PICL: A Programming Language for the Microcontroller PIC

Niklaus Wirth, Feb. 2005, rev. Oct. 2007

0. Introduction

PIC is the name of a single-chip microcontroller designed and fabricated by Microchip Inc. It features an ALU with basic operations for arithmetic, a 128 byte data memory, a 2048 word program memory, and 2 I/O ports (PIC 16C84). Here we present a programming language that is tailored to the PIC’s size and architecture. The challenge lay in postulating a design allowing to abstract from the peculiarities of the particular architecture and overcome the tedium of coding instruction by instruction with an assembler, and yet of letting the processor’s facilities be sufficiently transparent, that no excessively inefficient programs will be produced.

The language’s syntax is formally defined in terms of Extended BNF. Short examples of most syntactic constructs are provided to help explaining their meanings and to illustrate the use and structure of the language.

1. Symbols, Identifiers, Numbers and Comments

A program in the language PICL is a text to be considered as a sequence of symbols, identifier, and numbers. Blanks and line breaks are irrelevant. Identifiers are sequences of letters and digits. The first character must be a letter. Numbers are (unsigned) decimal integers. Symbols are special characters or character pairs, or reserved words. No blanks must occur within identifiers, numbers, and reserved words.

Constants, variables, and procedures are denoted by identifiers freely chosen by the programmer. These identifiers are specified by declarations.

```
ident  =  letter {letter | digit).
integer  =  digit {digit}.
hexdig  =  digit | “A“ | “B“| “C“| “D“| “E“| “F“.
set  =  “$“ hexdig hexdig.
constant  =  integer | set.
```

```
reserved words  =  BEGIN | END | INT | SET | BOOL | OR |
    INC | DEC | ROL | ROR |
    IF | THEN | ELSE | ELSIF | WHILE | DO | REPEAT | UNTIL |
    CONST | PROCEDURE | RETURN | MODULE.
symbol  =  + | - | * | / | & | = | # | < | > | <= | >= |
      ! | ? | ~ | , | ; | | | ( | ) | [ | ] | { | }| := | ->.
```

2. Data Types and Variable Declarations

There are three data types in PICL, and every variable is of a specific data type. Examples of declarations of variables are

```
INT x, y                BOOL b                SET u, v
```

The type INT denotes (unsigned) integers in the range 0 to 255. The type BOOL denotes truth values. Boolean variables are also called *flags*. The type SET denotes a set of 8 truths values, numbered with an *index* in the range of 0 to 7.

```
type  =  INT | SET | BOOL.
VariableDeclaration  =  type ident {“,” ident).
```

3. Expressions

Expressions denote either a variable, a constant, or a computed value. The operators are addition, subtraction for integers, and logical operators for sets. Expressions are said to be *evaluated*, and the resulting value is of type INT or SET. Examples:

x	x + 5	x – y	
u * v	u + v	u - v	NofBits(x)

When applied to operands of type INT, + and – denote addition and subtraction respectively. When applied to operands of type SET or BOOL, + denotes inclusive disjunction (or), * denotes conjunction (and), - denotes exclusive disjunction (xor).

```
expression  =  (ident | constant) [operator (identifier | constant)]  |
        ident “(“ [expression] “)”.
operator  =  “+” | “-“ | “&”.
```

An expression may be a function call. If the function has a parameter, the actual (value) parameter is enclosed in parentheses.

4. Conditions

A condition represents a computed truth value. It consists of either a conjunction or a disjunction of terms, and a term is either a Boolean variable or its negation, or a comparison. An asterisk stands for True. Examples:

```
x > y          x = 0              b             ~u.2
x >= 0  &  x < 10                 x = 1 OR y = 1
```

The symbol “~” denotes negation, “#” inequality.

```
term  =  ident relation (ident | constant) | [“~”] ident [“.” index].

relation  =  “=” | “#” | “<” | “>” | “<=” | “>=”.
index  =  integer.
conjunction  =  term {“&” term}.
disjunction  =  term {OR disjunction}.
condition  =  conjunction | disjunction.
```

5. Statements

Statements denote actions, and are said to be *executed*. There are simple statements and composite statements. The latter consist of components which are statements themselves. Simple statements are assignments, procedure calls, and commands operating on a variable. Assignments consist of an expression and a variable. Their execution consists of the evaluation of the expression and the assignment of the resulting value to the variable. Examples of assignments are:

```
x := y          y := 0          x := x + 1       u := u * v
```

Procedure calls consist of the identifier denoting the procedure to be activated, possibly followed by a parameter:

```
Output(100)
```

Examples of commands and queries acting on Boolean variables are:

! A.0	set A.0 to true
! ~B.3	set B.3 to false
? A.0	wait until A.0
? ~B.5	wait until not B.5

The queries are meaningful only if the operand is a port variable representing an input signal. The two ports of the PIC controller are predefined and denoted by A and B.

Examples of commands acting on integer variables are:

INC x	increment x (by 1)
DEC x	decrement x (by 1)
ROL x	rotate x by 1 bit via carry (S.0) to the left
ROR x	rotate x by 1 bit via carry (S.0) to the right

Note: The commands INC x and DEC x are not equivalent with the assignments x := x+1 and x := x-1, because the former do not affect the status flag S.0 (carry).

```
assignment  =  identifier “:=” expression.
call  =  identifier [“(“ expression “)”].
command  =  “!” [“~”] ident [“.” index] |
     (INC | DEC | ROL | ROR) ident.
query  =  “?” [“~”] ident [“.” index].
```

There are three forms of composite statements. The if statement expresses conditional execution of its component(s), the while and repeat statements express repeated execution.

Examples of if statements are:

```
IF x = y THEN z := 0 END
IF x > y THEN z := x; x := y; y := z END
IF x < y THEN z := y ELSIF y < x THEN z := x ELSE z := 0 END
```

Examples of while statements are:

```
WHILE x > 0 DO z := z + y; x := x-1 END
WHILE x > y DO x := x - y ELSIF y > x DO y := y – x END
```

An example of a repaet statement is:

```
REPEAT z := z - y; INC x UNTIL z < y
```

The syntax of statements is:

```
guardedStat  =  condition -> statseq.
ifstat  =  IF condition THEN StatSeq
     {ELSIF condition THEN StatSeq}
     [ELSE StatSeq] END .
whilestat  =  WHILE condition DO StatSeq
     {ELSIF condition DO StatSeq} END .
repeatstat  = REPEAT statseq (UNTIL condition | END).
StatSeq  =  statement {“;” statement}.
statement  =  assignment | call | command | query | ifstat | whilestat | repeatstat.
```

A repeat statement without a termination condition denotes a repetition forever.

6. Procedure Declarations

Procedures, also called subroutines, are sequences of statements that can be activated by procedure calls. A procedure may have a single parameter. It may have local variables that are visible only within the procedure. If a procedure has a result, it is called a function. Functions can be used in assignments. Their result type is indicated by the type symbol in the heading of the procedure, and the result is specified at its end.

Examples of a procedure and a function declaration are:

```
PROCEDURE Send(INT x);  {B.6 = clock, B.7 = data}
     INT n;
BEGIN n := 8;
     REPEAT !~B.6;
          IF x.0 THEN !B.7 ELSE !~B.7 END ;
          !B.6; ROR x; DEC n
     UNTIL n = 0
END Send
```

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

The syntax of procedure declarations is

```
ProcedureDeclaration  =  ProcedureHeading ";" ProcedureBody.
ProcedureHeading  = PROCEDURE ident ["(" FormalParameter ")"] [“:” type].
ProcedureBody  = [{VariableDeclaration ";"}]
      [BEGIN statseq][RETURN expression] END [ident].
FormalParameter  =  type ident.
```

The declaration of a procedure must textually precede any calls of the procedure. Therefore, recursion cannot occur. If the heading contains the specification of a result type for a function procedure, a return statement at the end of the body must specify the result.

7. Modules

An entire PICL program is called a module. It consists of a heading, specifying its name, constant and variable declarations, procedure and function declarations, and a sequence of statements, the main body.

```
Module  =  MODULE ident ";"
    [CONST {ident "=" constant ";"}]
    [type {ident {"," ident} ";"}]
    {ProcedureDeclaration ";"}
    [BEGIN statseq] END ident ".".
```

The following sample program rotates a single zero bit among the flags x.0 – x.7.

```
MODULE Rotate;
    CONST delay = 250;
    INT x, d0, d1;
BEGIN x := 1;
    !S.5; B := 0; !~S.5;
    REPEAT ROL x; B := x; d1 := delay;
        REPEAT DEC d1; d0 := delay;
            REPEAT DEC d0
            UNTIL d0 = 0;
            DEC d1
        UNTIL d1 = 0
    END
END Rotate.
```

Note: The first line of the procedure body serves to configure the controller. A, B, S are predeclared identifiers. A and B denote the controller’s input and output ports, and S denotes the controller’s status register. Every bit of A and B must be configured, and is initially set to input. In this example, all 8 ports B.0, B.1, … B.7 are set to output. The inner repetitions serve as delays. Assuming that LEDs are connected to the ports, the lights rotate and indicate the speed of the process.

The Syntax of PICL

```
ident  =  letter {letter | digit}.
integer  =  digit {digit}.
digit  =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
hexdig  =  digit | “A“ | “B“ | “C“ | “D“ | “E“ | “F“.
set  =  “$“ hexdig hexdig.

constant  =  integer | set.
type  = INT | SET | BOOL.
VariableDeclaration  =  type ident {"," ident}.
```

```
expression  = (ident | integer) [operator (ident | integer)] |
        ident "(" [expression] ")".
operator  =  ("+" | "-" | "*").
relation  = "=" | "#" | "<" | "<=" | ">" | ">=".
Bterm  =  ident relation (ident | integer) | [“~”] ident [“.” index].
index  =  integer.
conjunction  =  Bterm ["&" conjunction].
disjunction  =  Bterm ["OR" disjunction].
condition  =  conjunction | disjunction | “*”.

assignment  =  ident ":=" expression.
call  =  ident ["("expression")"].
command  =  "!" (["~"] ident ["." index])  |  (INC | DEC | ROL | ROR)  ident.
query  =   "?" ["~"] ident ["." index]).
IfStatement  =  IF condition THEN StatSeq
        {ELSIF condition THEN StatSeq}
        [ELSE StatSeq] END .
WhileStatement  = WHILE condition DO StatSeq
        {ELSIF condition DO StatSeq} END .
RepeatStatement  = REPEAT StatSeq (UNTIL condition | END).
StatSeq  =  statement {";" statement}.
statement  =  [assignment | call | command | query | IfStatement | WhileStatement | RepeatStatement].

ProcedureDeclaration  =  ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading  =  PROCEDURE ident ["(" FormalParameter ")"] [":" type].
ProcedureBody  = [{VariableDeclaration} ";"][BEGIN StatementSequence] [RETURN expression] END ident.
FormalParameter  =  type ident.
Module  =  MODULE ident ";"
        [CONST {ident "=" constant ";"}]
        [type {ident {"," ident} ";"}]
        { ProcedureDeclaration ";"}
        [BEGIN StatementSequence] END ident ".".
```