; Enhanced BASIC to assemble under 6502 simulator, $ver 2.22

; $E7E1 $E7CF $E7C6 $E7D3 $E7D1 $E7D5 $E7CF $E81E $E825

; 2.00 new revision numbers start here

; 2.01 fixed LCASE$() and UCASE$()

; 2.02 new get value routine done

; 2.03 changed RND() to galoise method

; 2.04 fixed SPC()

; 2.05 new get value routine fixed

; 2.06 changed USR() code

; 2.07 fixed STR$()

; 2.08 changed INPUT and READ to remove need for $00 start to input buffer

; 2.09 fixed RND()

; 2.10 integrated missed changes from an earlier version

; 2.20 added ELSE to IF .. THEN and fixed IF .. GOTO <statement> to cause error

; 2.21 fixed IF .. THEN RETURN to not cause error

; 2.22 fixed RND() breaking the get byte routine

; zero page use ..

LAB\_WARM = $00 ; BASIC warm start entry point

Wrmjpl = LAB\_WARM+1 ; BASIC warm start vector jump low byte

Wrmjph = LAB\_WARM+2 ; BASIC warm start vector jump high byte

Usrjmp = $0A ; USR function JMP address

Usrjpl = Usrjmp+1 ; USR function JMP vector low byte

Usrjph = Usrjmp+2 ; USR function JMP vector high byte

Nullct = $0D ; nulls output after each line

TPos = $0E ; BASIC terminal position byte

TWidth = $0F ; BASIC terminal width byte

Iclim = $10 ; input column limit

Itempl = $11 ; temporary integer low byte

Itemph = Itempl+1 ; temporary integer high byte

nums\_1 = Itempl ; number to bin/hex string convert MSB

nums\_2 = nums\_1+1 ; number to bin/hex string convert

nums\_3 = nums\_1+2 ; number to bin/hex string convert LSB

Srchc = $5B ; search character

Temp3 = Srchc ; temp byte used in number routines

Scnquo = $5C ; scan-between-quotes flag

Asrch = Scnquo ; alt search character

XOAw\_l = Srchc ; eXclusive OR, OR and AND word low byte

XOAw\_h = Scnquo ; eXclusive OR, OR and AND word high byte

Ibptr = $5D ; input buffer pointer

Dimcnt = Ibptr ; # of dimensions

Tindx = Ibptr ; token index

Defdim = $5E ; default DIM flag

Dtypef = $5F ; data type flag, $FF=string, $00=numeric

Oquote = $60 ; open quote flag (b7) (Flag: DATA scan; LIST quote; memory)

Gclctd = $60 ; garbage collected flag

Sufnxf = $61 ; subscript/FNX flag, 1xxx xxx = FN(0xxx xxx)

Imode = $62 ; input mode flag, $00=INPUT, $80=READ

Cflag = $63 ; comparison evaluation flag

TabSiz = $64 ; TAB step size (was input flag)

next\_s = $65 ; next descriptor stack address

; these two bytes form a word pointer to the item

; currently on top of the descriptor stack

last\_sl = $66 ; last descriptor stack address low byte

last\_sh = $67 ; last descriptor stack address high byte (always $00)

des\_sk = $68 ; descriptor stack start address (temp strings)

; = $70 ; End of descriptor stack

ut1\_pl = $71 ; utility pointer 1 low byte

ut1\_ph = ut1\_pl+1 ; utility pointer 1 high byte

ut2\_pl = $73 ; utility pointer 2 low byte

ut2\_ph = ut2\_pl+1 ; utility pointer 2 high byte

Temp\_2 = ut1\_pl ; temp byte for block move

FACt\_1 = $75 ; FAC temp mantissa1

FACt\_2 = FACt\_1+1 ; FAC temp mantissa2

FACt\_3 = FACt\_2+1 ; FAC temp mantissa3

dims\_l = FACt\_2 ; array dimension size low byte

dims\_h = FACt\_3 ; array dimension size high byte

TempB = $78 ; temp page 0 byte

Smeml = $79 ; start of mem low byte (Start-of-Basic)

Smemh = Smeml+1 ; start of mem high byte (Start-of-Basic)

Svarl = $7B ; start of vars low byte (Start-of-Variables)

Svarh = Svarl+1 ; start of vars high byte (Start-of-Variables)

Sarryl = $7D ; var mem end low byte (Start-of-Arrays)

Sarryh = Sarryl+1 ; var mem end high byte (Start-of-Arrays)

Earryl = $7F ; array mem end low byte (End-of-Arrays)

Earryh = Earryl+1 ; array mem end high byte (End-of-Arrays)

Sstorl = $81 ; string storage low byte (String storage (moving down))

Sstorh = Sstorl+1 ; string storage high byte (String storage (moving down))

Sutill = $83 ; string utility ptr low byte

Sutilh = Sutill+1 ; string utility ptr high byte

Ememl = $85 ; end of mem low byte (Limit-of-memory)

Ememh = Ememl+1 ; end of mem high byte (Limit-of-memory)

Clinel = $87 ; current line low byte (Basic line number)

Clineh = Clinel+1 ; current line high byte (Basic line number)

Blinel = $89 ; break line low byte (Previous Basic line number)

Blineh = Blinel+1 ; break line high byte (Previous Basic line number)

Cpntrl = $8B ; continue pointer low byte

Cpntrh = Cpntrl+1 ; continue pointer high byte

Dlinel = $8D ; current DATA line low byte

Dlineh = Dlinel+1 ; current DATA line high byte

Dptrl = $8F ; DATA pointer low byte

Dptrh = Dptrl+1 ; DATA pointer high byte

Rdptrl = $91 ; read pointer low byte

Rdptrh = Rdptrl+1 ; read pointer high byte

Varnm1 = $93 ; current var name 1st byte

Varnm2 = Varnm1+1 ; current var name 2nd byte

Cvaral = $95 ; current var address low byte

Cvarah = Cvaral+1 ; current var address high byte

Frnxtl = $97 ; var pointer for FOR/NEXT low byte

Frnxth = Frnxtl+1 ; var pointer for FOR/NEXT high byte

Tidx1 = Frnxtl ; temp line index

Lvarpl = Frnxtl ; let var pointer low byte

Lvarph = Frnxth ; let var pointer high byte

prstk = $99 ; precedence stacked flag

comp\_f = $9B ; compare function flag, bits 0,1 and 2 used

; bit 2 set if >

; bit 1 set if =

; bit 0 set if <

func\_l = $9C ; function pointer low byte

func\_h = func\_l+1 ; function pointer high byte

garb\_l = func\_l ; garbage collection working pointer low byte

garb\_h = func\_h ; garbage collection working pointer high byte

des\_2l = $9E ; string descriptor\_2 pointer low byte

des\_2h = des\_2l+1 ; string descriptor\_2 pointer high byte

g\_step = $A0 ; garbage collect step size

Fnxjmp = $A1 ; jump vector for functions

Fnxjpl = Fnxjmp+1 ; functions jump vector low byte

Fnxjph = Fnxjmp+2 ; functions jump vector high byte

g\_indx = Fnxjpl ; garbage collect temp index

FAC2\_r = $A3 ; FAC2 rounding byte

Adatal = $A4 ; array data pointer low byte

Adatah = Adatal+1 ; array data pointer high byte

Nbendl = Adatal ; new block end pointer low byte

Nbendh = Adatah ; new block end pointer high byte

Obendl = $A6 ; old block end pointer low byte

Obendh = Obendl+1 ; old block end pointer high byte

numexp = $A8 ; string to float number exponent count

expcnt = $A9 ; string to float exponent count

numbit = numexp ; bit count for array element calculations

numdpf = $AA ; string to float decimal point flag

expneg = $AB ; string to float eval exponent -ve flag

Astrtl = numdpf ; array start pointer low byte

Astrth = expneg ; array start pointer high byte

Histrl = numdpf ; highest string low byte

Histrh = expneg ; highest string high byte

Baslnl = numdpf ; BASIC search line pointer low byte

Baslnh = expneg ; BASIC search line pointer high byte

Fvar\_l = numdpf ; find/found variable pointer low byte

Fvar\_h = expneg ; find/found variable pointer high byte

Ostrtl = numdpf ; old block start pointer low byte

Ostrth = expneg ; old block start pointer high byte

Vrschl = numdpf ; variable search pointer low byte

Vrschh = expneg ; variable search pointer high byte

FAC1\_e = $AC ; FAC1 exponent

FAC1\_1 = FAC1\_e+1 ; FAC1 mantissa1

FAC1\_2 = FAC1\_e+2 ; FAC1 mantissa2

FAC1\_3 = FAC1\_e+3 ; FAC1 mantissa3

FAC1\_s = FAC1\_e+4 ; FAC1 sign (b7)

str\_ln = FAC1\_e ; string length

str\_pl = FAC1\_1 ; string pointer low byte

str\_ph = FAC1\_2 ; string pointer high byte

des\_pl = FAC1\_2 ; string descriptor pointer low byte

des\_ph = FAC1\_3 ; string descriptor pointer high byte

mids\_l = FAC1\_3 ; MID$ string temp length byte

negnum = $B1 ; string to float eval -ve flag

numcon = $B1 ; series evaluation constant count

FAC1\_o = $B2 ; FAC1 overflow byte

FAC2\_e = $B3 ; FAC2 exponent

FAC2\_1 = FAC2\_e+1 ; FAC2 mantissa1

FAC2\_2 = FAC2\_e+2 ; FAC2 mantissa2

FAC2\_3 = FAC2\_e+3 ; FAC2 mantissa3

FAC2\_s = FAC2\_e+4 ; FAC2 sign (b7)

FAC\_sc = $B8 ; FAC sign comparison, Acc#1 vs #2

FAC1\_r = $B9 ; FAC1 rounding byte

ssptr\_l = FAC\_sc ; string start pointer low byte

ssptr\_h = FAC1\_r ; string start pointer high byte

sdescr = FAC\_sc ; string descriptor pointer

csidx = $BA ; line crunch save index

Asptl = csidx ; array size/pointer low byte

Aspth = $BB ; array size/pointer high byte

Btmpl = Asptl ; BASIC pointer temp low byte

Btmph = Aspth ; BASIC pointer temp low byte

Cptrl = Asptl ; BASIC pointer temp low byte

Cptrh = Aspth ; BASIC pointer temp low byte

Sendl = Asptl ; BASIC pointer temp low byte

Sendh = Aspth ; BASIC pointer temp low byte

LAB\_IGBY = $BC ; get next BASIC byte subroutine

LAB\_GBYT = $C2 ; get current BASIC byte subroutine

Bpntrl = $C3 ; BASIC execute (get byte) pointer low byte

Bpntrh = Bpntrl+1 ; BASIC execute (get byte) pointer high byte

; = $D7 ; end of get BASIC char subroutine

Rbyte4 = $D8 ; extra PRNG byte

Rbyte1 = Rbyte4+1 ; most significant PRNG byte

Rbyte2 = Rbyte4+2 ; middle PRNG byte

Rbyte3 = Rbyte4+3 ; least significant PRNG byte

NmiBase = $DC ; NMI handler enabled/setup/triggered flags

; bit function

; === ========

; 7 interrupt enabled

; 6 interrupt setup

; 5 interrupt happened

; = $DD ; NMI handler addr low byte

; = $DE ; NMI handler addr high byte

IrqBase = $DF ; IRQ handler enabled/setup/triggered flags

; = $E0 ; IRQ handler addr low byte

; = $E1 ; IRQ handler addr high byte

; = $DE ; unused

; = $DF ; unused

; = $E0 ; unused

; = $E1 ; unused

; = $E2 ; unused

; = $E3 ; unused

; = $E4 ; unused

; = $E5 ; unused

; = $E6 ; unused

; = $E7 ; unused

; = $E8 ; unused

; = $E9 ; unused

; = $EA ; unused

; = $EB ; unused

; = $EC ; unused

; = $ED ; unused

; = $EE ; unused

Decss = $EF ; number to decimal string start

Decssp1 = Decss+1 ; number to decimal string start

; = $FF ; decimal string end

; token values needed for BASIC

; primary command tokens (can start a statement)

TK\_END = $80 ; END token

TK\_FOR = TK\_END+1 ; FOR token

TK\_NEXT = TK\_FOR+1 ; NEXT token

TK\_DATA = TK\_NEXT+1 ; DATA token

TK\_INPUT = TK\_DATA+1 ; INPUT token

TK\_DIM = TK\_INPUT+1 ; DIM token

TK\_READ = TK\_DIM+1 ; READ token

TK\_LET = TK\_READ+1 ; LET token

TK\_DEC = TK\_LET+1 ; DEC token

TK\_GOTO = TK\_DEC+1 ; GOTO token

TK\_RUN = TK\_GOTO+1 ; RUN token

TK\_IF = TK\_RUN+1 ; IF token

TK\_RESTORE = TK\_IF+1 ; RESTORE token

TK\_GOSUB = TK\_RESTORE+1 ; GOSUB token

TK\_RETIRQ = TK\_GOSUB+1 ; RETIRQ token

TK\_RETNMI = TK\_RETIRQ+1 ; RETNMI token

TK\_RETURN = TK\_RETNMI+1 ; RETURN token

TK\_REM = TK\_RETURN+1 ; REM token

TK\_STOP = TK\_REM+1 ; STOP token

TK\_ON = TK\_STOP+1 ; ON token

TK\_NULL = TK\_ON+1 ; NULL token

TK\_INC = TK\_NULL+1 ; INC token

TK\_WAIT = TK\_INC+1 ; WAIT token

TK\_LOAD = TK\_WAIT+1 ; LOAD token

TK\_SAVE = TK\_LOAD+1 ; SAVE token

TK\_DEF = TK\_SAVE+1 ; DEF token

TK\_POKE = TK\_DEF+1 ; POKE token

TK\_DOKE = TK\_POKE+1 ; DOKE token

TK\_CALL = TK\_DOKE+1 ; CALL token

TK\_DO = TK\_CALL+1 ; DO token

TK\_LOOP = TK\_DO+1 ; LOOP token

TK\_PRINT = TK\_LOOP+1 ; PRINT token

TK\_CONT = TK\_PRINT+1 ; CONT token

TK\_LIST = TK\_CONT+1 ; LIST token

TK\_CLEAR = TK\_LIST+1 ; CLEAR token

TK\_NEW = TK\_CLEAR+1 ; NEW token

TK\_WIDTH = TK\_NEW+1 ; WIDTH token

TK\_GET = TK\_WIDTH+1 ; GET token

TK\_SWAP = TK\_GET+1 ; SWAP token

TK\_BITSET = TK\_SWAP+1 ; BITSET token

TK\_BITCLR = TK\_BITSET+1 ; BITCLR token

TK\_IRQ = TK\_BITCLR+1 ; IRQ token

TK\_NMI = TK\_IRQ+1 ; NMI token

; secondary command tokens, can't start a statement

TK\_TAB = TK\_NMI+1 ; TAB token

TK\_ELSE = TK\_TAB+1 ; ELSE token

TK\_TO = TK\_ELSE+1 ; TO token

TK\_FN = TK\_TO+1 ; FN token

TK\_SPC = TK\_FN+1 ; SPC token

TK\_THEN = TK\_SPC+1 ; THEN token

TK\_NOT = TK\_THEN+1 ; NOT token

TK\_STEP = TK\_NOT+1 ; STEP token

TK\_UNTIL = TK\_STEP+1 ; UNTIL token

TK\_WHILE = TK\_UNTIL+1 ; WHILE token

TK\_OFF = TK\_WHILE+1 ; OFF token

; operator tokens

TK\_PLUS = TK\_OFF+1 ; + token

TK\_MINUS = TK\_PLUS+1 ; - token

TK\_MUL = TK\_MINUS+1 ; \* token

TK\_DIV = TK\_MUL+1 ; / token

TK\_POWER = TK\_DIV+1 ; ^ token

TK\_AND = TK\_POWER+1 ; AND token

TK\_EOR = TK\_AND+1 ; EOR token

TK\_OR = TK\_EOR+1 ; OR token

TK\_RSHIFT = TK\_OR+1 ; RSHIFT token

TK\_LSHIFT = TK\_RSHIFT+1 ; LSHIFT token

TK\_GT = TK\_LSHIFT+1 ; > token

TK\_EQUAL = TK\_GT+1 ; = token

TK\_LT = TK\_EQUAL+1 ; < token

; functions tokens

TK\_SGN = TK\_LT+1 ; SGN token

TK\_INT = TK\_SGN+1 ; INT token

TK\_ABS = TK\_INT+1 ; ABS token

TK\_USR = TK\_ABS+1 ; USR token

TK\_FRE = TK\_USR+1 ; FRE token

TK\_POS = TK\_FRE+1 ; POS token

TK\_SQR = TK\_POS+1 ; SQR token

TK\_RND = TK\_SQR+1 ; RND token

TK\_LOG = TK\_RND+1 ; LOG token

TK\_EXP = TK\_LOG+1 ; EXP token

TK\_COS = TK\_EXP+1 ; COS token

TK\_SIN = TK\_COS+1 ; SIN token

TK\_TAN = TK\_SIN+1 ; TAN token

TK\_ATN = TK\_TAN+1 ; ATN token

TK\_PEEK = TK\_ATN+1 ; PEEK token

TK\_DEEK = TK\_PEEK+1 ; DEEK token

TK\_SADD = TK\_DEEK+1 ; SADD token

TK\_LEN = TK\_SADD+1 ; LEN token

TK\_STRS = TK\_LEN+1 ; STR$ token

TK\_VAL = TK\_STRS+1 ; VAL token

TK\_ASC = TK\_VAL+1 ; ASC token

TK\_UCASES = TK\_ASC+1 ; UCASE$ token

TK\_LCASES = TK\_UCASES+1 ; LCASE$ token

TK\_CHRS = TK\_LCASES+1 ; CHR$ token

TK\_HEXS = TK\_CHRS+1 ; HEX$ token

TK\_BINS = TK\_HEXS+1 ; BIN$ token

TK\_BITTST = TK\_BINS+1 ; BITTST token

TK\_MAX = TK\_BITTST+1 ; MAX token

TK\_MIN = TK\_MAX+1 ; MIN token

TK\_PI = TK\_MIN+1 ; PI token

TK\_TWOPI = TK\_PI+1 ; TWOPI token

TK\_VPTR = TK\_TWOPI+1 ; VARPTR token

TK\_LEFTS = TK\_VPTR+1 ; LEFT$ token

TK\_RIGHTS = TK\_LEFTS+1 ; RIGHT$ token

TK\_MIDS = TK\_RIGHTS+1 ; MID$ token

; offsets from a base of X or Y

PLUS\_0 = $00 ; X or Y plus 0

PLUS\_1 = $01 ; X or Y plus 1

PLUS\_2 = $02 ; X or Y plus 2

PLUS\_3 = $03 ; X or Y plus 3

LAB\_STAK = $0100 ; stack bottom, no offset

LAB\_SKFE = LAB\_STAK+$FE

; flushed stack address

LAB\_SKFF = LAB\_STAK+$FF

; flushed stack address

ccflag = $0300 ; BASIC CTRL-C flag, 00 = enabled, 01 = dis

ccbyte = ccflag+1 ; BASIC CTRL-C byte

ccnull = ccbyte+1 ; BASIC CTRL-C byte timeout

VEC\_CC = ccnull+1 ; ctrl c check vector

VEC\_IN = VEC\_CC+2 ; input vector

VEC\_OUT = VEC\_IN+2 ; output vector

VEC\_LD = VEC\_OUT+2 ; load vector

VEC\_SV = VEC\_LD+2 ; save vector

; Ibuffs can now be anywhere in RAM, ensure that the max length is < $80

Ibuffs = IRQ\_vec+$14

; start of input buffer after IRQ/NMI code

Ibuffe = Ibuffs+$47 ; end of input buffer

Ram\_base = $0400 ; start of user RAM (set as needed, should be page aligned)

Ram\_top = $5000 ; end of user RAM+1 (set as needed, should be page aligned)

; This start can be changed to suit your system

.org $5000

; For convenience, put jump here to reset location so it can be

; run from the load address.

JMP RES\_vec

; BASIC cold start entry point

; new page 2 initialization, copy block to ccflag on

**LAB\_COLD**

LDY #PG2\_TABE-PG2\_TABS-1

; byte count-1

**LAB\_2D13**

LDA PG2\_TABS,Y ; get byte

STA ccflag,Y ; store in page 2

DEY ; decrement count

BPL LAB\_2D13 ; loop if not done

LDX #$FF ; set byte

STX Clineh ; set current line high byte (set immediate mode)

TXS ; reset stack pointer

LDA #$4C ; code for JMP

STA Fnxjmp ; save for jump vector for functions

; copy block from LAB\_2CEE to $00BC - $00D3

LDX #StrTab-LAB\_2CEE ; set byte count

**LAB\_2D4E**

LDA LAB\_2CEE-1,X ; get byte from table

STA LAB\_IGBY-1,X ; save byte in page zero

DEX ; decrement count

BNE LAB\_2D4E ; loop if not all done

; copy block from StrTab to $0000 - $0012

**LAB\_GMEM**

LDX #EndTab-StrTab-1 ; set byte count-1

**TabLoop**

LDA StrTab,X ; get byte from table

STA PLUS\_0,X ; save byte in page zero

DEX ; decrement count

BPL TabLoop ; loop if not all done

; set-up start values

LDA #$00 ; clear A

STA NmiBase ; clear NMI handler enabled flag

STA IrqBase ; clear IRQ handler enabled flag

STA FAC1\_o ; clear FAC1 overflow byte

STA last\_sh ; clear descriptor stack top item pointer high byte

LDA #$0E ; set default tab size

STA TabSiz ; save it

LDA #$03 ; set garbage collect step size for descriptor stack

STA g\_step ; save it

LDX #des\_sk ; descriptor stack start

STX next\_s ; set descriptor stack pointer

JSR LAB\_CRLF ; print CR/LF

LDA #<LAB\_MSZM ; point to memory size message (low addr)

LDY #>LAB\_MSZM ; point to memory size message (high addr)

JSR LAB\_18C3 ; print null terminated string from memory

JSR LAB\_INLN ; print "? " and get BASIC input

STX Bpntrl ; set BASIC execute pointer low byte

STY Bpntrh ; set BASIC execute pointer high byte

JSR LAB\_GBYT ; get last byte back

BNE LAB\_2DAA ; branch if not null (user typed something)

LDY #$00 ; else clear Y

; character was null so get memory size the hard way

; we get here with Y=0 and Itempl/h = Ram\_base

**LAB\_2D93**

INC Itempl ; increment temporary integer low byte

BNE LAB\_2D99 ; branch if no overflow

INC Itemph ; increment temporary integer high byte

LDA Itemph ; get high byte

CMP #>Ram\_top ; compare with top of RAM+1

BEQ LAB\_2DB6 ; branch if match (end of user RAM)

**LAB\_2D99**

LDA #$55 ; set test byte

STA (Itempl),Y ; save via temporary integer

CMP (Itempl),Y ; compare via temporary integer

BNE LAB\_2DB6 ; branch if fail

ASL ; shift test byte left (now $AA)

STA (Itempl),Y ; save via temporary integer

CMP (Itempl),Y ; compare via temporary integer

BEQ LAB\_2D93 ; if ok go do next byte

BNE LAB\_2DB6 ; branch if fail

**LAB\_2DAA**

JSR LAB\_2887 ; get FAC1 from string

LDA FAC1\_e ; get FAC1 exponent

CMP #$98 ; compare with exponent = 2^24

BCS LAB\_GMEM ; if too large go try again

JSR LAB\_F2FU ; save integer part of FAC1 in temporary integer

; (no range check)

**LAB\_2DB6**

LDA Itempl ; get temporary integer low byte

LDY Itemph ; get temporary integer high byte

CPY #<Ram\_base+1 ; compare with start of RAM+$100 high byte

BCC LAB\_GMEM ; if too small go try again

; uncomment these lines if you want to check on the high limit of memory. Note if

; Ram\_top is set too low then this will fail. default is ignore it and assume the

; users know what they're doing!

; CPY #>Ram\_top ; compare with top of RAM high byte

; BCC MEM\_OK ; branch if < RAM top

; BNE LAB\_GMEM ; if too large go try again

; else was = so compare low bytes

; CMP #<Ram\_top ; compare with top of RAM low byte

; BEQ MEM\_OK ; branch if = RAM top

; BCS LAB\_GMEM ; if too large go try again

;**MEM\_OK**

STA Ememl ; set end of mem low byte

STY Ememh ; set end of mem high byte

STA Sstorl ; set bottom of string space low byte

STY Sstorh ; set bottom of string space high byte

LDY #<Ram\_base ; set start addr low byte

LDX #>Ram\_base ; set start addr high byte

STY Smeml ; save start of mem low byte

STX Smemh ; save start of mem high byte

; this line is only needed if Ram\_base is not $xx00

; LDY #$00 ; clear Y

TYA ; clear A

STA (Smeml),Y ; clear first byte

INC Smeml ; increment start of mem low byte

; these two lines are only needed if Ram\_base is $xxFF

; BNE LAB\_2E05 ; branch if no rollover

; INC Smemh ; increment start of mem high byte

**LAB\_2E05**

JSR LAB\_CRLF ; print CR/LF

JSR LAB\_1463 ; do "NEW" and "CLEAR"

LDA Ememl ; get end of mem low byte

SEC ; set carry for subtract

SBC Smeml ; subtract start of mem low byte

TAX ; copy to X

LDA Ememh ; get end of mem high byte

SBC Smemh ; subtract start of mem high byte

JSR LAB\_295E ; print XA as unsigned integer (bytes free)

LDA #<LAB\_SMSG ; point to sign-on message (low addr)

LDY #>LAB\_SMSG ; point to sign-on message (high addr)

JSR LAB\_18C3 ; print null terminated string from memory

LDA #<LAB\_1274 ; warm start vector low byte

LDY #>LAB\_1274 ; warm start vector high byte

STA Wrmjpl ; save warm start vector low byte

STY Wrmjph ; save warm start vector high byte

JMP (Wrmjpl) ; go do warm start

; open up space in memory

; move (Ostrtl)-(Obendl) to new block ending at (Nbendl)

; Nbendl,Nbendh - new block end address (A/Y)

; Obendl,Obendh - old block end address

; Ostrtl,Ostrth - old block start address

; returns with ..

; Nbendl,Nbendh - new block start address (high byte - $100)

; Obendl,Obendh - old block start address (high byte - $100)

; Ostrtl,Ostrth - old block start address (unchanged)

**LAB\_11CF**

JSR LAB\_121F ; check available memory, "Out of memory" error if no room

; addr to check is in AY (low/high)

STA Earryl ; save new array mem end low byte

STY Earryh ; save new array mem end high byte

; open up space in memory

; move (Ostrtl)-(Obendl) to new block ending at (Nbendl)

; don't set array end

**LAB\_11D6**

SEC ; set carry for subtract

LDA Obendl ; get block end low byte

SBC Ostrtl ; subtract block start low byte

TAY ; copy MOD(block length/$100) byte to Y

LDA Obendh ; get block end high byte

SBC Ostrth ; subtract block start high byte

TAX ; copy block length high byte to X

INX ; +1 to allow for count=0 exit

TYA ; copy block length low byte to A

BEQ LAB\_120A ; branch if length low byte=0

; block is (X-1)\*256+Y bytes, do the Y bytes first

SEC ; set carry for add + 1, two's complement

EOR #$FF ; invert low byte for subtract

ADC Obendl ; add block end low byte

STA Obendl ; save corrected old block end low byte

BCS LAB\_11F3 ; branch if no underflow

DEC Obendh ; else decrement block end high byte

SEC ; set carry for add + 1, two's complement

**LAB\_11F3**

TYA ; get MOD(block length/$100) byte

EOR #$FF ; invert low byte for subtract

ADC Nbendl ; add destination end low byte

STA Nbendl ; save modified new block end low byte

BCS LAB\_1203 ; branch if no underflow

DEC Nbendh ; else decrement block end high byte

BCC LAB\_1203 ; branch always

**LAB\_11FF**

LDA (Obendl),Y ; get byte from source

STA (Nbendl),Y ; copy byte to destination

**LAB\_1203**

DEY ; decrement index

BNE LAB\_11FF ; loop until Y=0

; now do Y=0 indexed byte

LDA (Obendl),Y ; get byte from source

STA (Nbendl),Y ; save byte to destination

**LAB\_120A**

DEC Obendh ; decrement source pointer high byte

DEC Nbendh ; decrement destination pointer high byte

DEX ; decrement block count

BNE LAB\_1203 ; loop until count = $0

RTS

; check room on stack for A bytes

; stack too deep? do OM error

**LAB\_1212**

STA TempB ; save result in temp byte

TSX ; copy stack

CPX TempB ; compare new "limit" with stack

BCC LAB\_OMER ; if stack < limit do "Out of memory" error then warm start

RTS

; check available memory, "Out of memory" error if no room

; addr to check is in AY (low/high)

**LAB\_121F**

CPY Sstorh ; compare bottom of string mem high byte

BCC LAB\_124B ; if less then exit (is ok)

BNE LAB\_1229 ; skip next test if greater (tested <)

; high byte was =, now do low byte

CMP Sstorl ; compare with bottom of string mem low byte

BCC LAB\_124B ; if less then exit (is ok)

; addr is > string storage ptr (oops!)

**LAB\_1229**

PHA ; push addr low byte

LDX #$08 ; set index to save Adatal to expneg inclusive

TYA ; copy addr high byte (to push on stack)

; save misc numeric work area

**LAB\_122D**

PHA ; push byte

LDA Adatal-1,X ; get byte from Adatal to expneg ( ,$00 not pushed)

DEX ; decrement index

BPL LAB\_122D ; loop until all done

JSR LAB\_GARB ; garbage collection routine

; restore misc numeric work area

LDX #$00 ; clear the index to restore bytes

**LAB\_1238**

PLA ; pop byte

STA Adatal,X ; save byte to Adatal to expneg

INX ; increment index

CPX #$08 ; compare with end + 1

BMI LAB\_1238 ; loop if more to do

PLA ; pop addr high byte

TAY ; copy back to Y

PLA ; pop addr low byte

CPY Sstorh ; compare bottom of string mem high byte

BCC LAB\_124B ; if less then exit (is ok)

BNE LAB\_OMER ; if greater do "Out of memory" error then warm start

; high byte was =, now do low byte

CMP Sstorl ; compare with bottom of string mem low byte

BCS LAB\_OMER ; if >= do "Out of memory" error then warm start

; ok exit, carry clear

**LAB\_124B**

RTS

; do "Out of memory" error then warm start

**LAB\_OMER**

LDX #$0C ; error code $0C ("Out of memory" error)

; do error #X, then warm start

**LAB\_XERR**

JSR LAB\_CRLF ; print CR/LF

LDA LAB\_BAER,X ; get error message pointer low byte

LDY LAB\_BAER+1,X ; get error message pointer high byte

JSR LAB\_18C3 ; print null terminated string from memory

JSR LAB\_1491 ; flush stack and clear continue flag

LDA #<LAB\_EMSG ; point to " Error" low addr

LDY #>LAB\_EMSG ; point to " Error" high addr

**LAB\_1269**

JSR LAB\_18C3 ; print null terminated string from memory

LDY Clineh ; get current line high byte

INY ; increment it

BEQ LAB\_1274 ; go do warm start (was immediate mode)

; else print line number

JSR LAB\_2953 ; print " in line [LINE #]"

; BASIC warm start entry point

; wait for Basic command

**LAB\_1274**

; clear ON IRQ/NMI bytes

LDA #$00 ; clear A

STA IrqBase ; clear enabled byte

STA NmiBase ; clear enabled byte

LDA #<LAB\_RMSG ; point to "Ready" message low byte

LDY #>LAB\_RMSG ; point to "Ready" message high byte

JSR LAB\_18C3 ; go do print string

; wait for Basic command (no "Ready")

**LAB\_127D**

JSR LAB\_1357 ; call for BASIC input

**LAB\_1280**

STX Bpntrl ; set BASIC execute pointer low byte

STY Bpntrh ; set BASIC execute pointer high byte

JSR LAB\_GBYT ; scan memory

BEQ LAB\_127D ; loop while null

; got to interpret input line now ..

LDX #$FF ; current line to null value

STX Clineh ; set current line high byte

BCC LAB\_1295 ; branch if numeric character (handle new BASIC line)

; no line number .. immediate mode

JSR LAB\_13A6 ; crunch keywords into Basic tokens

JMP LAB\_15F6 ; go scan and interpret code

; handle new BASIC line

**LAB\_1295**

JSR LAB\_GFPN ; get fixed-point number into temp integer

JSR LAB\_13A6 ; crunch keywords into Basic tokens

STY Ibptr ; save index pointer to end of crunched line

JSR LAB\_SSLN ; search BASIC for temp integer line number

BCC LAB\_12E6 ; branch if not found

; aroooogah! line # already exists! delete it

LDY #$01 ; set index to next line pointer high byte

LDA (Baslnl),Y ; get next line pointer high byte

STA ut1\_ph ; save it

LDA Svarl ; get start of vars low byte

STA ut1\_pl ; save it

LDA Baslnh ; get found line pointer high byte

STA ut2\_ph ; save it

LDA Baslnl ; get found line pointer low byte

DEY ; decrement index

SBC (Baslnl),Y ; subtract next line pointer low byte

CLC ; clear carry for add

ADC Svarl ; add start of vars low byte

STA Svarl ; save new start of vars low byte

STA ut2\_pl ; save destination pointer low byte

LDA Svarh ; get start of vars high byte

ADC #$FF ; -1 + carry

STA Svarh ; save start of vars high byte

SBC Baslnh ; subtract found line pointer high byte

TAX ; copy to block count

SEC ; set carry for subtract

LDA Baslnl ; get found line pointer low byte

SBC Svarl ; subtract start of vars low byte

TAY ; copy to bytes in first block count

BCS LAB\_12D0 ; branch if overflow

INX ; increment block count (correct for =0 loop exit)

DEC ut2\_ph ; decrement destination high byte

**LAB\_12D0**

CLC ; clear carry for add

ADC ut1\_pl ; add source pointer low byte

BCC LAB\_12D8 ; branch if no overflow

DEC ut1\_ph ; else decrement source pointer high byte

CLC ; clear carry

; close up memory to delete old line

**LAB\_12D8**

LDA (ut1\_pl),Y ; get byte from source

STA (ut2\_pl),Y ; copy to destination

INY ; increment index

BNE LAB\_12D8 ; while <> 0 do this block

INC ut1\_ph ; increment source pointer high byte

INC ut2\_ph ; increment destination pointer high byte

DEX ; decrement block count

BNE LAB\_12D8 ; loop until all done

; got new line in buffer and no existing same #

**LAB\_12E6**

LDA Ibuffs ; get byte from start of input buffer

BEQ LAB\_1319 ; if null line just go flush stack/vars and exit

; got new line and it isn't empty line

LDA Ememl ; get end of mem low byte

LDY Ememh ; get end of mem high byte

STA Sstorl ; set bottom of string space low byte

STY Sstorh ; set bottom of string space high byte

LDA Svarl ; get start of vars low byte (end of BASIC)

STA Obendl ; save old block end low byte

LDY Svarh ; get start of vars high byte (end of BASIC)

STY Obendh ; save old block end high byte

ADC Ibptr ; add input buffer pointer (also buffer length)

BCC LAB\_1301 ; branch if no overflow from add

INY ; else increment high byte

**LAB\_1301**

STA Nbendl ; save new block end low byte (move to, low byte)

STY Nbendh ; save new block end high byte

JSR LAB\_11CF ; open up space in memory

; old start pointer Ostrtl,Ostrth set by the find line call

LDA Earryl ; get array mem end low byte

LDY Earryh ; get array mem end high byte

STA Svarl ; save start of vars low byte

STY Svarh ; save start of vars high byte

LDY Ibptr ; get input buffer pointer (also buffer length)

DEY ; adjust for loop type

**LAB\_1311**

LDA Ibuffs-4,Y ; get byte from crunched line

STA (Baslnl),Y ; save it to program memory

DEY ; decrement count

CPY #$03 ; compare with first byte-1

BNE LAB\_1311 ; continue while count <> 3

LDA Itemph ; get line # high byte

STA (Baslnl),Y ; save it to program memory

DEY ; decrement count

LDA Itempl ; get line # low byte

STA (Baslnl),Y ; save it to program memory

DEY ; decrement count

LDA #$FF ; set byte to allow chain rebuild. if you didn't set this

; byte then a zero already here would stop the chain rebuild

; as it would think it was the [EOT] marker.

STA (Baslnl),Y ; save it to program memory

**LAB\_1319**

JSR LAB\_1477 ; reset execution to start, clear vars and flush stack

LDX Smeml ; get start of mem low byte

LDA Smemh ; get start of mem high byte

LDY #$01 ; index to high byte of next line pointer

**LAB\_1325**

STX ut1\_pl ; set line start pointer low byte

STA ut1\_ph ; set line start pointer high byte

LDA (ut1\_pl),Y ; get it

BEQ LAB\_133E ; exit if end of program

; rebuild chaining of Basic lines

LDY #$04 ; point to first code byte of line

; there is always 1 byte + [EOL] as null entries are deleted

**LAB\_1330**

INY ; next code byte

LDA (ut1\_pl),Y ; get byte

BNE LAB\_1330 ; loop if not [EOL]

SEC ; set carry for add + 1

TYA ; copy end index

ADC ut1\_pl ; add to line start pointer low byte

TAX ; copy to X

LDY #$00 ; clear index, point to this line's next line pointer

STA (ut1\_pl),Y ; set next line pointer low byte

TYA ; clear A

ADC ut1\_ph ; add line start pointer high byte + carry

INY ; increment index to high byte

STA (ut1\_pl),Y ; save next line pointer low byte

BCC LAB\_1325 ; go do next line, branch always, carry clear

**LAB\_133E**

JMP LAB\_127D ; else we just wait for Basic command, no "Ready"

; print "? " and get BASIC input

**LAB\_INLN**

JSR LAB\_18E3 ; print "?" character

JSR LAB\_18E0 ; print " "

BNE LAB\_1357 ; call for BASIC input and return

; receive line from keyboard

; $08 as delete key (BACKSPACE on standard keyboard)

**LAB\_134B**

JSR LAB\_PRNA ; go print the character

DEX ; decrement the buffer counter (delete)

.byte $2C ; make LDX into BIT abs

; call for BASIC input (main entry point)

**LAB\_1357**

LDX #$00 ; clear BASIC line buffer pointer

**LAB\_1359**

JSR V\_INPT ; call scan input device

BCC LAB\_1359 ; loop if no byte

BEQ LAB\_1359 ; loop until valid input (ignore NULLs)

CMP #$07 ; compare with [BELL]

BEQ LAB\_1378 ; branch if [BELL]

CMP #$0D ; compare with [CR]

BEQ LAB\_1384 ; do CR/LF exit if [CR]

CPX #$00 ; compare pointer with $00

BNE LAB\_1374 ; branch if not empty

; next two lines ignore any non print character and [SPACE] if input buffer empty

CMP #$21 ; compare with [SP]+1

BCC LAB\_1359 ; if < ignore character

**LAB\_1374**

CMP #$08 ; compare with [BACKSPACE] (delete last character)

BEQ LAB\_134B ; go delete last character

**LAB\_1378**

CPX #Ibuffe-Ibuffs ; compare character count with max

BCS LAB\_138E ; skip store and do [BELL] if buffer full

STA Ibuffs,X ; else store in buffer

INX ; increment pointer

**LAB\_137F**

JSR LAB\_PRNA ; go print the character

BNE LAB\_1359 ; always loop for next character

**LAB\_1384**

JMP LAB\_1866 ; do CR/LF exit to BASIC

; announce buffer full

**LAB\_138E**

LDA #$07 ; [BELL] character into A

BNE LAB\_137F ; go print the [BELL] but ignore input character

; branch always

; crunch keywords into Basic tokens

; position independent buffer version ..

; faster, dictionary search version ....

**LAB\_13A6**

LDY #$FF ; set save index (makes for easy math later)

SEC ; set carry for subtract

LDA Bpntrl ; get basic execute pointer low byte

SBC #<Ibuffs ; subtract input buffer start pointer

TAX ; copy result to X (index past line # if any)

STX Oquote ; clear open quote/DATA flag

**LAB\_13AC**

LDA Ibuffs,X ; get byte from input buffer

BEQ LAB\_13EC ; if null save byte then exit

CMP #'\_' ; compare with "\_"

BCS LAB\_13EC ; if >= go save byte then continue crunching

CMP #'<' ; compare with "<"

BCS LAB\_13CC ; if >= go crunch now

CMP #'0' ; compare with "0"

BCS LAB\_13EC ; if >= go save byte then continue crunching

STA Scnquo ; save buffer byte as search character

CMP #$22 ; is it quote character?

BEQ LAB\_1410 ; branch if so (copy quoted string)

CMP #'\*' ; compare with "\*"

BCC LAB\_13EC ; if < go save byte then continue crunching

; else crunch now

**LAB\_13CC**

BIT Oquote ; get open quote/DATA token flag

BVS LAB\_13EC ; branch if b6 of Oquote set (was DATA)

; go save byte then continue crunching

STX TempB ; save buffer read index

STY csidx ; copy buffer save index

LDY #<TAB\_1STC ; get keyword first character table low address

STY ut2\_pl ; save pointer low byte

LDY #>TAB\_1STC ; get keyword first character table high address

STY ut2\_ph ; save pointer high byte

LDY #$00 ; clear table pointer

**LAB\_13D0**

CMP (ut2\_pl),Y ; compare with keyword first character table byte

BEQ LAB\_13D1 ; go do word\_table\_chr if match

BCC LAB\_13EA ; if < keyword first character table byte go restore

; Y and save to crunched

INY ; else increment pointer

BNE LAB\_13D0 ; and loop (branch always)

; have matched first character of some keyword

**LAB\_13D1**

TYA ; copy matching index

ASL ; \*2 (bytes per pointer)

TAX ; copy to new index

LDA TAB\_CHRT,X ; get keyword table pointer low byte

STA ut2\_pl ; save pointer low byte

LDA TAB\_CHRT+1,X ; get keyword table pointer high byte

STA ut2\_ph ; save pointer high byte

LDY #$FF ; clear table pointer (make -1 for start)

LDX TempB ; restore buffer read index

**LAB\_13D6**

INY ; next table byte

LDA (ut2\_pl),Y ; get byte from table

**LAB\_13D8**

BMI LAB\_13EA ; all bytes matched so go save token

INX ; next buffer byte

CMP Ibuffs,X ; compare with byte from input buffer

BEQ LAB\_13D6 ; go compare next if match

BNE LAB\_1417 ; branch if >< (not found keyword)

**LAB\_13EA**

LDY csidx ; restore save index

; save crunched to output

**LAB\_13EC**

INX ; increment buffer index (to next input byte)

INY ; increment save index (to next output byte)

STA Ibuffs,Y ; save byte to output

CMP #$00 ; set the flags, set carry

BEQ LAB\_142A ; do exit if was null [EOL]

; A holds token or byte here

SBC #':' ; subtract ":" (carry set by CMP #00)

BEQ LAB\_13FF ; branch if it was ":" (is now $00)

; A now holds token-$3A

CMP #TK\_DATA-$3A ; compare with DATA token - $3A

BNE LAB\_1401 ; branch if not DATA

; token was : or DATA

**LAB\_13FF**

STA Oquote ; save token-$3A (clear for ":", TK\_DATA-$3A for DATA)

**LAB\_1401**

EOR #TK\_REM-$3A ; effectively subtract REM token offset

BNE LAB\_13AC ; If wasn't REM then go crunch rest of line

STA Asrch ; else was REM so set search for [EOL]

; loop for REM, "..." etc.

**LAB\_1408**

LDA Ibuffs,X ; get byte from input buffer

BEQ LAB\_13EC ; branch if null [EOL]

CMP Asrch ; compare with stored character

BEQ LAB\_13EC ; branch if match (end quote)

; entry for copy string in quotes, don't crunch

**LAB\_1410**

INY ; increment buffer save index

STA Ibuffs,Y ; save byte to output

INX ; increment buffer read index

BNE LAB\_1408 ; loop while <> 0 (should never be 0!)

; not found keyword this go

**LAB\_1417**

LDX TempB ; compare has failed, restore buffer index (start byte!)

; now find the end of this word in the table

**LAB\_141B**

LDA (ut2\_pl),Y ; get table byte

PHP ; save status

INY ; increment table index

PLP ; restore byte status

BPL LAB\_141B ; if not end of keyword go do next

LDA (ut2\_pl),Y ; get byte from keyword table

BNE LAB\_13D8 ; go test next word if not zero byte (end of table)

; reached end of table with no match

LDA Ibuffs,X ; restore byte from input buffer

BPL LAB\_13EA ; branch always (all bytes in buffer are $00-$7F)

; go save byte in output and continue crunching

; reached [EOL]

**LAB\_142A**

INY ; increment pointer

INY ; increment pointer (makes it next line pointer high byte)

STA Ibuffs,Y ; save [EOL] (marks [EOT] in immediate mode)

INY ; adjust for line copy

INY ; adjust for line copy

INY ; adjust for line copy

DEC Bpntrl ; allow for increment (change if buffer starts at $xxFF)

RTS

; search Basic for temp integer line number from start of mem

**LAB\_SSLN**

LDA Smeml ; get start of mem low byte

LDX Smemh ; get start of mem high byte

; search Basic for temp integer line number from AX

; returns carry set if found

; returns Baslnl/Baslnh pointer to found or next higher (not found) line

; old 541 new 507

**LAB\_SHLN**

LDY #$01 ; set index

STA Baslnl ; save low byte as current

STX Baslnh ; save high byte as current

LDA (Baslnl),Y ; get pointer high byte from addr

BEQ LAB\_145F ; pointer was zero so we're done, do 'not found' exit

LDY #$03 ; set index to line # high byte

LDA (Baslnl),Y ; get line # high byte

DEY ; decrement index (point to low byte)

CMP Itemph ; compare with temporary integer high byte

BNE LAB\_1455 ; if <> skip low byte check

LDA (Baslnl),Y ; get line # low byte

CMP Itempl ; compare with temporary integer low byte

**LAB\_1455**

BCS LAB\_145E ; else if temp < this line, exit (passed line#)

**LAB\_1456**

DEY ; decrement index to next line ptr high byte

LDA (Baslnl),Y ; get next line pointer high byte

TAX ; copy to X

DEY ; decrement index to next line ptr low byte

LDA (Baslnl),Y ; get next line pointer low byte

BCC LAB\_SHLN ; go search for line # in temp (Itempl/Itemph) from AX

; (carry always clear)

**LAB\_145E**

BEQ LAB\_1460 ; exit if temp = found line #, carry is set

**LAB\_145F**

CLC ; clear found flag

**LAB\_1460**

RTS

; perform NEW

**LAB\_NEW**

BNE LAB\_1460 ; exit if not end of statement (to do syntax error)

**LAB\_1463**

LDA #$00 ; clear A

TAY ; clear Y

STA (Smeml),Y ; clear first line, next line pointer, low byte

INY ; increment index

STA (Smeml),Y ; clear first line, next line pointer, high byte

CLC ; clear carry

LDA Smeml ; get start of mem low byte

ADC #$02 ; calculate end of BASIC low byte

STA Svarl ; save start of vars low byte

LDA Smemh ; get start of mem high byte

ADC #$00 ; add any carry

STA Svarh ; save start of vars high byte

; reset execution to start, clear vars and flush stack

**LAB\_1477**

CLC ; clear carry

LDA Smeml ; get start of mem low byte

ADC #$FF ; -1

STA Bpntrl ; save BASIC execute pointer low byte

LDA Smemh ; get start of mem high byte

ADC #$FF ; -1+carry

STA Bpntrh ; save BASIC execute pointer high byte

; "CLEAR" command gets here

**LAB\_147A**

LDA Ememl ; get end of mem low byte

LDY Ememh ; get end of mem high byte

STA Sstorl ; set bottom of string space low byte

STY Sstorh ; set bottom of string space high byte

LDA Svarl ; get start of vars low byte

LDY Svarh ; get start of vars high byte

STA Sarryl ; save var mem end low byte

STY Sarryh ; save var mem end high byte

STA Earryl ; save array mem end low byte

STY Earryh ; save array mem end high byte

JSR LAB\_161A ; perform RESTORE command

; flush stack and clear continue flag

**LAB\_1491**

LDX #des\_sk ; set descriptor stack pointer

STX next\_s ; save descriptor stack pointer

PLA ; pull return address low byte

TAX ; copy return address low byte

PLA ; pull return address high byte

STX LAB\_SKFE ; save to cleared stack

STA LAB\_SKFF ; save to cleared stack

LDX #$FD ; new stack pointer

TXS ; reset stack

LDA #$00 ; clear byte

STA Cpntrh ; clear continue pointer high byte

STA Sufnxf ; clear subscript/FNX flag

**LAB\_14A6**

RTS

; perform CLEAR

**LAB\_CLEAR**

BEQ LAB\_147A ; if no following token go do "CLEAR"

; else there was a following token (go do syntax error)

RTS

; perform LIST [n][-m]

; bigger, faster version (a \_lot\_ faster)

**LAB\_LIST**

BCC LAB\_14BD ; branch if next character numeric (LIST n..)

BEQ LAB\_14BD ; branch if next character [NULL] (LIST)

CMP #TK\_MINUS ; compare with token for -

BNE LAB\_14A6 ; exit if not - (LIST -m)

; LIST [[n][-m]]

; this bit sets the n , if present, as the start and end

**LAB\_14BD**

JSR LAB\_GFPN ; get fixed-point number into temp integer

JSR LAB\_SSLN ; search BASIC for temp integer line number

; (pointer in Baslnl/Baslnh)

JSR LAB\_GBYT ; scan memory

BEQ LAB\_14D4 ; branch if no more characters

; this bit checks the - is present

CMP #TK\_MINUS ; compare with token for -

BNE LAB\_1460 ; return if not "-" (will be Syntax error)

; LIST [n]-m

; the - was there so set m as the end value

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_GFPN ; get fixed-point number into temp integer

BNE LAB\_1460 ; exit if not ok

**LAB\_14D4**

LDA Itempl ; get temporary integer low byte

ORA Itemph ; OR temporary integer high byte

BNE LAB\_14E2 ; branch if start set

LDA #$FF ; set for -1

STA Itempl ; set temporary integer low byte

STA Itemph ; set temporary integer high byte

**LAB\_14E2**

LDY #$01 ; set index for line

STY Oquote ; clear open quote flag

JSR LAB\_CRLF ; print CR/LF

LDA (Baslnl),Y ; get next line pointer high byte

; pointer initially set by search at LAB\_14BD

BEQ LAB\_152B ; if null all done so exit

JSR LAB\_1629 ; do CRTL-C check vector

INY ; increment index for line

LDA (Baslnl),Y ; get line # low byte

TAX ; copy to X

INY ; increment index

LDA (Baslnl),Y ; get line # high byte

CMP Itemph ; compare with temporary integer high byte

BNE LAB\_14FF ; branch if no high byte match

CPX Itempl ; compare with temporary integer low byte

BEQ LAB\_1501 ; branch if = last line to do (< will pass next branch)

LAB\_14FF ; else ..

BCS LAB\_152B ; if greater all done so exit

LAB\_1501

STY Tidx1 ; save index for line

JSR LAB\_295E ; print XA as unsigned integer

LDA #$20 ; space is the next character

LAB\_1508

LDY Tidx1 ; get index for line

AND #$7F ; mask top out bit of character

LAB\_150C

JSR LAB\_PRNA ; go print the character

CMP #$22 ; was it "quote" character

BNE LAB\_1519 ; branch if not

; we are either entering or leaving a pair of quotes

LDA Oquote ; get open quote flag

EOR #$FF ; toggle it

STA Oquote ; save it back

LAB\_1519

INY ; increment index

LDA (Baslnl),Y ; get next byte

BNE LAB\_152E ; branch if not [EOL] (go print character)

TAY ; else clear index

LDA (Baslnl),Y ; get next line pointer low byte

TAX ; copy to X

INY ; increment index

LDA (Baslnl),Y ; get next line pointer high byte

STX Baslnl ; set pointer to line low byte

STA Baslnh ; set pointer to line high byte

BNE LAB\_14E2 ; go do next line if not [EOT]

; else ..

LAB\_152B

RTS

LAB\_152E

BPL LAB\_150C ; just go print it if not token byte

; else was token byte so uncrunch it (maybe)

BIT Oquote ; test the open quote flag

BMI LAB\_150C ; just go print character if open quote set

LDX #>LAB\_KEYT ; get table address high byte

ASL ; \*2

ASL ; \*4

BCC LAB\_152F ; branch if no carry

INX ; else increment high byte

CLC ; clear carry for add

LAB\_152F

ADC #<LAB\_KEYT ; add low byte

BCC LAB\_1530 ; branch if no carry

INX ; else increment high byte

LAB\_1530

STA ut2\_pl ; save table pointer low byte

STX ut2\_ph ; save table pointer high byte

STY Tidx1 ; save index for line

LDY #$00 ; clear index

LDA (ut2\_pl),Y ; get length

TAX ; copy length

INY ; increment index

LDA (ut2\_pl),Y ; get 1st character

DEX ; decrement length

BEQ LAB\_1508 ; if no more characters exit and print

JSR LAB\_PRNA ; go print the character

INY ; increment index

LDA (ut2\_pl),Y ; get keyword address low byte

PHA ; save it for now

INY ; increment index

LDA (ut2\_pl),Y ; get keyword address high byte

LDY #$00

STA ut2\_ph ; save keyword pointer high byte

PLA ; pull low byte

STA ut2\_pl ; save keyword pointer low byte

LAB\_1540

LDA (ut2\_pl),Y ; get character

DEX ; decrement character count

BEQ LAB\_1508 ; if last character exit and print

JSR LAB\_PRNA ; go print the character

INY ; increment index

BNE LAB\_1540 ; loop for next character

; perform FOR

LAB\_FOR

LDA #$80 ; set FNX

STA Sufnxf ; set subscript/FNX flag

JSR LAB\_LET ; go do LET

PLA ; pull return address

PLA ; pull return address

LDA #$10 ; we need 16d bytes !

JSR LAB\_1212 ; check room on stack for A bytes

JSR LAB\_SNBS ; scan for next BASIC statement ([:] or [EOL])

CLC ; clear carry for add

TYA ; copy index to A

ADC Bpntrl ; add BASIC execute pointer low byte

PHA ; push onto stack

LDA Bpntrh ; get BASIC execute pointer high byte

ADC #$00 ; add carry

PHA ; push onto stack

LDA Clineh ; get current line high byte

PHA ; push onto stack

LDA Clinel ; get current line low byte

PHA ; push onto stack

LDA #TK\_TO ; get "TO" token

JSR LAB\_SCCA ; scan for CHR$(A) , else do syntax error then warm start

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

LDA FAC1\_s ; get FAC1 sign (b7)

ORA #$7F ; set all non sign bits

AND FAC1\_1 ; and FAC1 mantissa1

STA FAC1\_1 ; save FAC1 mantissa1

LDA #<LAB\_159F ; set return address low byte

LDY #>LAB\_159F ; set return address high byte

STA ut1\_pl ; save return address low byte

STY ut1\_ph ; save return address high byte

JMP LAB\_1B66 ; round FAC1 and put on stack (returns to next instruction)

LAB\_159F

LDA #<LAB\_259C ; set 1 pointer low addr (default step size)

LDY #>LAB\_259C ; set 1 pointer high addr

JSR LAB\_UFAC ; unpack memory (AY) into FAC1

JSR LAB\_GBYT ; scan memory

CMP #TK\_STEP ; compare with STEP token

BNE LAB\_15B3 ; jump if not "STEP"

;.was step so ..

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

LAB\_15B3

JSR LAB\_27CA ; return A=FF,C=1/-ve A=01,C=0/+ve

STA FAC1\_s ; set FAC1 sign (b7)

; this is +1 for +ve step and -1 for -ve step, in NEXT we

; compare the FOR value and the TO value and return +1 if

; FOR > TO, 0 if FOR = TO and -1 if FOR < TO. the value

; here (+/-1) is then compared to that result and if they

; are the same (+ve and FOR > TO or -ve and FOR < TO) then

; the loop is done

JSR LAB\_1B5B ; push sign, round FAC1 and put on stack

LDA Frnxth ; get var pointer for FOR/NEXT high byte

PHA ; push on stack

LDA Frnxtl ; get var pointer for FOR/NEXT low byte

PHA ; push on stack

LDA #TK\_FOR ; get FOR token

PHA ; push on stack

; interpreter inner loop

LAB\_15C2

JSR LAB\_1629 ; do CRTL-C check vector

LDA Bpntrl ; get BASIC execute pointer low byte

LDY Bpntrh ; get BASIC execute pointer high byte

LDX Clineh ; continue line is $FFxx for immediate mode

; ($00xx for RUN from immediate mode)

INX ; increment it (now $00 if immediate mode)

BEQ LAB\_15D1 ; branch if null (immediate mode)

STA Cpntrl ; save continue pointer low byte

STY Cpntrh ; save continue pointer high byte

LAB\_15D1

LDY #$00 ; clear index

LDA (Bpntrl),Y ; get next byte

BEQ LAB\_15DC ; branch if null [EOL]

CMP #':' ; compare with ":"

BEQ LAB\_15F6 ; branch if = (statement separator)

LAB\_15D9

JMP LAB\_SNER ; else syntax error then warm start

; have reached [EOL]

LAB\_15DC

LDY #$02 ; set index

LDA (Bpntrl),Y ; get next line pointer high byte

CLC ; clear carry for no "BREAK" message

BEQ LAB\_1651 ; if null go to immediate mode (was immediate or [EOT]

; marker)

INY ; increment index

LDA (Bpntrl),Y ; get line # low byte

STA Clinel ; save current line low byte

INY ; increment index

LDA (Bpntrl),Y ; get line # high byte

STA Clineh ; save current line high byte

TYA ; A now = 4

ADC Bpntrl ; add BASIC execute pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

BCC LAB\_15F6 ; branch if no overflow

INC Bpntrh ; else increment BASIC execute pointer high byte

LAB\_15F6

JSR LAB\_IGBY ; increment and scan memory

LAB\_15F9

JSR LAB\_15FF ; go interpret BASIC code from (Bpntrl)

LAB\_15FC

JMP LAB\_15C2 ; loop

; interpret BASIC code from (Bpntrl)

LAB\_15FF

BEQ LAB\_1628 ; exit if zero [EOL]

LAB\_1602

ASL ; \*2 bytes per vector and normalise token

BCS LAB\_1609 ; branch if was token

JMP LAB\_LET ; else go do implied LET

LAB\_1609

CMP #(TK\_TAB-$80)\*2 ; compare normalised token \* 2 with TAB

BCS LAB\_15D9 ; branch if A>=TAB (do syntax error then warm start)

; only tokens before TAB can start a line

TAY ; copy to index

LDA LAB\_CTBL+1,Y ; get vector high byte

PHA ; onto stack

LDA LAB\_CTBL,Y ; get vector low byte

PHA ; onto stack

JMP LAB\_IGBY ; jump to increment and scan memory

; then "return" to vector

; CTRL-C check jump. this is called as a subroutine but exits back via a jump if a

; key press is detected.

LAB\_1629

JMP (VEC\_CC) ; ctrl c check vector

; if there was a key press it gets back here ..

LAB\_1636

CMP #$03 ; compare with CTRL-C

; perform STOP

LAB\_STOP

BCS LAB\_163B ; branch if token follows STOP

; else just END

; END

LAB\_END

CLC ; clear the carry, indicate a normal program end

LAB\_163B

BNE LAB\_167A ; if wasn't CTRL-C or there is a following byte return

LDA Bpntrh ; get the BASIC execute pointer high byte

EOR #>Ibuffs ; compare with buffer address high byte (Cb unchanged)

BEQ LAB\_164F ; branch if the BASIC pointer is in the input buffer

; (can't continue in immediate mode)

; else ..

EOR #>Ibuffs ; correct the bits

LDY Bpntrl ; get BASIC execute pointer low byte

STY Cpntrl ; save continue pointer low byte

STA Cpntrh ; save continue pointer high byte

LAB\_1647

LDA Clinel ; get current line low byte

LDY Clineh ; get current line high byte

STA Blinel ; save break line low byte

STY Blineh ; save break line high byte

LAB\_164F

PLA ; pull return address low

PLA ; pull return address high

LAB\_1651

BCC LAB\_165E ; if was program end just do warm start

; else ..

LDA #<LAB\_BMSG ; point to "Break" low byte

LDY #>LAB\_BMSG ; point to "Break" high byte

JMP LAB\_1269 ; print "Break" and do warm start

LAB\_165E

JMP LAB\_1274 ; go do warm start

; perform RESTORE

LAB\_RESTORE

BNE LAB\_RESTOREn ; branch if next character not null (RESTORE n)

LAB\_161A

SEC ; set carry for subtract

LDA Smeml ; get start of mem low byte

SBC #$01 ; -1

LDY Smemh ; get start of mem high byte

BCS LAB\_1624 ; branch if no underflow

LAB\_uflow

DEY ; else decrement high byte

LAB\_1624

STA Dptrl ; save DATA pointer low byte

STY Dptrh ; save DATA pointer high byte

LAB\_1628

RTS

; is RESTORE n

LAB\_RESTOREn

JSR LAB\_GFPN ; get fixed-point number into temp integer

JSR LAB\_SNBL ; scan for next BASIC line

LDA Clineh ; get current line high byte

CMP Itemph ; compare with temporary integer high byte

BCS LAB\_reset\_search ; branch if >= (start search from beginning)

TYA ; else copy line index to A

SEC ; set carry (+1)

ADC Bpntrl ; add BASIC execute pointer low byte

LDX Bpntrh ; get BASIC execute pointer high byte

BCC LAB\_go\_search ; branch if no overflow to high byte

INX ; increment high byte

BCS LAB\_go\_search ; branch always (can never be carry clear)

; search for line # in temp (Itempl/Itemph) from start of mem pointer (Smeml)

LAB\_reset\_search

LDA Smeml ; get start of mem low byte

LDX Smemh ; get start of mem high byte

; search for line # in temp (Itempl/Itemph) from (AX)

LAB\_go\_search

JSR LAB\_SHLN ; search Basic for temp integer line number from AX

BCS LAB\_line\_found ; if carry set go set pointer

JMP LAB\_16F7 ; else go do "Undefined statement" error

LAB\_line\_found

; carry already set for subtract

LDA Baslnl ; get pointer low byte

SBC #$01 ; -1

LDY Baslnh ; get pointer high byte

BCS LAB\_1624 ; branch if no underflow (save DATA pointer and return)

BCC LAB\_uflow ; else decrement high byte then save DATA pointer and

; return (branch always)

; perform NULL

LAB\_NULL

JSR LAB\_GTBY ; get byte parameter

STX Nullct ; save new NULL count

LAB\_167A

RTS

; perform CONT

LAB\_CONT

BNE LAB\_167A ; if following byte exit to do syntax error

LDY Cpntrh ; get continue pointer high byte

BNE LAB\_166C ; go do continue if we can

LDX #$1E ; error code $1E ("Can't continue" error)

JMP LAB\_XERR ; do error #X, then warm start

; we can continue so ..

LAB\_166C

LDA #TK\_ON ; set token for ON

JSR LAB\_IRQ ; set IRQ flags

LDA #TK\_ON ; set token for ON

JSR LAB\_NMI ; set NMI flags

STY Bpntrh ; save BASIC execute pointer high byte

LDA Cpntrl ; get continue pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

LDA Blinel ; get break line low byte

LDY Blineh ; get break line high byte

STA Clinel ; set current line low byte

STY Clineh ; set current line high byte

RTS

; perform RUN

LAB\_RUN

BNE LAB\_1696 ; branch if RUN n

JMP LAB\_1477 ; reset execution to start, clear variables, flush stack and

; return

; does RUN n

LAB\_1696

JSR LAB\_147A ; go do "CLEAR"

BEQ LAB\_16B0 ; get n and do GOTO n (branch always as CLEAR sets Z=1)

; perform DO

LAB\_DO

LDA #$05 ; need 5 bytes for DO

JSR LAB\_1212 ; check room on stack for A bytes

LDA Bpntrh ; get BASIC execute pointer high byte

PHA ; push on stack

LDA Bpntrl ; get BASIC execute pointer low byte

PHA ; push on stack

LDA Clineh ; get current line high byte

PHA ; push on stack

LDA Clinel ; get current line low byte

PHA ; push on stack

LDA #TK\_DO ; token for DO

PHA ; push on stack

JSR LAB\_GBYT ; scan memory

JMP LAB\_15C2 ; go do interpreter inner loop

; perform GOSUB

LAB\_GOSUB

LDA #$05 ; need 5 bytes for GOSUB

JSR LAB\_1212 ; check room on stack for A bytes

LDA Bpntrh ; get BASIC execute pointer high byte

PHA ; push on stack

LDA Bpntrl ; get BASIC execute pointer low byte

PHA ; push on stack

LDA Clineh ; get current line high byte

PHA ; push on stack

LDA Clinel ; get current line low byte

PHA ; push on stack

LDA #TK\_GOSUB ; token for GOSUB

PHA ; push on stack

LAB\_16B0

JSR LAB\_GBYT ; scan memory

JSR LAB\_GOTO ; perform GOTO n

JMP LAB\_15C2 ; go do interpreter inner loop

; (can't RTS, we used the stack!)

; perform GOTO

LAB\_GOTO

JSR LAB\_GFPN ; get fixed-point number into temp integer

JSR LAB\_SNBL ; scan for next BASIC line

LDA Clineh ; get current line high byte

CMP Itemph ; compare with temporary integer high byte

BCS LAB\_16D0 ; branch if >= (start search from beginning)

TYA ; else copy line index to A

SEC ; set carry (+1)

ADC Bpntrl ; add BASIC execute pointer low byte

LDX Bpntrh ; get BASIC execute pointer high byte

BCC LAB\_16D4 ; branch if no overflow to high byte

INX ; increment high byte

BCS LAB\_16D4 ; branch always (can never be carry)

; search for line # in temp (Itempl/Itemph) from start of mem pointer (Smeml)

LAB\_16D0

LDA Smeml ; get start of mem low byte

LDX Smemh ; get start of mem high byte

; search for line # in temp (Itempl/Itemph) from (AX)

LAB\_16D4

JSR LAB\_SHLN ; search Basic for temp integer line number from AX

BCC LAB\_16F7 ; if carry clear go do "Undefined statement" error

; (unspecified statement)

; carry already set for subtract

LDA Baslnl ; get pointer low byte

SBC #$01 ; -1

STA Bpntrl ; save BASIC execute pointer low byte

LDA Baslnh ; get pointer high byte

SBC #$00 ; subtract carry

STA Bpntrh ; save BASIC execute pointer high byte

LAB\_16E5

RTS

LAB\_DONOK

LDX #$22 ; error code $22 ("LOOP without DO" error)

JMP LAB\_XERR ; do error #X, then warm start

; perform LOOP

LAB\_LOOP

TAY ; save following token

TSX ; copy stack pointer

LDA LAB\_STAK+3,X ; get token byte from stack

CMP #TK\_DO ; compare with DO token

BNE LAB\_DONOK ; branch if no matching DO

INX ; dump calling routine return address

INX ; dump calling routine return address

TXS ; correct stack

TYA ; get saved following token back

BEQ LoopAlways ; if no following token loop forever

; (stack pointer in X)

CMP #':' ; could be ':'

BEQ LoopAlways ; if :... loop forever

SBC #TK\_UNTIL ; subtract token for UNTIL, we know carry is set here

TAX ; copy to X (if it was UNTIL then Y will be correct)

BEQ DoRest ; branch if was UNTIL

DEX ; decrement result

BNE LAB\_16FC ; if not WHILE go do syntax error and warm start

; only if the token was WHILE will this fail

DEX ; set invert result byte

DoRest

STX Frnxth ; save invert result byte

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_EVEX ; evaluate expression

LDA FAC1\_e ; get FAC1 exponent

BEQ DoCmp ; if =0 go do straight compare

LDA #$FF ; else set all bits

DoCmp

TSX ; copy stack pointer

EOR Frnxth ; EOR with invert byte

BNE LoopDone ; if <> 0 clear stack and back to interpreter loop

; loop condition wasn't met so do it again

LoopAlways

LDA LAB\_STAK+2,X ; get current line low byte

STA Clinel ; save current line low byte

LDA LAB\_STAK+3,X ; get current line high byte

STA Clineh ; save current line high byte

LDA LAB\_STAK+4,X ; get BASIC execute pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

LDA LAB\_STAK+5,X ; get BASIC execute pointer high byte

STA Bpntrh ; save BASIC execute pointer high byte

JSR LAB\_GBYT ; scan memory

JMP LAB\_15C2 ; go do interpreter inner loop

; clear stack and back to interpreter loop

LoopDone

INX ; dump DO token

INX ; dump current line low byte

INX ; dump current line high byte

INX ; dump BASIC execute pointer low byte

INX ; dump BASIC execute pointer high byte

TXS ; correct stack

JMP LAB\_DATA ; go perform DATA (find : or [EOL])

; do the return without gosub error

LAB\_16F4

LDX #$04 ; error code $04 ("RETURN without GOSUB" error)

.byte $2C ; makes next line BIT LAB\_0EA2

LAB\_16F7 ; do undefined statement error

LDX #$0E ; error code $0E ("Undefined statement" error)

JMP LAB\_XERR ; do error #X, then warm start

; perform RETURN

LAB\_RETURN

BNE LAB\_16E5 ; exit if following token (to allow syntax error)

LAB\_16E8

PLA ; dump calling routine return address

PLA ; dump calling routine return address

PLA ; pull token

CMP #TK\_GOSUB ; compare with GOSUB token

BNE LAB\_16F4 ; branch if no matching GOSUB

LAB\_16FF

PLA ; pull current line low byte

STA Clinel ; save current line low byte

PLA ; pull current line high byte

STA Clineh ; save current line high byte

PLA ; pull BASIC execute pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

PLA ; pull BASIC execute pointer high byte

STA Bpntrh ; save BASIC execute pointer high byte

; now do the DATA statement as we could be returning into

; the middle of an ON <var> GOSUB n,m,p,q line

; (the return address used by the DATA statement is the one

; pushed before the GOSUB was executed!)

; perform DATA

LAB\_DATA

JSR LAB\_SNBS ; scan for next BASIC statement ([:] or [EOL])

; set BASIC execute pointer

LAB\_170F

TYA ; copy index to A

CLC ; clear carry for add

ADC Bpntrl ; add BASIC execute pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

BCC LAB\_1719 ; skip next if no carry

INC Bpntrh ; else increment BASIC execute pointer high byte

LAB\_1719

RTS

LAB\_16FC

JMP LAB\_SNER ; do syntax error then warm start

; scan for next BASIC statement ([:] or [EOL])

; returns Y as index to [:] or [EOL]

LAB\_SNBS

LDX #':' ; set look for character = ":"

.byte $2C ; makes next line BIT $00A2

; scan for next BASIC line

; returns Y as index to [EOL]

LAB\_SNBL

LDX #$00 ; set alt search character = [EOL]

LDY #$00 ; set search character = [EOL]

STY Asrch ; store search character

LAB\_1725

TXA ; get alt search character

EOR Asrch ; toggle search character, effectively swap with $00

STA Asrch ; save swapped search character

LAB\_172D

LDA (Bpntrl),Y ; get next byte

BEQ LAB\_1719 ; exit if null [EOL]

CMP Asrch ; compare with search character

BEQ LAB\_1719 ; exit if found

INY ; increment index

CMP #$22 ; compare current character with open quote

BNE LAB\_172D ; if not open quote go get next character

BEQ LAB\_1725 ; if found go swap search character for alt search character

; perform IF

LAB\_IF

JSR LAB\_EVEX ; evaluate the expression

JSR LAB\_GBYT ; scan memory

CMP #TK\_THEN ; compare with THEN token

BEQ LAB\_174B ; if it was THEN go do IF

; wasn't IF .. THEN so must be IF .. GOTO

CMP #TK\_GOTO ; compare with GOTO token

BNE LAB\_16FC ; if it wasn't GOTO go do syntax error

LDX Bpntrl ; save the basic pointer low byte

LDY Bpntrh ; save the basic pointer high byte

JSR LAB\_IGBY ; increment and scan memory

BCS LAB\_16FC ; if not numeric go do syntax error

STX Bpntrl ; restore the basic pointer low byte

STY Bpntrh ; restore the basic pointer high byte

LAB\_174B

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_174E ; if the result was zero go look for an ELSE

JSR LAB\_IGBY ; else increment and scan memory

BCS LAB\_174D ; if not numeric go do var or keyword

LAB\_174C

JMP LAB\_GOTO ; else was numeric so do GOTO n

; is var or keyword

LAB\_174D

CMP #TK\_RETURN ; compare the byte with the token for RETURN

BNE LAB\_174G ; if it wasn't RETURN go interpret BASIC code from (Bpntrl)

; and return to this code to process any following code

JMP LAB\_1602 ; else it was RETURN so interpret BASIC code from (Bpntrl)

; but don't return here

LAB\_174G

JSR LAB\_15FF ; interpret BASIC code from (Bpntrl)

; the IF was executed and there may be a following ELSE so the code needs to return

; here to check and ignore the ELSE if present

LDY #$00 ; clear the index

LDA (Bpntrl),Y ; get the next BASIC byte

CMP #TK\_ELSE ; compare it with the token for ELSE

BEQ LAB\_DATA ; if ELSE ignore the following statement

; there was no ELSE so continue execution of IF <expr> THEN <stat> [: <stat>]. any

; following ELSE will, correctly, cause a syntax error

RTS ; else return to the interpreter inner loop

; perform ELSE after IF

LAB\_174E

LDY #$00 ; clear the BASIC byte index

LDX #$01 ; clear the nesting depth

LAB\_1750

INY ; increment the BASIC byte index

LDA (Bpntrl),Y ; get the next BASIC byte

BEQ LAB\_1753 ; if EOL go add the pointer and return

CMP #TK\_IF ; compare the byte with the token for IF

BNE LAB\_1752 ; if not IF token skip the depth increment

INX ; else increment the nesting depth ..

BNE LAB\_1750 ; .. and continue looking

LAB\_1752

CMP #TK\_ELSE ; compare the byte with the token for ELSE

BNE LAB\_1750 ; if not ELSE token continue looking

DEX ; was ELSE so decrement the nesting depth

BNE LAB\_1750 ; loop if still nested

INY ; increment the BASIC byte index past the ELSE

; found the matching ELSE, now do <{n|statement}>

LAB\_1753

TYA ; else copy line index to A

CLC ; clear carry for add

ADC Bpntrl ; add the BASIC execute pointer low byte

STA Bpntrl ; save the BASIC execute pointer low byte

BCC LAB\_1754 ; branch if no overflow to high byte

INC Bpntrh ; else increment the BASIC execute pointer high byte

LAB\_1754

JSR LAB\_GBYT ; scan memory

BCC LAB\_174C ; if numeric do GOTO n

; the code will return to the interpreter loop at the

; tail end of the GOTO <n>

JMP LAB\_15FF ; interpret BASIC code from (Bpntrl)

; the code will return to the interpreter loop at the

; tail end of the <statement>

; perform REM, skip (rest of) line

LAB\_REM

JSR LAB\_SNBL ; scan for next BASIC line

JMP LAB\_170F ; go set BASIC execute pointer and return, branch always

LAB\_16FD

JMP LAB\_SNER ; do syntax error then warm start

; perform ON

LAB\_ON

CMP #TK\_IRQ ; was it IRQ token ?

BNE LAB\_NOIN ; if not go check NMI

JMP LAB\_SIRQ ; else go set-up IRQ

LAB\_NOIN

CMP #TK\_NMI ; was it NMI token ?

BNE LAB\_NONM ; if not go do normal ON command

JMP LAB\_SNMI ; else go set-up NMI

LAB\_NONM

JSR LAB\_GTBY ; get byte parameter

PHA ; push GOTO/GOSUB token

CMP #TK\_GOSUB ; compare with GOSUB token

BEQ LAB\_176B ; branch if GOSUB

CMP #TK\_GOTO ; compare with GOTO token

LAB\_1767

BNE LAB\_16FD ; if not GOTO do syntax error then warm start

; next character was GOTO or GOSUB

LAB\_176B

DEC FAC1\_3 ; decrement index (byte value)

BNE LAB\_1773 ; branch if not zero

PLA ; pull GOTO/GOSUB token

JMP LAB\_1602 ; go execute it

LAB\_1773

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_GFPN ; get fixed-point number into temp integer (skip this n)

; (we could LDX #',' and JSR LAB\_SNBL+2, then we

; just BNE LAB\_176B for the loop. should be quicker ..

; no we can't, what if we meet a colon or [EOL]?)

CMP #$2C ; compare next character with ","

BEQ LAB\_176B ; loop if ","

LAB\_177E

PLA ; else pull keyword token (run out of options)

; also dump +/-1 pointer low byte and exit

LAB\_177F

RTS

; takes n \* 106 + 11 cycles where n is the number of digits

; get fixed-point number into temp integer

LAB\_GFPN

LDX #$00 ; clear reg

STX Itempl ; clear temporary integer low byte

LAB\_1785

STX Itemph ; save temporary integer high byte

BCS LAB\_177F ; return if carry set, end of scan, character was

; not 0-9

CPX #$19 ; compare high byte with $19

TAY ; ensure Zb = 0 if the branch is taken

BCS LAB\_1767 ; branch if >=, makes max line # 63999 because next

; bit does \*$0A, = 64000, compare at target will fail

; and do syntax error

SBC #'0'-1 ; subtract "0", $2F + carry, from byte

TAY ; copy binary digit

LDA Itempl ; get temporary integer low byte

ASL ; \*2 low byte

ROL Itemph ; \*2 high byte

ASL ; \*2 low byte

ROL Itemph ; \*2 high byte, \*4

ADC Itempl ; + low byte, \*5

STA Itempl ; save it

TXA ; get high byte copy to A

ADC Itemph ; + high byte, \*5

ASL Itempl ; \*2 low byte, \*10d

ROL ; \*2 high byte, \*10d

TAX ; copy high byte back to X

TYA ; get binary digit back

ADC Itempl ; add number low byte

STA Itempl ; save number low byte

BCC LAB\_17B3 ; if no overflow to high byte get next character

INX ; else increment high byte

LAB\_17B3

JSR LAB\_IGBY ; increment and scan memory

JMP LAB\_1785 ; loop for next character

; perform DEC

LAB\_DEC

LDA #<LAB\_2AFD ; set -1 pointer low byte

.byte $2C ; BIT abs to skip the LDA below

; perform INC

LAB\_INC

LDA #<LAB\_259C ; set 1 pointer low byte

LAB\_17B5

PHA ; save +/-1 pointer low byte

LAB\_17B7

JSR LAB\_GVAR ; get var address

LDX Dtypef ; get data type flag, $FF=string, $00=numeric

BMI IncrErr ; exit if string

STA Lvarpl ; save var address low byte

STY Lvarph ; save var address high byte

JSR LAB\_UFAC ; unpack memory (AY) into FAC1

PLA ; get +/-1 pointer low byte

PHA ; save +/-1 pointer low byte

LDY #>LAB\_259C ; set +/-1 pointer high byte (both the same)

JSR LAB\_246C ; add (AY) to FAC1

JSR LAB\_PFAC ; pack FAC1 into variable (Lvarpl)

JSR LAB\_GBYT ; scan memory

CMP #',' ; compare with ","

BNE LAB\_177E ; exit if not "," (either end or error)

; was "," so another INCR variable to do

JSR LAB\_IGBY ; increment and scan memory

JMP LAB\_17B7 ; go do next var

IncrErr

JMP LAB\_1ABC ; do "Type mismatch" error then warm start

; perform LET

LAB\_LET

JSR LAB\_GVAR ; get var address

STA Lvarpl ; save var address low byte

STY Lvarph ; save var address high byte

LDA #TK\_EQUAL ; get = token

JSR LAB\_SCCA ; scan for CHR$(A), else do syntax error then warm start

LDA Dtypef ; get data type flag, $FF=string, $00=numeric

PHA ; push data type flag

JSR LAB\_EVEX ; evaluate expression

PLA ; pop data type flag

ROL ; set carry if type = string

JSR LAB\_CKTM ; type match check, set C for string

BNE LAB\_17D5 ; branch if string

JMP LAB\_PFAC ; pack FAC1 into variable (Lvarpl) and return

; string LET

LAB\_17D5

LDY #$02 ; set index to pointer high byte

LDA (des\_pl),Y ; get string pointer high byte

CMP Sstorh ; compare bottom of string space high byte

BCC LAB\_17F4 ; if less assign value and exit (was in program memory)

BNE LAB\_17E6 ; branch if >

; else was equal so compare low bytes

DEY ; decrement index

LDA (des\_pl),Y ; get pointer low byte

CMP Sstorl ; compare bottom of string space low byte

BCC LAB\_17F4 ; if less assign value and exit (was in program memory)

; pointer was >= to bottom of string space pointer

LAB\_17E6

LDY des\_ph ; get descriptor pointer high byte

CPY Svarh ; compare start of vars high byte

BCC LAB\_17F4 ; branch if less (descriptor is on stack)

BNE LAB\_17FB ; branch if greater (descriptor is not on stack)

; else high bytes were equal so ..

LDA des\_pl ; get descriptor pointer low byte

CMP Svarl ; compare start of vars low byte

BCS LAB\_17FB ; branch if >= (descriptor is not on stack)

LAB\_17F4

LDA des\_pl ; get descriptor pointer low byte

LDY des\_ph ; get descriptor pointer high byte

JMP LAB\_1811 ; clean stack, copy descriptor to variable and return

; make space and copy string

LAB\_17FB

LDY #$00 ; index to length

LDA (des\_pl),Y ; get string length

JSR LAB\_209C ; copy string

LDA des\_2l ; get descriptor pointer low byte

LDY des\_2h ; get descriptor pointer high byte

STA ssptr\_l ; save descriptor pointer low byte

STY ssptr\_h ; save descriptor pointer high byte

JSR LAB\_228A ; copy string from descriptor (sdescr) to (Sutill)

LDA #<FAC1\_e ; set descriptor pointer low byte

LDY #>FAC1\_e ; get descriptor pointer high byte

; clean stack and assign value to string variable

LAB\_1811

STA des\_2l ; save descriptor\_2 pointer low byte

STY des\_2h ; save descriptor\_2 pointer high byte

JSR LAB\_22EB ; clean descriptor stack, YA = pointer

LDY #$00 ; index to length

LDA (des\_2l),Y ; get string length

STA (Lvarpl),Y ; copy to let string variable

INY ; index to string pointer low byte

LDA (des\_2l),Y ; get string pointer low byte

STA (Lvarpl),Y ; copy to let string variable

INY ; index to string pointer high byte

LDA (des\_2l),Y ; get string pointer high byte

STA (Lvarpl),Y ; copy to let string variable

RTS

; perform GET

LAB\_GET

JSR LAB\_GVAR ; get var address

STA Lvarpl ; save var address low byte

STY Lvarph ; save var address high byte

JSR INGET ; get input byte

LDX Dtypef ; get data type flag, $FF=string, $00=numeric

BMI LAB\_GETS ; go get string character

; was numeric get

TAY ; copy character to Y

JSR LAB\_1FD0 ; convert Y to byte in FAC1

JMP LAB\_PFAC ; pack FAC1 into variable (Lvarpl) and return

LAB\_GETS

PHA ; save character

LDA #$01 ; string is single byte

BCS LAB\_IsByte ; branch if byte received

PLA ; string is null

LAB\_IsByte

JSR LAB\_MSSP ; make string space A bytes long A=$AC=length,

; X=$AD=Sutill=ptr low byte, Y=$AE=Sutilh=ptr high byte

BEQ LAB\_NoSt ; skip store if null string

PLA ; get character back

LDY #$00 ; clear index

STA (str\_pl),Y ; save byte in string (byte IS string!)

LAB\_NoSt

JSR LAB\_RTST ; check for space on descriptor stack then put address

; and length on descriptor stack and update stack pointers

JMP LAB\_17D5 ; do string LET and return

; perform PRINT

LAB\_1829

JSR LAB\_18C6 ; print string from Sutill/Sutilh

LAB\_182C

JSR LAB\_GBYT ; scan memory

; PRINT

LAB\_PRINT

BEQ LAB\_CRLF ; if nothing following just print CR/LF

LAB\_1831

CMP #TK\_TAB ; compare with TAB( token

BEQ LAB\_18A2 ; go do TAB/SPC

CMP #TK\_SPC ; compare with SPC( token

BEQ LAB\_18A2 ; go do TAB/SPC

CMP #',' ; compare with ","

BEQ LAB\_188B ; go do move to next TAB mark

CMP #';' ; compare with ";"

BEQ LAB\_18BD ; if ";" continue with PRINT processing

JSR LAB\_EVEX ; evaluate expression

BIT Dtypef ; test data type flag, $FF=string, $00=numeric

BMI LAB\_1829 ; branch if string

JSR LAB\_296E ; convert FAC1 to string

JSR LAB\_20AE ; print " terminated string to Sutill/Sutilh

LDY #$00 ; clear index

; don't check fit if terminal width byte is zero

LDA TWidth ; get terminal width byte

BEQ LAB\_185E ; skip check if zero

SEC ; set carry for subtract

SBC TPos ; subtract terminal position

SBC (des\_pl),Y ; subtract string length

BCS LAB\_185E ; branch if less than terminal width

JSR LAB\_CRLF ; else print CR/LF

LAB\_185E

JSR LAB\_18C6 ; print string from Sutill/Sutilh

BEQ LAB\_182C ; always go continue processing line

; CR/LF return to BASIC from BASIC input handler

LAB\_1866

LDA #$00 ; clear byte

STA Ibuffs,X ; null terminate input

LDX #<Ibuffs ; set X to buffer start-1 low byte

LDY #>Ibuffs ; set Y to buffer start-1 high byte

; print CR/LF

LAB\_CRLF

LDA #$0D ; load [CR]

JSR LAB\_PRNA ; go print the character

LDA #$0A ; load [LF]

BNE LAB\_PRNA ; go print the character and return, branch always

LAB\_188B

LDA TPos ; get terminal position

CMP Iclim ; compare with input column limit

BCC LAB\_1897 ; branch if less

JSR LAB\_CRLF ; else print CR/LF (next line)

BNE LAB\_18BD ; continue with PRINT processing (branch always)

LAB\_1897

SEC ; set carry for subtract

LAB\_1898

SBC TabSiz ; subtract TAB size

BCS LAB\_1898 ; loop if result was +ve

EOR #$FF ; complement it

ADC #$01 ; +1 (twos complement)

BNE LAB\_18B6 ; always print A spaces (result is never $00)

; do TAB/SPC

LAB\_18A2

PHA ; save token

JSR LAB\_SGBY ; scan and get byte parameter

CMP #$29 ; is next character )

BNE LAB\_1910 ; if not do syntax error then warm start

PLA ; get token back

CMP #TK\_TAB ; was it TAB ?

BNE LAB\_18B7 ; if not go do SPC

; calculate TAB offset

TXA ; copy integer value to A

SBC TPos ; subtract terminal position

BCC LAB\_18BD ; branch if result was < 0 (can't TAB backwards)

; print A spaces

LAB\_18B6

TAX ; copy result to X

LAB\_18B7

TXA ; set flags on size for SPC

BEQ LAB\_18BD ; branch if result was = $0, already here

; print X spaces

LAB\_18BA

JSR LAB\_18E0 ; print " "

DEX ; decrement count

BNE LAB\_18BA ; loop if not all done

; continue with PRINT processing

LAB\_18BD

JSR LAB\_IGBY ; increment and scan memory

BNE LAB\_1831 ; if more to print go do it

RTS

; print null terminated string from memory

LAB\_18C3

JSR LAB\_20AE ; print " terminated string to Sutill/Sutilh

; print string from Sutill/Sutilh

LAB\_18C6

JSR LAB\_22B6 ; pop string off descriptor stack, or from top of string

; space returns with A = length, X=$71=pointer low byte,

; Y=$72=pointer high byte

LDY #$00 ; reset index

TAX ; copy length to X

BEQ LAB\_188C ; exit (RTS) if null string

LAB\_18CD

LDA (ut1\_pl),Y ; get next byte

JSR LAB\_PRNA ; go print the character

INY ; increment index

DEX ; decrement count

BNE LAB\_18CD ; loop if not done yet

RTS

; Print single format character

; print " "

LAB\_18E0

LDA #$20 ; load " "

.byte $2C ; change next line to BIT LAB\_3FA9

; print "?" character

LAB\_18E3

LDA #$3F ; load "?" character

; print character in A

; now includes the null handler

; also includes infinite line length code

; note! some routines expect this one to exit with Zb=0

LAB\_PRNA

CMP #' ' ; compare with " "

BCC LAB\_18F9 ; branch if less (non printing)

; else printable character

PHA ; save the character

; don't check fit if terminal width byte is zero

LDA TWidth ; get terminal width

BNE LAB\_18F0 ; branch if not zero (not infinite length)

; is "infinite line" so check TAB position

LDA TPos ; get position

SBC TabSiz ; subtract TAB size, carry set by CMP #$20 above

BNE LAB\_18F7 ; skip reset if different

STA TPos ; else reset position

BEQ LAB\_18F7 ; go print character

LAB\_18F0

CMP TPos ; compare with terminal character position

BNE LAB\_18F7 ; branch if not at end of line

JSR LAB\_CRLF ; else print CR/LF

LAB\_18F7

INC TPos ; increment terminal position

PLA ; get character back

LAB\_18F9

JSR V\_OUTP ; output byte via output vector

CMP #$0D ; compare with [CR]

BNE LAB\_188A ; branch if not [CR]

; else print nullct nulls after the [CR]

STX TempB ; save buffer index

LDX Nullct ; get null count

BEQ LAB\_1886 ; branch if no nulls

LDA #$00 ; load [NULL]

LAB\_1880

JSR LAB\_PRNA ; go print the character

DEX ; decrement count

BNE LAB\_1880 ; loop if not all done

LDA #$0D ; restore the character (and set the flags)

LAB\_1886

STX TPos ; clear terminal position (X always = zero when we get here)

LDX TempB ; restore buffer index

LAB\_188A

AND #$FF ; set the flags

LAB\_188C

RTS

; handle bad input data

LAB\_1904

LDA Imode ; get input mode flag, $00=INPUT, $00=READ

BPL LAB\_1913 ; branch if INPUT (go do redo)

LDA Dlinel ; get current DATA line low byte

LDY Dlineh ; get current DATA line high byte

STA Clinel ; save current line low byte

STY Clineh ; save current line high byte

LAB\_1910

JMP LAB\_SNER ; do syntax error then warm start

; mode was INPUT

LAB\_1913

LDA #<LAB\_REDO ; point to redo message (low addr)

LDY #>LAB\_REDO ; point to redo message (high addr)

JSR LAB\_18C3 ; print null terminated string from memory

LDA Cpntrl ; get continue pointer low byte

LDY Cpntrh ; get continue pointer high byte

STA Bpntrl ; save BASIC execute pointer low byte

STY Bpntrh ; save BASIC execute pointer high byte

RTS

; perform INPUT

LAB\_INPUT

CMP #$22 ; compare next byte with open quote

BNE LAB\_1934 ; branch if no prompt string

JSR LAB\_1BC1 ; print "..." string

LDA #$3B ; load A with ";"

JSR LAB\_SCCA ; scan for CHR$(A), else do syntax error then warm start

JSR LAB\_18C6 ; print string from Sutill/Sutilh

; done with prompt, now get data

LAB\_1934

JSR LAB\_CKRN ; check not Direct, back here if ok

JSR LAB\_INLN ; print "? " and get BASIC input

LDA #$00 ; set mode = INPUT

CMP Ibuffs ; test first byte in buffer

BNE LAB\_1953 ; branch if not null input

CLC ; was null input so clear carry to exit program

JMP LAB\_1647 ; go do BREAK exit

; perform READ

LAB\_READ

LDX Dptrl ; get DATA pointer low byte

LDY Dptrh ; get DATA pointer high byte

LDA #$80 ; set mode = READ

LAB\_1953

STA Imode ; set input mode flag, $00=INPUT, $80=READ

STX Rdptrl ; save READ pointer low byte

STY Rdptrh ; save READ pointer high byte

; READ or INPUT next variable from list

LAB\_195B

JSR LAB\_GVAR ; get (var) address

STA Lvarpl ; save address low byte

STY Lvarph ; save address high byte

LDA Bpntrl ; get BASIC execute pointer low byte

LDY Bpntrh ; get BASIC execute pointer high byte

STA Itempl ; save as temporary integer low byte

STY Itemph ; save as temporary integer high byte

LDX Rdptrl ; get READ pointer low byte

LDY Rdptrh ; get READ pointer high byte

STX Bpntrl ; set BASIC execute pointer low byte

STY Bpntrh ; set BASIC execute pointer high byte

JSR LAB\_GBYT ; scan memory

BNE LAB\_1988 ; branch if not null

; pointer was to null entry

BIT Imode ; test input mode flag, $00=INPUT, $80=READ

BMI LAB\_19DD ; branch if READ

; mode was INPUT

JSR LAB\_18E3 ; print "?" character (double ? for extended input)

JSR LAB\_INLN ; print "? " and get BASIC input

STX Bpntrl ; set BASIC execute pointer low byte

STY Bpntrh ; set BASIC execute pointer high byte

LAB\_1985

JSR LAB\_GBYT ; scan memory

LAB\_1988

BIT Dtypef ; test data type flag, $FF=string, $00=numeric

BPL LAB\_19B0 ; branch if numeric

; else get string

STA Srchc ; save search character

CMP #$22 ; was it quote ?

BEQ LAB\_1999 ; branch if so

LDA #':' ; else search character is ":"

STA Srchc ; set new search character

LDA #',' ; other search character is ","

CLC ; clear carry for add

LAB\_1999

STA Asrch ; set second search character

LDA Bpntrl ; get BASIC execute pointer low byte

LDY Bpntrh ; get BASIC execute pointer high byte

ADC #$00 ; c is =1 if we came via the BEQ LAB\_1999, else =0

BCC LAB\_19A4 ; branch if no execute pointer low byte rollover

INY ; else increment high byte

LAB\_19A4

JSR LAB\_20B4 ; print Srchc or Asrch terminated string to Sutill/Sutilh

JSR LAB\_23F3 ; restore BASIC execute pointer from temp (Btmpl/Btmph)

JSR LAB\_17D5 ; go do string LET

JMP LAB\_19B6 ; go check string terminator

; get numeric INPUT

LAB\_19B0

JSR LAB\_2887 ; get FAC1 from string

JSR LAB\_PFAC ; pack FAC1 into (Lvarpl)

LAB\_19B6

JSR LAB\_GBYT ; scan memory

BEQ LAB\_19C5 ; branch if null (last entry)

CMP #',' ; else compare with ","

BEQ LAB\_19C2 ; branch if ","

JMP LAB\_1904 ; else go handle bad input data

; got good input data

LAB\_19C2

JSR LAB\_IGBY ; increment and scan memory

LAB\_19C5

LDA Bpntrl ; get BASIC execute pointer low byte (temp READ/INPUT ptr)

LDY Bpntrh ; get BASIC execute pointer high byte (temp READ/INPUT ptr)

STA Rdptrl ; save for now

STY Rdptrh ; save for now

LDA Itempl ; get temporary integer low byte (temp BASIC execute ptr)

LDY Itemph ; get temporary integer high byte (temp BASIC execute ptr)

STA Bpntrl ; set BASIC execute pointer low byte

STY Bpntrh ; set BASIC execute pointer high byte

JSR LAB\_GBYT ; scan memory

BEQ LAB\_1A03 ; if null go do extra ignored message

JSR LAB\_1C01 ; else scan for "," , else do syntax error then warm start

JMP LAB\_195B ; go INPUT next variable from list

; find next DATA statement or do "Out of DATA" error

LAB\_19DD

JSR LAB\_SNBS ; scan for next BASIC statement ([:] or [EOL])

INY ; increment index

TAX ; copy character ([:] or [EOL])

BNE LAB\_19F6 ; branch if [:]

LDX #$06 ; set for "Out of DATA" error

INY ; increment index, now points to next line pointer high byte

LDA (Bpntrl),Y ; get next line pointer high byte

BEQ LAB\_1A54 ; branch if end (eventually does error X)

INY ; increment index

LDA (Bpntrl),Y ; get next line # low byte

STA Dlinel ; save current DATA line low byte

INY ; increment index

LDA (Bpntrl),Y ; get next line # high byte

INY ; increment index

STA Dlineh ; save current DATA line high byte

LAB\_19F6

LDA (Bpntrl),Y ; get byte

INY ; increment index

TAX ; copy to X

JSR LAB\_170F ; set BASIC execute pointer

CPX #TK\_DATA ; compare with "DATA" token

BEQ LAB\_1985 ; was "DATA" so go do next READ

BNE LAB\_19DD ; go find next statement if not "DATA"

; end of INPUT/READ routine

LAB\_1A03

LDA Rdptrl ; get temp READ pointer low byte

LDY Rdptrh ; get temp READ pointer high byte

LDX Imode ; get input mode flag, $00=INPUT, $80=READ

BPL LAB\_1A0E ; branch if INPUT

JMP LAB\_1624 ; save AY as DATA pointer and return

; we were getting INPUT

LAB\_1A0E

LDY #$00 ; clear index

LDA (Rdptrl),Y ; get next byte

BNE LAB\_1A1B ; error if not end of INPUT

RTS

; user typed too much

LAB\_1A1B

LDA #<LAB\_IMSG ; point to extra ignored message (low addr)

LDY #>LAB\_IMSG ; point to extra ignored message (high addr)

JMP LAB\_18C3 ; print null terminated string from memory and return

; search the stack for FOR activity

; exit with z=1 if FOR else exit with z=0

LAB\_11A1

TSX ; copy stack pointer

INX ; +1 pass return address

INX ; +2 pass return address

INX ; +3 pass calling routine return address

INX ; +4 pass calling routine return address

LAB\_11A6

LDA LAB\_STAK+1,X ; get token byte from stack

CMP #TK\_FOR ; is it FOR token

BNE LAB\_11CE ; exit if not FOR token

; was FOR token

LDA Frnxth ; get var pointer for FOR/NEXT high byte

BNE LAB\_11BB ; branch if not null

LDA LAB\_STAK+2,X ; get FOR variable pointer low byte

STA Frnxtl ; save var pointer for FOR/NEXT low byte

LDA LAB\_STAK+3,X ; get FOR variable pointer high byte

STA Frnxth ; save var pointer for FOR/NEXT high byte

LAB\_11BB

CMP LAB\_STAK+3,X ; compare var pointer with stacked var pointer (high byte)

BNE LAB\_11C7 ; branch if no match

LDA Frnxtl ; get var pointer for FOR/NEXT low byte

CMP LAB\_STAK+2,X ; compare var pointer with stacked var pointer (low byte)

BEQ LAB\_11CE ; exit if match found

LAB\_11C7

TXA ; copy index

CLC ; clear carry for add

ADC #$10 ; add FOR stack use size

TAX ; copy back to index

BNE LAB\_11A6 ; loop if not at start of stack

LAB\_11CE

RTS

; perform NEXT

LAB\_NEXT

BNE LAB\_1A46 ; branch if NEXT var

LDY #$00 ; else clear Y

BEQ LAB\_1A49 ; branch always (no variable to search for)

; NEXT var

LAB\_1A46

JSR LAB\_GVAR ; get variable address

LAB\_1A49

STA Frnxtl ; store variable pointer low byte

STY Frnxth ; store variable pointer high byte

; (both cleared if no variable defined)

JSR LAB\_11A1 ; search the stack for FOR activity

BEQ LAB\_1A56 ; branch if found

LDX #$00 ; else set error $00 ("NEXT without FOR" error)

LAB\_1A54

BEQ LAB\_1ABE ; do error #X, then warm start

LAB\_1A56

TXS ; set stack pointer, X set by search, dumps return addresses

TXA ; copy stack pointer

SEC ; set carry for subtract

SBC #$F7 ; point to TO var

STA ut2\_pl ; save pointer to TO var for compare

ADC #$FB ; point to STEP var

LDY #>LAB\_STAK ; point to stack page high byte

JSR LAB\_UFAC ; unpack memory (STEP value) into FAC1

TSX ; get stack pointer back

LDA LAB\_STAK+8,X ; get step sign

STA FAC1\_s ; save FAC1 sign (b7)

LDA Frnxtl ; get FOR variable pointer low byte

LDY Frnxth ; get FOR variable pointer high byte

JSR LAB\_246C ; add (FOR variable) to FAC1

JSR LAB\_PFAC ; pack FAC1 into (FOR variable)

LDY #>LAB\_STAK ; point to stack page high byte

JSR LAB\_27FA ; compare FAC1 with (Y,ut2\_pl) (TO value)

TSX ; get stack pointer back

CMP LAB\_STAK+8,X ; compare step sign

BEQ LAB\_1A9B ; branch if = (loop complete)

; loop back and do it all again

LDA LAB\_STAK+$0D,X ; get FOR line low byte

STA Clinel ; save current line low byte

LDA LAB\_STAK+$0E,X ; get FOR line high byte

STA Clineh ; save current line high byte

LDA LAB\_STAK+$10,X ; get BASIC execute pointer low byte

STA Bpntrl ; save BASIC execute pointer low byte

LDA LAB\_STAK+$0F,X ; get BASIC execute pointer high byte

STA Bpntrh ; save BASIC execute pointer high byte

LAB\_1A98

JMP LAB\_15C2 ; go do interpreter inner loop

; loop complete so carry on

LAB\_1A9B

TXA ; stack copy to A

ADC #$0F ; add $10 ($0F+carry) to dump FOR structure

TAX ; copy back to index

TXS ; copy to stack pointer

JSR LAB\_GBYT ; scan memory

CMP #',' ; compare with ","

BNE LAB\_1A98 ; branch if not "," (go do interpreter inner loop)

; was "," so another NEXT variable to do

JSR LAB\_IGBY ; else increment and scan memory

JSR LAB\_1A46 ; do NEXT (var)

; evaluate expression and check is numeric, else do type mismatch

LAB\_EVNM

JSR LAB\_EVEX ; evaluate expression

; check if source is numeric, else do type mismatch

LAB\_CTNM

CLC ; destination is numeric

.byte $24 ; makes next line BIT $38

; check if source is string, else do type mismatch

LAB\_CTST

SEC ; required type is string

; type match check, set C for string, clear C for numeric

LAB\_CKTM

BIT Dtypef ; test data type flag, $FF=string, $00=numeric

BMI LAB\_1ABA ; branch if data type is string

; else data type was numeric

BCS LAB\_1ABC ; if required type is string do type mismatch error

LAB\_1AB9

RTS

; data type was string, now check required type

LAB\_1ABA

BCS LAB\_1AB9 ; exit if required type is string

; else do type mismatch error

LAB\_1ABC

LDX #$18 ; error code $18 ("Type mismatch" error)

LAB\_1ABE

JMP LAB\_XERR ; do error #X, then warm start

; evaluate expression

LAB\_EVEX

LDX Bpntrl ; get BASIC execute pointer low byte

BNE LAB\_1AC7 ; skip next if not zero

DEC Bpntrh ; else decrement BASIC execute pointer high byte

LAB\_1AC7

DEC Bpntrl ; decrement BASIC execute pointer low byte

LAB\_EVEZ

LDA #$00 ; set null precedence (flag done)

LAB\_1ACC

PHA ; push precedence byte

LDA #$02 ; 2 bytes

JSR LAB\_1212 ; check room on stack for A bytes

JSR LAB\_GVAL ; get value from line

LDA #$00 ; clear A

STA comp\_f ; clear compare function flag

LAB\_1ADB

JSR LAB\_GBYT ; scan memory

LAB\_1ADE

SEC ; set carry for subtract

SBC #TK\_GT ; subtract token for > (lowest comparison function)

BCC LAB\_1AFA ; branch if < TK\_GT

CMP #$03 ; compare with ">" to "<" tokens

BCS LAB\_1AFA ; branch if >= TK\_SGN (highest evaluation function +1)

; was token for > = or < (A = 0, 1 or 2)

CMP #$01 ; compare with token for =

ROL ; \*2, b0 = carry (=1 if token was = or <)

; (A = 0, 3 or 5)

EOR #$01 ; toggle b0

; (A = 1, 2 or 4. 1 if >, 2 if =, 4 if <)

EOR comp\_f ; EOR with compare function flag bits

CMP comp\_f ; compare with compare function flag

BCC LAB\_1B53 ; if <(comp\_f) do syntax error then warm start

; was more than one <, = or >)

STA comp\_f ; save new compare function flag

JSR LAB\_IGBY ; increment and scan memory

JMP LAB\_1ADE ; go do next character

; token is < ">" or > "<" tokens

LAB\_1AFA

LDX comp\_f ; get compare function flag

BNE LAB\_1B2A ; branch if compare function

BCS LAB\_1B78 ; go do functions

; else was < TK\_GT so is operator or lower

ADC #TK\_GT-TK\_PLUS ; add # of operators (+, -, \*, /, ^, AND, OR or EOR)

BCC LAB\_1B78 ; branch if < + operator

; carry was set so token was +, -, \*, /, ^, AND, OR or EOR

BNE LAB\_1B0B ; branch if not + token

BIT Dtypef ; test data type flag, $FF=string, $00=numeric

BPL LAB\_1B0B ; branch if not string

; will only be $00 if type is string and token was +

JMP LAB\_224D ; add strings, string 1 is in descriptor des\_pl, string 2

; is in line, and return

LAB\_1B0B

STA ut1\_pl ; save it

ASL ; \*2

ADC ut1\_pl ; \*3

TAY ; copy to index

LAB\_1B13

PLA ; pull previous precedence

CMP LAB\_OPPT,Y ; compare with precedence byte

BCS LAB\_1B7D ; branch if A >=

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

LAB\_1B1C

PHA ; save precedence

LAB\_1B1D

JSR LAB\_1B43 ; get vector, execute function then continue evaluation

PLA ; restore precedence

LDY prstk ; get precedence stacked flag

BPL LAB\_1B3C ; branch if stacked values

TAX ; copy precedence (set flags)

BEQ LAB\_1B9D ; exit if done

BNE LAB\_1B86 ; else pop FAC2 and return, branch always

LAB\_1B2A

ROL Dtypef ; shift data type flag into Cb

TXA ; copy compare function flag

STA Dtypef ; clear data type flag, X is 0xxx xxxx

ROL ; shift data type into compare function byte b0

LDX Bpntrl ; get BASIC execute pointer low byte

BNE LAB\_1B34 ; branch if no underflow

DEC Bpntrh ; else decrement BASIC execute pointer high byte

LAB\_1B34

DEC Bpntrl ; decrement BASIC execute pointer low byte

TK\_LT\_PLUS = TK\_LT-TK\_PLUS

LDY #TK\_LT\_PLUS\*3 ; set offset to last operator entry

STA comp\_f ; save new compare function flag

BNE LAB\_1B13 ; branch always

LAB\_1B3C

CMP LAB\_OPPT,Y ;.compare with stacked function precedence

BCS LAB\_1B86 ; branch if A >=, pop FAC2 and return

BCC LAB\_1B1C ; branch always

;.get vector, execute function then continue evaluation

LAB\_1B43

LDA LAB\_OPPT+2,Y ; get function vector high byte

PHA ; onto stack

LDA LAB\_OPPT+1,Y ; get function vector low byte

PHA ; onto stack

; now push sign, round FAC1 and put on stack

JSR LAB\_1B5B ; function will return here, then the next RTS will call

; the function

LDA comp\_f ; get compare function flag

PHA ; push compare evaluation byte

LDA LAB\_OPPT,Y ; get precedence byte

JMP LAB\_1ACC ; continue evaluating expression

LAB\_1B53

JMP LAB\_SNER ; do syntax error then warm start

; push sign, round FAC1 and put on stack

LAB\_1B5B

PLA ; get return addr low byte

STA ut1\_pl ; save it

INC ut1\_pl ; increment it (was ret-1 pushed? yes!)

; note! no check is made on the high byte! if the calling

; routine assembles to a page edge then this all goes

; horribly wrong !!!

PLA ; get return addr high byte

STA ut1\_ph ; save it

LDA FAC1\_s ; get FAC1 sign (b7)

PHA ; push sign

; round FAC1 and put on stack

LAB\_1B66

JSR LAB\_27BA ; round FAC1

LDA FAC1\_3 ; get FAC1 mantissa3

PHA ; push on stack

LDA FAC1\_2 ; get FAC1 mantissa2

PHA ; push on stack

LDA FAC1\_1 ; get FAC1 mantissa1

PHA ; push on stack

LDA FAC1\_e ; get FAC1 exponent

PHA ; push on stack

JMP (ut1\_pl) ; return, sort of

; do functions

LAB\_1B78

LDY #$FF ; flag function

PLA ; pull precedence byte

LAB\_1B7B

BEQ LAB\_1B9D ; exit if done

LAB\_1B7D

CMP #$64 ; compare previous precedence with $64

BEQ LAB\_1B84 ; branch if was $64 (< function)

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

LAB\_1B84

STY prstk ; save precedence stacked flag

; pop FAC2 and return

LAB\_1B86

PLA ; pop byte

LSR ; shift out comparison evaluation lowest bit

STA Cflag ; save comparison evaluation flag

PLA ; pop exponent

STA FAC2\_e ; save FAC2 exponent

PLA ; pop mantissa1

STA FAC2\_1 ; save FAC2 mantissa1

PLA ; pop mantissa2

STA FAC2\_2 ; save FAC2 mantissa2

PLA ; pop mantissa3

STA FAC2\_3 ; save FAC2 mantissa3

PLA ; pop sign

STA FAC2\_s ; save FAC2 sign (b7)

EOR FAC1\_s ; EOR FAC1 sign (b7)

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

LAB\_1B9D

LDA FAC1\_e ; get FAC1 exponent

RTS

; print "..." string to string util area

LAB\_1BC1

LDA Bpntrl ; get BASIC execute pointer low byte

LDY Bpntrh ; get BASIC execute pointer high byte

ADC #$00 ; add carry to low byte

BCC LAB\_1BCA ; branch if no overflow

INY ; increment high byte

LAB\_1BCA

JSR LAB\_20AE ; print quote terminated string to Sutill/Sutilh

JMP LAB\_23F3 ; restore BASIC execute pointer from temp and return

; get value from line

LAB\_GVAL

JSR LAB\_IGBY ; increment and scan memory

BCS LAB\_1BAC ; branch if not numeric character

; else numeric string found (e.g. 123)

LAB\_1BA9

JMP LAB\_2887 ; get FAC1 from string and return

; get value from line .. continued

; wasn't a number so ..

LAB\_1BAC

TAX ; set the flags

BMI LAB\_1BD0 ; if -ve go test token values

; else it is either a string, number, variable or (<expr>)

CMP #'$' ; compare with "$"

BEQ LAB\_1BA9 ; branch if "$", hex number

CMP #'%' ; else compare with "%"

BEQ LAB\_1BA9 ; branch if "%", binary number

CMP #'.' ; compare with "."

BEQ LAB\_1BA9 ; if so get FAC1 from string and return (e.g. was .123)

; it wasn't any sort of number so ..

CMP #$22 ; compare with quote

BEQ LAB\_1BC1 ; branch if open quote

; wasn't any sort of number so ..

; evaluate expression within parentheses

CMP #'(' ; compare with "("

BNE LAB\_1C18 ; if not "(" get (var), return value in FAC1 and $ flag

LAB\_1BF7

JSR LAB\_EVEZ ; evaluate expression, no decrement

; all the 'scan for' routines return the character after the sought character

; scan for ")" , else do syntax error then warm start

LAB\_1BFB

LDA #$29 ; load A with ")"

; scan for CHR$(A) , else do syntax error then warm start

LAB\_SCCA

LDY #$00 ; clear index

CMP (Bpntrl),Y ; check next byte is = A

BNE LAB\_SNER ; if not do syntax error then warm start

JMP LAB\_IGBY ; increment and scan memory then return

; scan for "(" , else do syntax error then warm start

LAB\_1BFE

LDA #$28 ; load A with "("

BNE LAB\_SCCA ; scan for CHR$(A), else do syntax error then warm start

; (branch always)

; scan for "," , else do syntax error then warm start

LAB\_1C01

LDA #$2C ; load A with ","

BNE LAB\_SCCA ; scan for CHR$(A), else do syntax error then warm start

; (branch always)

; syntax error then warm start

LAB\_SNER

LDX #$02 ; error code $02 ("Syntax" error)

JMP LAB\_XERR ; do error #X, then warm start

; get value from line .. continued

; do tokens

LAB\_1BD0

CMP #TK\_MINUS ; compare with token for -

BEQ LAB\_1C11 ; branch if - token (do set-up for functions)

; wasn't -n so ..

CMP #TK\_PLUS ; compare with token for +

BEQ LAB\_GVAL ; branch if + token (+n = n so ignore leading +)

CMP #TK\_NOT ; compare with token for NOT

BNE LAB\_1BE7 ; branch if not token for NOT

; was NOT token

TK\_EQUAL\_PLUS = TK\_EQUAL-TK\_PLUS

LDY #TK\_EQUAL\_PLUS\*3 ; offset to NOT function

BNE LAB\_1C13 ; do set-up for function then execute (branch always)

; do = compare

LAB\_EQUAL

JSR LAB\_EVIR ; evaluate integer expression (no sign check)

LDA FAC1\_3 ; get FAC1 mantissa3

EOR #$FF ; invert it

TAY ; copy it

LDA FAC1\_2 ; get FAC1 mantissa2

EOR #$FF ; invert it

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; get value from line .. continued

; wasn't +, -, or NOT so ..

LAB\_1BE7

CMP #TK\_FN ; compare with token for FN

BNE LAB\_1BEE ; branch if not token for FN

JMP LAB\_201E ; go evaluate FNx

; get value from line .. continued

; wasn't +, -, NOT or FN so ..

LAB\_1BEE

SBC #TK\_SGN ; subtract with token for SGN

BCS LAB\_1C27 ; if a function token go do it

JMP LAB\_SNER ; else do syntax error

; set-up for functions

LAB\_1C11

TK\_GT\_PLUS = TK\_GT-TK\_PLUS

LDY #TK\_GT\_PLUS\*3 ; set offset from base to > operator

LAB\_1C13

PLA ; dump return address low byte

PLA ; dump return address high byte

JMP LAB\_1B1D ; execute function then continue evaluation

; variable name set-up

; get (var), return value in FAC\_1 and $ flag

LAB\_1C18

JSR LAB\_GVAR ; get (var) address

STA FAC1\_2 ; save address low byte in FAC1 mantissa2

STY FAC1\_3 ; save address high byte in FAC1 mantissa3

LDX Dtypef ; get data type flag, $FF=string, $00=numeric

BMI LAB\_1C25 ; if string then return (does RTS)

LAB\_1C24

JMP LAB\_UFAC ; unpack memory (AY) into FAC1

LAB\_1C25

RTS

; get value from line .. continued

; only functions left so ..

; set up function references

; new for V2.0+ this replaces a lot of IF .. THEN .. ELSEIF .. THEN .. that was needed

; to process function calls. now the function vector is computed and pushed on the stack

; and the preprocess offset is read. if the preprocess offset is non zero then the vector

; is calculated and the routine called, if not this routine just does RTS. whichever

; happens the RTS at the end of this routine, or the end of the preprocess routine, calls

; the function code

; this also removes some less than elegant code that was used to bypass type checking

; for functions that returned strings

LAB\_1C27

ASL ; \*2 (2 bytes per function address)

TAY ; copy to index

LDA LAB\_FTBM,Y ; get function jump vector high byte

PHA ; push functions jump vector high byte

LDA LAB\_FTBL,Y ; get function jump vector low byte

PHA ; push functions jump vector low byte

LDA LAB\_FTPM,Y ; get function pre process vector high byte

BEQ LAB\_1C56 ; skip pre process if null vector

PHA ; push functions pre process vector high byte

LDA LAB\_FTPL,Y ; get function pre process vector low byte

PHA ; push functions pre process vector low byte

LAB\_1C56

RTS ; do function, or pre process, call

; process string expression in parenthesis

LAB\_PPFS

JSR LAB\_1BF7 ; process expression in parenthesis

JMP LAB\_CTST ; check if source is string then do function,

; else do type mismatch

; process numeric expression in parenthesis

LAB\_PPFN

JSR LAB\_1BF7 ; process expression in parenthesis

JMP LAB\_CTNM ; check if source is numeric then do function,

; else do type mismatch

; set numeric data type and increment BASIC execute pointer

LAB\_PPBI

LSR Dtypef ; clear data type flag, $FF=string, $00=numeric

JMP LAB\_IGBY ; increment and scan memory then do function

; process string for LEFT$, RIGHT$ or MID$

LAB\_LRMS

JSR LAB\_EVEZ ; evaluate (should be string) expression

JSR LAB\_1C01 ; scan for ",", else do syntax error then warm start

JSR LAB\_CTST ; check if source is string, else do type mismatch

PLA ; get function jump vector low byte

TAX ; save functions jump vector low byte

PLA ; get function jump vector high byte

TAY ; save functions jump vector high byte

LDA des\_ph ; get descriptor pointer high byte

PHA ; push string pointer high byte

LDA des\_pl ; get descriptor pointer low byte

PHA ; push string pointer low byte

TYA ; get function jump vector high byte back

PHA ; save functions jump vector high byte

TXA ; get function jump vector low byte back

PHA ; save functions jump vector low byte

JSR LAB\_GTBY ; get byte parameter

TXA ; copy byte parameter to A

RTS ; go do function

; process numeric expression(s) for BIN$ or HEX$

LAB\_BHSS

JSR LAB\_EVEZ ; process expression

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

LDA FAC1\_e ; get FAC1 exponent

CMP #$98 ; compare with exponent = 2^24

BCS LAB\_BHER ; branch if n>=2^24 (is too big)

JSR LAB\_2831 ; convert FAC1 floating-to-fixed

LDX #$02 ; 3 bytes to do

LAB\_CFAC

LDA FAC1\_1,X ; get byte from FAC1

STA nums\_1,X ; save byte to temp

DEX ; decrement index

BPL LAB\_CFAC ; copy FAC1 mantissa to temp

JSR LAB\_GBYT ; get next BASIC byte

LDX #$00 ; set default to no leading "0"s

CMP #')' ; compare with close bracket

BEQ LAB\_1C54 ; if ")" go do rest of function

JSR LAB\_SCGB ; scan for "," and get byte

JSR LAB\_GBYT ; get last byte back

CMP #')' ; is next character )

BNE LAB\_BHER ; if not ")" go do error

LAB\_1C54

RTS ; else do function

LAB\_BHER

JMP LAB\_FCER ; do function call error then warm start

; perform EOR

; added operator format is the same as AND or OR, precedence is the same as OR

; this bit worked first time but it took a while to sort out the operator table

; pointers and offsets afterwards!

LAB\_EOR

JSR GetFirst ; get first integer expression (no sign check)

EOR XOAw\_l ; EOR with expression 1 low byte

TAY ; save in Y

LDA FAC1\_2 ; get FAC1 mantissa2

EOR XOAw\_h ; EOR with expression 1 high byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform OR

LAB\_OR

JSR GetFirst ; get first integer expression (no sign check)

ORA XOAw\_l ; OR with expression 1 low byte

TAY ; save in Y

LDA FAC1\_2 ; get FAC1 mantissa2

ORA XOAw\_h ; OR with expression 1 high byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform AND

LAB\_AND

JSR GetFirst ; get first integer expression (no sign check)

AND XOAw\_l ; AND with expression 1 low byte

TAY ; save in Y

LDA FAC1\_2 ; get FAC1 mantissa2

AND XOAw\_h ; AND with expression 1 high byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; get first value for OR, AND or EOR

GetFirst

JSR LAB\_EVIR ; evaluate integer expression (no sign check)

LDA FAC1\_2 ; get FAC1 mantissa2

STA XOAw\_h ; save it

LDA FAC1\_3 ; get FAC1 mantissa3

STA XOAw\_l ; save it

JSR LAB\_279B ; copy FAC2 to FAC1 (get 2nd value in expression)

JSR LAB\_EVIR ; evaluate integer expression (no sign check)

LDA FAC1\_3 ; get FAC1 mantissa3

LAB\_1C95

RTS

; perform comparisons

; do < compare

LAB\_LTHAN

JSR LAB\_CKTM ; type match check, set C for string

BCS LAB\_1CAE ; branch if string

; do numeric < compare

LDA FAC2\_s ; get FAC2 sign (b7)

ORA #$7F ; set all non sign bits

AND FAC2\_1 ; and FAC2 mantissa1 (AND in sign bit)

STA FAC2\_1 ; save FAC2 mantissa1

LDA #<FAC2\_e ; set pointer low byte to FAC2

LDY #>FAC2\_e ; set pointer high byte to FAC2

JSR LAB\_27F8 ; compare FAC1 with FAC2 (AY)

TAX ; copy result

JMP LAB\_1CE1 ; go evaluate result

; do string < compare

LAB\_1CAE

LSR Dtypef ; clear data type flag, $FF=string, $00=numeric

DEC comp\_f ; clear < bit in compare function flag

JSR LAB\_22B6 ; pop string off descriptor stack, or from top of string

; space returns with A = length, X=pointer low byte,

; Y=pointer high byte

STA str\_ln ; save length

STX str\_pl ; save string pointer low byte

STY str\_ph ; save string pointer high byte

LDA FAC2\_2 ; get descriptor pointer low byte

LDY FAC2\_3 ; get descriptor pointer high byte

JSR LAB\_22BA ; pop (YA) descriptor off stack or from top of string space

; returns with A = length, X=pointer low byte,

; Y=pointer high byte

STX FAC2\_2 ; save string pointer low byte

STY FAC2\_3 ; save string pointer high byte

TAX ; copy length

SEC ; set carry for subtract

SBC str\_ln ; subtract string 1 length

BEQ LAB\_1CD6 ; branch if str 1 length = string 2 length

LDA #$01 ; set str 1 length > string 2 length

BCC LAB\_1CD6 ; branch if so

LDX str\_ln ; get string 1 length

LDA #$FF ; set str 1 length < string 2 length

LAB\_1CD6

STA FAC1\_s ; save length compare

LDY #$FF ; set index

INX ; adjust for loop

LAB\_1CDB

INY ; increment index

DEX ; decrement count

BNE LAB\_1CE6 ; branch if still bytes to do

LDX FAC1\_s ; get length compare back

LAB\_1CE1

BMI LAB\_1CF2 ; branch if str 1 < str 2

CLC ; flag str 1 <= str 2

BCC LAB\_1CF2 ; go evaluate result

LAB\_1CE6

LDA (FAC2\_2),Y ; get string 2 byte

CMP (FAC1\_1),Y ; compare with string 1 byte

BEQ LAB\_1CDB ; loop if bytes =

LDX #$FF ; set str 1 < string 2

BCS LAB\_1CF2 ; branch if so

LDX #$01 ; set str 1 > string 2

LAB\_1CF2

INX ; x = 0, 1 or 2

TXA ; copy to A

ROL ; \*2 (1, 2 or 4)

AND Cflag ; AND with comparison evaluation flag

BEQ LAB\_1CFB ; branch if 0 (compare is false)

LDA #$FF ; else set result true

LAB\_1CFB

JMP LAB\_27DB ; save A as integer byte and return

LAB\_1CFE

JSR LAB\_1C01 ; scan for ",", else do syntax error then warm start

; perform DIM

LAB\_DIM

TAX ; copy "DIM" flag to X

JSR LAB\_1D10 ; search for variable

JSR LAB\_GBYT ; scan memory

BNE LAB\_1CFE ; scan for "," and loop if not null

RTS

; perform << (left shift)

LAB\_LSHIFT

JSR GetPair ; get integer expression and byte (no sign check)

LDA FAC1\_2 ; get expression high byte

LDX TempB ; get shift count

BEQ NoShift ; branch if zero

CPX #$10 ; compare bit count with 16d

BCS TooBig ; branch if >=

Ls\_loop

ASL FAC1\_3 ; shift low byte

ROL ; shift high byte

DEX ; decrement bit count

BNE Ls\_loop ; loop if shift not complete

LDY FAC1\_3 ; get expression low byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform >> (right shift)

LAB\_RSHIFT

JSR GetPair ; get integer expression and byte (no sign check)

LDA FAC1\_2 ; get expression high byte

LDX TempB ; get shift count

BEQ NoShift ; branch if zero

CPX #$10 ; compare bit count with 16d

BCS TooBig ; branch if >=

Rs\_loop

LSR ; shift high byte

ROR FAC1\_3 ; shift low byte

DEX ; decrement bit count

BNE Rs\_loop ; loop if shift not complete

NoShift

LDY FAC1\_3 ; get expression low byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

TooBig

LDA #$00 ; clear high byte

TAY ; copy to low byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

GetPair

JSR LAB\_EVBY ; evaluate byte expression, result in X

STX TempB ; save it

JSR LAB\_279B ; copy FAC2 to FAC1 (get 2nd value in expression)

JMP LAB\_EVIR ; evaluate integer expression (no sign check)

; search for variable

; return pointer to variable in Cvaral/Cvarah

LAB\_GVAR

LDX #$00 ; set DIM flag = $00

JSR LAB\_GBYT ; scan memory (1st character)

LAB\_1D10

STX Defdim ; save DIM flag

LAB\_1D12

STA Varnm1 ; save 1st character

AND #$7F ; clear FN flag bit

JSR LAB\_CASC ; check byte, return C=0 if<"A" or >"Z"

BCS LAB\_1D1F ; branch if ok

JMP LAB\_SNER ; else syntax error then warm start

; was variable name so ..

LAB\_1D1F

LDX #$00 ; clear 2nd character temp

STX Dtypef ; clear data type flag, $FF=string, $00=numeric

JSR LAB\_IGBY ; increment and scan memory (2nd character)

BCC LAB\_1D2D ; branch if character = "0"-"9" (ok)

; 2nd character wasn't "0" to "9" so ..

JSR LAB\_CASC ; check byte, return C=0 if<"A" or >"Z"

BCC LAB\_1D38 ; branch if <"A" or >"Z" (go check if string)

LAB\_1D2D

TAX ; copy 2nd character

; ignore further (valid) characters in the variable name

LAB\_1D2E

JSR LAB\_IGBY ; increment and scan memory (3rd character)

BCC LAB\_1D2E ; loop if character = "0"-"9" (ignore)

JSR LAB\_CASC ; check byte, return C=0 if<"A" or >"Z"

BCS LAB\_1D2E ; loop if character = "A"-"Z" (ignore)

; check if string variable

LAB\_1D38

CMP #'$' ; compare with "$"

BNE LAB\_1D47 ; branch if not string

; to introduce a new variable type (% suffix for integers say) then this branch

; will need to go to that check and then that branch, if it fails, go to LAB\_1D47

; type is string

LDA #$FF ; set data type = string

STA Dtypef ; set data type flag, $FF=string, $00=numeric

TXA ; get 2nd character back

ORA #$80 ; set top bit (indicate string var)

TAX ; copy back to 2nd character temp

JSR LAB\_IGBY ; increment and scan memory

; after we have determined the variable type we need to come back here to determine

; if it's an array of type. this would plug in a%(b[,c[,d]])) integer arrays nicely

LAB\_1D47 ; gets here with character after var name in A

STX Varnm2 ; save 2nd character

ORA Sufnxf ; or with subscript/FNX flag (or FN name)

CMP #'(' ; compare with "("

BNE LAB\_1D53 ; branch if not "("

JMP LAB\_1E17 ; go find, or make, array

; either find or create var

; var name (1st two characters only!) is in Varnm1,Varnm2

; variable name wasn't var(... so look for plain var

LAB\_1D53

LDA #$00 ; clear A

STA Sufnxf ; clear subscript/FNX flag

LDA Svarl ; get start of vars low byte

LDX Svarh ; get start of vars high byte

LDY #$00 ; clear index

LAB\_1D5D

STX Vrschh ; save search address high byte

LAB\_1D5F

STA Vrschl ; save search address low byte

CPX Sarryh ; compare high address with var space end

BNE LAB\_1D69 ; skip next compare if <>

; high addresses were = so compare low addresses

CMP Sarryl ; compare low address with var space end

BEQ LAB\_1D8B ; if not found go make new var

LAB\_1D69

LDA Varnm1 ; get 1st character of var to find

CMP (Vrschl),Y ; compare with variable name 1st character

BNE LAB\_1D77 ; branch if no match

; 1st characters match so compare 2nd characters

LDA Varnm2 ; get 2nd character of var to find

INY ; index to point to variable name 2nd character

CMP (Vrschl),Y ; compare with variable name 2nd character

BEQ LAB\_1DD7 ; branch if match (found var)

DEY ; else decrement index (now = $00)

LAB\_1D77

CLC ; clear carry for add

LDA Vrschl ; get search address low byte

ADC #$06 ; +6 (offset to next var name)

BCC LAB\_1D5F ; loop if no overflow to high byte

INX ; else increment high byte

BNE LAB\_1D5D ; loop always (RAM doesn't extend to $FFFF !)

; check byte, return C=0 if<"A" or >"Z" or "a" to "z"

LAB\_CASC

CMP #'a' ; compare with "a"

BCS LAB\_1D83 ; go check <"z"+1

; check byte, return C=0 if<"A" or >"Z"

LAB\_1D82

CMP #'A' ; compare with "A"

BCC LAB\_1D8A ; exit if less

; carry is set

SBC #$5B ; subtract "Z"+1

SEC ; set carry

SBC #$A5 ; subtract $A5 (restore byte)

; carry clear if byte>$5A

LAB\_1D8A

RTS

LAB\_1D83

SBC #$7B ; subtract "z"+1

SEC ; set carry

SBC #$85 ; subtract $85 (restore byte)

; carry clear if byte>$7A

RTS

; reached end of variable mem without match

; .. so create new variable

LAB\_1D8B

PLA ; pop return address low byte

PHA ; push return address low byte

LAB\_1C18p2 = LAB\_1C18+2

CMP #<LAB\_1C18p2 ; compare with expected calling routine return low byte

BNE LAB\_1D98 ; if not get (var) go create new var

; This will only drop through if the call was from LAB\_1C18 and is only called

; from there if it is searching for a variable from the RHS of a LET a=b statement

; it prevents the creation of variables not assigned a value.

; value returned by this is either numeric zero (exponent byte is $00) or null string

; (descriptor length byte is $00). in fact a pointer to any $00 byte would have done.

; doing this saves 6 bytes of variable memory and 168 machine cycles of time

; this is where you would put the undefined variable error call e.g.

; ; variable doesn't exist so flag error

; LDX #$24 ; error code $24 ("undefined variable" error)

; JMP LAB\_XERR ; do error #X then warm start

; the above code has been tested and works a treat! (it replaces the three code lines

; below)

; else return dummy null value

LDA #<LAB\_1D96 ; low byte point to $00,$00

; (uses part of misc constants table)

LDY #>LAB\_1D96 ; high byte point to $00,$00

RTS

; create new numeric variable

LAB\_1D98

LDA Sarryl ; get var mem end low byte

LDY Sarryh ; get var mem end high byte

STA Ostrtl ; save old block start low byte

STY Ostrth ; save old block start high byte

LDA Earryl ; get array mem end low byte

LDY Earryh ; get array mem end high byte

STA Obendl ; save old block end low byte

STY Obendh ; save old block end high byte

CLC ; clear carry for add

ADC #$06 ; +6 (space for one var)

BCC LAB\_1DAE ; branch if no overflow to high byte

INY ; else increment high byte

LAB\_1DAE

STA Nbendl ; set new block end low byte

STY Nbendh ; set new block end high byte

JSR LAB\_11CF ; open up space in memory

LDA Nbendl ; get new start low byte

LDY Nbendh ; get new start high byte (-$100)

INY ; correct high byte

STA Sarryl ; save new var mem end low byte

STY Sarryh ; save new var mem end high byte

LDY #$00 ; clear index

LDA Varnm1 ; get var name 1st character

STA (Vrschl),Y ; save var name 1st character

INY ; increment index

LDA Varnm2 ; get var name 2nd character

STA (Vrschl),Y ; save var name 2nd character

LDA #$00 ; clear A

INY ; increment index

STA (Vrschl),Y ; initialise var byte

INY ; increment index

STA (Vrschl),Y ; initialise var byte

INY ; increment index

STA (Vrschl),Y ; initialise var byte

INY ; increment index

STA (Vrschl),Y ; initialise var byte

; found a match for var ((Vrschl) = ptr)

LAB\_1DD7

LDA Vrschl ; get var address low byte

CLC ; clear carry for add

ADC #$02 ; +2 (offset past var name bytes)

LDY Vrschh ; get var address high byte

BCC LAB\_1DE1 ; branch if no overflow from add

INY ; else increment high byte

LAB\_1DE1

STA Cvaral ; save current var address low byte

STY Cvarah ; save current var address high byte

RTS

; set-up array pointer (Adatal/h) to first element in array

; set Adatal,Adatah to Astrtl,Astrth+2\*Dimcnt+#$05

LAB\_1DE6

LDA Dimcnt ; get # of dimensions (1, 2 or 3)

ASL ; \*2 (also clears the carry !)

ADC #$05 ; +5 (result is 7, 9 or 11 here)

ADC Astrtl ; add array start pointer low byte

LDY Astrth ; get array pointer high byte

BCC LAB\_1DF2 ; branch if no overflow

INY ; else increment high byte

LAB\_1DF2

STA Adatal ; save array data pointer low byte

STY Adatah ; save array data pointer high byte

RTS

; evaluate integer expression

LAB\_EVIN

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

; evaluate integer expression (no check)

LAB\_EVPI

LDA FAC1\_s ; get FAC1 sign (b7)

BMI LAB\_1E12 ; do function call error if -ve

; evaluate integer expression (no sign check)

LAB\_EVIR

LDA FAC1\_e ; get FAC1 exponent

CMP #$90 ; compare with exponent = 2^16 (n>2^15)

BCC LAB\_1E14 ; branch if n<2^16 (is ok)

LDA #<LAB\_1DF7 ; set pointer low byte to -32768

LDY #>LAB\_1DF7 ; set pointer high byte to -32768

JSR LAB\_27F8 ; compare FAC1 with (AY)

LAB\_1E12

BNE LAB\_FCER ; if <> do function call error then warm start

LAB\_1E14

JMP LAB\_2831 ; convert FAC1 floating-to-fixed and return

; find or make array

LAB\_1E17

LDA Defdim ; get DIM flag

PHA ; push it

LDA Dtypef ; get data type flag, $FF=string, $00=numeric

PHA ; push it

LDY #$00 ; clear dimensions count

; now get the array dimension(s) and stack it (them) before the data type and DIM flag

LAB\_1E1F

TYA ; copy dimensions count

PHA ; save it

LDA Varnm2 ; get array name 2nd byte

PHA ; save it

LDA Varnm1 ; get array name 1st byte

PHA ; save it

JSR LAB\_EVIN ; evaluate integer expression

PLA ; pull array name 1st byte

STA Varnm1 ; restore array name 1st byte

PLA ; pull array name 2nd byte

STA Varnm2 ; restore array name 2nd byte

PLA ; pull dimensions count

TAY ; restore it

TSX ; copy stack pointer

LDA LAB\_STAK+2,X ; get DIM flag

PHA ; push it

LDA LAB\_STAK+1,X ; get data type flag

PHA ; push it

LDA FAC1\_2 ; get this dimension size high byte

STA LAB\_STAK+2,X ; stack before flag bytes

LDA FAC1\_3 ; get this dimension size low byte

STA LAB\_STAK+1,X ; stack before flag bytes

INY ; increment dimensions count

JSR LAB\_GBYT ; scan memory

CMP #',' ; compare with ","

BEQ LAB\_1E1F ; if found go do next dimension

STY Dimcnt ; store dimensions count

JSR LAB\_1BFB ; scan for ")" , else do syntax error then warm start

PLA ; pull data type flag

STA Dtypef ; restore data type flag, $FF=string, $00=numeric

PLA ; pull DIM flag

STA Defdim ; restore DIM flag

LDX Sarryl ; get array mem start low byte

LDA Sarryh ; get array mem start high byte

; now check to see if we are at the end of array memory (we would be if there were

; no arrays).

LAB\_1E5C

STX Astrtl ; save as array start pointer low byte

STA Astrth ; save as array start pointer high byte

CMP Earryh ; compare with array mem end high byte

BNE LAB\_1E68 ; branch if not reached array mem end

CPX Earryl ; else compare with array mem end low byte

BEQ LAB\_1EA1 ; go build array if not found

; search for array

LAB\_1E68

LDY #$00 ; clear index

LDA (Astrtl),Y ; get array name first byte

INY ; increment index to second name byte

CMP Varnm1 ; compare with this array name first byte

BNE LAB\_1E77 ; branch if no match

LDA Varnm2 ; else get this array name second byte

CMP (Astrtl),Y ; compare with array name second byte

BEQ LAB\_1E8D ; array found so branch

; no match

LAB\_1E77

INY ; increment index

LDA (Astrtl),Y ; get array size low byte

CLC ; clear carry for add

ADC Astrtl ; add array start pointer low byte

TAX ; copy low byte to X

INY ; increment index

LDA (Astrtl),Y ; get array size high byte

ADC Astrth ; add array mem pointer high byte

BCC LAB\_1E5C ; if no overflow go check next array

; do array bounds error

LAB\_1E85

LDX #$10 ; error code $10 ("Array bounds" error)

.byte $2C ; makes next bit BIT LAB\_08A2

; do function call error

LAB\_FCER

LDX #$08 ; error code $08 ("Function call" error)

LAB\_1E8A

JMP LAB\_XERR ; do error #X, then warm start

; found array, are we trying to dimension it?

LAB\_1E8D

LDX #$12 ; set error $12 ("Double dimension" error)

LDA Defdim ; get DIM flag

BNE LAB\_1E8A ; if we are trying to dimension it do error #X, then warm

; start

; found the array and we're not dimensioning it so we must find an element in it

JSR LAB\_1DE6 ; set-up array pointer (Adatal/h) to first element in array

; (Astrtl,Astrth points to start of array)

LDA Dimcnt ; get dimensions count

LDY #$04 ; set index to array's # of dimensions

CMP (Astrtl),Y ; compare with no of dimensions

BNE LAB\_1E85 ; if wrong do array bounds error, could do "Wrong

; dimensions" error here .. if we want a different

; error message

JMP LAB\_1F28 ; found array so go get element

; (could jump to LAB\_1F28 as all LAB\_1F24 does is take

; Dimcnt and save it at (Astrtl),Y which is already the

; same or we would have taken the BNE)

; array not found, so build it

LAB\_1EA1

JSR LAB\_1DE6 ; set-up array pointer (Adatal/h) to first element in array

; (Astrtl,Astrth points to start of array)

JSR LAB\_121F ; check available memory, "Out of memory" error if no room

; addr to check is in AY (low/high)

LDY #$00 ; clear Y (don't need to clear A)

STY Aspth ; clear array data size high byte

LDA Varnm1 ; get variable name 1st byte

STA (Astrtl),Y ; save array name 1st byte

INY ; increment index

LDA Varnm2 ; get variable name 2nd byte

STA (Astrtl),Y ; save array name 2nd byte

LDA Dimcnt ; get dimensions count

LDY #$04 ; index to dimension count

STY Asptl ; set array data size low byte (four bytes per element)

STA (Astrtl),Y ; set array's dimensions count

; now calculate the size of the data space for the array

CLC ; clear carry for add (clear on subsequent loops)

LAB\_1EC0

LDX #$0B ; set default dimension value low byte

LDA #$00 ; set default dimension value high byte

BIT Defdim ; test default DIM flag

BVC LAB\_1ED0 ; branch if b6 of Defdim is clear

PLA ; else pull dimension value low byte

ADC #$01 ; +1 (allow for zeroeth element)

TAX ; copy low byte to X

PLA ; pull dimension value high byte

ADC #$00 ; add carry from low byte

LAB\_1ED0

INY ; index to dimension value high byte

STA (Astrtl),Y ; save dimension value high byte

INY ; index to dimension value high byte

TXA ; get dimension value low byte

STA (Astrtl),Y ; save dimension value low byte

JSR LAB\_1F7C ; does XY = (Astrtl),Y \* (Asptl)

STX Asptl ; save array data size low byte

STA Aspth ; save array data size high byte

LDY ut1\_pl ; restore index (saved by subroutine)

DEC Dimcnt ; decrement dimensions count

BNE LAB\_1EC0 ; loop while not = 0

ADC Adatah ; add size high byte to first element high byte

; (carry is always clear here)

BCS LAB\_1F45 ; if overflow go do "Out of memory" error

STA Adatah ; save end of array high byte

TAY ; copy end high byte to Y

TXA ; get array size low byte

ADC Adatal ; add array start low byte

BCC LAB\_1EF3 ; branch if no carry

INY ; else increment end of array high byte

BEQ LAB\_1F45 ; if overflow go do "Out of memory" error

; set-up mostly complete, now zero the array

LAB\_1EF3

JSR LAB\_121F ; check available memory, "Out of memory" error if no room

; addr to check is in AY (low/high)

STA Earryl ; save array mem end low byte

STY Earryh ; save array mem end high byte

LDA #$00 ; clear byte for array clear

INC Aspth ; increment array size high byte (now block count)

LDY Asptl ; get array size low byte (now index to block)

BEQ LAB\_1F07 ; branch if low byte = $00

LAB\_1F02

DEY ; decrement index (do 0 to n-1)

STA (Adatal),Y ; zero byte

BNE LAB\_1F02 ; loop until this block done

LAB\_1F07

DEC Adatah ; decrement array pointer high byte

DEC Aspth ; decrement block count high byte

BNE LAB\_1F02 ; loop until all blocks done

INC Adatah ; correct for last loop

SEC ; set carry for subtract

LDY #$02 ; index to array size low byte

LDA Earryl ; get array mem end low byte

SBC Astrtl ; subtract array start low byte

STA (Astrtl),Y ; save array size low byte

INY ; index to array size high byte

LDA Earryh ; get array mem end high byte

SBC Astrth ; subtract array start high byte

STA (Astrtl),Y ; save array size high byte

LDA Defdim ; get default DIM flag

BNE LAB\_1F7B ; exit (RET) if this was a DIM command

; else, find element

INY ; index to # of dimensions

LAB\_1F24

LDA (Astrtl),Y ; get array's dimension count

STA Dimcnt ; save it

; we have found, or built, the array. now we need to find the element

LAB\_1F28

LDA #$00 ; clear byte

STA Asptl ; clear array data pointer low byte

LAB\_1F2C

STA Aspth ; save array data pointer high byte

INY ; increment index (point to array bound high byte)

PLA ; pull array index low byte

TAX ; copy to X

STA FAC1\_2 ; save index low byte to FAC1 mantissa2

PLA ; pull array index high byte

STA FAC1\_3 ; save index high byte to FAC1 mantissa3

CMP (Astrtl),Y ; compare with array bound high byte

BCC LAB\_1F48 ; branch if within bounds

BNE LAB\_1F42 ; if outside bounds do array bounds error

; else high byte was = so test low bytes

INY ; index to array bound low byte

TXA ; get array index low byte

CMP (Astrtl),Y ; compare with array bound low byte

BCC LAB\_1F49 ; branch if within bounds

LAB\_1F42

JMP LAB\_1E85 ; else do array bounds error

LAB\_1F45

JMP LAB\_OMER ; do "Out of memory" error then warm start

LAB\_1F48

INY ; index to array bound low byte

LAB\_1F49

LDA Aspth ; get array data pointer high byte

ORA Asptl ; OR with array data pointer low byte

BEQ LAB\_1F5A ; branch if array data pointer = null (skip multiply)

JSR LAB\_1F7C ; does XY = (Astrtl),Y \* (Asptl)

TXA ; get result low byte

ADC FAC1\_2 ; add index low byte from FAC1 mantissa2

TAX ; save result low byte

TYA ; get result high byte

LDY ut1\_pl ; restore index

LAB\_1F5A

ADC FAC1\_3 ; add index high byte from FAC1 mantissa3

STX Asptl ; save array data pointer low byte

DEC Dimcnt ; decrement dimensions count

BNE LAB\_1F2C ; loop if dimensions still to do

ASL Asptl ; array data pointer low byte \* 2

ROL ; array data pointer high byte \* 2

ASL Asptl ; array data pointer low byte \* 4

ROL ; array data pointer high byte \* 4

TAY ; copy high byte

LDA Asptl ; get low byte

ADC Adatal ; add array data start pointer low byte

STA Cvaral ; save as current var address low byte

TYA ; get high byte back

ADC Adatah ; add array data start pointer high byte

STA Cvarah ; save as current var address high byte

TAY ; copy high byte to Y

LDA Cvaral ; get current var address low byte

LAB\_1F7B

RTS

; does XY = (Astrtl),Y \* (Asptl)

LAB\_1F7C

STY ut1\_pl ; save index

LDA (Astrtl),Y ; get dimension size low byte

STA dims\_l ; save dimension size low byte

DEY ; decrement index

LDA (Astrtl),Y ; get dimension size high byte

STA dims\_h ; save dimension size high byte

LDA #$10 ; count = $10 (16 bit multiply)

STA numbit ; save bit count

LDX #$00 ; clear result low byte

LDY #$00 ; clear result high byte

LAB\_1F8F

TXA ; get result low byte

ASL ; \*2

TAX ; save result low byte

TYA ; get result high byte

ROL ; \*2

TAY ; save result high byte

BCS LAB\_1F45 ; if overflow go do "Out of memory" error

ASL Asptl ; shift multiplier low byte

ROL Aspth ; shift multiplier high byte

BCC LAB\_1FA8 ; skip add if no carry

CLC ; else clear carry for add

TXA ; get result low byte

ADC dims\_l ; add dimension size low byte

TAX ; save result low byte

TYA ; get result high byte

ADC dims\_h ; add dimension size high byte

TAY ; save result high byte

BCS LAB\_1F45 ; if overflow go do "Out of memory" error

LAB\_1FA8

DEC numbit ; decrement bit count

BNE LAB\_1F8F ; loop until all done

RTS

; perform FRE()

LAB\_FRE

LDA Dtypef ; get data type flag, $FF=string, $00=numeric

BPL LAB\_1FB4 ; branch if numeric

JSR LAB\_22B6 ; pop string off descriptor stack, or from top of string

; space returns with A = length, X=$71=pointer low byte,

; Y=$72=pointer high byte

; FRE(n) was numeric so do this

LAB\_1FB4

JSR LAB\_GARB ; go do garbage collection

SEC ; set carry for subtract

LDA Sstorl ; get bottom of string space low byte

SBC Earryl ; subtract array mem end low byte

TAY ; copy result to Y

LDA Sstorh ; get bottom of string space high byte

SBC Earryh ; subtract array mem end high byte

; save and convert integer AY to FAC1

LAB\_AYFC

LSR Dtypef ; clear data type flag, $FF=string, $00=numeric

STA FAC1\_1 ; save FAC1 mantissa1

STY FAC1\_2 ; save FAC1 mantissa2

LDX #$90 ; set exponent=2^16 (integer)

JMP LAB\_27E3 ; set exp=X, clear FAC1\_3, normalise and return

; perform POS()

LAB\_POS

LDY TPos ; get terminal position

; convert Y to byte in FAC1

LAB\_1FD0

LDA #$00 ; clear high byte

BEQ LAB\_AYFC ; always save and convert integer AY to FAC1 and return

; check not Direct (used by DEF and INPUT)

LAB\_CKRN

LDX Clineh ; get current line high byte

INX ; increment it

BNE LAB\_1F7B ; return if can continue not direct mode

; else do illegal direct error

LAB\_1FD9

LDX #$16 ; error code $16 ("Illegal direct" error)

LAB\_1FDB

JMP LAB\_XERR ; go do error #X, then warm start

; perform DEF

LAB\_DEF

JSR LAB\_200B ; check FNx syntax

STA func\_l ; save function pointer low byte

STY func\_h ; save function pointer high byte

JSR LAB\_CKRN ; check not Direct (back here if ok)

JSR LAB\_1BFE ; scan for "(" , else do syntax error then warm start

LDA #$80 ; set flag for FNx

STA Sufnxf ; save subscript/FNx flag

JSR LAB\_GVAR ; get (var) address

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

JSR LAB\_1BFB ; scan for ")" , else do syntax error then warm start

LDA #TK\_EQUAL ; get = token

JSR LAB\_SCCA ; scan for CHR$(A), else do syntax error then warm start

LDA Cvarah ; get current var address high byte

PHA ; push it

LDA Cvaral ; get current var address low byte

PHA ; push it

LDA Bpntrh ; get BASIC execute pointer high byte

PHA ; push it

LDA Bpntrl ; get BASIC execute pointer low byte

PHA ; push it

JSR LAB\_DATA ; go perform DATA

JMP LAB\_207A ; put execute pointer and variable pointer into function

; and return

; check FNx syntax

LAB\_200B

LDA #TK\_FN ; get FN" token

JSR LAB\_SCCA ; scan for CHR$(A) , else do syntax error then warm start

; return character after A

ORA #$80 ; set FN flag bit

STA Sufnxf ; save FN flag so array variable test fails

JSR LAB\_1D12 ; search for FN variable

JMP LAB\_CTNM ; check if source is numeric and return, else do type

; mismatch

; Evaluate FNx

LAB\_201E

JSR LAB\_200B ; check FNx syntax

PHA ; push function pointer low byte

TYA ; copy function pointer high byte

PHA ; push function pointer high byte

JSR LAB\_1BFE ; scan for "(", else do syntax error then warm start

JSR LAB\_EVEX ; evaluate expression

JSR LAB\_1BFB ; scan for ")", else do syntax error then warm start

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

PLA ; pop function pointer high byte

STA func\_h ; restore it

PLA ; pop function pointer low byte

STA func\_l ; restore it

LDX #$20 ; error code $20 ("Undefined function" error)

LDY #$03 ; index to variable pointer high byte

LDA (func\_l),Y ; get variable pointer high byte

BEQ LAB\_1FDB ; if zero go do undefined function error

STA Cvarah ; save variable address high byte

DEY ; index to variable address low byte

LDA (func\_l),Y ; get variable address low byte

STA Cvaral ; save variable address low byte

TAX ; copy address low byte

; now stack the function variable value before use

INY ; index to mantissa\_3

LAB\_2043

LDA (Cvaral),Y ; get byte from variable

PHA ; stack it

DEY ; decrement index

BPL LAB\_2043 ; loop until variable stacked

LDY Cvarah ; get variable address high byte

JSR LAB\_2778 ; pack FAC1 (function expression value) into (XY)

; (function variable), return Y=0, always

LDA Bpntrh ; get BASIC execute pointer high byte

PHA ; push it

LDA Bpntrl ; get BASIC execute pointer low byte

PHA ; push it

LDA (func\_l),Y ; get function execute pointer low byte

STA Bpntrl ; save as BASIC execute pointer low byte

INY ; index to high byte

LDA (func\_l),Y ; get function execute pointer high byte

STA Bpntrh ; save as BASIC execute pointer high byte

LDA Cvarah ; get variable address high byte

PHA ; push it

LDA Cvaral ; get variable address low byte

PHA ; push it

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

PLA ; pull variable address low byte

STA func\_l ; save variable address low byte

PLA ; pull variable address high byte

STA func\_h ; save variable address high byte

JSR LAB\_GBYT ; scan memory

BEQ LAB\_2074 ; branch if null (should be [EOL] marker)

JMP LAB\_SNER ; else syntax error then warm start

; restore Bpntrl,Bpntrh and function variable from stack

LAB\_2074

PLA ; pull BASIC execute pointer low byte

STA Bpntrl ; restore BASIC execute pointer low byte

PLA ; pull BASIC execute pointer high byte

STA Bpntrh ; restore BASIC execute pointer high byte

; put execute pointer and variable pointer into function

LAB\_207A

LDY #$00 ; clear index

PLA ; pull BASIC execute pointer low byte

STA (func\_l),Y ; save to function

INY ; increment index

PLA ; pull BASIC execute pointer high byte

STA (func\_l),Y ; save to function

INY ; increment index

PLA ; pull current var address low byte

STA (func\_l),Y ; save to function

INY ; increment index

PLA ; pull current var address high byte

STA (func\_l),Y ; save to function

RTS

; perform STR$()

LAB\_STRS

JSR LAB\_CTNM ; check if source is numeric, else do type mismatch

JSR LAB\_296E ; convert FAC1 to string

LDA #<Decssp1 ; set result string low pointer

LDY #>Decssp1 ; set result string high pointer

BEQ LAB\_20AE ; print null terminated string to Sutill/Sutilh

; Do string vector

; copy des\_pl/h to des\_2l/h and make string space A bytes long

LAB\_209C

LDX des\_pl ; get descriptor pointer low byte

LDY des\_ph ; get descriptor pointer high byte

STX des\_2l ; save descriptor pointer low byte

STY des\_2h ; save descriptor pointer high byte

; make string space A bytes long

; A=length, X=Sutill=ptr low byte, Y=Sutilh=ptr high byte

LAB\_MSSP

JSR LAB\_2115 ; make space in string memory for string A long

; return X=Sutill=ptr low byte, Y=Sutilh=ptr high byte

STX str\_pl ; save string pointer low byte

STY str\_ph ; save string pointer high byte

STA str\_ln ; save length

RTS

; Scan, set up string

; print " terminated string to Sutill/Sutilh

LAB\_20AE

LDX #$22 ; set terminator to "

STX Srchc ; set search character (terminator 1)

STX Asrch ; set terminator 2

; print [Srchc] or [Asrch] terminated string to Sutill/Sutilh

; source is AY

LAB\_20B4

STA ssptr\_l ; store string start low byte

STY ssptr\_h ; store string start high byte

STA str\_pl ; save string pointer low byte

STY str\_ph ; save string pointer high byte

LDY #$FF ; set length to -1

LAB\_20BE

INY ; increment length

LDA (ssptr\_l),Y ; get byte from string

BEQ LAB\_20CF ; exit loop if null byte [EOS]

CMP Srchc ; compare with search character (terminator 1)

BEQ LAB\_20CB ; branch if terminator

CMP Asrch ; compare with terminator 2

BNE LAB\_20BE ; loop if not terminator 2

LAB\_20CB

CMP #$22 ; compare with "

BEQ LAB\_20D0 ; branch if " (carry set if = !)

LAB\_20CF

CLC ; clear carry for add (only if [EOL] terminated string)

LAB\_20D0

STY str\_ln ; save length in FAC1 exponent

TYA ; copy length to A

ADC ssptr\_l ; add string start low byte

STA Sendl ; save string end low byte

LDX ssptr\_h ; get string start high byte

BCC LAB\_20DC ; branch if no low byte overflow

INX ; else increment high byte

LAB\_20DC

STX Sendh ; save string end high byte

LDA ssptr\_h ; get string start high byte

CMP #>Ram\_base ; compare with start of program memory

BCS LAB\_RTST ; branch if not in utility area

; string in utility area, move to string memory

TYA ; copy length to A

JSR LAB\_209C ; copy des\_pl/h to des\_2l/h and make string space A bytes

; long

LDX ssptr\_l ; get string start low byte

LDY ssptr\_h ; get string start high byte

JSR LAB\_2298 ; store string A bytes long from XY to (Sutill)

; check for space on descriptor stack then ..

; put string address and length on descriptor stack and update stack pointers

LAB\_RTST

LDX next\_s ; get string stack pointer

CPX #des\_sk+$09 ; compare with max+1

BNE LAB\_20F8 ; branch if space on string stack

; else do string too complex error

LDX #$1C ; error code $1C ("String too complex" error)

LAB\_20F5

JMP LAB\_XERR ; do error #X, then warm start

; put string address and length on descriptor stack and update stack pointers

LAB\_20F8

LDA str\_ln ; get string length

STA PLUS\_0,X ; put on string stack

LDA str\_pl ; get string pointer low byte

STA PLUS\_1,X ; put on string stack

LDA str\_ph ; get string pointer high byte

STA PLUS\_2,X ; put on string stack

LDY #$00 ; clear Y

STX des\_pl ; save string descriptor pointer low byte

STY des\_ph ; save string descriptor pointer high byte (always $00)

DEY ; Y = $FF

STY Dtypef ; save data type flag, $FF=string

STX last\_sl ; save old stack pointer (current top item)

INX ; update stack pointer

INX ; update stack pointer

INX ; update stack pointer

STX next\_s ; save new top item value

RTS

; Build descriptor

; make space in string memory for string A long

; return X=Sutill=ptr low byte, Y=Sutill=ptr high byte

LAB\_2115

LSR Gclctd ; clear garbage collected flag (b7)

; make space for string A long

LAB\_2117

PHA ; save string length

EOR #$FF ; complement it

SEC ; set carry for subtract (twos comp add)

ADC Sstorl ; add bottom of string space low byte (subtract length)

LDY Sstorh ; get bottom of string space high byte

BCS LAB\_2122 ; skip decrement if no underflow

DEY ; decrement bottom of string space high byte

LAB\_2122

CPY Earryh ; compare with array mem end high byte

BCC LAB\_2137 ; do out of memory error if less

BNE LAB\_212C ; if not = skip next test

CMP Earryl ; compare with array mem end low byte

BCC LAB\_2137 ; do out of memory error if less

LAB\_212C

STA Sstorl ; save bottom of string space low byte

STY Sstorh ; save bottom of string space high byte

STA Sutill ; save string utility ptr low byte

STY Sutilh ; save string utility ptr high byte

TAX ; copy low byte to X

PLA ; get string length back

RTS

LAB\_2137

LDX #$0C ; error code $0C ("Out of memory" error)

LDA Gclctd ; get garbage collected flag

BMI LAB\_20F5 ; if set then do error code X

JSR LAB\_GARB ; else go do garbage collection

LDA #$80 ; flag for garbage collected

STA Gclctd ; set garbage collected flag

PLA ; pull length

BNE LAB\_2117 ; go try again (loop always, length should never be = $00)

; garbage collection routine

LAB\_GARB

LDX Ememl ; get end of mem low byte

LDA Ememh ; get end of mem high byte

; re-run routine from last ending

LAB\_214B

STX Sstorl ; set string storage low byte

STA Sstorh ; set string storage high byte

LDY #$00 ; clear index

STY garb\_h ; clear working pointer high byte (flag no strings to move)

LDA Earryl ; get array mem end low byte

LDX Earryh ; get array mem end high byte

STA Histrl ; save as highest string low byte

STX Histrh ; save as highest string high byte

LDA #des\_sk ; set descriptor stack pointer

STA ut1\_pl ; save descriptor stack pointer low byte

STY ut1\_ph ; save descriptor stack pointer high byte ($00)

LAB\_2161

CMP next\_s ; compare with descriptor stack pointer

BEQ LAB\_216A ; branch if =

JSR LAB\_21D7 ; go garbage collect descriptor stack

BEQ LAB\_2161 ; loop always

; done stacked strings, now do string vars

LAB\_216A

ASL g\_step ; set step size = $06

LDA Svarl ; get start of vars low byte

LDX Svarh ; get start of vars high byte

STA ut1\_pl ; save as pointer low byte

STX ut1\_ph ; save as pointer high byte

LAB\_2176

CPX Sarryh ; compare start of arrays high byte

BNE LAB\_217E ; branch if no high byte match

CMP Sarryl ; else compare start of arrays low byte

BEQ LAB\_2183 ; branch if = var mem end

LAB\_217E

JSR LAB\_21D1 ; go garbage collect strings

BEQ LAB\_2176 ; loop always

; done string vars, now do string arrays

LAB\_2183

STA Nbendl ; save start of arrays low byte as working pointer

STX Nbendh ; save start of arrays high byte as working pointer

LDA #$04 ; set step size

STA g\_step ; save step size

LAB\_218B

LDA Nbendl ; get pointer low byte

LDX Nbendh ; get pointer high byte

LAB\_218F

CPX Earryh ; compare with array mem end high byte

BNE LAB\_219A ; branch if not at end

CMP Earryl ; else compare with array mem end low byte

BEQ LAB\_2216 ; tidy up and exit if at end

LAB\_219A

STA ut1\_pl ; save pointer low byte

STX ut1\_ph ; save pointer high byte

LDY #$02 ; set index

LDA (ut1\_pl),Y ; get array size low byte

ADC Nbendl ; add start of this array low byte

STA Nbendl ; save start of next array low byte

INY ; increment index

LDA (ut1\_pl),Y ; get array size high byte

ADC Nbendh ; add start of this array high byte

STA Nbendh ; save start of next array high byte

LDY #$01 ; set index

LDA (ut1\_pl),Y ; get name second byte

BPL LAB\_218B ; skip if not string array

; was string array so ..

LDY #$04 ; set index

LDA (ut1\_pl),Y ; get # of dimensions

ASL ; \*2

ADC #$05 ; +5 (array header size)

JSR LAB\_2208 ; go set up for first element

LAB\_21C4

CPX Nbendh ; compare with start of next array high byte

BNE LAB\_21CC ; branch if <> (go do this array)

CMP Nbendl ; else compare element pointer low byte with next array

; low byte

BEQ LAB\_218F ; if equal then go do next array

LAB\_21CC

JSR LAB\_21D7 ; go defrag array strings

BEQ LAB\_21C4 ; go do next array string (loop always)

; defrag string variables

; enter with XA = variable pointer

; return with XA = next variable pointer

LAB\_21D1

INY ; increment index (Y was $00)

LDA (ut1\_pl),Y ; get var name byte 2

BPL LAB\_2206 ; if not string, step pointer to next var and return

INY ; else increment index

LAB\_21D7

LDA (ut1\_pl),Y ; get string length

BEQ LAB\_2206 ; if null, step pointer to next string and return

INY ; else increment index

LDA (ut1\_pl),Y ; get string pointer low byte

TAX ; copy to X

INY ; increment index

LDA (ut1\_pl),Y ; get string pointer high byte

CMP Sstorh ; compare bottom of string space high byte

BCC LAB\_21EC ; branch if less

BNE LAB\_2206 ; if greater, step pointer to next string and return

; high bytes were = so compare low bytes

CPX Sstorl ; compare bottom of string space low byte

BCS LAB\_2206 ; if >=, step pointer to next string and return

; string pointer is < string storage pointer (pos in mem)

LAB\_21EC

CMP Histrh ; compare to highest string high byte

BCC LAB\_2207 ; if <, step pointer to next string and return

BNE LAB\_21F6 ; if > update pointers, step to next and return

; high bytes were = so compare low bytes

CPX Histrl ; compare to highest string low byte

BCC LAB\_2207 ; if <, step pointer to next string and return

; string is in string memory space

LAB\_21F6

STX Histrl ; save as new highest string low byte

STA Histrh ; save as new highest string high byte

LDA ut1\_pl ; get start of vars(descriptors) low byte

LDX ut1\_ph ; get start of vars(descriptors) high byte

STA garb\_l ; save as working pointer low byte

STX garb\_h ; save as working pointer high byte

DEY ; decrement index DIFFERS

DEY ; decrement index (should point to descriptor start)

STY g\_indx ; save index pointer

; step pointer to next string

LAB\_2206

CLC ; clear carry for add

LAB\_2207

LDA g\_step ; get step size

LAB\_2208

ADC ut1\_pl ; add pointer low byte

STA ut1\_pl ; save pointer low byte

BCC LAB\_2211 ; branch if no overflow

INC ut1\_ph ; else increment high byte

LAB\_2211

LDX ut1\_ph ; get pointer high byte

LDY #$00 ; clear Y

RTS

; search complete, now either exit or set-up and move string

LAB\_2216

DEC g\_step ; decrement step size (now $03 for descriptor stack)

LDX garb\_h ; get string to move high byte

BEQ LAB\_2211 ; exit if nothing to move

LDY g\_indx ; get index byte back (points to descriptor)

CLC ; clear carry for add

LDA (garb\_l),Y ; get string length

ADC Histrl ; add highest string low byte

STA Obendl ; save old block end low pointer

LDA Histrh ; get highest string high byte

ADC #$00 ; add any carry

STA Obendh ; save old block end high byte

LDA Sstorl ; get bottom of string space low byte

LDX Sstorh ; get bottom of string space high byte

STA Nbendl ; save new block end low byte

STX Nbendh ; save new block end high byte

JSR LAB\_11D6 ; open up space in memory, don't set array end

LDY g\_indx ; get index byte

INY ; point to descriptor low byte

LDA Nbendl ; get string pointer low byte

STA (garb\_l),Y ; save new string pointer low byte

TAX ; copy string pointer low byte

INC Nbendh ; correct high byte (move sets high byte -1)

LDA Nbendh ; get new string pointer high byte

INY ; point to descriptor high byte

STA (garb\_l),Y ; save new string pointer high byte

JMP LAB\_214B ; re-run routine from last ending

; (but don't collect this string)

; concatenate

; add strings, string 1 is in descriptor des\_pl, string 2 is in line

LAB\_224D

LDA des\_ph ; get descriptor pointer high byte

PHA ; put on stack

LDA des\_pl ; get descriptor pointer low byte

PHA ; put on stack

JSR LAB\_GVAL ; get value from line

JSR LAB\_CTST ; check if source is string, else do type mismatch

PLA ; get descriptor pointer low byte back

STA ssptr\_l ; set pointer low byte

PLA ; get descriptor pointer high byte back

STA ssptr\_h ; set pointer high byte

LDY #$00 ; clear index

LDA (ssptr\_l),Y ; get length\_1 from descriptor

CLC ; clear carry for add

ADC (des\_pl),Y ; add length\_2

BCC LAB\_226D ; branch if no overflow

LDX #$1A ; else set error code $1A ("String too long" error)

JMP LAB\_XERR ; do error #X, then warm start

LAB\_226D

JSR LAB\_209C ; copy des\_pl/h to des\_2l/h and make string space A bytes

; long

JSR LAB\_228A ; copy string from descriptor (sdescr) to (Sutill)

LDA des\_2l ; get descriptor pointer low byte

LDY des\_2h ; get descriptor pointer high byte

JSR LAB\_22BA ; pop (YA) descriptor off stack or from top of string space

; returns with A = length, ut1\_pl = pointer low byte,

; ut1\_ph = pointer high byte

JSR LAB\_229C ; store string A bytes long from (ut1\_pl) to (Sutill)

LDA ssptr\_l ;.set descriptor pointer low byte

LDY ssptr\_h ;.set descriptor pointer high byte

JSR LAB\_22BA ; pop (YA) descriptor off stack or from top of string space

; returns with A = length, X=ut1\_pl=pointer low byte,

; Y=ut1\_ph=pointer high byte

JSR LAB\_RTST ; check for space on descriptor stack then put string

; address and length on descriptor stack and update stack

; pointers

JMP LAB\_1ADB ;.continue evaluation

; copy string from descriptor (sdescr) to (Sutill)

LAB\_228A

LDY #$00 ; clear index

LDA (sdescr),Y ; get string length

PHA ; save on stack

INY ; increment index

LDA (sdescr),Y ; get source string pointer low byte

TAX ; copy to X

INY ; increment index

LDA (sdescr),Y ; get source string pointer high byte

TAY ; copy to Y

PLA ; get length back

; store string A bytes long from YX to (Sutill)

LAB\_2298

STX ut1\_pl ; save source string pointer low byte

STY ut1\_ph ; save source string pointer high byte

; store string A bytes long from (ut1\_pl) to (Sutill)

LAB\_229C

TAX ; copy length to index (don't count with Y)

BEQ LAB\_22B2 ; branch if = $0 (null string) no need to add zero length

LDY #$00 ; zero pointer (copy forward)

LAB\_22A0

LDA (ut1\_pl),Y ; get source byte

STA (Sutill),Y ; save destination byte

INY ; increment index

DEX ; decrement counter

BNE LAB\_22A0 ; loop while <> 0

TYA ; restore length from Y

LAB\_22A9

CLC ; clear carry for add

ADC Sutill ; add string utility ptr low byte

STA Sutill ; save string utility ptr low byte

BCC LAB\_22B2 ; branch if no carry

INC Sutilh ; else increment string utility ptr high byte

LAB\_22B2

RTS

; evaluate string

LAB\_EVST

JSR LAB\_CTST ; check if source is string, else do type mismatch

; pop string off descriptor stack, or from top of string space

; returns with A = length, X=pointer low byte, Y=pointer high byte

LAB\_22B6

LDA des\_pl ; get descriptor pointer low byte

LDY des\_ph ; get descriptor pointer high byte

; pop (YA) descriptor off stack or from top of string space

; returns with A = length, X=ut1\_pl=pointer low byte, Y=ut1\_ph=pointer high byte

LAB\_22BA

STA ut1\_pl ; save descriptor pointer low byte

STY ut1\_ph ; save descriptor pointer high byte

JSR LAB\_22EB ; clean descriptor stack, YA = pointer

PHP ; save status flags

LDY #$00 ; clear index

LDA (ut1\_pl),Y ; get length from string descriptor

PHA ; put on stack

INY ; increment index

LDA (ut1\_pl),Y ; get string pointer low byte from descriptor

TAX ; copy to X

INY ; increment index

LDA (ut1\_pl),Y ; get string pointer high byte from descriptor

TAY ; copy to Y

PLA ; get string length back

PLP ; restore status

BNE LAB\_22E6 ; branch if pointer <> last\_sl,last\_sh

CPY Sstorh ; compare bottom of string space high byte

BNE LAB\_22E6 ; branch if <>

CPX Sstorl ; else compare bottom of string space low byte

BNE LAB\_22E6 ; branch if <>

PHA ; save string length

CLC ; clear carry for add

ADC Sstorl ; add bottom of string space low byte

STA Sstorl ; save bottom of string space low byte

BCC LAB\_22E5 ; skip increment if no overflow

INC Sstorh ; increment bottom of string space high byte

LAB\_22E5

PLA ; restore string length

LAB\_22E6

STX ut1\_pl ; save string pointer low byte

STY ut1\_ph ; save string pointer high byte

RTS

; clean descriptor stack, YA = pointer

; checks if AY is on the descriptor stack, if so does a stack discard

LAB\_22EB

CPY last\_sh ; compare pointer high byte

BNE LAB\_22FB ; exit if <>

CMP last\_sl ; compare pointer low byte

BNE LAB\_22FB ; exit if <>

STA next\_s ; save descriptor stack pointer

SBC #$03 ; -3

STA last\_sl ; save low byte -3

LDY #$00 ; clear high byte

LAB\_22FB

RTS

; perform CHR$()

LAB\_CHRS

JSR LAB\_EVBY ; evaluate byte expression, result in X

TXA ; copy to A

PHA ; save character

LDA #$01 ; string is single byte

JSR LAB\_MSSP ; make string space A bytes long A=$AC=length,

; X=$AD=Sutill=ptr low byte, Y=$AE=Sutilh=ptr high byte

PLA ; get character back

LDY #$00 ; clear index

STA (str\_pl),Y ; save byte in string (byte IS string!)

JMP LAB\_RTST ; check for space on descriptor stack then put string

; address and length on descriptor stack and update stack

; pointers

; perform LEFT$()

LAB\_LEFT

PHA ; push byte parameter

JSR LAB\_236F ; pull string data and byte parameter from stack

; return pointer in des\_2l/h, byte in A (and X), Y=0

CMP (des\_2l),Y ; compare byte parameter with string length

TYA ; clear A

BEQ LAB\_2316 ; go do string copy (branch always)

; perform RIGHT$()

LAB\_RIGHT

PHA ; push byte parameter

JSR LAB\_236F ; pull string data and byte parameter from stack

; return pointer in des\_2l/h, byte in A (and X), Y=0

CLC ; clear carry for add-1

SBC (des\_2l),Y ; subtract string length

EOR #$FF ; invert it (A=LEN(expression$)-l)

LAB\_2316

BCC LAB\_231C ; branch if string length > byte parameter

LDA (des\_2l),Y ; else make parameter = length

TAX ; copy to byte parameter copy

TYA ; clear string start offset

LAB\_231C

PHA ; save string start offset

LAB\_231D

TXA ; copy byte parameter (or string length if <)

LAB\_231E

PHA ; save string length

JSR LAB\_MSSP ; make string space A bytes long A=$AC=length,

; X=$AD=Sutill=ptr low byte, Y=$AE=Sutilh=ptr high byte

LDA des\_2l ; get descriptor pointer low byte

LDY des\_2h ; get descriptor pointer high byte

JSR LAB\_22BA ; pop (YA) descriptor off stack or from top of string space

; returns with A = length, X=ut1\_pl=pointer low byte,

; Y=ut1\_ph=pointer high byte

PLA ; get string length back

TAY ; copy length to Y

PLA ; get string start offset back

CLC ; clear carry for add

ADC ut1\_pl ; add start offset to string start pointer low byte

STA ut1\_pl ; save string start pointer low byte

BCC LAB\_2335 ; branch if no overflow

INC ut1\_ph ; else increment string start pointer high byte

LAB\_2335

TYA ; copy length to A

JSR LAB\_229C ; store string A bytes long from (ut1\_pl) to (Sutill)

JMP LAB\_RTST ; check for space on descriptor stack then put string

; address and length on descriptor stack and update stack

; pointers

; perform MID$()

LAB\_MIDS

PHA ; push byte parameter

LDA #$FF ; set default length = 255

STA mids\_l ; save default length

JSR LAB\_GBYT ; scan memory

CMP #')' ; compare with ")"

BEQ LAB\_2358 ; branch if = ")" (skip second byte get)

JSR LAB\_1C01 ; scan for "," , else do syntax error then warm start

JSR LAB\_GTBY ; get byte parameter (use copy in mids\_l)

LAB\_2358

JSR LAB\_236F ; pull string data and byte parameter from stack

; return pointer in des\_2l/h, byte in A (and X), Y=0

DEX ; decrement start index

TXA ; copy to A

PHA ; save string start offset

CLC ; clear carry for sub-1

LDX #$00 ; clear output string length

SBC (des\_2l),Y ; subtract string length

BCS LAB\_231D ; if start>string length go do null string

EOR #$FF ; complement -length

CMP mids\_l ; compare byte parameter

BCC LAB\_231E ; if length>remaining string go do RIGHT$

LDA mids\_l ; get length byte

BCS LAB\_231E ; go do string copy (branch always)

; pull string data and byte parameter from stack

; return pointer in des\_2l/h, byte in A (and X), Y=0

LAB\_236F

JSR LAB\_1BFB ; scan for ")" , else do syntax error then warm start

PLA ; pull return address low byte (return address)

STA Fnxjpl ; save functions jump vector low byte

PLA ; pull return address high byte (return address)

STA Fnxjph ; save functions jump vector high byte

PLA ; pull byte parameter

TAX ; copy byte parameter to X

PLA ; pull string pointer low byte

STA des\_2l ; save it

PLA ; pull string pointer high byte

STA des\_2h ; save it

LDY #$00 ; clear index

TXA ; copy byte parameter

BEQ LAB\_23A8 ; if null do function call error then warm start

INC Fnxjpl ; increment function jump vector low byte

; (JSR pushes return addr-1. this is all very nice

; but will go tits up if either call is on a page

; boundary!)

JMP (Fnxjpl) ; in effect, RTS

; perform LCASE$()

LAB\_LCASE

JSR LAB\_EVST ; evaluate string

STA str\_ln ; set string length

TAY ; copy length to Y

BEQ NoString ; branch if null string

JSR LAB\_MSSP ; make string space A bytes long A=length,

; X=Sutill=ptr low byte, Y=Sutilh=ptr high byte

STX str\_pl ; save string pointer low byte

STY str\_ph ; save string pointer high byte

TAY ; get string length back

LC\_loop

DEY ; decrement index

LDA (ut1\_pl),Y ; get byte from string

JSR LAB\_1D82 ; is character "A" to "Z"

BCC NoUcase ; branch if not upper case alpha

ORA #$20 ; convert upper to lower case

NoUcase

STA (Sutill),Y ; save byte back to string

TYA ; test index

BNE LC\_loop ; loop if not all done

BEQ NoString ; tidy up and exit, branch always

; perform UCASE$()

LAB\_UCASE

JSR LAB\_EVST ; evaluate string

STA str\_ln ; set string length

TAY ; copy length to Y

BEQ NoString ; branch if null string

JSR LAB\_MSSP ; make string space A bytes long A=length,

; X=Sutill=ptr low byte, Y=Sutilh=ptr high byte

STX str\_pl ; save string pointer low byte

STY str\_ph ; save string pointer high byte

TAY ; get string length back

UC\_loop

DEY ; decrement index

LDA (ut1\_pl),Y ; get byte from string

JSR LAB\_CASC ; is character "a" to "z" (or "A" to "Z")

BCC NoLcase ; branch if not alpha

AND #$DF ; convert lower to upper case

NoLcase

STA (Sutill),Y ; save byte back to string

TYA ; test index

BNE UC\_loop ; loop if not all done

NoString

JMP LAB\_RTST ; check for space on descriptor stack then put string

; address and length on descriptor stack and update stack

; pointers

; perform SADD()

LAB\_SADD

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_GVAR ; get var address

JSR LAB\_1BFB ; scan for ")", else do syntax error then warm start

JSR LAB\_CTST ; check if source is string, else do type mismatch

LDY #$02 ; index to string pointer high byte

LDA (Cvaral),Y ; get string pointer high byte

TAX ; copy string pointer high byte to X

DEY ; index to string pointer low byte

LDA (Cvaral),Y ; get string pointer low byte

TAY ; copy string pointer low byte to Y

TXA ; copy string pointer high byte to A

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform LEN()

LAB\_LENS

JSR LAB\_ESGL ; evaluate string, get length in A (and Y)

JMP LAB\_1FD0 ; convert Y to byte in FAC1 and return

; evaluate string, get length in Y

LAB\_ESGL

JSR LAB\_EVST ; evaluate string

TAY ; copy length to Y

RTS

; perform ASC()

LAB\_ASC

JSR LAB\_ESGL ; evaluate string, get length in A (and Y)

BEQ LAB\_23A8 ; if null do function call error then warm start

LDY #$00 ; set index to first character

LDA (ut1\_pl),Y ; get byte

TAY ; copy to Y

JMP LAB\_1FD0 ; convert Y to byte in FAC1 and return

; do function call error then warm start

LAB\_23A8

JMP LAB\_FCER ; do function call error then warm start

; scan and get byte parameter

LAB\_SGBY

JSR LAB\_IGBY ; increment and scan memory

; get byte parameter

LAB\_GTBY

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

; evaluate byte expression, result in X

LAB\_EVBY

JSR LAB\_EVPI ; evaluate integer expression (no check)

LDY FAC1\_2 ; get FAC1 mantissa2

BNE LAB\_23A8 ; if top byte <> 0 do function call error then warm start

LDX FAC1\_3 ; get FAC1 mantissa3

JMP LAB\_GBYT ; scan memory and return

; perform VAL()

LAB\_VAL

JSR LAB\_ESGL ; evaluate string, get length in A (and Y)

BNE LAB\_23C5 ; branch if not null string

; string was null so set result = $00

JMP LAB\_24F1 ; clear FAC1 exponent and sign and return

LAB\_23C5

LDX Bpntrl ; get BASIC execute pointer low byte

LDY Bpntrh ; get BASIC execute pointer high byte

STX Btmpl ; save BASIC execute pointer low byte

STY Btmph ; save BASIC execute pointer high byte

LDX ut1\_pl ; get string pointer low byte

STX Bpntrl ; save as BASIC execute pointer low byte

CLC ; clear carry

ADC ut1\_pl ; add string length

STA ut2\_pl ; save string end low byte

LDA ut1\_ph ; get string pointer high byte

STA Bpntrh ; save as BASIC execute pointer high byte

ADC #$00 ; add carry to high byte

STA ut2\_ph ; save string end high byte

LDY #$00 ; set index to $00

LDA (ut2\_pl),Y ; get string end +1 byte

PHA ; push it

TYA ; clear A

STA (ut2\_pl),Y ; terminate string with $00

JSR LAB\_GBYT ; scan memory

JSR LAB\_2887 ; get FAC1 from string

PLA ; restore string end +1 byte

LDY #$00 ; set index to zero

STA (ut2\_pl),Y ; put string end byte back

; restore BASIC execute pointer from temp (Btmpl/Btmph)

LAB\_23F3

LDX Btmpl ; get BASIC execute pointer low byte back

LDY Btmph ; get BASIC execute pointer high byte back

STX Bpntrl ; save BASIC execute pointer low byte

STY Bpntrh ; save BASIC execute pointer high byte

RTS

; get two parameters for POKE or WAIT

LAB\_GADB

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

JSR LAB\_F2FX ; save integer part of FAC1 in temporary integer

; scan for "," and get byte, else do Syntax error then warm start

LAB\_SCGB

JSR LAB\_1C01 ; scan for "," , else do syntax error then warm start

LDA Itemph ; save temporary integer high byte

PHA ; on stack

LDA Itempl ; save temporary integer low byte

PHA ; on stack

JSR LAB\_GTBY ; get byte parameter

PLA ; pull low byte

STA Itempl ; restore temporary integer low byte

PLA ; pull high byte

STA Itemph ; restore temporary integer high byte

RTS

; convert float to fixed routine. accepts any value that fits in 24 bits, +ve or

; -ve and converts it into a right truncated integer in Itempl and Itemph

; save unsigned 16 bit integer part of FAC1 in temporary integer

LAB\_F2FX

LDA FAC1\_e ; get FAC1 exponent

CMP #$98 ; compare with exponent = 2^24

BCS LAB\_23A8 ; if >= do function call error then warm start

LAB\_F2FU

JSR LAB\_2831 ; convert FAC1 floating-to-fixed

LDA FAC1\_2 ; get FAC1 mantissa2

LDY FAC1\_3 ; get FAC1 mantissa3

STY Itempl ; save temporary integer low byte

STA Itemph ; save temporary integer high byte

RTS

; perform PEEK()

LAB\_PEEK

JSR LAB\_F2FX ; save integer part of FAC1 in temporary integer

LDX #$00 ; clear index

LDA (Itempl,X) ; get byte via temporary integer (addr)

TAY ; copy byte to Y

JMP LAB\_1FD0 ; convert Y to byte in FAC1 and return

; perform POKE

LAB\_POKE

JSR LAB\_GADB ; get two parameters for POKE or WAIT

TXA ; copy byte argument to A

LDX #$00 ; clear index

STA (Itempl,X) ; save byte via temporary integer (addr)

RTS

; perform DEEK()

LAB\_DEEK

JSR LAB\_F2FX ; save integer part of FAC1 in temporary integer

LDX #$00 ; clear index

LDA (Itempl,X) ; PEEK low byte

TAY ; copy to Y

INC Itempl ; increment pointer low byte

BNE Deekh ; skip high increment if no rollover

INC Itemph ; increment pointer high byte

Deekh

LDA (Itempl,X) ; PEEK high byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform DOKE

LAB\_DOKE

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

JSR LAB\_F2FX ; convert floating-to-fixed

STY Frnxtl ; save pointer low byte (float to fixed returns word in AY)

STA Frnxth ; save pointer high byte

JSR LAB\_1C01 ; scan for "," , else do syntax error then warm start

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

JSR LAB\_F2FX ; convert floating-to-fixed

TYA ; copy value low byte (float to fixed returns word in AY)

LDX #$00 ; clear index

STA (Frnxtl,X) ; POKE low byte

INC Frnxtl ; increment pointer low byte

BNE Dokeh ; skip high increment if no rollover

INC Frnxth ; increment pointer high byte

Dokeh

LDA Itemph ; get value high byte

STA (Frnxtl,X) ; POKE high byte

JMP LAB\_GBYT ; scan memory and return

; perform SWAP

LAB\_SWAP

JSR LAB\_GVAR ; get var1 address

STA Lvarpl ; save var1 address low byte

STY Lvarph ; save var1 address high byte

LDA Dtypef ; get data type flag, $FF=string, $00=numeric

PHA ; save data type flag

JSR LAB\_1C01 ; scan for "," , else do syntax error then warm start

JSR LAB\_GVAR ; get var2 address (pointer in Cvaral/h)

PLA ; pull var1 data type flag

EOR Dtypef ; compare with var2 data type

BPL SwapErr ; exit if not both the same type

LDY #$03 ; four bytes to swap (either value or descriptor+1)

SwapLp

LDA (Lvarpl),Y ; get byte from var1

TAX ; save var1 byte

LDA (Cvaral),Y ; get byte from var2

STA (Lvarpl),Y ; save byte to var1

TXA ; restore var1 byte

STA (Cvaral),Y ; save byte to var2

DEY ; decrement index

BPL SwapLp ; loop until done

RTS

SwapErr

JMP LAB\_1ABC ; do "Type mismatch" error then warm start

; perform CALL

LAB\_CALL

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

JSR LAB\_F2FX ; convert floating-to-fixed

LDA #>CallExit ; set return address high byte

PHA ; put on stack

LDA #<CallExit-1 ; set return address low byte

PHA ; put on stack

JMP (Itempl) ; do indirect jump to user routine

; if the called routine exits correctly then it will return to here. this will then get

; the next byte for the interpreter and return

CallExit

JMP LAB\_GBYT ; scan memory and return

; perform WAIT

LAB\_WAIT

JSR LAB\_GADB ; get two parameters for POKE or WAIT

STX Frnxtl ; save byte

LDX #$00 ; clear mask

JSR LAB\_GBYT ; scan memory

BEQ LAB\_2441 ; skip if no third argument

JSR LAB\_SCGB ; scan for "," and get byte, else SN error then warm start

LAB\_2441

STX Frnxth ; save EOR argument

LAB\_2445

LDA (Itempl),Y ; get byte via temporary integer (addr)

EOR Frnxth ; EOR with second argument (mask)

AND Frnxtl ; AND with first argument (byte)

BEQ LAB\_2445 ; loop if result is zero

LAB\_244D

RTS

; perform subtraction, FAC1 from (AY)

LAB\_2455

JSR LAB\_264D ; unpack memory (AY) into FAC2

; perform subtraction, FAC1 from FAC2

LAB\_SUBTRACT

LDA FAC1\_s ; get FAC1 sign (b7)

EOR #$FF ; complement it

STA FAC1\_s ; save FAC1 sign (b7)

EOR FAC2\_s ; EOR with FAC2 sign (b7)

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

LDA FAC1\_e ; get FAC1 exponent

JMP LAB\_ADD ; go add FAC2 to FAC1

; perform addition

LAB\_2467

JSR LAB\_257B ; shift FACX A times right (>8 shifts)

BCC LAB\_24A8 ;.go subtract mantissas

; add 0.5 to FAC1

LAB\_244E

LDA #<LAB\_2A96 ; set 0.5 pointer low byte

LDY #>LAB\_2A96 ; set 0.5 pointer high byte

; add (AY) to FAC1

LAB\_246C

JSR LAB\_264D ; unpack memory (AY) into FAC2

; add FAC2 to FAC1

LAB\_ADD

BNE LAB\_2474 ; branch if FAC1 was not zero

; copy FAC2 to FAC1

LAB\_279B

LDA FAC2\_s ; get FAC2 sign (b7)

; save FAC1 sign and copy ABS(FAC2) to FAC1

LAB\_279D

STA FAC1\_s ; save FAC1 sign (b7)

LDX #$04 ; 4 bytes to copy

LAB\_27A1

LDA FAC1\_o,X ; get byte from FAC2,X

STA FAC1\_e-1,X ; save byte at FAC1,X

DEX ; decrement count

BNE LAB\_27A1 ; loop if not all done

STX FAC1\_r ; clear FAC1 rounding byte

RTS

; FAC1 is non zero

LAB\_2474

LDX FAC1\_r ; get FAC1 rounding byte

STX FAC2\_r ; save as FAC2 rounding byte

LDX #FAC2\_e ; set index to FAC2 exponent addr

LDA FAC2\_e ; get FAC2 exponent

LAB\_247C

TAY ; copy exponent

BEQ LAB\_244D ; exit if zero

SEC ; set carry for subtract

SBC FAC1\_e ; subtract FAC1 exponent

BEQ LAB\_24A8 ; branch if = (go add mantissa)

BCC LAB\_2498 ; branch if <

; FAC2>FAC1

STY FAC1\_e ; save FAC1 exponent

LDY FAC2\_s ; get FAC2 sign (b7)

STY FAC1\_s ; save FAC1 sign (b7)

EOR #$FF ; complement A

ADC #$00 ; +1 (twos complement, carry is set)

LDY #$00 ; clear Y

STY FAC2\_r ; clear FAC2 rounding byte

LDX #FAC1\_e ; set index to FAC1 exponent addr

BNE LAB\_249C ; branch always

LAB\_2498

LDY #$00 ; clear Y

STY FAC1\_r ; clear FAC1 rounding byte

LAB\_249C

CMP #$F9 ; compare exponent diff with $F9

BMI LAB\_2467 ; branch if range $79-$F8

TAY ; copy exponent difference to Y

LDA FAC1\_r ; get FAC1 rounding byte

LSR PLUS\_1,X ; shift FAC? mantissa1

JSR LAB\_2592 ; shift FACX Y times right

; exponents are equal now do mantissa subtract

LAB\_24A8

BIT FAC\_sc ; test sign compare (FAC1 EOR FAC2)

BPL LAB\_24F8 ; if = add FAC2 mantissa to FAC1 mantissa and return

LDY #FAC1\_e ; set index to FAC1 exponent addr

CPX #FAC2\_e ; compare X to FAC2 exponent addr

BEQ LAB\_24B4 ; branch if =

LDY #FAC2\_e ; else set index to FAC2 exponent addr

; subtract smaller from bigger (take sign of bigger)

LAB\_24B4

SEC ; set carry for subtract

EOR #$FF ; ones complement A

ADC FAC2\_r ; add FAC2 rounding byte

STA FAC1\_r ; save FAC1 rounding byte

LDA PLUS\_3,Y ; get FACY mantissa3

SBC PLUS\_3,X ; subtract FACX mantissa3

STA FAC1\_3 ; save FAC1 mantissa3

LDA PLUS\_2,Y ; get FACY mantissa2

SBC PLUS\_2,X ; subtract FACX mantissa2

STA FAC1\_2 ; save FAC1 mantissa2

LDA PLUS\_1,Y ; get FACY mantissa1

SBC PLUS\_1,X ; subtract FACX mantissa1

STA FAC1\_1 ; save FAC1 mantissa1

; do ABS and normalise FAC1

LAB\_24D0

BCS LAB\_24D5 ; branch if number is +ve

JSR LAB\_2537 ; negate FAC1

; normalise FAC1

LAB\_24D5

LDY #$00 ; clear Y

TYA ; clear A

CLC ; clear carry for add

LAB\_24D9

LDX FAC1\_1 ; get FAC1 mantissa1

BNE LAB\_251B ; if not zero normalise FAC1

LDX FAC1\_2 ; get FAC1 mantissa2

STX FAC1\_1 ; save FAC1 mantissa1

LDX FAC1\_3 ; get FAC1 mantissa3

STX FAC1\_2 ; save FAC1 mantissa2

LDX FAC1\_r ; get FAC1 rounding byte

STX FAC1\_3 ; save FAC1 mantissa3

STY FAC1\_r ; clear FAC1 rounding byte

ADC #$08 ; add x to exponent offset

CMP #$18 ; compare with $18 (max offset, all bits would be =0)

BNE LAB\_24D9 ; loop if not max

; clear FAC1 exponent and sign

LAB\_24F1

LDA #$00 ; clear A

LAB\_24F3

STA FAC1\_e ; set FAC1 exponent

; save FAC1 sign

LAB\_24F5

STA FAC1\_s ; save FAC1 sign (b7)

RTS

; add FAC2 mantissa to FAC1 mantissa

LAB\_24F8

ADC FAC2\_r ; add FAC2 rounding byte

STA FAC1\_r ; save FAC1 rounding byte

LDA FAC1\_3 ; get FAC1 mantissa3

ADC FAC2\_3 ; add FAC2 mantissa3

STA FAC1\_3 ; save FAC1 mantissa3

LDA FAC1\_2 ; get FAC1 mantissa2

ADC FAC2\_2 ; add FAC2 mantissa2

STA FAC1\_2 ; save FAC1 mantissa2

LDA FAC1\_1 ; get FAC1 mantissa1

ADC FAC2\_1 ; add FAC2 mantissa1

STA FAC1\_1 ; save FAC1 mantissa1

BCS LAB\_252A ; if carry then normalise FAC1 for C=1

RTS ; else just exit

LAB\_2511

ADC #$01 ; add 1 to exponent offset

ASL FAC1\_r ; shift FAC1 rounding byte

ROL FAC1\_3 ; shift FAC1 mantissa3

ROL FAC1\_2 ; shift FAC1 mantissa2

ROL FAC1\_1 ; shift FAC1 mantissa1

; normalise FAC1

LAB\_251B

BPL LAB\_2511 ; loop if not normalised

SEC ; set carry for subtract

SBC FAC1\_e ; subtract FAC1 exponent

BCS LAB\_24F1 ; branch if underflow (set result = $0)

EOR #$FF ; complement exponent

ADC #$01 ; +1 (twos complement)

STA FAC1\_e ; save FAC1 exponent

; test and normalise FAC1 for C=0/1

LAB\_2528

BCC LAB\_2536 ; exit if no overflow

; normalise FAC1 for C=1

LAB\_252A

INC FAC1\_e ; increment FAC1 exponent

BEQ LAB\_2564 ; if zero do overflow error and warm start

ROR FAC1\_1 ; shift FAC1 mantissa1

ROR FAC1\_2 ; shift FAC1 mantissa2

ROR FAC1\_3 ; shift FAC1 mantissa3

ROR FAC1\_r ; shift FAC1 rounding byte

LAB\_2536

RTS

; negate FAC1

LAB\_2537

LDA FAC1\_s ; get FAC1 sign (b7)

EOR #$FF ; complement it

STA FAC1\_s ; save FAC1 sign (b7)

; twos complement FAC1 mantissa

LAB\_253D

LDA FAC1\_1 ; get FAC1 mantissa1

EOR #$FF ; complement it

STA FAC1\_1 ; save FAC1 mantissa1

LDA FAC1\_2 ; get FAC1 mantissa2

EOR #$FF ; complement it

STA FAC1\_2 ; save FAC1 mantissa2

LDA FAC1\_3 ; get FAC1 mantissa3

EOR #$FF ; complement it

STA FAC1\_3 ; save FAC1 mantissa3

LDA FAC1\_r ; get FAC1 rounding byte

EOR #$FF ; complement it

STA FAC1\_r ; save FAC1 rounding byte

INC FAC1\_r ; increment FAC1 rounding byte

BNE LAB\_2563 ; exit if no overflow

; increment FAC1 mantissa

LAB\_2559

INC FAC1\_3 ; increment FAC1 mantissa3

BNE LAB\_2563 ; finished if no rollover

INC FAC1\_2 ; increment FAC1 mantissa2

BNE LAB\_2563 ; finished if no rollover

INC FAC1\_1 ; increment FAC1 mantissa1

LAB\_2563

RTS

; do overflow error (overflow exit)

LAB\_2564

LDX #$0A ; error code $0A ("Overflow" error)

JMP LAB\_XERR ; do error #X, then warm start

; shift FCAtemp << A+8 times

LAB\_2569

LDX #FACt\_1-1 ; set offset to FACtemp

LAB\_256B

LDY PLUS\_3,X ; get FACX mantissa3

STY FAC1\_r ; save as FAC1 rounding byte

LDY PLUS\_2,X ; get FACX mantissa2

STY PLUS\_3,X ; save FACX mantissa3

LDY PLUS\_1,X ; get FACX mantissa1

STY PLUS\_2,X ; save FACX mantissa2

LDY FAC1\_o ; get FAC1 overflow byte

STY PLUS\_1,X ; save FACX mantissa1

; shift FACX -A times right (> 8 shifts)

LAB\_257B

ADC #$08 ; add 8 to shift count

BMI LAB\_256B ; go do 8 shift if still -ve

BEQ LAB\_256B ; go do 8 shift if zero

SBC #$08 ; else subtract 8 again

TAY ; save count to Y

LDA FAC1\_r ; get FAC1 rounding byte

BCS LAB\_259A ;.

LAB\_2588

ASL PLUS\_1,X ; shift FACX mantissa1

BCC LAB\_258E ; branch if +ve

INC PLUS\_1,X ; this sets b7 eventually

LAB\_258E

ROR PLUS\_1,X ; shift FACX mantissa1 (correct for ASL)

ROR PLUS\_1,X ; shift FACX mantissa1 (put carry in b7)

; shift FACX Y times right

LAB\_2592

ROR PLUS\_2,X ; shift FACX mantissa2

ROR PLUS\_3,X ; shift FACX mantissa3

ROR ; shift FACX rounding byte

INY ; increment exponent diff

BNE LAB\_2588 ; branch if range adjust not complete

LAB\_259A

CLC ; just clear it

RTS

; perform LOG()

LAB\_LOG

JSR LAB\_27CA ; test sign and zero

BEQ LAB\_25C4 ; if zero do function call error then warm start

BPL LAB\_25C7 ; skip error if +ve

LAB\_25C4

JMP LAB\_FCER ; do function call error then warm start (-ve)

LAB\_25C7

LDA FAC1\_e ; get FAC1 exponent

SBC #$7F ; normalise it

PHA ; save it

LDA #$80 ; set exponent to zero

STA FAC1\_e ; save FAC1 exponent

LDA #<LAB\_25AD ; set 1/root2 pointer low byte

LDY #>LAB\_25AD ; set 1/root2 pointer high byte

JSR LAB\_246C ; add (AY) to FAC1 (1/root2)

LDA #<LAB\_25B1 ; set root2 pointer low byte

LDY #>LAB\_25B1 ; set root2 pointer high byte

JSR LAB\_26CA ; convert AY and do (AY)/FAC1 (root2/(x+(1/root2)))

LDA #<LAB\_259C ; set 1 pointer low byte

LDY #>LAB\_259C ; set 1 pointer high byte

JSR LAB\_2455 ; subtract (AY) from FAC1 ((root2/(x+(1/root2)))-1)

LDA #<LAB\_25A0 ; set pointer low byte to counter

LDY #>LAB\_25A0 ; set pointer high byte to counter

JSR LAB\_2B6E ; ^2 then series evaluation

LDA #<LAB\_25B5 ; set -0.5 pointer low byte

LDY #>LAB\_25B5 ; set -0.5 pointer high byte

JSR LAB\_246C ; add (AY) to FAC1

PLA ; restore FAC1 exponent

JSR LAB\_2912 ; evaluate new ASCII digit

LDA #<LAB\_25B9 ; set LOG(2) pointer low byte

LDY #>LAB\_25B9 ; set LOG(2) pointer high byte

; do convert AY, FCA1\*(AY)

LAB\_25FB

JSR LAB\_264D ; unpack memory (AY) into FAC2

LAB\_MULTIPLY

BEQ LAB\_264C ; exit if zero

JSR LAB\_2673 ; test and adjust accumulators

LDA #$00 ; clear A

STA FACt\_1 ; clear temp mantissa1

STA FACt\_2 ; clear temp mantissa2

STA FACt\_3 ; clear temp mantissa3

LDA FAC1\_r ; get FAC1 rounding byte

JSR LAB\_2622 ; go do shift/add FAC2

LDA FAC1\_3 ; get FAC1 mantissa3

JSR LAB\_2622 ; go do shift/add FAC2

LDA FAC1\_2 ; get FAC1 mantissa2

JSR LAB\_2622 ; go do shift/add FAC2

LDA FAC1\_1 ; get FAC1 mantissa1

JSR LAB\_2627 ; go do shift/add FAC2

JMP LAB\_273C ; copy temp to FAC1, normalise and return

LAB\_2622

BNE LAB\_2627 ; branch if byte <> zero

JMP LAB\_2569 ; shift FCAtemp << A+8 times

; else do shift and add

LAB\_2627

LSR ; shift byte

ORA #$80 ; set top bit (mark for 8 times)

LAB\_262A

TAY ; copy result

BCC LAB\_2640 ; skip next if bit was zero

CLC ; clear carry for add

LDA FACt\_3 ; get temp mantissa3

ADC FAC2\_3 ; add FAC2 mantissa3

STA FACt\_3 ; save temp mantissa3

LDA FACt\_2 ; get temp mantissa2

ADC FAC2\_2 ; add FAC2 mantissa2

STA FACt\_2 ; save temp mantissa2

LDA FACt\_1 ; get temp mantissa1

ADC FAC2\_1 ; add FAC2 mantissa1

STA FACt\_1 ; save temp mantissa1

LAB\_2640

ROR FACt\_1 ; shift temp mantissa1

ROR FACt\_2 ; shift temp mantissa2

ROR FACt\_3 ; shift temp mantissa3

ROR FAC1\_r ; shift temp rounding byte

TYA ; get byte back

LSR ; shift byte

BNE LAB\_262A ; loop if all bits not done

LAB\_264C

RTS

; unpack memory (AY) into FAC2

LAB\_264D

STA ut1\_pl ; save pointer low byte

STY ut1\_ph ; save pointer high byte

LDY #$03 ; 4 bytes to get (0-3)

LDA (ut1\_pl),Y ; get mantissa3

STA FAC2\_3 ; save FAC2 mantissa3

DEY ; decrement index

LDA (ut1\_pl),Y ; get mantissa2

STA FAC2\_2 ; save FAC2 mantissa2

DEY ; decrement index

LDA (ut1\_pl),Y ; get mantissa1+sign

STA FAC2\_s ; save FAC2 sign (b7)

EOR FAC1\_s ; EOR with FAC1 sign (b7)

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

LDA FAC2\_s ; recover FAC2 sign (b7)

ORA #$80 ; set 1xxx xxx (set normal bit)

STA FAC2\_1 ; save FAC2 mantissa1

DEY ; decrement index

LDA (ut1\_pl),Y ; get exponent byte

STA FAC2\_e ; save FAC2 exponent

LDA FAC1\_e ; get FAC1 exponent

RTS

; test and adjust accumulators

LAB\_2673

LDA FAC2\_e ; get FAC2 exponent

LAB\_2675

BEQ LAB\_2696 ; branch if FAC2 = $00 (handle underflow)

CLC ; clear carry for add

ADC FAC1\_e ; add FAC1 exponent

BCC LAB\_2680 ; branch if sum of exponents <$0100

BMI LAB\_269B ; do overflow error

CLC ; clear carry for the add

.byte $2C ; makes next line BIT $1410

LAB\_2680

BPL LAB\_2696 ; if +ve go handle underflow

ADC #$80 ; adjust exponent

STA FAC1\_e ; save FAC1 exponent

BNE LAB\_268B ; branch if not zero

JMP LAB\_24F5 ; save FAC1 sign and return

LAB\_268B

LDA FAC\_sc ; get sign compare (FAC1 EOR FAC2)

STA FAC1\_s ; save FAC1 sign (b7)

LAB\_268F

RTS

; handle overflow and underflow

LAB\_2690

LDA FAC1\_s ; get FAC1 sign (b7)

BPL LAB\_269B ; do overflow error

; handle underflow

LAB\_2696

PLA ; pop return address low byte

PLA ; pop return address high byte

JMP LAB\_24F1 ; clear FAC1 exponent and sign and return

; multiply by 10

LAB\_269E

JSR LAB\_27AB ; round and copy FAC1 to FAC2

TAX ; copy exponent (set the flags)

BEQ LAB\_268F ; exit if zero

CLC ; clear carry for add

ADC #$02 ; add two to exponent (\*4)

BCS LAB\_269B ; do overflow error if > $FF

LDX #$00 ; clear byte

STX FAC\_sc ; clear sign compare (FAC1 EOR FAC2)

JSR LAB\_247C ; add FAC2 to FAC1 (\*5)

INC FAC1\_e ; increment FAC1 exponent (\*10)

BNE LAB\_268F ; if non zero just do RTS

LAB\_269B

JMP LAB\_2564 ; do overflow error and warm start

; divide by 10

LAB\_26B9

JSR LAB\_27AB ; round and copy FAC1 to FAC2

LDA #<LAB\_26B5 ; set pointer to 10d low addr

LDY #>LAB\_26B5 ; set pointer to 10d high addr

LDX #$00 ; clear sign

; divide by (AY) (X=sign)

LAB\_26C2

STX FAC\_sc ; save sign compare (FAC1 EOR FAC2)

JSR LAB\_UFAC ; unpack memory (AY) into FAC1

JMP LAB\_DIVIDE ; do FAC2/FAC1

; Perform divide-by

; convert AY and do (AY)/FAC1

LAB\_26CA

JSR LAB\_264D ; unpack memory (AY) into FAC2

; Perform divide-into

LAB\_DIVIDE

BEQ LAB\_2737 ; if zero go do /0 error

JSR LAB\_27BA ; round FAC1

LDA #$00 ; clear A

SEC ; set carry for subtract

SBC FAC1\_e ; subtract FAC1 exponent (2s complement)

STA FAC1\_e ; save FAC1 exponent

JSR LAB\_2673 ; test and adjust accumulators

INC FAC1\_e ; increment FAC1 exponent

BEQ LAB\_269B ; if zero do overflow error

LDX #$FF ; set index for pre increment

LDA #$01 ; set bit to flag byte save

LAB\_26E4

LDY FAC2\_1 ; get FAC2 mantissa1

CPY FAC1\_1 ; compare FAC1 mantissa1

BNE LAB\_26F4 ; branch if <>

LDY FAC2\_2 ; get FAC2 mantissa2

CPY FAC1\_2 ; compare FAC1 mantissa2

BNE LAB\_26F4 ; branch if <>

LDY FAC2\_3 ; get FAC2 mantissa3

CPY FAC1\_3 ; compare FAC1 mantissa3

LAB\_26F4

PHP ; save FAC2-FAC1 compare status

ROL ; shift the result byte

BCC LAB\_2702 ; if no carry skip the byte save

LDY #$01 ; set bit to flag byte save

INX ; else increment the index to FACt

CPX #$02 ; compare with the index to FACt\_3

BMI LAB\_2701 ; if not last byte just go save it

BNE LAB\_272B ; if all done go save FAC1 rounding byte, normalise and

; return

LDY #$40 ; set bit to flag byte save for the rounding byte

LAB\_2701

STA FACt\_1,X ; write result byte to FACt\_1 + index

TYA ; copy the next save byte flag

LAB\_2702

PLP ; restore FAC2-FAC1 compare status

BCC LAB\_2704 ; if FAC2 < FAC1 then skip the subtract

TAY ; save FAC2-FAC1 compare status

LDA FAC2\_3 ; get FAC2 mantissa3

SBC FAC1\_3 ; subtract FAC1 mantissa3

STA FAC2\_3 ; save FAC2 mantissa3

LDA FAC2\_2 ; get FAC2 mantissa2

SBC FAC1\_2 ; subtract FAC1 mantissa2

STA FAC2\_2 ; save FAC2 mantissa2

LDA FAC2\_1 ; get FAC2 mantissa1

SBC FAC1\_1 ; subtract FAC1 mantissa1

STA FAC2\_1 ; save FAC2 mantissa1

TYA ; restore FAC2-FAC1 compare status

; FAC2 = FAC2\*2

LAB\_2704

ASL FAC2\_3 ; shift FAC2 mantissa3

ROL FAC2\_2 ; shift FAC2 mantissa2

ROL FAC2\_1 ; shift FAC2 mantissa1

BCS LAB\_26F4 ; loop with no compare

BMI LAB\_26E4 ; loop with compare

BPL LAB\_26F4 ; loop always with no compare

; do A<<6, save as FAC1 rounding byte, normalise and return

LAB\_272B

LSR ; shift b1 - b0 ..

ROR ; ..

ROR ; .. to b7 - b6

STA FAC1\_r ; save FAC1 rounding byte

PLP ; dump FAC2-FAC1 compare status

JMP LAB\_273C ; copy temp to FAC1, normalise and return

; do "Divide by zero" error

LAB\_2737

LDX #$14 ; error code $14 ("Divide by zero" error)

JMP LAB\_XERR ; do error #X, then warm start

; copy temp to FAC1 and normalise

LAB\_273C

LDA FACt\_1 ; get temp mantissa1

STA FAC1\_1 ; save FAC1 mantissa1

LDA FACt\_2 ; get temp mantissa2

STA FAC1\_2 ; save FAC1 mantissa2

LDA FACt\_3 ; get temp mantissa3

STA FAC1\_3 ; save FAC1 mantissa3

JMP LAB\_24D5 ; normalise FAC1 and return

; unpack memory (AY) into FAC1

LAB\_UFAC

STA ut1\_pl ; save pointer low byte

STY ut1\_ph ; save pointer high byte

LDY #$03 ; 4 bytes to do

LDA (ut1\_pl),Y ; get last byte

STA FAC1\_3 ; save FAC1 mantissa3

DEY ; decrement index

LDA (ut1\_pl),Y ; get last-1 byte

STA FAC1\_2 ; save FAC1 mantissa2

DEY ; decrement index

LDA (ut1\_pl),Y ; get second byte

STA FAC1\_s ; save FAC1 sign (b7)

ORA #$80 ; set 1xxx xxxx (add normal bit)

STA FAC1\_1 ; save FAC1 mantissa1

DEY ; decrement index

LDA (ut1\_pl),Y ; get first byte (exponent)

STA FAC1\_e ; save FAC1 exponent

STY FAC1\_r ; clear FAC1 rounding byte

RTS

; pack FAC1 into Adatal

LAB\_276E

LDX #<Adatal ; set pointer low byte

LAB\_2770

LDY #>Adatal ; set pointer high byte

BEQ LAB\_2778 ; pack FAC1 into (XY) and return

; pack FAC1 into (Lvarpl)

LAB\_PFAC

LDX Lvarpl ; get destination pointer low byte

LDY Lvarph ; get destination pointer high byte

; pack FAC1 into (XY)

LAB\_2778

JSR LAB\_27BA ; round FAC1

STX ut1\_pl ; save pointer low byte

STY ut1\_ph ; save pointer high byte

LDY #$03 ; set index

LDA FAC1\_3 ; get FAC1 mantissa3

STA (ut1\_pl),Y ; store in destination

DEY ; decrement index

LDA FAC1\_2 ; get FAC1 mantissa2

STA (ut1\_pl),Y ; store in destination

DEY ; decrement index

LDA FAC1\_s ; get FAC1 sign (b7)

ORA #$7F ; set bits x111 1111

AND FAC1\_1 ; AND in FAC1 mantissa1

STA (ut1\_pl),Y ; store in destination

DEY ; decrement index

LDA FAC1\_e ; get FAC1 exponent

STA (ut1\_pl),Y ; store in destination

STY FAC1\_r ; clear FAC1 rounding byte

RTS

; round and copy FAC1 to FAC2

LAB\_27AB

JSR LAB\_27BA ; round FAC1

; copy FAC1 to FAC2

LAB\_27AE

LDX #$05 ; 5 bytes to copy

LAB\_27B0

LDA FAC1\_e-1,X ; get byte from FAC1,X

STA FAC1\_o,X ; save byte at FAC2,X

DEX ; decrement count

BNE LAB\_27B0 ; loop if not all done

STX FAC1\_r ; clear FAC1 rounding byte

LAB\_27B9

RTS

; round FAC1

LAB\_27BA

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_27B9 ; exit if zero

ASL FAC1\_r ; shift FAC1 rounding byte

BCC LAB\_27B9 ; exit if no overflow

; round FAC1 (no check)

LAB\_27C2

JSR LAB\_2559 ; increment FAC1 mantissa

BNE LAB\_27B9 ; branch if no overflow

JMP LAB\_252A ; normalise FAC1 for C=1 and return

; get FAC1 sign

; return A=FF,C=1/-ve A=01,C=0/+ve

LAB\_27CA

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_27D7 ; exit if zero (already correct SGN(0)=0)

; return A=FF,C=1/-ve A=01,C=0/+ve

; no = 0 check

LAB\_27CE

LDA FAC1\_s ; else get FAC1 sign (b7)

; return A=FF,C=1/-ve A=01,C=0/+ve

; no = 0 check, sign in A

LAB\_27D0

ROL ; move sign bit to carry

LDA #$FF ; set byte for -ve result

BCS LAB\_27D7 ; return if sign was set (-ve)

LDA #$01 ; else set byte for +ve result

LAB\_27D7

RTS

; perform SGN()

LAB\_SGN

JSR LAB\_27CA ; get FAC1 sign

; return A=$FF/-ve A=$01/+ve

; save A as integer byte

LAB\_27DB

STA FAC1\_1 ; save FAC1 mantissa1

LDA #$00 ; clear A

STA FAC1\_2 ; clear FAC1 mantissa2

LDX #$88 ; set exponent

; set exp=X, clearFAC1 mantissa3 and normalise

LAB\_27E3

LDA FAC1\_1 ; get FAC1 mantissa1

EOR #$FF ; complement it

ROL ; sign bit into carry

; set exp=X, clearFAC1 mantissa3 and normalise

LAB\_STFA

LDA #$00 ; clear A

STA FAC1\_3 ; clear FAC1 mantissa3

STX FAC1\_e ; set FAC1 exponent

STA FAC1\_r ; clear FAC1 rounding byte

STA FAC1\_s ; clear FAC1 sign (b7)

JMP LAB\_24D0 ; do ABS and normalise FAC1

; perform ABS()

LAB\_ABS

LSR FAC1\_s ; clear FAC1 sign (put zero in b7)

RTS

; compare FAC1 with (AY)

; returns A=$00 if FAC1 = (AY)

; returns A=$01 if FAC1 > (AY)

; returns A=$FF if FAC1 < (AY)

LAB\_27F8

STA ut2\_pl ; save pointer low byte

LAB\_27FA

STY ut2\_ph ; save pointer high byte

LDY #$00 ; clear index

LDA (ut2\_pl),Y ; get exponent

INY ; increment index

TAX ; copy (AY) exponent to X

BEQ LAB\_27CA ; branch if (AY) exponent=0 and get FAC1 sign

; A=FF,C=1/-ve A=01,C=0/+ve

LDA (ut2\_pl),Y ; get (AY) mantissa1 (with sign)

EOR FAC1\_s ; EOR FAC1 sign (b7)

BMI LAB\_27CE ; if signs <> do return A=FF,C=1/-ve

; A=01,C=0/+ve and return

CPX FAC1\_e ; compare (AY) exponent with FAC1 exponent

BNE LAB\_2828 ; branch if different

LDA (ut2\_pl),Y ; get (AY) mantissa1 (with sign)

ORA #$80 ; normalise top bit

CMP FAC1\_1 ; compare with FAC1 mantissa1

BNE LAB\_2828 ; branch if different

INY ; increment index

LDA (ut2\_pl),Y ; get mantissa2

CMP FAC1\_2 ; compare with FAC1 mantissa2

BNE LAB\_2828 ; branch if different

INY ; increment index

LDA #$7F ; set for 1/2 value rounding byte

CMP FAC1\_r ; compare with FAC1 rounding byte (set carry)

LDA (ut2\_pl),Y ; get mantissa3

SBC FAC1\_3 ; subtract FAC1 mantissa3

BEQ LAB\_2850 ; exit if mantissa3 equal

; gets here if number <> FAC1

LAB\_2828

LDA FAC1\_s ; get FAC1 sign (b7)

BCC LAB\_282E ; branch if FAC1 > (AY)

EOR #$FF ; else toggle FAC1 sign

LAB\_282E

JMP LAB\_27D0 ; return A=FF,C=1/-ve A=01,C=0/+ve

; convert FAC1 floating-to-fixed

LAB\_2831

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_287F ; if zero go clear FAC1 and return

SEC ; set carry for subtract

SBC #$98 ; subtract maximum integer range exponent

BIT FAC1\_s ; test FAC1 sign (b7)

BPL LAB\_2845 ; branch if FAC1 +ve

; FAC1 was -ve

TAX ; copy subtracted exponent

LDA #$FF ; overflow for -ve number

STA FAC1\_o ; set FAC1 overflow byte

JSR LAB\_253D ; twos complement FAC1 mantissa

TXA ; restore subtracted exponent

LAB\_2845

LDX #FAC1\_e ; set index to FAC1

CMP #$F9 ; compare exponent result

BPL LAB\_2851 ; if < 8 shifts shift FAC1 A times right and return

JSR LAB\_257B ; shift FAC1 A times right (> 8 shifts)

STY FAC1\_o ; clear FAC1 overflow byte

LAB\_2850

RTS

; shift FAC1 A times right

LAB\_2851

TAY ; copy shift count

LDA FAC1\_s ; get FAC1 sign (b7)

AND #$80 ; mask sign bit only (x000 0000)

LSR FAC1\_1 ; shift FAC1 mantissa1

ORA FAC1\_1 ; OR sign in b7 FAC1 mantissa1

STA FAC1\_1 ; save FAC1 mantissa1

JSR LAB\_2592 ; shift FAC1 Y times right

STY FAC1\_o ; clear FAC1 overflow byte

RTS

; perform INT()

LAB\_INT

LDA FAC1\_e ; get FAC1 exponent

CMP #$98 ; compare with max int

BCS LAB\_2886 ; exit if >= (already int, too big for fractional part!)

JSR LAB\_2831 ; convert FAC1 floating-to-fixed

STY FAC1\_r ; save FAC1 rounding byte

LDA FAC1\_s ; get FAC1 sign (b7)

STY FAC1\_s ; save FAC1 sign (b7)

EOR #$80 ; toggle FAC1 sign

ROL ; shift into carry

LDA #$98 ; set new exponent

STA FAC1\_e ; save FAC1 exponent

LDA FAC1\_3 ; get FAC1 mantissa3

STA Temp3 ; save for EXP() function

JMP LAB\_24D0 ; do ABS and normalise FAC1

; clear FAC1 and return

LAB\_287F

STA FAC1\_1 ; clear FAC1 mantissa1

STA FAC1\_2 ; clear FAC1 mantissa2

STA FAC1\_3 ; clear FAC1 mantissa3

TAY ; clear Y

LAB\_2886

RTS

; get FAC1 from string

; this routine now handles hex and binary values from strings

; starting with "$" and "%" respectively

LAB\_2887

LDY #$00 ; clear Y

STY Dtypef ; clear data type flag, $FF=string, $00=numeric

LDX #$09 ; set index

LAB\_288B

STY numexp,X ; clear byte

DEX ; decrement index

BPL LAB\_288B ; loop until numexp to negnum (and FAC1) = $00

BCC LAB\_28FE ; branch if 1st character numeric

; get FAC1 from string .. first character wasn't numeric

CMP #'-' ; else compare with "-"

BNE LAB\_289A ; branch if not "-"

STX negnum ; set flag for -ve number (X = $FF)

BEQ LAB\_289C ; branch always (go scan and check for hex/bin)

; get FAC1 from string .. first character wasn't numeric or -

LAB\_289A

CMP #'+' ; else compare with "+"

BNE LAB\_289D ; branch if not "+" (go check for hex/bin)

; was "+" or "-" to start, so get next character

LAB\_289C

JSR LAB\_IGBY ; increment and scan memory

BCC LAB\_28FE ; branch if numeric character

; code here for hex and binary numbers

LAB\_289D

CMP #'$' ; else compare with "$"

BNE LAB\_NHEX ; branch if not "$"

JMP LAB\_CHEX ; branch if "$"

LAB\_NHEX

CMP #'%' ; else compare with "%"

BNE LAB\_28A3 ; branch if not "%" (continue original code)

JMP LAB\_CBIN ; branch if "%"

LAB\_289E

JSR LAB\_IGBY ; increment and scan memory (ignore + or get next number)

LAB\_28A1

BCC LAB\_28FE ; branch if numeric character

; get FAC1 from string .. character wasn't numeric, -, +, hex or binary

LAB\_28A3

CMP #'.' ; else compare with "."

BEQ LAB\_28D5 ; branch if "."

; get FAC1 from string .. character wasn't numeric, -, + or .

CMP #'E' ; else compare with "E"

BNE LAB\_28DB ; branch if not "E"

; was "E" so evaluate exponential part

JSR LAB\_IGBY ; increment and scan memory

BCC LAB\_28C7 ; branch if numeric character

CMP #TK\_MINUS ; else compare with token for -

BEQ LAB\_28C2 ; branch if token for -

CMP #'-' ; else compare with "-"

BEQ LAB\_28C2 ; branch if "-"

CMP #TK\_PLUS ; else compare with token for +

BEQ LAB\_28C4 ; branch if token for +

CMP #'+' ; else compare with "+"

BEQ LAB\_28C4 ; branch if "+"

BNE LAB\_28C9 ; branch always

LAB\_28C2

ROR expneg ; set exponent -ve flag (C, which=1, into b7)

LAB\_28C4

JSR LAB\_IGBY ; increment and scan memory

LAB\_28C7

BCC LAB\_2925 ; branch if numeric character

LAB\_28C9

BIT expneg ; test exponent -ve flag

BPL LAB\_28DB ; if +ve go evaluate exponent

; else do exponent = -exponent

LDA #$00 ; clear result

SEC ; set carry for subtract

SBC expcnt ; subtract exponent byte

JMP LAB\_28DD ; go evaluate exponent

LAB\_28D5

ROR numdpf ; set decimal point flag

BIT numdpf ; test decimal point flag

BVC LAB\_289E ; branch if only one decimal point so far

; evaluate exponent

LAB\_28DB

LDA expcnt ; get exponent count byte

LAB\_28DD

SEC ; set carry for subtract

SBC numexp ; subtract numerator exponent

STA expcnt ; save exponent count byte

BEQ LAB\_28F6 ; branch if no adjustment

BPL LAB\_28EF ; else if +ve go do FAC1\*10^expcnt

; else go do FAC1/10^(0-expcnt)

LAB\_28E6

JSR LAB\_26B9 ; divide by 10

INC expcnt ; increment exponent count byte

BNE LAB\_28E6 ; loop until all done

BEQ LAB\_28F6 ; branch always

LAB\_28EF

JSR LAB\_269E ; multiply by 10

DEC expcnt ; decrement exponent count byte

BNE LAB\_28EF ; loop until all done

LAB\_28F6

LDA negnum ; get -ve flag

BMI LAB\_28FB ; if -ve do - FAC1 and return

RTS

; do - FAC1 and return

LAB\_28FB

JMP LAB\_GTHAN ; do - FAC1 and return

; do unsigned FAC1\*10+number

LAB\_28FE

PHA ; save character

BIT numdpf ; test decimal point flag

BPL LAB\_2905 ; skip exponent increment if not set

INC numexp ; else increment number exponent

LAB\_2905

JSR LAB\_269E ; multiply FAC1 by 10

PLA ; restore character

AND #$0F ; convert to binary

JSR LAB\_2912 ; evaluate new ASCII digit

JMP LAB\_289E ; go do next character

; evaluate new ASCII digit

LAB\_2912

PHA ; save digit

JSR LAB\_27AB ; round and copy FAC1 to FAC2

PLA ; restore digit

JSR LAB\_27DB ; save A as integer byte

LDA FAC2\_s ; get FAC2 sign (b7)

EOR FAC1\_s ; toggle with FAC1 sign (b7)

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

LDX FAC1\_e ; get FAC1 exponent

JMP LAB\_ADD ; add FAC2 to FAC1 and return

; evaluate next character of exponential part of number

LAB\_2925

LDA expcnt ; get exponent count byte

CMP #$0A ; compare with 10 decimal

BCC LAB\_2934 ; branch if less

LDA #$64 ; make all -ve exponents = -100 decimal (causes underflow)

BIT expneg ; test exponent -ve flag

BMI LAB\_2942 ; branch if -ve

JMP LAB\_2564 ; else do overflow error

LAB\_2934

ASL ; \* 2

ASL ; \* 4

ADC expcnt ; \* 5

ASL ; \* 10

LDY #$00 ; set index

ADC (Bpntrl),Y ; add character (will be $30 too much!)

SBC #'0'-1 ; convert character to binary

LAB\_2942

STA expcnt ; save exponent count byte

JMP LAB\_28C4 ; go get next character

; print " in line [LINE #]"

LAB\_2953

LDA #<LAB\_LMSG ; point to " in line " message low byte

LDY #>LAB\_LMSG ; point to " in line " message high byte

JSR LAB\_18C3 ; print null terminated string from memory

; print Basic line #

LDA Clineh ; get current line high byte

LDX Clinel ; get current line low byte

; print XA as unsigned integer

LAB\_295E

STA FAC1\_1 ; save low byte as FAC1 mantissa1

STX FAC1\_2 ; save high byte as FAC1 mantissa2

LDX #$90 ; set exponent to 16d bits

SEC ; set integer is +ve flag

JSR LAB\_STFA ; set exp=X, clearFAC1 mantissa3 and normalise

LDY #$00 ; clear index

TYA ; clear A

JSR LAB\_297B ; convert FAC1 to string, skip sign character save

JMP LAB\_18C3 ; print null terminated string from memory and return

; convert FAC1 to ASCII string result in (AY)

; not any more, moved scratchpad to page 0

LAB\_296E

LDY #$01 ; set index = 1

LDA #$20 ; character = " " (assume +ve)

BIT FAC1\_s ; test FAC1 sign (b7)

BPL LAB\_2978 ; branch if +ve

LDA #$2D ; else character = "-"

LAB\_2978

STA Decss,Y ; save leading character (" " or "-")

LAB\_297B

STA FAC1\_s ; clear FAC1 sign (b7)

STY Sendl ; save index

INY ; increment index

LDX FAC1\_e ; get FAC1 exponent

BNE LAB\_2989 ; branch if FAC1<>0

; exponent was $00 so FAC1 is 0

LDA #'0' ; set character = "0"

JMP LAB\_2A89 ; save last character, [EOT] and exit

; FAC1 is some non zero value

LAB\_2989

LDA #$00 ; clear (number exponent count)

CPX #$81 ; compare FAC1 exponent with $81 (>1.00000)

BCS LAB\_299A ; branch if FAC1=>1

; FAC1<1

LDA #<LAB\_294F ; set pointer low byte to 1,000,000

LDY #>LAB\_294F ; set pointer high byte to 1,000,000

JSR LAB\_25FB ; do convert AY, FCA1\*(AY)

LDA #$FA ; set number exponent count (-6)

LAB\_299A

STA numexp ; save number exponent count

LAB\_299C

LDA #<LAB\_294B ; set pointer low byte to 999999.4375 (max before sci note)

LDY #>LAB\_294B ; set pointer high byte to 999999.4375

JSR LAB\_27F8 ; compare FAC1 with (AY)

BEQ LAB\_29C3 ; exit if FAC1 = (AY)

BPL LAB\_29B9 ; go do /10 if FAC1 > (AY)

; FAC1 < (AY)

LAB\_29A7

LDA #<LAB\_2947 ; set pointer low byte to 99999.9375

LDY #>LAB\_2947 ; set pointer high byte to 99999.9375

JSR LAB\_27F8 ; compare FAC1 with (AY)

BEQ LAB\_29B2 ; branch if FAC1 = (AY) (allow decimal places)

BPL LAB\_29C0 ; branch if FAC1 > (AY) (no decimal places)

; FAC1 <= (AY)

LAB\_29B2

JSR LAB\_269E ; multiply by 10

DEC numexp ; decrement number exponent count

BNE LAB\_29A7 ; go test again (branch always)

LAB\_29B9

JSR LAB\_26B9 ; divide by 10

INC numexp ; increment number exponent count

BNE LAB\_299C ; go test again (branch always)

; now we have just the digits to do

LAB\_29C0

JSR LAB\_244E ; add 0.5 to FAC1 (round FAC1)

LAB\_29C3

JSR LAB\_2831 ; convert FAC1 floating-to-fixed

LDX #$01 ; set default digits before dp = 1

LDA numexp ; get number exponent count

CLC ; clear carry for add

ADC #$07 ; up to 6 digits before point

BMI LAB\_29D8 ; if -ve then 1 digit before dp

CMP #$08 ; A>=8 if n>=1E6

BCS LAB\_29D9 ; branch if >= $08

; carry is clear

ADC #$FF ; take 1 from digit count

TAX ; copy to A

LDA #$02 ;.set exponent adjust

LAB\_29D8

SEC ; set carry for subtract

LAB\_29D9

SBC #$02 ; -2

STA expcnt ;.save exponent adjust

STX numexp ; save digits before dp count

TXA ; copy to A

BEQ LAB\_29E4 ; branch if no digits before dp

BPL LAB\_29F7 ; branch if digits before dp

LAB\_29E4

LDY Sendl ; get output string index

LDA #$2E ; character "."

INY ; increment index

STA Decss,Y ; save to output string

TXA ;.

BEQ LAB\_29F5 ;.

LDA #'0' ; character "0"

INY ; increment index

STA Decss,Y ; save to output string

LAB\_29F5

STY Sendl ; save output string index

LAB\_29F7

LDY #$00 ; clear index (point to 100,000)

LDX #$80 ;

LAB\_29FB

LDA FAC1\_3 ; get FAC1 mantissa3

CLC ; clear carry for add

ADC LAB\_2A9C,Y ; add -ve LSB

STA FAC1\_3 ; save FAC1 mantissa3

LDA FAC1\_2 ; get FAC1 mantissa2

ADC LAB\_2A9B,Y ; add -ve NMSB

STA FAC1\_2 ; save FAC1 mantissa2

LDA FAC1\_1 ; get FAC1 mantissa1

ADC LAB\_2A9A,Y ; add -ve MSB

STA FAC1\_1 ; save FAC1 mantissa1

INX ;

BCS LAB\_2A18 ;

BPL LAB\_29FB ; not -ve so try again

BMI LAB\_2A1A ;

LAB\_2A18

BMI LAB\_29FB ;

LAB\_2A1A

TXA ;

BCC LAB\_2A21 ;

EOR #$FF ;

ADC #$0A ;

LAB\_2A21

ADC #'0'-1 ; add "0"-1 to result

INY ; increment index ..

INY ; .. to next less ..

INY ; .. power of ten

STY Cvaral ; save as current var address low byte

LDY Sendl ; get output string index

INY ; increment output string index

TAX ; copy character to X

AND #$7F ; mask out top bit

STA Decss,Y ; save to output string

DEC numexp ; decrement # of characters before the dp

BNE LAB\_2A3B ; branch if still characters to do

; else output the point

LDA #$2E ; character "."

INY ; increment output string index

STA Decss,Y ; save to output string

LAB\_2A3B

STY Sendl ; save output string index

LDY Cvaral ; get current var address low byte

TXA ; get character back

EOR #$FF ;

AND #$80 ;

TAX ;

CPY #$12 ; compare index with max

BNE LAB\_29FB ; loop if not max

; now remove trailing zeroes

LDY Sendl ; get output string index

LAB\_2A4B

LDA Decss,Y ; get character from output string

DEY ; decrement output string index

CMP #'0' ; compare with "0"

BEQ LAB\_2A4B ; loop until non "0" character found

CMP #'.' ; compare with "."

BEQ LAB\_2A58 ; branch if was dp

; restore last character

INY ; increment output string index

LAB\_2A58

LDA #$2B ; character "+"

LDX expcnt ; get exponent count

BEQ LAB\_2A8C ; if zero go set null terminator and exit

; exponent isn't zero so write exponent

BPL LAB\_2A68 ; branch if exponent count +ve

LDA #$00 ; clear A

SEC ; set carry for subtract

SBC expcnt ; subtract exponent count adjust (convert -ve to +ve)

TAX ; copy exponent count to X

LDA #'-' ; character "-"

LAB\_2A68

STA Decss+2,Y ; save to output string

LDA #$45 ; character "E"

STA Decss+1,Y ; save exponent sign to output string

TXA ; get exponent count back

LDX #'0'-1 ; one less than "0" character

SEC ; set carry for subtract

LAB\_2A74

INX ; increment 10's character

SBC #$0A ;.subtract 10 from exponent count

BCS LAB\_2A74 ; loop while still >= 0

ADC #':' ; add character ":" ($30+$0A, result is 10 less that value)

STA Decss+4,Y ; save to output string

TXA ; copy 10's character

STA Decss+3,Y ; save to output string

LDA #$00 ; set null terminator

STA Decss+5,Y ; save to output string

BEQ LAB\_2A91 ; go set string pointer (AY) and exit (branch always)

; save last character, [EOT] and exit

LAB\_2A89

STA Decss,Y ; save last character to output string

; set null terminator and exit

LAB\_2A8C

LDA #$00 ; set null terminator

STA Decss+1,Y ; save after last character

; set string pointer (AY) and exit

LAB\_2A91

LDA #<Decssp1 ; set result string low pointer

LDY #>Decssp1 ; set result string high pointer

RTS

; perform power function

LAB\_POWER

BEQ LAB\_EXP ; go do EXP()

LDA FAC2\_e ; get FAC2 exponent

BNE LAB\_2ABF ; branch if FAC2<>0

JMP LAB\_24F3 ; clear FAC1 exponent and sign and return

LAB\_2ABF

LDX #<func\_l ; set destination pointer low byte

LDY #>func\_l ; set destination pointer high byte

JSR LAB\_2778 ; pack FAC1 into (XY)

LDA FAC2\_s ; get FAC2 sign (b7)

BPL LAB\_2AD9 ; branch if FAC2>0

; else FAC2 is -ve and can only be raised to an

; integer power which gives an x +j0 result

JSR LAB\_INT ; perform INT

LDA #<func\_l ; set source pointer low byte

LDY #>func\_l ; set source pointer high byte

JSR LAB\_27F8 ; compare FAC1 with (AY)

BNE LAB\_2AD9 ; branch if FAC1 <> (AY) to allow Function Call error

; this will leave FAC1 -ve and cause a Function Call

; error when LOG() is called

TYA ; clear sign b7

LDY Temp3 ; save mantissa 3 from INT() function as sign in Y

; for possible later negation, b0

LAB\_2AD9

JSR LAB\_279D ; save FAC1 sign and copy ABS(FAC2) to FAC1

TYA ; copy sign back ..

PHA ; .. and save it

JSR LAB\_LOG ; do LOG(n)

LDA #<garb\_l ; set pointer low byte

LDY #>garb\_l ; set pointer high byte

JSR LAB\_25FB ; do convert AY, FCA1\*(AY) (square the value)

JSR LAB\_EXP ; go do EXP(n)

PLA ; pull sign from stack

LSR ; b0 is to be tested, shift to Cb

BCC LAB\_2AF9 ; if no bit then exit

; Perform negation

; do - FAC1

LAB\_GTHAN

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_2AF9 ; exit if FAC1\_e = $00

LDA FAC1\_s ; get FAC1 sign (b7)

EOR #$FF ; complement it

STA FAC1\_s ; save FAC1 sign (b7)

LAB\_2AF9

RTS

; perform EXP() (x^e)

LAB\_EXP

LDA #<LAB\_2AFA ; set 1.443 pointer low byte

LDY #>LAB\_2AFA ; set 1.443 pointer high byte

JSR LAB\_25FB ; do convert AY, FCA1\*(AY)

LDA FAC1\_r ; get FAC1 rounding byte

ADC #$50 ; +$50/$100

BCC LAB\_2B2B ; skip rounding if no carry

JSR LAB\_27C2 ; round FAC1 (no check)

LAB\_2B2B

STA FAC2\_r ; save FAC2 rounding byte

JSR LAB\_27AE ; copy FAC1 to FAC2

LDA FAC1\_e ; get FAC1 exponent

CMP #$88 ; compare with EXP limit (256d)

BCC LAB\_2B39 ; branch if less

LAB\_2B36

JSR LAB\_2690 ; handle overflow and underflow

LAB\_2B39

JSR LAB\_INT ; perform INT

LDA Temp3 ; get mantissa 3 from INT() function

CLC ; clear carry for add

ADC #$81 ; normalise +1

BEQ LAB\_2B36 ; if $00 go handle overflow

SEC ; set carry for subtract

SBC #$01 ; now correct for exponent

PHA ; save FAC2 exponent

; swap FAC1 and FAC2

LDX #$04 ; 4 bytes to do

LAB\_2B49

LDA FAC2\_e,X ; get FAC2,X

LDY FAC1\_e,X ; get FAC1,X

STA FAC1\_e,X ; save FAC1,X

STY FAC2\_e,X ; save FAC2,X

DEX ; decrement count/index

BPL LAB\_2B49 ; loop if not all done

LDA FAC2\_r ; get FAC2 rounding byte

STA FAC1\_r ; save as FAC1 rounding byte

JSR LAB\_SUBTRACT ; perform subtraction, FAC2 from FAC1

JSR LAB\_GTHAN ; do - FAC1

LDA #<LAB\_2AFE ; set counter pointer low byte

LDY #>LAB\_2AFE ; set counter pointer high byte

JSR LAB\_2B84 ; go do series evaluation

LDA #$00 ; clear A

STA FAC\_sc ; clear sign compare (FAC1 EOR FAC2)

PLA ;.get saved FAC2 exponent

JMP LAB\_2675 ; test and adjust accumulators and return

; ^2 then series evaluation

LAB\_2B6E

STA Cptrl ; save count pointer low byte

STY Cptrh ; save count pointer high byte

JSR LAB\_276E ; pack FAC1 into Adatal

LDA #<Adatal ; set pointer low byte (Y already $00)

JSR LAB\_25FB ; do convert AY, FCA1\*(AY)

JSR LAB\_2B88 ; go do series evaluation

LDA #<Adatal ; pointer to original # low byte

LDY #>Adatal ; pointer to original # high byte

JMP LAB\_25FB ; do convert AY, FCA1\*(AY) and return

; series evaluation

LAB\_2B84

STA Cptrl ; save count pointer low byte

STY Cptrh ; save count pointer high byte

LAB\_2B88

LDX #<numexp ; set pointer low byte

JSR LAB\_2770 ; set pointer high byte and pack FAC1 into numexp

LDA (Cptrl),Y ; get constants count

STA numcon ; save constants count

LDY Cptrl ; get count pointer low byte

INY ; increment it (now constants pointer)

TYA ; copy it

BNE LAB\_2B97 ; skip next if no overflow

INC Cptrh ; else increment high byte

LAB\_2B97

STA Cptrl ; save low byte

LDY Cptrh ; get high byte

LAB\_2B9B

JSR LAB\_25FB ; do convert AY, FCA1\*(AY)

LDA Cptrl ; get constants pointer low byte

LDY Cptrh ; get constants pointer high byte

CLC ; clear carry for add

ADC #$04 ; +4 to low pointer (4 bytes per constant)

BCC LAB\_2BA8 ; skip next if no overflow

INY ; increment high byte

LAB\_2BA8

STA Cptrl ; save pointer low byte

STY Cptrh ; save pointer high byte

JSR LAB\_246C ; add (AY) to FAC1

LDA #<numexp ; set pointer low byte to partial @ numexp

LDY #>numexp ; set pointer high byte to partial @ numexp

DEC numcon ; decrement constants count

BNE LAB\_2B9B ; loop until all done

RTS

; RND(n), 32 bit Galoise version. make n=0 for 19th next number in sequence or n<>0

; to get 19th next number in sequence after seed n. This version of the PRNG uses

; the Galois method and a sample of 65536 bytes produced gives the following values.

; Entropy = 7.997442 bits per byte

; Optimum compression would reduce these 65536 bytes by 0 percent

; Chi square distribution for 65536 samples is 232.01, and

; randomly would exceed this value 75.00 percent of the time

; Arithmetic mean value of data bytes is 127.6724, 127.5 would be random

; Monte Carlo value for Pi is 3.122871269, error 0.60 percent

; Serial correlation coefficient is -0.000370, totally uncorrelated would be 0.0

LAB\_RND

LDA FAC1\_e ; get FAC1 exponent

BEQ NextPRN ; do next random # if zero

; else get seed into random number store

LDX #Rbyte4 ; set PRNG pointer low byte

LDY #$00 ; set PRNG pointer high byte

JSR LAB\_2778 ; pack FAC1 into (XY)

NextPRN

LDX #$AF ; set EOR byte

LDY #$13 ; do this nineteen times

LoopPRN

ASL Rbyte1 ; shift PRNG most significant byte

ROL Rbyte2 ; shift PRNG middle byte

ROL Rbyte3 ; shift PRNG least significant byte

ROL Rbyte4 ; shift PRNG extra byte

BCC Ninc1 ; branch if bit 32 clear

TXA ; set EOR byte

EOR Rbyte1 ; EOR PRNG extra byte

STA Rbyte1 ; save new PRNG extra byte

Ninc1

DEY ; decrement loop count

BNE LoopPRN ; loop if not all done

LDX #$02 ; three bytes to copy

CopyPRNG

LDA Rbyte1,X ; get PRNG byte

STA FAC1\_1,X ; save FAC1 byte

DEX

BPL CopyPRNG ; loop if not complete

LDA #$80 ; set the exponent

STA FAC1\_e ; save FAC1 exponent

ASL ; clear A

STA FAC1\_s ; save FAC1 sign

JMP LAB\_24D5 ; normalise FAC1 and return

; perform COS()

LAB\_COS

LDA #<LAB\_2C78 ; set (pi/2) pointer low byte

LDY #>LAB\_2C78 ; set (pi/2) pointer high byte

JSR LAB\_246C ; add (AY) to FAC1

; perform SIN()

LAB\_SIN

JSR LAB\_27AB ; round and copy FAC1 to FAC2

LDA #<LAB\_2C7C ; set (2\*pi) pointer low byte

LDY #>LAB\_2C7C ; set (2\*pi) pointer high byte

LDX FAC2\_s ; get FAC2 sign (b7)

JSR LAB\_26C2 ; divide by (AY) (X=sign)

JSR LAB\_27AB ; round and copy FAC1 to FAC2

JSR LAB\_INT ; perform INT

LDA #$00 ; clear byte

STA FAC\_sc ; clear sign compare (FAC1 EOR FAC2)

JSR LAB\_SUBTRACT ; perform subtraction, FAC2 from FAC1

LDA #<LAB\_2C80 ; set 0.25 pointer low byte

LDY #>LAB\_2C80 ; set 0.25 pointer high byte

JSR LAB\_2455 ; perform subtraction, (AY) from FAC1

LDA FAC1\_s ; get FAC1 sign (b7)

PHA ; save FAC1 sign

BPL LAB\_2C35 ; branch if +ve

; FAC1 sign was -ve

JSR LAB\_244E ; add 0.5 to FAC1

LDA FAC1\_s ; get FAC1 sign (b7)

BMI LAB\_2C38 ; branch if -ve

LDA Cflag ; get comparison evaluation flag

EOR #$FF ; toggle flag

STA Cflag ; save comparison evaluation flag

LAB\_2C35

JSR LAB\_GTHAN ; do - FAC1

LAB\_2C38

LDA #<LAB\_2C80 ; set 0.25 pointer low byte

LDY #>LAB\_2C80 ; set 0.25 pointer high byte

JSR LAB\_246C ; add (AY) to FAC1

PLA ; restore FAC1 sign

BPL LAB\_2C45 ; branch if was +ve

; else correct FAC1

JSR LAB\_GTHAN ; do - FAC1

LAB\_2C45

LDA #<LAB\_2C84 ; set pointer low byte to counter

LDY #>LAB\_2C84 ; set pointer high byte to counter

JMP LAB\_2B6E ; ^2 then series evaluation and return

; perform TAN()

LAB\_TAN

JSR LAB\_276E ; pack FAC1 into Adatal

LDA #$00 ; clear byte

STA Cflag ; clear comparison evaluation flag

JSR LAB\_SIN ; go do SIN(n)

LDX #<func\_l ; set sin(n) pointer low byte

LDY #>func\_l ; set sin(n) pointer high byte

JSR LAB\_2778 ; pack FAC1 into (XY)

LDA #<Adatal ; set n pointer low addr

LDY #>Adatal ; set n pointer high addr

JSR LAB\_UFAC ; unpack memory (AY) into FAC1

LDA #$00 ; clear byte

STA FAC1\_s ; clear FAC1 sign (b7)

LDA Cflag ; get comparison evaluation flag

JSR LAB\_2C74 ; save flag and go do series evaluation

LDA #<func\_l ; set sin(n) pointer low byte

LDY #>func\_l ; set sin(n) pointer high byte

JMP LAB\_26CA ; convert AY and do (AY)/FAC1

LAB\_2C74

PHA ; save comparison evaluation flag

JMP LAB\_2C35 ; go do series evaluation

; perform USR()

LAB\_USR

JSR Usrjmp ; call user code

JMP LAB\_1BFB ; scan for ")", else do syntax error then warm start

; perform ATN()

LAB\_ATN

LDA FAC1\_s ; get FAC1 sign (b7)

PHA ; save sign

BPL LAB\_2CA1 ; branch if +ve

JSR LAB\_GTHAN ; else do - FAC1

LAB\_2CA1

LDA FAC1\_e ; get FAC1 exponent

PHA ; push exponent

CMP #$81 ; compare with 1

BCC LAB\_2CAF ; branch if FAC1<1

LDA #<LAB\_259C ; set 1 pointer low byte

LDY #>LAB\_259C ; set 1 pointer high byte

JSR LAB\_26CA ; convert AY and do (AY)/FAC1

LAB\_2CAF

LDA #<LAB\_2CC9 ; set pointer low byte to counter

LDY #>LAB\_2CC9 ; set pointer high byte to counter

JSR LAB\_2B6E ; ^2 then series evaluation

PLA ; restore old FAC1 exponent

CMP #$81 ; compare with 1

BCC LAB\_2CC2 ; branch if FAC1<1

LDA #<LAB\_2C78 ; set (pi/2) pointer low byte

LDY #>LAB\_2C78 ; set (pi/2) pointer high byte

JSR LAB\_2455 ; perform subtraction, (AY) from FAC1

LAB\_2CC2

PLA ; restore FAC1 sign

BPL LAB\_2D04 ; exit if was +ve

JMP LAB\_GTHAN ; else do - FAC1 and return

; perform BITSET

LAB\_BITSET

JSR LAB\_GADB ; get two parameters for POKE or WAIT

CPX #$08 ; only 0 to 7 are allowed

BCS FCError ; branch if > 7

LDA #$00 ; clear A

SEC ; set the carry

S\_Bits

ROL ; shift bit

DEX ; decrement bit number

BPL S\_Bits ; loop if still +ve

INX ; make X = $00

ORA (Itempl,X) ; or with byte via temporary integer (addr)

STA (Itempl,X) ; save byte via temporary integer (addr)

LAB\_2D04

RTS

; perform BITCLR

LAB\_BITCLR

JSR LAB\_GADB ; get two parameters for POKE or WAIT

CPX #$08 ; only 0 to 7 are allowed

BCS FCError ; branch if > 7

LDA #$FF ; set A

S\_Bitc

ROL ; shift bit

DEX ; decrement bit number

BPL S\_Bitc ; loop if still +ve

INX ; make X = $00

AND (Itempl,X) ; and with byte via temporary integer (addr)

STA (Itempl,X) ; save byte via temporary integer (addr)

RTS

FCError

JMP LAB\_FCER ; do function call error then warm start

; perform BITTST()

LAB\_BTST

JSR LAB\_IGBY ; increment BASIC pointer

JSR LAB\_GADB ; get two parameters for POKE or WAIT

CPX #$08 ; only 0 to 7 are allowed

BCS FCError ; branch if > 7

JSR LAB\_GBYT ; get next BASIC byte

CMP #')' ; is next character ")"

BEQ TST\_OK ; if ")" go do rest of function

JMP LAB\_SNER ; do syntax error then warm start

TST\_OK

JSR LAB\_IGBY ; update BASIC execute pointer (to character past ")")

LDA #$00 ; clear A

SEC ; set the carry

T\_Bits

ROL ; shift bit

DEX ; decrement bit number

BPL T\_Bits ; loop if still +ve

INX ; make X = $00

AND (Itempl,X) ; AND with byte via temporary integer (addr)

BEQ LAB\_NOTT ; branch if zero (already correct)

LDA #$FF ; set for -1 result

LAB\_NOTT

JMP LAB\_27DB ; go do SGN tail

; perform BIN$()

LAB\_BINS

CPX #$19 ; max + 1

BCS BinFErr ; exit if too big ( > or = )

STX TempB ; save # of characters ($00 = leading zero remove)

LDA #$18 ; need A byte long space

JSR LAB\_MSSP ; make string space A bytes long

LDY #$17 ; set index

LDX #$18 ; character count

NextB1

LSR nums\_1 ; shift highest byte

ROR nums\_2 ; shift middle byte

ROR nums\_3 ; shift lowest byte bit 0 to carry

TXA ; load with "0"/2

ROL ; shift in carry

STA (str\_pl),Y ; save to temp string + index

DEY ; decrement index

BPL NextB1 ; loop if not done

LDA TempB ; get # of characters

BEQ EndBHS ; branch if truncate

TAX ; copy length to X

SEC ; set carry for add !

EOR #$FF ; 1's complement

ADC #$18 ; add 24d

BEQ GoPr2 ; if zero print whole string

BNE GoPr1 ; else go make output string

; this is the exit code and is also used by HEX$()

; truncate string to remove leading "0"s

EndBHS

TAY ; clear index (A=0, X=length here)

NextB2

LDA (str\_pl),Y ; get character from string

CMP #'0' ; compare with "0"

BNE GoPr ; if not "0" then go print string from here

DEX ; decrement character count

BEQ GoPr3 ; if zero then end of string so go print it

INY ; else increment index

BPL NextB2 ; loop always

; make fixed length output string - ignore overflows!

GoPr3

INX ; need at least 1 character

GoPr

TYA ; copy result

GoPr1

CLC ; clear carry for add

ADC str\_pl ; add low address

STA str\_pl ; save low address

LDA #$00 ; do high byte

ADC str\_ph ; add high address

STA str\_ph ; save high address

GoPr2

STX str\_ln ; X holds string length

JSR LAB\_IGBY ; update BASIC execute pointer (to character past ")")

JMP LAB\_RTST ; check for space on descriptor stack then put address

; and length on descriptor stack and update stack pointers

BinFErr

JMP LAB\_FCER ; do function call error then warm start

; perform HEX$()

LAB\_HEXS

CPX #$07 ; max + 1

BCS BinFErr ; exit if too big ( > or = )

STX TempB ; save # of characters

LDA #$06 ; need 6 bytes for string

JSR LAB\_MSSP ; make string space A bytes long

LDY #$05 ; set string index

SED ; need decimal mode for nibble convert

LDA nums\_3 ; get lowest byte

JSR LAB\_A2HX ; convert A to ASCII hex byte and output

LDA nums\_2 ; get middle byte

JSR LAB\_A2HX ; convert A to ASCII hex byte and output

LDA nums\_1 ; get highest byte

JSR LAB\_A2HX ; convert A to ASCII hex byte and output

CLD ; back to binary

LDX #$06 ; character count

LDA TempB ; get # of characters

BEQ EndBHS ; branch if truncate

TAX ; copy length to X

SEC ; set carry for add !

EOR #$FF ; 1's complement

ADC #$06 ; add 6d

BEQ GoPr2 ; if zero print whole string

BNE GoPr1 ; else go make output string (branch always)

; convert A to ASCII hex byte and output .. note set decimal mode before calling

LAB\_A2HX

TAX ; save byte

AND #$0F ; mask off top bits

JSR LAB\_AL2X ; convert low nibble to ASCII and output

TXA ; get byte back

LSR ; /2 shift high nibble to low nibble

LSR ; /4

LSR ; /8

LSR ; /16

LAB\_AL2X

CMP #$0A ; set carry for +1 if >9

ADC #'0' ; add ASCII "0"

STA (str\_pl),Y ; save to temp string

DEY ; decrement counter

RTS

LAB\_NLTO

STA FAC1\_e ; save FAC1 exponent

LDA #$00 ; clear sign compare

LAB\_MLTE

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

TXA ; restore character

JSR LAB\_2912 ; evaluate new ASCII digit

; gets here if the first character was "$" for hex

; get hex number

LAB\_CHEX

JSR LAB\_IGBY ; increment and scan memory

BCC LAB\_ISHN ; branch if numeric character

ORA #$20 ; case convert, allow "A" to "F" and "a" to "f"

SBC #'a' ; subtract "a" (carry set here)

CMP #$06 ; compare normalised with $06 (max+1)

BCS LAB\_EXCH ; exit if >"f" or <"0"

ADC #$0A ; convert to nibble

LAB\_ISHN

AND #$0F ; convert to binary

TAX ; save nibble

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_MLTE ; skip multiply if zero

ADC #$04 ; add four to exponent (\*16 - carry clear here)

BCC LAB\_NLTO ; if no overflow do evaluate digit

LAB\_MLTO

JMP LAB\_2564 ; do overflow error and warm start

LAB\_NXCH

TAX ; save bit

LDA FAC1\_e ; get FAC1 exponent

BEQ LAB\_MLBT ; skip multiply if zero

INC FAC1\_e ; increment FAC1 exponent (\*2)

BEQ LAB\_MLTO ; do overflow error if = $00

LDA #$00 ; clear sign compare

LAB\_MLBT

STA FAC\_sc ; save sign compare (FAC1 EOR FAC2)

TXA ; restore bit

JSR LAB\_2912 ; evaluate new ASCII digit

; gets here if the first character was "%" for binary

; get binary number

LAB\_CBIN

JSR LAB\_IGBY ; increment and scan memory

EOR #'0' ; convert "0" to 0 etc.

CMP #$02 ; compare with max+1

BCC LAB\_NXCH ; branch exit if < 2

LAB\_EXCH

JMP LAB\_28F6 ; evaluate -ve flag and return

; ctrl-c check routine. includes limited "life" byte save for INGET routine

; now also the code that checks to see if an interrupt has occurred

CTRLC

LDA ccflag ; get [CTRL-C] check flag

BNE LAB\_FBA2 ; exit if inhibited

JSR V\_INPT ; scan input device

BCC LAB\_FBA0 ; exit if buffer empty

STA ccbyte ; save received byte

LDX #$20 ; "life" timer for bytes

STX ccnull ; set countdown

JMP LAB\_1636 ; return to BASIC

LAB\_FBA0

LDX ccnull ; get countdown byte

BEQ LAB\_FBA2 ; exit if finished

DEC ccnull ; else decrement countdown

LAB\_FBA2

LDX #NmiBase ; set pointer to NMI values

JSR LAB\_CKIN ; go check interrupt

LDX #IrqBase ; set pointer to IRQ values

JSR LAB\_CKIN ; go check interrupt

LAB\_CRTS

RTS

; check whichever interrupt is indexed by X

LAB\_CKIN

LDA PLUS\_0,X ; get interrupt flag byte

BPL LAB\_CRTS ; branch if interrupt not enabled

; we disable the interrupt here and make two new commands RETIRQ and RETNMI to

; automatically enable the interrupt when we exit

ASL ; move happened bit to setup bit

AND #$40 ; mask happened bits

BEQ LAB\_CRTS ; if no interrupt then exit

STA PLUS\_0,X ; save interrupt flag byte

TXA ; copy index ..

TAY ; .. to Y

PLA ; dump return address low byte, call from CTRL-C

PLA ; dump return address high byte

LDA #$05 ; need 5 bytes for GOSUB

JSR LAB\_1212 ; check room on stack for A bytes

LDA Bpntrh ; get BASIC execute pointer high byte

PHA ; push on stack

LDA Bpntrl ; get BASIC execute pointer low byte

PHA ; push on stack

LDA Clineh ; get current line high byte

PHA ; push on stack

LDA Clinel ; get current line low byte

PHA ; push on stack

LDA #TK\_GOSUB ; token for GOSUB

PHA ; push on stack

LDA PLUS\_1,Y ; get interrupt code pointer low byte

STA Bpntrl ; save as BASIC execute pointer low byte

LDA PLUS\_2,Y ; get interrupt code pointer high byte

STA Bpntrh ; save as BASIC execute pointer high byte

JMP LAB\_15C2 ; go do interpreter inner loop

; can't RTS, we used the stack! the RTS from the ctrl-c

; check will be taken when the RETIRQ/RETNMI/RETURN is

; executed at the end of the subroutine

; get byte from input device, no waiting

; returns with carry set if byte in A

INGET

JSR V\_INPT ; call scan input device

BCS LAB\_FB95 ; if byte go reset timer

LDA ccnull ; get countdown

BEQ LAB\_FB96 ; exit if empty

LDA ccbyte ; get last received byte

SEC ; flag we got a byte

LAB\_FB95

LDX #$00 ; clear X

STX ccnull ; clear timer because we got a byte

LAB\_FB96

RTS

; these routines only enable the interrupts if the set-up flag is set

; if not they have no effect

; perform IRQ {ON|OFF|CLEAR}

LAB\_IRQ

LDX #IrqBase ; set pointer to IRQ values

.byte $2C ; make next line BIT abs.

; perform NMI {ON|OFF|CLEAR}

LAB\_NMI

LDX #NmiBase ; set pointer to NMI values

CMP #TK\_ON ; compare with token for ON

BEQ LAB\_INON ; go turn on interrupt

CMP #TK\_OFF ; compare with token for OFF

BEQ LAB\_IOFF ; go turn off interrupt

EOR #TK\_CLEAR ; compare with token for CLEAR, A = $00 if = TK\_CLEAR

BEQ LAB\_INEX ; go clear interrupt flags and return

JMP LAB\_SNER ; do syntax error then warm start

LAB\_IOFF

LDA #$7F ; clear A

AND PLUS\_0,X ; AND with interrupt setup flag

BPL LAB\_INEX ; go clear interrupt enabled flag and return

LAB\_INON

LDA PLUS\_0,X ; get interrupt setup flag

ASL ; Shift bit to enabled flag

ORA PLUS\_0,X ; OR with flag byte

LAB\_INEX

STA PLUS\_0,X ; save interrupt flag byte

JMP LAB\_IGBY ; update BASIC execute pointer and return

; these routines set up the pointers and flags for the interrupt routines

; note that the interrupts are also enabled by these commands

; perform ON IRQ

LAB\_SIRQ

CLI ; enable interrupts

LDX #IrqBase ; set pointer to IRQ values

.byte $2C ; make next line BIT abs.

; perform ON NMI

LAB\_SNMI

LDX #NmiBase ; set pointer to NMI values

STX TempB ; save interrupt pointer

JSR LAB\_IGBY ; increment and scan memory (past token)

JSR LAB\_GFPN ; get fixed-point number into temp integer

LDA Smeml ; get start of mem low byte

LDX Smemh ; get start of mem high byte

JSR LAB\_SHLN ; search Basic for temp integer line number from AX

BCS LAB\_LFND ; if carry set go set-up interrupt

JMP LAB\_16F7 ; else go do "Undefined statement" error and warm start

LAB\_LFND

LDX TempB ; get interrupt pointer

LDA Baslnl ; get pointer low byte

SBC #$01 ; -1 (carry already set for subtract)

STA PLUS\_1,X ; save as interrupt pointer low byte

LDA Baslnh ; get pointer high byte

SBC #$00 ; subtract carry

STA PLUS\_2,X ; save as interrupt pointer high byte

LDA #$C0 ; set interrupt enabled/setup bits

STA PLUS\_0,X ; set interrupt flags

LAB\_IRTS

RTS

; return from IRQ service, restores the enabled flag.

; perform RETIRQ

LAB\_RETIRQ

BNE LAB\_IRTS ; exit if following token (to allow syntax error)

LDA IrqBase ; get interrupt flags

ASL ; copy setup to enabled (b7)

ORA IrqBase ; OR in setup flag

STA IrqBase ; save enabled flag

JMP LAB\_16E8 ; go do rest of RETURN

; return from NMI service, restores the enabled flag.

; perform RETNMI

LAB\_RETNMI

BNE LAB\_IRTS ; exit if following token (to allow syntax error)

LDA NmiBase ; get set-up flag

ASL ; copy setup to enabled (b7)

ORA NmiBase ; OR in setup flag

STA NmiBase ; save enabled flag

JMP LAB\_16E8 ; go do rest of RETURN

; MAX() MIN() pre process

LAB\_MMPP

JSR LAB\_EVEZ ; process expression

JMP LAB\_CTNM ; check if source is numeric, else do type mismatch

; perform MAX()

LAB\_MAX

JSR LAB\_PHFA ; push FAC1, evaluate expression,

; pull FAC2 and compare with FAC1

BPL LAB\_MAX ; branch if no swap to do

LDA FAC2\_1 ; get FAC2 mantissa1

ORA #$80 ; set top bit (clear sign from compare)

STA FAC2\_1 ; save FAC2 mantissa1

JSR LAB\_279B ; copy FAC2 to FAC1

BEQ LAB\_MAX ; go do next (branch always)

; perform MIN()

LAB\_MIN

JSR LAB\_PHFA ; push FAC1, evaluate expression,

; pull FAC2 and compare with FAC1

BMI LAB\_MIN ; branch if no swap to do

BEQ LAB\_MIN ; branch if no swap to do

LDA FAC2\_1 ; get FAC2 mantissa1

ORA #$80 ; set top bit (clear sign from compare)

STA FAC2\_1 ; save FAC2 mantissa1

JSR LAB\_279B ; copy FAC2 to FAC1

BEQ LAB\_MIN ; go do next (branch always)

; exit routine. don't bother returning to the loop code

; check for correct exit, else so syntax error

LAB\_MMEC

CMP #')' ; is it end of function?

BNE LAB\_MMSE ; if not do MAX MIN syntax error

PLA ; dump return address low byte

PLA ; dump return address high byte

JMP LAB\_IGBY ; update BASIC execute pointer (to chr past ")")

LAB\_MMSE

JMP LAB\_SNER ; do syntax error then warm start

; check for next, evaluate and return or exit

; this is the routine that does most of the work

LAB\_PHFA

JSR LAB\_GBYT ; get next BASIC byte

CMP #',' ; is there more ?

BNE LAB\_MMEC ; if not go do end check

; push FAC1

JSR LAB\_27BA ; round FAC1

LDA FAC1\_s ; get FAC1 sign

ORA #$7F ; set all non sign bits

AND FAC1\_1 ; AND FAC1 mantissa1 (AND in sign bit)

PHA ; push on stack

LDA FAC1\_2 ; get FAC1 mantissa2

PHA ; push on stack

LDA FAC1\_3 ; get FAC1 mantissa3

PHA ; push on stack

LDA FAC1\_e ; get FAC1 exponent

PHA ; push on stack

JSR LAB\_IGBY ; scan and get next BASIC byte (after ",")

JSR LAB\_EVNM ; evaluate expression and check is numeric,

; else do type mismatch

; pop FAC2 (MAX/MIN expression so far)

PLA ; pop exponent

STA FAC2\_e ; save FAC2 exponent

PLA ; pop mantissa3

STA FAC2\_3 ; save FAC2 mantissa3

PLA ; pop mantissa1

STA FAC2\_2 ; save FAC2 mantissa2

PLA ; pop sign/mantissa1

STA FAC2\_1 ; save FAC2 sign/mantissa1

STA FAC2\_s ; save FAC2 sign

; compare FAC1 with (packed) FAC2

LDA #<FAC2\_e ; set pointer low byte to FAC2

LDY #>FAC2\_e ; set pointer high byte to FAC2

JMP LAB\_27F8 ; compare FAC1 with FAC2 (AY) and return

; returns A=$00 if FAC1 = (AY)

; returns A=$01 if FAC1 > (AY)

; returns A=$FF if FAC1 < (AY)

; perform WIDTH

LAB\_WDTH

CMP #',' ; is next byte ","

BEQ LAB\_TBSZ ; if so do tab size

JSR LAB\_GTBY ; get byte parameter

TXA ; copy width to A

BEQ LAB\_NSTT ; branch if set for infinite line

CPX #$10 ; else make min width = 16d

BCC TabErr ; if less do function call error and exit

; this next compare ensures that we can't exit WIDTH via an error leaving the

; tab size greater than the line length.

CPX TabSiz ; compare with tab size

BCS LAB\_NSTT ; branch if >= tab size

STX TabSiz ; else make tab size = terminal width

LAB\_NSTT

STX TWidth ; set the terminal width

JSR LAB\_GBYT ; get BASIC byte back

BEQ WExit ; exit if no following

CMP #',' ; else is it ","

BNE LAB\_MMSE ; if not do syntax error

LAB\_TBSZ

JSR LAB\_SGBY ; scan and get byte parameter

TXA ; copy TAB size

BMI TabErr ; if >127 do function call error and exit

CPX #$01 ; compare with min-1

BCC TabErr ; if <=1 do function call error and exit

LDA TWidth ; set flags for width

BEQ LAB\_SVTB ; skip check if infinite line

CPX TWidth ; compare TAB with width

BEQ LAB\_SVTB ; ok if =

BCS TabErr ; branch if too big

LAB\_SVTB

STX TabSiz ; save TAB size

; calculate tab column limit from TAB size. The Iclim is set to the last tab

; position on a line that still has at least one whole tab width between it

; and the end of the line.

WExit

LDA TWidth ; get width

BEQ LAB\_SULP ; branch if infinite line

CMP TabSiz ; compare with tab size

BCS LAB\_WDLP ; branch if >= tab size

STA TabSiz ; else make tab size = terminal width

LAB\_SULP

SEC ; set carry for subtract

LAB\_WDLP

SBC TabSiz ; subtract tab size

BCS LAB\_WDLP ; loop while no borrow

ADC TabSiz ; add tab size back

CLC ; clear carry for add

ADC TabSiz ; add tab size back again

STA Iclim ; save for now

LDA TWidth ; get width back

SEC ; set carry for subtract

SBC Iclim ; subtract remainder

STA Iclim ; save tab column limit

LAB\_NOSQ

RTS

TabErr

JMP LAB\_FCER ; do function call error then warm start

; perform SQR()

LAB\_SQR

LDA FAC1\_s ; get FAC1 sign

BMI TabErr ; if -ve do function call error

LDA FAC1\_e ; get exponent

BEQ LAB\_NOSQ ; if zero just return

; else do root

JSR LAB\_27AB ; round and copy FAC1 to FAC2

LDA #$00 ; clear A

STA FACt\_3 ; clear remainder

STA FACt\_2 ; ..

STA FACt\_1 ; ..

STA TempB ; ..

STA FAC1\_3 ; clear root

STA FAC1\_2 ; ..

STA FAC1\_1 ; ..

LDX #$18 ; 24 pairs of bits to do

LDA FAC2\_e ; get exponent

LSR ; check odd/even

BCS LAB\_SQE2 ; if odd only 1 shift first time

LAB\_SQE1

ASL FAC2\_3 ; shift highest bit of number ..

ROL FAC2\_2 ; ..

ROL FAC2\_1 ; ..

ROL FACt\_3 ; .. into remainder

ROL FACt\_2 ; ..

ROL FACt\_1 ; ..

ROL TempB ; .. never overflows

LAB\_SQE2

ASL FAC2\_3 ; shift highest bit of number ..

ROL FAC2\_2 ; ..

ROL FAC2\_1 ; ..

ROL FACt\_3 ; .. into remainder

ROL FACt\_2 ; ..

ROL FACt\_1 ; ..

ROL TempB ; .. never overflows

ASL FAC1\_3 ; root = root \* 2

ROL FAC1\_2 ; ..

ROL FAC1\_1 ; .. never overflows

LDA FAC1\_3 ; get root low byte

ROL ; \*2

STA Temp3 ; save partial low byte

LDA FAC1\_2 ; get root low mid byte

ROL ; \*2

STA Temp3+1 ; save partial low mid byte

LDA FAC1\_1 ; get root high mid byte

ROL ; \*2

STA Temp3+2 ; save partial high mid byte

LDA #$00 ; get root high byte (always $00)

ROL ; \*2

STA Temp3+3 ; save partial high byte

; carry clear for subtract +1

LDA FACt\_3 ; get remainder low byte

SBC Temp3 ; subtract partial low byte

STA Temp3 ; save partial low byte

LDA FACt\_2 ; get remainder low mid byte

SBC Temp3+1 ; subtract partial low mid byte

STA Temp3+1 ; save partial low mid byte

LDA FACt\_1 ; get remainder high mid byte

SBC Temp3+2 ; subtract partial high mid byte

TAY ; copy partial high mid byte

LDA TempB ; get remainder high byte

SBC Temp3+3 ; subtract partial high byte

BCC LAB\_SQNS ; skip sub if remainder smaller

STA TempB ; save remainder high byte

STY FACt\_1 ; save remainder high mid byte

LDA Temp3+1 ; get remainder low mid byte

STA FACt\_2 ; save remainder low mid byte

LDA Temp3 ; get partial low byte

STA FACt\_3 ; save remainder low byte

INC FAC1\_3 ; increment root low byte (never any rollover)

LAB\_SQNS

DEX ; decrement bit pair count

BNE LAB\_SQE1 ; loop if not all done

SEC ; set carry for subtract

LDA FAC2\_e ; get exponent

SBC #$80 ; normalise

ROR ; /2 and re-bias to $80

ADC #$00 ; add bit zero back in (allow for half shift)

STA FAC1\_e ; save it

JMP LAB\_24D5 ; normalise FAC1 and return

; perform VARPTR()

LAB\_VARPTR

JSR LAB\_IGBY ; increment and scan memory

JSR LAB\_GVAR ; get var address

JSR LAB\_1BFB ; scan for ")" , else do syntax error then warm start

LDY Cvaral ; get var address low byte

LDA Cvarah ; get var address high byte

JMP LAB\_AYFC ; save and convert integer AY to FAC1 and return

; perform PI

LAB\_PI

LDA #<LAB\_2C7C ; set (2\*pi) pointer low byte

LDY #>LAB\_2C7C ; set (2\*pi) pointer high byte

JSR LAB\_UFAC ; unpack memory (AY) into FAC1

DEC FAC1\_e ; make result = PI

RTS

; perform TWOPI

LAB\_TWOPI

LDA #<LAB\_2C7C ; set (2\*pi) pointer low byte

LDY #>LAB\_2C7C ; set (2\*pi) pointer high byte

JMP LAB\_UFAC ; unpack memory (AY) into FAC1 and return

; system dependant i/o vectors

; these are in RAM and are set by the monitor at start-up

V\_INPT

JMP (VEC\_IN) ; non halting scan input device

V\_OUTP

JMP (VEC\_OUT) ; send byte to output device

V\_LOAD

JMP (VEC\_LD) ; load BASIC program

V\_SAVE

JMP (VEC\_SV) ; save BASIC program

; The rest are tables messages and code for RAM

; the rest of the code is tables and BASIC start-up code

PG2\_TABS

.byte $00 ; ctrl-c flag - $00 = enabled

.byte $00 ; ctrl-c byte - GET needs this

.byte $00 ; ctrl-c byte timeout - GET needs this

.word CTRLC ; ctrl c check vector

; .word xxxx ; non halting key input - monitor to set this

; .word xxxx ; output vector - monitor to set this

; .word xxxx ; load vector - monitor to set this

; .word xxxx ; save vector - monitor to set this

PG2\_TABE

; character get subroutine for zero page

; For a 1.8432MHz 6502 including the JSR and RTS

; fastest (>=":") = 29 cycles = 15.7uS

; slowest (<":") = 40 cycles = 21.7uS

; space skip = +21 cycles = +11.4uS

; inc across page = +4 cycles = +2.2uS

; the target address for the LDA at LAB\_2CF4 becomes the BASIC execute pointer once the

; block is copied to it's destination, any non zero page address will do at assembly

; time, to assemble a three byte instruction.

; page 0 initialisation table from $BC

; increment and scan memory

LAB\_2CEE

INC Bpntrl ; increment BASIC execute pointer low byte

BNE LAB\_2CF4 ; branch if no carry

; else

INC Bpntrh ; increment BASIC execute pointer high byte

; page 0 initialisation table from $C2

; scan memory

LAB\_2CF4

LDA $FFFF ; get byte to scan (addr set by call routine)

CMP #TK\_ELSE ; compare with the token for ELSE

BEQ LAB\_2D05 ; exit if ELSE, not numeric, carry set

CMP #':' ; compare with ":"

BCS LAB\_2D05 ; exit if >= ":", not numeric, carry set

CMP #' ' ; compare with " "

BEQ LAB\_2CEE ; if " " go do next

SEC ; set carry for SBC

SBC #'0' ; subtract "0"

SEC ; set carry for SBC

SBC #$D0 ; subtract -"0"

; clear carry if byte = "0"-"9"

LAB\_2D05

RTS

; page zero initialisation table $00-$12 inclusive

StrTab

.byte $4C ; JMP opcode

.word LAB\_COLD ; initial warm start vector (cold start)

.byte $00 ; these bytes are not used by BASIC

.word $0000 ;

.word $0000 ;

.word $0000 ;

.byte $4C ; JMP opcode

.word LAB\_FCER ; initial user function vector ("Function call" error)

.byte $00 ; default NULL count

.byte $00 ; clear terminal position

.byte $00 ; default terminal width byte

.byte $F2 ; default limit for TAB = 14

.word Ram\_base ; start of user RAM

EndTab

LAB\_MSZM

.byte $0D,$0A,"Memory size ",$00

LAB\_SMSG

.byte " Bytes free",$0D,$0A,$0A

.byte "Enhanced BASIC 2.22",$0A,$00

; numeric constants and series

; constants and series for LOG(n)

LAB\_25A0

.byte $02 ; counter

.byte $80,$19,$56,$62 ; 0.59898

.byte $80,$76,$22,$F3 ; 0.96147

;## .byte $80,$76,$22,$F1 ; 0.96147

.byte $82,$38,$AA,$40 ; 2.88539

;## .byte $82,$38,$AA,$45 ; 2.88539

LAB\_25AD

.byte $80,$35,$04,$F3 ; 0.70711 1/root 2

LAB\_25B1

.byte $81,$35,$04,$F3 ; 1.41421 root 2

LAB\_25B5

.byte $80,$80,$00,$00 ; -0.5

LAB\_25B9

.byte $80,$31,$72,$18 ; 0.69315 LOG(2)

; numeric PRINT constants

LAB\_2947

.byte $91,$43,$4F,$F8 ; 99999.9375 (max value with at least one decimal)

LAB\_294B

.byte $94,$74,$23,$F7 ; 999999.4375 (max value before scientific notation)

LAB\_294F

.byte $94,$74,$24,$00 ; 1000000

; EXP(n) constants and series

LAB\_2AFA

.byte $81,$38,$AA,$3B ; 1.4427 (1/LOG base 2 e)

LAB\_2AFE

.byte $06 ; counter

.byte $74,$63,$90,$8C ; 2.17023e-4

.byte $77,$23,$0C,$AB ; 0.00124

.byte $7A,$1E,$94,$00 ; 0.00968

.byte $7C,$63,$42,$80 ; 0.05548

.byte $7E,$75,$FE,$D0 ; 0.24023

.byte $80,$31,$72,$15 ; 0.69315

.byte $81,$00,$00,$00 ; 1.00000

;## .byte $07 ; counter

;## .byte $74,$94,$2E,$40 ; -1/7! (-1/5040)

;## .byte $77,$2E,$4F,$70 ; 1/6! ( 1/720)

;## .byte $7A,$88,$02,$6E ; -1/5! (-1/120)

;## .byte $7C,$2A,$A0,$E6 ; 1/4! ( 1/24)

;## .byte $7E,$AA,$AA,$50 ; -1/3! (-1/6)

;## .byte $7F,$7F,$FF,$FF ; 1/2! ( 1/2)

;## .byte $81,$80,$00,$00 ; -1/1! (-1/1)

;## .byte $81,$00,$00,$00 ; 1/0! ( 1/1)

; trigonometric constants and series

LAB\_2C78

.byte $81,$49,$0F,$DB ; 1.570796371 (pi/2) as floating #

LAB\_2C84

.byte $04 ; counter

.byte $86,$1E,$D7,$FB ; 39.7109

;## .byte $86,$1E,$D7,$BA ; 39.7109

.byte $87,$99,$26,$65 ;-76.575

;## .byte $87,$99,$26,$64 ;-76.575

.byte $87,$23,$34,$58 ; 81.6022

.byte $86,$A5,$5D,$E1 ;-41.3417

;## .byte $86,$A5,$5D,$E0 ;-41.3417

LAB\_2C7C

.byte $83,$49,$0F,$DB ; 6.28319 (2\*pi) as floating #

;## .byte $83,$49,$0F,$DA ; 6.28319 (2\*pi) as floating #

LAB\_2CC9

.byte $08 ; counter

.byte $78,$3A,$C5,$37 ; 0.00285

.byte $7B,$83,$A2,$5C ;-0.0160686

.byte $7C,$2E,$DD,$4D ; 0.0426915

.byte $7D,$99,$B0,$1E ;-0.0750429

.byte $7D,$59,$ED,$24 ; 0.106409

.byte $7E,$91,$72,$00 ;-0.142036

.byte $7E,$4C,$B9,$73 ; 0.199926

.byte $7F,$AA,$AA,$53 ;-0.333331

;## .byte $08 ; counter

;## .byte $78,$3B,$D7,$4A ; 1/17

;## .byte $7B,$84,$6E,$02 ;-1/15

;## .byte $7C,$2F,$C1,$FE ; 1/13

;## .byte $7D,$9A,$31,$74 ;-1/11

;## .byte $7D,$5A,$3D,$84 ; 1/9

;## .byte $7E,$91,$7F,$C8 ;-1/7

;## .byte $7E,$4C,$BB,$E4 ; 1/5

;## .byte $7F,$AA,$AA,$6C ;-1/3

LAB\_1D96 = \*+1 ; $00,$00 used for undefined variables

LAB\_259C

.byte $81,$00,$00,$00 ; 1.000000, used for INC

LAB\_2AFD

.byte $81,$80,$00,$00 ; -1.00000, used for DEC. must be on the same page as +1.00

; misc constants

LAB\_1DF7

.byte $90 ;-32768 (uses first three bytes from 0.5)

LAB\_2A96

.byte $80,$00,$00,$00 ; 0.5

LAB\_2C80

.byte $7F,$00,$00,$00 ; 0.25

LAB\_26B5

.byte $84,$20,$00,$00 ; 10.0000 divide by 10 constant

; This table is used in converting numbers to ASCII.

LAB\_2A9A

LAB\_2A9B = LAB\_2A9A+1

LAB\_2A9C = LAB\_2A9B+1

.byte $FE,$79,$60 ; -100000

.byte $00,$27,$10 ; 10000

.byte $FF,$FC,$18 ; -1000

.byte $00,$00,$64 ; 100

.byte $FF,$FF,$F6 ; -10

.byte $00,$00,$01 ; 1

LAB\_CTBL

.word LAB\_END-1 ; END

.word LAB\_FOR-1 ; FOR

.word LAB\_NEXT-1 ; NEXT

.word LAB\_DATA-1 ; DATA

.word LAB\_INPUT-1 ; INPUT

.word LAB\_DIM-1 ; DIM

.word LAB\_READ-1 ; READ

.word LAB\_LET-1 ; LET

.word LAB\_DEC-1 ; DEC new command

.word LAB\_GOTO-1 ; GOTO

.word LAB\_RUN-1 ; RUN

.word LAB\_IF-1 ; IF

.word LAB\_RESTORE-1 ; RESTORE modified command

.word LAB\_GOSUB-1 ; GOSUB

.word LAB\_RETIRQ-1 ; RETIRQ new command

.word LAB\_RETNMI-1 ; RETNMI new command

.word LAB\_RETURN-1 ; RETURN

.word LAB\_REM-1 ; REM

.word LAB\_STOP-1 ; STOP

.word LAB\_ON-1 ; ON modified command

.word LAB\_NULL-1 ; NULL modified command

.word LAB\_INC-1 ; INC new command

.word LAB\_WAIT-1 ; WAIT

.word V\_LOAD-1 ; LOAD

.word V\_SAVE-1 ; SAVE

.word LAB\_DEF-1 ; DEF

.word LAB\_POKE-1 ; POKE

.word LAB\_DOKE-1 ; DOKE new command

.word LAB\_CALL-1 ; CALL new command

.word LAB\_DO-1 ; DO new command

.word LAB\_LOOP-1 ; LOOP new command

.word LAB\_PRINT-1 ; PRINT

.word LAB\_CONT-1 ; CONT

.word LAB\_LIST-1 ; LIST

.word LAB\_CLEAR-1 ; CLEAR

.word LAB\_NEW-1 ; NEW

.word LAB\_WDTH-1 ; WIDTH new command

.word LAB\_GET-1 ; GET new command

.word LAB\_SWAP-1 ; SWAP new command

.word LAB\_BITSET-1 ; BITSET new command

.word LAB\_BITCLR-1 ; BITCLR new command

.word LAB\_IRQ-1 ; IRQ new command

.word LAB\_NMI-1 ; NMI new command

; function pre process routine table

LAB\_FTPL

LAB\_FTPM = LAB\_FTPL+$01

.word LAB\_PPFN-1 ; SGN(n) process numeric expression in ()

.word LAB\_PPFN-1 ; INT(n) "

.word LAB\_PPFN-1 ; ABS(n) "

.word LAB\_EVEZ-1 ; USR(x) process any expression

.word LAB\_1BF7-1 ; FRE(x) "

.word LAB\_1BF7-1 ; POS(x) "

.word LAB\_PPFN-1 ; SQR(n) process numeric expression in ()

.word LAB\_PPFN-1 ; RND(n) "

.word LAB\_PPFN-1 ; LOG(n) "

.word LAB\_PPFN-1 ; EXP(n) "

.word LAB\_PPFN-1 ; COS(n) "

.word LAB\_PPFN-1 ; SIN(n) "

.word LAB\_PPFN-1 ; TAN(n) "

.word LAB\_PPFN-1 ; ATN(n) "

.word LAB\_PPFN-1 ; PEEK(n) "

.word LAB\_PPFN-1 ; DEEK(n) "

.word $0000 ; SADD() none

.word LAB\_PPFS-1 ; LEN($) process string expression in ()

.word LAB\_PPFN-1 ; STR$(n) process numeric expression in ()

.word LAB\_PPFS-1 ; VAL($) process string expression in ()

.word LAB\_PPFS-1 ; ASC($) "

.word LAB\_PPFS-1 ; UCASE$($) "

.word LAB\_PPFS-1 ; LCASE$($) "

.word LAB\_PPFN-1 ; CHR$(n) process numeric expression in ()

.word LAB\_BHSS-1 ; HEX$(n) "

.word LAB\_BHSS-1 ; BIN$(n) "

.word $0000 ; BITTST() none

.word LAB\_MMPP-1 ; MAX() process numeric expression

.word LAB\_MMPP-1 ; MIN() "

.word LAB\_PPBI-1 ; PI advance pointer

.word LAB\_PPBI-1 ; TWOPI "

.word $0000 ; VARPTR() none

.word LAB\_LRMS-1 ; LEFT$() process string expression

.word LAB\_LRMS-1 ; RIGHT$() "

.word LAB\_LRMS-1 ; MID$() "

; action addresses for functions

LAB\_FTBL

LAB\_FTBM = LAB\_FTBL+$01

.word LAB\_SGN-1 ; SGN()

.word LAB\_INT-1 ; INT()

.word LAB\_ABS-1 ; ABS()

.word LAB\_USR-1 ; USR()

.word LAB\_FRE-1 ; FRE()

.word LAB\_POS-1 ; POS()

.word LAB\_SQR-1 ; SQR()

.word LAB\_RND-1 ; RND() modified function

.word LAB\_LOG-1 ; LOG()

.word LAB\_EXP-1 ; EXP()

.word LAB\_COS-1 ; COS()

.word LAB\_SIN-1 ; SIN()

.word LAB\_TAN-1 ; TAN()

.word LAB\_ATN-1 ; ATN()

.word LAB\_PEEK-1 ; PEEK()

.word LAB\_DEEK-1 ; DEEK() new function

.word LAB\_SADD-1 ; SADD() new function

.word LAB\_LENS-1 ; LEN()

.word LAB\_STRS-1 ; STR$()

.word LAB\_VAL-1 ; VAL()

.word LAB\_ASC-1 ; ASC()

.word LAB\_UCASE-1 ; UCASE$() new function

.word LAB\_LCASE-1 ; LCASE$() new function

.word LAB\_CHRS-1 ; CHR$()

.word LAB\_HEXS-1 ; HEX$() new function

.word LAB\_BINS-1 ; BIN$() new function

.word LAB\_BTST-1 ; BITTST() new function

.word LAB\_MAX-1 ; MAX() new function

.word LAB\_MIN-1 ; MIN() new function

.word LAB\_PI-1 ; PI new function

.word LAB\_TWOPI-1 ; TWOPI new function

.word LAB\_VARPTR-1 ; VARPTR() new function

.word LAB\_LEFT-1 ; LEFT$()

.word LAB\_RIGHT-1 ; RIGHT$()

.word LAB\_MIDS-1 ; MID$()

; hierarchy and action addresses for operator

LAB\_OPPT

.byte $79 ; +

.word LAB\_ADD-1

.byte $79 ; -

.word LAB\_SUBTRACT-1

.byte $7B ; \*

.word LAB\_MULTIPLY-1

.byte $7B ; /

.word LAB\_DIVIDE-1

.byte $7F ; ^

.word LAB\_POWER-1

.byte $50 ; AND

.word LAB\_AND-1

.byte $46 ; EOR new operator

.word LAB\_EOR-1

.byte $46 ; OR

.word LAB\_OR-1

.byte $56 ; >> new operator

.word LAB\_RSHIFT-1

.byte $56 ; << new operator

.word LAB\_LSHIFT-1

.byte $7D ; >

.word LAB\_GTHAN-1

.byte $5A ; =

.word LAB\_EQUAL-1

.byte $64 ; <

.word LAB\_LTHAN-1

; keywords start with ..

; this is the first character table and must be in alphabetic order

TAB\_1STC

.byte "\*"

.byte "+"

.byte "-"

.byte "/"

.byte "<"

.byte "="

.byte ">"

.byte "?"

.byte "A"

.byte "B"

.byte "C"

.byte "D"

.byte "E"

.byte "F"

.byte "G"

.byte "H"

.byte "I"

.byte "L"

.byte "M"

.byte "N"

.byte "O"

.byte "P"

.byte "R"

.byte "S"

.byte "T"

.byte "U"

.byte "V"

.byte "W"

.byte "^"

.byte $00 ; table terminator

; pointers to keyword tables

TAB\_CHRT

.word TAB\_STAR ; table for "\*"

.word TAB\_PLUS ; table for "+"

.word TAB\_MNUS ; table for "-"

.word TAB\_SLAS ; table for "/"

.word TAB\_LESS ; table for "<"

.word TAB\_EQUL ; table for "="

.word TAB\_MORE ; table for ">"

.word TAB\_QEST ; table for "?"

.word TAB\_ASCA ; table for "A"

.word TAB\_ASCB ; table for "B"

.word TAB\_ASCC ; table for "C"

.word TAB\_ASCD ; table for "D"

.word TAB\_ASCE ; table for "E"

.word TAB\_ASCF ; table for "F"

.word TAB\_ASCG ; table for "G"

.word TAB\_ASCH ; table for "H"

.word TAB\_ASCI ; table for "I"

.word TAB\_ASCL ; table for "L"

.word TAB\_ASCM ; table for "M"

.word TAB\_ASCN ; table for "N"

.word TAB\_ASCO ; table for "O"

.word TAB\_ASCP ; table for "P"

.word TAB\_ASCR ; table for "R"

.word TAB\_ASCS ; table for "S"

.word TAB\_ASCT ; table for "T"

.word TAB\_ASCU ; table for "U"

.word TAB\_ASCV ; table for "V"

.word TAB\_ASCW ; table for "W"

.word TAB\_POWR ; table for "^"

; tables for each start character, note if a longer keyword with the same start

; letters as a shorter one exists then it must come first, else the list is in

; alphabetical order as follows ..

; [keyword,token

; [keyword,token]]

; end marker (#$00)

TAB\_STAR

.byte TK\_MUL,$00 ; \*

TAB\_PLUS

.byte TK\_PLUS,$00 ; +

TAB\_MNUS

.byte TK\_MINUS,$00 ; -

TAB\_SLAS

.byte TK\_DIV,$00 ; /

TAB\_LESS

LBB\_LSHIFT

.byte "<",TK\_LSHIFT ; << note - "<<" must come before "<"

.byte TK\_LT ; <

.byte $00

TAB\_EQUL

.byte TK\_EQUAL,$00 ; =

TAB\_MORE

LBB\_RSHIFT

.byte ">",TK\_RSHIFT ; >> note - ">>" must come before ">"

.byte TK\_GT ; >

.byte $00

TAB\_QEST

.byte TK\_PRINT,$00 ; ?

TAB\_ASCA

LBB\_ABS

.byte "BS(",TK\_ABS ; ABS(

LBB\_AND

.byte "ND",TK\_AND ; AND

LBB\_ASC

.byte "SC(",TK\_ASC ; ASC(

LBB\_ATN

.byte "TN(",TK\_ATN ; ATN(

.byte $00

TAB\_ASCB

LBB\_BINS

.byte "IN$(",TK\_BINS ; BIN$(

LBB\_BITCLR

.byte "ITCLR",TK\_BITCLR ; BITCLR

LBB\_BITSET

.byte "ITSET",TK\_BITSET ; BITSET

LBB\_BITTST

.byte "ITTST(",TK\_BITTST

; BITTST(

.byte $00

TAB\_ASCC

LBB\_CALL

.byte "ALL",TK\_CALL ; CALL

LBB\_CHRS

.byte "HR$(",TK\_CHRS ; CHR$(

LBB\_CLEAR

.byte "LEAR",TK\_CLEAR ; CLEAR

LBB\_CONT

.byte "ONT",TK\_CONT ; CONT

LBB\_COS

.byte "OS(",TK\_COS ; COS(

.byte $00

TAB\_ASCD

LBB\_DATA

.byte "ATA",TK\_DATA ; DATA

LBB\_DEC

.byte "EC",TK\_DEC ; DEC

LBB\_DEEK

.byte "EEK(",TK\_DEEK ; DEEK(

LBB\_DEF

.byte "EF",TK\_DEF ; DEF

LBB\_DIM

.byte "IM",TK\_DIM ; DIM

LBB\_DOKE

.byte "OKE",TK\_DOKE ; DOKE note - "DOKE" must come before "DO"

LBB\_DO

.byte "O",TK\_DO ; DO

.byte $00

TAB\_ASCE

LBB\_ELSE

.byte "LSE",TK\_ELSE ; ELSE

LBB\_END

.byte "ND",TK\_END ; END

LBB\_EOR

.byte "OR",TK\_EOR ; EOR

LBB\_EXP

.byte "XP(",TK\_EXP ; EXP(

.byte $00

TAB\_ASCF

LBB\_FN

.byte "N",TK\_FN ; FN

LBB\_FOR

.byte "OR",TK\_FOR ; FOR

LBB\_FRE

.byte "RE(",TK\_FRE ; FRE(

.byte $00

TAB\_ASCG

LBB\_GET

.byte "ET",TK\_GET ; GET

LBB\_GOSUB

.byte "OSUB",TK\_GOSUB ; GOSUB

LBB\_GOTO

.byte "OTO",TK\_GOTO ; GOTO

.byte $00

TAB\_ASCH

LBB\_HEXS

.byte "EX$(",TK\_HEXS ; HEX$(

.byte $00

TAB\_ASCI

LBB\_IF

.byte "F",TK\_IF ; IF

LBB\_INC

.byte "NC",TK\_INC ; INC

LBB\_INPUT

.byte "NPUT",TK\_INPUT ; INPUT

LBB\_INT

.byte "NT(",TK\_INT ; INT(

LBB\_IRQ

.byte "RQ",TK\_IRQ ; IRQ

.byte $00

TAB\_ASCL

LBB\_LCASES

.byte "CASE$(",TK\_LCASES

; LCASE$(

LBB\_LEFTS

.byte "EFT$(",TK\_LEFTS ; LEFT$(

LBB\_LEN

.byte "EN(",TK\_LEN ; LEN(

LBB\_LET

.byte "ET",TK\_LET ; LET

LBB\_LIST

.byte "IST",TK\_LIST ; LIST

LBB\_LOAD

.byte "OAD",TK\_LOAD ; LOAD

LBB\_LOG

.byte "OG(",TK\_LOG ; LOG(

LBB\_LOOP

.byte "OOP",TK\_LOOP ; LOOP

.byte $00

TAB\_ASCM

LBB\_MAX

.byte "AX(",TK\_MAX ; MAX(

LBB\_MIDS

.byte "ID$(",TK\_MIDS ; MID$(

LBB\_MIN

.byte "IN(",TK\_MIN ; MIN(

.byte $00

TAB\_ASCN

LBB\_NEW

.byte "EW",TK\_NEW ; NEW

LBB\_NEXT

.byte "EXT",TK\_NEXT ; NEXT

LBB\_NMI

.byte "MI",TK\_NMI ; NMI

LBB\_NOT

.byte "OT",TK\_NOT ; NOT

LBB\_NULL

.byte "ULL",TK\_NULL ; NULL

.byte $00

TAB\_ASCO

LBB\_OFF

.byte "FF",TK\_OFF ; OFF

LBB\_ON

.byte "N",TK\_ON ; ON

LBB\_OR

.byte "R",TK\_OR ; OR

.byte $00

TAB\_ASCP

LBB\_PEEK

.byte "EEK(",TK\_PEEK ; PEEK(

LBB\_PI

.byte "I",TK\_PI ; PI

LBB\_POKE

.byte "OKE",TK\_POKE ; POKE

LBB\_POS

.byte "OS(",TK\_POS ; POS(

LBB\_PRINT

.byte "RINT",TK\_PRINT ; PRINT

.byte $00

TAB\_ASCR

LBB\_READ

.byte "EAD",TK\_READ ; READ

LBB\_REM

.byte "EM",TK\_REM ; REM

LBB\_RESTORE

.byte "ESTORE",TK\_RESTORE

; RESTORE

LBB\_RETIRQ

.byte "ETIRQ",TK\_RETIRQ ; RETIRQ

LBB\_RETNMI

.byte "ETNMI",TK\_RETNMI ; RETNMI

LBB\_RETURN

.byte "ETURN",TK\_RETURN ; RETURN

LBB\_RIGHTS

.byte "IGHT$(",TK\_RIGHTS

; RIGHT$(

LBB\_RND

.byte "ND(",TK\_RND ; RND(

LBB\_RUN

.byte "UN",TK\_RUN ; RUN

.byte $00

TAB\_ASCS

LBB\_SADD

.byte "ADD(",TK\_SADD ; SADD(

LBB\_SAVE

.byte "AVE",TK\_SAVE ; SAVE

LBB\_SGN

.byte "GN(",TK\_SGN ; SGN(

LBB\_SIN

.byte "IN(",TK\_SIN ; SIN(

LBB\_SPC

.byte "PC(",TK\_SPC ; SPC(

LBB\_SQR

.byte "QR(",TK\_SQR ; SQR(

LBB\_STEP

.byte "TEP",TK\_STEP ; STEP

LBB\_STOP

.byte "TOP",TK\_STOP ; STOP

LBB\_STRS

.byte "TR$(",TK\_STRS ; STR$(

LBB\_SWAP

.byte "WAP",TK\_SWAP ; SWAP

.byte $00

TAB\_ASCT

LBB\_TAB

.byte "AB(",TK\_TAB ; TAB(

LBB\_TAN

.byte "AN(",TK\_TAN ; TAN(

LBB\_THEN

.byte "HEN",TK\_THEN ; THEN

LBB\_TO

.byte "O",TK\_TO ; TO

LBB\_TWOPI

.byte "WOPI",TK\_TWOPI ; TWOPI

.byte $00

TAB\_ASCU

LBB\_UCASES

.byte "CASE$(",TK\_UCASES

; UCASE$(

LBB\_UNTIL

.byte "NTIL",TK\_UNTIL ; UNTIL

LBB\_USR

.byte "SR(",TK\_USR ; USR(

.byte $00

TAB\_ASCV

LBB\_VAL

.byte "AL(",TK\_VAL ; VAL(

LBB\_VPTR

.byte "ARPTR(",TK\_VPTR ; VARPTR(

.byte $00

TAB\_ASCW

LBB\_WAIT

.byte "AIT",TK\_WAIT ; WAIT

LBB\_WHILE

.byte "HILE",TK\_WHILE ; WHILE

LBB\_WIDTH

.byte "IDTH",TK\_WIDTH ; WIDTH

.byte $00

TAB\_POWR

.byte TK\_POWER,$00 ; ^

; new decode table for LIST

; Table is ..

; byte - keyword length, keyword first character

; word - pointer to rest of keyword from dictionary

; note if length is 1 then the pointer is ignored

LAB\_KEYT

.byte 3,'E'

.word LBB\_END ; END

.byte 3,'F'

.word LBB\_FOR ; FOR

.byte 4,'N'

.word LBB\_NEXT ; NEXT

.byte 4,'D'

.word LBB\_DATA ; DATA

.byte 5,'I'

.word LBB\_INPUT ; INPUT

.byte 3,'D'

.word LBB\_DIM ; DIM

.byte 4,'R'

.word LBB\_READ ; READ

.byte 3,'L'

.word LBB\_LET ; LET

.byte 3,'D'

.word LBB\_DEC ; DEC

.byte 4,'G'

.word LBB\_GOTO ; GOTO

.byte 3,'R'

.word LBB\_RUN ; RUN

.byte 2,'I'

.word LBB\_IF ; IF

.byte 7,'R'

.word LBB\_RESTORE ; RESTORE

.byte 5,'G'

.word LBB\_GOSUB ; GOSUB

.byte 6,'R'

.word LBB\_RETIRQ ; RETIRQ

.byte 6,'R'

.word LBB\_RETNMI ; RETNMI

.byte 6,'R'

.word LBB\_RETURN ; RETURN

.byte 3,'R'

.word LBB\_REM ; REM

.byte 4,'S'

.word LBB\_STOP ; STOP

.byte 2,'O'

.word LBB\_ON ; ON

.byte 4,'N'

.word LBB\_NULL ; NULL

.byte 3,'I'

.word LBB\_INC ; INC

.byte 4,'W'

.word LBB\_WAIT ; WAIT

.byte 4,'L'

.word LBB\_LOAD ; LOAD

.byte 4,'S'

.word LBB\_SAVE ; SAVE

.byte 3,'D'

.word LBB\_DEF ; DEF

.byte 4,'P'

.word LBB\_POKE ; POKE

.byte 4,'D'

.word LBB\_DOKE ; DOKE

.byte 4,'C'

.word LBB\_CALL ; CALL

.byte 2,'D'

.word LBB\_DO ; DO

.byte 4,'L'

.word LBB\_LOOP ; LOOP

.byte 5,'P'

.word LBB\_PRINT ; PRINT

.byte 4,'C'

.word LBB\_CONT ; CONT

.byte 4,'L'

.word LBB\_LIST ; LIST

.byte 5,'C'

.word LBB\_CLEAR ; CLEAR

.byte 3,'N'

.word LBB\_NEW ; NEW

.byte 5,'W'

.word LBB\_WIDTH ; WIDTH

.byte 3,'G'

.word LBB\_GET ; GET

.byte 4,'S'

.word LBB\_SWAP ; SWAP

.byte 6,'B'

.word LBB\_BITSET ; BITSET

.byte 6,'B'

.word LBB\_BITCLR ; BITCLR

.byte 3,'I'

.word LBB\_IRQ ; IRQ

.byte 3,'N'

.word LBB\_NMI ; NMI

; secondary commands (can't start a statement)

.byte 4,'T'

.word LBB\_TAB ; TAB

.byte 4,'E'

.word LBB\_ELSE ; ELSE

.byte 2,'T'

.word LBB\_TO ; TO

.byte 2,'F'

.word LBB\_FN ; FN

.byte 4,'S'

.word LBB\_SPC ; SPC

.byte 4,'T'

.word LBB\_THEN ; THEN

.byte 3,'N'

.word LBB\_NOT ; NOT

.byte 4,'S'

.word LBB\_STEP ; STEP

.byte 5,'U'

.word LBB\_UNTIL ; UNTIL

.byte 5,'W'

.word LBB\_WHILE ; WHILE

.byte 3,'O'

.word LBB\_OFF ; OFF

; opperators

.byte 1,'+'

.word $0000 ; +

.byte 1,'-'

.word $0000 ; -

.byte 1,'\*'

.word $0000 ; \*

.byte 1,'/'

.word $0000 ; /

.byte 1,'^'

.word $0000 ; ^

.byte 3,'A'

.word LBB\_AND ; AND

.byte 3,'E'

.word LBB\_EOR ; EOR

.byte 2,'O'

.word LBB\_OR ; OR

.byte 2,'>'

.word LBB\_RSHIFT ; >>

.byte 2,'<'

.word LBB\_LSHIFT ; <<

.byte 1,'>'

.word $0000 ; >

.byte 1,'='

.word $0000 ; =

.byte 1,'<'

.word $0000 ; <

; functions

.byte 4,'S' ;

.word LBB\_SGN ; SGN

.byte 4,'I' ;

.word LBB\_INT ; INT

.byte 4,'A' ;

.word LBB\_ABS ; ABS

.byte 4,'U' ;

.word LBB\_USR ; USR

.byte 4,'F' ;

.word LBB\_FRE ; FRE

.byte 4,'P' ;

.word LBB\_POS ; POS

.byte 4,'S' ;

.word LBB\_SQR ; SQR

.byte 4,'R' ;

.word LBB\_RND ; RND

.byte 4,'L' ;

.word LBB\_LOG ; LOG

.byte 4,'E' ;

.word LBB\_EXP ; EXP

.byte 4,'C' ;

.word LBB\_COS ; COS

.byte 4,'S' ;

.word LBB\_SIN ; SIN

.byte 4,'T' ;

.word LBB\_TAN ; TAN

.byte 4,'A' ;

.word LBB\_ATN ; ATN

.byte 5,'P' ;

.word LBB\_PEEK ; PEEK

.byte 5,'D' ;

.word LBB\_DEEK ; DEEK

.byte 5,'S' ;

.word LBB\_SADD ; SADD

.byte 4,'L' ;

.word LBB\_LEN ; LEN

.byte 5,'S' ;

.word LBB\_STRS ; STR$

.byte 4,'V' ;

.word LBB\_VAL ; VAL

.byte 4,'A' ;

.word LBB\_ASC ; ASC

.byte 7,'U' ;

.word LBB\_UCASES ; UCASE$

.byte 7,'L' ;

.word LBB\_LCASES ; LCASE$

.byte 5,'C' ;

.word LBB\_CHRS ; CHR$

.byte 5,'H' ;

.word LBB\_HEXS ; HEX$

.byte 5,'B' ;

.word LBB\_BINS ; BIN$

.byte 7,'B' ;

.word LBB\_BITTST ; BITTST

.byte 4,'M' ;

.word LBB\_MAX ; MAX

.byte 4,'M' ;

.word LBB\_MIN ; MIN

.byte 2,'P' ;

.word LBB\_PI ; PI

.byte 5,'T' ;

.word LBB\_TWOPI ; TWOPI

.byte 7,'V' ;

.word LBB\_VPTR ; VARPTR

.byte 6,'L' ;

.word LBB\_LEFTS ; LEFT$

.byte 7,'R' ;

.word LBB\_RIGHTS ; RIGHT$

.byte 5,'M' ;

.word LBB\_MIDS ; MID$

; BASIC messages, mostly error messages

LAB\_BAER

.word ERR\_NF ;$00 NEXT without FOR

.word ERR\_SN ;$02 syntax

.word ERR\_RG ;$04 RETURN without GOSUB

.word ERR\_OD ;$06 out of data

.word ERR\_FC ;$08 function call

.word ERR\_OV ;$0A overflow

.word ERR\_OM ;$0C out of memory

.word ERR\_US ;$0E undefined statement

.word ERR\_BS ;$10 array bounds

.word ERR\_DD ;$12 double dimension array

.word ERR\_D0 ;$14 divide by 0

.word ERR\_ID ;$16 illegal direct

.word ERR\_TM ;$18 type mismatch

.word ERR\_LS ;$1A long string

.word ERR\_ST ;$1C string too complex

.word ERR\_CN ;$1E continue error

.word ERR\_UF ;$20 undefined function

.word ERR\_LD ;$22 LOOP without DO

; I may implement these two errors to force definition of variables and

; dimensioning of arrays before use.

; .word ERR\_UV ;$24 undefined variable

; the above error has been tested and works (see code and comments below LAB\_1D8B)

; .word ERR\_UA ;$26 undimensioned array

ERR\_NF .byte "NEXT without FOR",$00

ERR\_SN .byte "Syntax",$00

ERR\_RG .byte "RETURN without GOSUB",$00

ERR\_OD .byte "Out of DATA",$00

ERR\_FC .byte "Function call",$00

ERR\_OV .byte "Overflow",$00

ERR\_OM .byte "Out of memory",$00

ERR\_US .byte "Undefined statement",$00

ERR\_BS .byte "Array bounds",$00

ERR\_DD .byte "Double dimension",$00

ERR\_D0 .byte "Divide by zero",$00

ERR\_ID .byte "Illegal direct",$00

ERR\_TM .byte "Type mismatch",$00

ERR\_LS .byte "String too long",$00

ERR\_ST .byte "String too complex",$00

ERR\_CN .byte "Can't continue",$00

ERR\_UF .byte "Undefined function",$00

ERR\_LD .byte "LOOP without DO",$00

;ERR\_UV .byte "Undefined variable",$00

; the above error has been tested and works (see code and comments below LAB\_1D8B)

;ERR\_UA .byte "Undimensioned array",$00

LAB\_BMSG .byte $0D,$0A,"Break",$00

LAB\_EMSG .byte " Error",$00

LAB\_LMSG .byte " in line ",$00

LAB\_RMSG .byte $0D,$0A,"Ready",$0D,$0A,$00

LAB\_IMSG .byte " Extra ignored",$0D,$0A,$00

LAB\_REDO .byte " Redo from start",$0D,$0A,$00

AA\_end\_basic