Introduction

This language has been designed for the programmer with an in-depth knowledge of ATARI's 8 bit computers. It may also be useful to know a little 6502 machine code in order to extend the functionality of the language via user defined functions and procedures.
This language will also be of use to somebody just starting to learn 6502 machine code as small (or large) machine code routines may be incorporated directly within TCL programs themselves.
The language was written as an exercise for myself but take a look at Enigmatix! for an example of what can be achieved.
It currently works with DOS XL by just entering TCL and hitting return. It should work with other DOS systems but it may be necessary to specify the run address yourself. The run address is 0x2600.
There are probably many errors contained within this document since it was last updated long before the last version of the compiler (1986). I have just updated a few areas I knew were wrong but memories fade and I have probably missed some bits.
If you have any problems or queries regarding this documentation send an Email message to myself and I'll find the answer and update this guide.

System Requirements

The minimum system requirements in order to run TCL are as follows:

- An Atari 8 bit computer (e.g. Atari 400, 800, 800XL, 130XE)
- 24K of memory
- A Disk Drive

Syntax Conventions

The following conventions are used in this manual when defining the syntax of a command :-

1. Capital letters designate the names of the procedures or functions and must be typed in exactly as shown (eg. LIST, COMPILE, AND, GOTO etc).
2. Lower case letters specify the type of argument to the commands :-

   lno

   A label number between 1 and 65535, inclusive.

   hxnum

   A string of hexedecimal numbers.

   dcnum

   A positive decimal number.

   exp

   Any valid expression.

   string

   A string of ASCII characters enclosed in double quotes (eg. "String of Characters").

   adr

The address of some data, adr may be an expression.

3. Items enclosed within square brackets denote an optional part of the syntax (eg. exp [,exp...]). When an optional item is followed by ... the item may be repeated as many times as needed.

BOOT

Purpose: To compile the source program directly to a BOOT Disk
Syntax: BOOT

Description

Before pressing return insert a blank formatted disk in D1: the compiled program will be written to the disk as a single part boot disk. The disk image can then be copied onto a standard multiboot type menu.

COMPILE

Purpose: To compile the source program.
Syntax: COMPILE [#file1] [,#file2]

Description

This command will call the compiler which will go through the program line by line, converting it into machine code. If file1 is specified the object code will be created and sent to file1, together with a small runtime package (about 1.5K). If file2 is specified the listing will be produced and sent to this file2.

DOS

Purpose: To return to the disk operating system.
Syntax: DOS

Description

This command will return you to the disk operating system. The compiler may be re-entered by simply typing RUN in a memory resident dos, such as Sparta Dos or Dos XL, or by typing RUN 2600.

ENTER

Purpose: To merge source code from file with loaded source code
Syntax: ENTER #filespec

LIST

Purpose: To list a program to a device.
Syntax: LIST [#filespec,][dcnum1 [,dcnum2]]

Description

This command will list the source program to the specified device. If a device is not specified, the screen editor will be used as default. If only dcnum1 is specified, then only that line will be displayed, but if both dcnum1 and dcnum2 are specified, all the lines between and including dcnum1 and dcnum2 will be listed. When dcnum1 and dcnum2 are not specified, the complete source program will be listed out to the required device.

Example

This will list all the lines between 2000 and 2050 inclusive to the printer.

```
LIST #P:,2000,2050
```

LOAD

Purpose: To reload a stored source program into memory.
Syntax: LOAD #filespec

Description

This command will first clear the program that currently resides in memory, and will then load the previously saved source file into memory, from the specified filespec.

NEW

Purpose: To erase program in editors memory.
Syntax: NEW

Description

This command is used to clear an old program out of memory, before starting to enter a new program. Once this command has been issued, the program cannot be retrieved, and therefore it must be saved to disk if it will be needed again.

RENUMBER

Purpose: To renumber all line numbers (not labels)
Syntax: RENUMBER [dcnum1] [,dcnum2]

Description

Used to renumber all the editors line numbers (not label numbers). If no dcnums are specified the source program will be renumbered starting from line 10 in increments of 10. If dcnum1 is specified, this will be taken as the start lines number, and if dcnum2 is specified, this will be taken as the increment. The renumber command is mainly used to insert spaces between line numbers, so that lines can be inserted between them.

Example

This will renumber the current program from line 1000 in increments of 10.

```
RENUMBER 1000,10
```

RUN

Purpose: To execute the program without creating an object file.
Syntax: RUN

Description

This command is similar to the COMPILE command, except that no files may be given. When the compile is in run mode it must perform a third pass on the sourc program to determine a safe place to store the object code.
NOTE: If any errors are detected, the compile program will not be run.

SAVE

Purpose: To save a source program.
Syntax: SAVE #filespec

Description

Used to save a program so that you can resume editing it at a later stage.

SIZE

Purpose: To display remaining memory
Syntax: SIZE

Description

Used to find out how much memory is left for use with the COMPILER. If you start to run out of memory you can save the current file, and INCLUDE it in the main program, this will give more memory for the compiler to build the symbol tables.

AFTER

Purpose: After time period, go to address
Syntax: AFTER (*exp*, *adr*)

Description

When this command is executed it sets up one of the count down timers to count down from the value of *exp*. When the timer reaches zero control will be passed to *adr* which is an address of some machine code, or commands in this language. The called routine must execute a RETURN command to resume normal proccessing. If a return is not used the stack could quickly overflow causing the program to crash. To avoid this you could reset the stack pointer to $FF (LDX #$FF : TXS), but if this is done, it is the same as the program, being totally restarted from the current point (all return addresses will have been lost). The time may range from 1 to 65535 where 1 is 1/50th Second.

Example

The following example will switch on the sound channel and continue to process the rest of the program. Then after 10/50th of a second the program will call 'SOUNDOFF' which switches the sound off, and returns back to the main program.

```
SOUND (0,100,10,10)
AFTER (10,SOUNDOFF)

REST OF PROGRAM

POINTER SOUNDOFF
SOUND (0,0,0,0)
RETURN
```

ASCII

Purpose: To include ASCII text in program
Syntax: ASCII *string*

Description

This command enables ASCII text to be placed in the code, which can be used for filenames etc. The command must be positioned so that it will never be executed as this will make the program crash. If necessary jump over the text with a GOTO command.

Example

The following example will store the characters "D:DISKFILE.DAT" in memory. These characters are accessed by defining a pointer just before them. The program opens the disk file via the pointer.

```
        GOTO 10
        POINTER FILENAME
        ASCII "D:DISKFILE.DAT"
 10     OPEN (1,FILENAME,4,0)
```

BEGIN

Purpose: To define the start of the program
Syntax: BEGIN

Description

This command allows you to start the program execution from any point in the source code. There must be one, and only one BEGIN statement in the program. If at runtime the program reaches the associated END statement the program will either 1. Return to the language (if RUN command was used), 2. Return to DOS (if loaded from DOS) or 3. REBOOT (If booted from disk).

CLOSE

Purpose: To close an IOCB channel.
Syntax: CLOSE (exp)

Description

This command will close the specified IOCB, which will allow it to be used again at a later time. The exp must evaluate to between 0 and 7 although if it is out of range it will be truncated & no error will be issued.

Example

The following code ensures that channel one is closed before you try to re-open it.

```
        CLOSE (1)
        OPEN (1,FILENAME,4,0)
```

CODE

Purpose: To enter hexedecimal data.
Syntax: CODE hxnum [hxnum...]

Description

This command will enable you to enter hexedecimal data into memory at the current memory location (at compile time). This command enables you to include machine language directly in the program. Values can be passed to and from the routine using the PUSH & PULL commands.

Example

This places the code to change the screen colour to black in memory. The comments after give the assembler source code (not required).

```
        CODE A9008DC602
        ; LDA #0
```

```
; STA $2C6
```

COLOUR

Purpose: To select a new colour
Syntax: COLOUR (exp)

Description

This command (note the ENGLISH spelling) is used to select a new colour register. The colour is used to select what value will be stored on the screen. The graphics hardware will interpret this value differently from one graphics mode to another. In text modes it will select a character, and in graphics mode it will select a colour for each screen pixel. The pen colours may be changed by the SETCOLOUR command.

DO

Purpose: Repeat some code a given number of times
Syntax: DO var = exp1, exp2, exp3

Description

This command is used to repeat a loop a given number of times. The exp1 is the initial value, exp2 is the termination number, and exp3 is the increment the control variable takes each time round the loop. Note that an increment of 65535 is equal to -1 allowing you to count down.

Example

This will cause the 'X' variable (control variable), to take the values 1 through 10, and write them to the screen.

```
DO X=1,10,1
  WRITE (X)
END DO
```

DPOKE

Purpose: To store values in memory
Syntax: DPOKE (adr,exp)

Description

This command will store a 16 bit number denoted by the values of exp, into the specified memory address. The 16 bit number will be stored low byte first and high byte at adr+1.

Example

This will store the address of reset at memory address 2 and then change memory address 9 to indicate cassette boot. This will cause execution begining again at RESET when the reset key is pressed.

```
POINTER RESET
DPOKE (2,RESET)
POKE (9,2)
```

DRAWTO

Purpose: To draw a line on the screen
Syntax: DRAWTO (exp1,exp2)

Description

This command will draw a line from the current graphics cursor position, using the last colour selected to the location of exp1 and exp2. exp1 is the 'X' coordinate and exp2 is the 'Y' coordinate.

Example

This selects graphic mode 8 without a text window using colour 1. Then the program will fill the screen going from left to right.

```
GRAPHICS (8+16)
COLOUR (1)
DO X=0,319,1
  PLOT (X,0)
  DRAWTO (X,191)
END DO
```

ELSE

This command is used within a conditional IF block, and will cause all the code after it up to the END IF statement, to be executed, if the IF or all ELSE IF conditions evaluated to FALSE.

ELSE IF

This command is used within a conditional IF block, and will cause all the code following it, up to the next ELSE IF, ELSE or END IF (at the same level of nesting) to be executed, if the expression evaluates to TRUE, and the expression in the IF, and any preceding ELSE IF statements evaluated to FALSE. There must be a space between ELSE and IF.

END

Purpose: To execute a cold start/Terminate Function
Syntax: END

Description

This command is used to terminate a block of instructions. If it terminates a function, the function returns to the calling statement. If it is used outside a function (terminating the BEGIN statement) it will perform a coldstart.

END DO

Purpose: To define end of DO loop.
Syntax: END DO

Description

This command terminates the current DO loop. It must always be lower down the source program than its associated DO command. There must be a space between END and DO.

Example

This will loop round and take the values of X from 1 up to and including 10. Note that the loop will not be terminated until the X variable has had the value of ten when the END DO is reached. If the X increment was 2, the X variable would never reach 10 & therefore the loop would NEVER terminate.

```
DO X=1,10,1
  WRITE (X)
END DO
```

END IF

This command is used to terminate a conditional IF block. There must be a space between END and IF.

END WHILE

This command is used to terminate a WHILE exp DO block. When the expression in the WHILE command evaluates to FALSE the program continue after the END WHILE command. There must be a space between END and WHILE.

ERROR

Purpose: To force a runtime error
Syntax: ERROR exp

Description

This command will allow a your programs to generate a runtime error. This error is trapable in the normal way.

EVERY

Purpose: Every time period gosub address
Syntax: EVERY (exp, adr)

Description

This command is identical to the AFTER command exept that the interupt will occur every x/50ths of a second. The interrupt can be disabled by issuing the command with a time of zero.

FAST

Purpose: To increase the execution speed
Syntax: FAST

Description

This command will increase the execution speed of the program. It does this by disabling all interupts and by switching the screen display off. This should be used while the screen display is not required, and then by executing a SLOW command at the end.

Example

This program first switches the screen and interupts off. Sets the 'Y' variable to 0, and then adds 1 to 'Y' 5000 times. The screen and interupts are then re-enabled and the value of 'Y' is displayed on the screen.

```
FAST
Y=0
DO X=1,5000,1
  Y=Y+1
END DO
SLOW
WRITE (Y)
```

FILL

Purpose: Fill an area of memory with a value.
Syntax: FILL (adr, exp1, exp2)

Description

This command fills an area of memory from memory location adr for exp1 consecutive locations with the low byte value of exp2.

Example

This fills page 6 (memory locations 1536 to 1791 inclusive) with a value of zero.

```
FILL (1636,256,0)
```

FLOAT

Purpose: To place a floating point number into memory
Syntax: FLOAT const [,const ...]

Description

Allow you to place a floating point number into memory at the current memory address. The number can be located by placing a pointer variable directly before the FLOAT command. Although the compiler does not support floating point numbers, they can be implemented through the use of procedures and functions (see section on floating point procedures), but they cannot be used in expressions. ie. you could define an add command (ADD (destination,source)) which could add the floating point number at the address pointed to by source to the number pointed to by destination, with the result stored at destination.

FUNCTION

Purpose: User defined functions with arguments
Syntax: FUNCTION lblnm (var1 [,var2....])

Description

Since their are no inbuilt functions in the language, this command enable you to define your own. The function will be called lblnm and a variable of the same name will also be automaticaly set up to contain the result. The arguments will be given values when the routine is called. The result of the function is passed back to the calling routine by assigning the value to the functions name. The function is terminated by executing an END command.

Example

This will cause the value of 10 to be printed on the screen.

```
VARIABLE X
FUNCTION TEST (X)
TEST=X*2
END
;
VARIABLE Y
BEGIN
  Y=TEST(5)
  WRITE (Y)
END
```

GET

Purpose: To get recieve some data from an IOCB
Syntax: GET (exp1, adr, exp2)

Description

This command is used to receive some data from the IOCB specified by exp1, this may be in the range of 0-7 but will be truncated if required. The command will recieve exp2 bytes from the IOCB and store them in consecutive memory locations starting with adr.

Example

This program inputs 5 characters from the keyboard, and store them at 'KEYPRESS'. The characters will then be printed out via the screen editor.

```
        POINTER KEYBRD
        ASCII "K:"
        POINTER KEYPRESS
        RESERVE 5
        ;
        BEGIN
          OPEN (1,KEYBRD,4,0)
          GET (1,KEYPRESS,5)
          PRINT (0,KEYPRESS,5)
          CLOSE (1)
        END
```

GOSUB

Purpose: To call a subroutine.
Syntax: GOSUB lblnm

Description

Argh not GOSUB. I wouldn't even it in a language today! Anyway it enable you to call a label number directly. To return from the subroutine a RETURN command must be executed. This command can be useful to save rewriting similar pieces of code more than once.

Example

This program will first write out the value 1 followed by the value 2. In this case the END has the same effect as a return statement.

```
        VARIABLE X
        BEGIN
          X=1
          GOSUB 10
          X=2
     10   WRITE ("X = ",X)
        END
```

GOTO

Purpose: To change flow of program
Syntax: GOTO lbl

Description

Oh no! another one! This command executes in the same way as the GOSUB command, exept that it does not expect a RETURN command.

Example

This program loops until X is equal to 5, and then it goes to label 20 and write out the current value of 'X' (5) before returning control to DOS.

```
        VARIABLE X
        BEGIN
        X=0
     10 IF X<>5 THEN
          X=X+1
```

```
  GOTO 10
END IF
WRITE (X)
END
```

GRAPHICS

Purpose: To change screen mode
Syntax: GRAPHICS (exp)

Description

This command will call the operating system to set up the required graphics mode using IOCB 6. Normaly the screen will be set up in a split screen mode, but by adding 16 to exp, the screen will be set up without a split screen. Adding 32 to the exp will cause the screen to be set up without clearing the screen memory.

Example

This command will set up the high resolution one colour mode (320 by 192) without a text window using IOCB 6.

```
GRAPHICS (8+16)
```

IF

Purpose: To make decisions
Syntax: IF exp THEN (or DO)

Description

This command starts a conditional block of code. The first block will be executed if the expression is TRUE (non zero low byte value), otherwise the program will resume after the ELSE, ELSE IF (if exp true) or END IF

Example

```
VARIABLE X
BEGIN
  X=0
  WHILE X<3 DO
    X=X+1
    IF (X=1) THEN
      WRITE ("ONE")
    ELSE IF X=2 THEN
      WRITE ("TWO")
    ELSE
      WRITE ("THREE")
    END IF
  END WHILE
END
```

INCLUDE

Purpose: To include source programs from a file
Syntax: INCLUDE #filespec

Description

This command will INCLUDE the specified file name in with the code as it is being compiled. This can be used to bring in library routines, or it may be used to build large programs, where it is not all resident at the same time.

Example

This program includes the file D:PEEK.FUN into the main program at compile time (if it exists on the disk).

```
VARIABLE X
INCLUDE #D:PEEK.FUN
BEGIN
  X=PEEK (20)
  WRITE (X)
END
```

INPUT

Purpose: To input records terminated with an end of file.
Syntax: INPUT (exp1, adr, exp2)

Description

This command is used to input records terminated with an end of file marker. exp1 is the IOCB to use, adr is the address where the recieved data is to be stored and exp2 is the maximum number of bytes that can be in the record. Any extra bytes will be ignored.

Example

This program enters up to ten bytes from IOCB 0 (screen editor) and store them at TEXT. The program then PRINTs it out.

```
POINTER TEXT
RESERVE 10
BEGIN
  INPUT (0,TEXT,10)
  PRINT (0,TEXT,10)
END
```

IRQ OFF

Purpose: Disable interupt requests
Syntax: IRQ OFF

Description

This command ensures that no interrupts can occur. It disable both the EVERY and the AFTER interrupts. The non maskable interupts are not be affected.

IRQ ON

Purpose: Enable interput requests
Syntax: IRQ ON

Description

This command enables all the interrupt requests again.

LOCAL

Purpose: Define a new local variable region
Syntax: LOCAL

Description

This command defines the start of a new local variable region. A variable defined as being local is only available for use in the local region it was defined in. Up to 250 local regions may be used. Local variables are defined by preceding the variable name with a '?' character.

Example

This program defines two local regions both with the variable 'X' defined. In the first it is set to 2, and in the second it is set to 6. While the program is executing in the first local region 'X' will have the value of 2. Therefore the number 2 will be written to the screen by the WRITE statement.

```
        LOCAL
        VARIABLE ?X
        BEGIN
          ?X=2
          GOSUB 10
          WRITE (?X)
        END

        LOCAL
        VARIABLE ?X
     10 ?X=6
        RETURN
```

MOVE

Purpose: To move blocks of memory around
Syntax: MOVE (adr1, adr2, exp)

Description

This command moves exp bytes from adr1 to adr2. Care must be taken so that an overlap of adr1 and adr2 does not occur.

Example

This command will move all of page 6 into page 7, leaving page 6 un-change.

```
        MOVE (1536,1792,256)
```

OPEN

Purpose: Open a input / output file
Syntax: OPEN (exp1, adr, exp2, exp3)

Description

This commands opens an input or output file. exp1 is the IOCB to use, adr is an address that contains the name of the file to open, exp2 holds the input/output operation and exp3 holds a device dependent auxiliary code.
Valid values for exp2 are:-

```
  4  =  Input operation
  6  =  Disk directory operation
  8  =  Output operation
  9  =  End of file append (output)
 12  =  Input and output operations ( together )
```

Example

This procedure displays the disk directory on the screen using the supplied IOCB.

```
        LOCAL
```

```
        VARIABLE ?IOCHAN
        POINTER ?FSPEC
        ASCII "D:*.*"
        POINTER ?TEXT
        RESERVE 20
        PROCEDURE DIR (?IOCHAN)
          CLOSE (?IOCHAN)
          OPEN (?IOCHAN,?FSPEC,6,0)
          TRAP 65532
  65530   INPUT (?IOCHAN,?TEXT,20)
          PRINT (0,?TEXT,20)
          GOTO 65530
  65532   CLOSE (?IOCHAN)
          POP
          TRAP -1
        END
```

PLOT

Purpose: To plot a single point on the screen
Syntax: PLOT (exp1, exp2)

Description

This plots a single point on the screen using the last selected colour. exp1 holds the 'X' coordinates and exp2 holds the 'Y' coordinates.

Example

This program set up graphics mode 16 and then draw bands of colour going from the top to the bottom of the screen.

```
        VARIABLE X
        BEGIN
          GRAPHICS (11)
          DO X=0,79,1
            COLOUR (X MOD 16)
            PLOT (X,0)
            DRAWTO (X,191)
          END DO
     10   GOTO 10
        END
```

POINTER

Purpose: To locate a section of the compiled code.
Syntax: POINTER lblnm

Description

This command defines a variable, but instead of the variable being initialised to zero, it is initialised to the value of the next address. This allows you to locate parts of the compiled code (text, data and buffers etc).

Example

This program will open a file called D:FILE.DAT, and then continue and execute the rest of program.

```
        POINTER Filename
        ASCII "D:FILE.DAT"
        BEGIN
          OPEN (1,Filename,4,0)
```

```
  Rest of program

  CLOSE (1)
END
```

POKE

Purpose: To store values in memory
Syntax: POKE (adr,exp)

Description

This command stores an 8 bit number denoted by the low byte of exp at memory address adr.

Example

This stores a zero at memory address 710, which in this case turns the screen black.

```
POKE (710,0)
```

POP

Purpose: To clear values off stack
Syntax: POP

Description

This command pops a 16 bit number off the stack. This can be used to remove unwanted return addresses or parameters from stack.

Example

In this case the POP command has the effect of turning the GOSUB 10 into a GOTO 10 (ie. once it has been POPed off the stack you cannot return).

```
GOSUB 10

rest of program

10 IF (EXIT) THEN
     RETURN
   END IF
   POP

   continue program here
```

PRINT

Purpose: To print out a buffer
Syntax: PRINT (exp1, adr, exp2)

Description

This command prints a buffer to a given device. The buffer is considered to be empty when either an end of file is found in the data or when the maximum number of bytes has been sent. exp1 holds the number of the IOCB to use, adr holds the address of the data buffer, and exp2 holds the maximum number of bytes in the buffer.

Example

This program enters up to 10 characters from the screen editor, and output up to 10 characters to the screen editor.

```
POINTER BUFFER
RESERVE 10
```

```
BEGIN
  INPUT (0,BUFFER,10)
  PRINT (0,BUFFER,10)
END
```

PROCEDURE

Purpose: To add new commands to the language
Syntax: PROCEDURE lblnm (var1 [,var2...])

Description

Defines the start of a new command. The new command may be called just like any of the resident commands (ie. by typing its name followed by correct number of arguments). A procedure is terminated by an END statement.

Example

This procedure allows you to directly write an expression to the screen. Normally you would have to calculate the value in a tempoary variable before writing it out.

```
LOCAL
VARIABLE ?EXP
PROCEDURE WriteEXP (?EXP)
  WRITE (?EXP)
END
;
BEGIN
  WriteEXP (5+6*7*100/2)
END
```

PULL

Purpose: To obtain values off the stack
Syntax: PULL (var1 [,var2...])

Description

This command causes a number of 16 bit values to be pulled off the stack (one for each variable specified). The first number pulled off the stack will be used as the high byte and the second byte as the low byte.

Example

This has the effect of swapping the values of 'A' and 'B'. (since 'A' is pushed on First and 'B' is pushed on second, 'B' is the first one pulled off, and it is stored in 'A')

```
PUSH (A,B)
PULL (A,B)
```

PUSH

Purpose: To place 16 bit values on stack
Syntax: PUSH (exp1 [,exp2...])

Description

This command will allow you to be able to push values onto the 6502 stack for use by your machine language routines etc. The low byte will be stored on the stack first followed by the high byte of the expression. For an example see the previous command (PULL).

PUT

Purpose: To send some data to a file
Syntax: PUT (exp1, adr, exp2)*

Description

This command will put the data starting at memory address adr for exp2 consecutive memory locations to the IOCB specified by exp1.

Example

This will send an EOL (End of line) character to the editor.

```
POINTER Return
CODE 9B
BEGIN
  PUT (0,Return,1)
END
```

REPEAT

Purpose: Repeats a loop until a condition is met.
Syntax: REPEAT

Description

This command is used in conjunction with the UNTIL command, and allows you to repeat all the lines between the 'REPEAT' command and the 'UNTIL' command until a given condition evaluates to true (1).

Example

This causes the loop to be repeated until the 'X' variable has a value of 10.

```
REPEAT
  X=X+1
  WRITE (X)
UNTIL X=10
```

RESERVE

Purpose: To reserve some memory
Syntax: RESERVE dcnum

Description

This reserves a specified number of bytes within the compiled program for use as buffers etc.

Example

This will reserve 10 bytes in memory, there are no restrictions as to what this space is to be used for, but the program must never execute this statement, unless you know there is a machine langauge file stored there. Usually this command will be preceded with a POINTER definition so that you can locate the reserved memory.

```
RESERVE 10
```

RETURN

Purpose: Return from a routine
Syntax: RETURN

Description

From inside the main program (not in a subroutine etc) this command will return you to DOS. This command is also used to return from a GOSUB.

SETCOLOUR

Purpose: Change a colour registers value
Syntax: SETCOLOUR (exp1, exp2, exp3)

Description

This command is used to define a new colour for the specified colour register. exp1 holds the colour register (0 to 8) where register 0 is address 704, 1 is address 705 etc. exp2 holds the new colour and exp3 holds the luminance of the colour. Only the lower 4 bits of both exp2 and exp3 have any significace.

SLOW

Purpose: Enables screen display and interupts
Syntax: SLOW

Description

This command is used to bring the computer out of fast mode. (ie. switch the display and interupts on again). For an example see the FAST command.

SOUND

Purpose: Control sound voices
Syntax: SOUND (exp1, exp2, exp3, exp4)

Description

This enables you to use the ATARI's sound registers. exp1 holds the voice number (0 to 4), exp2 holds the frequency (0 to 255), exp3 holds the distortion level (0 to 15) and exp4 holds the volume (0 to 15)

TRAP

Purpose: To trap input / output errors
Syntax: TRAP (lno)

Description

This command re-direct the program to your error handling routine should an error be detected. The error code will be stored at location 195 (as in basic). If the supplied label number is invalid or missing, the default error routine is used. This default handler outputs a message giving the error number and the approximate address where the error occured. With reference to the compilation address map you should be able to identify the approximatly line in error. A user defined error handler may return to the source line after the error by executing a RETURN command. If the program should continue without returning to the line after the error, you must execute a POP command to clear the stack (as in the definition of the DIR procedure). The only statement that you can't return to is after it has happend in a WRITE statement.

UNTIL

Purpose: Terminate REPEAT UNTIL loop
Syntax: UNTIL exp

Description

This command is used to terminate a REPEAT UNTIL loop structure, and can be used so that a loop will REPEAT, UNTIL a condition is true (1).

Example

This loop will be repeated until the low byte value of CONSOL is equal to 6 (ie. the start key has been pressed)

```
VARIABLE 53279=CONSOL
BEGIN
  REPEAT
    rest of loop
  UNTIL (CONSOL .AND. 255)=6
END
```

VARIABLE

Purpose: To manipulate numeric data.
Syntax: VARIABLE [dcnum=]lblnm [,[dcnum=]lblnm...]

Description

This command is used to define some variables. A variable is a name that represents a numeric value. This numeric value may be changed by an assignment statement. If the lblnm (label name) is preceded by 'dcnum=' then this dcnum will be taken as the address of the variable. If dcnum is not specified, then the compiler will select its own memory address. Defining your own memory locations for the variable will give you the ability to manipulate these locations. Care must be taken when using these variables as you must take into consideration the other half of the word (if you are only using a byte). A local variable is defined by the first character of the variable name being a '?' character.

Example

```
VARIABLE 53279=CONSOL, 53248=HPOSP0
```

To access low byte of CONSOL you must mask out the high byte with a bitwise '.AND.'opperation:

```
VALUE = CONSOL .AND. 255
```

To access the high of CONSOL you must divide divide by 256:

```
VALUE = CONSOL / 256
```

To change the low byte (only) of HPOSP0 (53248), their is a special character that you place in from of the variable that is about to be assigned:

```
<HPOSP0=25 PRE <>
```

the high byte is change in the same way but with a different character preceding the variable name. In this case address 53249:

```
>HPOSP0=25
```

In addition, the result of the expression may be stored at the addressxi pointed to by the value of the variable by preceding the variable name with an '@' character, you may still use the '<' or '>' character as well, to change only the low byte or high byte of the address pointed to by the variable.

Example

to fill a block of memory without the fill command you may use any of the following examples.

```
VARIABLE ADR
BEGIN
  DO ADR=1536,1791,1
```

```
    <@ADR=0
  END DO

  DO ADR=1536,1790,2
    @ADR=0
  END
END
```

WHILE

Purpose: Execute a block of code while the expression is TRUE
Syntax: WHILE exp DO

Description

This commad will first evaluate the expression, if the result is TRUE (low byte of result is non zero) the following code up to the END WHILE at the same nesting level will be executed, the program will then loop round and test the expression again. If the expression is FALSE, the program will continue on te line following the END WHILE command.

Example

```
VARIABLE 53279=CONSOL
BEGIN
  WHILE (CONSOL.AND.255)=7 DO
    WRITE ("PRESS OPTION, SELECT OR START")
  END WHILE
END
```

WRITE

Purpose: Write data and variables to device
Syntax: WRITE [dcnum,] (string/var/const[,string/var/const...]) [;]

Description

This command will write out the data to the IOCB specified by dcnum. If dcnum is not given the default IOCB is zero (screen editor). All items to be printed must be seperated by commas. If there is a ';' character following the close bracket, no end of line will be written to the device. Normally when writting the value of a variable, the value will be integer but by preceding the write statement with a F (FWRITE) all the following variables will be taken as pointers to floating point numbers, and the floating point number will be printed out instead.

Example

This short program will write out 'The value of 'A' is 25' twice to the default device (screen editor). Note the use of the ';' for the second WRITE command. Then the program then prints 3.142.

```
VARIABLE A
POINTER PI
FLOAT 3.142
BEGIN
  A=25
  WRITE ("The value of 'A' is ",A)
  WRITE ("The value of 'A' is ");
  WRITE (A)
  FWRITE ("PI = ",PI)
END
```

Introduction to Assembler Commands

There are several assembler instructions defined within the compiler to make it easier to interface with machine code routines. For example:

```
JSR $E456 ( in decimal )
TYA
PHA
PHA
PULL (STATUS)
IF (STATUS .AND. 255)<>136 THEN
  RETURN
END IF
ERROR 136
```

This example calls address $E456 and pushes the status onto the stack. The pull command then places it in the variable 'STATUS' and test the low byte for being 136 (end of file). If the end of file has been reached a runtime error will be generated. These commands are exactly the same as the assembler commands of the same names. Reference 6502 manual for further details.
NOTE: Since it will not be clear where in memory your machine code will go, any machine code must be re-locatable. It is possible to locate the code by preceding it with a pointer definition, the defined variable will contain the address of the code.

JMP

Purpose: Jump to a machine language routine
Syntax: JMP dcnum

JSR

Purpose: Jumps to a machine language subroutine
Syntax: JSR dcnum

PHA

Purpose: Pushes accumulator onto stack
Syntax: PHA

PHP

Purpose: Pushes status register onto stack
Syntax: PHP

PLA

Purpose: Pulls accumulator off stack
Syntax: PLA

PLP

Purpose: Pulls status register off stack
Syntax: PLP

RTI

Purpose: Return from non-maskable interupt
Syntax: RTI

TAX

Purpose: Transfers 'A' into 'X'
Syntax: TAX

TAY

Purpose: Transfers 'A' into 'Y'
Syntax: TAY

TSX

Purpose: Transfers stack pointer into 'X'
Syntax: TSX

TXA

Purpose: Transfers 'X' into 'A'
Syntax: TXA

TXS

Purpose: Transfers 'X' into stack pointer
Syntax: TXS

TYA

Purpose: Transfers 'Y' into 'A'
Syntax: TYA

Operators

An expression is a valid combination of operands and operators which the compiled object code will evaluate to a 16 bit unsigned integer number. No error detection is made for an overflow. Since each variable is a 2 byte word (16 bits) each variable can hold a number from 0 to 65535 inclusive.

Operator: ( )

These operators will force everything inside the brackets to be evaluated before anything outside them. This will enable you to force the expressions into being evaluated in a different order.

Operator: [ ]

These operators will force everything in the brackets to be executed in the order they are given. The brackets may not be nested, and there may only be the '+' and '-' operator's included in the brackets, along with constants and variables, in addition there are no function calls allowed. These operators where added to provide a facility for very fast addition and subtraction of numbers. The stack is not used at all resulting in very fast code.

Operators: * / MOD + -

These operators are the normal arithmetic operators.

```
'*'   Multiplication
'/'   Division
'MOD' Returns remainder
'+'   Addition
'-'   Subtraction)
```

They perform 16-bit unsigned integer arithmetic, and will ignore overflows.

Operators: = =< < <> > >=

These operators are used for comparisons and will return one if the result is true, and zero if the result is false.

```
'='        .EQ.  is equal to.
'=<' '<='  .LE.  is less than or equal to
'<'        .LT.  is less than
'<>'       .NE.  is not equal to
'>'        .GT.  is greater than
'>=' '=>'  .GE.  is greater than or equal to
```

Operators: AND OR EOR

- AND will return a value of 1 if both operators are not equal to zero.
- OR will return a value of 1 if either of the operators is not zero.
- EOR will return a value of 1 if only one of the operators is not zero.

Example:

```
0 AND 0     0
0 AND 1     0
1 AND 0     0
1 AND 1     1

0 OR 0      0
0 OR 1      1
1 OR 1      1
1 OR 1      1

0 EOR 0     0
0 EOR 1     1
1 EOR 0     1
1 EOR 1     0
```

Operators: .AND. .OR. .EOR.

These operators work in the same way as AND OR & EOR but they work on each bit in the operands.

Example:

```
%0011 AND %0101   %0001
%0011 OR  %0101   %0111
%0011 EOR %0101   %0110
```

Note that the % means that the numbers following are in binary. The compiler does NOT support binary.

Operator Precedence

The following are the precedence levels of each of the opperators. These are used to decide which operator should be executed first.

4. ( ) [ ]
5. * / MOD
6. + -
7. = =< < <> >= > etc.

8. AND OR EOR
9. .AND. .OR. .EOR.

DPEEK

Purpose: To look at a word address (2 bytes)
External: PEEK Function
Syntax: var = DPEEK (adr)

Description

This function will return the value of the memory address. The value will be the 16 bits found in the specified address and the following address. Address holds low byte and address+1 holds high byte.

```
LOCAL
VARIABLE ?ADR
FUNCTION DPEEK (?ADR)
  DPEEK = PEEK(?ADR)+256*PEEK(?ADR+1)
RETURN
```

LOCATE

Purpose: To return contents of screen location
External: None
Syntax: var = LOCATE (exp1,exp2)

Description

This function will return the contents of the screen location specified by exp1 and exp2. Exp1 is the 'X' position and exp2 is the 'Y' position

```
LOCAL
VARIABLE ?XPOS,?YPOS
FUNCTION LOCATE (?XPOS,?YPOS)
  PUSH (?XPOS,?YPOS)
  CODE 686885546885
  CODE 56688555A260
  CODE A9009D48039D
  CODE 4903A9079D42
  CODE 032056E448A9
  CODE 0048
  PULL (LOCATE)
END
```

NOT

Purpose: To return the logical NOT of an expression.
Syntax: var = NOT (exp)

Description

This function returns the logical NOT of exp, (ie. non zero returns TRUE (1) and zero returns FALSE (0)).

```
LOCAL
VARIABLE ?NUM
FUNCTION NOT (?NUM)
  NOT=1*(?NUM<>0)
END
```

PADDLE

Purpose: To return the position of a paddle
External: PEEK Function
Syntax: var = PADDLE (exp)

Description

This function will return the value of one of the paddles specified by the lowest 3 bits of exp.

```
LOCAL
VARIABLE ?NUM
FUNCTION PADDLE (?NUM)
  PADDLE = PEEK(624+(?NUM .AND. 7))
END
```

PEEK

Purpose: To look at a byte address
External: None
Syntax: var = PEEK (adr)

Description

This function will return the 8 bit value that is stored at memory location specified by adr.

```
LOCAL
FUNCTION PEEK (?ADR)
  CODE AD
  VARIABLE ?ADR
  PHA
  CODE A900
  PHA
  PULL (PEEK)
END
```

Note the trick used by defining the variable after the LDA opcode. The variable is assigned before the LDA instruction which then loads the value from the address specified by the variable.

PTRIG

Purpose: To return the status of a paddle trigger
External: PEEK Function
Syntax: var = PTRIG (exp)

Description

This function will return a 0 if the specified trigger is pressed and a 1 if it is not pressed. The required trigger is specified by the lower 3 bits of exp.

```
LOCAL
VARIABLE ?NUM
FUNCTION PTRIG (?NUM)
  PTRIG = PEEK(636+(?NUM .AND. 7))
END
```

RND

Purpose: To generate a random number
External: None
Syntax: var = RND (exp)

Description

This function will return a random number between 0 and the value of exp-1.

```
LOCAL
VARIABLE ?NUM,53770=?RANDOM
FUNCTION RND (?NUM)
  RND=(?RANDOM*256+(?RANDOM .AND. 255)) MOD ?NUM
END
```

STICK

Purpose: Return joystick position
External: PEEK Function
Syntax: var = STICK (exp)

Description

This function will read the position of one of the joysticks. The joystick to be read is specified by the lowest 2 bits of exp.

```
LOCAL
VARIABLE ?NUM
FUNCTION STICK (?NUM)
  STICK = PEEK(632+(?NUM .AND. 3))
END
```

STRIG

Purpose: To return the status of a joystick trigger
External: PEEK Function
Syntax: var = STRIG (exp)

Description

This function will return the status of the specified joystick trigger, 0 means trigger is pressed and 1 means trigger is not pressed. The trigger number is specified by the lower 2 bits of exp.

```
LOCAL
VARIABLE ?NUM
FUNCTION STRIG (?NUM)
  STRIG = PEEK(644+(?NUM .AND. 3))
END
```

VAL

Purpose: Returns numeric value of text data
External: None
Syntax: var = VAL (adr)

Description

This function returns the numeric value of the characters pointed to by adr.

```
LOCAL
```

```
VARIABLE ?ADR
FUNCTION VAL (?ADR)
  PUSH (?ADR)
  CODE 6885F46885F3
  CODE A90085F22000
  CODE D8B00520D2D9
  CODE 9006A90085D4
  CODE 85D5A5D448A5
  CODE D548
  PULL (VAL)
END
```

Example

This would set the variable 'A' equal to 45 and then write it out onto the screen.

```
POINTER NUMBER
ASCII "45"
CODE 9B
VARIABLE A
BEGIN
A=VAL(NUMBER)
WRITE (A)
RETURN
```

Floating point support procedures

Since the language does not have any inbuilt floating point procedures, here is a short program that will allow you to do limited calculations with floating point numbers. The number can only be operated on by one opperation at a time using the following procedures. If an error is detected by the floating point routines then an error number of 1 will be issued.
Note that FR1 & FR2 are both pointers to a floating point numbers, these numbers are printed out by using the 'FWRITE' command.

ADD

Purpose: To add 2 floating point numbers together
Syntax: ADD (FR1,FR2)

Description

This procedure will add FR2 to FR1 and store the result in FR1.

DIV

Purpose: To divide 2 floating point numbers
Syntax: DIV (FR1,FR2)

Description

This procedure will divide FR1 by FR2 and store the result in FR1.

EQUALS

Purpose: To set a floating point number equal to another
Syntax: EQUALS (FR1,FR2)

Description

This procedure will set FR1 equal to the value of FR2.

EXP

Purpose: To return the base 10 exponential
Syntax: EXP (FR1)

Description

This procedure will return the base 10 exponential and store the result in FR1.

EXPe

Purpose: To return the natural exponential
Syntax: EXPe (FR1)

Description

This procedure will return the natural exponential and store the result in FR1.

LOG

Purpose: To return the base 10 logarithm
Syntax: LOG (FR1)

Description

This procedure will take the base 10 logarithm of FR1 and store the result back into FR1.

LOGe

Purpose: To return the natural logarithm.
Syntax: LOGe (FR1)

Description

This procedure will take the natural logarithm of FR1 and store the result back into FR1.

MULT

Purpose: To multiply 2 floating point numbers together
Syntax: MULT (FR1,FR2)

Description

This procedure will multiply FR1 by FR2 and store the result in FR1.

SUB

Purpose: To subtract 2 floating point numbers
Syntax: SUB (FR1,FR2)

Description

This procedure will subtract FR2 from FR1 and store the result in FR1.

Listing of floating point support package

```
;
; ================
;  Floating point
; support routines
; (c) 8th May 1986
; ================
```

```
;
LOCAL
VARIABLE ?FP1,?FP2,2=?ADR,?NUM,?STATUS
;
; ======================
; Floating point addition
; ======================
;
PROCEDURE ADD (?FP1,?FP2)
  FP (2,55910)
END
;
; ==========================
; Floating point subtraction
; ==========================
;
PROCEDURE SUB (?FP1,?FP2)
  FP (2,55904)
END
;
; ============================
; Floating point multiplication
; ============================
;
PROCEDURE MULT (?FP1,?FP2)
  FP (2,56027)
END
;
; ======================
; Floating point division
; ======================
;
PROCEDURE DIV (?FP1,?FP2)
  FP (2,56104)
END
;
; =========
; FP1 = FP2
; =========
;
PROCEDURE EQUALS (?FP1,?FP2)
  MOVE (?FP2,?FP1,6)
END
;
; ==================
; Floating point LOGe
; ==================
;
PROCEDURE LOGe (?FP1)
  FP (1,57037)
END
;
; =================
; Floating point LOG
; =================
;
PROCEDURE LOG (?FP1)
```

```
        FP (1,57041)
      END
      ;
      ; ==================
      ; Floating point EXPe
      ; ==================
      ;
      PROCEDURE EXPe (?FP1)
        FP (1,56768)
      END
      ;
      ; ==================
      ; Floating point EXPe
      ; ==================
      ;
      PROCEDURE EXP (?FP1)
        FP (1,56780)
      END
      ;
      ; ==========================
      ; Call floating point package
      ; ==========================
      ;
      PROCEDURE FP (?NUM,?ADR)
        MOVE (?FP1,212,6)
        IF (?NUM<>1) THEN
          MOVE (?FP2,224,6)
        END IF
        GOSUB 65020
        PHP
        PHP
        PULL (?STATUS)
        IF (?STATUS .AND. 1)<>0 THEN
          ERROR 1
        END IF
        MOVE (212,?FP1,6)
      END
65020 CODE 6C0200
```

Compiler Directives

;

Purpose: To place comments within the program
Syntax: ; Any characters

Description

This directive will enable you to place comments within your source program. These comments will be ignored when the program is compiled, and will not be included in the object code.

.LIST

Purpose: To enable listing during compilation
Syntax: .LIST

Description

This directive tells the compiler to generate a listing. If a list device is specified in the COMPILE statement. This is the default condition.

.NOLIST

Purpose: To stop a listing from being produced.
Syntax: .NOLIST

Description

This directive stops the listing from being produced. It can be turned back on by using the .LIST directive.

.MAP

Purpose: To produce a memory map of program
Syntax: .MAP

Description

This directive tells the compiler to produce a memory map. A memory map will list the names of global variables, functions, procedures and line labels together with the address in memory. The labels will be listed in the order they where defined.

.NOMAP

Purpose: To stop the memory map from being produced.
Syntax: .NOMAP

Description

This command will stop the memory map from being produced at the end of the compilation. The option can be switched back on by a .MAP instruction anywhere else up to the end of the program.

Memory Usage

The compiler generates a non relocatable object file, the code is also generated in a way where the variables are embeded within the object program, therefore it is not possible to place the code into a ROM or EPROM. The first thing the compiler does is write a 1.8K run time package to the output file prior to generating the object code. The following tables gives a description of memory usage at run time.

```
    Page Zero:

        $0000  to  $007F    Used by Operating system
        $0080  to  $0097    Used by Compiled program
        $0098  to  $00AF    Used if Every or After is used
        $00BO  to  $00C2    Free memory ( COMPILE MODE )
        $00C0  to  $00C2    Free memory ( RUN MODE )
        $00C3               Last error number
        $00C4  to  $00FF    Free memory
```

```
        $0100  to  $01FF    Stack for everything ( Return addresses &
arguments etc )
        $0200  to  $05FF    Used by Operating system
        $0600  to  $06FF    Free memory ( COMPILE MODE )
        $0680  to  $06FF    Free memory ( RUN MODE )
        $0700  to  APPMHI   DOS & Compiled program
        APPMHI to  MEMTOP   Screen displays
```

Note that if you use the resident floating point package, some extra page zero addresses will be used (See Atari's Operating System manuals for more details).

Appendix A - Limitations

Depth of Stack

Since this program uses the 6502 hardware stack for virtualy all data movements and control transfers, it is advisable not to nest procedures and functions to deep as the stack will quickly fill up. If the stack does overflow, the program cannot detect it. Therefore it is VERY important that if you jump out of a subroutine you pop off the return address or addresses.

The stack is used for evaluation of expressions, passing parameters to procedures or functions & holding return addresses, as well as any data you may have PUSHed on. In general if you don't jump out of a subroutine, procedure or function you should not have any problems.

WRITE command

In this command the arguments may not be an expression. They may only be constants or variables.

AFTER & EVERY commands

Although you may use any of the predefined procedures in this language within the portion of code called by the EVERY and AFTER commands, you should not use any that are not resident, unless you can be sure that the called function or procedure was not in use when the interrupt occured.

If you use a FUNCTION in both the main code and in the EVERY or AFTER routine, then the FUNCTION will probably return incorrect results. This is because the functions use statically allocated variables. When the interrupt routine calls the FUNCTION, the FUNCTION will lose all its previous values that it was in the middle of calculating.

Appendix B - Error Codes

10. Start of program (BEGIN) not defined
11. Out of memory
12. Number out of range (less than 0 or greater than 65535)
13. Label not defined
14. Variable not defined
15. Incorrect number of parameters
16. Quatation marks missing

17. Error in expression
18. Invalid expression
19. Invalid character
20. Constant expected
21. Requested line number missing
22. Syntax error in line
23. Procedure not defined
24. Duplicate label
25. Construct nested to deep
26. To many local regions (greater than 250)
27. Nested INCLUDE requested
28. Begin already defined
29. Duplicate label number
30. Function not defined
31. Operators expected
32. Variable or Constant expected
33. Invalid nesting of IF and WHILE commands. eg IF exp THEN ... END WHILE

Appendix C - Known Bugs

34. No checking is made for running out of memory when building symbol tables, if characters start to appear on screen this has probably occured. In this case save the source program to disk and write a small routine to INCLUDE it in. This will give more memory for use as a symbol table.
35. No checking is made for passing the correct number of arguments to a function or procedure, and if these are incorrect it could crash the program from either filling the stack up or pulling the return address off stack.

Appendix D – Breakout Program

```
1000 ;
1010 ; **********************
1020 ; * *
1030 ; * Break Out *
1040 ; * ========== *
1050 ; * *
1060 ; * (c) Copyright 1986 *
1070 ; * by David Firth *
1080 ; * *
1090 ; * 6th April 1986 *
1100 ; * *
1110 ; **********************
1120 ;
1130 ;
1140 ; ========================
1150 ; Define system variables
1160 ; ========================
1170 ;
1180 VARIABLE 77=ATRACT
1190 VARIABLE 88=SCREENADR
1200 VARIABLE 106=RAMTOP
1210 VARIABLE 512=VDSLST
1220 VARIABLE 559=DMACTL
1230 VARIABLE 560=DLIST
1240 VARIABLE 624=PADDLE
1250 VARIABLE 636=PTRIG
1260 VARIABLE 751=CURSOR
```

```
1270 VARIABLE 756=CHBASE
1280 VARIABLE 53248=HPOSP0
1290 VARIABLE 53249=HPOSP1
1300 VARIABLE 53250=HPOSP2
1310 VARIABLE 53251=HPOSP3
1320 VARIABLE 53252=HPOSM0
1330 VARIABLE 53253=HPOSM1
1340 VARIABLE 53254=HPOSM2
1350 VARIABLE 53255=HPOSM3
1360 VARIABLE 53270=COLPF0
1370 VARIABLE 53277=GRACTL
1380 VARIABLE 53278=HITCLR
1390 VARIABLE 53279=CONSOL
1400 VARIABLE 53770=RANDOM
1410 VARIABLE 54279=PMBASE
1420 VARIABLE 54282=WSYNC
1430 VARIABLE 54283=VCOUNT
1440 VARIABLE 54286=NMIEN
1450 ;
1460 ; ======================
1470 ; Define Global variables
1480 ; ======================
1490 ;
1500 VARIABLE Pmbase,MISSILES
1510 VARIABLE PLAYER0,PLAYER1
1520 VARIABLE PLAYER2,Dummy
1530 VARIABLE XPOS,YPOS
1540 VARIABLE XDIR,YDIR
1550 VARIABLE XTIME,YTIME
1560 VARIABLE TEMP,BATPOS
1570 VARIABLE SCORE,HIGH
1580 VARIABLE HITLAST,Bricks
1590 POINTER HIGHDATA
1600 CODE 10101010
1610 ;
1620 ; ====================
1630 ; Include library files
1640 ; ====================
1650 ;
1660 INCLUDE #H:PEEK.FUN
1670 INCLUDE #H:DPEEK.FUN
1680 ;
1690 ; =============
1700 ; Plot the Ball
1710 ; =============
1720 ;
1730 LOCAL
1740 VARIABLE ?X,?Y
1750 ;
1760 PROCEDURE BALL (?X,?Y)
1770   MOVE (BALL,[MISSILES+?Y],4)
1780   <HPOSM1=?X
1790 END
1800 ;
1810 ; =================
1820 ; Add ?VAL to score
1830 ; =================
```

```
1840 ;
1850 LOCAL
1860 VARIABLE ?VAL,?OFFSET,?COUNT,?TEMP
1870 PROCEDURE ADD (?VAL,?OFFSET)
1880   ?COUNT=4
1890   REPEAT
1900     ?TEMP=PEEK([SCREENADR+?OFFSET])+?VAL
1910     ?VAL=0
1920     IF ?TEMP=26 THEN
1930       ?TEMP=[?TEMP-10]
1940       ?VAL=1
1950     END IF
1960     POKE ([SCREENADR+?OFFSET],?TEMP)
1970     ?OFFSET=[?OFFSET-1]
1980     ?COUNT=[?COUNT-1]
1990   UNTIL ?COUNT=0 OR ?VAL=0
2000 END
2010 ;
2020 ; ====================
2030 ; Switch the sound off
2040 ; ====================
2050 ;
2060 POINTER SOUNDOFF
2070   SOUND (0,0,0,0)
2080   RETURN
2090 ;
2100 ; =================
2110 ; Data for the ball
2120 ; =================
2130 ;
2140 POINTER BALL
2150 CODE 000C0C00
2160 ;
2170 ; ===================
2180 ; Change Display List
2190 ; and start VBI & DLI
2200 ; New character set.
2210 ; ===================
2220 ;
2230 PROCEDURE GRAPHMOD (Dummy)
2240   MOVE (NEWDLIST,DLIST+7,22)
2250   TEMP=((RAMTOP.AND.255)-12)*256
2260   MOVE ((CHBASE.AND.255)*256,TEMP,1024)
2270   FILL (TEMP+8,7,85)
2280   POKE (TEMP+15,0)
2290   <CHBASE=TEMP/256
2300   <NMIEN=0
2310   VDSLST=DLI
2320   <NMIEN=192
2330 END
2340 ;
2350 ; =================
2360 ; Data for modified
2370 ; Display list.
2380 ; =================
2390 ;
2400 POINTER NEWDLIST
```

```
2410 CODE 040484848484
2420 CODE 848404040404
2430 CODE 040404040404
2440 CODE 04040404
2450 ;
2460 ; ====================
2470 ; Display List Interupt
2480 ; ====================
2490 ;
2500 POINTER DLI
2510   PHA
2520   <COLPF0=VCOUNT
2530   PLA
2540   RTI
2550 ;
2560 ; =================
2570 ; Display the Title
2580 ; =================
2590 ;
2600 PROCEDURE TITLE (Dummy)
2610   GRAPHICS (18)
2620   FILL (53248,8,0)
2630   SETCOLOUR (4,5,10)
2640   SETCOLOUR (5,7,10)
2650   SETCOLOUR (6,9,10)
2660   SETCOLOUR (7,12,10)
2670   SETCOLOUR (8,0,0)
2680   WRITE 6,(" BREAK OUT")
2690   WRITE 6,(" ")
2700   WRITE 6,(" tcl demonstration")
2710   WRITE 6,(" ")
2720   WRITE 6,(" ¶ÔÈ ÁÐÒÉÌ ±¹¸¶")
2730   WRITE 6,(" ")
2740   WRITE 6,(" INSERT PADDLES")
2750   WRITE 6,(" ")
2760   WRITE 6,("ðòåóó óôáòô ôï âåçéî")
2770   REPEAT
2780   UNTIL ((CONSOL.AND.255)=6) OR ((PTRIG.AND.255)=0)
2790 END
2800 ;
2810 ; ==================
2820 ; Initialize program
2830 ; ==================
2840 ;
2850 PROCEDURE INIT (Dummy)
2860   POKE (82,0)
2870   GRAPHICS (0)
2880   GRAPHMOD (0)
2890   <CURSOR=1
2900   WRITE ("SCORE 0000 HIGH SCORE 0000 BALLS 3");
2910   WRITE ("________________________________________")
2920   Pmbase=(RAMTOP.AND.255)-8
2930   MOVE (HIGHDATA,SCREENADR+25,4)
2940   <PMBASE=Pmbase
2950   Pmbase=Pmbase*256
2960   FILL (Pmbase,1024,0)
2970   SETCOLOUR (0,7,10)
```

```
2980   SETCOLOUR (1,0,10)
2990   SETCOLOUR (2,0,10)
3000   SETCOLOUR (4,5,8)
3010   SETCOLOUR (6,0,0)
3020   MISSILES=Pmbase+384
3030   PLAYER0=Pmbase+624
3040   PLAYER1=Pmbase+663
3050   PLAYER2=Pmbase+791
3060   FILL (PLAYER1,93,3)
3070   FILL (PLAYER2,93,192)
3080   FILL (PLAYER0,4,255)
3090   <HPOSP1=40
3100   <HPOSP2=208
3110   <DMACTL=46
3120   <GRACTL=3
3130 END
3140 ;
3150 ; ==================
3160 ; Re-Draw the Bricks
3170 ; ==================
3180 ;
3190 PROCEDURE BRICKS (Dummy)
3200   FILL (SCREENADR+200,240,1)
3210   FILL(MISSILES+YPOS,4,0)
3220   SETBALL (0)
3230   <HITCLR=0
3240   Bricks=120
3250 END
3260 ;
3270 ; ========================
3280 ; Set balls start position
3290 ; ========================
3300 ;
3310 PROCEDURE SETBALL (Dummy)
3320   XPOS=128
3330   YPOS=75
3340   XDIR=1-2*((RANDOM.AND.255)<128)
3350   YDIR=1
3360   XTIME=((RANDOM.AND.255) MOD 3)+3
3370   YTIME=10
3380 END
3390 ;
3400 ; ====================
3410 ; Move the Bat, this is
3420 ; an Interrupt task.
3430 ; ====================
3440 ;
3450 POINTER MOVEBAT
3460   BATPOS=255-(PADDLE.AND.255)
3470   IF (BATPOS<48) THEN
3480     BATPOS=48
3490   ELSE IF (BATPOS200) THEN
3500     BATPOS=200
3510   END IF
3520   <HPOSP0=BATPOS
3530   <ATRACT=100
3540   RETURN
```

```
3550 ;
3560 ; ====================
3570 ; Remove a Brick at the
3580 ; coordinates of ball.
3590 ; ====================
3600 ;
3610 LOCAL
3620 VARIABLE ?X,?Y,?TEMP
3630 PROCEDURE REMOVE (?X,?Y)
3640   ?X=[?X-48]/4
3650   ?Y=[?Y-36]/4
3660   ?TEMP=([SCREENADR+200+?X]+?Y*40).AND.65534
3670   IF (DPEEK(?TEMP)<>0) THEN
3680     DPOKE (?TEMP,0)
3690     ADD ([6-?Y],9)
3700     SCORE=[SCORE+6-?Y]
3710     SOUND (0,[?Y+1]*30,10,10)
3720     AFTER (3,SOUNDOFF)
3730     IF (HITLAST=0) THEN
3740       YDIR=1
3750     ELSE
3760       YDIR=65535
3770     END IF
3780     Bricks=[Bricks-1]
3790     IF (Bricks=0) THEN
3800       BRICKS (0)
3810     ELSE IF ([?Y+2]<YTIME) THEN
3820       YTIME=[?Y+2]
3830     END IF
3840   END IF
3850 END
3860 ;
3870 ; ========================
3880 ; Move the Ball and return
3890 ; 0 if you missed the ball
3900 ; 2 Normal condition
3910 ; ========================
3920 ;
3930 LOCAL
3940 VARIABLE ?Xtime,?Ytime
3950 ;
3960 FUNCTION MOVEBALL (Dummy)
3970   MOVEBALL=2
3980   IF PEEK(53249) THEN
3990     REMOVE (XPOS,YPOS)
4000     <HITCLR=0
4010   END IF
4020   IF PEEK(53257) THEN
4030     SOUND (0,50,10,10)
4040     AFTER (3,SOUNDOFF)
4050     YDIR=65535
4060     HITLAST=0
4070     <HITCLR=0
4080     IF (XPOS[BATPOS+3]) THEN
4090       XDIR=65535
4100     ELSE IF (XPOS>[BATPOS+3]) THEN
4110       XDIR=1
```

```
4120     ELSE
4130       XDIR=((RANDOM.AND.255) MOD 3)-1
4140     END IF
4150     XTIME=(RANDOM.AND.7)+2
4160   END IF
4170   IF (YPOS=116) THEN
4180     MOVEBALL=0
4190   ELSE IF (YPOS=<23) THEN
4200     YDIR=1
4210     YTIME=1
4220     HITLAST=1
4230     SOUND (0,75,10,10)
4240     AFTER (3,SOUNDOFF)
4250   END IF
4260   IF (XPOS<49) THEN
4270     XDIR=1
4280     SOUND (0,25,10,10)
4290     AFTER (2,SOUNDOFF)
4300   ELSE IF (XPOS205) THEN
4310     XDIR=65535
4320     SOUND (0,25,10,10)
4330     AFTER (2,SOUNDOFF)
4340   END IF
4350   ?Xtime=[?Xtime-1]
4360   IF (?Xtime=65535) THEN
4370     ?Xtime=XTIME
4380     XPOS=[XPOS+XDIR]
4390   END IF
4400   ?Ytime=[?Ytime-1]
4410   IF (?Ytime=65535) THEN
4420     ?Ytime=YTIME
4430     YPOS=[YPOS+YDIR]
4440   END IF
4450   BALL (XPOS,YPOS)
4460 END
4470 ;
4480 ; ======================
4490 ; Begin the Main Routine
4500 ; ======================
4510 ;
4520 LOCAL
4530 VARIABLE ?BALL
4540 ;
4550 BEGIN
4560   REPEAT
4570     TITLE (0)
4580     INIT (0)
4590     BRICKS (0)
4600     EVERY (1,MOVEBAT)
4610     ?BALL=3
4620     SCORE=0
4630     REPEAT
4640       REPEAT
4650       UNTIL (PTRIG.AND.255)=0
4660       SETBALL (0)
4670       REPEAT
4680       UNTIL MOVEBALL(0)=0
```

```
4690       FILL (MISSILES+YPOS,4,0)
4700       ?BALL=?BALL-1
4710       POKE (SCREENADR+39,?BALL+16)
4720     UNTIL ?BALL=0
4730     IF (SCOREHIGH) THEN
4740       HIGH=SCORE
4750       MOVE (SCREENADR+6,HIGHDATA,4)
4760       MOVE (HIGHDATA,SCREENADR+25,4)
4770     END IF
4780     EVERY (0,MOVEBAT)
4790     POKE (20,0)
4800     REPEAT
4810     UNTIL PEEK(20)=200
4820   UNTIL 0
4830 END
```

Appendix E - Background Music Program

```
1000 ;
1010 ; ==============
1020 ; TCL Music Demo
1030 ; ==============
1040 ;
1050 INCLUDE #H1:PEEK.FUN
1060 INCLUDE #H1:DPEEK.FUN
1070 VARIABLE 20=RTCLOCK
1080 VARIABLE 53274=COLBK
1090 VARIABLE 54282=WSYNC
1100 VARIABLE 54283=VCOUNT
1110 VARIABLE TMP
1120 BEGIN
1130   POKE (TIMER,1)
1140   POKE (TIMER+1,1)
1150   POKE (TIMER+2,1)
1160   POKE (TIMER+3,1)
1170   DPOKE (TABLE,DATA1)
1180   DPOKE (TABLE+2,DATA2)
1190   DPOKE (TABLE+4,DATA3)
1200   DPOKE (TABLE+6,DATA4)
1210   POKE (53768,0)
1220   POKE (562,3)
1230   POKE (53775,3)
1240   EVERY (1,MUSIC)
1250   REPEAT
1260     TMP=[VCOUNT+RTCLOCK]
1270     <WSYNC=0
1280     <COLBK=TMP
1290   UNTIL 0
1300 END
1310 ;
1320 ; ======================
1330 ; Music Interrupt handler
1340 ; ======================
1350 ;
1360 POINTER TIMER
```

```
1370 RESERVE 4
1380 VARIABLE CHANNEL,TIME,BASE
1390 POINTER MUSIC
1400   DO CHANNEL=0,3,1
1410     TIME=PEEK([TIMER+CHANNEL])
1420     IF TIME0 THEN
1430       TIME=[TIME-1]
1440       IF TIME=0 THEN
1450         BASE=DPEEK([TABLE+CHANNEL+CHANNEL])
1460         SOUND (CHANNEL,PEEK(BASE),10,10)
1470         TIME=PEEK([BASE+1])
1480         DPOKE ([TABLE+CHANNEL+CHANNEL],[BASE+2])
1490       END IF
1500       POKE ([TIMER+CHANNEL],TIME)
1510     END IF
1520   END DO
1530 RETURN
1540 POINTER TABLE
1550 RESERVE 8
1560 POINTER DATA1
1570 CODE 3506400648065106
1580 CODE 3506400648065106
1590 CODE 5524000C35063C06
1600 CODE 4006480635063C06
1610 CODE 40064806510C350B
1620 CODE 0002350B000C3506
1630 CODE 2F062A0628063506
1640 CODE 2F062A0628062A0C
1650 CODE 3C18000C3C063506
1660 CODE 2F062A063C063506
1670 CODE 2F062A062F0C4018
1680 CODE 000C350640064806
1690 CODE 51063506400648065106
1700 CODE 5524000C35063C06
1710 CODE 4006480635063C06
1720 CODE 40064806510C350B
1730 CODE 0002350B000C3506
1740 CODE 2F062A0628063506
1750 CODE 2F062A0628062F06
1760 CODE 2A06280623062F06
1770 CODE 2A06280623062808
1780 CODE 2A082D082A081F08
1790 CODE 23082824000C
1800 CODE 0000
1810 POINTER DATA2
1820 CODE 0030000C350B0002
1830 CODE 350B000C0030000C
1840 CODE 4018000C00300030
1850 CODE 0030003000030000C
1860 CODE 350B0002350B000C
1870 CODE 0030000C4018000C
1880 CODE 0030003000300030
1890 CODE 0000
1900 POINTER DATA3
1910 CODE A20C6C060006A20C6C060006
1920 CODE 900C6C0C900C6C0C
1930 CODE AD0C6C060006AD0C6C060006
```

```
1940 CODE A20C6C0CA20C6C0C
1950 CODE A20C6C060006A20C6C060006
1960 CODE D90C790CD90C790C
1970 CODE D90C79060006D90C79060006
1980 CODE A20C6C0CA20C6C0C
1990 CODE A20C6C060006A20C6C060006
2000 CODE 900C6C0C900C6C0C
2010 CODE AD0C6C060006AD0C6C060006
2020 CODE A20C6C0CA20C6C0C
2030 CODE A20C6C060006A20C6C060006
2040 CODE F30C90060006F30C90060006
2050 CODE D91079080008D9087908
2060 CODE A20C6C0B00026C0B000C
2070 CODE 0000
2080 POINTER DATA4
2090 CODE 000C8006001280060006
2100 CODE 000C790C000C790C
2110 CODE 000C79060012790C
2120 CODE 000C800C000C800C
2130 CODE 000C8006001280060006
2140 CODE 000CAD0C000CAD0C
2150 CODE 000CAD060012AD060006
2160 CODE 000C800C000C800C
2170 CODE 000C8006001280060006
2180 CODE 000C790C000C790C
2190 CODE 000C79060012790C
2200 CODE 000C800C000C800C
2210 CODE 000C8006001280060006
2220 CODE 000CC1060012C1060006
2230 CODE 0010AD080010AD08
2240 CODE 000C8018000C
2250 CODE 0000
```

Appendix F – Eight Way Scrolling Program

```
1000 VARIABLE 54276=HSCROL
1010 VARIABLE 54277=VSCROL
1020 VARIABLE 560=DLIST
1030 VARIABLE 632=STICK0
1040 VARIABLE 548=VVBLKD
1050 VARIABLE 106=RAMTOP
1060 VARIABLE SCREENMEM
1070 VARIABLE I,K,L
1080 POINTER LINEADR
1090 RESERVE 136
1100 BEGIN
1110   <RAMTOP=(RAMTOP.AND.255)-44
1120   SCREENMEM=(RAMTOP.AND.255)*256
1130   L=SCREENMEM
1140   DO I=0,67,1
1150     K=GETLINE(L)
1160     FILL (K,160,0)
1170     DPOKE (LINEADR+I*2,K)
1180     L=[K+160]
1190   END DO
```

```
1200   SCROLL (0,0)
1210   DLIST=MYDLIST
1220   DO I=0,67,1
1230     XPLOT (I,I,I)
1240     XPLOT (159-I,67-I,I)
1250   END DO
1260   VVBLKD=VBI
1265 10 GOTO 10
1270 END
1280 VARIABLE XC,YC
1290 VARIABLE XF,YF
1300 VARIABLE CSFLAG
1310 POINTER VBI
1320   CSFLAG=0
1330   IF (STICK0.AND.1)=0 THEN
1340     YF=[YF-1]
1350   ELSE IF (STICK0.AND.2)=0 THEN
1360     YF=[YF+1]
1370   END IF
1380   IF (STICK0.AND.4)=0 THEN
1390     XF=[XF+1]
1400   ELSE IF (STICK0.AND.8)=0 THEN
1410     XF=[XF-1]
1420   END IF
1430   IF (XF=65535) THEN
1440     IF XC<116 THEN
1450       XC=[XC+4]
1460       XF=15
1470       <CSFLAG=1
1480     ELSE
1490       XF=0
1500     END IF
1510   ELSE IF (XF=16) THEN
1520     IF XC0 THEN
1530       XC=[XC-4]
1540       XF=0
1550       <CSFLAG=1
1560     ELSE
1570       XF=15
1580     END IF
1590   END IF
1600   IF (YF=65535) THEN
1610     IF YC0 THEN
1620       YC=[YC-1]
1630       YF=7
1640       <CSFLAG=1
1650     ELSE
1660       YF=0
1670     END IF
1680   ELSE IF (YF=8) THEN
1690     IF YC<50 THEN
1700       YC=[YC+1]
1710       YF=0
1720       <CSFLAG=1
1730     ELSE
1740       YF=7
1750     END IF
```

```
1760   END IF
1770   IF (CSFLAG) THEN
1780     SCROLL (XC,YC)
1790   END IF
1800   <HSCROL=XF
1810   <VSCROL=YF
1820   CODE 4C62E4
1830 ;
1840 ; ==================
1850 ; Adjust Display List
1860 ; to Compensate for
1870 ; the Course Scroll.
1880 ; ==================
1890 ;
1900 LOCAL
1910 VARIABLE ?XPOS,?YPOS
1920 VARIABLE ?ADR,?I,?J
1930 VARIABLE ?ARRAY
1940 PROCEDURE SCROLL (?XPOS,?YPOS)
1950   ?ARRAY=[?YPOS+?YPOS+LINEADR]
1960   ?I=[MYDLIST+4]
1970   DO ?J=1,17,1
1980     ?ADR=DPEEK(?ARRAY)
1990     @?I=[?ADR+?XPOS]
2000     ?I=[?I+3]
2010     ?ARRAY=[?ARRAY+2]
2020   END DO
2030   @?I=MYDLIST
2040 END
2050 ;
2060 ; ==================
2070 ; Plot a Character
2080 ; at X,Y within the
2090 ; Scrollable Screen.
2100 ; ==================
2110 ;
2120 LOCAL
2130 VARIABLE ?XPOS,?YPOS
2140 VARIABLE ?CHAR,?ARRAY
2150 VARIABLE ?ADR,?I
2160 PROCEDURE XPLOT (?XPOS,?YPOS,?CHAR)
2170   ?ARRAY=[?YPOS+?YPOS+LINEADR]
2180   ?ADR=DPEEK(?ARRAY)
2190   ?ADR=[?ADR+?XPOS]
2200   <@?ADR=?CHAR
2210 END
2220 ;
2230 ; ==================
2240 ; Allocate a 160 Byte
2250 ; Block of Memory
2260 ; that does not cross
2270 ; Antics 4KB Boundary
2280 ; ==================
2290 ;
2300 LOCAL
2310 VARIABLE ?ADR1,?ADR2
2320 FUNCTION GETLINE(?ADR1)
```

```
2330   ?ADR2=?ADR1+159
2340   IF (?ADR1 .AND. 4096)<>(?ADR2 .AND. 4096) THEN
2350   ?ADR1=(?ADR2 .AND. 63488)
2360   END IF
2370   GETLINE=?ADR1
2380 END
2390 ;
2400 ; ======================
2410 ; Implement PEEK Function
2420 ; syntax: var=PEEK(adr)
2430 ; ======================
2440 ;
2450 LOCAL
2460 FUNCTION PEEK (?address)
2470   CODE AD
2480   VARIABLE ?address
2490   PHA
2500   CODE A900
2510   PHA
2520   PULL (PEEK)
2530 END
2540 ;
2550 ; ========================
2560 ; Implement DPEEK Function
2570 ; syntax: var=DPEEK(adr)
2580 ; ========================
2590 ;
2600 LOCAL
2610 FUNCTION DPEEK (?address1)
2620   ?address2=[?address1+1]
2630   CODE AD
2640   VARIABLE ?address1
2650   PHA
2660   CODE AD
2670   VARIABLE ?address2
2680   PHA
2690   PULL (DPEEK)
2700 END
2710 ;
2720 ; ==================
2730 ; Display List for a
2740 ; Scrollable Display
2750 ; which uses Mode 4.
2760 ; ==================
2770 ;
2780 POINTER MYDLIST
2790 CODE 707070
2800 CODE 740000
2810 CODE 740000
2820 CODE 740000
2830 CODE 740000
2840 CODE 740000
2850 CODE 740000
2860 CODE 740000
2870 CODE 740000
2880 CODE 740000
2890 CODE 740000
```

```
2900 CODE 740000
2910 CODE 740000
2920 CODE 740000
2930 CODE 740000
2940 CODE 740000
2950 CODE 740000
2960 CODE 540000
2970 CODE 410000
```