```
{{
Modified Tiny Basic for use with Propeller Demo Board and Hydra.
Original Tiny Basic written by Tomas Rokicki & Radical Eye Software.

Copyright (c) 2008 Michael Green.  See end of file for terms of use.
}}

'' 2009-07-09 - Corrected WRITE and PRINT for case of -2,147,483,648
'' 2010-06-04 - Changed fsrw to Rokicki's version 2.6 for FAT16/FAT32
'' 2010-06-05 - Strip out all references to SD card I/O
''              Change all console I/O to use FullDuplexSerialMG
'' 2010-06-05 - Added SERVO statement to control servos
''              Changed getline to handle possible half duplex input
'' 2010-06-18 - Added REBOOT statement
'' 2010-06-23 - Added comments.  Turn off servo when program stops.
''              Other miscellaneous small fixes

obj
   ser : "FullDuplexSerialMG"          ' Modified FullDuplexSerial
   i2c : "Basic_I2C_Driver"            ' From Object Exchange
   srv : "Servo32v7"                   ' From Propeller Servo Controller

con
   version   = 1                       ' Major version
   release   = 001                     ' Minor release
   testLevel = 0                       ' Test change level

   USB_Rx    = 31                      ' USB console receive pin
   USB_Tx    = 30                      ' USB console transmit pin
   USB_Baud  = 2400                    ' USB console Baud

   Aux_RxTx  = 27                      ' Auxiliary serial I/O pin
   Activity  = 26                      ' Activity LED

   progsize  = 16384                   ' Space reserved for program
   _clkmode  = xtal1 + pll16x
   _xinfreq  =  5_000_000
   _stack    = 100                     ' Roughly 4 nested expressions
   _free     = (progsize + 12) / 4

   maxstack  = 20                      ' Maximum stack depth
   linelen   = 256                     ' Maximum input line length
   quote     = 34                      ' Double quote
   caseBit   = !32                     ' Uppercase/Lowercase bit
   userPtr   = $7FEC                   ' Pointer to program memory
   ranSeed   = $12345678               ' Initial pseudo-random seed

'' Internal SPIN information at fixed locations in RAM/EEPROM image

   spinPbase = $0006                   ' must be $0010 (program base addr)
   spinVbase = $0008                   ' number of longs loaded times 4
   spinDbase = $000A                   ' above where $FFF9FFFF's get placed
   spinPcurr = $000C                   ' points to SPIN code
   spinDcurr = $000E                   ' points to local stack

var
   long sp, tp, eop, rv, nextlineloc, curlineno, pauseTime
   long vars[26], stack[maxstack], serialIsUSB
   long forStep[26], forLimit[26], forLoop[26]
   word outputs, servosOn
   byte tline[linelen], tailLine[linelen], inVars[26], fileOpened

'' Table of tokens (zero-terminated strings).  The program is scanned for these
'' and the original text is replaced by the number of the token + 128.  The use
'' of tokens simplifies the parsing of the program.  Note that a token should
'' not be a prefix of another token (RUN and RUNIT for example).  If you need
'' something like this, the longer token should come first.

dat
```

```
    tok0  byte "IF", 0
    tok1  byte "THEN", 0
    tok2  byte "INPUT", 0    ' INPUT {"<prompt>";} <var> {,<var>}
    tok3  byte "PRINT", 0    ' PRINT {USING "<format>";} ...
    tok4  byte "GOTO", 0
    tok5  byte "GOSUB", 0
    tok6  byte "RETURN", 0
    tok7  byte "REM", 0
    tok8  byte "NEW", 0
    tok9  byte "LIST", 0
    tok10 byte "RUN", 0
    tok11 byte "RND", 0
    tok12 byte "OPEN", 0     ' OPEN " <file> ",<mode>
    tok13 byte "READ", 0     ' READ <var> {,<var>}
    tok14 byte "WRITE", 0    ' WRITE {USING "<format>";} ...
    tok15 byte "CLOSE", 0    ' CLOSE
    tok16 byte "DELETE", 0   ' DELETE " <file> "
    tok17 byte "RENAME", 0   ' RENAME " <file> "," <file> "
    tok18 byte "FILES", 0    ' FILES
    tok19 byte "SAVE", 0     ' SAVE or SAVE [<expr>] or SAVE "<file>"
    tok20 byte "LOAD", 0     ' LOAD or LOAD [<expr>] or LOAD "<file>"
    tok21 byte "NOT" ,0      ' NOT <logical>
    tok22 byte "AND" ,0      ' <logical> AND <logical>
    tok23 byte "OR", 0       ' <logical> OR <logical>
    tok24 byte "SHL", 0      ' <expr> SHL <expr>
    tok25 byte "SHR", 0      ' <expr> SHR <expr>
    tok26 byte "FOR", 0      ' FOR <var> = <expr> TO <expr>
    tok27 byte "TO", 0
    tok28 byte "STEP", 0     '  optional STEP <expr>
    tok29 byte "NEXT", 0     ' NEXT <var>
    tok30 byte "INA", 0      ' INA [ <expr> ]
    tok31 byte "OUTA", 0     ' OUTA [ <expr> ] = <expr>
    tok32 byte "PAUSE", 0    ' PAUSE <time ms> {,<time us>}
    tok33 byte "USING", 0    ' PRINT USING "<format>"; ...
    tok34 byte "ROL", 0      ' <expr> ROL <expr>
    tok35 byte "ROR", 0      ' <expr> ROR <expr>
    tok36 byte "SAR", 0      ' <expr> SAR <expr>
    tok37 byte "REV", 0      ' <expr> REV <expr>
    tok38 byte "BYTE", 0     ' BYTE [ <expr> ]
    tok39 byte "WORD", 0     ' WORD [ <expr> ]
    tok40 byte "LONG", 0     ' LONG [ <expr> ]
    tok41 byte "CNT", 0
    tok42 byte "PHSA", 0
    tok43 byte "PHSB", 0
    tok44 byte "FRQA", 0
    tok45 byte "FRQB", 0
    tok46 byte "CTRA", 0
    tok47 byte "CTRB", 0
    tok48 byte "DISPLAY", 0  ' DISPLAY <expr> {,<expr>}
    tok49 byte "KEYCODE", 0  ' KEYCODE
    tok50 byte "LET", 0
    tok51 byte "STOP", 0
    tok52 byte "END", 0
    tok53 byte "EEPROM", 0   ' EEPROM[ <expr> ]
    tok54 byte "FILE", 0     ' FILE
    tok55 byte "MEM", 0      ' MEM
    tok56 byte "SPIN", 0     ' SPIN [<expr>] or SPIN "<file>"
    tok57 byte "COPY", 0     ' COPY [<expr>],"<file>" or COPY "<file>",[<expr>] or
                             ' COPY [<expr>],<expr> where <expr> are different
    tok58 byte "DUMP", 0     ' DUMP <expr>,<expr> or DUMP [<expr>],<expr>
    tok59 byte "SERVO", 0    ' SERVO <expr>,<expr>
                             ' SERVO <expr>,<expr>,<expr>
    tok60 byte "BAUD", 0     ' BAUD = <expr>
    tok61 byte "REBOOT", 0   ' REBOOT

'' This is a table of offsets of the token text strings.  The token scanner
'' starts at "toks" and stops at "tokx" (or when a match occurs).  You can
'' add to the end of the table.  The actual token stored is +128.

```

```
   toks  word @tok0, @tok1, @tok2, @tok3, @tok4, @tok5, @tok6, @tok7
         word @tok8, @tok9, @tok10, @tok11, @tok12, @tok13, @tok14, @tok15
         word @tok16, @tok17, @tok18, @tok19, @tok20, @tok21, @tok22, @tok23
         word @tok24, @tok25, @tok26, @tok27, @tok28, @tok29, @tok30, @tok31
         word @tok32, @tok33, @tok34, @tok35, @tok36, @tok37, @tok38, @tok39
         word @tok40, @tok41, @tok42, @tok43, @tok44, @tok45, @tok46, @tok47
         word @tok48, @tok49, @tok50, @tok51, @tok52, @tok53, @tok54, @tok55
         word @tok56, @tok57, @tok58, @tok59, @tok60, @tok61
   tokx  word

   syn   byte "Syntax Error", 0
   ln    byte "Invalid Line Number", 0

   crlf  byte ser#Cr, ser#Lf, 0 ' New line

'' Main method.  Initializes program, displays banner, then does main loop
'' which reads an input line, processes it, and displays any error messages.

PUB main | err, s
   initSerial(USB_Baud)          ' Initialize serial port (to USB_Baud)
   outa[Activity] := 0           ' Activity LED off initially
   dira[Activity] := 1
   rv := ranSeed                 ' Set initial random seed
   pauseTime := 0
   outputs := 0
   long[userPtr] := userPtr - progsize ' Allocate memory
   ser.str(string("PSC Basic"))
   if version > 0 or release > 0 or testLevel > 0
     ser.str(string(" Version "))
     ser.dec(version)
     ser.tx(".")
     if release < 100
       ser.tx("0")
     if release < 10
       ser.tx("0")
     ser.dec(release)
     if testLevel > 0
       ser.tx("a"+testLevel-1)
   ser.str(@crlf)
   servosOn := 0                 ' No servos in use
   srv.start                     ' Start servo controller
   srv.ramp                      ' Start servo ramp handler
   waitcnt(clkfreq + cnt)
   clearall                      ' Clear program space and variables
   s := 0
   curlineno := -1
   repeat
      err := \doline(s)          ' Read an input line and interpret it
      s := 0
      outa[Activity] := 0
      if err                     ' Show any error messages
         showError(err)

'' Displays error message (address of string in "err") including line number
'' if program is executing.  Sets next line address to end of program to stop
'' execution.

PRI showError(err)
   if curlineno => 0
      ser.str(string("IN LINE "))
      ser.dec(curlineno)
      ser.tx(" ")
   putlinet(err)
   nextlineloc := eop - 2

'' Get a line of text from the input device.  Handles backspace and return.
'' If input is programming port, echo it and handle backspace.  If input is
'' auxiliary serial port, flush transmitter and receiver initially and don't
'' echo.  Set "tp" to start of text string.
```

```

PRI getline | i, c
   ifnot serialIsUSB
      ser.txflush                ' Wait for transmit to finish
      ser.rxflush                ' Remove any echoed characters
   i := 0
   repeat
      c := ser.rx
      if c == ser#Bsp
         if i > 0
            if serialIsUSB       ' Echo input if USB console
               ser.str(string(ser#Bsp," ",ser#Bsp))
            i--
      elseif c == ser#Cr
         if serialIsUSB          ' Echo input if USB console
            ser.tx(c)
            ser.tx(ser#Lf)
         tline[i] := 0
         tp := @tline
         return
      elseif i < linelen-1
         if serialIsUSB          ' Echo input if USB console
            ser.tx(c)
         tline[i++] := c

'' Display a tokenized text line.  Replace any tokens (byte >= 128) with the
'' text from the token table.  If an unknown token is found, display its value
'' in braces {}.  If token is not REM, put a space after it.

pri putlinet(s) | c, ntoks
   ntoks := (@tokx - @toks) / 2
   repeat while c := byte[s++]
      if c => 128
         if (c -= 128) < ntoks
            ser.str(@@toks[c])
            if c <> 7    ' REM
               ser.tx(" ")
         else
            ser.tx("{")
            ser.dec(c)
            ser.tx("}")
      else
         ser.tx(c)
   ser.str(@crlf)

'' Skip spaces in the input line ("tp" is the pointer) and return the first
'' non-blank character found.  A zero byte also terminates the operation.

pri spaces | c
   repeat
      c := byte[tp]
      if c == 0 or c > " "
         return c
      tp++

'' Skip a single character in the input line (unless the zero byte terminator)
'' and call "spaces" to return the next non-blank.

pri skipspaces
   if byte[tp]
      tp++
   return spaces

'' The next input item is assumed to be a decimal number.  Return the value of
'' that number and leave "tp" pointing to the non-digit following the number.
'' If the next input character is not a digit, return zero.

pri parseliteral | r, c
   r := 0
   repeat
      c := byte[tp]
      if c < "0" or c > "9"
         return r
      r := r * 10 + c - "0"
      tp++

'' Move the remainder of the program beginning at "at" upwards in memory a
'' distance "delta" to make room for a new line.

pri movprog(at, delta)
   if eop + delta + 2 - long[userPtr] > progsize
      abort string("NO MEMORY")
   bytemove(at+delta, at, eop-at)
   eop += delta

'' Convert upper case letters to lower case and return the index of the letter

pri fixvar(c)
   if c => "a"
      c -= 32
   return c - "A"

'' Return true if the supplied character is a letter (a variable name)

pri isvar(c)
   c := fixvar(c)
   return c => 0 and c < 26

'' Scan the input line.  Replace any tokens with their 128+ code.  Any
'' characters within double quotes (") are copied literally as are comments
'' beginning with the REM token.

pri tokenize | tok, c, at, put, state, i, j, ntoks
   ntoks := (@tokx - @toks) / 2
   at := tp
   put := tp
   state := 0
   repeat while c := byte[at]
      if c == quote
         if state == "Q"
            state := 0
         elseif state == 0
            state := "Q"
      if state == 0
         repeat i from 0 to ntoks-1
            tok := @@toks[i]
            j := 0
            repeat while byte[tok] and ((byte[tok] ^ byte[j+at]) & caseBit) == 0
               j++
               tok++
            if byte[tok] == 0 and not isvar(byte[j+at])
               byte[put++] := 128 + i
               at += j
               if i == 7
                  state := "R"
               else
                  repeat while byte[at] == " "
                     at++
                  state := "F"
               quit
         if state == "F"
            state := 0
         else
            byte[put++] := byte[at++]
      else
         byte[put++] := byte[at++]
   byte[put] := 0

```

```
'' Return the byte-aligned word value at location "loc"

pri wordat(loc)
   return (byte[loc]<<8)+byte[loc+1]

'' Return the address of the line in the program that has a line number greater
'' or equal to the supplied line number.

pri findline(lineno) | at
   at := long[userPtr]
   repeat while wordat(at) < lineno
      at += 3 + strsize(at+2)
   return at

'' Scan the line number, tokenize the rest of the line, and insert it
'' in the appropriate position in the stored program.  Delete any existing
'' line with the same line number.

pri insertline | lineno, fc, loc, locat, newlen, oldlen
   lineno := parseliteral
   if lineno < 0 or lineno => 65535
      abort @ln
   tokenize
   fc := spaces
   loc := findline(lineno)
   locat := wordat(loc)
   newlen := 3 + strsize(tp)
   if locat == lineno
      oldlen := 3 + strsize(loc+2)
      if fc == 0
         movprog(loc+oldlen, -oldlen)
      else
         movprog(loc+oldlen, newlen-oldlen)
   elseif fc
      movprog(loc, newlen)
   if fc
      byte[loc] := lineno >> 8
      byte[loc+1] := lineno
      bytemove(loc+2, tp, newlen-2)

'' Clear all variables to zero, clear the data stack, initialize the PAUSE
'' time, and set the program execution point to the beginning of the program

pri clearvars
   longfill(@vars, 0, 26)
   pauseTime := 0
   nextlineloc := long[userPtr]
   sp := 0

'' Clear the current program and the data stack

pri newprog
   byte[long[userPtr]][0] := 255
   byte[long[userPtr]][1] := 255
   byte[long[userPtr]][2] := 0
   eop := long[userPtr] + 2
   nextlineloc := eop - 2
   sp := 0

'' Clear the program space, variables, and data stack

pri clearall
   newprog
   clearvars

'' Check for a possible data stack overflow and push the current program
'' pointer into the stack (for a GOSUB)

pri pushstack
```

```
   if sp => constant(maxstack-1)
      abort string("RECURSION ERROR")
   stack[sp++] := nextlineloc

'' Process a possible EEPROM address, I/O pin number, or other [] bracketed
'' value depending on the "delim" value.  If ".", then a simple I/O pin
'' number is expected and checked for range 0-31.  A pin range like "1..5"
'' is allowed.  If present, the first value is returned in the high word of
'' the result and the second value is returned in the low word of the result.
'' If ",", then an EEPROM address is expected with the I/O pin number as the
'' first value, then a comma followed by the EEPROM address.  The pin number
'' in this case must be even and the EEPROM address in the range 0..$7FFFF.
'' The pin number is shifted into position for the sdspiFemto low level I2C
'' routines to use directly.  Any other "delim" value allows any single
'' expression value to be in the brackets and this is returned unchanged.

pri getAddress(delim) | t
   if spaces <> "["
      abort @syn
   skipspaces
   result := expr
   if delim == "." and (result < 0 or result > 31)
      abort string("Invalid pin number")
   if delim == "." or delim == ","
      if spaces == delim
         if delim == "."                 ' Handle the form <expr>..<expr>
            if byte[++tp] <> "."
               abort @syn
            result <<= 8
            skipspaces
            t := expr
            if t < 0 or t > 31
               abort string("Invalid pin number")
            result |= t | $10000
         else                            ' Handle the form <expr>,<expr>
            if result & 1 or result < 0 or result > 31
               abort string("Invalid pin number")
            skipspaces
            result := (result << 18) | (expr & $7FFFF)
      elseif delim == ","
         result &= $7FFFF
   if spaces <> "]"
      abort @syn
   tp++

'' This is the "lowest level" routine in the recursive descent expression
'' parser.

pri factor | tok, t, i
   tok := spaces
   tp++
   case tok
      "(":                               ' Recurse to handle ( <expr> )
         t := expr
         if spaces <> ")"
            abort @syn
         tp++
         return t
      "a".."z","A".."Z":                 ' Scalar variable, return value
         return vars[fixvar(tok)]
      158: ' INA [ <expr>{..<expr>} ] ' Input register (INA) bit/bits
         t := getAddress(".")
         if t > $FFFF
           tok := t & $FF
           t := (t >> 8) & $FF
           repeat i from t to tok
              outputs &= ! |< i
           dira[t..tok]~
           return ina[t..tok]
```

```
          else
            outputs &= ! |< t
            dira[t]~
            return ina[t]
       166: ' BYTE [ <expr> ]          ' Hub memory byte value given address
          return byte[getAddress(" ")]
       167: ' WORD [ <expr> ]          ' Hub memory word value given address
          return word[getAddress(" ")]
       168: ' LONG [ <expr> ]          ' Hub memory long value given address
          return long[getAddress(" ")]
       181: ' EEPROM [ <expr> ]        ' EEPROM byte value given pin# / address
          t := getAddress(",")
          t := i2c.readByte(i2c#bootPin,i2c#EEPROM,t)
          if t < 0
             abort string("EEPROM read")
          return t
       182: ' FILE                     ' Byte from currently open file
          abort @syn
       183: ' MEM                      ' Return available program memory size
          return progsize - (eop - long[userPtr] )
       169: ' CNT                      ' Return system clock register (CNT)
          return CNT
       170: ' PHSA                     ' Return counter A phase value
          return PHSA
       171: ' PHSB                     ' Return counter B phase value
          return PHSB
       172: ' FRQA                     ' Return counter A frequency value
          return FRQA
       173: ' FRQB                     ' Return counter B frequency value
          return FRQB
       174: ' CTRA                     ' Return counter A control value
          return CTRA
       175: ' CTRB                     ' Return counter B control value
          return CTRB
       177: ' KEYCODE                  ' Return current input character (or -1)
          return ser.rx
       139: ' RND <factor>             ' Return pseudo-random value and
          return rv? // factor         '  limit range to supplied value
       "-":                            ' Unary negate
          return - factor
       "!":                            ' Unary bitwise not
          return ! factor
       "$", "%", quote, "0".."9":      ' Hex, binary, character, decimal constant
          --tp
          return getAnyNumber
       other:
          abort @syn

pri shifts | tok, t                    ' Shift operations
   t := factor
   tok := spaces
   if tok == 152                       ' SHL - Left logical shift
      tp++
      return t << factor
   elseif tok == 153                   ' SHR - Right logical shift
      tp++
      return t >> factor
   elseif tok == 162                   ' ROL - Left rotate
      tp++
      return t <- factor
   elseif tok == 163                   ' ROR - Right rotate
      tp++
      return t -> factor
   elseif tok == 164                   ' SAR - Right arithmetic shift
      tp++
      return t ~> factor
   elseif tok == 165                   ' REV - Reverse specified # of bits
      tp++
      return t >< factor
   else
      return t

pri bitFactor | tok, t
   t := shifts
   repeat
      tok := spaces
      if tok == "&"                        ' Bitwise and operation
         tp++
         t &= shifts
      else
         return t

pri bitTerm | tok, t
   t := bitFactor
   repeat
      tok := spaces
      if tok == "|"                        ' Bitwise or operation
         tp++
         t |= bitFactor
      elseif tok == "^"                    ' Bitwise exclusive or operation
         tp++
         t ^= bitFactor
      else
         return t

pri term | tok, t
   t := bitTerm
   repeat
      tok := spaces
     if tok == "*"                         ' Multiplication
        tp++
        t *= bitTerm
     elseif tok == "/"                     ' Division or modulus
        if byte[++tp] == "/"
           tp++
           t //= bitTerm
        else
           t / =bitTerm
     else
        return t

pri arithExpr | tok, t
   t := term
   repeat
      tok := spaces
      if tok == "+"                        ' Addition
         tp++
         t += term
      elseif tok == "-"                    ' Subtraction
         tp++
         t -= term
      else
         return t

pri compare | op, a, b, c                  ' Comparison operations
   a := arithExpr
   op := 0
   spaces
   repeat
      c := byte[tp]                        ' Allow combinations of characters
      case c                               '  to occur in any order
         "<": op |= 1
              tp++
         ">": op |= 2
              tp++
         "=": op |= 4
              tp++
         other: quit
```

```
  case op
     0: return a
     1: return a < arithExpr
     2: return a > arithExpr
     3: return a <> arithExpr
     4: return a == arithExpr
     5: return a =< arithExpr
     6: return a => arithExpr
     7: abort string("Invalid comparison")

pri logicNot | tok
  tok := spaces
  if tok == 149                        ' NOT - Logical not
     tp++
     return not compare
  return compare

pri logicAnd | t, tok
  t := logicNot
  repeat
     tok := spaces
     if tok == 150                     ' AND - Logical and
        tp++
        t := t and logicNot
     else
        return t

pri expr | tok, t
  t := logicAnd
  repeat
     tok := spaces
     if tok == 151                     ' OR - Logical or
        tp++
        t := t or logicAnd
     else
        return t

'' Handle "pseudo-assignment" where some special token comes first, then an
'' assignment (equal sign) and an arbitrary expression.  Return the value of
'' the expression.

pri specialExpr
  if spaces <> "="
     abort @syn
  skipspaces
  return expr

'' This routine handles all statements and sequences of statements separated
'' by ":".  It handles the interrupted multiple interpretation of PAUSEs as
'' well as the FOR / NEXT looping.

pri texec | ht, nt, restart, thisLine, uS, a,b,c,d,w, f0,f1,f2,f3,f4,f5,f6,f7
  uS := clkfreq / 1_000_000
  thisLine := tp - 2
  restart := 1
  repeat while restart
     restart := 0
     ht := spaces
     if ht == 0                        ' Quit at the end of the line
        return
     nt := skipspaces
     if isvar(ht) and nt == "="        ' Variable assignment
        tp++
        vars[fixvar(ht)] := expr
     elseif ht => 128
        case ht
           128: ' IF <expr> THEN       ' If <expr> succeeds, continue
              a := expr                '  interpreting rest of the line
              if spaces <> 129
```

```
                  abort string("MISSING THEN")
               skipspaces
               if not a
                  return
               restart := 1
            130: ' INPUT {"<prompt>";} <var> {, <var>}
               if nt == quote          ' Display prompt if present
                  c := byte[++tp]
                  repeat while c <> quote and c
                     ser.tx(c)
                     c := byte[++tp]
                  if c <> quote
                     abort @syn
                  if skipspaces <> ";"
                     abort @syn
                  nt := skipspaces
               if not isvar(nt)        ' Must be at least one variable
                  abort @syn
               b := 0
               inVars[b++] := fixvar(nt)
               repeat while skipspaces == ","
                  nt := skipspaces     ' Parse the sequence of variables
                  if not isvar(nt) or b == 26
                     abort @syn
                  inVars[b++] := fixvar(nt)
               getline                 ' Get a line of input text & tokenize
               tokenize
               repeat a from 1 to b    ' Store any values provided in variables
                  vars[inVars[a-1]] := expr
                  if a < b
                     if spaces == ","
                        skipspaces
            131: ' PRINT
               a := 0
               repeat
                  nt := spaces
                  if nt == 0 or nt == ":"
                     quit
                  if nt == quote       ' Display string constant
                     tp++
                     repeat
                        c := byte[tp++]
                        if c == 0 or c == quote
                           quit
                        ser.tx(c)
                        a++
                  else                 ' Display <expr> value in decimal
                     d := (b := expr) == negx
                     if b < 0
                        b := ||(b+d)
                        ser.tx("-")
                        a++
                     c := 1_000_000_000
                     d~
                     repeat 10
                        if b => c
                           ser.tx(b / c + "0")
                           a++
                           b //= c
                           d~~
                        elseif d or c == 1
                           ser.tx("0")
                           a++
                        c /= 10
                  nt := spaces
                  if nt == ";"
                     tp++
                  elseif nt == ","     ' Comma as separator, use display columns
                     ser.tx(" ")
                     a++
                     repeat while a & 7
                        ser.tx(" ")
                        a++
                     tp++
                  elseif nt == 0 or nt == ":"
                     ser.str(@crlf)   ' Terminate display line unless ":"
                     quit
                  else
                     abort @syn
            132, 133: ' GOTO, GOSUB  ' GOTO or GOSUB label
               a := expr
               if a < 0 or a => 65535
                  abort @ln
               b := findline(a)       ' Use nearest line greater than requested
               if wordat(b) <> a
                  abort @ln
               if ht == 133           ' If GOSUB, save pointer to next line
                  pushstack
               nextlineloc := b
            134: ' RETURN             ' Return to saved pointer to next line
               if sp == 0             '  unless there's a stack underflow
                  abort string("INVALID RETURN")
               nextlineloc := stack[--sp]
            135: ' REM                ' Ignore rest of remark line
               repeat while skipspaces
            136: ' NEW                ' Clear variables and erase program
               clearall
            137: ' LIST {<expr> {,<expr>}}
               b := 0                 ' Default line range
               c := 65535
               if spaces <> 0         ' At least one parameter
                  b := c := expr
                  if spaces == ","    ' List requested line or range of lines
                     skipspaces
                     c := expr
               a := long[userPtr]
               repeat while a+2 < eop
                  d := wordat(a)
                  if d => b and d =< c
                     ser.dec(d)
                     ser.tx(" ")
                     putlinet(a+2)
                  a += 3 + strsize(a+2)
            138: ' RUN                ' Clear variables and start executing
                  clearvars
            140: ' OPEN " <file> ", R/W/A
               abort @syn             ' Not used in this version
            141: ' READ <var> {, <var> }
               abort @syn             ' Not used in this version
            142: ' WRITE ...
               abort @syn             ' Not used in this version
            143: ' CLOSE
               abort @syn             ' Not used in this version
            144: ' DELETE " <file> "
               abort @syn             ' Not used in this version
            145: ' RENAME " <file> "," <file> "
               abort @syn             ' Not used in this version
            146: ' FILES
               abort @syn             ' Not used in this version
            147: ' SAVE or SAVE [ <expr> ]
               if (nt := spaces) == quote
                  abort @syn
               else                   ' Save program to EEPROM
                  if nt == "["                    ' Align save area for paged writes
                     a := getaddress(",") + 64
                     if (a & 63) == 63
                        a += 64
                     a &= $7FFC0
```

```
                 else
                    a := (userPtr - progsize - 62) & $7FC0
                 nt := spaces
                 if nt <> 0 and nt <> ":"
                    abort @syn                     ' Write program to EEPROM
                 d := eop - long[userPtr] + 1
                 if i2c.writeWord(i2c#bootPin,i2c#EEPROM,a-2,d) ' Program size
                    abort string("Save EEPROM write")
                 w := cnt                          ' prepare to check for a timeout
                 repeat while i2c.WriteWait(i2c#bootPin,i2c#EEPROM,a-2)
                   if cnt - w > clkfreq / 10
                     abort string("Save EEPROM timeout")
                 repeat c from 0 to d step 64    ' Write the program itself
                   if i2c.writePage(i2c#bootPin,i2c#EEPROM,a+c,long[userPtr]+c,d-c<#64)
                     abort string("Save EEPROM write")
                   w := cnt                        ' prepare to check for a timeout
                   repeat while i2c.WriteWait(i2c#bootPin,i2c#EEPROM,a+c)
                     if cnt - w > clkfreq / 10
                       abort string("Save EEPROM timeout")
           148: ' LOAD or LOAD [ <expr> ]
              if (nt := spaces) == quote
                 abort @syn
              else                       ' Load program from EEPROM
                 if nt == "["                      ' Align save area for paged writes
                    a := getaddress(",") + 64
                    if (a & 63) == 63
                       a += 64
                    a &=$7FFC0
                 else
                    a := (userPtr - progsize - 62) & $7FC0
                 nt := spaces
                 if nt <> 0 and nt <> ":"
                    abort @syn                     ' Read program from EEPROM
                 d := i2c.readWord(i2c#bootPin,i2c#EEPROM,a-2)
                 if d < 0
                    abort string("Load EEPROM read")
                 d &= $FFFF
                 if d < 3 or d > progsize          ' Read program size & check
                    abort string("Invalid program size")
                 c := @tailLine                    ' Save statement tail
                 repeat while byte[c++] := byte[tp++]
                 tp := @tailLine                   ' Scan copy after load
                 if i2c.readPage(i2c#bootPin,i2c#EEPROM,a,long[userPtr],d)
                    abort string("Load EEPROM read")
                 eop := long[userPtr] + d - 1
                 nextlineloc := eop - 2            ' Leave it stopped
           154: ' FOR <var> = <expr> TO <expr> {STEP <expr>}
              ht := spaces
              if ht == 0
                 abort @syn
              nt := skipspaces
              if not isvar(ht) or nt <> "="
                 abort @syn
              a := fixvar(ht)            ' Get FOR variable index
              skipspaces
              vars[a] := expr
              if spaces <> 155           ' TO - Save FOR limit
                 abort @syn
              skipspaces
              forLimit[a] := expr
              if spaces == 156           ' STEP - Save step size
                 skipspaces
                 forStep[a] := expr
              else
                 forStep[a] := 1         ' Default step is 1
              if spaces
                 abort @syn
              forLoop[a] := nextlineloc            ' Save address of line
              if forStep[a] < 0                    '  following the FOR
```

```
                  b := vars[a] => forLimit[a]
               else                              ' Initially past the limit?
                  b := vars[a] =< forLimit[a]
               if not b                          ' Search for matching NEXT
                  repeat while nextlineloc < eop-2
                       curlineno := wordat(nextlineloc)
                     tp := nextlineloc + 2
                     nextlineloc := tp + strsize(tp) + 1
                     if spaces == 157            ' NEXT <var>
                        nt := skipspaces         ' Variable has to agree
                        if not isvar(nt)
                           abort @syn
                        if fixvar(nt) == a       ' If match, continue after
                           quit                  '  the matching NEXT
            157: ' NEXT <var>
               nt := spaces
               if not isvar(nt)      ' Get variable name to match up
                  abort @syn
               a := fixvar(nt)
               vars[a] += forStep[a]             ' Increment or decrement the
               if forStep[a] < 0                 '  FOR variable and check for
                  b := vars[a] => forLimit[a]
               else                              '  the limit value
                  b := vars[a] =< forLimit[a]
               if b                              ' If continuing loop, go to
                  nextlineloc := forLoop[a]      '  statement after FOR
               tp++
            159: ' OUTA [ <expr>{..<expr>} ] = <expr>
               a := getAddress(".")
               if a > $FFFF          ' Set output register bit or range of bits
                  b := a & $FF       '  to the supplied value.  Note DIRA is
                  a := (a >> 8) & $FF '  set implicitly
                  outa[a..b] := specialExpr
                  dira[a..b]~~
                  repeat c from a to b
                     outputs |= |< c
               else
                  outa[a] := specialExpr
                  dira[a]~~
                  outputs |= |< a
            160: ' PAUSE <expr> {,<expr>}
               if pauseTime == 0                 ' If no active pause time, set it
                  spaces                         '  with a minimum time of 50us
                  pauseTime := expr * 1000
                  if spaces == ","               ' First (or only) value is in ms
                     skipspaces
                     pauseTime += expr           ' Second value is in us
                  pauseTime #>= 50
               if pauseTime < 10_050             ' Normally pause at most 10ms at a time,
                  waitcnt(pauseTime * uS + cnt)  '  but, if that would leave < 50us,
                  pauseTime := 0                 '   pause the whole amount now
               else
                  a := pauseTime <# 10_000
                  waitcnt(a * uS + cnt)          ' Otherwise, pause at most 10ms and
                  nextlineloc := thisLine        '  re-execute the PAUSE for the rest
                  pauseTime -= 10_000
            166: ' BYTE [ <expr> ] = <expr>
               a := getAddress(" ")  ' Set byte value to specified expression
               byte[a] := specialExpr
            167: ' WORD [ <expr> ] = <expr>
               a := getAddress(" ")  ' Set word value to specified expression
               word[a] := specialExpr
            168: ' LONG [ <expr> ] = <expr>
               a := getAddress(" ")  ' Set long value to specified expression
               long[a] := specialExpr
            170: ' PHSA =            ' Set counter A phase register
               PHSA := specialExpr
            171: ' PHSB =            ' Set counter B phase register
               PHSB := specialExpr
            172: ' FRQA =             ' Set counter A frequency register
               FRQA := specialExpr
            173: ' FRQB =             ' Set counter B frequence register
               FRQB := specialExpr
            174: ' CTRA =             ' Set counter A control register
               CTRA := specialExpr
            175: ' CTRB =             ' Set counter B control register
               CTRB := specialExpr
            176: ' DISPLAY <expr> {,<expr>}
               spaces                ' Output bytes using specified expressions
               ser.tx(expr)
               repeat while spaces == ","
                  skipspaces
                  ser.tx(expr)
            178: ' LET <var> = <expr>
               nt := spaces          ' Set variable to expression (using LET)
               if not isvar(nt)
                  abort @syn
               tp++
               vars[fixvar(nt)] := specialExpr
            179: ' STOP               ' Stop program execution
               nextlineloc := eop-2
               return
            180: ' END                ' Stop program execution
               nextlineloc := eop-2
               return
            181: ' EEPROM [ <expr> ] = <expr>
               a := getAddress(",")  ' Set EEPROM byte to specified expression
               if i2c.writeByte(i2c#bootPin,i2c#EEPROM,a,specialExpr)
                  abort string("EEPROM write")
               repeat while i2c.WriteWait(i2c#bootPin,i2c#EEPROM,a)
                  if cnt - w > clkfreq / 10
                     abort string("EEPROM timeout")
            182: ' FILE = <expr>
               abort @syn            ' Unused in this version
            184: ' SPIN [{<expr>,}<expr>] or "<file>"
               abort @syn            ' Unused in this version
            185: ' COPY [<expr>],[<expr>] where <expr> are different
               abort @syn            ' Unused in this version
            186: ' DUMP <expr>,<expr> or DUMP [<expr>],<expr>
               if spaces == "["      ' Display a specified portion of hub
                  c := getAddress(",")' memory or EEPROM to the display
                  a := c & $F80000
                  b := c & $07FFFF
               else
                  a := -1
                  b := expr
               if spaces <> ","
                  abort @syn
               skipspaces
               dumpMemory(a,b,expr)
            187: ' SERVO <expr>,<expr>{,<expr>}
               a := expr             ' Set servo to specified position
               if a < 0 or a > 15    ' Parameters are I/O pin #, pulse width
                  abort string("Servo pin < 0 or > 15")
               if spaces <> ","
                  abort @syn
               skipspaces
               b := expr             ' Invalid values turn off servo
               if spaces == ","
                  skipspaces
                  c := expr          ' Optional is ramp speed value
                  if c < 0 or c > 63
                     abort string("Ramp speed < 0 or > 63")
                  srv.SetRamp(a,b,RampSpeed[c])
               else
                  srv.Set(a,b)
               if b < srv#LowRange or b > srv#HighRange
                  servosOn &= ! |< a ' Mark I/O pin not in use
               else
                  servosOn |= |< a    ' Mark I/O pin in use
            188: ' BAUD = <expr>      ' Set Baud and reinitialize serial I/O
               initSerial(specialExpr)
            189: ' REBOOT             ' Reboot
               reboot
            139: ' RND = <expr>       ' Initialize the random seed
               rv := specialExpr
      else                            ' Unknown statement token
         abort(@syn)
      if spaces == ":"                ' ":" separates statements on a line
         restart := 1
         tp++

'' Reinitialize the serial port

pri initSerial(Baud)
   if ina[USB_Rx] == 0                ' Check to see if USB port is powered
      dira[USB_Tx] := 1
      outa[USB_Tx] := 0               ' Force Prop Tx line LOW if USB not connected
      serialIsUSB := false
      ser.start(Aux_RxTx, Aux_RxTx, %11000, Baud) ' Ignore echo
   else
      serialIsUSB := true
      ser.start(USB_Rx, USB_Tx, %00000, Baud)

'' Process a line of program text.  This can be a string or a user entered line.
'' Check for a break key and halt execution if present.  Proceed from statement
'' to statement if not stopped.  Clean up servos, I/O state, and PAUSE bookkeeping
'' if program stopped.

pri doline(s) | c                     ' Execute the string in s or wait for input
   curlineno := -1
   if ser.breakCheck(ser#Esc)         ' Was the "break key" pressed?
      repeat ser#rxBufSize
         if ser.rx == ser#Esc         ' Flush input up to first "break key"
            quit
      nextlineloc := eop-2            ' Stop the program
   if nextlineloc < eop-2
      curlineno := wordat(nextlineloc)
      tp := nextlineloc + 2
      nextlineloc := tp + strsize(tp) + 1
      outa[Activity] := 1             ' Activity LED on while executing Basic
      texec
   else
      pauseTime := 0                  ' Stop any PAUSE in effect
      repeat c from 0 to 15           ' Turn off any servos in use
         if servosOn & |< c
            srv.Set(c,0)
      servosOn := 0
      repeat c from 0 to 27           ' Turn off any I/O pins in use
         if outputs & |< c
            dira[c]~
            outa[c]~
      outputs := 0
      outa[Activity] := 0             ' Turn off activity LED
      if s
         bytemove(tp:=@tline,s,strsize(s)+1) ' Execute the supplied string
      else
         putlinet(string("OK"))       ' Get a text line
         getline
      c := spaces                     ' Look for a line number
      if "0" =< c and c =< "9"
         insertline                   ' Insert the line in the proper place
         nextlineloc := eop - 2
      else
         tokenize                     ' Tokenize the line, then execute it
         if spaces
            outa[Activity] := 1       ' Activity LED on when executing Basic
            texec

'' Return the value of a numeric constant next in the input stream.
'' This can be a quoted character constant or a decimal, hexadecimal,
'' or binary format numeric constant.

PRI getAnyNumber | c, t
   case c := byte[tp]
      quote:
         if result := byte[++tp]
            if byte[++tp] == quote
              tp++
            else
               abort string("missing closing quote")
         else
            abort string("end of line in string")
      "$":
         c := byte[++tp]
         if (t := hexDigit(c)) < 0
            abort string("invalid hex character")
         result := t
         c := byte[++tp]
         repeat until (t := hexDigit(c)) < 0
            result := result << 4 | t
            c := byte[++tp]
      "%":
         c := byte[++tp]
         if not (c == "0" or c == "1")
            abort string("invalid binary character")
         result := c - "0"
         c := byte[++tp]
         repeat while c == "0" or c == "1"
            result := result << 1 | (c - "0")
            c := byte[++tp]
      "0".."9":
         result := c - "0"
         c := byte[++tp]
         repeat while c => "0" and c =< "9"
            result := result * 10 + c - "0"
            c := byte[++tp]
      other:
        abort string("invalid literal value")

'' Convert hexadecimal character to the corresponding value or -1 if invalid.

PRI hexDigit(c)
   if c => "0" and c =< "9"
      return c - "0"
   if c => "A" and c =< "F"
      return c - "A" + 10
   if c => "a" and c =< "f"
      return c - "a" + 10
   return -1

'' This routine dumps a portion of the RAM/ROM to the display (pin == -1).
'' If pin is not -1, it is an EEPROM address in the form required by the
'' I2C routines in i2cSpiInit.  The specified address is or'd with this.
'' The format is 8 bytes wide with hexadecimal and ASCII.

PUB dumpMemory(pin,addr,size) | i, c, p, first, buf0, buf1, buf2
   addr &= $7FFFF
   first := true
   p := addr & $7FFF8
   repeat while p < (addr + size)
      if first
         ser.hex(addr,5)
         first := false
      else
         ser.hex(p,5)
      ser.tx(":")
      repeat i from 0 to 7
         byte[@buf0][i] := " "
         if p => addr and p < (addr + size)
            if pin <> -1
               c := i2c.readByte(i2c#bootPin,i2c#EEPROM,p)
               if c < 0
                  abort string("EEPROM read")
            else
               c := byte[p]
            ser.hex(c,2)
            if c => " " and c =< "~"
               byte[@buf0][i] := c
         else
            ser.tx(" ")
            ser.tx(" ")
         ser.tx(" ")
         p++
      buf2 := 0
      ser.tx("|")
      ser.str(@buf0)
      ser.tx("|")
      ser.str(@crlf)

DAT
'' Lookup table for Speeds 0 to 63

'Note: A formula can be applied that will produce values that are within 6.27%
'
'      For ramp speeds between 7 and 63 ...
'
'      RampSpeed = ((Speed - 6)* 70231)/1600
'
'      For speeds between 1 and 6, a polynomial fit could be used to produce
'      values that are within 3.1%, but with as few numbers that are used you
'      don't really gain much by not using a lookup table...
'
'      RampSpeed = (158 * Speed^3 - 742 * Speed^2 + 2888 * Speed + 9264)/1600
'
'Note:
'      RampSpeed - is equal to the delay from one Servo Increment to the next.  The resolution
'                  of the delay is 20ms, so a value of 50 = 1 second ; a value of 3000 = 1 minute

RampSpeed     word      1,7,8,10,12,16,22,43,83,123,184,222,261,299,339,379,426,473,520,567,615
              word      662,709,756,808,859,910,961,1012,1064,1115,1166,1207,1248,1288,1329,1370
              word      1411,1451,1492,1533,1574,1614,1655,1696,1737,1778,1818,1859,1900,1941,1981
              word      2022,2063,2104,2144,2185,2226,2267,2307,2348,2389,2430,2470

{{
```