

# A Microcontroller System for Experimentation

Niklaus Wirth, 16. 4. 2008

## Abstract

We report on the design of a simple board for use in teaching about microcontrollers. The project also describes a small language specifically designed for the PIC microcontroller. The project showed that hands-on experience is essential in teaching about sensors and controlling devices, and that the use of a small, hand-tailored language eases programming considerably.

## 1. Introduction

This work is largely motivated by my (former) teaching activities. In a course explaining the basic principles of computers, and then computer architectures, it is a good idea to start small and simple. Such simple computers still exist, fortunately, and provide the opportunity to directly offer hands-on experiences in a laboratory. This is always much more impressive than simulation or emulation. Furthermore, it can be established at very low cost, if the teacher is willing to lay hands on by himself.

Here we present a small board equipped with two microcontrollers PIC fabricated by Microchip Inc. On this board are drivers for stepper motors and a 2-phase motor, as well as DA and AD converters. This allows to explain and experiment with peripheral devices in a direct way, a topic that is often neglected in computer science curricula. The chapter on applications is therefore held in a rather tutorial style.

Microcontrollers are (still) the domain of assembler codes for programming. In fact, there is much to say for this, as the instruction sets are simple and the resources quite limited. Specifications of conventional languages would have to be accompanied by long lists of restrictions, of do's and don't do's.

Nevertheless, programming with assemblers is tedious and error-prone, and the resulting texts are cryptic and long. It was therefore felt to be worth a try to explore a "middle way", to postulate a small language adapted to the needs and limitations of a microcontroller, in this case the PIC. The challenge lies in finding a form that offers the advantages of program structure, yet reflects the limitations and the underlying architecture in such a way that neither is the programmer misled nor does the compiler have to perform complicated and obscuring "optimization". With this in mind, the language PICL was designed. It is described in a separate Report. Its compiler and the necessary support tools, such as a program loader, are programmed in Oberon available for PCs. Its description is contained in another Memo.

As a preparation and prerequisite for the subsequently described applications, the following chapter provides an overview over the PIC's structure, reduced to the minimum necessary. Even when programming in PICL, it is important for an engineer to be familiar with the PIC's architecture, its resources and limitations.

## 2. The PIC Architecture

The PIC processor is a typical microcontroller insofar, as control unit, arithmetic/logical unit, and memory are all placed on a single chip.

The PIC features a Harvard architecture: The control unit fetches instruction from the instruction memory, and the arithmetic unit handles data from a data memory. The two memories are separate, the former is implemented as a read-only memory (ROM), and the data memory is a regular random-access memory (RAM). Its elements are called registers, which is somewhat misleading, because there is one genuine register, the W-register, directly coupled with the arithmetic/logic unit (ALU).

Also typical for a microcontroller is that its system bus is not accessible from outside the chip. Its interface consists of two registers (A and B) consisting of 5 and 8 bits respectively, which can be controlled and configured independently.

What makes this chip (the PIC16C84) particularly appropriate for experimentation, is that the program memory is electrically programmable, erasable, and reloadable (EEPROM).

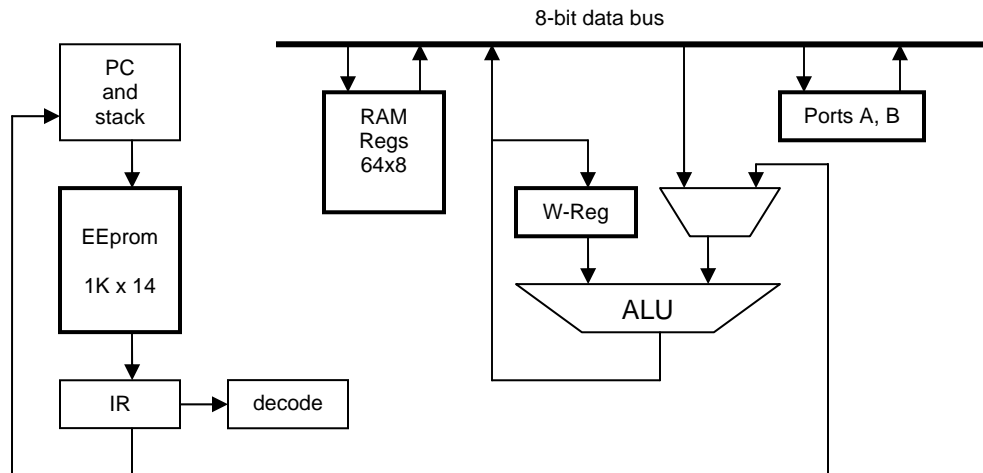


Fig. 1 The PIC architecture

The PIC's instruction set is refreshingly small and simple. Instructions consist of 14 bits and are divided into 4 classes as shown in Fig.2. The byte-oriented instructions operate on the W-register (accumulator) and the addressed byte in memory, and they store the result either in the same memory location or in the W-register. They include addition, subtraction, and the basic logical operations. The bit-oriented instructions set, clear, or test a single bit addressed by the field *adr*, and numbered by the field *b*. The literal data instructions are like the byte-oriented instruction, except that one operand is the literal contained in the instruction instead of the addressed byte in memory. Furthermore, there are a jump and a subroutine call instruction with an 11-bit address field. (For further details, we refer to the manual).

Two of the addressable locations (registers) are the A and B ports, and one is the status register S, containing a zero and a carry bit, the conventional condition code.

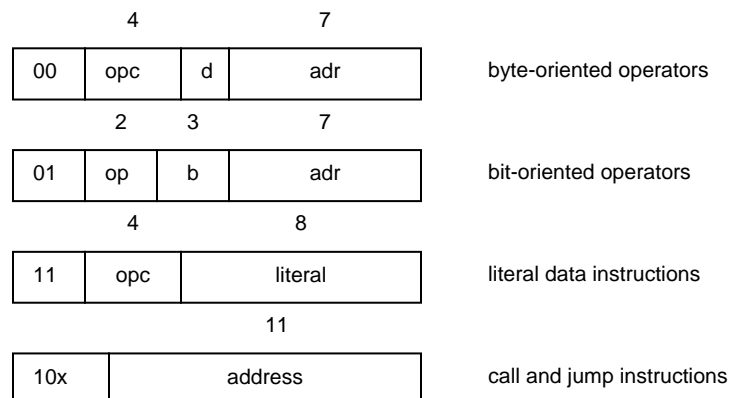


Fig. 2. PIC instruction formats

### 3. Support Tools

The main tools used in these experiments are the PICL compiler and the program loader. They are available in the form of commands (in any text):

PICL.Compile @ selects the marked text for compilation  
 PICL.Program loads the compiler program into program memory

In addition, further commands are available in the same module:

PICL.Verify verifies the preceding loading  
 PICL.Reset puts the PIC in reset mode (keeps the reset signal low)  
 PCL.Run start execution (removes the reset signal)  
 PICL.Erase erases the loaded program

This tool uses a 5-wire serial connection to the PIC via the PC's parallel port. The signal assignments are as follows:

Signal	PC	PIC
Data	D0 out / D5 in	B7
Clock	D1 out →	B6 in
Data	D5 in ←	A4 out
Reset'	D2 out →	MCR
Program	D3 out →	MCR

Note: MCR is a 3-level signal and therefore connects to 2 binary signals at the parallel port. For further details, see Sect. 5.

## 4. Applications

### 4.1. A serial data link

For the following applications, in fact for practically all applications, a communication link to the host computer is a prerequisite. Therefore we first describe a pair of read/write procedures for the PIC, communicating with its corresponding pair on the PC. The latter are described in Oberon, the former in PICL.

The first step in designing such a package is the definition of the data protocol. This protocol is determined by the available lines. Here we have 3 lines available. This implies a serial protocol, and we choose a byte (8 bits) as unit for transmission. A fairly obvious choice in our case is a *handshake* protocol, because it is timing-independent. This is desirable, particularly if the two partners have quite different speeds.

#### 4.1.1. Serial communication using the handshake protocol

The underlying principle is that the sender issues a request signal on the request line, and at the same time applies the data bit on the data line. After the receiver has noticed the request and sensed the data line, it issues an acknowledge signal. After noticing this acknowledgement, the sender resets the request and proceeds with the transmission of the next bit. This process is shown in Fig. 3. The idle values of *req* and *ack* are 1.

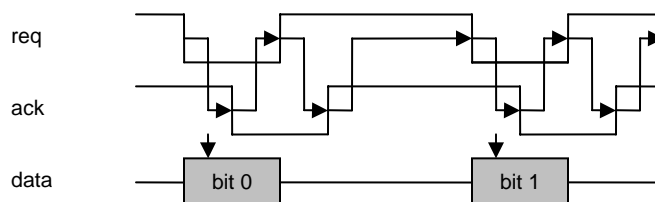


Fig. 3. Handshake protocol

The procedures for sending and receiving for the host computer (PC) use auxiliary procedures *wait(b)* for delaying until *ack = b*, and *S(b)* for applying *b* to the data line.

```

PROCEDURE wait(b: INTEGER);
  VAR ch: CHAR;
BEGIN (*test D5, ack*)
  REPEAT SYSTEM.PORTIN(in, ch) UNTIL ORD(ch) DIV 20H MOD 2 = b
END wait;

PROCEDURE S(d: LONGINT); (*send d*)
BEGIN SYSTEM.PORTOUT(out, d)
END S;

PROCEDURE R(VAR b: INTEGER);
  VAR ch: CHAR;
BEGIN (*read D6, dat*) SYSTEM.PORTIN(in, ch); b := ORD(ch) DIV 40H MOD 2
END R;

PROCEDURE Send*(d: LONGINT);
  VAR i: INTEGER;
BEGIN wait(1);
  FOR i := 0 TO 7 DO
    S(d MOD 2 + 4); wait(0);
    S(d MOD 2 + 6); wait(1);
    d := d DIV 2
  END ;
  S(7)
END Send;

PROCEDURE Receive*(VAR d: LONGINT);
  VAR x, b, i: INTEGER;
BEGIN x := 0;
  FOR i := 0 TO 7 DO
    wait(0);
    R(b); x := (x DIV 2) + (b * 80H); S(5);
    wait(1); S(7)
  END ;
  d := x
END Receive;

```

The procedures for sending and receiving for the PIC are straight-forward. Note that in PILC the statement *!s* sets *s* to 1, *!~s* sets *s* to 0, and *?s* waits, until *s = 1*.

```

PROCEDURE Rec(): INT;
  INT i, x;
BEGIN x := 0; i := 8;
  REPEAT ?~B.6; ROR x;
    IF B.7 THEN !x.7 ELSE !~x.7 END ;
    !~A.4; ?B.6; !A.4; DEC i
  UNTIL i = 0;
  RETURN x
END Rec;

PROCEDURE Send(INT x);
  INT i;
BEGIN ?B.6; !S.5; !~B.7; !~S.5; i := 8;
  REPEAT
    IF x.0 THEN !B.7 ELSE !~B.7 END ;
    !~A.4; ROR x; ?~B.6; !A.4; ?B.6; DEC i
  UNTIL i = 0;
  !S.5; !B.7; !~S.5
END Send;

```

This protocol is symmetric with respect to the two partners. The *req* and *ack* lines simply exchange their roles. However, the data line is always driven by the sender, and therefore needs to be bidirectional. Here, the data line is driven by the PC, except when the PIC sends a byte. Setting the data line (B.7 is a tri-state pin) to output mode is achieved by the first line of procedure

Send. (Bit 5 of the status register *S* enables the program to access the tri-state control register). The transmission rate achieved by this protocol is about 50 Kbit/s, if a 3.68 MHz oscillator is used for the PIC.

#### 4.1.2. Serial communication using an asynchronous protocol

In this case, all three lines maintain their direction independent of who is sender and who is receiver. The clock is always generated by the same partner, which is therefore called the *master*.

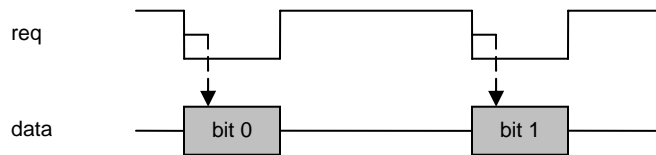


Fig. 4a. Asynchronous protocol; master = sender

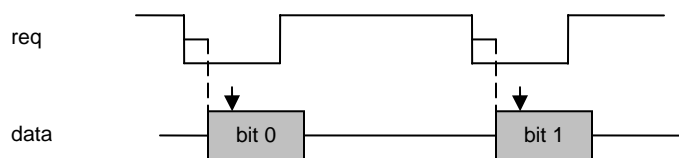


Fig. 4b. Asynchronous protocol; master = receiver

The disadvantage of this solution is its dependence on timing. If the master is sender (Fig. 4a), then there is no feedback from the receiver as to whether the data were received correctly. And if the master is the receiver, there is no certainty that the sender is actually providing data. This may seem to be quite unacceptable, but it is often used in practice and works well, provided the timing is correct. Sender and receiver in the faster partner must include appropriate delays, and if the partner changes, the delays must be adjusted.

The procedures for the PC, which acts as master, use auxiliary procedures *S* and *R* for accessing the parallel port, and *wait* for delaying the process.

```

CONST out = 378H; in = 379H; (*port addresses*)
      del0 = 2000; del1 = 1000; (*processor dependent*)

PROCEDURE wait(k: LONGINT);
BEGIN
  REPEAT DEC(k) UNTIL k = 0
END wait;

PROCEDURE S(d: LONGINT);
BEGIN SYSTEM.PORTOUT(out, d)
END S;

PROCEDURE R(VAR b: INTEGER);
  VAR ch: CHAR;
BEGIN (*read D6, dat*) SYSTEM.PORTIN(in, ch); b := ORD(ch) DIV 20H MOD 2
END R;

PROCEDURE Send*(d: LONGINT);
  VAR i: INTEGER;
BEGIN
  FOR i := 0 TO 7 DO
    S(d MOD 2 + 4); wait(del0);
    S(d MOD 2 + 6); wait(del1);
    d := d DIV 2
  END ;
  S(7)

```

```

END Send;

PROCEDURE Receive*(VAR d: LONGINT);
  VAR x, b, i: INTEGER;
BEGIN x := 0;
  FOR i := 0 TO 7 DO
    S(5); wait(del0);
    S(7); R(b); x := (x DIV 2) + (b * 80H); wait(del1)
  END ;
  d := x
END Receive;

```

The routines for the PIC use B7 and A4 for data, and B6 for the clock. B7 and B6 are permanently configured for input, A4 is configured for output with idle value 1.

```

PROCEDURE Rec(): INT;
  INT i, d;
BEGIN d := 0; i := 8;
  REPEAT ?~B.6; ROR d;
    IF B.7 THEN !d.7 ELSE !~d.7 END ;
    ?B.6; DEC i
  UNTIL i = 0;
  RETURN d
END Rec;

PROCEDURE Send(INT x);
  INT i;
BEGIN i := 8;
  REPEAT ?~B.6;
    IF x.0 THEN !A.4 ELSE !~A.4 END ;
    ROR x; ?B.6; DEC i
  UNTIL i = 0
END Send;

```

## 4.2. A Digital to Analog Converter

Converting a digital signal encoded as an integer into an analog voltage is fairly straight-forward. An integer  $x$  is typically represented by  $n$  bits such that

$$x = x_{n-1}2^{n-1} + \dots + x_12^1 + x_02^0, \quad x_i = 0 \text{ or } 1$$

A voltage  $v$  corresponding (analog) to  $x$  is obtained by feeding the sum of currents into a differential amplifier, where each current corresponds to a bit of  $x$ , which control a current switch feeding its current either to the amplifier or to ground. (Given a high-gain amplifier, we can think of both its input being at zero potential). The currents are fed through identical resistors ( $2R$ ) from a so-called R-2R ladder (see Fig. 5) This ladder is such that each component halves the voltage, i.e.

$$V_i = V_{\text{ref}} * 2^{n-1},$$

where  $V_{\text{ref}}$  is a so-called reference voltage. Hence, the output voltage is  $V_{\text{ref}} * (x/2^n)$ . The disadvantage of this simple circuit is that an inverting amplifier is used which required a negative supply voltage. This is inconvenient, as it also produces a negative output.

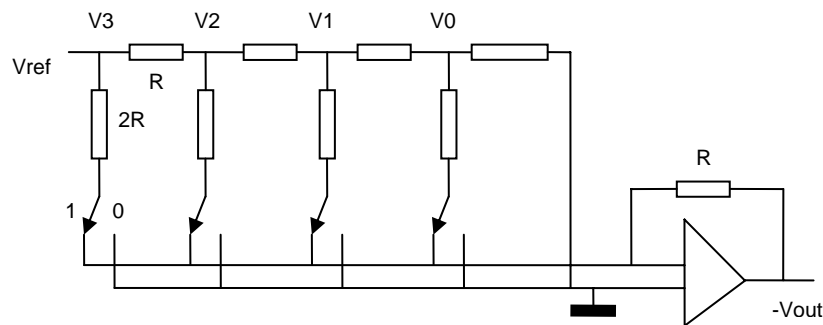


Fig. 5. Current switching R-2R ladder

A more convenient, and fortunately even simpler solution is to make use of the dual role of voltage and current in resistor networks, and to exchange  $V_{ref}$  and  $V_{out}$  as input and output of the ladder. Fig. 6 shows that this solution does not even require an amplifier.

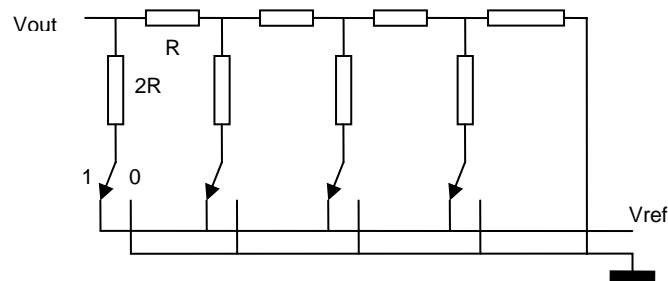


Fig. 6. Voltage switching backward R-2R ladder

Such DA converters are available as single chips. Some accept the input in parallel with  $n$  input pins, others provide a shift register and accept a single, serial signal and a clock for controlling the shifter. The latter are slower, but more convenient in connection with microcontrollers. We select the MAX539 device, an 8-pin DIP. It has 3 digital inputs: the serial data (connected to the PIC  $B.0$  signal), the shift clock (connected to the PIC  $B.1$  signal), and an enable signal (connected to the PIC  $A.0$  signal). The input values consist of 12 bits ( $n = 12$ ) and must be fed with the most significant bit first. Our solution feeds 8 bits only, followed by 4 insignificant bits of arbitrary value.

The PIC driver procedure contains a single repeat statement, shifting the input  $d$  onto the data line  $B.0$ .

```

PROCEDURE DAConversion(INT d);
  INT i, x;
  BEGIN !~B.1; x := d;
    i := 8;
    REPEAT !~B.1;
      IF d.7 THEN !B.0 ELSE !~B.0 END ;
      !B.1; ROL d; DEC i
    UNTIL i = 0;
    i := 4;
    REPEAT !~B.1; ROL d; !B.1; DEC i
    UNTIL i = 0;
    B := x
  END DAConversion

```

We have (intentionally) neglected the problems of timing. The clock rate must of course be appropriate for the converter. The program above yield a rate of ... MHz, whereas the converter's maximum rate is 1 MHz.

### 4.3. An Analog to Digital Converter

A-to-D conversion is significantly more complicated, and there are several different ways to accomplish it. Basic to all solutions is a voltage comparator, i.e. a differential amplifier. It delivers a single bit, indication which of the two inputs is higher.

Sequential ADCs use a single comparator and a series of  $2^n$  identical resistors, providing all voltages  $V_k = V_{ref} * (k/2^n)$  for  $k = 0 \dots 2^n - 1$ . The converter then searches for the  $V_k$  closest to  $V_{in}$  by performing a binary search. This takes some time, as there are  $n/2$  comparisons to be made.

Much faster are converters using  $2^n$  comparators, so-called *flash* converters (typically used in digital oscilloscopes). The task is then to find the largest  $k$  with  $V_k = 0$ , which can be accomplished by a purely combinational digital circuit.

Here we use a sequential converter providing a serial output signal, the TLC549, also a 8-pin DIP. Its data output is connected to the PIC's A.3 port bit, the shift clock comes from B.1, and the enable signal from A.1. The data is delivered with the most-significant bit first (MSB), as shown in Fig. 7. The data must be sampled after the clock rises.

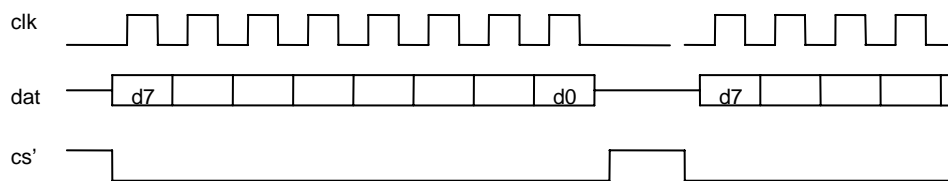


Fig. 7. ADC converter signals and timing

The chip contains a buffer holding the last measurement. This means that fetching a byte delivers the data buffered from the preceding sample, and at the same time triggers the next sample (AD conversion result).

```

PROCEDURE ADConversion(): INT;
  INT d, i;
BEGIN !~B.1; d := 0; !~S.0; i := 8;
  REPEAT !~B.1; ROL d;
    IF A.3 THEN !d.0 ELSE !~d.0 END ;
    !B.1; DEC i
  UNTIL i = 0;
  RETURN d
END ADConversion;

```

#### 4.4. Controlling a Stepper Motor

Stepper motors are used to turn the shaft to an exact position. They are typically used in disk drives for positioning the read/write heads over the rotating disk. Their principle is explained in Fig. 8. The rotor is pulled by a magnetic field applied to one of the windings or poles. By applying current to consecutive poles, a movement results. The figure shows an arrangement with 4 poles only. In reality there are many poles, typically 400 corresponding to the degrees of a full circle. Every fourth pole is placed in the same winding (wire). Hence there are exactly 4 windings, called *phases*, independent of the actual number of poles. Current always flows in the same direction in all windings. Therefore this scheme is called *unipolar*.

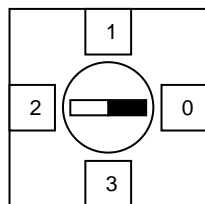




Fig. 8. Schematic of a 4-pole stepping motor

The task of the microcontroller is to generate pulses on the 4 wires with appropriate timing. An open-collector driver is placed between the PIC and each phase of the motor. We use two 75477 8-pin DIPs, each containing two drivers with an additional input for enabling or disabling, as shown in Fig. 9. A 0 at the PIC's output represents current flowing and a magnetic field generated, a 1 that it is off.

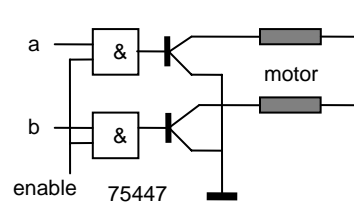


Fig. 9. A 75447 motor driver pair

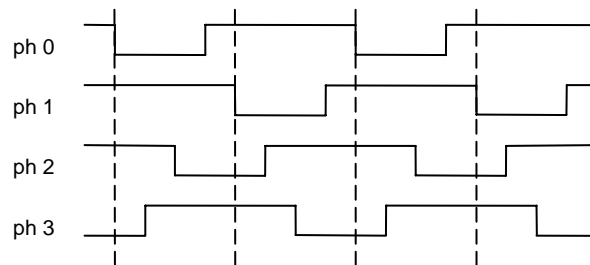


Fig. 10. The 4 phase signals for a unipolar stepper motor

The four phase signals are derived from the signal diagram shown in Fig. 10. They are generated by the following PIC program. We assume ports B0 – B3 to be the phase signals.  $n$  denotes the number of steps to be moved, and  $on$  and  $off$  are delay values determining the pulse width and thereby the motor speed.

```

PROCEDURE delay(INT k);
BEGIN
  REPEAT k := k - 1 UNTIL k = 0
END delay;

PROCEDURE phase(INT x);
BEGIN B := x; delay(on); B := 0; delay(off)
END phase;

PROCEDURE StepForward(INT n);
BEGIN
  REPEAT phase(1); phase(3); phase(2); phase(6); phase(4); phase(12); phase(8); phase(9); DEC n
  UNTIL n = 0
END StepForward;

PROCEDURE StepBackward(INT n);
BEGIN
  REPEAT phase(9); phase(8); phase(12); phase(4); phase(6); phase(2); phase(3); phase(1); DEC n
  UNTIL n = 0
END StepBackward;

```

It is mandatory to ensure that no current flows, when the motor is idle, and after a reset signal to the PIC. The former requires that the 4 phase outputs are set to 1. A reset causes all PIC ports to assume a high-impedance state (input). Therefore no current will be drawn.

A more modern type of stepper motor is the *bipolar* motor. We note that in the unipolar type there exist always pairs of wires, one wire for each direction of current flow. This can be simplified in the interest of economy to a single wire instead of a pair. The disadvantage is that drivers must be provided at both ends of the coil that are able to both source and sink currents. Together the coil and its two drivers form a bridge as shown in Fig. 11. For our bipolar motor experiment we use the SGS L293E chip, which contains two pairs of amplifiers. The schema to the right shows the push-pull (totem-pole) configuration of transistor pairs in each amplifier. - The same signals can be used for both kinds of motor.

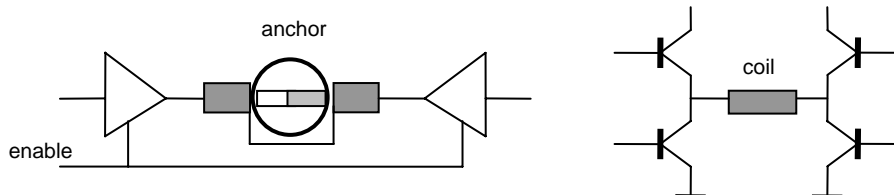


Fig. 11. Bipolar motor driver (one phase only)

#### 4.5. Running a 2-phase Motor

Our next experiment concerns the driving of a motor as it is typically used in small equipment such as disks or diskettes. Instead of in steps, they proceed continuously. They are organized similar to stepping motors, though, but have only 2 phases instead of 4, that is, there is a single bidirectional winding (coil) or two unidirectional coils.

Evidently, current has to be applied alternatively to the 2 phases. As soon as the anchor has reached the position to which it was attracted by the magnetic field, the direction of the current is switched, and the anchor moves into the other direction. This, then, results in a continuous rotation. The switch is called the alternator, and until recent times consisted of two contacts mounted on the rotor shaft and two static brushes contacting them, thereby alternating between the rotating contacts.

With the advent of power transistors it became possible to switch also high currents. The transistors are controlled by signals obtained by sensors monitoring the rotor's position. These sensors are either optical or magnetic. In this way, a brushless motor is obtained. It operates without losses of energy in the brushes, and needs no brush maintenance.

Here we use two MOSFETs (power transistors) IRF Z40, which are driven directly by the output signals of the PIC (B.4 and B.5). The sensor signal from the motor is fed to port A.3 as shown in Fig. 12. In the idle state both output must be low to cut the transistors off, and in order to be safe in the reset state with high-impedance outputs, weak pull-down resistors (4.7K) should be provided (not shown here).

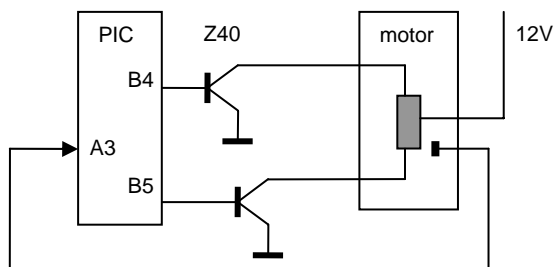


Fig. 12. 2-phase motor controller circuit

The speed of a motor is determined by the current provided. Until recent times, the current was controlled by a resistor in series with the coils. This implies that part of the energy is wasted in

those resistors. The modern method is to not vary the value of the current, but the duration of its flow, that is, not to apply a permanent current, but rather pulses. The speed is then determined by the width of the pulses, and the method is called *pulse width modulation* (PWM). A micro-controller is the ideal agent to vary the pulse width. Now that the energy sinking resistors are gone, the setup becomes considerably more efficient.

The evident idea is to turn on the current for a time determined by a given speed parameter, and then to turn it off until the sensor signals the start of the next period, at which moment the current is turned on again. However, this scheme is unstable. Assume that the turn-on time is fixed by  $t_{on}$ , and that the time until the next phase is  $t_{off}$ . The average current is then  $i_{max} * t_{on} / (t_{on} + t_{off})$ . Now suppose that this value accelerates the rotor. As  $t_{on}$  is fixed, a faster rotation causes  $t_{off}$  to decrease, and therefore the quotient and the average current and therefore the speed to further increase. This continues until the maximum possible speed is reached, independent of the initial value of  $t_{on}$ .

The difficulty is solved by letting the speed parameter be  $t_{off}$  rather than  $t_{on}$ . Each phase therefore begins with waiting  $t_{off}$  units of time with the current switched off, then turning it on until the sensor signal causes the current to be switched off again. Now an increase in speed causes  $t_{on}$  to become shorter, decreasing the current and counteracting the increase in speed. The resulting driver program is straight-forward. It consists of a loop with the two phases. It terminates when a byte is received over the communication line, i.e. when its clock signal goes low.

```
PROCEDURE TwoPhaseMotor(INT del);
  INT k;
  BEGIN
    REPEAT
      REPEAT k := del;
        REPEAT delay(50); k := k - 1 UNTIL ~A.3 OR k = 0;
        IF A.3 THEN !B.4; ?~A.3; !~B.4 END ;
        k := del;
        REPEAT delay(50); k := k - 1 UNTIL A.3 OR k = 0;
        IF ~A.3 THEN !B.5; ?A.3; !~B.5 END ;
      UNTIL ~B.6;
      del := Rec()
    UNTIL del = 0
  END TwoPhaseMotor;
```

#### 4.6. A Temperature Sensor

As a last experiment we show the use of a temperature sensor. We chose the Dallas 1620, an 8-pin DIP. It is in fact more complex than needed here, and we restrict our considerations to the single task of receiving a value indication the current ambient temperature. In this case. the value consists of 9 bits, received sequentially, LSB first.

The input signal is connected to the PIC port A.3, the clock again to B.1. In the repeat statement, A.3 is copied into  $d.7$ , which is then shifted right. The 1620 delivers the temperature in half degrees centigrade.

This chip can also serve as alarm, issuing a trigger signal when temperature exceeds or drops below a certain value. Commands are provided to load these limiting values into internal registers. Also sensing the current temperature therefore requires issuing a command. This is done by the auxiliary procedure *Tout*. It sends an 8-bit value to the 1620 sensor. This routine is also used to initialize the sensor correctly when the program is started after loading. As it uses the data line A.3 as an output, its port needs to be reconfigured. This is done in the first line of procedure *Tout* by setting A.3 to output.

```
PROCEDURE ReadTemp(): INT;
  INT d, i;
  BEGIN !A.2; TOut($AA); i := 9;
    REPEAT !~B.1; ROR d;
      IF A.3 THEN !d.7 ELSE !~d.7 END ;
      !B.1; DEC i
  END
```

```

    UNTIL i = 0;
    !~A.2; ROL d; RETURN d
END ReadTemp;

PROCEDURE TOut(INT d);
    INT i, x;
    BEGIN !~B.1; !S.5; !~A.3; !~S.5; x := d;
    i := 8;
    REPEAT !~B.1;
        IF d.0 THEN !A.3 ELSE !~A.3 END ;
        !B.1; ROR d; DEC i
    UNTIL i = 0;
    !S.5; !A.3; !~S.5
END TOut;

```

## 5. Integration of Experiments

### 5.1. Hardware

If we wish to integrate all the described experiments on a single board, we must realize that the PIC has too few pins to accommodate all devices. Two PICs are required. The signals and their chosen pin assignments are summarized as follows for the two PICs:

Serial data link	data	in	B7
	data	out	A4
	clock	out	B6
Digital to analog	data	out	B0
	clock	in	B1
	enable	out	A0
Analog to digital	data	in	A3
	clock	out	B1
	enable	out	A1
Temperature sensor	data	in/out	B0
	clock	out	B1
	enable	out	A2
LEDs	data	out	B0 – B5

---

Serial data link	data	in	B7
	data	out	A4
	clock	out	B6
Stepper motors	4 phases	out	B0 – B3
	enable	out	A0, A1
2-phase motor	2 phases	out	B4, B5
	sensor	in	A3
LED	data	out	A2

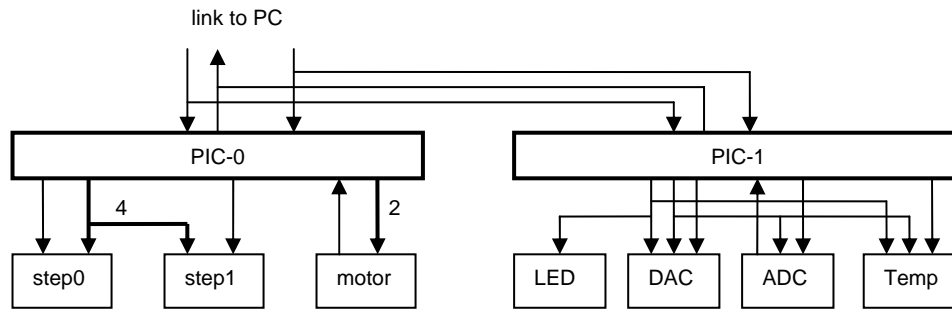


Fig. 13. Block diagram of PIC board

The connection to the host computer consists of a clock line and a data line in each direction. These lines are shared by the two processors. This implies that they would interfere if running at the same time. The following scheme is used in order to avoid this situation. When starting (after reset), programs must first sense the data line (B.7). Programs on PIC-0 must go into an idle loop, if B.7 = 0, and those running on PIC-1 enter an idle loop, if B.7 = 1. The outgoing data line is connected to port A.4, which is an open-collector output, thus allowing the two outputs to be tied together. A pull-up resistor must be provided.

And finally, a provision must be made for programming the processors, i.e. for loading programs into the EEPROM. Here the specification of the PIC determines that B.6 be the clock line, and B.7 the data line. (We therefore have chosen the same assignments for the communication in general). The exception is the outgoing data line. In programming mode, it is used for verification of loaded programs, i.e. as output. This implies that B.7 is used bidirectionally using its tri-state facility. If the PC keeps D0 high, the line signal can be read at D6 thanks to a diode. The connections are shown in Fig. 14.

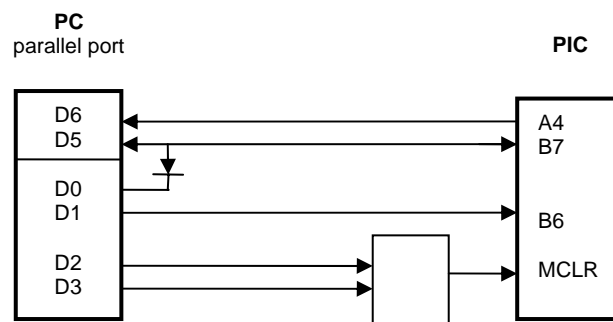
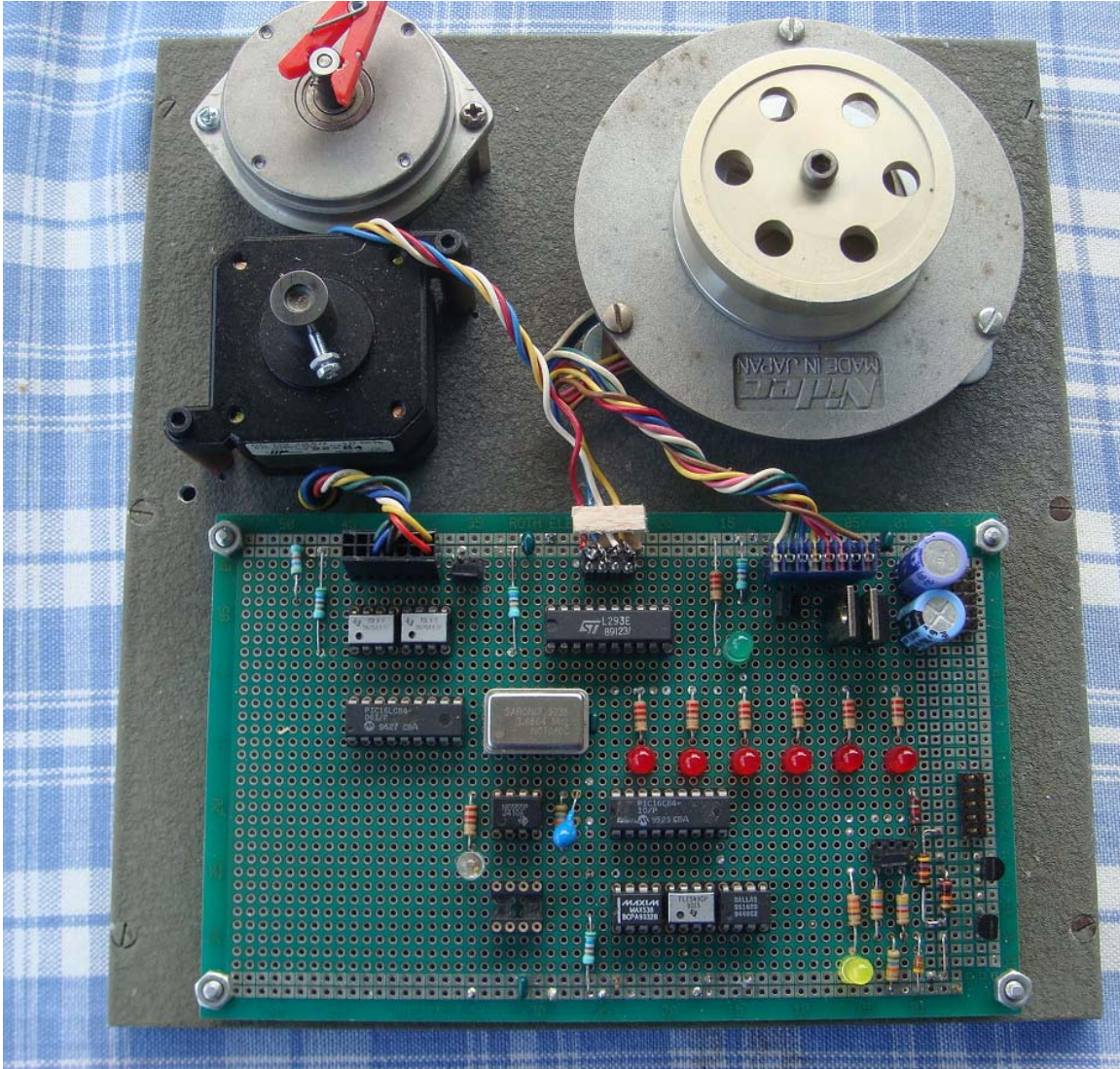


Fig. 14. Connections between PC and PICs

The MCLR (master clear, reset) signal of the PIC accepts three voltage levels, depending on the logic levels of PC outputs D2 (reset) and D3 (program):

D3 (prog)	D2 (rst)	MCLR
0	0	0V (reset)
0	1	5V (run)
1	0	12V (program)



## 5.2. Software

Integration of the drivers means that there is a common loop accepting commands from the communication line and dispatching control to the various drivers. We assume that for every command, the first byte determines the driver to be called, and subsequent bytes are parameters. The following commands are provided for the processor driving motors:

code	command	parameters
0	mirror	receive/send check
1	activate stepper forward	device, no.of steps, ontime, offtime
2	activate stepper backward	device, no.of steps, ontime, offtime
3	activate 2-phase motor	offtime
4	idle	

The parameters *ontime* and *offtime* determine the pulse width and thereby the speed.

```
INT cmd, dev, dat, on, off;
```

```
PROCEDURE Idle;
```

```
BEGIN B := $0F; A := $14;
```

```
  REPEAT !A.2; longdelay; !~A.2; longdelay UNTIL ~B.6
```

```
END Idle;
```

```

BEGIN B := $CF; A := $10;
!S.5; B := $C0; A := $08; !~S.5; (*set tri-state control*)
IF B.7 THEN
  REPEAT cmd := Rec();
    IF cmd = 0 THEN dat := Rec(); Send(dat)
    ELSIF cmd = 1 THEN dev := Rec(); dat := Rec(); on := Rec(); off := Rec();
      IF dev = 0 THEN !A.0 ELSIF dev = 1 THEN !A.1 ELSIF dev = 2 THEN !A.0; !A.1 END ;
      StepForward(dat); !~A.0; !~A.1
    ELSIF cmd = 2 THEN dev := Rec(); dat := Rec(); on := Rec(); off := Rec();
      IF dev = 0 THEN !A.0 ELSIF dev = 1 THEN !A.1 ELSIF dev = 2 THEN !A.0; !A.1 END ;
      StepBackward(dat); !~A.0; !~A.1
    ELSIF cmd = 3 THEN dat := Rec(); TwoPhaseMotor(dat)
    ELSIF cmd = 4 THEN Idle
  END
END
ELSE
  REPEAT !A.2; longdelay; !~A.2; longdelay END
END
END

```

The program for the PIC driving DAC and ADC has the same structure: an infinite loop accepting commands:

code	command	parameters
0	mirror	receive/send check
8	DA conversion	data
9	AD conversion	data (to PC)
10	sense temperature	data (to PC)
11	show counter on LEDs	-
12	show shifter on LEDs	-
13	multiply	x, y
14	divide	x, y

```

INT cmd, dat, dat1;
BEGIN B := $FF; A := $13;
!S.5; B := $C0; A := $08; !~S.5;
IF ~B.7 THEN
  !A.2; TOut(3); TOut($EE); !~A.2;
  REPEAT cmd := Rec();
    IF cmd = 0 THEN dat := Rec(); Send(dat); B := dat
    ELSIF cmd = 8 THEN dat := Rec(); DAConversion(dat)
    ELSIF cmd = 9 THEN dat := ADConversion(); Send(dat)
    ELSIF cmd = 10 THEN dat := ReadTemp(); Send(dat)
    ELSIF cmd = 11 THEN Count
    ELSIF cmd = 12 THEN Shift
    ELSIF cmd = 13 THEN dat := Rec(); dat1 := Rec(); Multiply; Send(dat1); Send(dat)
    ELSIF cmd = 14 THEN dat := Rec(); dat1 := Rec(); Divide; Send(dat); Send(dat1)
  END
END
ELSE
  REPEAT !B.0; longdelay; !~B.0; longdelay END
END
END

```

The partner program on the PC is written in Oberon and provides a very convenient environment for experimentation. This is mostly due to the possibility to declare in one and the same module an arbitrary number of procedures acting as commands. They can be activated by a mouse click on text written anywhere, for example in the window containing the program text. An example is shown below. The procedures are contained in two separate modules called *Motors* and *DAC*.

```

Motors.TestIO 55      Motors.Idle
Motors.StepForward 0 50 200 50
Motors.StepBackward 0 50 200 70
Motors.StepForward 1 100 120 120
Motors.StepBackward 1 100 70 70

```

Motors.Turn 10	Motors.SetSpeed 1	Motors.SetSpeed 50	Motors.SetSpeed 0
DAC.TestIO 85	DAC.TestIO 170~		
DAC.DAC 0	DAC.DAC 64	DAC.DAC 128	DAC.DAC 192
DAC.DAC 255			
DAC.ADC	DAC.Count	DAC.Shift	DAC.Temp
DAC.Multiply 2 5	DAC.Divide 14 5		

## 6. Conclusions

The first point to be emphasized is the importance of physical experimentation and hands-on experience when teaching about computers sensing or driving equipment. The presented setup demonstrates that with little effort and using inexpensive parts a system can be designed and built that shows how computers operate in ways and on problems that are often ignored in CS curricula.

The second point is that experimentation can be vastly enhanced by adequate software environments. Here this includes both the system and the language Oberon. They are supplemented by the language PICL for the microcontroller PIC. After many years of skepticism the author is now convinced that even a very small language can considerably ease the writing of programs, even small ones, and free the programmers of the tedious coding with assemblers. An additional benefit of the simple language is a very small compiler.

This project involved language design, compiler construction, familiarity with a microcontroller, interface design, even insider's knowhow about how to get access to the PC's parallel port. The hard lesson was, as always, that the devil hides in the details. It was fun to outwit him!