The Language PICL and its Implementation

Niklaus Wirth, 20. Sept. 2007

1. Introduction

PICL is a small, experimental language for the PIC single-chip microcomputer. The class of computers which PIC represents is characterized by a wordlength of 8, a small set of simple instructions, a small memory of at most 1K cells for data and equally much for the program, and by integrated ports for input and output. They are typically used for small programs for control or data acquisition systems, also called embedded systems. Their programs are mostly permanent and do not change.

All these factors call for programming with utmost economy. The general belief is that therefore programming in a high-level language is out of the question. Engineers wish to be in total control of a program, and therefore shy away from complex languages and compiler generating code that often is unpredictable and/or obscure.

We much sympathize with this reservation and precaution, particularly in view of the overwhelming size and complexity of common languages and their compilers. We therefore decided to investigate, whether or not a language could be designed in such a way that the reservations would be unjustified, and the language would indeed be beneficial for programmers even of tiny systems.

We chose the PIC processor, because it is widely used, features the typical limitations of such single-chip systems, and seems to have been designed without consideration of high-level language application. The experiment therefore appeared as a challenge to both language design and implementation.

The requirements for such a language were that it should be small and regular, structured and not verbose, yet reflecting the characteristics of the underlying processor. In order to understand the challenge of bridgeing the gap between high-level abstractions and the concrete architecture, we must first obtain a picture of the processor, reduced to its essentials.

2. The Architecture of the PIC processor

The PIC processor is a typical Harvard architecture, i.e. a von Neumann machine with separate memories for program and data. In this experiment, we used the PIC 16C84, which uses an internal RAM for 64 bytes of data, and an EEPROM for 2k words of program. The first 12 bytes of data memory have special functions. They are the status register, a timer, input/ouput ports, etc. There is only one true register, the W-Register (not part of the RAM), which acts as an accumulator in the ALU, and on which data instructions operate. In the following diagram (see Fig. 1) we omit the “registers” with special functions.

There is a rather small instruction set with 4 formats for

1. Byte-oriented instructions consisting of opcode and operand address:

   MOV, ADD, SUB, AND, IOR, XOR,
   DEC, INC, DECFSZ, INCFSZ (increment/decrement and skip if result is zero)

2. Byte-oriented instructions consisting of opcode and literal operand:

   MOV, ADD, SUB, AND, IOR, XOR,
   GOTO, CALL, RETURN

3. Bit-oriented instructions consisting of opcode, operand address, and bit number:

   BFS, BFC (set/clear bit)
   BTFSC, BTFSS (bit test, skip if clear/set)

4. Jump instructions with an 11-bit absolute address.

Addresses are only 7 bits long, bit numbers range from 0 to 7 (see Fig. 2).

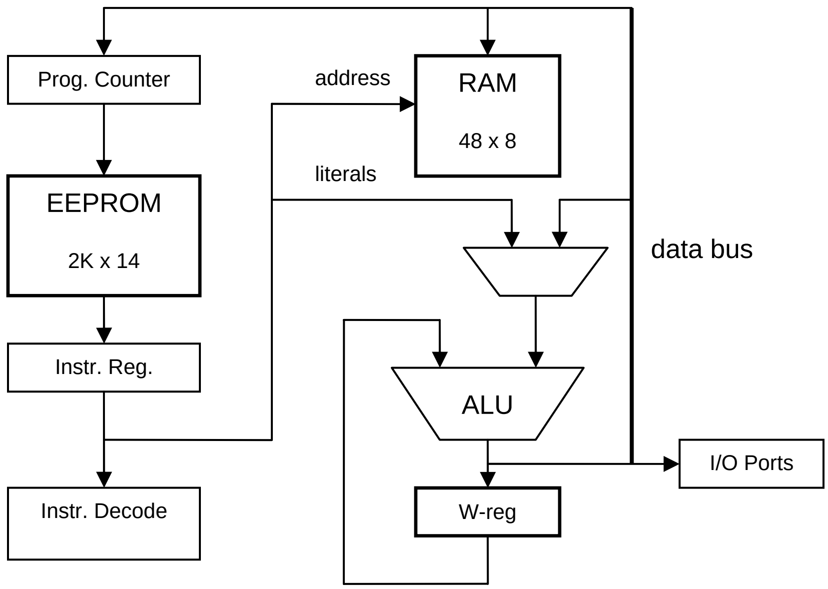

Fig. 1 The PIC architecture

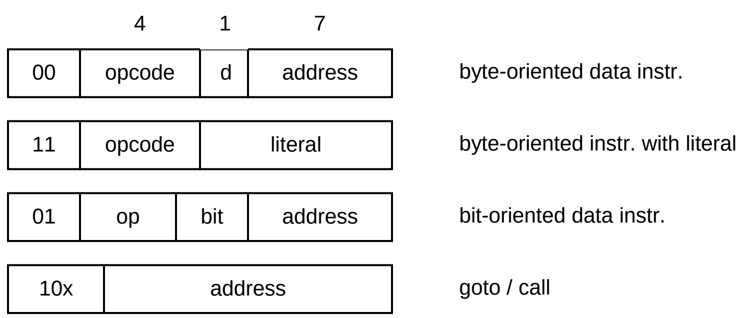

Fig. 2. PIC instruction formats

3. The Language PICL

The language is concisely defined in a separate report. Here we merely point out its particular characteristics which distinguish it from conventional languages. Like conventional languages, however, it consist of constant, variable, and procedure declarations, followed by statements of various forms. The simplest forms, again like in conventional languages, are the assignment and the procedure call. Assignments consist of a destination variable and an expression. The latter is restricted to be a variable, a constant, or an operator and its operand pair. No concatenation of operations and no parentheses are provided. This is in due consideration of the PIC's simple facilities and ALU architecture. Examples can be found in the section on code patterns below.

Conditional and repetitive statements are given the modern forms suggested by E. W. Dijkstra. They may appear as somewhat cryptic. However, given the small sizes of programs, this seemed to be appropriate.

Conditional statements have the form shown at the left and explained in terms of conventional notation to the right.

```
[cond -> StatSeq]                        IF cond THEN Statseq END
[cond -> StatSeq0 |* StatSeq1 ]          IF cond THEN Statseq0 ELSE StatSeq1 END
[cond0 -> StatSeq0 | cond1 -> StatSeq1]  IF cond0 THEN Statseq0 ELSIF cond1 THEN StatSeq1END
```

Repetitive statements have the form:

```
{cond -> StatSeq}                        WHILE cond DO Statseq END
{cond0 -> StatSeq0 | cond1 -> StatSeq1}  WHILE cond0 DO Statseq0 ELSIF cond1 DO StatSeq1END
```

There is also the special case mirroring a restricted form of for statement. Details will be explained in the section on code patterns below.

```
{| ident, xpression -> StatSeq}
```

Procedures can have at most a single (value) parameter. They can be functions with a result that can be assigned to a variable. Recursion is not allowed, and the depth of calls can be at most 8. These restrictions are a direct consequence of architectural limitations and our effort to do without complicated, hidden mechanisms, such as a call stack, local variables, etc. Whereas the syntax of PICL is to provide the conveniences of high-level languages, its semantics are to mirror the facilities and limitations of the processor clearly and honestly.

4. The PICL Compiler

The compiler consists of two modules, the scanner, and the parser and code generator. The scanner recognizes symbols in the source text. The parser uses the straight-forward method of syntax analysis by recursive descent. It maintains a linear list of declared identifiers for constants, variables, and procedures.

5. Code Patterns

In order to exhibit the correspondence between language constructs and assembler code, a sequence of short samples is listed, followed by the code generated by the compiler.

```
MODULE Assignments;
  CONST N = 10;
  INT x, z;
BEGIN z := x; z := N; z := 0;
END Assignments.

  0 0000080C     MOVFW  0  12     move x to W
  1 0000008D     MOVWF  1  13     move W to z
  2 0000300A     MOVLW    10      move 10 to W
  3 0000008D     MOVWF  1  13     move W to z
  4 0000018D     CLRF   1  13     z := 0
```

Statements operating on a single operand are called *operators*. They are denoted by an exclamation mark, and correspond to a single instruction.

```
MODULE Operators;
  BOOL b; SET s; INT x;
BEGIN !b; !~s.3;
  INC x; DEC x; ROL x; ROR x
END Operators.

  0 0000140C     BSF    0  12     !b       set b.0
  1 00001018     BCF    3  13     !~s.3    clear s.3
  2 00000A8E     INCF   1  14     INC x
  3 0000038E     DECF   1  14     DEC z
  4 00000D8E     RLF    1  14     ROL x    rotate x left via carry (S.0)
  5 00000C8E     RRF    1  14     ROR x    rotate x right via carry (S.0)
```

Statements testing a bit and waiting until the bit is set or reset are called *queries*. They are denoted by a question mark, and they are applied to elements of input ports A and B.

```
MODULE Queries;
BEGIN ?A; ?~B.3
END Queries.

  0 00001C05        BTFSS  0  5        ?A        wait until A.0 true
  1 00002800        GOTO     0
  2 00001986        BTFSC  3  6        ?~B.3     wait until B.3 false
  3 00002802        GOTO     2
```

Expressions have the simple form *x op y*. Both operands must be of the same type. The type determines the operation. For example, + for integers denoted addition, + for sets denotes logical or. If the result is assigned to the first operand, the compiler makes use of the possibility that the result of an instruction may be written to the operand instead of the W-register. This saves one instruction.

```
MODULE Expressions;
   INT x, y, z; SET u, v, w;
BEGIN z := x+3; z := y-3; z := x+y; x := x+y; z := 15-x;
   w := u + $07; w := u * $0F; w := u - v; u := u - v
END Expressions.

  0 00003003        MOVLW     3
  1 0000070C        ADDWF  0  12
  2 0000008E        MOVWF  1  14       z := x+3
  3 00003003        MOVLW     3
  4 0000020D        SUBWF  0  13
  5 0000008E        MOVWF  1  14       z := y-3
  6 0000080D        MOVFW  0  13
  7 0000070C        ADDWF  0  12
  8 0000008E        MOVWF  1  14       z := x+y
  9 0000080D        MOVFW  0  13
 10 0000078C        ADDWF  1  12       x := x+y
 11 0000080C        MOVFW  0  12
 12 00003C0F        SUBLW    15
 13 0000008E        MOVWF  1  14       z := 15-x
 14 00003007        MOVLW     7
 15 0000040F        IORWF  0  15
 16 00000091        MOVWF  1  17       w := u + $07
 17 0000300F        MOVLW    15
 18 0000050F        ANDWF  0  15
 19 00000091        MOVWF  1  17       w := u * $0F
 20 00000810        MOVFW  0  16
 21 0000060F        XORWF  0  15
 22 00000091        MOVWF  1  17       w := u - v
 23 00000810        MOVFW  0  16
 24 0000068F        XORWF  1  15       u := u - v
```

Conditions yield a truth value. They consist of comparisons and bit tests concatenated by either logical disjunctions (or), or by conjunctions (and). Here, the conditions are part of if statements of the form *IF cond THEN statement END*.

```
MODULE Conditions;
   INT x, y, z, w; SET s; BOOL b;
BEGIN
   IF x = y THEN z := 0 END ;
   IF x = y & y # z & z >= w THEN z := 0 END ;
   IF x < y OR y <= z OR z > w THEN z := 0 END
END Conditions.

  0 0000080D        MOVFW  0  13       y
  1 0000020C        SUBWF  0  12       x - y
  2 00001D03        BTFSS  2   3       = 0?  (test S.3)
  3 00002805        GOTO     5
  4 0000018E        CLRF   1  14       z := 0

  5 0000080D        MOVFW  0  13
  6 0000020C        SUBWF  0  12       x – y
```

```
 7 00001D03     BTFSS  2   3     = 0?
 8 00002812     GOTO    18
 9 0000080E     MOVFW  0  14
10 0000020D     SUBWF  0  13     y - z
11 00001903     BTFSC  2   3     #0?
12 00002812     GOTO    18
13 0000080F     MOVFW  0  15
14 0000020E     SUBWF  0  14     z - w
15 00001C03     BTFSS  0   3     >=0?
16 00002812     GOTO    18
17 0000018E     CLRF   1  14     z := 0

18 0000080D     MOVFW  0  13
19 0000020C     SUBWF  0  12     x - y
20 00001C03     BTFSS  0   3     >=0?

21 0000281E     GOTO    30
22 0000080D     MOVFW  0  13
23 0000020E     SUBWF  0  14     z - y
24 00001803     BTFSC  0   3     <0?
25 0000281E     GOTO    30
26 0000080E     MOVFW  0  14
27 0000020F     SUBWF  0  15     w - z
28 00001803     BTFSC  0   3     <0?
29 0000281F     GOTO    31
30 0000018E     CLRF   1  14     z := 0
```

Statements preceded by an if clause are called *guarded* statements. They are executed only if the guard is true.

```
MODULE IfStatements;
   INT x; BOOL p, q;
BEGIN
   IF p THEN x := 0-x END ;
   IF p THEN x := 1 ELSIF q THEN x := 2 END ;
   IF p THEN x := 3 ELSIF q THEN x := 4 ELSE x := 5 END
END IfStatements.
```

```
 0 00001C0D     BTFSS  0  13     p.0?
 1 00002805     GOTO     5
 2 0000080C     MOVFW  0  12
 3 00003C00     SUBLW     0
 4 0000008C     MOVWF  1  12     x := -x

 5 00001C0D     BTFSS  0  13     p.0?
 6 0000280A     GOTO    10
 7 00003001     MOVLW     1
 8 0000008C     MOVWF  1  12     x := 1
 9 0000280E     GOTO    14
10 00001C0E     BTFSS  0  14     q.0?
11 0000280E     GOTO    14
12 00003002     MOVLW     2
13 0000008C     MOVWF  1  12     x := 2

14 00001C0D     BTFSS  0  13     p.0?
15 00002813     GOTO    19
16 00003003     MOVLW     3
17 0000008C     MOVWF  1  12     x := 3
18 0000281A     GOTO    26
19 00001C0E     BTFSS  0  14     q.0?
20 00002818     GOTO    24
21 00003004     MOVLW     4
22 0000008C     MOVWF  1  12     x := 4
23 0000281A     GOTO    26
24 00003005     MOVLW     5
25 0000008C     MOVWF  1  12     x := 5
```

While statements are sequences of guarded statements separated by “|” and enclosed in braces.

```
MODULE WhileStatements;
```

```
   INT x, y, z; BOOL b;
BEGIN
   WHILE x # 0 DO z := z + y; x := x – 1 END ;
   WHILE x = y & ~b DO !b END ;
   WHILE x >= y OR b DO !~b END ;
END WhileStatements.

  0 0000080C       MOVFW  0  12      x
  1 00001903       BTFSC  2   3      =0?
  2 00002808       GOTO     8
  3 0000080D       MOVFW  0  13
  4 0000078E       ADDWF  1  14      z := z = y
  5 00003001       MOVLW     1
  6 0000028C       SUBWF  1  12      x := x - 1
  7 00002800       GOTO     0

  8 0000080D       MOVFW  0  13
  9 0000020C       SUBWF  0  12      x - y
 10 00001D03       BTFSS  2   3      #0?
 11 00002810       GOTO    16
 12 0000180F       BTFSC  0  15      ~b.0?
 13 00002810       GOTO    16
 14 0000140F       BSF    0  15      !b.0
 15 00002808       GOTO     8

 16 0000080D       MOVFW  0  13
 17 0000020C       SUBWF  0  12      x - y
 18 00001803       BTFSC  0   3      <0?
 19 00002816       GOTO    22
 20 00001C0F       BTFSS  0  15      b.0?
 21 00002818       GOTO    24
 22 0000100F       BCF    0  15      !~b.0
 23 00002810       GOTO    16
```

Repeat statement have their test for termination at the end and are therefore executed at least once. There is only one goto instruction jumping backward to the beginning of the repeat statement.

```
MODULE RepeatStat;
   INT x, y;
BEGIN
   REPEAT x := x + 10; y := y - 1 UNTIL y = 0;
   REPEAT DEC y UNTIL y = 0
END RepeatStat.

  0   0000300A     MOVLW    10
  1   0000078C     ADDWF  1  12      x := x + 10
  2   00003001     MOVLW     1
  3   0000028D     SUBWF  1  13
  4   0000080D     MOVFW  0  13      y := y - 1
  5   00001D03     BTFSS  2   3      = 0 ?
  6   00002800     GOTO     0

  7   00000B8D     DECFSZ  1  13     y := y – 1; = 0?
  8   00002807     GOTO     7
```

The compiler recognizes the special case, where the statement ends by decrementing a variable and then testing it for zero, as is shown by the second statement in the preceding example. In this case, subtraction, and test with skip are contractable into a single instruction DECFSZ (decrement and skip if zero). This case is recognized, however, only if decrementing is done by the DEC operator.

Procedures may have a single parameter, which is passed via the W-register, and they may have a result, which is also passed via the W-register.

```
MODULE Procedures;
   INT x, y;
```

```
PROCEDURE NofBits(INT x): INT;
   INT cnt, n;
BEGIN cnt := 0; n := 8;
   REPEAT
      IF x.0 THEN INC cnt END ];
      ROR x; DEC n
   UNTIL n = 0;
   RETURN cnt
END NofBits;

PROCEDURE Swap;
   INT z;
BEGIN z := x; x := y; y := z
END Swap;

PROCEDURE P(INT a);
BEGIN
   x := a + 10
END P;

BEGIN Swap; P(y); x := NofBits(y)
END Procedures.
```

```
 0 00002819          GOTO    25
 1 0000008E NofBits  MOVWF  1  14     x := W (parameter)
 2 0000018F          CLRF   1  15     cnt := 0
 3 00003008          MOVLW     8
 4 00000090          MOVWF  1  16     n := 8
 5 00001C0E          BTFSS  0  14     x.0?
 6 00002808          GOTO     8
 7 00000A8F          INCF   1  15     !+cnt
 8 00000C8E          RRF    1  14     !>x
 9 00000B90          DECFSZ  1  16
10 00002805          GOTO     5
11 0000080F          MOVFW  0  15     W := cnt
12 00000008          RET

13 0000080C Swap     MOVFW  0  12
14 00000091          MOVWF  1  17     z := x
15 0000080D          MOVFW  0  13
16 0000008C          MOVWF  1  12     x := y
17 00000811          MOVFW  0  17
18 0000008D          MOVWF  1  13     y := z
19 00000008          RET

20 00000092  P       MOVWF  1  18     a := W
21 0000300A          MOVLW    10
22 00000712          ADDWF  0  18
23 0000008C          MOVWF  1  12     x := a + 10
24 00000008          RET

25 0000200D          CALL    13       Swap
26 0000080D          MOVFW  0  13     W := y
27 00002014          CALL    20       P
28 0000080D          MOVFW  0  13     W := y
29 00002001          CALL     1       NofBits
30 0000008C          MOVWF  1  12     x := W
```

6. Applications

The following two procedures show how to use PIC facilities to implement multiplication and division (of 8-bit non-negative integers).

```
PROCEDURE Multiply;
   INT x, y, z, n;
BEGIN z := 0; n := 8;
   REPEAT
      IF x.0 THEN z := z+y END ;
      ROR z; ROR x; DEC n
```

```
   UNTIL n = 0
END Multiply.

zh,z := x*y              16-bit product double length register

  0 0000018E             CLRF   1  14         z := 0
  1 00003008             MOVLW     8
  2 0000008F             MOVWF  1  15         n := 8
  3 00001C0C             BTFSS  0  12         x.0?
  4 00002807             GOTO     7
  5 0000080D             MOVFW  0  13
  6 0000078E             ADDWF  1  14         z := z + y
  7 00000C8E             RRF    1  14         !>z  rotate via carry
  8 00000C8C             RRF    1  12         !>x
  9 00000B8F             DECFSZ  1  15
 10 00002803             GOTO     3
```

```
PROCEDURE Divide;
   INT r, q, d, n;
BEGIN r := 0; n := 8;
   REPEAT ROL q; ROL r;
      IF r >= d THEN r := r - d; INC q END
      DEC n
   UNTIL n = 0
END Divide.

q := r DIV d; r := r MOD d;  r,q form a double length register

  0 0000018C             CLRF   1  12         r := 0
  1 00003008             MOVLW     8
  2 0000008F             MOVWF  1  15         n := 8
  3 00000D8D             RLF    1  13         !<q
  4 00000D8C             RLF    1  12         !<r
  5 0000080E             MOVFW  0  14
  6 0000020C             SUBWF  0  12         r - d
  7 00001C03             BTFSS  0   3         <0?
  8 0000280C             GOTO    12
  9 0000080E             MOVFW  0  14
 10 0000028C             SUBWF  1  12         r := r - d
 11 00000A8D             INCF   1  13
 12 00000B8F             DECFSZ  1  15
 13 00002803             GOTO     3
```

The following procedures serve for sending and receiving a byte. Transmission occurs over a 3-wire connection, using the conventional hand-shake protocol. Port A.3 is an output. It serves for signaling a request to receive a bit. Port B.6 is an input and serves for transmittithe data. B.7 is usually in the receiving mode and switched to output only when a byte is to be sent. In the idle state, both request and acknowledge signals are high (1).

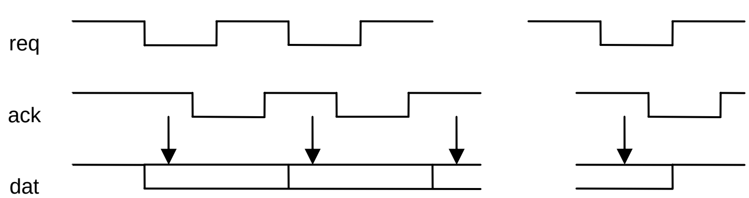

Fig. 3. Transmission protocol

```
PROCEDURE Send(INT x);
   INT n;
BEGIN ?B.6;                              wait for ack = 1
   !S.5; !~B.7; !~S.5; n := 8;           switch B.7 to output
   REPEAT
      IFx.0 -> !B.7 ELSE !~B.7 END ;     apply data
```

```
      !~A.3;                              issue request
      ?~B.6;                              wait for ack
      !A.3; ROR x;                        reset req, shift data
      ?B.6; DEC n                         wait for ack reset
   UNTIL n = 0;
   !S.5; !B.7; !~S.5                      reset B.7 to input
END Send;

PROCEDURE Receive;
   INT n;
BEGIN d := 0; n := 8;                     result to global vaiable d
   REPEAT
      ?~B.6; ROR d;                       wait for req
      IF B.7 THEN !d.7 ELSE !~d.7 END ;   sense data
      !~A.3;                              issue ack
      ?B.6;                               wait for req reset
      !A.3; DEC n                         reset ack
   UNTIL n = 0
END Receive;
```

Another version of the same procedures also uses three lines. But it is asymmetric: There is a master and a slave. The clock is always delivered by the master on B.6 independent of the direction of the data transmission on A3 and B7.

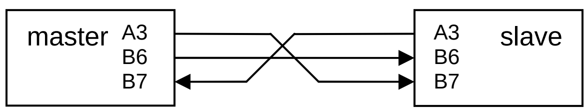

Fig. 4. Master-slave configuration

When sending, the data is applied to A.3, when receiving, the data is on B.7. The advantage of this scheme is that no line ever switches its direction, the disadvantage is its dependence on the relative speeds of the two partners. The clock must be sufficiently slow so that the slave may follow. There is no acknowledgement.

```
Master                                  Slave

PROCEDURE Send(INT x);                  PROCEDURE Receive;
   INT n;                                  INT n;
BEGIN n := 8;                           BEGIN d := 0; n := 8;                 result to global vaiable d
   REPEAT                                  REPEAT ?~B.6; !>d;                 wait for clock low
     IF x.0 THEN !A.3 ELSE !~A.3 END;        IF B.7 THEN !d.7 ELSE ~d.7 END;  sense data
     !~B.6; !>x; !B.6; DEC n                ?B.6; DEC n                       wait for clock high
   UNTIL n = 0                             UNTIL n = 0
END Send;                               END Receive;

PROCEDURE Receive;                      PROCEDURE Send(INT x);
   INT n;                                  INT n;
BEGIN d := 0; n := 8;                   BEGIN n := 8;
   REPEAT !~B.6; ROR d;                    REPEAT ?~B.6;                      wait for clock low
     IF B.7 THEN !d.7 ELSE ~d.7 END;         IF x.0 THEN !A.3 ELSE !~A.3 END; apply data
     !B.6; DEC n                             ROR x ?B.6; DEC n                wait for clock high
   UNTIL n = 0                             UNTIL n = 0
END Receive;                            END Send;
```

7. Conclusions

The motivation behind this experiment in language design and implementation had been the question: Are high-level languages truly inappropriate for very small computers? The answer is: Not really, if the language is designed in consideration of the stringent limitations. I justify my answer out of the experience made in using the language for some small sample programs. The corresponding assembler code is rather long, and it is not readily understandable. Convincing oneself of its correctness is rather tedious (and itself error-prone). In the new notation, it is not easy either, but definitely easier due to the structure of the text.

In order to let the regularity of this notation stand out as its main characteristic, completeness was sacrificed, that is, a few of the PIC's facilities were left out. For example, indirect addressing, or adding multiple-byte values (adding with carry). Corresponding constructs can easily be added.

One might complain that this notation is rather cryptic too, almost like assembler code. However, the command (!) and query (?) facilities are compact and useful, not just cryptic. Programs for computers with 64 bytes of data and 2K of program storage are inherently short; their descriptions should therefore not be longwinded. After my initial doubts, the new notation appears as a definite improvement over conventional assembler code.

The compiler was written in the language Oberon. It consists of a scanner and a parser module of 2 and 4 pages of source code respectively (including the routines for loading and verifying the generated code into the PIC's ROM). The parser uses the time-honored principle of top-down, recursive descent. Parsing and gode generation occur in a single pass.