

```
1  {{
2  Modified Tiny Basic for use with Propeller Demo Board and Hydra.
3  Original Tiny Basic written by Tomas Rokicki & Radical Eye Software.
4
5  Copyright (c) 2008 Michael Green. See end of file for terms of use.
6  }}
7
8  '' 2009-07-09 - Corrected WRITE and PRINT for case of -2,147,483,648
9  '' 2010-06-04 - Changed fsrw to Rokicki's version 2.6 for FAT16/FAT32
10 '' 2010-06-05 - Strip out all references to SD card I/O
11 '' Change all console I/O to use FullDuplexSerialMG
12 '' 2010-06-05 - Added SERVO statement to control servos
13 '' Changed getline to handle possible half duplex input
14 '' 2010-06-18 - Added REBOOT statement
15 '' 2010-06-23 - Added comments. Turn off servo when program stops.
16 '' Other miscellaneous small fixes
17
18 obj
19   ser : "FullDuplexSerialMG"      ' Modified FullDuplexSerial
20   i2c : "Basic_I2C_Driver"        ' From Object Exchange
21   srv : "Servo32v7"              ' From Propeller Servo Controller
22
23 con
24   version = 1                    ' Major version
25   release = 001                  ' Minor release
26   testLevel = 0                  ' Test change level
27
28   USB_Rx = 31                    ' USB console receive pin
29   USB_Tx = 30                    ' USB console transmit pin
30   USB_Baud = 2400                ' USB console Baud
31
32   Aux_RxTx = 27                  ' Auxiliary serial I/O pin
33   Activity = 26                  ' Activity LED
34
35   progsz = 16384                  ' Space reserved for program
36   _clkmode = xtal1 + pll16x
37   _xinfreq = 5_000_000
38   _stack = 100                   ' Roughly 4 nested expressions
39   _free = (progsz + 12) / 4
40
41   maxstack = 20                  ' Maximum stack depth
42   linelen = 256                  ' Maximum input line length
43   quote = 34                    ' Double quote
44   caseBit = !32                  ' Uppercase/Lowercase bit
45   userPtr = $7FEC                ' Pointer to program memory
46   ranSeed = $12345678            ' Initial pseudo-random seed
47
48 '' Internal SPIN information at fixed locations in RAM/EEPROM image
49
50   spinPbase = $0006              ' must be $0010 (program base addr)
51   spinVbase = $0008              ' number of longs loaded times 4
52   spinDbase = $000A              ' above where $FFF9FFFF's get placed
53   spinPcurr = $000C              ' points to SPIN code
54   spinDcurr = $000E              ' points to local stack
55
56 var
57   long sp, tp, eop, rv, nextlineloc, curlineno, pauseTime
58   long vars[26], stack[maxstack], serialIsUSB
59   long forStep[26], forLimit[26], forLoop[26]
60   word outputs, servosOn
61   byte tline[linelen], tailLine[linelen], inVars[26], fileOpened
62
63 '' Table of tokens (zero-terminated strings). The program is scanned for these
64 '' and the original text is replaced by the number of the token + 128. The use
65 '' of tokens simplifies the parsing of the program. Note that a token should
66 '' not be a prefix of another token (RUN and RUNIT for example). If you need
67 '' something like this, the longer token should come first.
68
69 dat
```

```

70 tok0 byte "IF", 0
71 tok1 byte "THEN", 0
72 tok2 byte "INPUT", 0 ' INPUT {"<prompt>";} <var> {,<var>}
73 tok3 byte "PRINT", 0 ' PRINT {USING "<format>";} ...
74 tok4 byte "GOTO", 0
75 tok5 byte "GOSUB", 0
76 tok6 byte "RETURN", 0
77 tok7 byte "REM", 0
78 tok8 byte "NEW", 0
79 tok9 byte "LIST", 0
80 tok10 byte "RUN", 0
81 tok11 byte "RND", 0
82 tok12 byte "OPEN", 0 ' OPEN " <file> ",<mode>
83 tok13 byte "READ", 0 ' READ <var> {,<var>}
84 tok14 byte "WRITE", 0 ' WRITE {USING "<format>";} ...
85 tok15 byte "CLOSE", 0 ' CLOSE
86 tok16 byte "DELETE", 0 ' DELETE " <file> "
87 tok17 byte "RENAME", 0 ' RENAME " <file> "," <file> "
88 tok18 byte "FILES", 0 ' FILES
89 tok19 byte "SAVE", 0 ' SAVE or SAVE [<expr>] or SAVE "<file>"
90 tok20 byte "LOAD", 0 ' LOAD or LOAD [<expr>] or LOAD "<file>"
91 tok21 byte "NOT", 0 ' NOT <logical>
92 tok22 byte "AND", 0 ' <logical> AND <logical>
93 tok23 byte "OR", 0 ' <logical> OR <logical>
94 tok24 byte "SHL", 0 ' <expr> SHL <expr>
95 tok25 byte "SHR", 0 ' <expr> SHR <expr>
96 tok26 byte "FOR", 0 ' FOR <var> = <expr> TO <expr>
97 tok27 byte "TO", 0
98 tok28 byte "STEP", 0 ' optional STEP <expr>
99 tok29 byte "NEXT", 0 ' NEXT <var>
100 tok30 byte "INA", 0 ' INA [ <expr> ]
101 tok31 byte "OUTA", 0 ' OUTA [ <expr> ] = <expr>
102 tok32 byte "PAUSE", 0 ' PAUSE <time ms> {,<time us>}
103 tok33 byte "USING", 0 ' PRINT USING "<format>"; ...
104 tok34 byte "ROL", 0 ' <expr> ROL <expr>
105 tok35 byte "ROR", 0 ' <expr> ROR <expr>
106 tok36 byte "SAR", 0 ' <expr> SAR <expr>
107 tok37 byte "REV", 0 ' <expr> REV <expr>
108 tok38 byte "BYTE", 0 ' BYTE [ <expr> ]
109 tok39 byte "WORD", 0 ' WORD [ <expr> ]
110 tok40 byte "LONG", 0 ' LONG [ <expr> ]
111 tok41 byte "CNT", 0
112 tok42 byte "PHSA", 0
113 tok43 byte "PHSB", 0
114 tok44 byte "FRQA", 0
115 tok45 byte "FRQB", 0
116 tok46 byte "CTRA", 0
117 tok47 byte "CTRB", 0
118 tok48 byte "DISPLAY", 0 ' DISPLAY <expr> {,<expr>}
119 tok49 byte "KEYCODE", 0 ' KEYCODE
120 tok50 byte "LET", 0
121 tok51 byte "STOP", 0
122 tok52 byte "END", 0
123 tok53 byte "EEPROM", 0 ' EEPROM[ <expr> ]
124 tok54 byte "FILE", 0 ' FILE
125 tok55 byte "MEM", 0 ' MEM
126 tok56 byte "SPIN", 0 ' SPIN [<expr>] or SPIN "<file>"
127 tok57 byte "COPY", 0 ' COPY [<expr>],"<file>" or COPY "<file>",<expr>] or
    ' COPY [<expr>],<expr> where <expr> are different
128
129 tok58 byte "DUMP", 0 ' DUMP <expr>,<expr> or DUMP [<expr>],<expr>
130 tok59 byte "SERVO", 0 ' SERVO <expr>,<expr>
131 ' SERVO <expr>,<expr>,<expr>
132 tok60 byte "BAUD", 0 ' BAUD = <expr>
133 tok61 byte "REBOOT", 0 ' REBOOT
134
135 '' This is a table of offsets of the token text strings. The token scanner
136 '' starts at "toks" and stops at "tokx" (or when a match occurs). You can
137 '' add to the end of the table. The actual token stored is +128.
138

```

```

139   toks word @tok0, @tok1, @tok2, @tok3, @tok4, @tok5, @tok6, @tok7
140         word @tok8, @tok9, @tok10, @tok11, @tok12, @tok13, @tok14, @tok15
141         word @tok16, @tok17, @tok18, @tok19, @tok20, @tok21, @tok22, @tok23
142         word @tok24, @tok25, @tok26, @tok27, @tok28, @tok29, @tok30, @tok31
143         word @tok32, @tok33, @tok34, @tok35, @tok36, @tok37, @tok38, @tok39
144         word @tok40, @tok41, @tok42, @tok43, @tok44, @tok45, @tok46, @tok47
145         word @tok48, @tok49, @tok50, @tok51, @tok52, @tok53, @tok54, @tok55
146         word @tok56, @tok57, @tok58, @tok59, @tok60, @tok61
147   tokx word
148
149   syn byte "Syntax Error", 0
150   ln byte "Invalid Line Number", 0
151
152   crlf byte ser#Cr, ser#Lf, 0 ' New line
153
154   '' Main method. Initializes program, displays banner, then does main loop
155   '' which reads an input line, processes it, and displays any error messages.
156
157   PUB main | err, s
158       initSerial(USB_Baud)          ' Initialize serial port (to USB_Baud)
159       outa[Activity] := 0            ' Activity LED off initially
160       dira[Activity] := 1
161       rv := ranSeed                  ' Set initial random seed
162       pauseTime := 0
163       outputs := 0
164       long[userPtr] := userPtr - progsz ' Allocate memory
165       ser.str(string("PSC Basic"))
166       if version > 0 or release > 0 or testLevel > 0
167           ser.str(string(" Version "))
168           ser.dec(version)
169           ser.tx(".")
170           if release < 100
171               ser.tx("0")
172           if release < 10
173               ser.tx("0")
174           ser.dec(release)
175           if testLevel > 0
176               ser.tx("a"+testLevel-1)
177       ser.str(@crlf)
178       servosOn := 0                  ' No servos in use
179       srv.start                        ' Start servo controller
180       srv.ramp                        ' Start servo ramp handler
181       waitcnt(clkfreq + cnt)
182       clearall                        ' Clear program space and variables
183       s := 0
184       curlineno := -1
185       repeat
186           err := \doline(s)          ' Read an input line and interpret it
187           s := 0
188           outa[Activity] := 0
189           if err                      ' Show any error messages
190               showError(err)
191
192   '' Displays error message (address of string in "err") including line number
193   '' if program is executing. Sets next line address to end of program to stop
194   '' execution.
195
196   PRI showError(err)
197       if curlineno => 0
198           ser.str(string("IN LINE "))
199           ser.dec(curlineno)
200           ser.tx(" ")
201       putlnet(err)
202       nextlineloc := eop - 2
203
204   '' Get a line of text from the input device. Handles backspace and return.
205   '' If input is programming port, echo it and handle backspace. If input is
206   '' auxiliary serial port, flush transmitter and receiver initially and don't
207   '' echo. Set "tp" to start of text string.
    
```

```
208
209 PRI getline | i, c
210     ifnot serialIsUSB
211         ser.txflush           ' Wait for transmit to finish
212         ser.rxflush           ' Remove any echoed characters
213     i := 0
214     repeat
215         c := ser.rx
216         if c == ser#Bsp
217             if i > 0
218                 if serialIsUSB           ' Echo input if USB console
219                     ser.str(string(ser#Bsp, " ", ser#Bsp))
220                 i--
221             elseif c == ser#Cr
222                 if serialIsUSB           ' Echo input if USB console
223                     ser.tx(c)
224                     ser.tx(ser#Lf)
225                     tline[i] := 0
226                     tp := @tline
227                     return
228             elseif i < linelen-1
229                 if serialIsUSB           ' Echo input if USB console
230                     ser.tx(c)
231                     tline[i++] := c
232
233 '' Display a tokenized text line.  Replace any tokens (byte >= 128) with the
234 '' text from the token table.  If an unknown token is found, display its value
235 '' in braces {}.  If token is not REM, put a space after it.
236
237 pri putlinet(s) | c, ntoks
238     ntoks := (@tokx - @toks) / 2
239     repeat while c := byte[s++]
240         if c => 128
241             if (c - 128) < ntoks
242                 ser.str(@@toks[c])
243                 if c <> 7 ' REM
244                     ser.tx(" ")
245             else
246                 ser.tx("{")
247                 ser.dec(c)
248                 ser.tx("}")
249         else
250             ser.tx(c)
251     ser.str(@crlf)
252
253 '' Skip spaces in the input line ("tp" is the pointer) and return the first
254 '' non-blank character found.  A zero byte also terminates the operation.
255
256 pri spaces | c
257     repeat
258         c := byte[tp]
259         if c == 0 or c > " "
260             return c
261         tp++
262
263 '' Skip a single character in the input line (unless the zero byte terminator)
264 '' and call "spaces" to return the next non-blank.
265
266 pri skipspace
267     if byte[tp]
268         tp++
269     return spaces
270
271 '' The next input item is assumed to be a decimal number.  Return the value of
272 '' that number and leave "tp" pointing to the non-digit following the number.
273 '' If the next input character is not a digit, return zero.
274
275 pri parseliteral | r, c
276     r := 0
```

```
277 repeat
278   c := byte[tp]
279   if c < "0" or c > "9"
280     return r
281   r := r * 10 + c - "0"
282   tp++
283
284 '' Move the remainder of the program beginning at "at" upwards in memory a
285 '' distance "delta" to make room for a new line.
286
287 pri movprog(at, delta)
288   if eop + delta + 2 - long[userPtr] > progsz
289     abort string("NO MEMORY")
290   bytemove(at+delta, at, eop-at)
291   eop += delta
292
293 '' Convert upper case letters to lower case and return the index of the letter
294
295 pri fixvar(c)
296   if c == "A"
297     c -= 32
298   return c - "A"
299
300 '' Return true if the supplied character is a letter (a variable name)
301
302 pri isvar(c)
303   c := fixvar(c)
304   return c == 0 and c < 26
305
306 '' Scan the input line. Replace any tokens with their 128+ code. Any
307 '' characters within double quotes (") are copied literally as are comments
308 '' beginning with the REM token.
309
310 pri tokenize | tok, c, at, put, state, i, j, ntoks
311   ntoks := (@tokx - @toks) / 2
312   at := tp
313   put := tp
314   state := 0
315   repeat while c := byte[at]
316     if c == quote
317       if state == "Q"
318         state := 0
319       elseif state == 0
320         state := "Q"
321     if state == 0
322       repeat i from 0 to ntoks-1
323         tok := @toks[i]
324         j := 0
325         repeat while byte[tok] and ((byte[tok] ^ byte[j+at]) & caseBit) == 0
326           j++
327         tok++
328         if byte[tok] == 0 and not isvar(byte[j+at])
329           byte[put++] := 128 + i
330           at += j
331           if i == 7
332             state := "R"
333           else
334             repeat while byte[at] == " "
335               at++
336             state := "F"
337           quit
338         if state == "F"
339           state := 0
340         else
341           byte[put++] := byte[at++]
342     else
343       byte[put++] := byte[at++]
344   byte[put] := 0
345
```

```
346 `` Return the byte-aligned word value at location "loc"
347
348 pri wordat(loc)
349     return (byte[loc]<<8)+byte[loc+1]
350
351 `` Return the address of the line in the program that has a line number greater
352 `` or equal to the supplied line number.
353
354 pri findline(lineno) | at
355     at := long[userPtr]
356     repeat while wordat(at) < lineno
357         at += 3 + strsize(at+2)
358     return at
359
360 `` Scan the line number, tokenize the rest of the line, and insert it
361 `` in the appropriate position in the stored program. Delete any existing
362 `` line with the same line number.
363
364 pri insertline | lineno, fc, loc, locat, newlen, oldlen
365     lineno := parseliteral
366     if lineno < 0 or lineno => 65535
367         abort @ln
368     tokenize
369     fc := spaces
370     loc := findline(lineno)
371     locat := wordat(loc)
372     newlen := 3 + strsize(tp)
373     if locat == lineno
374         oldlen := 3 + strsize(loc+2)
375         if fc == 0
376             movprog(loc+oldlen, -oldlen)
377         else
378             movprog(loc+oldlen, newlen-oldlen)
379     elseif fc
380         movprog(loc, newlen)
381     if fc
382         byte[loc] := lineno >> 8
383         byte[loc+1] := lineno
384         bytemove(loc+2, tp, newlen-2)
385
386 `` Clear all variables to zero, clear the data stack, initialize the PAUSE
387 `` time, and set the program execution point to the beginning of the program
388
389 pri clearvars
390     longfill(@vars, 0, 26)
391     pauseTime := 0
392     nextlineloc := long[userPtr]
393     sp := 0
394
395 `` Clear the current program and the data stack
396
397 pri newprog
398     byte[long[userPtr]][0] := 255
399     byte[long[userPtr]][1] := 255
400     byte[long[userPtr]][2] := 0
401     eop := long[userPtr] + 2
402     nextlineloc := eop - 2
403     sp := 0
404
405 `` Clear the program space, variables, and data stack
406
407 pri clearall
408     newprog
409     clearvars
410
411 `` Check for a possible data stack overflow and push the current program
412 `` pointer into the stack (for a GOSUB)
413
414 pri pushstack
```

```

415   if sp => constant(maxstack-1)
416       abort string("RECURSION ERROR")
417   stack[sp++] := nextlineloc
418
419   `` Process a possible EEPROM address, I/O pin number, or other [] bracketed
420   `` value depending on the "delim" value. If ".", then a simple I/O pin
421   `` number is expected and checked for range 0-31. A pin range like "1..5"
422   `` is allowed. If present, the first value is returned in the high word of
423   `` the result and the second value is returned in the low word of the result.
424   `` If ",", then an EEPROM address is expected with the I/O pin number as the
425   `` first value, then a comma followed by the EEPROM address. The pin number
426   `` in this case must be even and the EEPROM address in the range 0..$7FFFF.
427   `` The pin number is shifted into position for the sdspiFemto low level I2C
428   `` routines to use directly. Any other "delim" value allows any single
429   `` expression value to be in the brackets and this is returned unchanged.
430
431   pri getAddress(delim) | t
432       if spaces <> "["
433           abort @syn
434       skipspace
435       result := expr
436       if delim == "." and (result < 0 or result > 31)
437           abort string("Invalid pin number")
438       if delim == "." or delim == ","
439           if spaces == delim
440               if delim == "."
441                   if byte[+tp] <> "."
442                       abort @syn
443                   result <<= 8
444                   skipspace
445                   t := expr
446                   if t < 0 or t > 31
447                       abort string("Invalid pin number")
448                   result |= t | $10000
449               else
450                   `` Handle the form <expr>..<expr>
451                   if result & 1 or result < 0 or result > 31
452                       abort string("Invalid pin number")
453                   skipspace
454                   result := (result << 18) | (expr & $7FFFF)
455               elseif delim == ","
456                   result &= $7FFFF
457       if spaces <> "]"
458           abort @syn
459       tp++
460
461   `` This is the "lowest level" routine in the recursive descent expression
462   `` parser.
463   pri factor | tok, t, i
464       tok := spaces
465       tp++
466       case tok
467           "("
468               `` Recurse to handle ( <expr> )
469               t := expr
470               if spaces <> ")"
471                   abort @syn
472               tp++
473               return t
474           "a".."z", "A".."Z"
475               `` Scalar variable, return value
476               return vars[fixvar(tok)]
477           158: " INA [ <expr>{..<expr>} ] "
478               `` Input register (INA) bit/bits
479               t := getAddress(".")
480               if t > $FFFF
481                   tok := t & $FF
482                   t := (t >> 8) & $FF
483               repeat i from t to tok
484                   outputs &= ! |< i
485               dira[t..tok]~
486               return ina[t..tok]

```

```

484     else
485         outputs &= ! |< t
486         dira[t]~
487         return ina[t]
488 166: ' BYTE [ <expr> ]      ' Hub memory byte value given address
489     return byte[getAddress(" ")]
490 167: ' WORD [ <expr> ]     ' Hub memory word value given address
491     return word[getAddress(" ")]
492 168: ' LONG [ <expr> ]    ' Hub memory long value given address
493     return long[getAddress(" ")]
494 181: ' EEPROM [ <expr> ]  ' EEPROM byte value given pin# / address
495     t := getAddress(" ")
496     t := i2c.readByte(i2c#bootPin,i2c#EEPROM,t)
497     if t < 0
498         abort string("EEPROM read")
499     return t
500 182: ' FILE                ' Byte from currently open file
501     abort @syn
502 183: ' MEM                 ' Return available program memory size
503     return progsz - (eop - long[userPtr] )
504 169: ' CNT                 ' Return system clock register (CNT)
505     return CNT
506 170: ' PHSA               ' Return counter A phase value
507     return PHSA
508 171: ' PHSB               ' Return counter B phase value
509     return PHSB
510 172: ' FRQA               ' Return counter A frequency value
511     return FRQA
512 173: ' FRQB               ' Return counter B frequency value
513     return FRQB
514 174: ' CTRA               ' Return counter A control value
515     return CTRA
516 175: ' CTRB               ' Return counter B control value
517     return CTRB
518 177: ' KEYCODE            ' Return current input character (or -1)
519     return ser.rx
520 139: ' RND <factor>       ' Return pseudo-random value and
521     return rv? // factor    ' limit range to supplied value
522 "-":                      ' Unary negate
523     return - factor
524 "!":                      ' Unary bitwise not
525     return ! factor
526 "$", "%", quote, "0".."9": ' Hex, binary, character, decimal constant
527     --tp
528     return getAnyNumber
529 other:
530     abort @syn
531
532 pri shifts | tok, t      ' Shift operations
533     t := factor
534     tok := spaces
535     if tok == 152        ' SHL - Left logical shift
536         tp++
537         return t << factor
538     elseif tok == 153    ' SHR - Right logical shift
539         tp++
540         return t >> factor
541     elseif tok == 162    ' ROL - Left rotate
542         tp++
543         return t <- factor
544     elseif tok == 163    ' ROR - Right rotate
545         tp++
546         return t -> factor
547     elseif tok == 164    ' SAR - Right arithmetic shift
548         tp++
549         return t ~> factor
550     elseif tok == 165    ' REV - Reverse specified # of bits
551         tp++
552         return t >< factor

```



```
553     else
554         return t
555
556 pri bitFactor | tok, t
557     t := shifts
558     repeat
559         tok := spaces
560         if tok == "&"          ' Bitwise and operation
561             tp++
562             t &= shifts
563         else
564             return t
565
566 pri bitTerm | tok, t
567     t := bitFactor
568     repeat
569         tok := spaces
570         if tok == "|"          ' Bitwise or operation
571             tp++
572             t |= bitFactor
573         elseif tok == "^"      ' Bitwise exclusive or operation
574             tp++
575             t ^= bitFactor
576         else
577             return t
578
579 pri term | tok, t
580     t := bitTerm
581     repeat
582         tok := spaces
583         if tok == "*"          ' Multiplication
584             tp++
585             t *= bitTerm
586         elseif tok == "/"      ' Division or modulus
587             if byte[+tp] == "/"
588                 tp++
589                 t /= bitTerm
590             else
591                 t /= bitTerm
592         else
593             return t
594
595 pri arithExpr | tok, t
596     t := term
597     repeat
598         tok := spaces
599         if tok == "+"          ' Addition
600             tp++
601             t += term
602         elseif tok == "-"      ' Subtraction
603             tp++
604             t -= term
605         else
606             return t
607
608 pri compare | op, a, b, c      ' Comparison operations
609     a := arithExpr
610     op := 0
611     spaces
612     repeat
613         c := byte[tp]          ' Allow combinations of characters
614         case c                  ' to occur in any order
615             "<": op |= 1
616             tp++
617             ">": op |= 2
618             tp++
619             "=: op |= 4
620             tp++
621             other: quit
```

```
622   case op
623       0: return a
624       1: return a < arithExpr
625       2: return a > arithExpr
626       3: return a <> arithExpr
627       4: return a == arithExpr
628       5: return a <= arithExpr
629       6: return a >= arithExpr
630       7: abort string("Invalid comparison")
631
632 pri logicNot | tok
633     tok := spaces
634     if tok == 149                ' NOT - Logical not
635         tp++
636         return not compare
637     return compare
638
639 pri logicAnd | t, tok
640     t := logicNot
641     repeat
642         tok := spaces
643         if tok == 150            ' AND - Logical and
644             tp++
645             t := t and logicNot
646         else
647             return t
648
649 pri expr | tok, t
650     t := logicAnd
651     repeat
652         tok := spaces
653         if tok == 151            ' OR - Logical or
654             tp++
655             t := t or logicAnd
656         else
657             return t
658
659 '' Handle "pseudo-assignment" where some special token comes first, then an
660 '' assignment (equal sign) and an arbitrary expression. Return the value of
661 '' the expression.
662
663 pri specialExpr
664     if spaces <> "="
665         abort @syn
666     skipspaces
667     return expr
668
669 '' This routine handles all statements and sequences of statements separated
670 '' by ":". It handles the interrupted multiple interpretation of PAUSEs as
671 '' well as the FOR / NEXT looping.
672
673 pri texec | ht, nt, restart, thisLine, uS, a,b,c,d,w, f0,f1,f2,f3,f4,f5,f6,f7
674     uS := clkfreq / 1_000_000
675     thisLine := tp - 2
676     restart := 1
677     repeat while restart
678         restart := 0
679         ht := spaces
680         if ht == 0                ' Quit at the end of the line
681             return
682         nt := skipspaces
683         if isvar(ht) and nt == "=" ' Variable assignment
684             tp++
685             vars[fixvar(ht)] := expr
686         elseif ht => 128
687             case ht
688                 128: ' IF <expr> THEN ' If <expr> succeeds, continue
689                     a := expr        ' interpreting rest of the line
690                     if spaces <> 129
```

```

691         abort string("MISSING THEN")
692     skipspace
693     if not a
694         return
695     restart := 1
696 130: ' INPUT {"<prompt>";} <var> {, <var>}
697     if nt == quote      Display prompt if present
698         c := byte[++tp]
699         repeat while c <> quote and c
700             ser.tx(c)
701             c := byte[++tp]
702         if c <> quote
703             abort @syn
704         if skipspace <> ";"
705             abort @syn
706         nt := skipspace
707     if not isvar(nt)      Must be at least one variable
708         abort @syn
709     b := 0
710     inVars[b++] := fixvar(nt)
711     repeat while skipspace == ";"
712         nt := skipspace      Parse the sequence of variables
713         if not isvar(nt) or b == 26
714             abort @syn
715         inVars[b++] := fixvar(nt)
716     getline              Get a line of input text & tokenize
717     tokenize
718     repeat a from 1 to b      Store any values provided in variables
719         vars[inVars[a-1]] := expr
720         if a < b
721             if space == ";"
722                 skipspace
723 131: ' PRINT
724     a := 0
725     repeat
726         nt := space
727         if nt == 0 or nt == ":"
728             quit
729         if nt == quote      Display string constant
730             tp++
731             repeat
732                 c := byte[tp++]
733                 if c == 0 or c == quote
734                     quit
735                 ser.tx(c)
736                 a++
737         else                Display <expr> value in decimal
738             d := (b := expr) == negx
739             if b < 0
740                 b := -(b+d)
741                 ser.tx("-")
742                 a++
743             c := 1_000_000_000
744             d~
745             repeat 10
746                 if b >= c
747                     ser.tx(b / c + "0")
748                     a++
749                     b /= c
750                     d~~
751                 elseif d or c == 1
752                     ser.tx("0")
753                     a++
754                     c /= 10
755             nt := space
756             if nt == ";"
757                 tp++
758             elseif nt == ","      Comma as separator, use display columns
759                 ser.tx(" ")

```

```

760         a++
761         repeat while a & 7
762             ser.tx(" ")
763             a++
764             tp++
765         elseif nt == 0 or nt == ":"
766             ser.str(@crlf) ' Terminate display line unless ":"
767             quit
768         else
769             abort @syn
132, 133: ' GOTO, GOSUB ' GOTO or GOSUB label
771         a := expr
772         if a < 0 or a => 65535
773             abort @ln
774         b := findline(a) ' Use nearest line greater than requested
775         if wordat(b) <> a
776             abort @ln
777         if ht == 133 ' If GOSUB, save pointer to next line
778             pushstack
779         nextlineloc := b
780 134: ' RETURN ' Return to saved pointer to next line
781         if sp == 0 ' unless there's a stack underflow
782             abort string("INVALID RETURN")
783         nextlineloc := stack[--sp]
784 135: ' REM ' Ignore rest of remark line
785         repeat while skipspace
786 136: ' NEW ' Clear variables and erase program
787         clearall
788 137: ' LIST {<expr> {,<expr>}}
789         b := 0 ' Default line range
790         c := 65535
791         if spaces <> 0 ' At least one parameter
792             b := c := expr
793             if spaces == "," ' List requested line or range of lines
794                 skipspace
795                 c := expr
796         a := long[userPtr]
797         repeat while a+2 < eop
798             d := wordat(a)
799             if d => b and d <= c
800                 ser.dec(d)
801                 ser.tx(" ")
802                 putlnet(a+2)
803             a += 3 + strsize(a+2)
804 138: ' RUN ' Clear variables and start executing
805         clearvars
806 140: ' OPEN " <file> ", R/W/A
807         abort @syn ' Not used in this version
808 141: ' READ <var> {, <var> }
809         abort @syn ' Not used in this version
810 142: ' WRITE ...
811         abort @syn ' Not used in this version
812 143: ' CLOSE
813         abort @syn ' Not used in this version
814 144: ' DELETE " <file> "
815         abort @syn ' Not used in this version
816 145: ' RENAME " <file> ", " <file> "
817         abort @syn ' Not used in this version
818 146: ' FILES
819         abort @syn ' Not used in this version
820 147: ' SAVE or SAVE [ <expr> ]
821         if (nt := spaces) == quote
822             abort @syn
823         else ' Save program to EEPROM
824             if nt == "[" ' Align save area for paged writes
825                 a := getaddress(" ") + 64
826                 if (a & 63) == 63
827                     a += 64
828                 a &= $7FFC0
    
```

```

829     else
830         a := (userPtr - progsz - 62) & $7FC0
831     nt := spaces
832     if nt <> 0 and nt <> ":"
833         abort @syn ' Write program to EEPROM
834     d := eop - long[userPtr] + 1
835     if i2c.writeWord(i2c#bootPin,i2c#EEPROM,a-2,d) ' Program size
836         abort string("Save EEPROM write")
837     w := cnt ' prepare to check for a timeout
838     repeat while i2c.WriteWait(i2c#bootPin,i2c#EEPROM,a-2)
839         if cnt - w > clkfreq / 10
840             abort string("Save EEPROM timeout")
841     repeat c from 0 to d step 64 ' Write the program itself
842         if i2c.writePage(i2c#bootPin,i2c#EEPROM,a+c,long[userPtr]+c,d-c<#64)
843             abort string("Save EEPROM write")
844         w := cnt ' prepare to check for a timeout
845         repeat while i2c.WriteWait(i2c#bootPin,i2c#EEPROM,a+c)
846             if cnt - w > clkfreq / 10
847                 abort string("Save EEPROM timeout")
148: ' LOAD or LOAD [ <expr> ]
849     if (nt := spaces) == quote
850         abort @syn
851     else ' Load program from EEPROM
852         if nt == "[" ' Align save area for paged writes
853             a := getaddress(",") + 64
854             if (a & 63) == 63
855                 a += 64
856             a &= $7FFC0
857         else
858             a := (userPtr - progsz - 62) & $7FC0
859         nt := spaces
860         if nt <> 0 and nt <> ":"
861             abort @syn ' Read program from EEPROM
862         d := i2c.readWord(i2c#bootPin,i2c#EEPROM,a-2)
863         if d < 0
864             abort string("Load EEPROM read")
865         d &= $FFFF
866         if d < 3 or d > progsz ' Read program size & check
867             abort string("Invalid program size")
868         c := @tailline ' Save statement tail
869         repeat while byte[c++] := byte[tp++]
870             tp := @tailline ' Scan copy after load
871         if i2c.readPage(i2c#bootPin,i2c#EEPROM,a,long[userPtr],d)
872             abort string("Load EEPROM read")
873         eop := long[userPtr] + d - 1
874         nextlineloc := eop - 2 ' Leave it stopped
154: ' FOR <var> = <expr> TO <expr> {STEP <expr>}
876         ht := spaces
877         if ht == 0
878             abort @syn
879         nt := skipspace
880         if not isvar(ht) or nt <> "="
881             abort @syn
882         a := fixvar(ht) ' Get FOR variable index
883         skipspace
884         vars[a] := expr
885         if spaces <> 155 ' TO - Save FOR limit
886             abort @syn
887         skipspace
888         forLimit[a] := expr
889         if spaces == 156 ' STEP - Save step size
890             skipspace
891             forStep[a] := expr
892         else
893             forStep[a] := 1 ' Default step is 1
894         if spaces
895             abort @syn
896         forLoop[a] := nextlineloc ' Save address of line
897         if forStep[a] < 0 ' following the FOR
    
```

```

898         b := vars[a] => forLimit[a]
899     else                                     ' Initially past the limit?
900         b := vars[a] =< forLimit[a]
901     if not b                               ' Search for matching NEXT
902         repeat while nextlineloc < eop-2
903             curlineno := wordat(nextlineloc)
904             tp := nextlineloc + 2
905             nextlineloc := tp + strsize(tp) + 1
906             if spaces == 157                ' NEXT <var>
907                 nt := skipspaces           ' Variable has to agree
908                 if not isvar(nt)
909                     abort @syn
910                 if fixvar(nt) == a         ' If match, continue after
911                     quit                   ' the matching NEXT
912 157: ' NEXT <var>
913     nt := spaces
914     if not isvar(nt)                       ' Get variable name to match up
915         abort @syn
916     a := fixvar(nt)
917     vars[a] += forStep[a]                  ' Increment or decrement the
918     if forStep[a] < 0                      ' FOR variable and check for
919         b := vars[a] => forLimit[a]
920     else                                   ' the limit value
921         b := vars[a] =< forLimit[a]
922     if b                                   ' If continuing loop, go to
923         nextlineloc := forLoop[a]         ' statement after FOR
924     tp++
925 159: ' OUTA [ <expr>{..  

926     a := getAddress("..  

927     if a > $FFFF                          ' Set output register bit or range of bits
928         b := a & $FF                      ' to the supplied value. Note DIRA is
929         a := (a >> 8) & $FF              ' set implicitly
930         outa[a..b] := specialExpr
931         dira[a..b]~~
932         repeat c from a to b
933             outputs |= |< c
934     else
935         outa[a] := specialExpr
936         dira[a]~~
937         outputs |= |< a
938 160: ' PAUSE <expr> {,<expr>}
939     if pauseTime == 0                     ' If no active pause time, set it
940         spaces                             ' with a minimum time of 50us
941         pauseTime := expr * 1000
942         if spaces == ","                  ' First (or only) value is in ms
943             skipspaces
944             pauseTime += expr             ' Second value is in us
945         pauseTime #>= 50
946     if pauseTime < 10_050                 ' Normally pause at most 10ms at a time,
947         waitcnt(pauseTime * uS + cnt)    ' but, if that would leave < 50us,
948         pauseTime := 0                   ' pause the whole amount now
949     else
950         a := pauseTime <# 10_000
951         waitcnt(a * uS + cnt)             ' Otherwise, pause at most 10ms and
952         nextlineloc := thisLine           ' re-execute the PAUSE for the rest
953         pauseTime -= 10_000
954 166: ' BYTE [ <expr> ] = <expr>
955     a := getAddress("..  

956     byte[a] := specialExpr
957 167: ' WORD [ <expr> ] = <expr>
958     a := getAddress("..  

959     word[a] := specialExpr
960 168: ' LONG [ <expr> ] = <expr>
961     a := getAddress("..  

962     long[a] := specialExpr
963 170: ' PHSA =
964     PHSA := specialExpr                  ' Set counter A phase register
965 171: ' PHSB =
966     PHSB := specialExpr                  ' Set counter B phase register

```

```

967 172: ' FRQA = ' Set counter A frequency register
968     FRQA := specialExpr
969 173: ' FRQB = ' Set counter B frequency register
970     FRQB := specialExpr
971 174: ' CTRA = ' Set counter A control register
972     CTRA := specialExpr
973 175: ' CTB = ' Set counter B control register
974     CTB := specialExpr
975 176: ' DISPLAY <expr> {,<expr>}
976     spaces ' Output bytes using specified expressions
977     ser.tx(expr)
978     repeat while spaces == ",",
979         skipspace
980         ser.tx(expr)
981 178: ' LET <var> = <expr>
982     nt := spaces ' Set variable to expression (using LET)
983     if not isvar(nt)
984         abort @syn
985     tp++
986     vars[fixvar(nt)] := specialExpr
987 179: ' STOP ' Stop program execution
988     nextlineloc := eop-2
989     return
990 180: ' END ' Stop program execution
991     nextlineloc := eop-2
992     return
993 181: ' EEPROM [ <expr> ] = <expr>
994     a := getAddress(",") ' Set EEPROM byte to specified expression
995     if i2c.writeByte(i2c#bootPin,i2c#EEPROM,a,specialExpr)
996         abort string("EEPROM write")
997     repeat while i2c.Wait(i2c#bootPin,i2c#EEPROM,a)
998         if cnt - w > clkfreq / 10
999             abort string("EEPROM timeout")
1000 182: ' FILE = <expr>
1001     abort @syn ' Unused in this version
1002 184: ' SPIN [{<expr>,<expr>} or "<file>"]
1003     abort @syn ' Unused in this version
1004 185: ' COPY [<expr>],[<expr>] where <expr> are different
1005     abort @syn ' Unused in this version
1006 186: ' DUMP <expr>,<expr> or DUMP [<expr>],[<expr>]
1007     if spaces == "[" ' Display a specified portion of hub
1008         c := getAddress(",") ' memory or EEPROM to the display
1009         a := c & $F80000
1010         b := c & $07FFFF
1011     else
1012         a := -1
1013         b := expr
1014     if spaces <> ",",
1015         abort @syn
1016     skipspace
1017     dumpMemory(a,b,expr)
1018 187: ' SERVO <expr>,<expr>{,<expr>}
1019     a := expr ' Set servo to specified position
1020     if a < 0 or a > 15 ' Parameters are I/O pin #, pulse width
1021         abort string("Servo pin < 0 or > 15")
1022     if spaces <> ",",
1023         abort @syn
1024     skipspace
1025     b := expr ' Invalid values turn off servo
1026     if spaces == ",",
1027         skipspace
1028         c := expr ' Optional is ramp speed value
1029         if c < 0 or c > 63
1030             abort string("Ramp speed < 0 or > 63")
1031         srv.SetRamp(a,b,RampSpeed[c])
1032     else
1033         srv.Set(a,b)
1034     if b < srv#LowRange or b > srv#HighRange
1035         servosOn &= ! |< a ' Mark I/O pin not in use

```

```

1036         else
1037             servosOn |= |< a      ' Mark I/O pin in use
1038         188: ' BAUD = <expr>      ' Set Baud and reinitialize serial I/O
1039             initSerial(specialExpr)
1040         189: ' REBOOT              ' Reboot
1041             reboot
1042         139: ' RAND = <expr>      ' Initialize the random seed
1043             rv := specialExpr
1044         else                      ' Unknown statement token
1045             abort(@syn)
1046         if spaces == "':"      ' ";" separates statements on a line
1047             restart := 1
1048             tp++
1049
1050     '' Reinitialize the serial port
1051
1052 pri initSerial(Baud)
1053     if ina[USB_Rx] == 0          ' Check to see if USB port is powered
1054         dira[USB_Tx] := 1
1055         outa[USB_Tx] := 0        ' Force Prop Tx line LOW if USB not connected
1056         serialIsUSB := false
1057         ser.start(Aux_RxTx, Aux_RxTx, %11000, Baud) ' Ignore echo
1058     else
1059         serialIsUSB := true
1060         ser.start(USB_Rx, USB_Tx, %00000, Baud)
1061
1062 '' Process a line of program text. This can be a string or a user entered line.
1063 '' Check for a break key and halt execution if present. Proceed from statement
1064 '' to statement if not stopped. Clean up servos, I/O state, and PAUSE bookkeeping
1065 '' if program stopped.
1066
1067 pri doline(s) | c                ' Execute the string in s or wait for input
1068     curlineno := -1
1069     if ser.breakCheck(ser#Esc)    ' Was the "break key" pressed?
1070         repeat ser#rxBufSize
1071             if ser.rx == ser#Esc  ' Flush input up to first "break key"
1072                 quit
1073             nextlineloc := eop-2  ' Stop the program
1074     if nextlineloc < eop-2
1075         curlineno := wordat(nextlineloc)
1076         tp := nextlineloc + 2
1077         nextlineloc := tp + strsize(tp) + 1
1078         outa[Activity] := 1        ' Activity LED on while executing Basic
1079         texec
1080     else
1081         pauseTime := 0            ' Stop any PAUSE in effect
1082         repeat c from 0 to 15     ' Turn off any servos in use
1083             if servosOn & |< c
1084                 srv.Set(c,0)
1085         servosOn := 0
1086         repeat c from 0 to 27     ' Turn off any I/O pins in use
1087             if outputs & |< c
1088                 dira[c]~
1089                 outa[c]~
1090         outputs := 0
1091         outa[Activity] := 0        ' Turn off activity LED
1092         if s
1093             bytemove(tp:=@tline,s,strsize(s)+1) ' Execute the supplied string
1094         else
1095             putlnet(string("OK"))  ' Get a text line
1096             getline
1097             c := spaces            ' Look for a line number
1098             if "0" <= c and c <= "9"
1099                 insertline        ' Insert the line in the proper place
1100                 nextlineloc := eop - 2
1101             else
1102                 tokenize          ' Tokenize the line, then execute it
1103                 if spaces
1104                     outa[Activity] := 1 ' Activity LED on when executing Basic
    
```



```
1105         texec
1106
1107     '' Return the value of a numeric constant next in the input stream.
1108     '' This can be a quoted character constant or a decimal, hexadecimal,
1109     '' or binary format numeric constant.
1110
1111 PRI getAnyNumber | c, t
1112     case c := byte[tp]
1113         quote:
1114             if result := byte[++tp]
1115                 if byte[++tp] == quote
1116                     tp++
1117                 else
1118                     abort string("missing closing quote")
1119             else
1120                 abort string("end of line in string")
1121         "s":
1122             c := byte[++tp]
1123             if (t := hexDigit(c)) < 0
1124                 abort string("invalid hex character")
1125             result := t
1126             c := byte[++tp]
1127             repeat until (t := hexDigit(c)) < 0
1128                 result := result << 4 | t
1129             c := byte[++tp]
1130         "%":
1131             c := byte[++tp]
1132             if not (c == "0" or c == "1")
1133                 abort string("invalid binary character")
1134             result := c - "0"
1135             c := byte[++tp]
1136             repeat while c == "0" or c == "1"
1137                 result := result << 1 | (c - "0")
1138             c := byte[++tp]
1139         "0".."9":
1140             result := c - "0"
1141             c := byte[++tp]
1142             repeat while c >= "0" and c <= "9"
1143                 result := result * 10 + c - "0"
1144             c := byte[++tp]
1145         other:
1146             abort string("invalid literal value")
1147
1148     '' Convert hexadecimal character to the corresponding value or -1 if invalid.
1149
1150 PRI hexDigit(c)
1151     if c >= "0" and c <= "9"
1152         return c - "0"
1153     if c >= "A" and c <= "F"
1154         return c - "A" + 10
1155     if c >= "a" and c <= "f"
1156         return c - "a" + 10
1157     return -1
1158
1159 '' This routine dumps a portion of the RAM/ROM to the display (pin == -1).
1160 '' If pin is not -1, it is an EEPROM address in the form required by the
1161 '' I2C routines in i2cSpiInit. The specified address is or'd with this.
1162 '' The format is 8 bytes wide with hexadecimal and ASCII.
1163
1164 PUB dumpMemory(pin,addr,size) | i, c, p, first, buf0, buf1, buf2
1165     addr &= $7FFFF
1166     first := true
1167     p := addr & $7FFF8
1168     repeat while p < (addr + size)
1169         if first
1170             ser.hex(addr,5)
1171             first := false
1172         else
1173             ser.hex(p,5)
```

```

1174   ser.tx(" ")
1175   repeat i from 0 to 7
1176     byte[@buf0][i] := " "
1177     if p => addr and p < (addr + size)
1178       if pin <> -1
1179         c := i2c.readByte(i2c#bootPin,i2c#EEPROM,p)
1180         if c < 0
1181           abort string("EEPROM read")
1182         else
1183           c := byte[p]
1184           ser.hex(c,2)
1185           if c => " " and c <= "~"
1186             byte[@buf0][i] := c
1187         else
1188           ser.tx(" ")
1189           ser.tx(" ")
1190           ser.tx(" ")
1191           p++
1192   buf2 := 0
1193   ser.tx("|")
1194   ser.str(@buf0)
1195   ser.tx("|")
1196   ser.str(@crlf)
1197
1198 DAT
1199 '' Lookup table for Speeds 0 to 63
1200
1201 'Note: A formula can be applied that will produce values that are within 6.27%
1202 '
1203 '   For ramp speeds between 7 and 63 ...
1204 '
1205 '   RampSpeed = ((Speed - 6)* 70231)/1600
1206 '
1207 '   For speeds between 1 and 6, a polynomial fit could be used to produce
1208 '   values that are within 3.1%, but with as few numbers that are used you
1209 '   don't really gain much by not using a lookup table...
1210 '
1211 '   RampSpeed = (158 * Speed^3 - 742 * Speed^2 + 2888 * Speed + 9264)/1600
1212 '
1213 'Note:
1214 '   RampSpeed - is equal to the delay from one Servo Increment to the next. The resolution
1215 '   of the delay is 20ms, so a value of 50 = 1 second ; a value of 3000 = 1 minute
1216
1217 RampSpeed      word      1,7,8,10,12,16,22,43,83,123,184,222,261,299,339,379,426,473,520,567,615
1218                word      662,709,756,808,859,910,961,1012,1064,1115,1166,1207,1248,1288,1329,1370
1219                word      1411,1451,1492,1533,1574,1614,1655,1696,1737,1778,1818,1859,1900,1941,1981
1220                word      2022,2063,2104,2144,2185,2226,2267,2307,2348,2389,2430,2470
1221
1222 {{
1223     TERMS OF USE: MIT License
1224
1225     Permission is hereby granted, free of charge, to any person obtaining a copy
1226     of this software and associated documentation files (the "Software"), to deal
1227     in the Software without restriction, including without limitation the rights
1228     to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
1229     copies of the Software, and to permit persons to whom the Software is
1230     furnished to do so, subject to the following conditions:
1231
1232     The above copyright notice and this permission notice shall be included in
1233     all copies or substantial portions of the Software.
1234
1235     THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
1236     IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
1237     FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
1238     AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
1239     LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
1240     OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
1241     THE SOFTWARE.
1242 }}

```