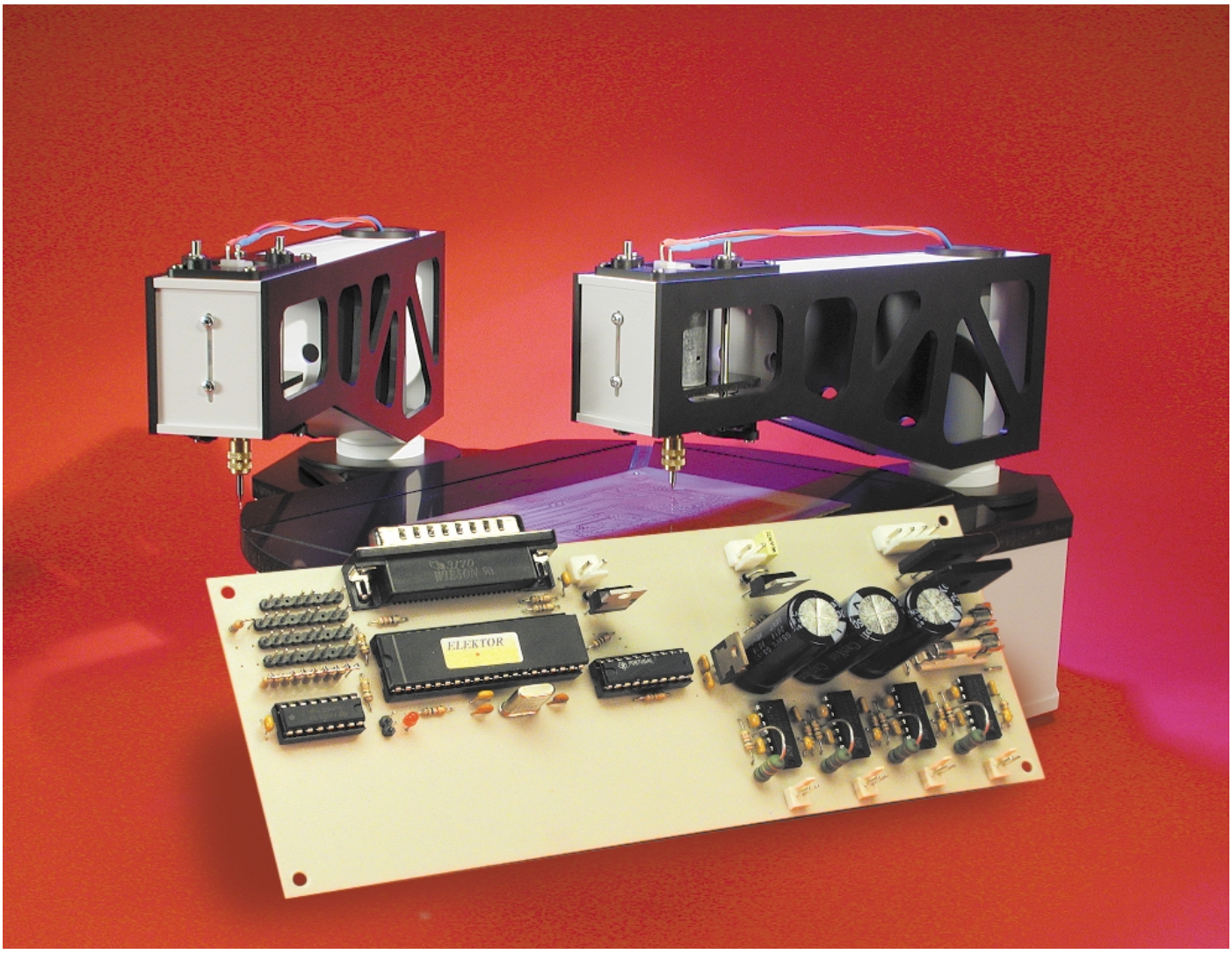
PCB Drilling Machine (2)

Part 2: The controller

Design by T. Müller (Radix GmbH) www.radixgmbh.de

Our PCB drilling machine provides a link between a Windows PC and the motors and solenoids of the machine itself: a microcontroller integrated with a high current drive circuit.



A universal controller board

The controller board, with its power output stages and integrated microcontroller, has been designed not just for use with the PCB drilling machine, but also with enough flexibility to be used for many other purposes. The driver stages make it a very handy controller board for other equipment or machines that employ stepper motors and other high current devices. For this reason there are a few places in the layout where we have made additions that are not strictly necessary for the drilling machine application.

To take one example, the output stages are controlled by a GAL. The GAL is simply programmed to combine the PWM signal with a two-bit address to activate the solenoid drive, and (with an enable input) the drill motors. This function could easily have been carried out using a simple address decoder such as a 74138. However, by using a GAL we can completely reconfigure the eight output stages. To this end, you will see that we have routed extra signals from the microcontroller to the GAL, in particular signals that can be assigned special functions in the microcontroller.

Series resistors and capacitors can be fitted at the inputs to the driver stages. The power MOSFETs used in the drilling machine controller do not require these; but if you wish to use bipolar transistors in the drive stages, the spaces we have left on the board will come in handy.

In the previous instalment (March 2001) we described in detail the basic principles of the CNC drilling machine. The controller is just as unconventional: the machine is not driven directly from a PC, but indirectly via a special controller circuit. For trouble-free communication between the Windows PC and the controller over the Centronics interface we need to make use of several tricks that take us beyond the standard Centronics protocol.

Let's start by taking a look at the rather straightforward hardware of the controller board. The basic circuit consists on one side of a microcontroller with a PC interface and on the other the output drive circuits, connected to I/O-port pins. Of course there is rather more to the design than that.

Real-time with Windows?

How can we control a stepper motor from a PC? It goes without saying that we need an output drive circuit capable of supplying the current required by the motor, since none of the PC's interfaces can drive a motor directly. And then, in order to make the motor step smoothly, we need very accurate timing.

At first blush it would be ideal if we could somehow connect the output stages directly to the PC and handle practically all the control signals directly as bits from the PC. The circuit would then be very simple, consisting of just a couple of latches and an address decoder.

Unfortunately, Windows has a rather unpleasant characteristic: its operations are totally asynchronous. It is impossible to say at exactly what time an event will occur. Approximate timing is possible, but as the specified time between events gets smaller and more precise, something eventually will go wrong. Windows on its own was not designed for control applications, and indeed is not suitable for them. It may be easy to use, but from the hardware point of view it is no more suitable than older operating systems such as DOS.

First of all, modern programming languages offer hardly any commands for access to the I/O-ports; and second, multitasking prevents direct hardware access by programs that attempt to bypass the operating system. Special drivers are required, such as `port.dll` found in the *Elektor Electronics* book 'PC Ports under Windows' (to be published soon).

Let us suppose that you have available a programming language that lets you control the port signals at will. We wish to generate a continuous square wave on one of the data pins of the Centronics port at a given frequency. In software this is a trivial matter: we simply need to toggle the bit in question regularly. In order to obtain the desired frequency, the appropriate delay must be used. If the delay is reasonably large, say 0.5 s, we obtain a perfect output signal with a frequency of 1 Hz.

If, however, we reduce the delay to say 0.5 ms and hence raise the frequency to 1 kHz, we hit a problem. The right frequency appears at the output, but only when averaged over a long period. The individual pulse lengths are a little longer or shorter in a random pattern, and some are very different from the specified period. Why is this?

Of course, modern processors are fast enough to toggle a bit 2,000 times per second. No, the reason is that Windows, as a multitasking operating system, has other things to do while generating our output waveform. Updating the mouse position, checking for incoming e-mail, reading audio data from a CD, communicating with peripherals: all these take processor time.

Windows is event-driven, which means that the various parts of the machine can cause the operating system to carry out operations. This can be seen when printing, for example: Windows launches a process to communicate with the printer, and supplies it with new data as and when it is ready to accept it. When that happens and how much data is transferred at a time is totally undefined, but nevertheless consumes processor time. This can interfere with the timing of our square wave and cause variations in its period.

There are two ways to overcome this problem. We could shut down all other processes and disable multitasking while we generate our square wave. Then we might as well use DOS, which is what most manufacturers of this type of control software for small machines do. Under DOS, an application program can use the full power of the processor for itself. Of course, we sacrifice the ease of use of Windows, and, in spite of its power, the computer cannot be used for anything else.

There are various operating system add-ons available for Windows which ameliorate this situation and provide quasi-real-time facilities. These are mostly VXD or TSR pro-

grams that are loaded when Windows starts up and then run in the background. These programs run briefly, but at an exact moment in time; and they run at the highest priority to guarantee having the necessary processor power available to them.

These programs are rather critical and considerably increase the chances of Windows crashing. They also permanently consume memory and processor time, because they are always present and must always be ready to spring into action when required.

These programs are also often written using undocumented features or 'back doors' in the operating system, and so are usually only fully compatible with one particular version of the system.

Offloading the synchronisation problem

We have therefore chosen an alternative approach to driving a stepper motor under Windows. Synchronisation is not left up to Windows software, but rather transferred to a microcontroller. The microcontroller receives a control command (for example for a motor pulse), and rather than executing it immediately, waits for further timing information to indicate when the data are to be processed. In this way, as long as the overall system is fast enough, we can obtain perfectly synchronised results using Windows.

The information passed to the microcontroller to generate a square wave runs as follows: bit high in 500 µs, bit low in 500 µs, Repeat. This information is sufficient for the microcontroller to produce the signal independently, requiring no further intervention from the PC. The PC must deliver the data fast enough, each command arriving within 500 µs. This should not present any difficulty for a reasonably fast PC.

To prevent the PC having to wait until the microcontroller is ready for a new command, the controller is equipped with a small FIFO memory with space for 40 entries. This allows the controller to be well decoupled from the PC. The PC can send off 40 precalculated commands for switching the various output signals to the controller and then has 40×500 µs to prepare new data or carry out other tasks.

Shortly before the FIFO empties, this state is flagged to the PC, which can then refill the FIFO with new data. This does not affect the controller, which continues executing commands one after another with perfect timing.

Using the Centronics interface

Communication with the controller could be carried out over any PC interface. The Centronics and V24 (serial) interfaces are always available, and are particularly suitable candidates.

There are certain problems associated with the V24 interface: getting the wiring right and arranging handshaking can be full of pitfalls. Matters are also more complicated for the microcontroller, which must first convert the serial bit stream into parallel bytes. And, if the microcontroller does not contain a UART, the controller must detect and interpret the start bits itself. This consumes processor time in the controller and ultimately creates difficulties in ensuring timely execution of commands.

For these reasons, we have chosen the Centronics interface. This interface is also superior from the point of view of speed, because eight bits are transferred at a time: this also makes the data easy to process. The data are simply written to a specific I/O address and the job is done. It is also convenient for the microcontroller to receive the data in a simple 8-bit format.

Eight bits may be a convenient number, but it is too few to specify a command with timing information. Multiple bytes must be assembled together to construct a useful command so that the controller can know what to do with the data. For our application, two bytes are enough for each command, but that is too much to transfer in one go over the Centronics interface. But when the 16 bits are divided into two independent bytes, a synchronisation problem can arise.

The strobe-edge trick

How can the controller know which of the bytes is the first and which the second? Counting is one option, but not a very reliable one: a single transmission error and all successive commands will be wrongly interpreted. The error must be detected and the controller reset. Alternatively, the data bytes could include further information to allow the controller to identify which is which.

However, this also has its disadvantages. The marker bits naturally reduce the amount of real information transferred, and so we have fewer bits available for commands.

For this reason we distinguish the bytes externally using a means already provided within the Centronics interface. Take a look at the protocol as shown in **Figure 1**. First consider normal operation, described at the top of the figure. The interface, however, will let us transfer two bytes at a time without modification. How that is achieved is described at the bottom of the figure.

The handshaking process is unchanged: we have simply changed the meaning of the signals. Two bytes are coalesced into a single packet, where the falling edge of the STROBE signal marks the first byte, and the rising edge, the second.

If an error occurs when the first byte is already in the process of being sent, STROBE will eventually rise, if only because of the pull-up resistor fitted to the controller board. The transfer is then terminated, and the pair of bytes remains together. Synchronisation problems are therefore impossible, even though the contents of the two bytes will be in error. If on the other hand we had simply counted bytes, then all subsequent bytes would be interpreted wrongly, as they would all be interchanged. If we were to continue in this manner, the controller would cease operating correctly hence the system would have to be re-initialised.

At start-up the control software sends a RESET command, which causes the controller to clear its FIFOs and bring to a halt all movements in progress in the system.

Controller functions

We have now described the main functions of the controller: accepting the 16-bit wide instructions, storing them in the FIFO, and synchronously executing the commands stored in the FIFO. Of course, the calculation of trajectories is not handled by the microcontroller, since it is not designed for trigonometric calculations and would be too slow. It would also imply programming the dynamic characteristics of the system once and for all into the microcontroller.

Instead, the movements are pre-

processed in the PC and then sent to the controller in the form of primitive instructions such as *motor 1 one step counterclockwise*. All movements are defined in the PC in this way. It is possible to compute the step information for any desired motion and then send it to the controller: and thus the microcontroller never needs to be reprogrammed.

Data format

Now that we have explained how the command data are calculated in the PC and transferred over the Centronics interface to be synchronously processed by the controller, we can take a look at to the internal structure of the command data.

The two bytes are concatenated to form a word. This word contains a data value, or operand, along with an address which determines how the operand is interpreted: in other words, the particular command to be executed. The number of address and data bit is variable, the number of data bits shrinking as more address bits are required. A command with a short address field can contain a longer operand, and vice versa. This kind of addressing scheme can be processed using a sequence of branches, decoding the command using the 'divide and conquer' principle.

This works as follows: the first address bit determines whether we are dealing with one particular type of command (when the bit is a '1') or not (when it is '0'). In the second case we still do not know what type of command we are processing, so we examine the second bit. If this is a '1', then we have decoded a second class of commands, while if it is '0', we proceed to examine the third bit, and so on. Of course, as we continue to examine bits is this way, there remains less and less space for the operand data.

The advantage is that we only need one address bit for the first command type, and so we have a full 15 bits available for the operand data. For commands whose address field is, say, three bits long we have 13 bits available. Note that a three-bit address field allows at most eight different commands to be separately decoded.

1. Quiescent condition:
STROBE from PC to microcontroller is high. BUSY from microcontroller to PC is low.

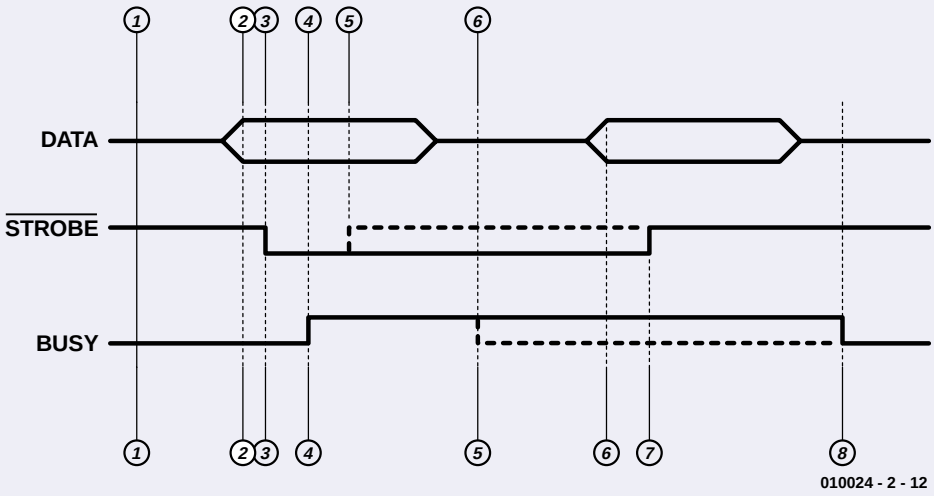
2. The PC wishes to transfer data as usual, and puts the data byte on the eight data lines.

3. The PC takes STROBE low. The connected microcontroller (or peripheral device) sees the low state and knows that the data are ready to be read.

4. The microcontroller takes the data byte and sets BUSY high in acknowledgement.

5. The PC takes the STROBE signal high again.

6. When the microcontroller has processed the byte, it indicates this state by taking the BUSY signal low. This completes the transfer of a byte. The system is ready to transfer another byte, the control signals (BUSY and STROBE) being once more in their original states.



1. Quiescent condition:
STROBE from PC to microcontroller is high. BUSY from microntroller to PC is low.

2. The PC wishes to transfer data using the special mode, and puts the data byte on the eight data lines.

3. The PC takes STROBE low. The microcontroller sees the low state and knows that the data are ready to be read.

4. The microcontroller now takes the data byte and sets BUSY high in acknowledgement.

5. The microcontroller processes the byte immediately and leaves the BUSY signal high.

6. The PC observes that the BUSY signal is high and therefore that the first byte has been accepted. It then puts the second byte on the data lines.

7. The PC indicates that the second byte is present by taking STROBE high again. This causes the microcontroller to accept this byte also.

8. When the microcontroller has finished its processing, it sets BUSY low again.

Figure 1. The original Centronics protocol (above) and as modified for the drilling machine (below).

The variable-length address field also allows a higher priority to be assigned to more common or more urgent commands. For a rarely used command, or one that initiates a lengthy action, we can allow the address decoding to take a little longer.

The commands which are most commonly used, and which sent to the controller at the fastest rate, are without doubt the stepper motor commands; they also require a long operand, which is provided by the highest priority command format. The format of the commands for sending stepping pulses to the motors is set out in **Table 1**.

The second command class is the drilling command. Three address bits can be used for this command without causing any difficulty because its execution time is much longer than the motor step commands, and because it requires a shorter operand. The meaning of the various bits in the drilling command is set out in **Table 2**.

These two commands suffice for basic operation of the PCB drilling machine. In fact there is a range of other commands which are less relevant to our discussion and which are not described here.

Brawn and brains

At first glance the circuit diagram in **Figure 2** may look like an impenetrably complex piece of electronics, but on closer inspection we see that the brains of the circuit are actually very simple. The intelligence is concentrated in a type PIC16C64 microcontroller from Microchip, clocked at 20 MHz. This microcontroller boasts the magnificent total of 33 individually addressable I/O-ports, but only 2 K of EPROM program memory and 128 bytes of internal SRAM, here used for (among other things) the FIFO memory. The PIC16C64 is equipped with peripherals such as a real-time clock, timer/counter and capture/compare inputs. In this application we do not use the 3-wire

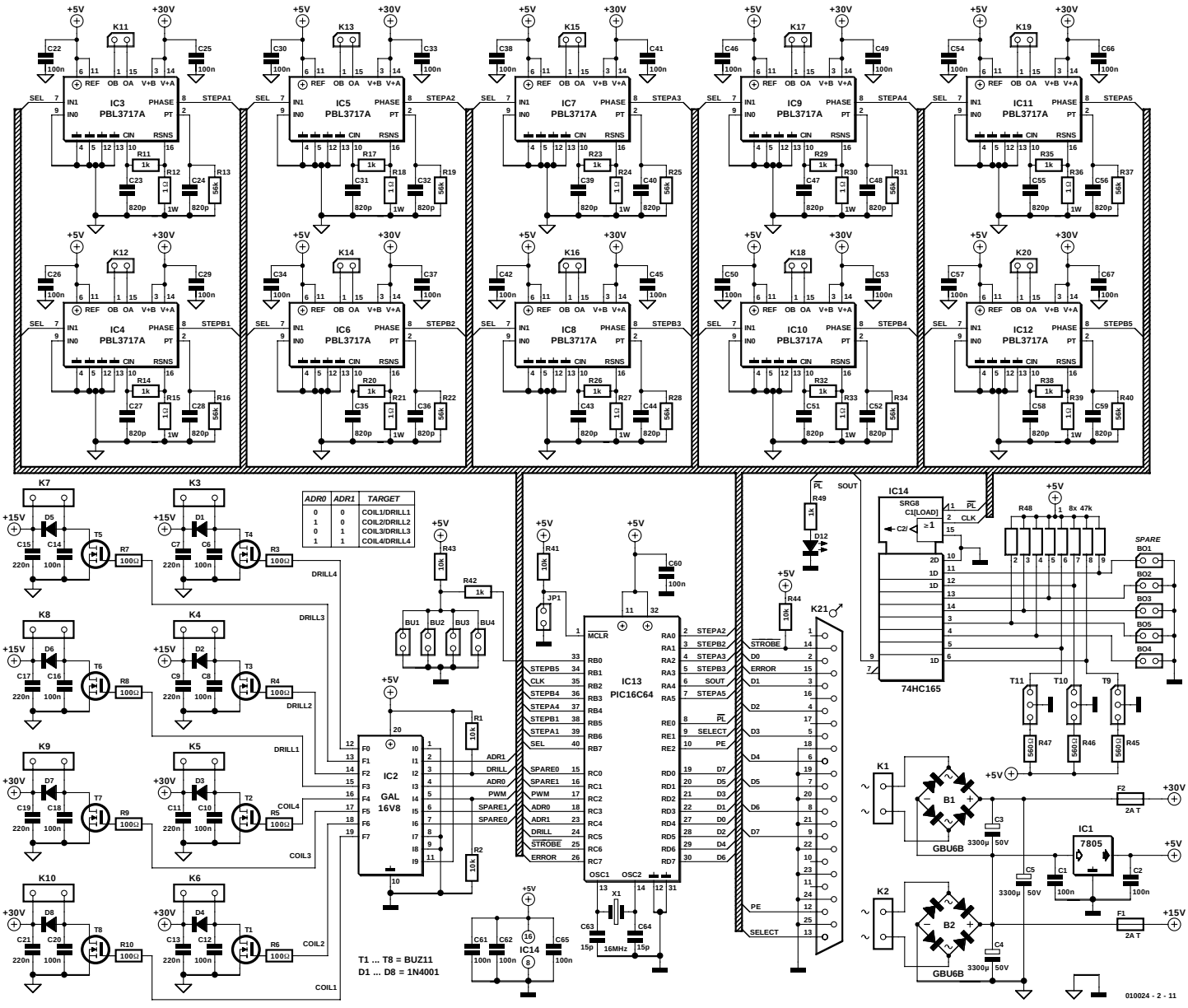


Figure 2. The drilling machine controller board: lots of brawn, and a little bit of brains!

Table 1: Motor pulse command

MSB	Byte1				Byte2				LSB
1	Z	Z	Z	D	T	T	T	Z Z Z Z	Z Z Z Z

Addressing: bit 7, byte 1 = '1'.

Z (11 bits): 'Time'
The time value specifies how long AFTER the execution of the motor pulse command the controller must wait before executing the next instruction. If the value is zero, then the next command (assuming it is also a motor pulse command) is executed immediately and synchronously with the present one.

T (3 bits): 'Target'
These three bits select the stepper motor drive stage affected by the command. The order runs from Target 1 ('000'), the output stage driving connectors K11 and K12, through to Target 5 ('100'), the output stage driving connectors K19 and K20.
Target addresses 6, 7 and 8 ('101', '110' and '111') are reserved.

D (1 bit): 'Direction'
This bit specifies the direction in which the motors are to turn. D=1 specifies clockwise rotation, D=0 counterclockwise.

Table 2: Drilling command

MSB	Byte1							Byte2				LSB
0	T	T	T	1	1	B	B	B B B D	D	D	D	D

Addressing: bit 7, byte 1 = '0'; bits 2 and 3, byte 1 = '11'.

T (3 bits): 'Target'
These three bits select the output stage for the drilling command. The machine can be fitted with up to four arms, and each of these arms has its own drill and a dedicated driver stage on the controller board. The targets are numbered from left to right, starting with Target 1 ('000') at connector K10, where the solenoid for the first arm is connected. The corresponding drill motor is connected to K8. Target 4 ('011') is driven from connectors K5 (solenoid) and K3 (drill motor). Targets 5 to 8 ('100' to '111') are reserved.

D (5 bits): 'Drilling force'
The force applied to the drill by the solenoid can be controlled by this five-bit data value. The value sets the amuont of energy delivered to the solenoid, and can be set to any number in the range 0 to 31.

B (5 bits): 'Braking control'
When the drilling head is withdrawn the solenoid cannot simply be switched off, since this would cause long-term wear on the mechanism. To prevent undue stress on the motor guide spring, the magnetic field in the solenoid is gradually reduced, so that the drilling head is withdrawn gently.
In principle this value can be set in advance for a particular weight of drilling head. Here, however, we have the possibility of optimising the value. If, for example, you use a more powerful drill motor or a 3-jaw chuck instead of the collet chuck, the weight of the drilling head may be significantly different.

synchronous serial SPI/I²C bus.

The modified Centronics interface uses port D (data signals), RC6 (active-low STROBE), RE1 (SELECT) and RC7 (ERROR). The BUSY signal from the microcontroller is not wired to the standard connection on pin 11, but to the PAPER EMPTY signal (PE) on pin 12. There are several reasons for this: first, it prevents the PCB drilling machine from springing into action when a print job is started under Windows, and second, it allows a printer to operate in parallel with the machine. Further, pulses on the STROBE and BUSY signals appear to affect the internal interrupt processing of Windows: and that is best left well alone.

R44 is the pull-up resistor referred to above that pulls STROBE high in the event of a system crash, thus deactivating the signal.

The microcontroller determines the configuration of the drilling machine via the drilling head limit switches connected to **Bo1-Bo4**. If a drilling arm is fitted and the head is in the 'up' position, the corresponding switch is closed. If an arm is not equipped with a limit switch, the corresponding contacts remain open.

OptSpare is an optical input that, like the switch input Spare, is not used in the present design. **OptBase** is an optical input used to calibrate the reference positions of arm 1, arm 2 and the rotating table. **OptAdd** does the same job for arms 3 and 4. How exactly the calibration is carried out will be described in a later article in this series. The LED can be used to indicate that the machine is operating: it lights during drilling and during the calibration process.

It may seem that 33 I/O-ports are plenty, but unfortunately are not enough for all the signals just described. We therefore use a shift register type 74HC165 (IC14) to combine the bits into a single byte read serially by the microcontroller over a single port bit RA4 (SOUT). A parallel load pulse loads the bits into an intermediate register where they are buffered before being clocked out using the CLK signal (RB2).

The final item of information read by the controller is the 'stop' signal from the solenoid limit switch. These switches are connected to BU1-BU4. Since the current must be switched off quickly, it is not feasible to read these signals out slowly via a shift register. Instead, the stop signal drives the interrupt input RB0 low. Since only one drilling head is actuated at a time, the four limit switches can be connected in parallel.

The rest of the circuit consists of the four identical output drive stages for the solenoids (T1, T2, T7 and T8), the drill motors (T3-T6) and ten identical integrated stepper motor

drivers type PBL3717A from ST Microelectronics. The datasheet for this IC can readily be found on ST's website at www.st.com.

First let us consider the drill motors and solenoids. These are not driven directly, but rather via a programmable logic device type GAL 16V8. You can read the reasons for this in the box *A universal controller board*. Address signals ADR0 (RC3) and ADR1 (RC4) select from the at most four motors and solenoids in the order shown. The level on DRILL (RC5) selects either the motor (when RC5 is low) or the solenoid (when RC5 is high). In this way we make sure that in the case of a failure in the circuit the drill will not turn and the drilling head will be up. When the circuit is switched on, the PIC's ports are in a high impedance state, and resistors R1 and R2 hold the signals at suitable levels to ensure that the motors and solenoids are safely turned off.

The controller activates the selected motor/solenoid pair using port pin RC2. The pin does not just turn on and off, but provide a current control using a PWM signal, different for the solenoid and for the motor.

The solenoids are controlled using a rather complicated current waveform. The timing depends on the instantaneous position of the drilling head (which is determined from the two switches on the head guide), the value specified in the drilling command, and on whether an error situation arises due, for example, to a timeout resulting from a jammed or dirty guide. The important fact is that the PWM control value is calculated at each point in time to actuate the guide solenoid as smoothly as possible.

The solenoid coils are rated for continuous operation at 12 V, and they are driven (although only momentarily) from the same +30 V supply as the stepper motors. This produces an enormous magnetic field to force the drilling head down. As soon as the appropriate Bo switch opens, the PIC can deduce that the solenoid has moved and it then proceeds to regulate the current to the value specified in the drilling command using the PWM signal. Although the solenoid is initially overdriven, in continuous operation it is only driven at effectively 12 V with an 80% duty cycle. This is within the specified limits. Overall, with typical activity, the duty cycle is only a few percent, and in practice the coils only get warm to the touch.

The drill motors are controlled during switch-on and running using a PWM ramp that reaches 100% duty cycle during drilling.

The stepper motors are activated using the STEP signals. Each PBL3717A integrated driver controls one motor winding, and two drivers (STEPAx and STEPBx) are allocated to each motor. The drivers are responsible for producing the drive pulses and for converting the +5 V logic levels to +30 V, the operating voltage of the stepper motors. The microcontroller activates the driver via the level (repsectively from STEPAx and STEPBx) at the PHASE input, and a current then flows from OA to OB. Inputs IN0 and IN1 have a special function: they allow control of the output stage drive current.

This is normally achieved using a retriggerable timing circuit for each output stage. If no more motion is required of the motors, the pulses stop and the current is reduced in the motors. Here, this is realised via the signal from RB7 (pin 40) connected to the IN1 pins of all the PBL3717s. When motion is required, all the motors are driven with full current (even those which are not to move), by setting IN1 = 0. Shortly after the motion stops, IN1 is set to 1 again, and the current drive is reduced to 19%.

Why do we increase the current to the motors not involved in a motion? Partly because in this way the power supply is loaded to the same extent independent of the motion, and partly because, as a result of various resonances in the system, those motors driven at 19% of the maximum current can be jogged out of position. Although this is rather unlikely, it would lead to errors almost impossible to track down.

(010024-2)

In the next instalment of this series we get down to the construction of the controller board as well as the nuts and bolts of the project. The various parts of the PCB drilling machine will be brought together, with illustrations — and lots of glue and grease!