Stepper Motors Uncovered (2)

Part 2 (final): a universal 4-channel unipolar stepper drive

Design by Timothy G. Constandinou

Having covered the fundamentals to stepping motors and drive systems, this second and final part provides a comprehensive design to a four-channel unipolar stepper drive with complete interface electronics for direct operation from a standard PC.

This second part of the article includes full details to build, test and use a low-cost 4-channel stepper motor drive which can be tailored to your applications. The project includes the RS-232 interface for direct connection to the PC, a custom high-level control language for executing commands sent to the controller and the drive electronics to power the motors. In addition, the PC communication software will be explained in some detail allowing full customisation to your specific requirements. This software is compatible with all Microsoft Windows 32-bit platforms and was developed in Borland Delphi.

The RS232 serial interface

The RS232 serial interface standard, defined over four decades ago, has remained a favourite for low bandwidth communications using the personal computer. Since virtually all PCs are shipped with at least one RS232 port, and many of today's microcontrollers have, or are easily extended with, RS232 interface circuitry, the RS232 port is an affordable as well as straight-forward option for home-built projects.

Normally recognised as a 9-pin sub-D plug labelled COM1 or COM2, the RS232 serial port has nine connections. Half duplex two-way communication can be achieved by using only three pins (2, 3 and 5). The complete pin-out of the port is shown in **Figure 1**. Unlike the standard TTL levels, RS232 data is bipolar, using +3 to +25 V to represent a logic '0' and –3 to –25 V to represent a logic '1'. This scheme makes relatively

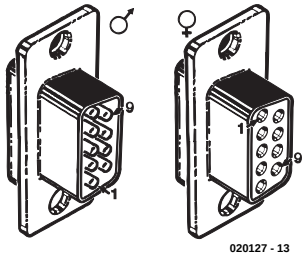

Figure 1 RS232 port pinning.

Pin	Signal
1	Data Carrier Detect *(DCD)*
2	Received Data *(RxD)*
3	Transmitted Data *(TxD)*
4	Data Terminal Ready *(DTR)*
5	Signal Ground *(SG)*
6	Data Set Ready *(DSR)*
7	Request to Send *(RTS)*
8	Clear to Send *(CTS)*
9	Ring Indicator *(RI)*

long-distance communication possible — however, additional interface electronics are required to convert RS232 voltage levels to/from TTL.

At the computer end, communicating with just about any hardware port is much the same as handling a data file on disk. Each port has a unique address. For access it is opened, the required data is transferred and then it is closed. The only additional requirement is that the port properties be set up ('configured') before usage; for example, data bitrate, parity bit and data timeout.

Hardware

Figure 2 shows the circuit diagram of the stepper drive and interface. This is quite straightforward. Starting from the RS232 input (K1) the transmit (Tx) and receive (Rx) lines are connected to a level converter chip (IC2). As previously mentioned, this has the task of converting the RS232 bipolar voltage levels — for example, a swing of –9V/+9V to TTL swing (defined as +5V/0V). Note that

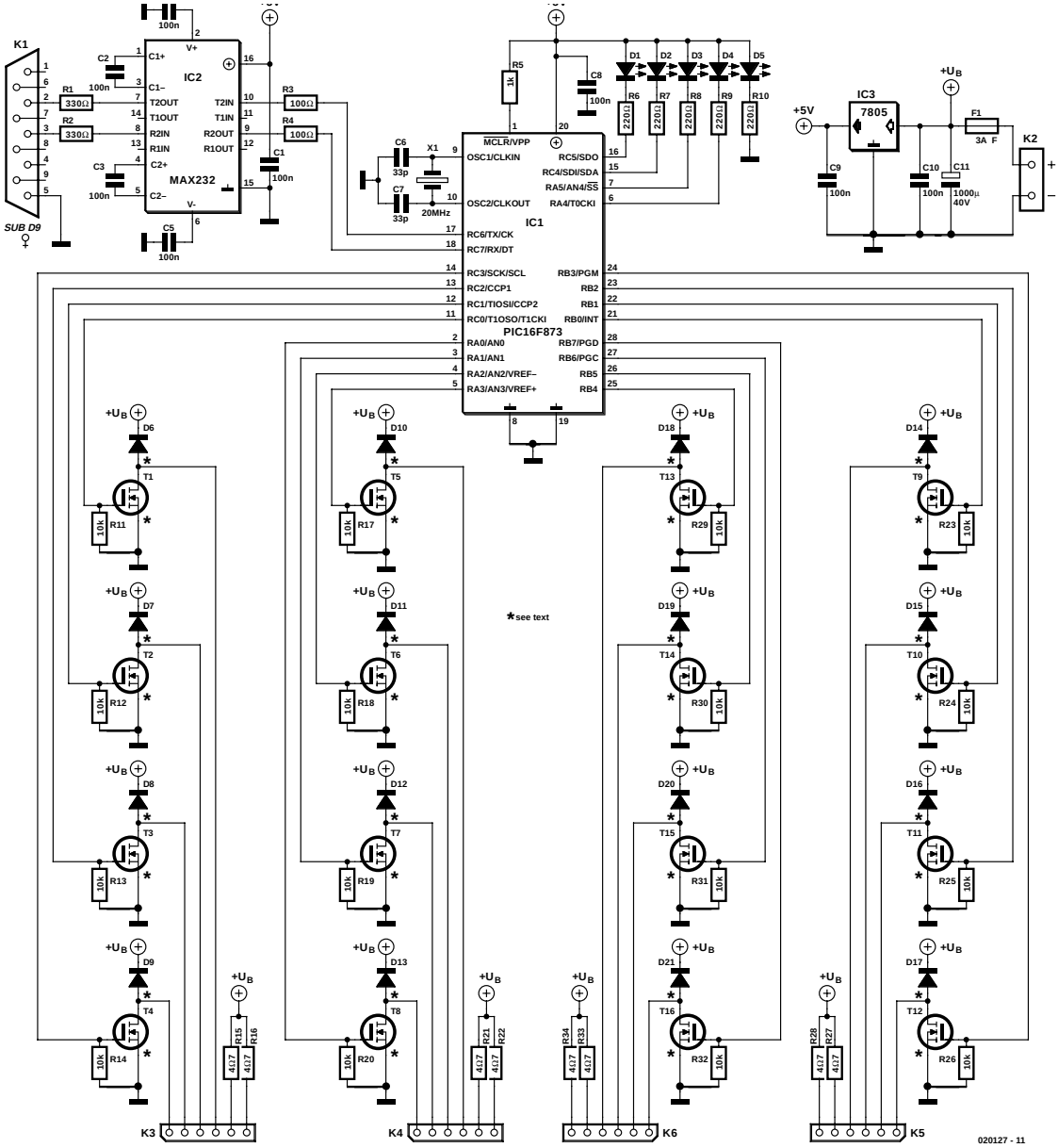

Figure 2. Circuit diagram of the driver board.

this is internally done by using a switched capacitor technique to create a higher double-ended supply (±9V).

The TTL level-converted signals are then connected to the UART (Universal Asynchronous Receive and Transmit) pins of the PIC microcontroller (IC1). The RS232 I/O pins have been connected through series resistors R1 and R2 and similarly, on the converted side, R3 and R4, primarily for protection, in case something goes wrong!

The linear regulator (IC3) is required to provide the +5 V regulated supply to the PIC MCU and

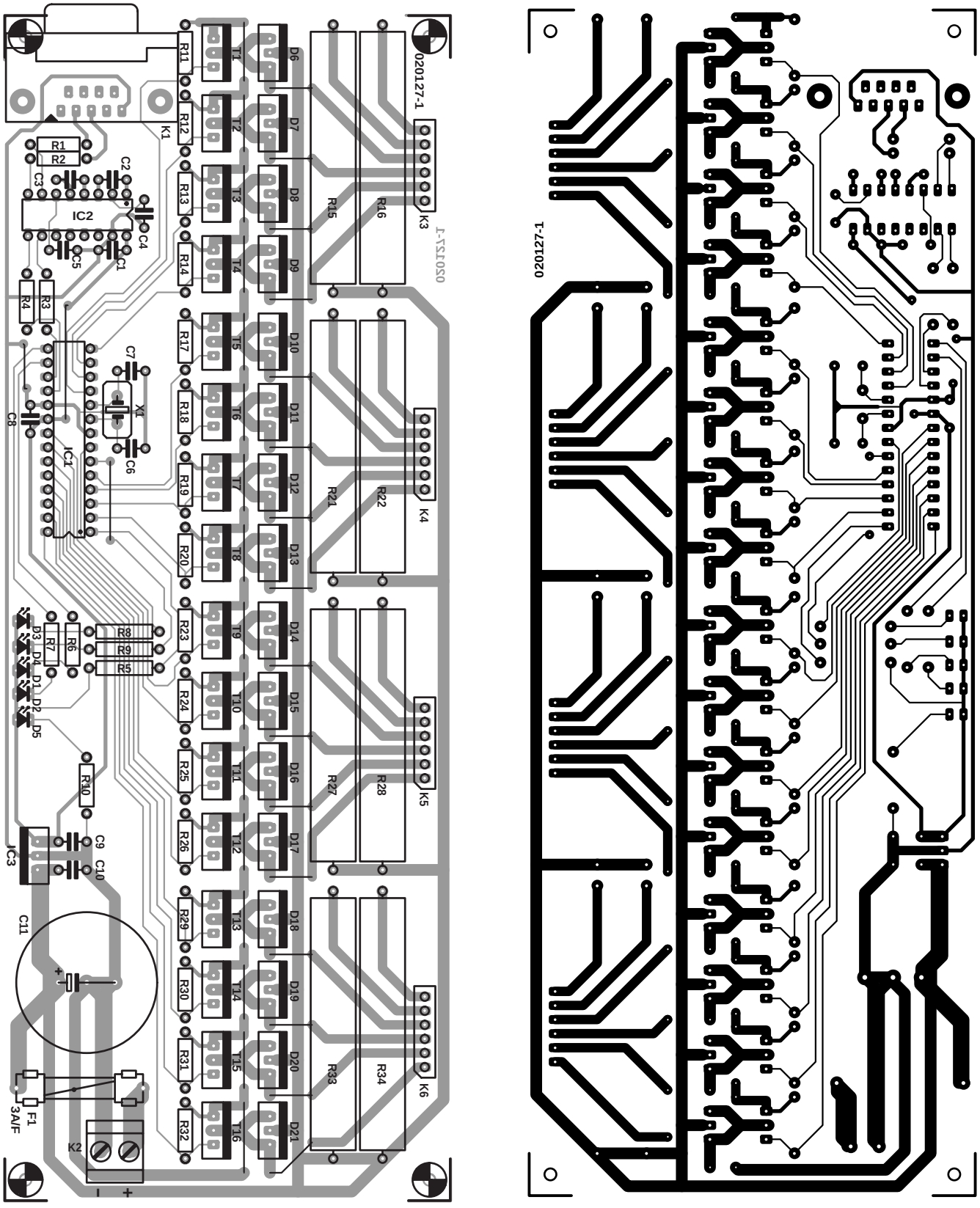

Figure 3. PCB design for the stepper motor driver board.

RS232 interface chips. IC1 employs capacitors C6, C7 and quartz crystal X1 in conjunction with an internal bistable to form a precision 20 MHz oscillator required by the UART. Pin 1 of the PIC is pulled high through R1, as resetting the microcontroller is not required. All remaining I/O ports (20 pins) are configured as outputs and connected to the stepper motor phase drives and LED indicators.

The stepper motor drive scheme used is a resistance-limited unipolar drive, suitable for 5, 6 and 8-wire low-power stepper motors. This provides a low-cost and simple means to powering a unipolar winding. However, it does suffer from inefficiency due to power dissipated in the ballast resistors.

The phase drive circuit uses logic-level MOSFET devices driven directly from the microcontroller output to power the stepper motor windings. Various Logic Level FETs may be applied here, see the **inset**. Fast recovery diodes are required to provide a return path for the energy stored in the motor windings and to prevent damage to the MOSFET devices owing to back-EMF discharges. Again, a number of devices are available to choose from, see the relevant inset. The ballast resistors are used to limit the current through the phase winding, but inevitably will dissipate power. The resistor values shown should be calculated for the specific stepper motor used. It is essential to have the manufacturer's data on the specific stepper motor including data on the winding impedance, as well as nominal current and voltage ratings. If you do not have this available it is not advisable to obtain just the resistance using a multimeter as no data will be available on the motors' real power ratings. **Table 1** gives an example of selecting ballast resistor value and rating for two different stepper motors for different supply voltages.

These values can be calculated as follows:

$$R_{ballast} = V_{supply} / (I_{motor} - R_{motor})$$

$$P_{ballast} = 0.5\ (I^2_{motor} \times R_{ballast})$$

Some points to note: because the motor is driven in full-step mode, the windings are only powered half the time, therefore the power rating for the relevant ballast resistor may be only half the energy dissipation normally expected. The voltage supply should be chosen to lie between 10 V and 30 V — the higher the supply, the more power delivered to the motor. This should be higher than the voltage rating of the motor — don't forget there is a voltage drop across the ballast resistor. Also, please note that the maximum current rating (per winding) that can be driven using this PCB should not exceed 1 A.

Table 1. Ballast resistor values (examples)

V_{supply} (Volts)	I_{motor} (Amps)	R_{motor} (Ohms)	$R_{ballast}$ (Ohms)	$P_{ballast}$ (Watt)
15	1.00	5	10	5.0
20			15	7.5
25			20	10.0
30			25	12.5
15	0.500	15	15	1.9
20			25	3.1
25			35	4.4
30			45	5.6

Construction

All components in this circuit are assembled directly onto a PCB, whose copper track layout and component mounting plan are given in **Figure 3**. Sockets should be fitted for the two ICs with a DIL (dual-in-line) footprint, while IC3 should be soldered directly onto the PCB. It is advisable to firstly assemble the lower profile components such as links, resistors,

COMPONENTS LIST

Resistors:
R1,R2 = 330Ω
R3,R4 = 100Ω
R5 = 1kΩ
R6-R10 = 220Ω
R11-R14,R17-R20,R23-R26,R29-R32 = 10kΩ
R15,R16,R21,R22,R27,R28,R33,R34 = 18Ω 5 watt (see text)

Capacitors:
C1-C5,C8,C9,C10 = 100nF
C6,C7 = 33pF
C11 = 1000µF 40V radial

Semiconductors:
D1-D4 = LED, green, 3mm
D5 = LED, red, 3mm
D6-D21 = MBR2060CT (Farnell # 247-157) (see inset)
IC1 = PIC16F873-20/SP (not available ready-programmed)
IC2 = MAX232CPE
IC3 = 7805CP
T1-T16 = Logic-level MOSFET, for example, RFD14N05L (Farnell # 516-399) (see inset)

Miscellaneous:
F1 = fuse, 3AF (fast) with PCB mount holder
K1 = 9-way sub-D socket (female), PCB mount
K2 = 2-way PCB terminal block, 5mm lead pitch
K3-K6 = 6-way SIL pinheader
X1 = 20MHz quartz crystal
PCB, order code 020127-1 from The PCBShop
Disk, contains all source code files, order code **020127-11** or Free Download

Logic Level FETs and Fast Recovery Diodes

In this circuit, the choice of logic level FET (positions T1-T16) and fast recovery diodes (positions D6-D21) will be governed by availability and the power rating of the stepper motor(s) used.

FETs				
Type	**I_{max} (A)**	**U_{max} (V)**	**R_i (mΩ)**	**Note**
RFD14N05L	14	50	100	Farnell # 515-399, Fairchild
BUK100-50GL	13.5	50	125	
BUK101-50GS	30	50	50	
IRLI2203N	61	30	7	
Diodes				
Type	**I_{max} (A)**	**U_{max} (V)**		
MBR1045CT	10	45		Farnell # 878-364
MBR1545CT	15	45		Farnell # 878-194
etc.				

Listing I. Firmware source code.

```
// main.c – Main program code

#include <16f873.h>
#include <ports.h>
#include <protocol.h>
#use delay (clock=20000000)
#use rs232(baud=38400, xmit=tx, rcv=rc)

int astep=1, bstep=1, cstep=1, dstep=1;
long max=800, min=470;

// initialises the ports by defining whether the tri-state buffers should be input or output
void setup_ports(void) { set_tris_a(0x00);set_tris_b(0x00);set_tris_c(0xF0);set_uart_speed(38400); }

// resets one motor to initial state
void reset_motor(int motor) {
    if (motor==1) {output_low(a_1);output_low(a_2);output_low(a_3);output_low(a_4);output_high(led_a);}
    if (motor==2) {output_low(b_1);output_low(b_2);output_low(b_3);output_low(b_4);output_high(led_b);}
    if (motor==3) {output_low(c_1);output_low(c_2);output_low(c_3);output_low(c_4);output_high(led_c);}
    if (motor==4) {output_low(d_1);output_low(d_2);output_low(d_3);output_low(d_4);output_high(led_d);} }

// resets all ports to initial states
void reset_ports(void) { reset_motor(1);reset_motor(2);reset_motor(3);reset_motor(4);putc(ACKNOWLEDGE); }

// creates a delay which constitutes the step pulse duration
void delay_micro(long delay) { long n;for(n=1;n<=delay;n+=3)delay_us(6); }

// changes powered phases according to current step required
void power_motor(int axis, step) {
    if (axis==1) {
        if (step==1) {output_bit(a_1,1);output_bit(a_2,0);output_bit(a_3,0);output_bit(a_4,1);}
        if (step==2) {output_bit(a_1,0);output_bit(a_2,1);output_bit(a_3,0);output_bit(a_4,1);}
        if (step==3) {output_bit(a_1,0);output_bit(a_2,1);output_bit(a_3,1);output_bit(a_4,0);}
        if (step==4) {output_bit(a_1,1);output_bit(a_2,0);output_bit(a_3,1);output_bit(a_4,0);}
        output_low(led_a); }
    if (axis==2) {
        if (step==1) {output_bit(b_1,1);output_bit(b_2,0);output_bit(b_3,0);output_bit(b_4,1);}
        if (step==2) {output_bit(b_1,0);output_bit(b_2,1);output_bit(b_3,0);output_bit(b_4,1);}
        if (step==3) {output_bit(b_1,0);output_bit(b_2,1);output_bit(b_3,1);output_bit(b_4,0);}
        if (step==4) {output_bit(b_1,1);output_bit(b_2,0);output_bit(b_3,1);output_bit(b_4,0);}
        output_low(led_b); }
    if (axis==3) {
        if (step==1) {output_bit(c_1,1);output_bit(c_2,0);output_bit(c_3,0);output_bit(c_4,1);}
        if (step==2) {output_bit(c_1,0);output_bit(c_2,1);output_bit(c_3,0);output_bit(c_4,1);}
        if (step==3) {output_bit(c_1,0);output_bit(c_2,1);output_bit(c_3,1);output_bit(c_4,0);}
        if (step==4) {output_bit(c_1,1);output_bit(c_2,0);output_bit(c_3,1);output_bit(c_4,0);}
        output_low(led_c); }
    if (axis==4) {
        if (step==1) {output_bit(d_1,1);output_bit(d_2,0);output_bit(d_3,0);output_bit(d_4,1);}
        if (step==2) {output_bit(d_1,0);output_bit(d_2,1);output_bit(d_3,0);output_bit(d_4,1);}
        if (step==3) {output_bit(d_1,0);output_bit(d_2,1);output_bit(d_3,1);output_bit(d_4,0);}
        if (step==4) {output_bit(d_1,1);output_bit(d_2,0);output_bit(d_3,1);output_bit(d_4,0);}
        output_low(led_d); } }

// Moves a specified motor by a specified amount of steps in a specified direction.
int move(short direction, long steps, int axis, step) {
    long n, delay, accsteps;
    delay=max; accsteps=max-min;
    for(n=1;n<=steps;n++) {
        if(n<=accsteps)delay–;
        if(steps-n<=accsteps)delay++;
        if(direction==0)step–;else step++;
        if(step==0)step=4;
```

```
        if(step==5)step=1;
        power_motor(axis, step); delay_micro(delay); reset_motor(axis); } return(step); }

// Reads in 2 bytes from the UART and returns a 16-bit integer (range 0-65535)
long readlong(void) { return(256*getc() + getc()); }

// Main Program
void main(void) {
    char incomm;
    long steps;
    setup_ports(); reset_ports();
    while(0==0) {
        output_low(led_a); output_low(led_b); output_low(led_c); output_low(led_d);
        incomm=getc();
        output_high(led_a); output_high(led_b); output_high(led_c); output_high(led_d);
        switch(incomm) {
            case RESET:     reset_ports();                                    break;
            case SETUP_ACC: min=readlong(); max=readlong();                   break;
            case MOVE_A_FW: steps=readlong(); astep=move(0, steps, 1, astep); break;
            case MOVE_A_RV: steps=readlong(); astep=move(1, steps, 1, astep); break;
            case MOVE_B_FW: steps=readlong(); bstep=move(0, steps, 2, bstep); break;
            case MOVE_B_RV: steps=readlong(); bstep=move(1, steps, 2, bstep); break;
            case MOVE_C_FW: steps=readlong(); cstep=move(0, steps, 3, cstep); break;
            case MOVE_C_RV: steps=readlong(); cstep=move(1, steps, 3, cstep); break;
            case MOVE_D_FW: steps=readlong(); dstep=move(0, steps, 4, dstep); break;
            case MOVE_D_RV: steps=readlong(); dstep=move(1, steps, 4, dstep); break; } putc(ACKNOWLEDGE); } }
```

// ports.h – defines pin assignments

```
#define tx    PIN_C6
#define rc    PIN_C7
#define a_1   PIN_C3
#define a_2   PIN_C2
#define a_3   PIN_C1
#define a_4   PIN_C0
#define b_1   PIN_A0
#define b_2   PIN_A1
#define b_3   PIN_A2
#define b_4   PIN_A3
#define c_1   PIN_B3
#define c_2   PIN_B2
#define c_3   PIN_B1
#define c_4   PIN_B0
#define d_1   PIN_B7
#define d_2   PIN_B6
#define d_3   PIN_B5
#define d_4   PIN_B4
#define led_a PIN_A5
#define led_b PIN_A4
#define led_c PIN_C5
#define led_d PIN_C4
```

// protocol.h – defines communication protocol

```
#define RESET       1
#define ACKNOWLEDGE 2
#define SETUP_ACC   10
#define MOVE_A_FW   20
#define MOVE_A_RV   21
#define MOVE_B_FW   22
#define MOVE_B_RV   23
#define MOVE_C_FW   24
#define MOVE_C_RV   25
#define MOVE_D_FW   26
#define MOVE_D_RV   27
```

Listing 2. Test program to run on the PC.

```
unit main;

interface

uses Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls, QCCom32, Buttons, ExtCtrls;

type
  TForm1 = class(TForm)
    QCPort: T_QCCom32;
    Commport: TComboBox;
    xclgroup: TRadioGroup;
    setup_acc, move_a_rv, move_a_fw, move_b_rv,     move_b_fw, move_c_rv, move_c_fw, move_d_rv, move_d_fw, reset: TRa-
dioButton;
    parameter1, parameter2: TEdit;
    commportlabel, parameterlabel: TLabel;
    Executebutton: TBitBtn;
    autoreset: TCheckBox;
    procedure CommportChange(Sender: TObject);
    procedure ExecutebuttonClick(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure setup_accClick(Sender: TObject);
    procedure move_a_fwClick(Sender: TObject);
    procedure move_a_rvClick(Sender: TObject);
    procedure move_b_fwClick(Sender: TObject);
    procedure move_b_rvClick(Sender: TObject);
    procedure move_c_fwClick(Sender: TObject);
    procedure move_c_rvClick(Sender: TObject);
    procedure resetClick(Sender: TObject);
    procedure move_d_fwClick(Sender: TObject);
    procedure move_d_rvClick(Sender: TObject);
  private { Private declarations }
  public  { Public declarations }
  end;

var Form1: TForm1;

Implementation {$R *.DFM}

  procedure TForm1.resetClick(Sender: TObject); begin parameter1.Enabled := FALSE; parameter2.Enabled := FALSE; end;

  procedure TForm1.setup_accClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := TRUE; end;

  procedure TForm1.move_a_fwClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_a_rvClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_b_fwClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_b_rvClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_c_fwClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_c_rvClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_d_fwClick(Sender: TObject); begin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.move_d_rvClick(Sender: TObject); egin parameter1.Enabled := TRUE; parameter2.Enabled := FALSE; end;

  procedure TForm1.CommportChange(Sender: TObject);
    begin QCPort.Port := Commport.ItemIndex + 1; end;

  procedure TForm1.FormShow(Sender: TObject); begin QCPort.Port := 1; CommPort.ItemIndex := 0; end;
```

```
  procedure TForm1.ExecutebuttonClick(Sender: TObject);
    var
      commandcode : char;
      command     : string;
    begin
      Executebutton.Enabled := FALSE;

      if reset.Checked     then commandcode := char(1);
      if setup_acc.Checked then commandcode := char(10);

      if move_a_fw.Checked then commandcode := char(20);
      if move_a_rv.Checked then commandcode := char(21);
      if move_b_fw.Checked then commandcode := char(22);
      if move_b_rv.Checked then commandcode := char(23);
      if move_c_fw.Checked then commandcode := char(24);
      if move_c_rv.Checked then commandcode := char(25);
      if move_d_fw.Checked then commandcode := char(26);
      if move_d_rv.Checked then commandcode := char(27);

      QCPort.Open; setlength(command, 1);
command[1] := commandcode; QCPort.Write(command);

      if (parameter1.enabled) then
        begin setlength(command, 2);
          command[1] := char(strtoint(parameter1.text) div 256);
          command[2] := char(strtoint(parameter1.text) mod 256);
          QCPort.Write(command); end;

      if (parameter2.enabled) then begin
          setlength(command, 2);
          command[1] := char(strtoint(parameter2.text) div 256);
          command[2] := char(strtoint(parameter2.text) mod 256);
          QCPort.Write(command); end;

      while(QCPort.Read = '') do;

      if autoreset.Checked then begin
          setlength(command, 1);
          command[1] := char(1);
          QCPort.Write(command);
          while(QCPort.Read = '') do; end;

      QCPort.Close; Executebutton.Enabled := TRUE;
    end;
end.
```

DIL sockets, ceramic capacitors, etc., mainly for convenience. Take special care to observe the correct polarity of all semiconductors and electrolytic capacitors before soldering. Also, the ballast resistors should be mounted slightly off the board surface as they will become hot during operation. It is advisable to use ceramic standoffs to space these resistors above the board.

If all four channels are not required, you may populate, for example, only two of the four channels of the stepper motor drivers.

When the soldering is finished, the PIC microcontroller and MAX232 ICs may be installed in their DIL sockets. You can program your own PIC for the project using the source code available under number **020127-11** on disk or from the Free Downloads section of our website at www.elektor-electronics.co.uk. For the more ambitious readers wanting to customize the PIC firmware or add functionality, a full overview including some guidelines is provided in the following section. It is advisable to test the project with the original firmware before attempting to modify it.

The controller software

The PIC microcontroller's function is to receive commands from the PC via the RS232 port and execute them. It is responsible for generating the stepping sequence which will control the power delivered to the motor. This also produces the acceleration and deceleration cycles for optimal stepping response of a given motor. By having this low level interface the pulse timings can be guaranteed to be precise.

So why bother having a microcontroller at all? Why not control the stepper motor drive directly from the computer? Although such real-time control was possible in the past with DOS-based programs, unfortunately this is no longer the case. This is because of the

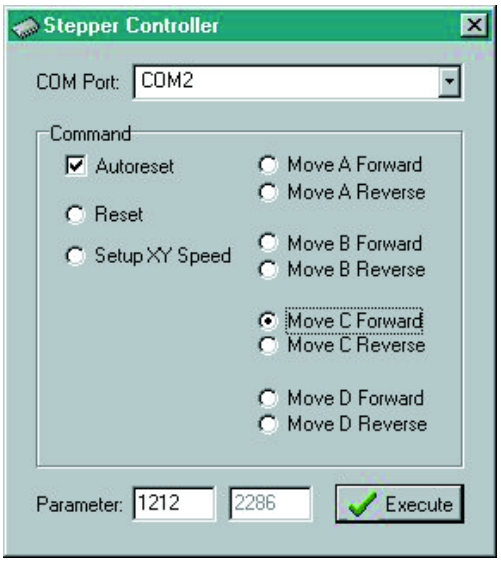

Figure 4. The stepper motor 'command' program in action on the PC.

multi-tasking and multi-threaded nature of recent 32-bit Windows operating systems, time-slicing the processor usage thus precluding stable and accurate timings.

The firmware for the project was programmed using an affordable third-party C compiler supplied by CCS, which is fully compatible with the Microchip MPLAB environment. For more details on this compiler, a full language reference is available online on the CCS website.

The code is divided into three files: main.c, protocol.h and ports.h. The main program is within main.c, with the pin assignments (to variable names) defined in ports.h and the custom communication protocol defined in protocol.h. This firmware source code is given in **Listing 1**.

The custom communication protocol used in this project is very simple. For every command a one-byte value is transmitted and if the command requires additional parameters these are sent in succession. For example, to tell the controller to move the specific motor in one direction for 1000 steps, three bytes are required, the first defining the command and the other two bytes specifying the number of steps (within the range: 0 to 65535). Depending on the initial command byte, the total length of transmission for that command is defined. After executing the command the microcontroller will reply with an acknowledge byte to notify the PC software that it is free to receive more commands if required.

The main program module firstly initialises and resets all the I/O ports including the UART with the bitrate set to 38,400 bits/s. The program then comprises an endless loop awaiting a single byte to be received on the UART. On receiving the command byte, program control is given to the appropriate command section, which may receive further bytes on the UART.

The available commands are listed below:

RESET (byte 1): resets all I/O ports.

SETUP_ACC (byte 10): Followed by an additional four bytes to set the minimum and maximum step delays for the stepper motor motion (both are 16-bit integers). On executing a MOVE command the step delays will initially be at maximum, reducing gradually in duration until the minimum delay has been reached. Further steps will have this minimum delay. Towards the end of the command cycle the step delays will increase until the maximum is again reached. This action implements the acceleration and deceleration in every MOVE command.

MOVE_A_FW (byte 20): Followed by an additional two bytes (one 16-bit integer) to specify how many steps motor A will move in the forward direction.

MOVE_A_RV (byte 21): Followed by an additional two bytes (one 16-bit integer) to specify how many steps motor A will move in the reverse direction.

MOVE_B_FW (byte 22)
MOVE_B_RV (byte 23)
MOVE_C_FW (byte 24)
MOVE_C_RV (byte 25)
MOVE_D_FW (byte 26)
MOVE_D_RV (byte 27)
These are as for **MOVE_A_FW** and **MOVE_A_RV** but for motors B, C and D respectively.

When programming your own PIC microcontroller, don't forget to turn off the DEBUG_MODE feature. Ensure POWER_ON_RESET is enabled and disable the WATCHDOG_TIMER and BROWN_OUT_DETECT features. Also ensure the clock speed is set to 20 MHz.

Recommended programmers and development kits for the microcontroller used here include the Elektor Electronics **PICProg 2003** (September 2003) and the Microchip PICSTART and ICD module (requiring an additional 28-pin header). Alternatively, Taylec Ltd. provide a very affordable equivalent to the ICD module (for under £50), fully compatible with the Microchip software, available for free download.

The PC software

The PC software was programmed in Borland Delphi 4. A freeware (VCL) Visual Component Library was used in order to access the serial port called QCCOM32.

Included in **Listing 2** is a test program to illustrate how commands are sent through the RS232 port to the stepper motor controller. This is again available in the Free Downloads section on our website at www.elektor-electronics.co.uk.

It is important to ensure the QCCOM32 properties are set to exactly match the initialisation of the UART in the firmware, especially bitrate=38400. For each command to be sent to the controller, the port is opened, the required bytes are transmitted, then the program waits until it receives the acknowledge signal and finally the port is closed.

Test and practical use

Before powering up, it is important to check all components are correctly placed and that the soldering is clean. Unplug all the stepper motors and power up. First, use an ammeter to check the current drawn from the power supply. Next, use a voltmeter to see if the supply rails are correct. If anything appears wrong at this stage, immediately power off and check the PCB and connections.

All five LEDs should light up when the circuit powers up properly. If this is the case, the microcontroller is up and running. However, if only one LED lights up, then there is power to the circuit but the microcontroller firmware is not being executed correctly. Assuming the microcontroller has been programmed successfully you should then check it receives supply voltage on the relevant pins. If all is in order then you should check the oscillator components (X1, C6 and C7) to ensure these are fitted correctly. If still no

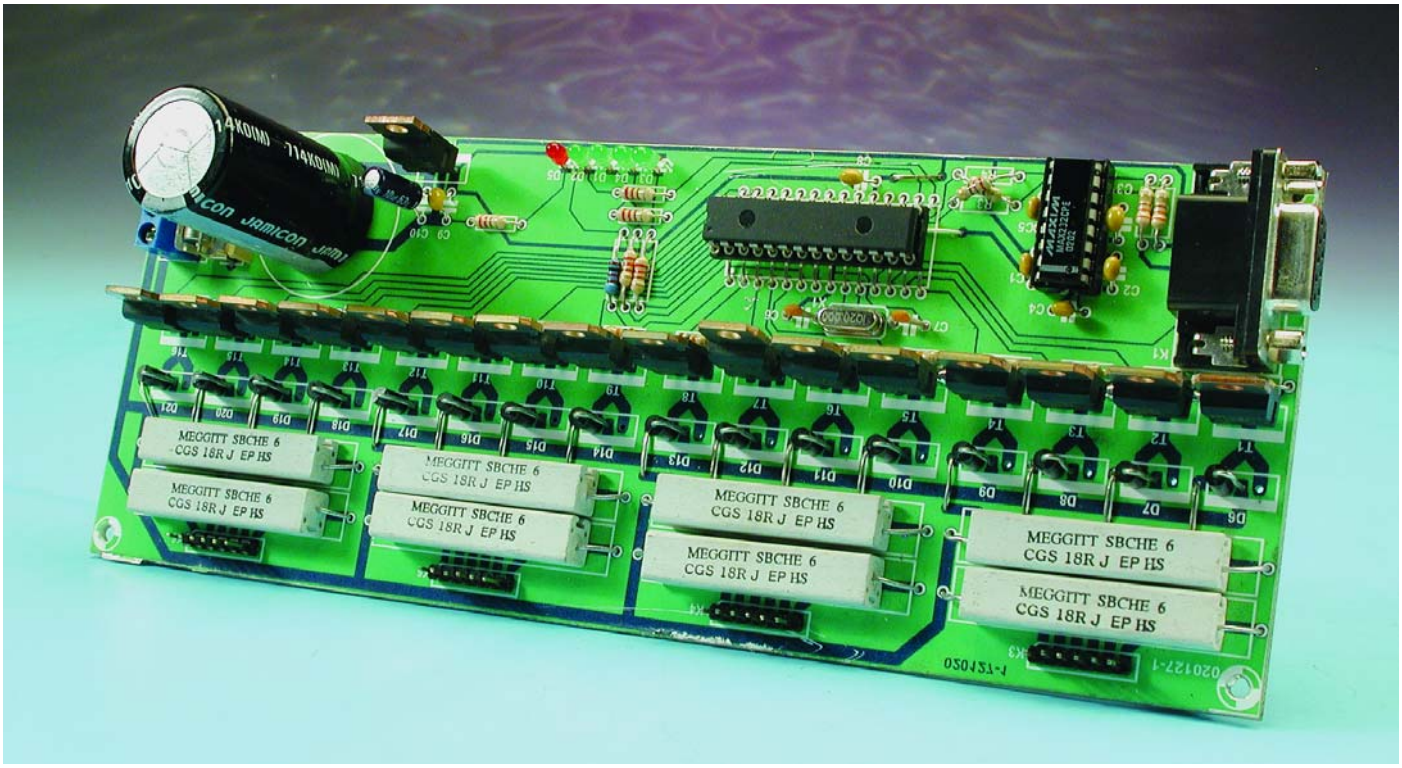

Figure 5. Our finished prototype of the stepper motor driver.

joy try reprogramming or replacing the microcontroller IC.

Once the circuit starts up correctly, use a 1:1 (non-crossed) D-9 female to D-9 male cable to connect the controller PCB to the computer RS232 port. Run the test software on the PC and select the correct COM port setting. Then try testing any of the commands. On sending a command, four LEDs should go out and one LED should light indicating which channel is in use. Once the command has been executed the four LEDs will light up. If this works as expected, turn off the controller PCB and connect a motor to one channel. It is important to ensure the phases and common taps are all correctly connected. Next, power up again and retest. The motor should spin smoothly, accelerating and decelerating when starting and stopping. If the motor seems to skip or the movement is jerky, check that the phases are connected in the correct order and that the acceleration rate is not too fast for that stepper motor. Remember, lower delay rates mean faster rotation. If you set up the speed with equal delays, for example, 800-800, there will be no acceleration or deceleration. Most stepper motors should work with 500-1000 step delays.

Once all required channels have been tested and are found to be working, you can customize the command (PC) software or control (PIC) software to include your own commands and improvements. One powerful variation could be to multiplex the motors, enabling more than one axis to spin at any time. Applications of the driver board described here may be found in robotics, for accurate positioning of mechanical parts in telescopes, robots, cameras, etc., or for precision movement and placement as required in CNC machine tools.

(020127-2)

Free Downloads

PIC and PC software (source code files). File number: **020127-11.**zip
PCB layout in PDF format. File number: **020127-1.zip**

www.elektor-electronics.co.uk/dl/dl.htm, select month of publication.

Useful links

Microchip PIC 16F87X microcontroller datasheet:
www.microchip.com/download/lit/pline/picmicro/families/16f87x/30292c.pdf

Direct download link to the QCCOM32 VCL for RS232 I/O in Borland Delphi:
http://geocities.com/scottpinkham/delphi/qccom32.zip

Low-cost PIC development tools compatible with Microchip MPLAB environment:
www.taylec.co.uk

A PIC C compiler compatible with Microchip MPLAB environment:
www.ccsinfo.com

VCLs for hardware port access and control:
www.programmersheaven.com, www.torry.net, www.codeguru.com

Useful literature

'Serial Port Complete' by Jan Axelson, ISBN: 0965081923
'PICProg 2003', Elektor Electronics September 2003.