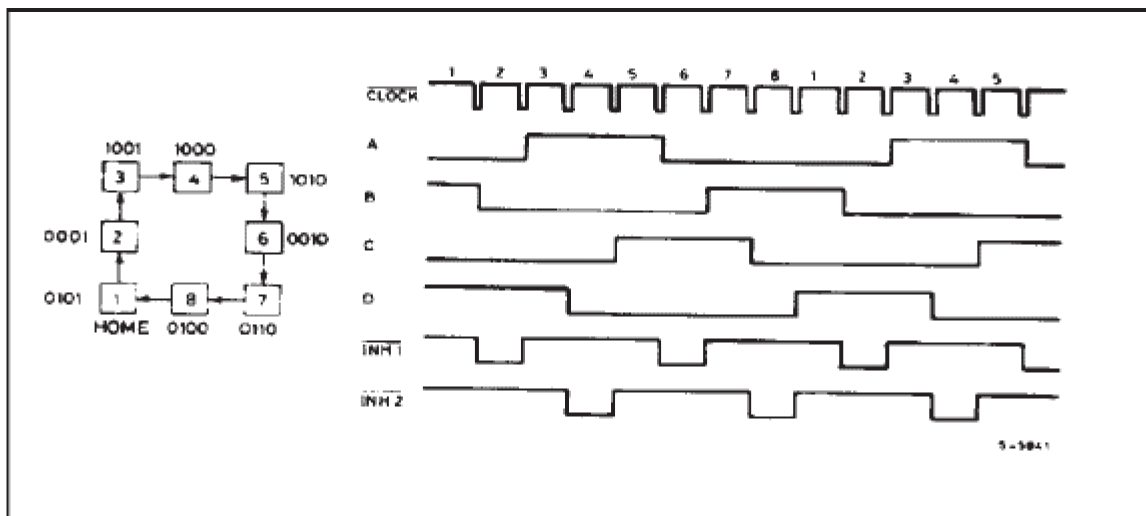
## MOTOR DRIVING PHASE SEQUENCES

The L297's translator generates phase sequences for normal drive, wave drive and half step modes. The state sequences and output waveforms for these three modes are shown below. In all cases the translator advances on the low to high transition of $\overline{\text{CLOCK}}$.

Clockwise rotation is indicate; for anticlockwise rotation the sequences are simply reversed $\overline{\text{RESET}}$ restores the translator to state 1, where ABCD = 0101.

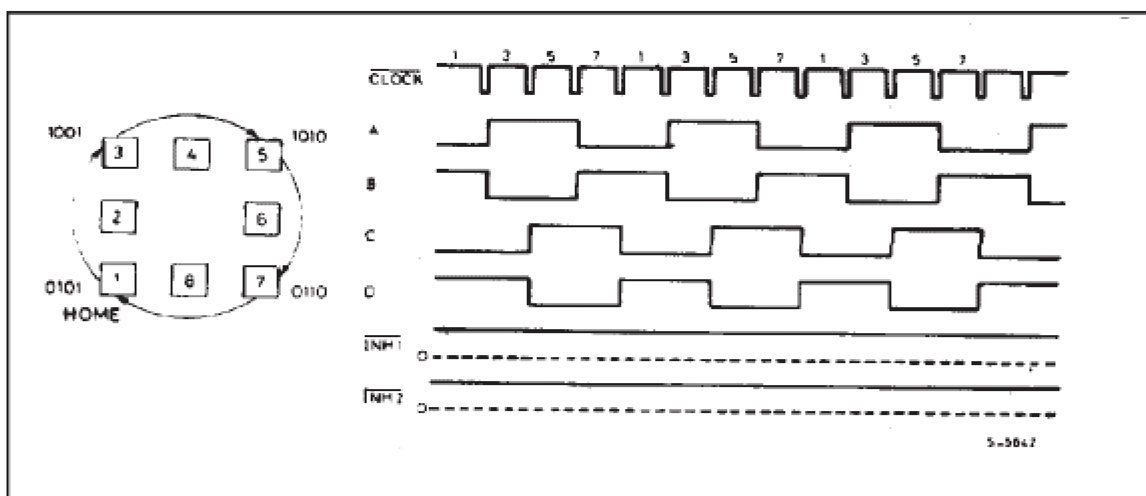
### HALF STEP MODE

Half step mode is selected by a high level on the HALF/$\overline{\text{FULL}}$ input.




### NORMAL DRIVE MODE

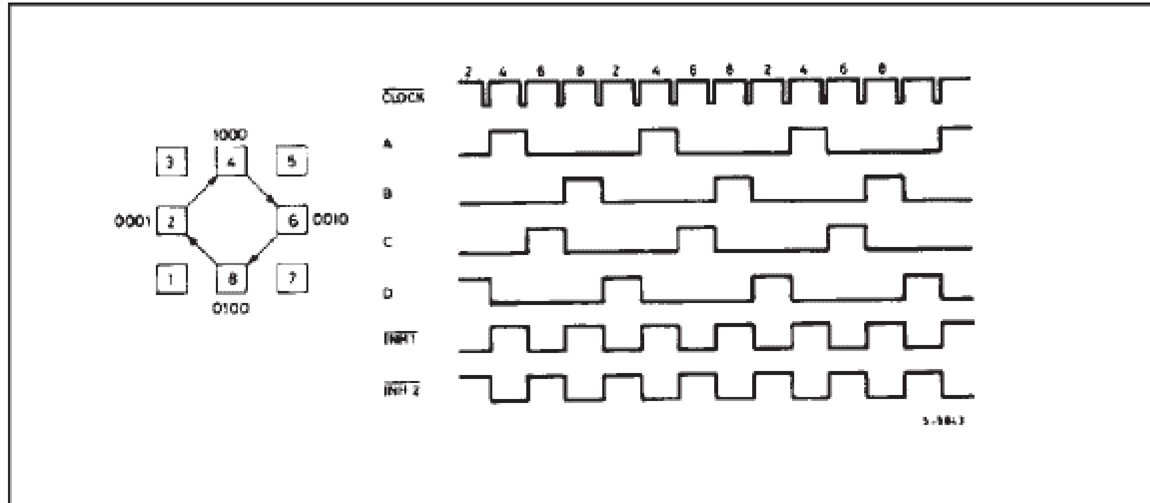
Normal drive mode (also called "two-phase-on" drive) is selected by a low level on the HALF/$\overline{\text{FULL}}$ input when the translator is at an odd numbered state (1, 3, 5 or 7). In this mode the $\overline{\text{INH1}}$ and $\overline{\text{INH2}}$ outputs remain high throughout.



**MOTOR DRIVING PHASE SEQUENCES** (continued)

WAVE DRIVE MODE

Wave drive mode (also called "one-phase-on" drive) is selected by a low level on the HALF/$\overline{\text{FULL}}$ input when the translator is at an even numbered state (2, 4, 6 or 8).



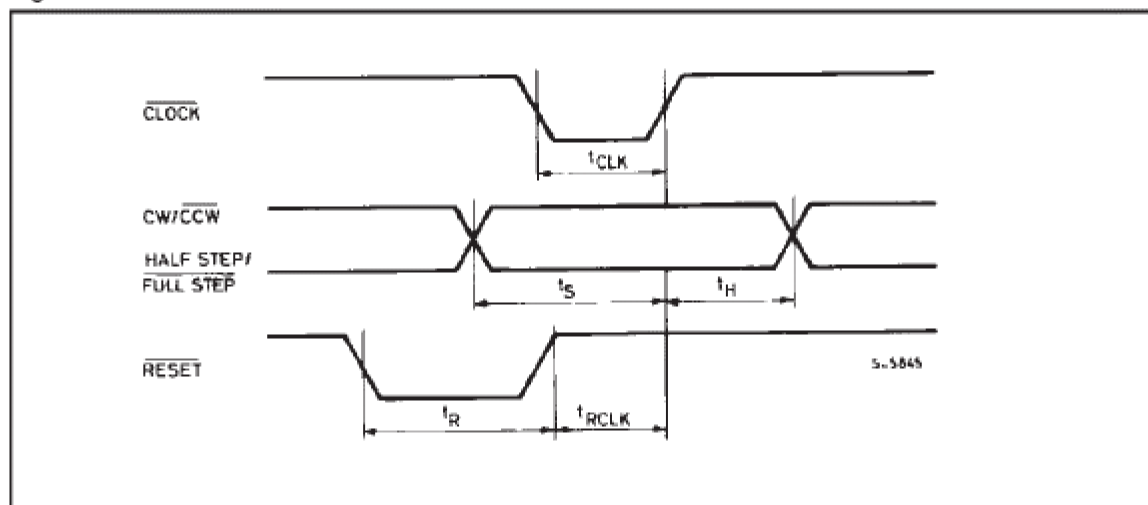

**ELECTRICAL CHARACTERISTICS** (Refer to the block diagram $T_{amb}$ = 25°C, $V_s$ = 5V unless otherwise specified)

| Symbol | Parameter | Test conditions | | Min. | Typ | Max. | Unit |
|---|---|---|---|---|---|---|---|
| $V_s$ | Supply voltage (pin 12) | | | 4.75 | | 7 | V |
| $I_s$ | Quiescent supply current (pin 12) | Outputs floating | | | 50 | 80 | mA |
| $V_i$ | Input voltage (pin 11, 17, 18, 19, 20) | | Low | | | 0.6 | V |
| | | | High | 2 | | $V_s$ | V |
| $I_i$ | Input current (pin 11, 17, 18, 19, 20) | | $V_i$ = L | | 100 | µA | |
| | | | $V_i$ = H | | | 10 | µA |
| $V_{en}$ | Enable input voltage (pin 10) | | Low | | | 1.3 | V |
| | | | High | 2 | | $V_s$ | V |
| $I_{en}$ | Enable input current (pin 10) | | $V_{en}$ = L | | | 100 | µA |
| | | | $V_{en}$ = H | | | 10 | µA |
| $V_o$ | Phase output voltage (pins 4, 6, 7, 9) | $I_o$ = 10mA | $V_{OL}$ | | | 0.4 | V |
| | | $I_o$ = 5mA | $V_{OH}$ | 3.9 | | | V |
| $V_{inh}$ | Inhibit output voltage (pins 5, 8) | $I_o$ = 10mA | $V_{inh\,L}$ | | | 0.4 | V |
| | | $I_o$ = 5mA | $V_{inh\,H}$ | 3.9 | | | V |
| $V_{SYNC}$ | Sync Output Voltage | $I_o$ = 5mA | $V_{SYNC\,H}$ | 3.3 | | | V |
| | | $I_o$ = 5mA | $V_{SYNC\,V}$ | | | 0.8 | |

**ELECTRICAL CHARACTERISTICS** (continued)

| Symbol | Parameter | Test conditions | Min. | Typ | Max. | Unit |
|---|---|---|---|---|---|---|
| $I_{leak}$ | Leakage current (pin 3) | $V_{CE}$ = 7 V | | | 1 | µA |
| $V_{sat}$ | Saturation voltage (pin 3) | I = 5 mA | | | 0.4 | V |
| $V_{off}$ | Comparators offset voltage (pins 13, 14, 15) | $V_{ref}$ = 1 V | | | 5 | mV |
| $I_o$ | Comparator bias current (pins 13, 14, 15) | | -100 | | 10 | µA |
| $V_{ref}$ | Input reference voltage (pin 15) | | 0 | | 3 | V |
| $t_{CLK}$ | Clock time | | 0.5 | | | µs |
| $t_S$ | Set up time | | 1 | | | µs |
| $t_H$ | Hold time | | 4 | | | µs |
| $t_R$ | Reset time | | 1 | | | µs |
| $t_{RCLK}$ | Reset to clock delay | | 1 | | | µs |

**Figure 1.**



## APPLICATION INFORMATION

TWO PHASE BIPOLAR STEPPER MOTOR CONTROL CIRCUIT

This circuit drives bipolar stepper motors with winding currents up to 2A. The diodes are fast 2A types.

Figure 2.

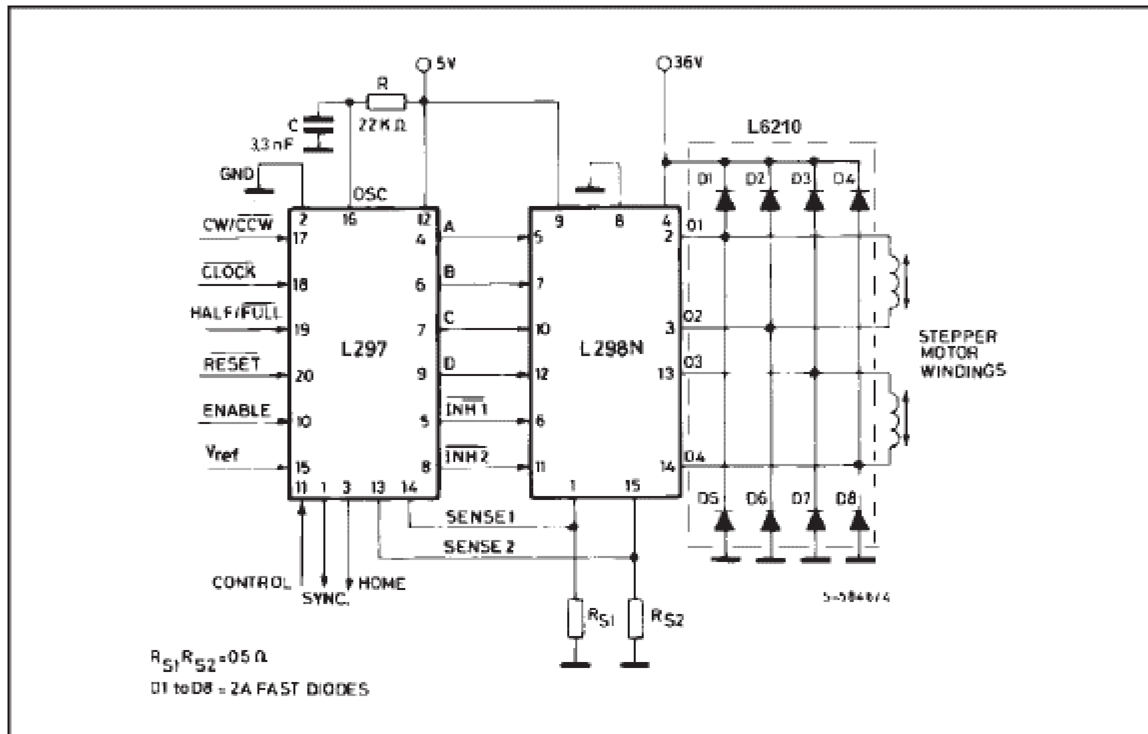
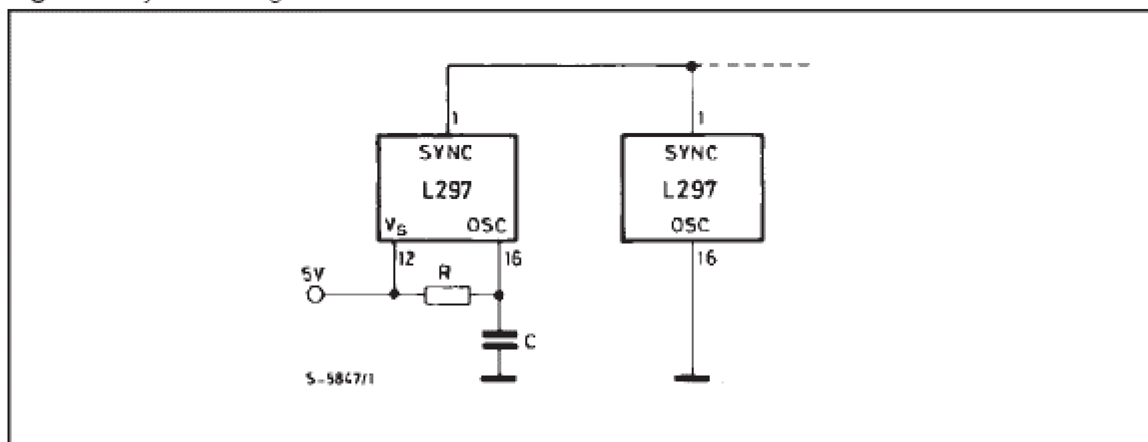



Figure 3 : Synchronising L297s

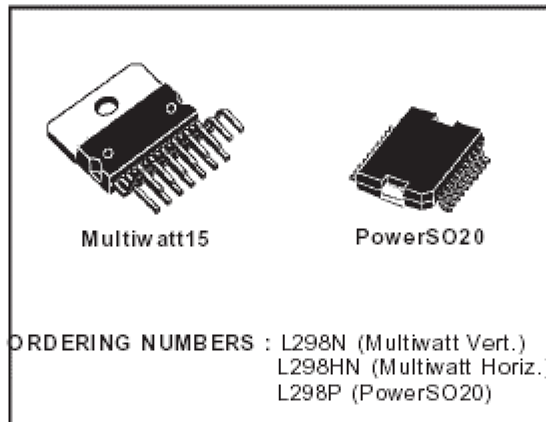


# APPENDIX 5 – L298 DUAL FULL-BRIDGE DRIVER IC

# L298

## DUAL FULL-BRIDGE DRIVER

- OPERATING SUPPLY VOLTAGE UP TO 46 V
- TOTAL DC CURRENT UP TO 4 A
- LOW SATURATION VOLTAGE
- OVERTEMPERATURE PROTECTION
- LOGICAL "0" INPUT VOLTAGE UP TO 1.5 V (HIGH NOISE IMMUNITY)

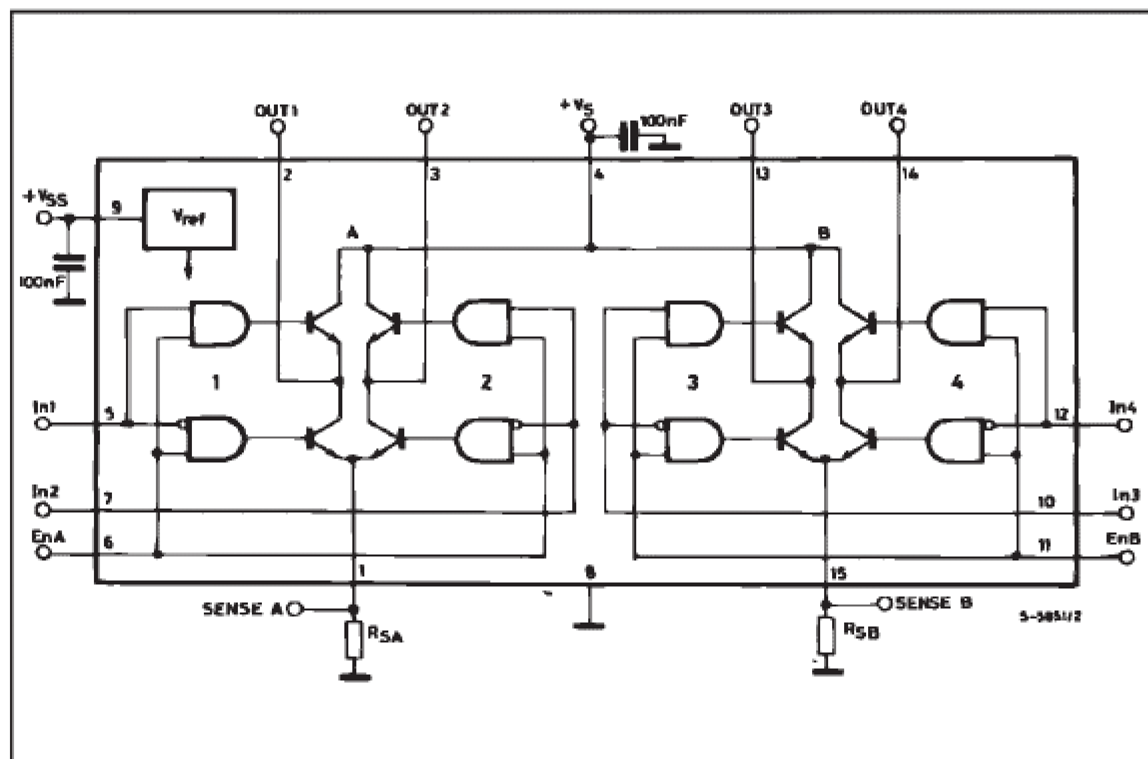



ORDERING NUMBERS : L298N (Multiwatt Vert.)
L298HN (Multiwatt Horiz.)
L298P (PowerSO20)

### DESCRIPTION

The L298 is an integrated monolithic circuit in a 15-lead Multiwatt and PowerSO20 packages. It is a high voltage, high current dual full-bridge driver designed to accept standard TTL logic levels and drive inductive loads such as relays, solenoids, DC and stepping motors. Two enable inputs are provided to enable or disable the device independently of the input signals. The emitters of the lower transistors of each bridge are connected together and the corresponding external terminal can be used for the connection of an external sensing resistor. An additional supply input is provided so that the logic works at a lower voltage.

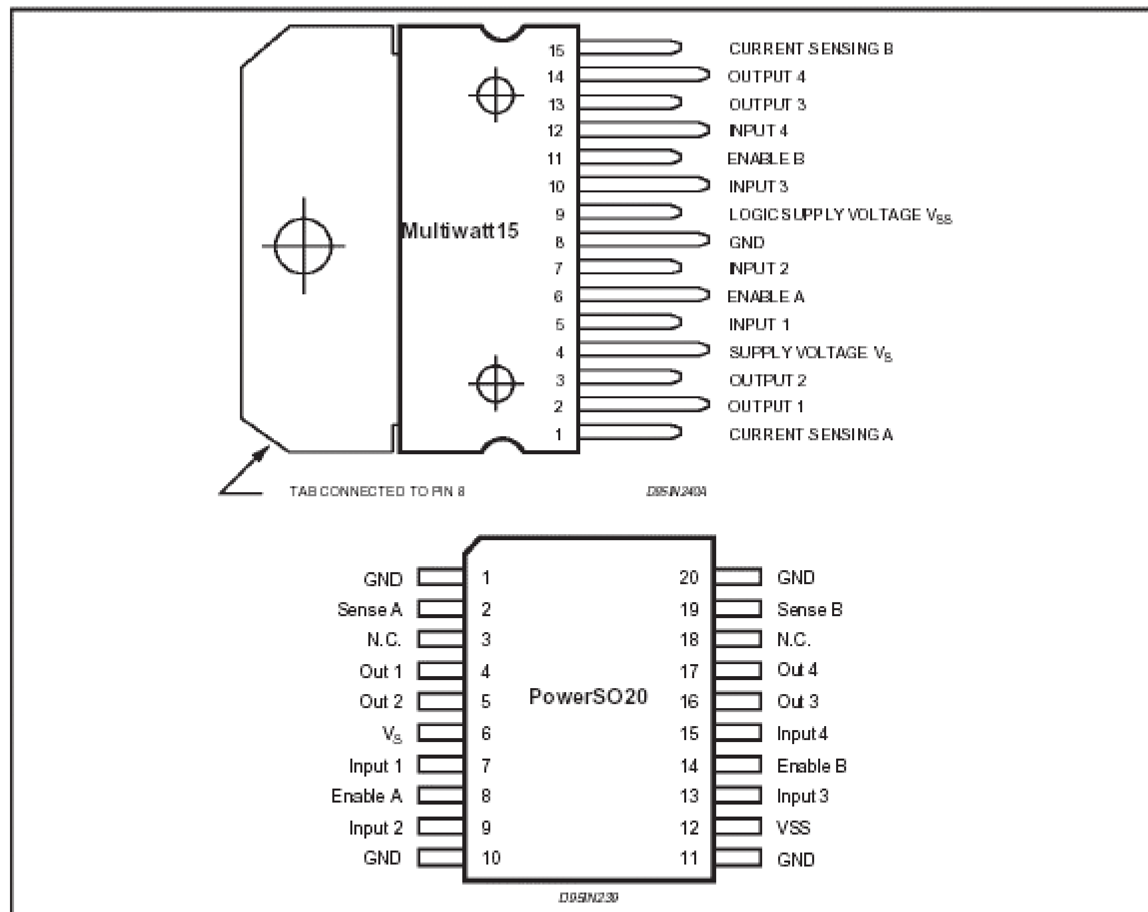
### BLOCK DIAGRAM



## ABSOLUTE MAXIMUM RATINGS

| Symbol | Parameter | Value | Unit |
|---|---|---|---|
| $V_S$ | Power Supply | 50 | V |
| $V_{SS}$ | Logic Supply Voltage | 7 | V |
| $V_I$,$V_{en}$ | Input and Enable Voltage | –0.3 to 7 | V |
| $I_O$ | Peak Output Current (each Channel)<br>– Non Repetitive (t = 100μs)<br>–Repetitive (80% on –20% off; $t_{on}$ = 10ms)<br>–DC Operation | <br>3<br>2.5<br>2 | <br>A<br>A<br>A |
| $V_{sens}$ | Sensing Voltage | –1 to 2.3 | V |
| $P_{tot}$ | Total Power Dissipation ($T_{case}$ = 75°C) | 25 | W |
| $T_{op}$ | Junction Operating Temperature | –25 to 130 | °C |
| $T_{stg}$, $T_j$ | Storage and Junction Temperature | –40 to 150 | °C |

## PIN CONNECTIONS (top view)




## THERMAL DATA

| Symbol | Parameter | | PowerSO20 | Multiwatt15 | Unit |
|---|---|---|---|---|---|
| $R_{th\ j\text{-}case}$ | Thermal Resistance Junction-case | Max. | – | 3 | °C/W |
| $R_{th\ j\text{-}amb}$ | Thermal Resistance Junction-ambient | Max. | 13 (*) | 35 | °C/W |

(*) Mounted on aluminum substrate

**PIN FUNCTIONS** (refer to the block diagram)

| MW.15 | PowerSO | Name | Function |
|---|---|---|---|
| 1;15 | 2;19 | Sense A; Sense B | Between this pin and ground is connected the sense resistor to control the current of the load. |
| 2;3 | 4;5 | Out 1; Out 2 | Outputs of the Bridge A; the current that flows through the load connected between these two pins is monitored at pin 1. |
| 4 | 6 | $V_S$ | Supply Voltage for the Power Output Stages. A non-inductive 100nF capacitor must be connected between this pin and ground. |
| 5;7 | 7;9 | Input 1; Input 2 | TTL Compatible Inputs of the Bridge A. |
| 6;11 | 8;14 | Enable A; Enable B | TTL Compatible Enable Input: the L state disables the bridge A (enable A) and/or the bridge B (enable B). |
| 8 | 1,10,11,20 | GND | Ground. |
| 9 | 12 | VSS | Supply Voltage for the Logic Blocks. A100nF capacitor must be connected between this pin and ground |
| 10; 12 | 13;15 | Input 3; Input 4 | TTL Compatible Inputs of the Bridge B. |
| 13; 14 | 16;17 | Out 3; Out 4 | Outputs of the Bridge B. The current that flows through the load connected between these two pins is monitored at pin 15. |
| – | 3;18 | N.C. | Not Connected |

**ELECTRICAL CHARACTERISTICS** ($V_S$ = 42V; $V_{SS}$ = 5V, $T_j$ = 25°C; unless otherwise specified)

| Symbol | Parameter | Test Conditions | | Min. | Typ. | Max. | Unit |
|---|---|---|---|---|---|---|---|
| $V_S$ | Supply Voltage (pin 4) | Operative Condition | | $V_{IH}$ +2.5 | | 46 | V |
| $V_{SS}$ | Logic Supply Voltage (pin 9) | | | 4.5 | 5 | 7 | V |
| $I_S$ | Quiescent Supply Current (pin 4) | $V_{en}$ = H; $I_L$ = 0 | $V_i$ = L<br>$V_i$ = H | | 13<br>50 | 22<br>70 | mA<br>mA |
| | | $V_{en}$ = L | $V_i$ = X | | | 4 | mA |
| $I_{SS}$ | Quiescent Current from $V_{SS}$ (pin 9) | $V_{en}$ = H; $I_L$ = 0 | $V_i$ = L<br>$V_i$ = H | | 24<br>7 | 36<br>12 | mA<br>mA |
| | | $V_{en}$ = L | $V_i$ = X | | | 6 | mA |
| $V_{iL}$ | Input Low Voltage (pins 5, 7, 10, 12) | | | –0.3 | | 1.5 | V |
| $V_{iH}$ | Input High Voltage (pins 5, 7, 10, 12) | | | 2.3 | | VSS | V |
| $I_{iL}$ | Low Voltage Input Current (pins 5, 7, 10, 12) | $V_i$ = L | | | | –10 | µA |
| $I_{iH}$ | High Voltage Input Current (pins 5, 7, 10, 12) | Vi = H $\leq V_{SS}$ –0.6V | | | 30 | 100 | µA |
| $V_{en}$ = L | Enable Low Voltage (pins 6, 11) | | | –0.3 | | 1.5 | V |
| $V_{en}$ = H | Enable High Voltage (pins 6, 11) | | | 2.3 | | $V_{SS}$ | V |
| $I_{en}$ = L | Low Voltage Enable Current (pins 6, 11) | $V_{en}$ = L | | | | –10 | µA |
| $I_{en}$ = H | High Voltage Enable Current (pins 6, 11) | $V_{en}$ = H $\leq V_{SS}$ –0.6V | | | 30 | 100 | µA |
| $V_{CEsat(H)}$ | Source Saturation Voltage | $I_L$ = 1A<br>$I_L$ = 2A | | 0.95 | 1.35<br>2 | 1.7<br>2.7 | V<br>V |
| $V_{CEsat(L)}$ | Sink Saturation Voltage | $I_L$ = 1A (5)<br>$I_L$ = 2A (5) | | 0.85 | 1.2<br>1.7 | 1.6<br>2.3 | V<br>V |
| $V_{CEsat}$ | Total Drop | $I_L$ = 1A (5)<br>$I_L$ = 2A (5) | | 1.80 | | 3.2<br>4.9 | V<br>V |
| $V_{sens}$ | Sensing Voltage (pins 1, 15) | | | –1 (1) | | 2 | V |

**ELECTRICAL CHARACTERISTICS** (continued)

| Symbol | Parameter | Test Conditions | Min. | Typ. | Max. | Unit |
|---|---|---|---|---|---|---|
| $T_1$ ($V_i$) | Source Current Turn-off Delay | 0.5 $V_i$ to 0.9 $I_L$ (2); (4) | | 1.5 | | μs |
| $T_2$ ($V_i$) | Source Current Fall Time | 0.9 $I_L$ to 0.1 $I_L$ (2); (4) | | 0.2 | | μs |
| $T_3$ ($V_i$) | Source Current Turn-on Delay | 0.5 $V_i$ to 0.1 $I_L$ (2); (4) | | 2 | | μs |
| $T_4$ ($V_i$) | Source Current Rise Time | 0.1 $I_L$ to 0.9 $I_L$ (2); (4) | | 0.7 | | μs |
| $T_5$ ($V_i$) | Sink Current Turn-off Delay | 0.5 $V_i$ to 0.9 $I_L$ (3); (4) | | 0.7 | | μs |
| $T_6$ ($V_i$) | Sink Current Fall Time | 0.9 $I_L$ to 0.1 $I_L$ (3); (4) | | 0.25 | | μs |
| $T_7$ ($V_i$) | Sink Current Turn-on Delay | 0.5 $V_i$ to 0.9 $I_L$ (3); (4) | | 1.6 | | μs |
| $T_8$ ($V_i$) | Sink Current Rise Time | 0.1 $I_L$ to 0.9 $I_L$ (3); (4) | | 0.2 | | μs |
| fc ($V_i$) | Commutation Frequency | $I_L$ = 2A | | 25 | 40 | KHz |
| $T_1$ ($V_{en}$) | Source Current Turn-off Delay | 0.5 $V_{en}$ to 0.9 $I_L$ (2); (4) | | 3 | | μs |
| $T_2$ ($V_{en}$) | Source Current Fall Time | 0.9 $I_L$ to 0.1 $I_L$ (2); (4) | | 1 | | μs |
| $T_3$ ($V_{en}$) | Source Current Turn-on Delay | 0.5 $V_{en}$ to 0.1 $I_L$ (2); (4) | | 0.3 | | μs |
| $T_4$ ($V_{en}$) | Source Current Rise Time | 0.1 $I_L$ to 0.9 $I_L$ (2); (4) | | 0.4 | | μs |
| $T_5$ ($V_{en}$) | Sink Current Turn-off Delay | 0.5 $V_{en}$ to 0.9 $I_L$ (3); (4) | | 2.2 | | μs |
| $T_6$ ($V_{en}$) | Sink Current Fall Time | 0.9 $I_L$ to 0.1 $I_L$ (3); (4) | | 0.35 | | μs |
| $T_7$ ($V_{en}$) | Sink Current Turn-on Delay | 0.5 $V_{en}$ to 0.9 $I_L$ (3); (4) | | 0.25 | | μs |
| $T_8$ ($V_{en}$) | Sink Current Rise Time | 0.1 $I_L$ to 0.9 $I_L$ (3); (4) | | 0.1 | | μs |

1) 1)Sensing voltage can be –1 V for t ≤ 50 μsec; in steady state $V_{sens}$ min ≥ –0.5 V.
2) See fig. 2.
3) See fig. 4.
4) The load must be a pure resistor.

**Figure 1 :** Typical Saturation Voltage vs. Output Current.

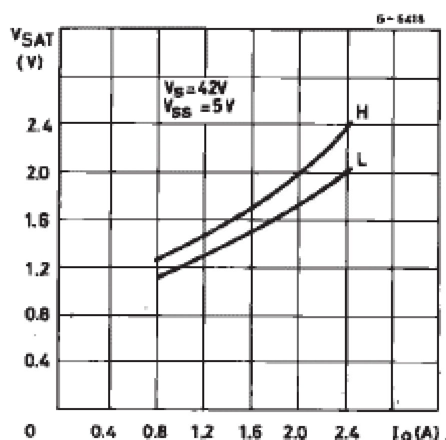
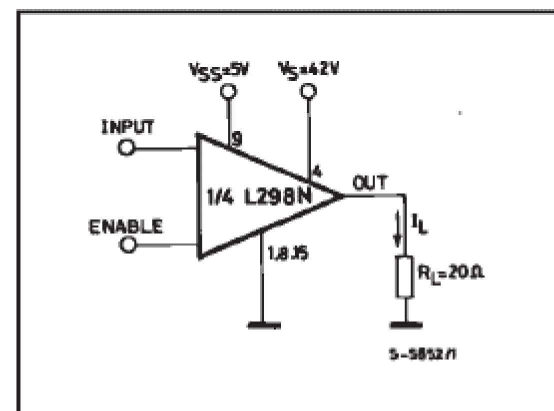



**Figure 2 :** Switching Times Test Circuits.




Note : For INPUT Switching, set EN = H
For ENABLE Switching, set IN = H

**Figure 3 :** Source Current Delay Times vs. Input or Enable Switching.

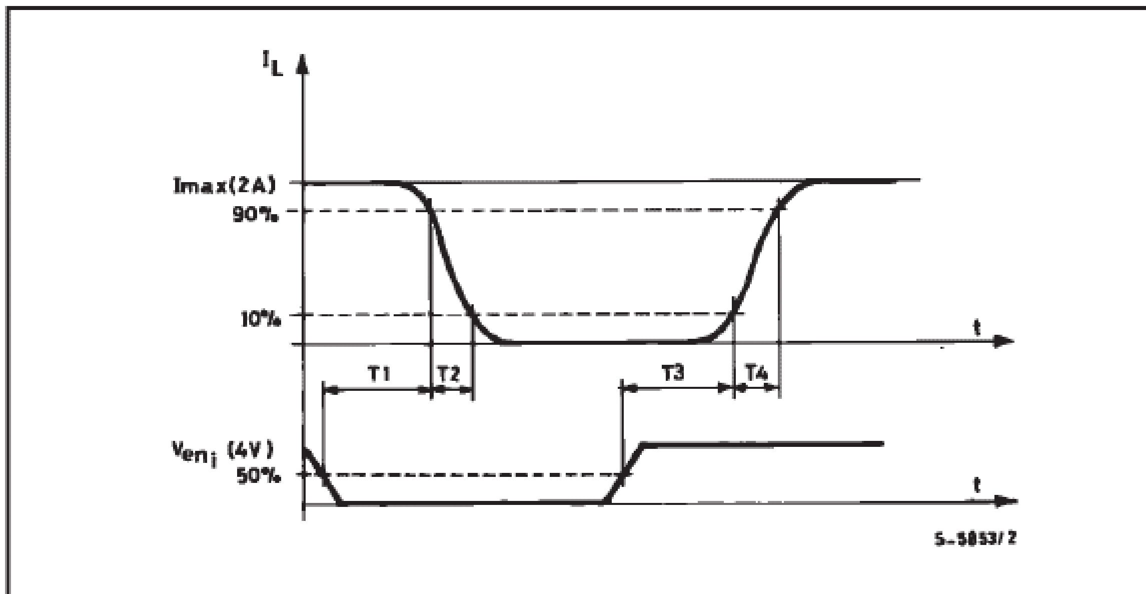
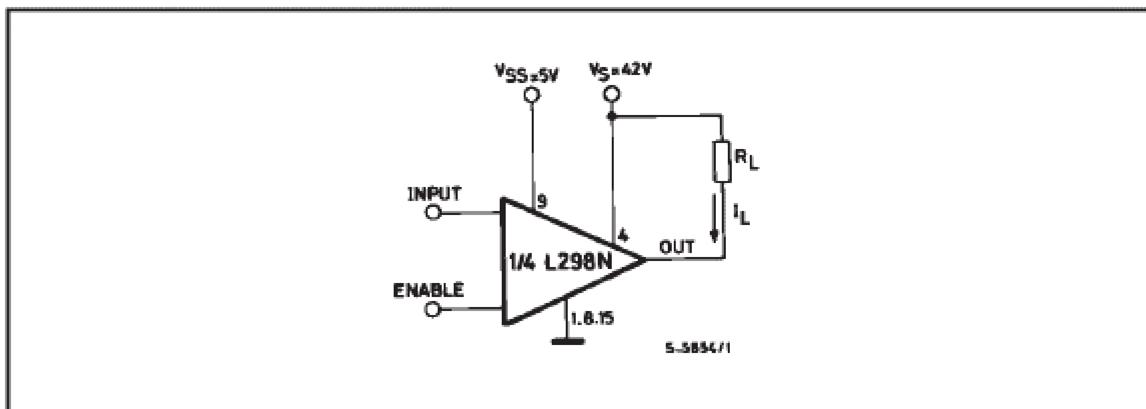



**Figure 4 :** Switching Times Test Circuits.




Note : For INPUT Switching, set EN = H
For ENABLE Switching, set IN = L

**Figure 5 :** Sink Current Delay Times vs. Input 0 V Enable Switching.

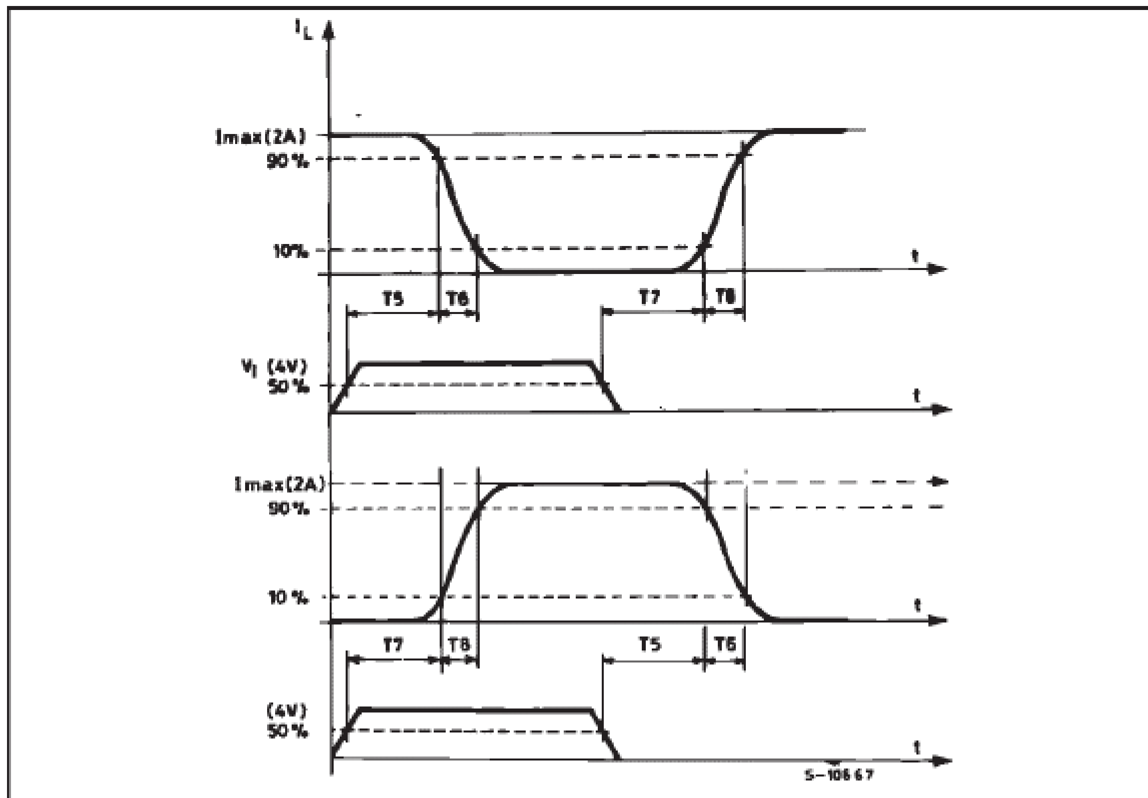
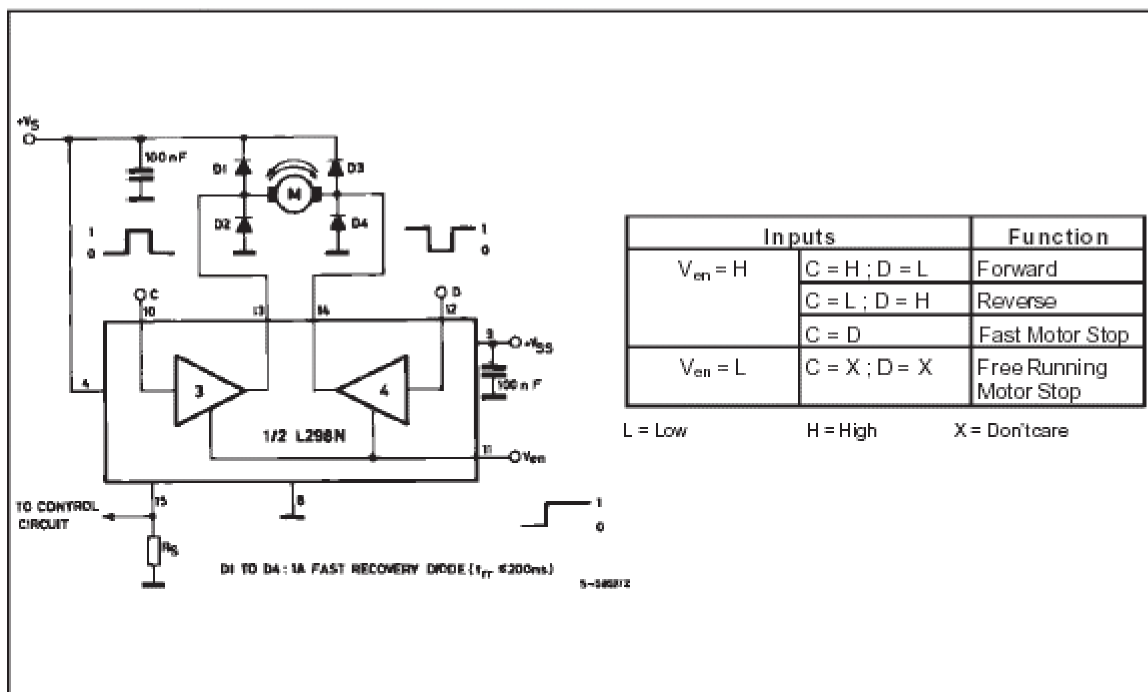


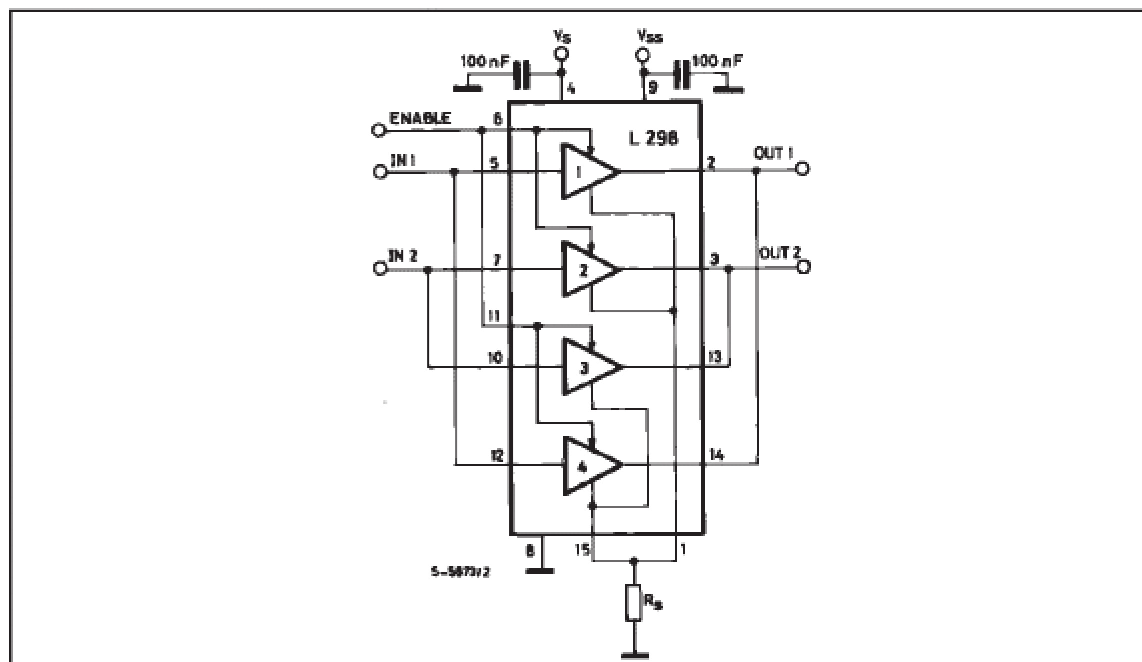

**Figure 6 :** Bidirectional DC Motor Control.




| Inputs | | Function |
|---|---|---|
| $V_{en}$ = H | C = H ; D = L | Forward |
| | C = L ; D = H | Reverse |
| | C = D | Fast Motor Stop |
| $V_{en}$ = L | C = X ; D = X | Free Running Motor Stop |

L = Low H = High X = Don't care

**Figure 7 :** For higher currents, outputs can be paralleled. Take care to parallel channel 1 with channel 4 and channel 2 with channel 3.




**APPLICATION INFORMATION (Refer to the block diagram)**

1.1. POWER OUTPUT STAGE

The L298 integrates two power output stages (A ; B). The power output stage is a bridge configuration and its outputs can drive an inductive load in common or differenzial mode, depending on the state of the inputs. The current that flows through the load comes out from the bridge at the sense output : an external resistor ($R_{SA}$ ; $R_{SB}$.) allows to detect the intensity of this current.

1.2. INPUT STAGE

Each bridge is driven by means of four gates the input of which are In1 ; In2 ; EnA and In3 ; In4 ; EnB. The In inputs set the bridge state when The En input is high ; a low state of the En input inhibits the bridge. All the inputs are TTL compatible.

2. SUGGESTIONS

A non inductive capacitor, usually of 100 nF, must be foreseen between both Vs and Vss, to ground, as near as possible to GND pin. When the large capacitor of the power supply is too far from the IC, a second smaller one must be foreseen near the L298.

The sense resistor, not of a wire wound type, must be grounded near the negative pole of Vs that must be near the GND pin of the I.C.

Each input must be connected to the source of the driving signals by means of a very short path.

Turn-On and Turn-Off : Before to Turn-ON the Supply Voltage and before to Turn it OFF, the Enable input must be driven to the Low state.

3. APPLICATIONS

Fig 6 shows a bidirectional DC motor control Schematic Diagram for which only one bridge is needed. The external bridge of diodes D1 to D4 is made by four fast recovery elements (trr ≤ 200 nsec) that must be chosen of a VF as low as possible at the worst case of the load current.

The sense output voltage can be used to control the current amplitude by chopping the inputs, or to provide overcurrent protection by switching low the enable input.

The brake function (Fast motor stop) requires that the Absolute Maximum Rating of 2 Amps must never be overcome.

When the repetitive peak current needed from the load is higher than 2 Amps, a paralleled configuration can be chosen (See Fig.7).

An external bridge of diodes are required when inductive loads are driven and when the inputs of the IC are chopped ; Shottky diodes would be preferred.

This solution can drive until 3 Amps In DC operation and until 3.5 Amps of a repetitive peak current.

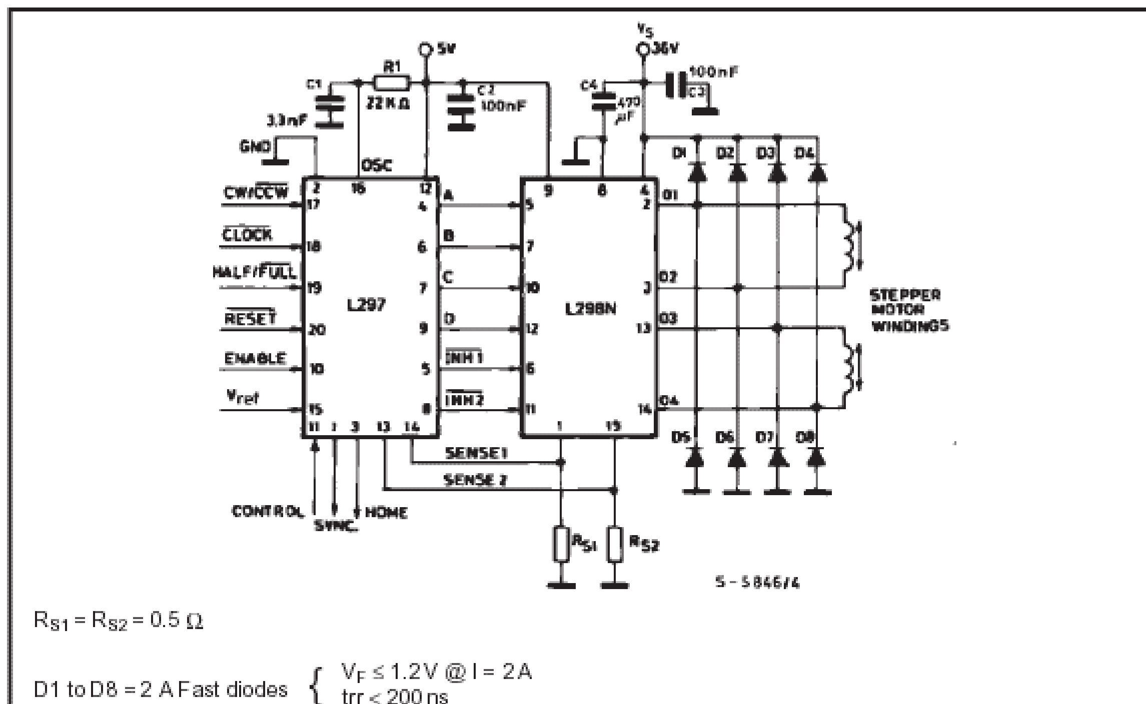
On Fig 8 it is shown the driving of a two phase bipolar stepper motor ; the needed signals to drive the inputs of the L298 are generated, in this example, from the IC L297.

Fig 9 shows an example of P.C.B. designed for the application of Fig 8.

Fig 10 shows a second two phase bipolar stepper motor control circuit where the current is controlled by the I.C. L6506.

**Figure 8 :** Two Phase Bipolar Stepper Motor Circuit.

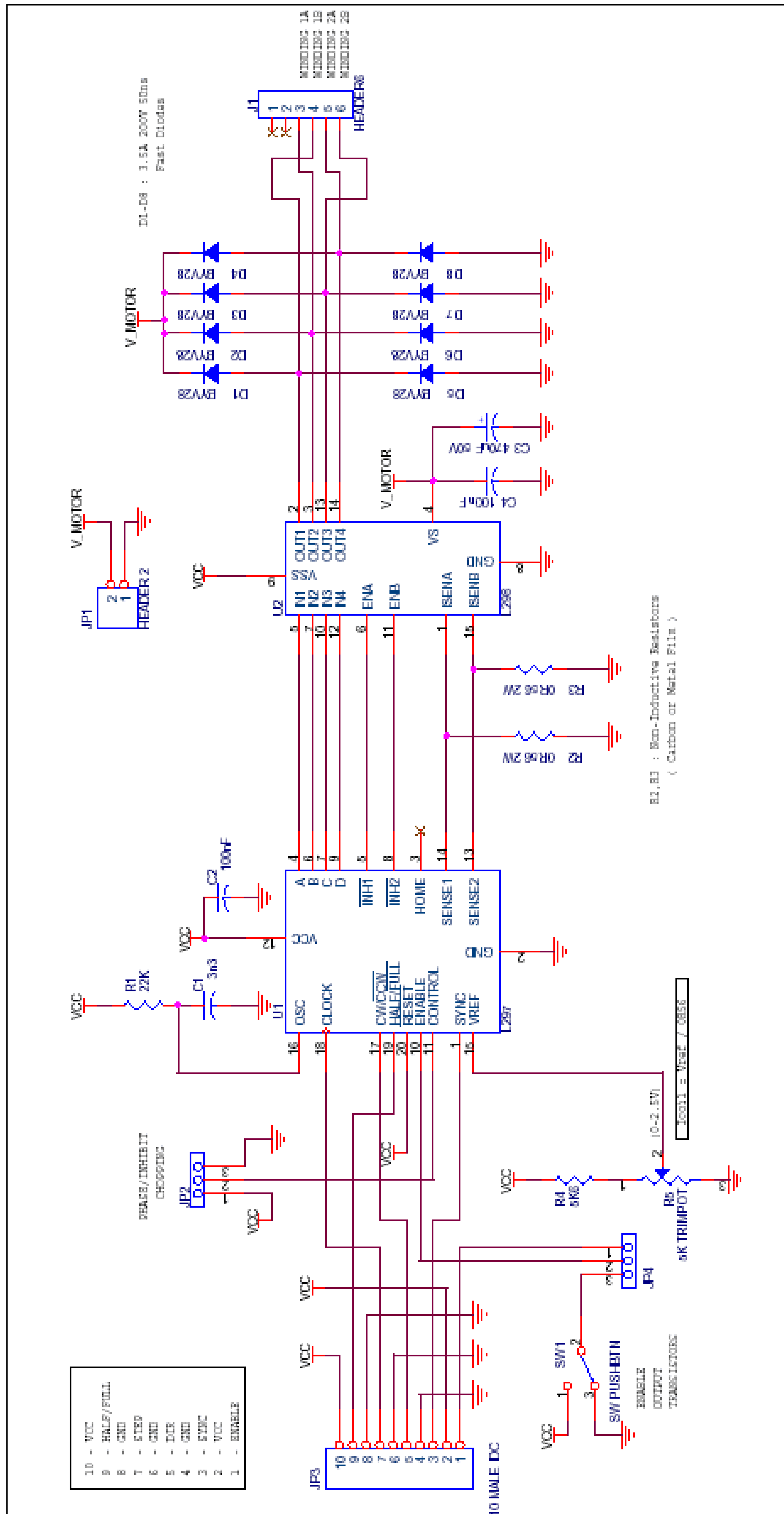
This circuit drives bipolar stepper motors with winding currents up to 2 A. The diodes are fast 2 A types.



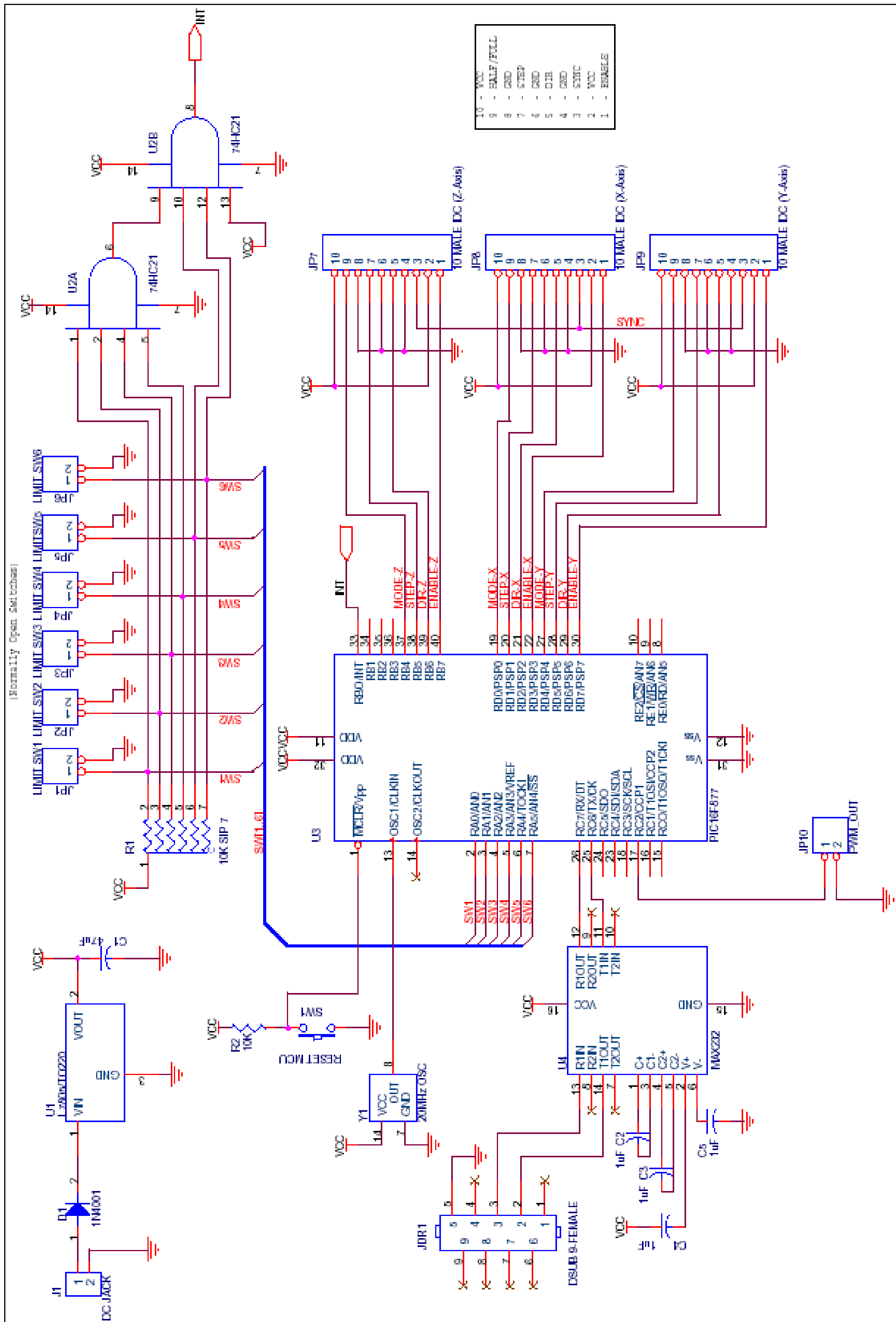

$R_{S1} = R_{S2} = 0.5\ \Omega$

D1 to D8 = 2 A Fast diodes $\left\{ \begin{array}{l} V_F \leq 1.2V \ @ \ I = 2A \\ trr \leq 200\ ns \end{array} \right.$

# APPENDIX 6 – STEPPER MOTOR DRIVER SCHEMATIC



# APPENDIX 7 – CONTROL BOARD SCHEMATIC



## APPENDIX 8 – 3-D SOLID MODELS OF THE MECHANICS