Here's the test source code:

~~~~
```
a = 2
a = -2
serout(porta,300,1,a)
for a = 1 to 2
 for b = 1 to 2
  for c = 1 to 2
   for d = 1 to 2
   pause_ms(1)
   next
  next
 next
next
for a = 1 to 100
  b = 1
  pause_ms(100)
  nap(100)
  sleepmode(100)
next
for i = 1 to 10 step 2
 while a = a
   a=eeread(10)
   eewrite(10,a)
 wend
next
repeat
  a=2
  if (a=2 OR b=2) AND NOT a=0 THEN a=2
  for i = 1 to 2
    pulseout(porta,1,100,a)
    a = pulsein(porta,1,100)
  next
until a >= 0
a = rctime(porta,1)
a = random()
a = input(porta,a)
output(portb,a)
portdira(10)
portdira(%00000101)
portdirb(a)
a = a << 1
a = a >> 7
gosub calculate
sound(100,b*5/2+3-1)
a = serinp(portb,300,1)
calculate:
a = 2
if a = 2 then b = a
return
IF (a=-0) AND (b=0) then c = (2-3)*4
a=0
b=0
c=0
```
~~~~

```
END
~~~
```
Here's the compiler source code:
```
{ BASIC Compiler for PIC16C84 micro-controllers                © D L Bird
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~~~~~~
  Version 1.0 - Started 14.05.98
          1.1 - Added expression evaluator
          1.2 - Added conditional-evaluator
          1.3 - Added major functions
  Version 1.4 - Added I/O functions
          1.5 - Corrected fault with GOTOZ instruction
                Value of TRUE changed from 01 to FF
                Added check for too-many GOSUBs at run-time
                Added check for missing program END statement
                Added EEREAD and EEWRITE procedures
                Added PULSEIN and PULSEOUT procedures
                Added internal range checking function
          1.6 - Added PORTDIRA and PORTDIRB procedures
                Corrected fault with MODULUS and bitOR definitions
                Source ON/OFF display switch removed
                Corrected EEWRITE fault, using address not address contents
          1.7 - Added check for illegal FOR-NEXT variable name
                Corrected variable and for-next assignment addresses
                Corrected too-many variables error fault
                Changed Baud-rate constants 0=300..3=2400 for an 8-bit push
                Added check on COUNT for valid target variable
                Improved EEREAD/EEWRITE error checking
                Corrected fault with "strings" being forced to lower-case
          1.8 - Added SHIFT-LEFT and SHIFT-RIGHT operators
                Added RCTIME procedure
                (C=0, then charge C-R counting fixed duration pulses needed)
                Added binary representation to assignments eg. %10101010
                Added port_pin internal test procedure to reduce source
                Removed valid_port/pin checks, now in port_pin_test
                Corrected COUNT fault, pushed address contents not address
                Corrected RANDOM fault, pushed address contents not address
                Corrected fault with logical operator look-ahead
          1.9   Correct fault with comment routine at End-Of-File
                Reduced complexity of various procedures
                Improved clarity of procedure names
                Fixed problem of not recoginising unary +
                Fixed problem of IF labels, reset to 1, problem with gotos
                Moved variable bas addresses in range of actual memory
                Corrected fault with branch testing BTFSC changed to BTFSS
          2.0   Added NAP, SLEEP, removed COUNT function as capability
                provided by PULSEIN
          2.1   Corrected fault with FOR-NEXT no STEP resulting in undefined
                STEP value, now set to 1. Corrected fault with FOR-NEXT
                variable space checking.
  Looping
  ---------
  FOR variable = <expression> TO <expression> [STEP] <expression>
    <statements>
  NEXT
  WHILE <expression>
    <statements>
  WEND
  REPEAT
    <statements>
  UNTIL <expression>
```

```
  Branching                                Operators
  ---------                                ---------
  GOSUB  <label>                           AND (&), OR (@|), XOR (^), NOT (_)
  RETURN <from subroutine>                 +, -, *, /, \, NEGATE, (, )
  GOTO   <label>                           =, <> (#), >, <, >= (%), <= ($)
  IF <expression> THEN <statement>         SHIFTL(®), SHIFTR(¯)
                                           Where \ = Modulus of division

  I/O                                      General
  ---                                      -------
  PRINT<expression>                        RANDOM
  OUTPUT(port,pin)
  INPUT(port,pin)
  PULSEIN(port,pin,period)
  PULSEOUT(port,pin,period,variable|constant)
  EEREAD(address,variable)
  EEWRITE(address,variable|constant)
  PORTDIRA(variable|constant)
  PORTDIRB(variable|constant)
  RCTIME(port,pin)
  NAP(variableÝconstant)
  SLEEPMODE(variableÝconstant)

  Serial I/O                               Sound
  ----------                               -----
  SEROUT(port,baud,pin,<expression>);      SOUND(duration,frequency)
  SERINP(port,baud,pin) - function
  (Baud=0=300, 1=600, 2=1200, 3=2400)

  Delay                                    Debug
  -----                                    -----
  PAUSE_MS(<expression>)                   END <of program>
}
PROGRAM picomp21 (INPUT,OUTPUT,source);
USES CRT;

CONST
   max_var  = 7;  max_gosub   = 6;  max_for    = 3; for_start = 20;
   constant = -1; unknown_var = -2; goto_label = -3; label_text = -1;
   etx = ^C; nl = ^J; sp = ' '; leftcomment  = '{'; rightcomment = '}';
   bitOR   = '@'; bitAND = '&'; bitXOR   = '^'; bitNOT   = '~';
   NEGATE  = '_'; NOTEQU = '#'; LSTHANEQ = '$'; GRTHANEQ = '%';
   MODULUS = '\'; SHIFTL = '®'; SHIFTR   = '¯';

TYPE
   for_vtype = RECORD
               used          : BOOLEAN;
               name          : string;
               addr_low      : LONGINT;
               addr_high     : LONGINT;
               step          : LONGINT
            END;

   var_type = RECORD
              used   : BOOLEAN;
              name   : string;
              vaddr  : LONGINT;
            END;

    error_type = (noequals,nosuchvar,nofor,noto,misingnext,mislabel,
                  nobracket,nocomment,mispara,nomorevars,nomorefors,
```

```
                  nomoregosubs,mistake,misingthen,nowhile,norepeat,
                  misgosub,portsyntax,pinsyntax,range,onlyint,onlyvar,
                  missingend,addrange,binary);

    charset = SET OF CHAR;

    stackpointer   = ^stackcomponent;
    stackcomponent = RECORD
                       sc   : CHAR;
                       next : stackpointer
                     END;

VAR
   source    : TEXT;
   ch, x     : CHAR;
   numbers, capitalletters, smallletters, letters, alphanumeric, unaries,
   operators,
   binaries  : charset;
   token,
   line      : string;
   line_no, rep_add, if_add, for_add, while_add, gosubs, var_add,
   ptr,pc    : LONGINT;
   for_loop  : ARRAY [1..max_var+1] OF for_vtype;
   variable  : ARRAY [1..max_var+1] OF var_type;
   stack,
   operstack : stackpointer;

PROCEDURE expression(pos : LONGINT);FORWARD;
PROCEDURE statement;FORWARD;

PROCEDURE initialise;
VAR i : LONGINT;
BEGIN
  CLRSCR;
  FOR i := 1 TO max_var DO
  BEGIN
    WITH for_loop[i] DO
    BEGIN
      used := FALSE;
      name := '';
      addr_low  := for_start-3+i*3;
      addr_high := addr_low + 1;
      step      := addr_high + 1
    END;
    WITH variable[i] DO
    BEGIN
      used  := FALSE;
      name  := '';
      vaddr := 0;
    END
  END;
  stack         := NIL;
  operstack     := NIL;
  var_add       := 18;
  line_no       := 0;
  for_add       := 0;
  while_add     := 0;
  if_add        := 0;
  rep_add       := 0;
  gosubs        := 0;
  pc            := 0;
```

```
  numbers       := ['0'..'9'];
  binaries      := ['0','1'];
  capitalletters:= ['A'..'Z'];
  smallletters  := ['a'..'z'];
  unaries       := ['-','+'];
  operators     := ['(',')','+','-','*','/'];
  operators     := operators + [MODULUS,bitAND,bitOR,bitXOR,bitNOT];
  letters       := capitalletters + smallletters;
  alphanumeric  := letters + numbers;
END;

PROCEDURE push(sch : CHAR);
VAR entry : stackpointer;
BEGIN
  NEW(entry);
  WITH entry^ DO
  BEGIN
    sc := sch;
    next := stack
  END;
  stack := entry
END;

FUNCTION pull:CHAR;
VAR old_entry : stackpointer;
    sch       : CHAR;
BEGIN
  old_entry := stack;
  sch := old_entry^.sc;
  stack := old_entry^.next;
  DISPOSE(old_entry);
  pull := sch
END;

PROCEDURE operpush(sch : CHAR);
VAR entry : stackpointer;
BEGIN
  NEW(entry);
  WITH entry^ DO
  BEGIN
    sc := sch;
    next := operstack
  END;
  operstack := entry
END;

FUNCTION operpull:CHAR;
VAR old_entry : stackpointer;
    sch       : CHAR;
BEGIN
  old_entry := operstack;
  sch := old_entry^.sc;
  operstack := old_entry^.next;
  DISPOSE(old_entry);
  operpull := sch
END;

PROCEDURE error(err : error_type);
BEGIN
  WRITELN(' ':ptr+6,'^');
  WRITE('Error - at line ',line_no,', ',chr(7));
```

```
  CASE err OF
    noto         : WRITELN('Missing TO');
    noequals     : WRITELN('Missing =');
    nosuchvar    : WRITELN('No Such Variable');
    misingnext   : BEGIN
                     WRITELN('No matching NEXT associated with FOR ');
                     WRITELN(for_loop[for_add].name,' = ...')
                   END;
    nofor        : WRITELN('No matching FOR statement');
    nowhile      : WRITELN('No matching WHILE statement');
    norepeat     : WRITELN('No matching REPEAT statement');
    misgosub     : WRITELN('No target for GOSUB statement');
    nocomment    : WRITELN('No matching } to complete comment');
    mispara      : WRITELN('Missing parameter');
    misingthen   : WRITELN('Missing THEN');
    mislabel     : WRITELN('Missing LABEL');
    nomorevars   : WRITELN('No more VARIABLE space');
    nomorefors   : WRITELN('Warning no more FOR-LOOP variable space');
    nomoregosubs : WRITELN('Warning GOSUB stack space limited at Runtime');
    mistake      : WRITELN('Mistake !');
    nobracket    : WRITELN('Missing bracket');
    portsyntax   : WRITELN('Wrong PORT identity');
    pinsyntax    : WRITELN('Wrong PIN identity');
    range        : WRITELN('Value out of range');
    addrange     : WRITELN('Value out of address range');
    onlyint      : WRITELN('Only integers allowed');
    onlyvar      : WRITELN('Only variables allowed');
    missingend   : WRITELN('Missing END statement');
    binary       : WRITELN('Error in binary representation');
  END;
  ch := nl
END;

PROCEDURE skip_spaces;
BEGIN
  WHILE line[ptr] = sp DO ptr := ptr + 1
END;

PROCEDURE read_line;
VAR i : LONGINT;
BEGIN
  ptr := 1;
  IF NOT EOF(source) THEN
  BEGIN
    line_no := line_no + 1;
    READLN(source,line)
  END ELSE ch := etx;
  FOR i := 1 TO length(line) DO IF line[i] IN capitalletters
   THEN line[i] := CHR(ORD(line[i]) + ORD('a') - ORD('A'));
  WRITELN('; *** : ',line)
END;

PROCEDURE comment;
VAR found : BOOLEAN;
BEGIN
  found := FALSE;
  WHILE NOT (EOF(source) OR found) DO
  BEGIN
    REPEAT
      IF line[ptr] = rightcomment THEN found := TRUE;
      ptr := ptr + 1
```

```
    UNTIL found OR (ptr > length(line));
    IF NOT found THEN read_line;
  END
END;

FUNCTION nextsymbol:string;
VAR text : string;
BEGIN
  text := '';
  skip_spaces;
  IF line[ptr] IN ['=',','] THEN
  BEGIN
    text := line[ptr];
    ptr := ptr + 1
  END
  ELSE
  IF line[ptr] = leftcomment THEN comment
  ELSE
  WHILE (ptr <= length(line)) AND NOT (line[ptr] IN ['=',',','(',')',sp]) DO
  BEGIN
    ch := line[ptr];
    IF ch IN capitalletters THEN ch := CHR(ORD(ch) + ORD('a') - ORD('A'));
    text := text + ch;
    ptr := ptr + 1
  END;
  x := readkey;
  nextsymbol := text
END;

FUNCTION sym_to_baud(baud : string):string;
BEGIN
  sym_to_baud := '2'; {in case of error}
  IF baud = '300'  THEN sym_to_baud := '0';
  IF baud = '600'  THEN sym_to_baud := '1';
  IF baud = '1200' THEN sym_to_baud := '2';
  IF baud = '2400' THEN sym_to_baud := '3';
END;

FUNCTION valid_num(num : string):BOOLEAN;
VAR i : LONGINT;
BEGIN
  valid_num := TRUE;
  IF num <> '' THEN
  BEGIN
    FOR i := 1 TO length(num) DO
     IF NOT (num[i] IN numbers) THEN valid_num := FALSE
  END ELSE valid_num := FALSE
END;

FUNCTION valid_baud(baud : string):BOOLEAN;
BEGIN
  valid_baud := FALSE;
  IF valid_num(baud) THEN
  BEGIN
    IF (baud = '300') OR (baud = '600') OR (baud = '1200') OR
       (baud = '2400') THEN valid_baud := TRUE
  END ELSE error(mistake)
END;

FUNCTION found(temp : string):BOOLEAN;
BEGIN
```

```
  found := FALSE;
  IF length(line)-length(temp) > 0 THEN
    IF pos(temp,line) = ptr THEN found := TRUE
END;

PROCEDURE call(target : string);
BEGIN
  pc := pc + 1;
  WRITELN('':20,'call ',target)
END;

PROCEDURE return;
BEGIN
  pc := pc + 1;
  WRITELN('':20,'return')
END;

PROCEDURE push_val(txt_val : string; num_val : INTEGER);
VAR instr : string;
BEGIN {push literal value}
  pc := pc + 1;
  IF txt_val <> '' THEN WRITELN('':20,'movlw ',txt_val)
  ELSE WRITELN('':20,'movlw ',num_val);
  call('push')
END;

PROCEDURE push_cont(txt_val : string; num_val : INTEGER);
VAR instr1,instr2 : string;
BEGIN {push contents of an address}
  pc := pc + 1;
  IF txt_val <> '' THEN WRITELN('':20,'movf ',txt_val,',w')
   ELSE WRITELN('':20,'movf ',num_val,',w');
  call('push')
END;

PROCEDURE goto_x(txt_val : string; num_val : INTEGER);
BEGIN
  pc := pc + 1;
  IF num_val = label_text THEN WRITELN('':20,'GOTO ',txt_val)
    ELSE WRITELN('':20,'goto ',txt_val,num_val)
END;

PROCEDURE gotoz(label_name : string; label_val : INTEGER);
BEGIN
  call('pull');  {get the status of the compare 0=FALSE FF=TRUE}
  WRITELN('':20,'btfsc status,z'); {keep doing loop while TRUE}
  WRITELN('':20,'goto ',label_name,label_val); {Skip this (end) if FALSE}
  pc := pc + 2
END;

PROCEDURE label_op(txt_val : string; num_val : INTEGER);
BEGIN
  IF num_val = label_text THEN WRITELN(txt_val)
   ELSE WRITELN(txt_val,num_val)
END;

PROCEDURE push_str(txt_val : string);
VAR i  : LONGINT;
    ch : CHAR;
BEGIN {push text string, prefixed by CR, print until CR is reached}
  push_val('',13);
```

```
  FOR i := 1 TO LENGTH(txt_val) DO push(txt_val[i]);
  WHILE stack <> NIL DO
  BEGIN
    ch := pull;
    WRITELN('':20,'movlw ','''',ch,'''');
    call('push');
  END
END;

FUNCTION sym_to_oper(ch : CHAR):string;
BEGIN
  CASE ch OF
  bitOR    : call('or');
  bitXOR   : call('xor');
  bitAND   : call('and');
  '='      : call('equal');
  '>'      : call('grthan');
  '<'      : call('lsthan');
  LSTHANEQ : call('lsthaneq');
  GRTHANEQ : call('grthaneq');
  NOTEQU   : call('notequal');
  '+'      : call('add');
  '-'      : call('sub');
  '*'      : call('multiply');
  '/'      : call('divide');
  '\'      : call('modulus');
  NEGATE   : call('negate');
  bitNOT   : call('not');
  SHIFTL   : call('shiftl');
  SHIFTR   : call('shiftr');
  END
END;

FUNCTION var_address(var_name : string):LONGINT;
VAR i,result : LONGINT;
    found    : BOOLEAN;
BEGIN
  found := FALSE;
  i := 1;
  REPEAT
    IF variable[i].name = var_name THEN found := TRUE;
    i := i + 1;
  UNTIL found OR (i > max_var);
  IF found THEN result := variable[i-1].vaddr;
  IF NOT found THEN
  BEGIN {see if its a temporary FOR-NEXT variable}
    i := 1;
    REPEAT
      IF for_loop[i].name = var_name THEN found := TRUE;
      i := i + 1
    UNTIL found OR (i > max_for);
    IF found THEN result := for_loop[i-1].addr_low
  END;
  IF NOT found THEN {could be a label or constant}
  BEGIN
    i := 0; {may be a constant}
    REPEAT
      i := i + 1
    UNTIL NOT (var_name[i] IN numbers) OR (i > LENGTH(var_name));
    IF i <= LENGTH(var_name) THEN
    BEGIN
```

```
      IF var_name[length(var_name)] = ':' THEN result := goto_label
      ELSE result := unknown_var
    END
    ELSE result := constant
  END;
  var_address := result
END;

PROCEDURE do_for_loop;
VAR temp : string;
BEGIN {FOR var = expr1 TO expr2 [STEP] expr3}
  for_add := 1;
  WHILE (for_loop[for_add].used = TRUE) AND (for_add <= max_for) DO
    for_add := for_add + 1;
  IF for_add > max_for THEN error(nomorefors);
  WITH for_loop[for_add] DO
  BEGIN
    push_val('',addr_low);
    used := TRUE;
    name := nextsymbol;
    IF name[1] IN numbers THEN error(mistake);
    IF nextsymbol <> '=' THEN error(noequals)
      ELSE expression(pos('to',line));
    call('unstack_to_addr');
    push_val('',addr_high);
    IF nextsymbol <> 'to' THEN error(noto) ELSE
    BEGIN
      IF pos('step',line) > 0 THEN
      BEGIN
        expression(pos('step',line));
        call('unstack_to_addr');
        temp := nextsymbol; {discard STEP,no need to error check}
        push_val('',step);
        expression(length(line));
      END
      ELSE
      BEGIN
        expression(length(line));
        call('unstack_to_addr');
        push_val('',step);
        push_val('',1)
      END
    END;
    call('unstack_to_addr');
    label_op('FORNEXT',for_add);
  END
END;

PROCEDURE do_next;
BEGIN {NEXT}
  IF for_add = 0 THEN error(nofor) ELSE
  BEGIN
    WITH for_loop[for_add] DO
    BEGIN
      push_val('',addr_low);  {loop variable}
      push_cont('',step);     {step contents}
      push_cont('',addr_low); {var  contents}
      call('add');
      call('unstack_to_addr');
      push_cont('',addr_high);
      push_cont('',addr_low);
```

```
      call('lsthan');
      gotoz('FORNEXT',for_add);
      name := '';
      used := FALSE {release FOR-NEXT pair}
    END;
    for_add := for_add - 1
  END
END;

PROCEDURE expect_variable_only(temp : String);
BEGIN
  IF var_address(temp) = constant THEN error(onlyvar)
   ELSE IF var_address(temp) = unknown_var THEN error(nosuchvar)
    ELSE push_val('',var_address(temp));
END;

PROCEDURE expect_variable_or_constant(temp :String);
BEGIN
  IF var_address(temp) = constant THEN
  BEGIN
    IF NOT valid_num(temp) THEN error(mistake) ELSE push_val(temp,0)
  END
  ELSE
  IF var_address(temp) = unknown_var THEN error(nosuchvar)
    ELSE push_cont('',var_address(temp));
END;

{------------------START OF PROCEDURES/FUNCTIONS--------------------}

PROCEDURE do_pause;
BEGIN {PAUSE_MS(expression)}
  IF line[ptr] <> '(' THEN error(nobracket);
  expression(length(line));
  call('pause_ms');
END;

PROCEDURE do_print;
VAR pos : INTEGER;
BEGIN {PRINT expression}
  WHILE (ptr <= length(line)) DO
  BEGIN
    pos := ptr;
    REPEAT
      pos := pos + 1
    UNTIL (line[pos] IN ['=',',']) OR (pos >= length(line));
    expression(pos);
    IF (line[ptr-1] <> '"') AND (line[ptr-2] <> '"') THEN call('print')
  END
END;

PROCEDURE do_sound;
VAR temp,port : string;
BEGIN {SOUND(duration,frequency) on PORTA PIN=RA4}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol;
  expect_variable_or_constant(temp); {look for DURATION}
  IF (ptr = length(line)) OR (line[length(line)] <> ')') THEN error(mispara)
   ELSE
  BEGIN
    operpush('(');
```

```
    expression(length(line));
  END;
  call('sound')
END;

PROCEDURE do_pulsein;
VAR temp,port : string;
BEGIN {PULSEIN(port,pin,duration}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol;
  IF (temp <> 'porta') AND (temp <> 'portb') THEN error(portsyntax)
    ELSE port := temp;
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {look for PIN}
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {look for duration}
  skip_spaces;
  IF line[ptr] <> ')' THEN error(nobracket);
  call('pulsein_'+port);
  call('unstack_to_addr')
END;

PROCEDURE do_pulseout;
VAR temp,port : string;
BEGIN {PULSEOUT(port,pin,duration,variable|constant}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol; {process PORT}
  IF (temp <> 'porta') AND (temp <> 'portb') THEN error(portsyntax)
    ELSE port := temp;
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {process PIN}
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {process DURATION}
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp);
  skip_spaces;
  IF line[ptr] <> ')' THEN error(nobracket);
  call('pulseout_'+port)
END;

PROCEDURE do_rctime;
VAR temp,port : string;
BEGIN {RCTIME(port,pin}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol; {process PORT}
  IF (temp <> 'porta') AND (temp <> 'portb') THEN error(portsyntax)
    ELSE port := temp;
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {process PIN}
  skip_spaces;
  IF line[ptr] <> ')' THEN error(nobracket);
  call('rctime_'+port);
  call('unstack_to_addr')
END;

PROCEDURE do_serinpout;
VAR temp,port : string;
BEGIN {SEROUT(port,rate,pin,variable) SERINP(port,rate,pin)}
```

```
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol;
  IF (temp <> 'porta') AND (temp <> 'portb') THEN error(portsyntax)
    ELSE port := temp;
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  IF valid_baud(temp) THEN push_val(sym_to_baud(temp),0) ELSE error(range);
  IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  expect_variable_or_constant(temp); {process PIN}
  IF (token = 'serout') THEN
   IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
  IF token = 'serout' THEN expect_variable_or_constant(temp) ELSE skip_spaces;
  IF line[ptr] <> ')' THEN error(nobracket);
  IF token = 'serinp' THEN
  BEGIN
    call('serinp_'+port);
    call('unstack_to_addr')
  END ELSE call('serout_'+port)
END;

PROCEDURE do_inoutput;
VAR port : string;
BEGIN {INPUT(port)/OUTPUT(port,expression)}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  port := nextsymbol;
  IF (port <> 'porta') AND (port <> 'portb') THEN error(portsyntax);
  IF token = 'output' THEN
  BEGIN
    operpush('(');
    expression(length(line)-1);
    call(token+'_'+port);
  END
  ELSE
  BEGIN
    call(token+'_'+port);
    call('unstack_to_addr')
  END
END;

PROCEDURE do_random;
BEGIN {RANDOM}
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  IF line[ptr] <> ')' THEN error(nobracket);
  call('random');
  call('unstack_to_addr');
END;

PROCEDURE do_sleepmode;
BEGIN {SLEEPMODE(variable|constant)}
  skip_spaces;
  IF line[ptr] <> '(' THEN error(nobracket);
  IF (ptr = length(line)) OR (line[length(line)] <> ')') THEN error(mispara)
   ELSE expression(length(line));
  call('sleepmode')
END;

PROCEDURE do_nap;
BEGIN {NAP(variable|constant)}
  skip_spaces;
  IF line[ptr] <> '(' THEN error(nobracket);
  IF (ptr = length(line)) OR (line[length(line)] <> ')') THEN error(mispara)
```

```
   ELSE expression(length(line));
  call('nap')
END;

PROCEDURE do_portdir;
BEGIN {PORTDIRA/PORTDIRB(mask)}
  skip_spaces;
  IF line[ptr] <> '(' THEN error(nobracket);
  IF (ptr = length(line)) OR (line[length(line)] <> ')') THEN error(mispara)
   ELSE expression(length(line));
  call(token)
END;

PROCEDURE do_eesqrd;
VAR temp : string;
    num,code  : LONGINT;
BEGIN {EEREAD(address)/EEWRITE(address,variable)}
  skip_spaces;
  IF line[ptr] = '(' THEN ptr := ptr + 1 ELSE error(nobracket);
  temp := nextsymbol;
  expect_variable_or_constant(temp);
  IF token = 'eewrite' THEN
  BEGIN
    IF nextsymbol = ',' THEN temp := nextsymbol ELSE error(mispara);
    expect_variable_or_constant(temp);
  END;
  skip_spaces;
  IF line[ptr] <> ')' THEN error(nobracket);
  IF token = 'eeread' THEN
  BEGIN
    call('eeread');
    call('unstack_to_addr')
  END
  ELSE call('eewrite');
END;

PROCEDURE do_goto;
VAR temp : string;
BEGIN {GOTO label}
  temp := nextsymbol;
  IF temp = '' THEN error(mislabel) ELSE goto_x(temp,label_text);
END;

PROCEDURE do_end;
BEGIN {END of program}
  IF gosubs > max_gosub THEN error(nomoregosubs);
  IF for_add > 0 THEN error(misingnext);
END;

PROCEDURE do_if;
BEGIN {IF conditional-expression THEN statement}
  if_add := if_add + 1;
  expression(POS('then',line));
  token := nextsymbol;
  IF token <> 'then' THEN error(misingthen);
  gotoz('if',if_add);
  token := nextsymbol;
  statement;
  label_op('if',if_add);
END;
```

```
PROCEDURE do_while;
BEGIN {WHILE}
  IF token = 'while' THEN
  BEGIN
    label_op('whileT',while_add);
    while_add := while_add + 1;
    expression(length(line));
    gotoz('whileF',while_add-1);
  END
  ELSE
  BEGIN
    IF while_add <= 0 THEN error(nowhile) ELSE
    BEGIN
      while_add := while_add - 1;
      goto_x('whileT',while_add);
      label_op('whileF',while_add)
    END
  END
END;

PROCEDURE do_repeat;
BEGIN {REPEAT}
  IF token = 'until' THEN
  BEGIN
    IF rep_add <= 0 THEN error(norepeat) ELSE
    BEGIN
      rep_add := rep_add - 1;
      expression(length(line));
      gotoz('repeat',rep_add+1)
    END
  END
  ELSE
  BEGIN
    rep_add := rep_add + 1;
    label_op('repeat',rep_add)
  END
END;

PROCEDURE do_gosub;
VAR temp : string;
BEGIN {GOSUB label}
  temp := nextsymbol;
  IF temp = '' THEN error(misgosub)
  ELSE
  BEGIN
    call(temp);
    gosubs := gosubs + 1
  END
END;

PROCEDURE do_return;
BEGIN {RETURN}
  return;
  IF gosubs > 0 THEN gosubs := gosubs - 1;
END;

{------------------END OF PROCEDURES/FUNCTIONS--------------------}
PROCEDURE var_assignment(var_name : string);
VAR i : LONGINT;
BEGIN
  IF var_name[length(var_name)] = ':' THEN label_op(var_name,label_text)
```

```
  ELSE
  IF var_address(var_name) = unknown_var THEN
  BEGIN
    i := 0;
    REPEAT {find next free location}
      i := i + 1;
    UNTIL (variable[i].used = FALSE) OR (i > max_var);
    IF i <= max_var THEN
    BEGIN
      WITH variable[i] DO
      BEGIN
        used    := TRUE;
        name    := var_name;
        vaddr   := var_add;
        var_add := var_add + 1; {+1 for 8-bit variables, +2 for 16-bit}
      END
    END ELSE error(nomorevars)
  END {otherwise leave it alone}
END;

PROCEDURE expression(pos : LONGINT);
VAR temp : string;
    tch  : CHAR;

 FUNCTION precedence(precd : CHAR):LONGINT;
 VAR r : LONGINT;
 BEGIN
   r := 1;
   CASE precd OF
     bitNOT  : r := 1;
     bitOR,
     bitXOR  : r := 2; {OR,XOR}
     bitAND  : r := 3; {AND}
     '=','>',
     '<',
     LSTHANEQ,
     GRTHANEQ,
     NOTEQU,
     SHIFTL,
     SHIFTR  : r := 4; {=,>,<,<=,>=,<>,®,¯}
     '+','-' : r := 5; {+-}
     '*','/',
     MODULUS : r := 6; {*/\}
     NEGATE,
     '(',')' : r := 7; {(),Unary -+}
   END;
   precedence := r
 END;

 PROCEDURE expect_operand;
 VAR temp1 : string;
     addr,st : LONGINT;
 BEGIN
   temp1 := '';
   skip_spaces;
   IF ptr > length(line) THEN ch := ' ' ELSE ch := line[ptr];
   IF (ch IN unaries) OR found('not') THEN
   BEGIN
     IF ch IN unaries THEN
     BEGIN
       IF ch = '-' THEN operpush(NEGATE);
```

```
    ptr := ptr + 1
  END
  ELSE
  BEGIN
    operpush(bitNOT);
    ptr := ptr + 3
  END;
  expect_operand
END
ELSE
IF ch IN letters THEN
BEGIN
  WHILE (ptr <= length(line)) AND (line[ptr] IN alphanumeric) DO
  BEGIN
    temp1 := temp1 + line[ptr];
    ptr := ptr + 1
  END;
  IF (temp1 = 'false') OR (temp1 = 'true') THEN
  BEGIN
    IF temp1 = 'true' THEN push_val('',255) ELSE push_val('',0)
  END
  ELSE
  BEGIN
    addr := var_address(temp1);
    IF addr = unknown_var THEN error(nosuchvar) ELSE push_cont('',addr);
  END
END
ELSE
IF ch IN numbers THEN
BEGIN
  WHILE (ptr <= length(line)) AND (line[ptr] IN numbers) DO
  BEGIN
    temp1 := temp1 + line[ptr];
    ptr := ptr + 1
  END;
  push_val(temp1,0);
  IF (ptr <= length(line)) AND (line[ptr] IN letters) THEN error(mistake)
END
ELSE
IF ch = '"' THEN
BEGIN
  ptr := ptr + 1; {move past "}
  WHILE (ptr <= length(line)) AND (line[ptr] <> '"') DO
  BEGIN
    temp1 := temp1 + line[ptr];
    ptr := ptr + 1
  END;
  ptr := ptr + 1; {move past last "}
  push_str(temp1);
  call('print');
END
ELSE
IF ch = '%' THEN
BEGIN {%binary notation expected}
  ptr := ptr + 1; {move past %}
  addr := 0;
  st := ptr;
  WHILE (ptr <= length(line)) AND (line[ptr] IN Binaries) DO
  BEGIN
    addr := addr * 2 + ORD(line[ptr]) - 48;
    ptr := ptr + 1
```

```
    END;
    IF (ptr - st) = 8 THEN push_val('',addr) ELSE error(binary)
  END
  ELSE IF ch = leftcomment THEN comment
END;

PROCEDURE check_for_logicals;
VAR temp2 : string;
BEGIN
  IF (line[ptr] IN ['x','a','o']) AND (ptr < (length(line)-2)) THEN
  BEGIN
    IF line[ptr] = 'o' THEN temp2 := line[ptr] + line[ptr+1]
    ELSE temp2 := line[ptr] + line[ptr+1] + line[ptr+2];
    IF (temp2 = 'xor') OR (temp2 = 'and') OR (temp2 = 'or') THEN
    BEGIN
      ptr := ptr + 3;
      IF temp2 = 'or' THEN
      BEGIN
        ptr := ptr - 1;
        ch := bitOR
      END
      ELSE IF temp2 = 'and' THEN ch := bitAND
      ELSE IF temp2 = 'xor' THEN ch := bitXOR
    END
  END
END;

PROCEDURE expect_operator;
VAR tch : CHAR;
    new_pred,
    old_pred : LONGINT;
BEGIN
  skip_spaces;
  IF ptr > length(line) THEN ch := ' ' ELSE ch := line[ptr];
  check_for_logicals;
  ptr := ptr + 1;
  IF ch IN operators THEN
  BEGIN
    new_pred := precedence(ch);
    IF operstack <> NIL THEN
    BEGIN
      tch := operpull; operpush(tch); {duplicate it}
      IF (ch IN [bitAND,bitOR,bitXOR]) AND (tch = ')')
        THEN error(nobracket);
      old_pred := precedence(tch);
      IF ch = '(' THEN operpush(ch)
      ELSE
      BEGIN
        IF ch <> ')' THEN
        BEGIN
          IF old_pred >= new_pred THEN
          BEGIN
            WHILE ((operstack <> NIL) AND (old_pred >= new_pred)) DO
            BEGIN
              tch := operpull;
              old_pred := precedence(tch);
              IF old_pred >= new_pred THEN sym_to_oper(tch)
              ELSE operpush(tch);
            END;
            operpush(ch)
          END ELSE operpush(ch)
```

```
        END
      END;
      IF ch = ')' THEN
      BEGIN
        REPEAT
          tch := operpull;
          IF tch <> '(' THEN sym_to_oper(tch);
        UNTIL ((operstack = NIL) OR (tch = '('));
      END
    END
    ELSE operpush(ch);
    IF ch = ')' THEN expect_operator
  END
  ELSE
  IF ch = leftcomment THEN comment
  ELSE
  BEGIN
    if ch IN ['=','>','<'] THEN
    BEGIN
      IF (ch='>') AND (line[ptr] = '=') THEN
      BEGIN
        operpush(GRTHANEQ); {>=}
        ptr := ptr + 1
      END
      ELSE
      IF (ch='<') AND (line[ptr] = '=') THEN
      BEGIN
        operpush(LSTHANEQ); {<=}
        ptr := ptr + 1
      END
      ELSE
      IF (ch='<') AND (line[ptr] = '>') THEN
      BEGIN
        operpush(NOTEQU); {<>}
        ptr := ptr + 1
      END
      ELSE
      IF (ch='>') AND (line[ptr] = '>') THEN
      BEGIN
        operpush(SHIFTR); {¯}
        ptr := ptr + 1
      END
      ELSE
      IF (ch='<') AND (line[ptr] = '<') THEN
      BEGIN
        operpush(SHIFTL); {®}
        ptr := ptr + 1
      END
      ELSE
      operpush(ch)
    END
  END
 END;

BEGIN {expression}
  WHILE (ptr <= length(line)) AND (ptr <= pos) DO
  BEGIN
    expect_operand;
    expect_operator;
  END;
  ptr := ptr - 1;
```

```
  WHILE operstack <> NIL DO
  BEGIN
    tch := operpull;
    IF (operstack = NIL) AND (tch IN ['(',')']) THEN error(nobracket)
    ELSE sym_to_oper(tch);
  END;
END;

FUNCTION func_follows:BOOLEAN;
VAR temp : string;
    i    : LONGINT;
BEGIN
  temp := '';
  skip_spaces;
  i := ptr;
  func_follows := FALSE;
  WHILE line[i] IN letters DO
  BEGIN
    temp := temp + line[i];
    i := i + 1
  END;
  IF temp = 'serinp'  THEN func_follows := TRUE;
  IF temp = 'input'   THEN func_follows := TRUE;
  IF temp = 'rctime'  THEN func_follows := TRUE;
  IF temp = 'pulsein' THEN func_follows := TRUE;
  IF temp = 'random'  THEN func_follows := TRUE;
  IF temp = 'eeread'  THEN func_follows := TRUE;
END;

PROCEDURE statement;
VAR addr : LONGINT;
BEGIN
       IF  token = 'for'        THEN do_for_loop
  ELSE IF  token = 'next'       THEN do_next
  ELSE IF  token = 'goto'       THEN do_goto
  ELSE IF  token = 'nap'        THEN do_nap
  ELSE IF  token = 'sleepmode'  THEN do_sleepmode
  ELSE IF  token = 'print'      THEN do_print
  ELSE IF  token = 'pause_ms'   THEN do_pause
  ELSE IF  token = 'gosub'      THEN do_gosub
  ELSE IF  token = 'return'     THEN do_return
  ELSE IF  token = 'end'        THEN do_end
  ELSE IF  token = 'if'         THEN do_if
  ELSE IF  token = 'sound'      THEN do_sound
  ELSE IF  token = 'pulsein'    THEN do_pulsein
  ELSE IF  token = 'pulseout'   THEN do_pulseout
  ELSE IF  token = 'random'     THEN do_random
  ELSE IF  token = 'rctime'     THEN do_rctime
  ELSE IF (token = 'serout')   OR (token = 'serinp')   THEN do_serinpout
  ELSE IF (token = 'input')    OR (token = 'output')   THEN do_inoutput
  ELSE IF (token = 'while')    OR (token = 'wend')     THEN do_while
  ELSE IF (token = 'repeat')   OR (token = 'until')    THEN do_repeat
  ELSE IF (token = 'eeread')   OR (token = 'eewrite')  THEN do_eesqrd
  ELSE IF (token = 'portdira') OR (token = 'portdirb') THEN do_portdir
  ELSE
  BEGIN
    addr := var_address(token);
    IF addr = unknown_var THEN
    BEGIN
      var_assignment(token);
      addr := var_address(token);
```

```
      push_val('',addr);
      token := nextsymbol;
      IF token <> '=' THEN error(noequals)
      ELSE
      BEGIN
        IF func_follows THEN
        BEGIN
          token := nextsymbol;
          statement
        END
        ELSE
        BEGIN
          expression(length(line));
          call('unstack_to_addr')
        END
      END
    END
    ELSE
    BEGIN
      IF addr = goto_label THEN label_op(token,label_text)
      ELSE
      IF addr = constant THEN error(mistake)
      ELSE
      BEGIN
        token := nextsymbol;
        IF token <> '=' THEN error(noequals)
        ELSE
        BEGIN
          push_val('',addr);
          IF func_follows then
          BEGIN
            token := nextsymbol;
            statement
          END
          ELSE
          BEGIN
            expression(length(line));
            call('unstack_to_addr');
          END
        END
      END
    END
  END
END;

BEGIN
  initialise;
  ASSIGN(output,'');
  REWRITE(output);
  ASSIGN(source,'clock.bas');
  RESET(source);
  writeln(':20,'               processor 16c84');
  writeln(':20,'               radix dec');
  writeln(':20,'               include p16cxx.inc');
  WHILE NOT EOF(source) DO
  BEGIN
    read_line;
    token := nextsymbol;
    IF (ch <> etx) AND (token<>'') THEN statement
  END;
  writeln('pull                  return');  writeln('push                  return');
```

```
  writeln('unstack_to_addr        return');  writeln('add                   return');
  writeln('sub                    return');  writeln('multiply              return');
  writeln('divide                 return');  writeln('lsthan                return');
  writeln('grthan                 return');  writeln('equal                 return');
  writeln('notequal               return');  writeln('lsthaneq              return');
  writeln('grthaneq               return');  writeln('sound                 return');
  writeln('rctime_porta           return');  writeln('rctime_portb          return');
  writeln('random                 return');  writeln('pulseout_porta        return');
  writeln('pulseout_portb         return');  writeln('serinp_porta          return');
  writeln('serinp_portb           return');  writeln('serout_porta          return');
  writeln('serout_portb           return');  writeln('or                    return');
  writeln('xor                    return');  writeln('and                   return');
  writeln('modulus                return');  writeln('negate                return');
  writeln('not                    return');  writeln('shiftl                return');
  writeln('shiftr                 return');  writeln('nap                   return');
  writeln('sleepmode              return');  writeln('pulsein_porta         return');
  writeln('pulsein_portb          return');  writeln('eeread                return');
  writeln('eewrite                return');  writeln('print                 return');
  writeln('portdira               return');  writeln('portdirb              return');
  writeln('input_porta            return');  writeln('input_portb           return');
  writeln('output_porta           return');  writeln('output_portb          return');
  writeln('pause_ms               return');  writeln('                      END');
  WRITELN(';  *** Program size = ',pc+45,' Bytes');
  CLOSE(output);
  READLN
END.
{1243 Lines}
```