# CHAPTER 1: INTRODUCTION

## PROJECT BACKGROUND

Stepper motors have become a major part of all industrial processes involving accurate positioning. Nowadays, stepper motors can be found in many computer peripherals, business machines, process control hardware and Computer Numerical Control (CNC) machines.

Technological breakthroughs in stepper motor industry have made their torque and speed ratings comparable to those of other low-power DC motor types. Stepper motors are thus becoming more favorable for their rotational drive power and fine positioning accuracy.

The advancement of stepper motor technology is highly linked to the rapid-growing technology of **stepper motor controllers**. A stepper motor controller is a device that interfaces a high-level digital system to one or more stepper motors. In other words, it translates high-level commands received from a control system, such as a computer or a PLC, into driving voltage signals that operate the stepper motors. Controllers that can operate multiple stepper motors at the same time are known as **multi-axis controllers**.

Many devices that incorporate multiple stepper motors require complex control and **synchronization** between the different axes of motion controlled by the stepper motors. An example of this would be a robotic arm; to produce a certain motion, one stepper motor may need to operate at twice the speed of the others, while another may need to reverse its direction at a certain time instance.

This complex control is often arranged by a multi-axis controller. Such controllers are capable of operating all stepper motors in different arrangements and in complete synchronization.

Another key feature of a stepper motor controller is **Asynchronous Operation**. A controller that is engaged with driving the motors should still be able to report to the controlling digital system and, even better, be able to receive further commands to possibly alter the active job.

Below are two example applications that demonstrate the need for multi-axis stepper motor controllers:

## 1. Robotic Arm

A robotic arm is a motor-based actuator that can move and grab objects. Stepper motors are the dominant type of motors in robotic arms due to the high positioning-accuracy nature of the application.

A robotic arm will require synchronized motion of all integrated motors in order to produce the desired arm action.

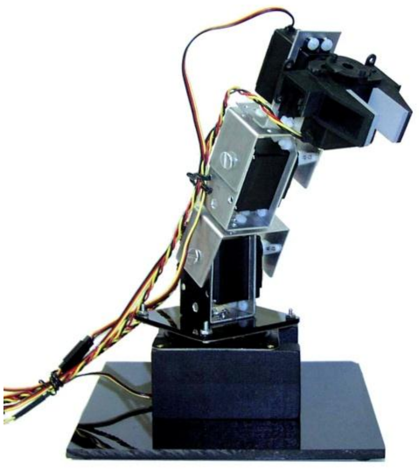


Figure 1.1: Robotic Arm

## 2. XYZ Gantry

A gantry motion system is an assembled set of linear actuators that can provide motion in two or three dimensions. An XYZ gantry will include no less than three stepper motors; one per each axis of motion.

Operating an XYZ gantry may require driving stepper motors at different speeds. It is also typical to change the motion of the moving stroke abruptly due to a system-related event such as reaching the travel limit of one of the actuators.

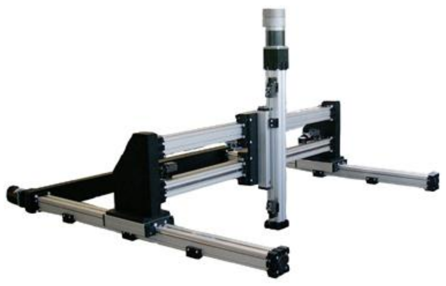


Figure 1.2: XYZ Gantry

## OUR CONTROLLER

The controller presented in this graduation project is a multi-axis stepper motor controller. It can drive and fully synchronize the motion of **three** stepper motors. It receives commands from a computer or possibly other digital systems via a **serial link**, and drives the stepper motors accordingly.

The controller has a number of interesting features:

1. **Wide Range of Operating Configurations:**
   Each axis can be configured and controlled independently to operate in one of many available running configurations.

2. **Fully Asynchronous Operation**:
   While driving the motors, the controller will continue to receive commands and report status messages via serial communication.

3. **Synchronized and Accurate Positioning**:
   The controller can drive motors in complete synchronization, and with a perfect positioning accuracy.

4. **Active Job Adjustment:**
   The controller can change motion parameters during job execution.

5. **Push Reporting:**
   The controller actively notifies the computer at task completion events.

In addition to all the features presented by the controller device itself, we have provided a computer software application that allows users to create **running scripts** for the controller. This application will translate scripts designed by users through a Graphical User Interface (GUI) to corresponding commands and transmit them to the controller instantly. The computer software provides numerous features that expand the capabilities of the controller even more! See Chapter 6 for more details.

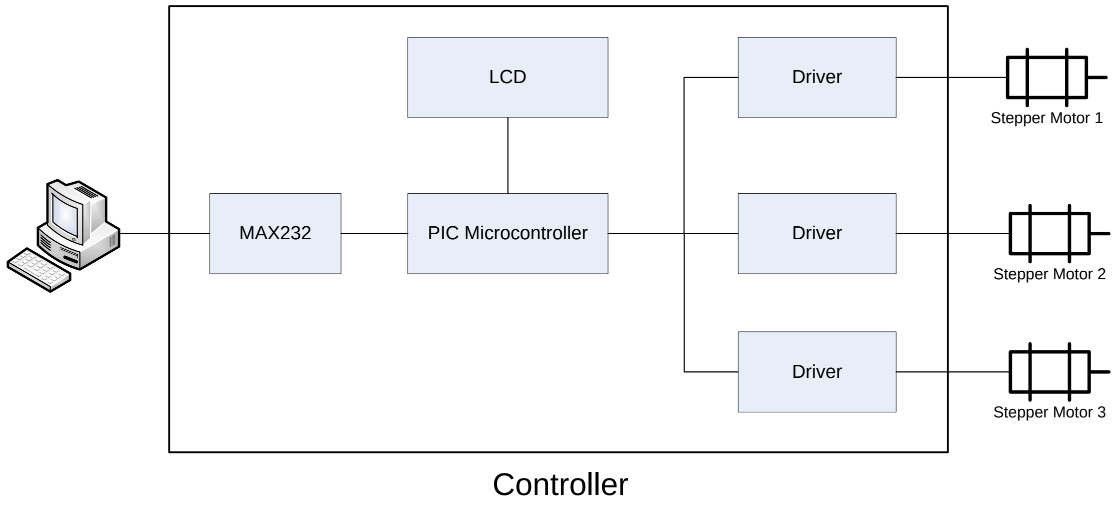


Figure 1.3: Controller Block Diagram

Figure 1.3 shows the block diagram for our controller. The controller consists mainly of a PIC microcontroller, a Liquid Crystal Display (LCD), three driving circuits and a MAX232 chip. For a complete circuit diagram, refer to Figure 5.1 (Chapter 5).

MAX232 is a chip that is used to convert TTL voltage levels to suitable levels for serial transmission and vice versa. It is an intermediate block that interfaces the PIC microcontroller to the computer's serial port.

The PIC microcontroller is the core of the stepper motor controller. It is responsible for parsing commands received through serial transmission and generating proper signals to the driving circuits to operate the motors.

The three driving circuits convert logic signals received from the microcontroller to high-voltage signals that operate the stepper motors. Each of them is controlled independently.

The LCD is used to display controller status, and is handled by the microcontroller.

For details about the individual parts of the controller, see Chapter 5.

# CHAPTER 2: STEPPER MOTORS

## INTRODUCTION

A stepper motor is an electromagnetic device that converts digital pulses into a mechanical rotational motion. It essentially consists of a permanent magnet rotating shaft, called the **rotor**, surrounded by electromagnets called **stators**.

The general principle behind it is very simple; the electromagnets surrounding the rotor are activated by applying voltage to them, at which point the rotor becomes attracted to the activated electromagnet causing a "step".

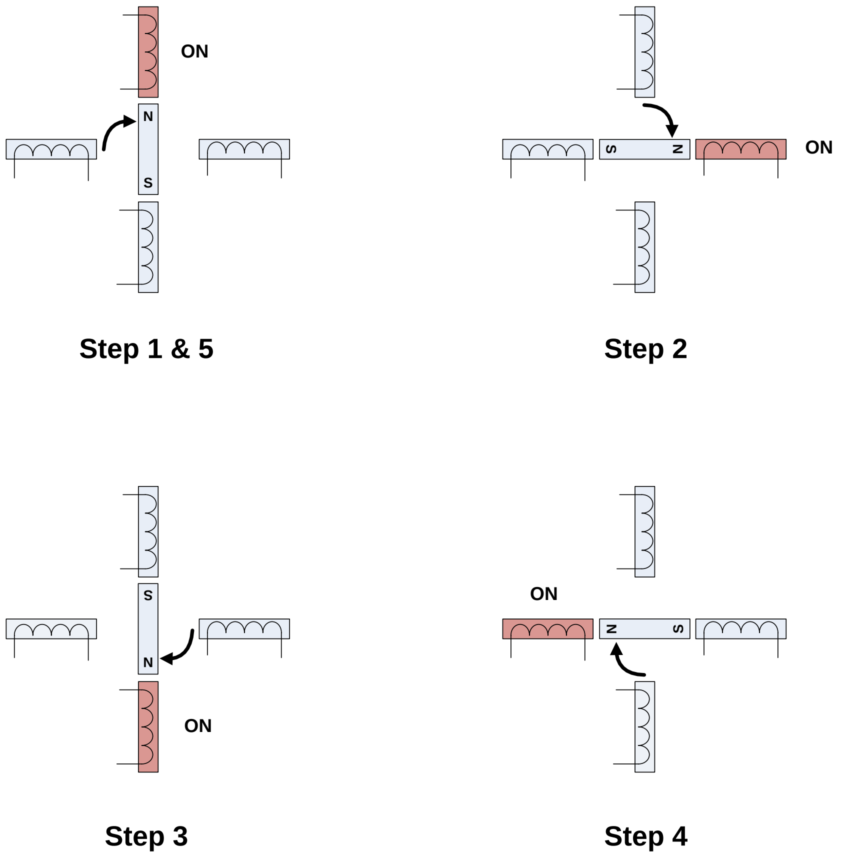


Figure 2.1: Stepper Motor Rotation

Figure 2.1 demonstrates one complete clockwise stepper motor rotation. The magnet at the top is deactivated and the one to the right is activated causing the rotor to make a 90 degree clockwise step, the same procedure is repeated for the bottom and the left electromagnets respectively until we finally reach the starting position, thus, completing a whole revolution.

## WIRING AND CONNECTIONS

Stepper motors are characterized by the number of electromagnetic coils they have. Each electromagnet is called a **phase**.

A two phase stepper motor has two basic winding arrangements for its electromagnetic coils; it can be either **unipolar** or **bipolar**.

### Bipolar Stepper Motors

A bipolar stepper motor has a single coil winding per phase; giving two leads per phase, that is; four leads for a two phase stepper motor as shown in Figure 2.2.

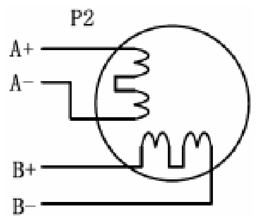


Figure 2.2: Bipolar Stepper Motor Windings

In order to reverse the magnetic pole, the current passing through the winding needs to be reversed. This type of motors is the least flexible, however, it is the easiest to wire.

### Unipolar Stepper Motors

The other arrangement is a unipolar stepper motor, which has two coil windings per phase, one for each direction of the current. Current in each winding flows only in one direction, therefore, the magnetic pole can be reversed without the need to actually reverse the direction of flow of the electric current.

In each phase, the two windings have one end in common, which will give three leads per phase, and hence six leads for a two phase stepper motor.

## Wiring Unipolar Stepper Motors as Bipolar

Unipolar stepper motors can be converted into bipolar with either of two simple wiring configurations. The first configuration is called the **Half Coil** configuration, and is shown in Figure 2.3.

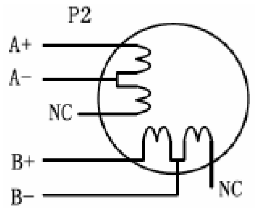


Figure 2.3: Half Coil Connection

The half coil configuration uses 50% of the motor phase windings; it will become similar to a 4-lead bipolar motor. This will result in a lower inductance, hence lower torque output. This configuration is more favorable at higher speeds due to its output stability.

The other configuration is called the **Full Coil** configuration, shown in Figure 2.4.

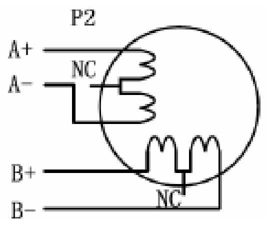


Figure 2.4: Full Coil Connection

This configuration uses all motor phase windings; it will also become similar to a 4-lead bipolar motor. However, this kind of winding is used in applications where high torque at low speeds is desired.

## Wiring 8-lead Stepper Motors as 4-lead

In addition to 4-lead and 6-lead, there are 8-lead stepper motors. This kind of motor configuration is similar to the unipolar wiring arrangement. However, the two windings in each phase are not connected, so there are no common leads.

This type offers a high degree of flexibility to system designers, because it can be connected either in series or in parallel, which satisfies a wide range of applications.

A series motor configuration is shown in Figure 2.5, it would be used in applications where a higher torque at lower speeds is required. The performance of such motors will start to decrease at higher speeds.

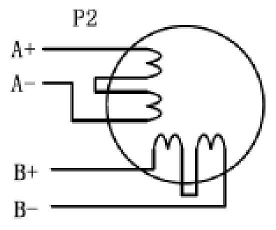


Figure 2.5: 8-Lead Motor Series Connection

A parallel motor configuration is shown in Figure 2.6, it gives a more stable output at low speeds with low torque. However, due to its low inductance, it will generate higher torque at higher speeds.

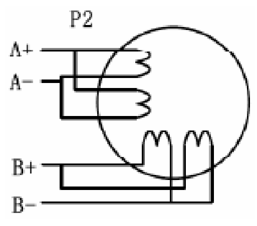


Figure 2.6: 8-Lead Motor Parallel Connection

In short, any type of 2-phase or 4-phase stepper motors can be wired as 4-lead bipolar. That’s why we chose the M325 drivers in our controller design; they provide the maximum level of compatibility with all stepper motor types.

# CHAPTER 3: PIC MICROCONTROLLERS

## INTRODUCTION

Microchip PIC Microcontrollers are simply on-chip computers! They are Integrated Circuits (ICs) that contain a basic processor, a register file (RAM) and a number of I/O ports. They are powerful tiny RISC machines.

PIC Microcontrollers come in many sizes and families, all having the same architecture but with different memory sizes and slightly different features. The PIC Microcontroller operating at the core of this stepper motor controller, being also the most popular in the Jordanian market, is **PIC16F877**.

PIC16F877 is a 40-pin microcontroller. It is a simple 8-bit machine with 8K words of program memory, 368 bytes of general purpose 8-bit registers (register file) and 33 configurable I/O ports.

PIC16F877 has numerous features. Some of the remarkable features that are used and greatly added to the functionality of our stepper motor controller are:

- **Universal Synchronous Receiver/Transmitter (USART) Unit:**

  The microcontroller has a built-in USART module that operates independently of program execution.

- **Timer-based Interrupts:**

  The microcontroller provides a number of counters that increment automatically on the device’s clock tick and raise an interrupt when they overflow to zero.

- **Low Instruction Execution Time:**

  The microcontroller executes one instruction every 4 clock cycles. Using a 4 MHz crystal oscillator would yield an execution throughput of 1 Mega instruction per second.

PIC microcontrollers are programmed in Microchip PIC assembly language. A program is created and assembled using PC software. A hardware programmer is then used to burn the program into the microcontroller's flash program memory.

The microcontroller is then taken off, mounted on a circuit board and will begin executing as soon as it powers up.

The chip requires CMOS supply voltages to operate. Optionally, a crystal oscillator can be used to provide clocking signal for the microcontroller. PIC16F877 supports up to 20 MHz oscillator frequency [4]. The one used in this project is a 4 MHz oscillator. It provides an adequate speed for processing.

Input/output pins are grouped into **Ports**. PIC16F877 has the following ports:

- PORTA = 5 bits
- PORTB = 8 bits
- PORTC = 8 bits
- PORTD = 8 bits
- PORTE = 3 bits

Each port can be configured as either input or output, and can be read or written to through program instructions. Almost all the pins on PIC microcontrollers are **multi-purpose**; they can be used for many different functions, but are multiplexed through internal tri-state buffers to one function at a time.

The ports of the PIC microcontroller in our project are used for serial communication, LCD interfacing, and stepper motor drivers interfacing. With the exception of pin RxD of serial communication, all ports are configured as output.

**PROGRAMMING**

With all the great capabilities and features of PIC microcontrollers, their instruction set contains only few and very basic instructions. There are no **`MUL`** or **`DIV`** instructions for instance. Instructions are generally byte-oriented or bit-oriented register manipulators. Special functions and utilization of device peripherals is brought about by changing certain registers in the register file known as **Special Registers**. For example, to enable a built-in timer interrupt, one has to modify a “Timer Control Word” at a specific address of the register file. This requires the programmer to have good knowledge of the register file. Add to that the 8-bit architecture of the microcontroller which can make using it in high-level applications a tiring process.

For those reasons, and more, a need for a high-level language compiler is needed. Luckily, some PIC C Language compilers exist in the market. These compilers make it a lot easier to code and manage these little machines.

In this project, we have made big use of **Mikroelektronika’s MikroC Compiler**. Using this compiler, we have approached levels of high-programming close to those on our personal computers. Some of the remarkable capabilities that are featured by this compiler and greatly aided in the development of our project are:

- Complete C Language Syntax and Feature Support
- 8, 16 and 32-bit data structures
- Automatic Register File Management
- Extensive Libraries and built-in routines such as:
    - USART Library
    - LCD Library
    - Data Type Conversion Library

Using this compiler greatly aided us in developing our **Controller Firmware**; the software that the PIC microcontroller in our project is running.

See Chapter 6 for detailed information on the Firmware.

# CHAPTER 4: SERIAL COMMUNICATION

## INTRODUCTION

Serial Communication is the process of sending data bit by bit. It is an important theme in computing and communications. This method of communication allows data to be interchanged between devices through a limited number of communication channels (typically only one or two).

There are many standards and specifications for serial communications. One of the oldest and still most widely used is the **RS232** specification.

RS232 has been developed for communicating Data Communications Equipment (DCE) and Data Terminal Equipment (DTE), both being equal ends for data to be sent and received, only with different wire configurations.

In computers, RS232 is implemented in **Serial Ports**. Those 9-pin ports have been used since the early days of computer ports to transmit data serially, They are still built into computers and used until this day.

Table 4.1 shown below shows the pinout for a typical 9-pin serial port:

| Signal | Abbreviation | Direction | Pin Number |
|---|---|---|---|
| Common Ground | G | - | 5 |
| Transmitted Data | TxD | OUT | 3 |
| Received Data | RxD | IN | 2 |
| Data Terminal Ready | DTR | OUT | 4 |
| Data Set Ready | DSR | IN | 6 |
| Request to Send | RTS | OUT | 7 |
| Clear to Send | CTS | IN | 8 |
| Carrier Detect | DCD | IN | 1 |
| Ring Indicator | RI | IN | 9 |

Table 4.1: DB9 Serial Port Pinout

## FLOW CONTROL

Flow control is the mechanism by which the flow of data in a serial link is coordinated. Serial communication is defined by a data flow rate measured in **baud** (bits/sec). A serial link, thus, has a fixed number of bits flowing through as agreed upon by the two communicating nodes.

Sometimes, one or both of the communicating devices maybe unable to receive and process data at the communication speed; it can't "cope" with the communication link data rate. This happens mainly due to being engaged with some other activity, possibly processing, or managing other IO devices. In this case, a method of controlling the data flow rate is needed.

The flow control mechanism must provide a mean for one device to signal its status to the other. A busy device that can't process any further incoming data needs to quickly communicate its busy status to the other device to slow or even postpone transmission. Similarly, a ready device needs to acknowledge the other device whenever it is ready to receive data.

There are many ways by which this can be arranged, each has its own advantages and disadvantages. Generally, Flow Control mechanisms are classified as either **Software-based** or **Hardware-based**.

In RS232, Software-based flow control relies on the transmission of special characters such as XON and XOFF. Whenever a device is busy and can't receive any incoming data, it sends the XOFF character to instruct the other device to stop sending. Transmission can be resumed by sending the XON character.

This method is easy to implement but is actually inefficient due to the time it takes the special characters to be transmitted. By the time an overloaded receiving device finishes transmitting an XOFF character, it would probably be missing an incoming data character which will cause a communication error.

To address this, RS232 defines hardware-based flow control using extra lines to transmit ready and busy status flags between communicating devices. The signals DTR, DSR, RTS and CTS shown in Table 4.1 are used for this purpose.

Hardware flow control is much more reliable than software flow control. However, its obvious disadvantage is the need for more lines to transmit the extra info.

Hardware and software flow controls pose a “reliability vs. number of signal lines” tradeoff, it’s up to the user to decide which to use. In our project, we had to actually use a bit of both!

**Null Modem Connection**

Null modem connection is a typical connection used to “fool” systems that have built-in hardware flow control, when being connected with simple systems that do not have any kind of flow control mechanisms [2].

An example of this would be a personal computer communicating with a PIC microcontroller, the case in our project. This connection simply short-circuits the Request to Send (RTS) and Clear to Send (CTS) lines at one end of the link. This would loopback any “Transmission Request” signals to “Clear to Transmit” signals making the PC or other high-end devices believe that the other communication device cleared it to send data.

Null Modem connection is simply a method of overriding built-in hardware flow control. Unfortunately, it wasn’t that easy for our project. We had to develop our very own flow control mechanism, as to be explained in the following section.

**Implemented Flow Control**

Serial communication is used in this project to interface the PIC microcontroller to a computer running a GUI based application (Read more in Section 6.3).

During the development of serial communication for our controller, we’ve come across a serious flow control problem. Whenever a New Line character (Enter Key) is received at the microcontroller end, the microcontroller software (explained in Section 6.3) branches off the main loop and begins parsing and processing the command received in buffer. This takes a considerable long time and causes the microcontroller to miss the first few characters of the subsequent command, even at low baud rates such as 2400.

As a first attempt to solve the problem, we implemented hardware flow control through the Clear to Send (CTS) line. We programmed the microcontroller to pull the line low whenever it begins processing received data, so as to instruct the computer to pause transmission while the microcontroller finishes processing.

Unfortunately, this didn't work as well. After pulling the CTS line high and receiving a New Line character, the microcontroller was too slow to pull the line down again that it missed some characters of the next command as well.

Our solution to the problem was by implementing a "clocking signal" through the CTS line. The microcontroller will toggle the CTS line whenever it is ready to receive the next character. On the other end, the computer won't transmit next character until the value of the CTS line changes.

Unlike the first flow control attempt, this method will not result in the loss of any characters. This is simply because the microcontroller doesn't have to signal the start and end of flow; the "line toggle" signal itself implies **both** the start and the end of transmission. The microcontroller will thus not have to raise a "stop flow" flag which might come late and cause it to miss some transmitted characters.

Our implemented flow control mechanism, joining a bit of hardware and software flow control, resulted in a superior reliability connection. One big downside of using this method, however, is that it slows the transmission rate way below the baud rate.

Nevertheless, considering our application, the importance of a reliable connection exceeds that of fast data transmission rate. Communication with the controller doesn't require transmission of large amounts of data in the first place.

## COMMUNICATION PROTOCOL

### Introduction

For the purpose of communicating PCs or other devices with our controller via the serial port we defined a communication protocol. The controller will interchange **commands** and **messages** with the other terminal in a command prompt sequence. Commands are defined as the instructions sent to the controller, and messages are the status replies.

### Protocol Specifications

1. Transmission of commands and messages always ends with a Carriage Return and New Line characters, codes 0x0D and 0x0A respectively.
2. The controller will echo all characters received from the other end.
3. The controller will output a prompt sequence "**`>>`**" (without the quotes) whenever it is ready to accept a new command.
4. Communication is case sensitive.
5. The controller will respond to each received command immediately with either **`OK`** or **`Undefined command`**, indicating whether the received command was recognized and processed, or not, respectively.
6. The allowed characters in transmission are (a-z), (A-Z), (0-9), (+,-), Space and Enter Keys. All other characters are discarded by the controller.
7. Baud rate for communication is 9600, 8 data bits, no parity checking, 1 stop bit, and no flow control. These are Windows HyperTerminal's default settings.

**Protocol Commands**

The controller commands are divided into three groups:

1. Group I: Axis Configuration Commands

| | | |
|---|---|---|
| Command | : | **`axis1 on`** |
| Result | : | enables motion of stepper motor 1 |
| Command | : | **`axis1 off`** |
| Result | : | disable motion of stepper motor 1 |
| Command | : | **`axis1 +`** |
| Result | : | sets or changes the direction of motion of stepper motor 1 to forward |
| Command | : | **`axis1 –`** |
| Result | : | sets or changes the direction of motion of stepper motor 1 to backward |
| Command | : | **`axis1 steps %NUMBER%`** |
| Result | : | sets or changes the number of steps that stepper motor 1 is moving/going to move by. The numeric argument should be a positive integer number between 0 and 4,294,967,296. In case 0 is specified, the axis will keep operating till the end of the task. |
| Command | : | **`axis1 period %NUMBER%`** |
| Result | : | sets or changes the time (in ms) between each two successive axis steps. The numeric argument should be a positive integer number between 0 and 4,294,967,296 |

The same commands are available to both axis 2 and axis 3, using the **`axis2`** and **`axis3`** prefixes respectively.

2. Group II: Job Processing Commands

| | | |
|---|---|---|
| Command | : | **`go`** |
| Result | : | begins running each enabled axis as specified by the axis configuration commands |
| Command | : | **`stop`** |
| Result | : | halts the motion of all axis |

3. Group III: Miscellaneous Commands

| | | |
|---|---|---|
| Command | : | **`ping`** |
| Result | : | does not do any meaningful processing, this command causes the controller to respond by **`OK`** to signal a recognized command. This command is used to verify that the controller is connected and running. |
| Command | : | **`version`** |
| Result | : | returns the version of the controller’s Firmware |
| Command | : | **`screen 1`** |
| Result | : | switches LCD to display “Ready” |
| Command | : | **`screen 2`** |
| Result | : | switches LCD to display the status and direction of each of the three axis. |
| Command | : | **`communication test`** |
| Result | : | does not do any meaningful processing, the controller responds to this command by the **`Undefined command`** message once every three times. This command is used to test the fail-retry mechanism of the computer software. |

## Controller Task Execution Procedure

Typical operation of the controller involves defining the motion attributes of all axes, then issuing the `Go` command. All axes will begin execution simultaneously.

Each motion job configured by the axis configuration commands and started by the **`Go`** command is known as a **task**. Any task involves the motion of one or more stepper motors. The controller will send a **`Task completed`** message to acknowledge the computer of the completion of each task. The computer can then begin configuring for the next task. This workflow is illustrated in the flow chart in Figure 4.1.

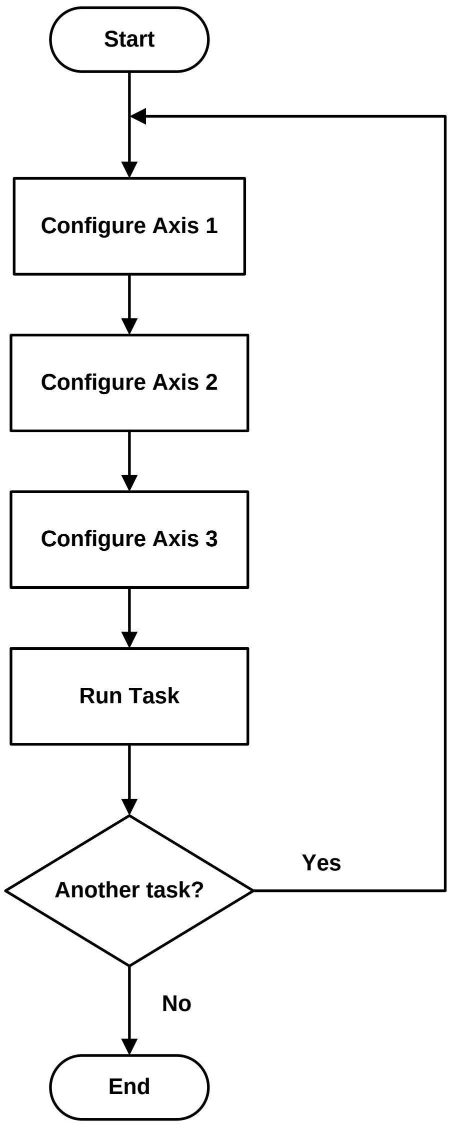


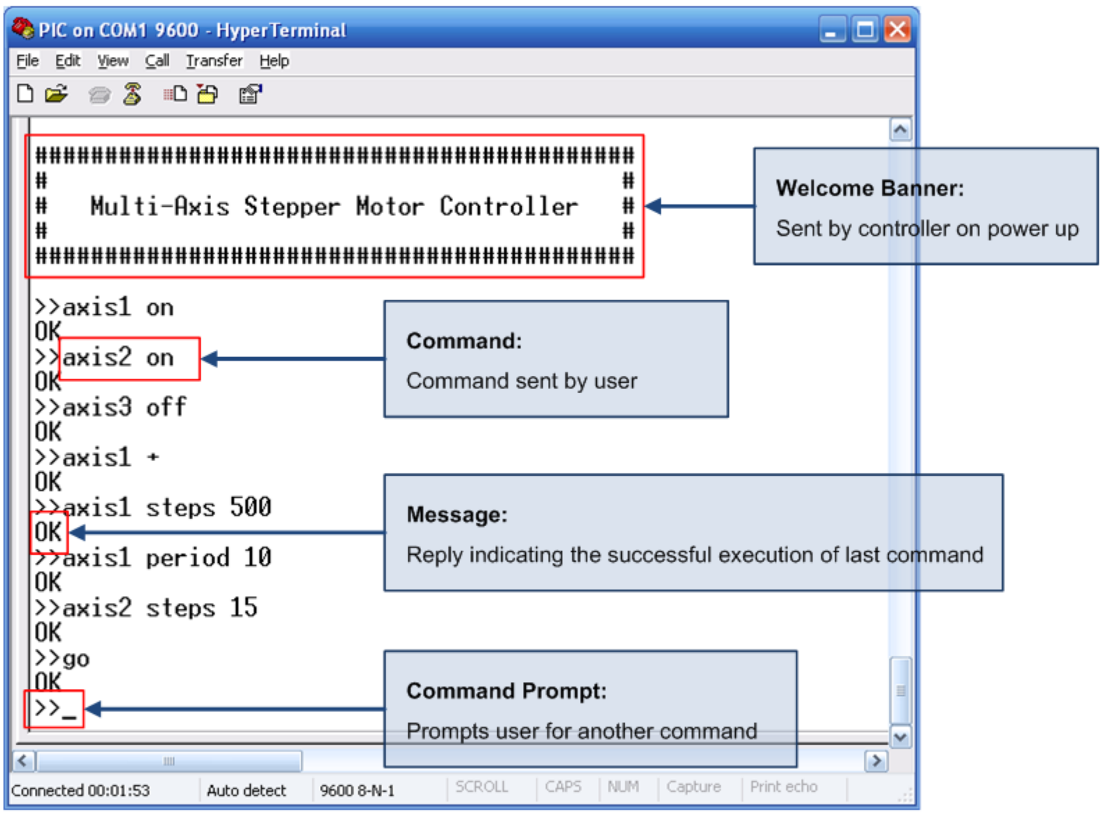
Figure 4.1: Typical Operation Flowchart

## Example Communication

To manually send commands to the controller, you can use any terminal program that can handle serial port communication such as Windows HyperTerminal.

You can send commands as specified by the protocol specifications, and according to the typical execution procedure illustrated in the previous section. Figure 4.2 shown below demonstrates an active HyperTerminal controller connection with commands executing a simple task.

While this is the underlying protocol for communication, please note that you can use the computer software illustrated in Chapter 6 to build an execution script consisting of multiple tasks using a Graphical User Interface (GUI). The computer software will undertake translating your script into the corresponding controller commands and performing the communication with controller on the back end.


PIC on COM1 9600 - HyperTerminal
File Edit View Call Transfer Help
Welcome Banner:
Sent by controller on power up
Command:
Command sent by user
Message:
Reply indicating the successful execution of last command
Command Prompt:
Prompts user for another command
Connected 00:01:53
Auto detect
9600 8-N-1
SCROLL
CAPS
NUM
Capture
Print echo

```
############################################
#                                          #
#    Multi-Axis Stepper Motor Controller   #
#                                          #
############################################
>>axis1 on
OK
>>axis2 on
OK
>>axis3 off
OK
>>axis1 +
OK
>>axis1 steps 500
OK
>>axis1 period 10
OK
>>axis2 steps 15
OK
>>go
OK
>>_
```

Figure 4.2: Example HyperTerminal Communication

# CHAPTER 5: CONTROLLER CIRCUIT

## CIRCUIT DIAGRAM

Figure 5.1: Complete Controller Circuit Diagram

## CIRCUIT COMPONENTS

### Liquid Crystal Display (LCD)

Liquid Crystal Display (LCD) screens are very common in electronic devices. LCDs provide an easy way to display alphanumeric figures and even simple graphics. LCDs extend from simple monochromatic screens in digital watches to large coloured computer screens [3].

The LCD screen implemented in this controller is used to display axes' status. It is a common 16 characters x 2 lines display model that is compatible with the Hitachi HD44780 protocol. It has a built-in controller.

The Hitachi HD44780 protocol defines means of communication between the LCD controller and the digital system interfacing the LCD. The LCD has 8 data pins (D0-D7) and 3 control pins: Enable (E), Register Select (RS), and Read/Write (R/W). To send data to the LCD, a byte value is set on pins D0-D7 then a command is initiated by changing the values of the control pins.

Hitachi HD44780 compatible LCDs have two types of memory: Display Data RAM (DDRAM) and Character Generator RAM (CGRAM). DDRAM holds standard character display data for letters, numbers and common symbols. CGRAM can be written to by user commands to make the LCD display custom characters. This feature is utilized in this project to create some custom characters that are displayed in "screen 2".

Mikroelektronika's MikroC has an LCD library that enables easy utilization of Hitachi HD44780 compatible LCDs.

It should be noted that the LCD used in this project takes sometime to "boot up". Thus, the controller was programmed to delay LCD interfacing by 500 ms to give the LCD enough time to initialize.

There are two modes or "screens" for LCD display in this controller. They are switched to through the commands **`screen 1`** and **`screen 2`**.

Screen 1 is merely a display of the word "Ready". It is switched to automatically by the Computer Software application whenever a script finishes executing. It is shown in Figure 5.2.

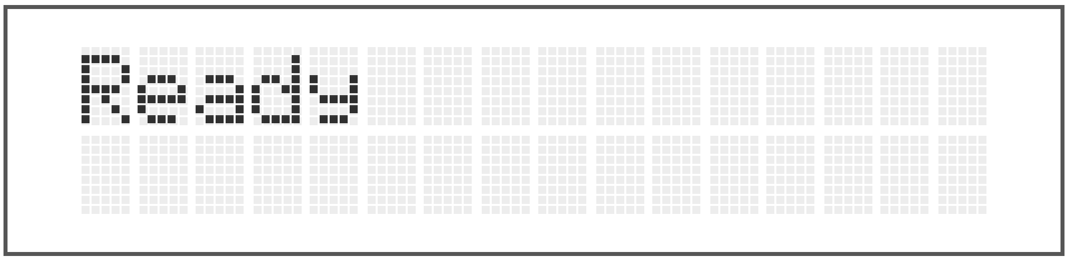


Figure 5.2: Controller's LCD Screen 1

Screen 2 presents more significant data. It displays the status of the three axes of the controller. Below each axis number are two symbols. The rectangle indicates whether the axis is enabled or not; a filled rectangle means that the axis motion is enabled. The arrow represents the direction of motion of the axis; an arrow pointing upwards means that the axis' motion is in forward direction.

Screen 2 is switched to automatically by the Computer Software application whenever a script starts executing. It is also updated instantly by the controller whenever an axis configuration command is processed. Screen 2 is shown in Figure 5.3 below.

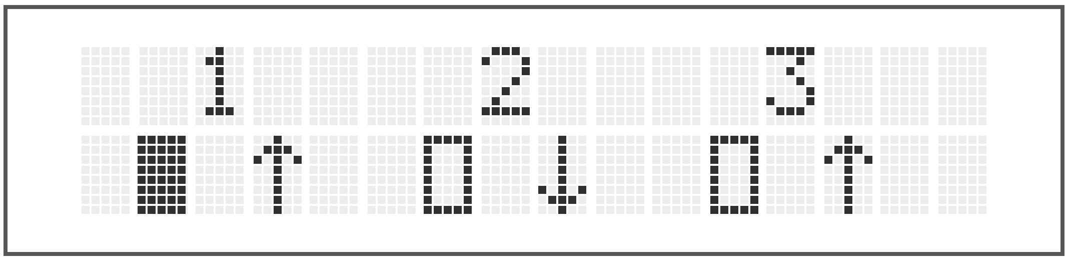


Figure 5.3: Controller's LCD Screen 2

**MAX232**

The RS232 protocol defines transmission voltage levels as (-15V ~ -3V) for high logic level and (+3V ~ +15V) for low logic level. TTL defines voltage levels as (+2V ~ +5V) for high logic level and (0V ~ +0.8V) for low level logic [1].

Therefore, it is necessary to convert a signal's voltage levels before transmitting it through or receiving it from an RS232 interface. MAX232 is a chip that does the job using only four capacitors and a single +5V supply voltage.

| Channel | Pins | | Type | Use in Controller |
|---|---|---|---|---|
| | In | Out | | |
| 1 | T1IN (11) | T1OUT(14) | Driver | CTS |
| 2 | T2IN(10) | T2OUT(7) | Driver | TxD |
| 3 | R1IN(13) | R1OUT(12) | Receiver | - |
| 4 | R2IN(8) | R2OUT(9) | Receiver | RxD |

Table 5.1: MAX232 Channels

Table 5.1 shows the four channels of a MAX232 chip. MAX232 contains two (TTL to RS232) converters, known as Drivers. It also contains two (RS232 to TTL) converters, known as receivers.

Table 5.1 also shows the use of each channel in the controller circuit. Both drivers were used to transform the outgoing TxD and Clear To Send (CTS) signals to RS232 voltage levels. Receiver2 channel was used to transform the incoming RxD signal to TTL voltage levels. Receiver1 was unused.

For a complete wiring diagram, refer to Figure 5.1 (Chapter 5).

## M325 Microstepping Driver

The M325 stepper motor driver is a driving circuit that can drive any 2-phase or 4-phase stepper motor [6]. It is powered by a (12V ~ 24V) supply, and can provide currents up to 2.5 A [6].

The driver has three logic inputs, all are opto-coupled to increase interface flexibility and separate input signals from internal driving circuits.

| Pin | Function |
|---|---|
| PUL | Pulse Signal. Each rising or falling edge (configurable via jumpers) drives the connected stepper motor one step. Supports up to 100Khz pulse rate. |
| DIR | Direction Signal, it has two voltage levels (0 or 5V). This signal determines the direction of rotation of the stepper motor. |
| ENA | Enable signal, it has two voltage levels (0 or 5V). This signal enables/disables the driver. |
| OPTO | Supply voltage for opto-couplers. Typically its 5V. |

Table 5.2: M325 Driving Signals

Table 5.2 shows the input signals that control the driver.

The driver's outputs are four lead connectors that can be connected to any 4-lead, 6-lead or 8-lead stepper motors using either direct or special wiring configurations.

For a complete circuit wiring diagram, refer to Figure 5.1 (Chapter 5).