MultiBASIC v.0.32

An Atari BASIC compatible interpreter for 65C816 Atari machines

Document revised 24.V.2006.

Table of Contents

MultiBASIC is a BASIC interpreter for Atari 8-bit machines. The dialect used is an extension of Turbo BASIC XL. Since there is no Turbo BASIC XL source code available, it has been reverse-engineered from the scratch. This is why the two early alpha releases of MultiBASIC were rather buggy.

Hardware requirements:

* an Atari 400, 800, XL or XE computer.
* 65C816 CPU.
* a ROM based OS that allows to execute programs in native 65C816 mode.
* 40k RAM.

Future versions may require an amount of directly addressable RAM past the first 64k. Also a floppy disk drive (or preferably hard disk) is recommended.

This document describes differences between MultiBASIC and the Turbo BASIC XL 1.5. It is assumed, that all Turbo BASIC XL keywords and operators are either already implemented in identical manner as in the original, or will be implemented in subsequent versions of MultiBASIC. The current revision is already highly compatible with the Turbo BASIC XL, only few keywords are missing (CIRCLE, PAINT, RENUM, TEXT, TRACE, INSTR and UINSTR). Some structural features of the Turbo BASIC XL such as line labels and procedures are already implemented, and 256 variables can be declared too. Few annoying bugs present in Atari BASIC and Turbo BASIC XL are avoided (MultiBASIC has own ones for sure, though, but at the other hand the current version 0.32 seems to be fairly sane and stable).

Turbo BASIC XL compatibility is provided at the token level, i.e. a tokenized Turbo BASIC XL program loaded to MultiBASIC interpreter should work without problems. Also a MultiBASIC program, if keywords used are confined to these known to the Turbo BASIC XL interpreter, should work when loaded to Turbo BASIC XL.

There are two things where Turbo BASIC XL is still superior than MultiBASIC: the first one is the interpretation speed. Although MultiBASIC is fairly fast, and even noticeably (up to factor 1,2) outperforms Turbo BASIC in some operations, some other operations, especially division and square root calculation, are slow (division reaches 77% of the speed it has under TBXL, SQR is only 68%). This will hopefully get speeded up in future versions of the interpreter.

The other problem is the amount of memory available for your BASIC program: the best you can currently get is about 21k (this depends on the MEMLO value). It could be solved by moving parts of the interpreter to the RAM shadowing the OS ROM, or use 130XE memory banking, but it probably won't be done as MultiBASIC is generally being written for machines, which apart of the 65C816 also feature some directly addressable RAM past the first 64k segment. The main goal is to use that memory to store the interpreter itself and BASIC program, so that one could have 64k for BASIC code, another 64k for dimensions and strings, and yet use the rest of available memory to execute the interpreter at full speed. This is yet to be done (I don't have such a machine myself yet, and it is impossible to properly debug the code without), but already a lot of internal changes have been done in the interpreter code in orgder to prepare it to that. For now, the current 0.32 version does not use the additional memory, but already allows to manipulate that freely and provides an interface to the OS' allocator.

The intention is also to implement as much of BASIC XE as possible, but tokens can't be the same (most of them are already allocated for Turbo BASIC XL keywords). A 100% keyword compatibility doesn't seem possible either as there are conflicts at the syntax level and/or in the keyword meaning between Turbo BASIC XL and BASIC XE, that makes a full compatibility with both impossible to achieve. I thus decided to favour the more popular Turbo BASIC XL.

There are few general differences in the actual operation worth to be mentioned:

1) many MultiBASIC operations that imply opening an I/O channel (e.g. DIR) use the first available IOCB instead of the channel #7;

2) addresses are 24-bit;

3) Atari BASIC is known to scan the entire program from the beginning searching for the desired line each time a GOTO or GOSUB is done. E.g. a GOTO 1001 located in line 1000 causes the interpreter to walk through all the program lines beginning at the very first one until it meets the line 1001; and the same on GOSUB, RESTORE et cetera.

MultiBASIC does that only for backward jumps, e.g. a 1000 GOTO 999 makes the interpreter to find the line 999 the hard way. If the destination line number of a GOTO or GOSUB is equal to or bigger than the current line number, the search starts at the current line. This speeds up greatly all loops that fit completely in one line (note that current version of MultiBASIC does not feature loop precompilation, this is yet to be done). This technique reduces the number of necessary searches.

4) To avoid line searching completely use labels (#) in GO#, GOTO, GOSUB, RESTORE and TRAP statements instead of line numbers. Note that Turbo BASIC XL did not permit using labels in GOTO and GOSUB, MultiBASIC does.

5) in statements which assign values to numeric variables (GET, INPUT, LOCATE, NOTE, READ, STATUS) both Atari BASIC and Turbo BASIC XL do not permit elements of numeric tables; even though NOTE and STATUS – apparently by an accident – allow such syntax: STATUS #1,L(0), but the results are meaningless. This limitation is present also in BASIC XE and also in current version of MultiBASIC (keywords like %GET, FLEN and TELL apply here). I consider this behaviour illogical and I'll tend to eliminate this.

6) Internal variables of the interpreter traditionally present on the zero page between $000080 and $0000D3 are now private and not accessible any more. The only location used by interpreter is the floating point register FR0 ($D4-$D9), despite that the upper half of the zero page is empty, and may be used freely in machine code subroutines invoked from within a BASIC program.

7) The interpreter itself runs in the native 65C816 mode most of the time. The CPU however will be switched to the 6502 emulation mode, whenever system calls are performed and also when calling a user defined function via USR(). Normally this behaviour shouldn't be visible to the programmer, there's however one exception: interrupts. Writing an interrupt handler for your BASIC program you should bear in mind that interrupts are asynchronous, and so an interrupt can occur when the CPU is executing the 65C816 native code of the interpreter, as well as when it is executing the 6502 emulation code of the OS or a function called through USR(). Thus any interrupt handler should be written so that it could execute in either mode, or separate handlers for both modes should be provided. This is especially critical for the Display List Interrupt. Please refer to the OS documentation to read a more detailed discussion of the problem.

I. Dealing with the BASIC program

FILENAME [”filename”]

Defines a default filename for LOAD and SAVE – **LOAD ”D:FOO.BAS”** is the same as **FILENAME ”D:FOO.BAS”:LOAD**. If executed without the argument, it empties (deletes) the default filename.

LOAD [”filename”]

Loads the specified file to the memory. The file must contain a tokenized BASIC program, or error 21 is generated otherwise. The argument is optional. If not given, LOAD will attempt to use default filename. The default filename is:

1) the one that was explicitly given last time the LOAD ”filename” (or RUN ”filename”) was executed;

2) the one that was passed to the MultiBASIC interpreter in the command line arguments as a name of a program to load;

3) the one defined by NEW or FILENAME command (see there).

If no argument is given and no default filename is defined, an error number 31 occurs. **LOAD ”D:FOO.BAS”** is the same as **FILENAME ”D:FOO.BAS”:LOAD**

NEW [”filename”]

Clears the BASIC memory deleting any program contained there. The optional argument defines new program name for LOAD and SAVE. NEW also clears the command line.

RUN [”filename”[,line]]

RUN [line]

Executes the tokenized BASIC program. If the filename is given, the specified file is first loaded to the memory. The program must be in tokenized form (i.e. created by SAVE), or error 21 occurs. The additional argument ‘line’ specifies the line number where the program execution shall begin. If there is no such line, the immediately following one is assumed. RUN ”filename” command clears the command line and defines the new default filename for subsequent SAVE and LOAD commands.

SAVE [”filename”]

Creates a file and saves the BASIC program to it, in tokenized form. The argument is optional. If not given, default filename is assumed. The default filename is:

1) the one that was explicitly given last time the LOAD ”filename”, or RUN ”filename” was executed;

2) the one that was passed to the MultiBASIC interpreter in the command line arguments as a name of a program to load;

3) the one defined by NEW or FILENAME command (see there).

If no argument is given and no default filename is defined, an error number 31 occurs. If default filename is defined, and SAVE with an explicit filename is executed, the default filename does not change (it is assumed that the user working on a program would this way make a backup copy before doing major changes). If default filename is undefined, and SAVE with explicit filename is executed, this filename becomes the default. **SAVE ”D:FOO.BAS”** is the same as **FILENAME ”D:FOO.BAS”:SAVE**

II. Command line arguments

ARGC

Holds the number of argument passed as command line. 0 means that no command line arguments were given (the MultiBASIC executable does not count here). If ARGC is not zero, the first argument is always a path to the .BAS program that is currently loaded to the memory, and further arguments, if there are any, are its (the program’s) parameters. The ARGC value is changed by LOAD, RUN and NEW. NEW deletes the command line and zeroes the ARGC. LOAD and RUN – the one which loads a program – delete the command line and establish a new value of ARGC, which will be at least 1 after these commands are completed. The first argument will be replaced with the name of the program that has just been loaded.

ARGV$(n)

Returns a text string that is the n-th command line argument passed to the BASIC program. The first argument is always a pathname of the program itself. The ‘n’ parameter range is 0-65535. If the number is equal to or bigger than the ARGC value, the returned string consists of the EOL character only. To print all the command line arguments:

```
10 X=0
20 WHILE X<ARGC
30 ? ARGV$(X):X=X+1
40 ENDWHILE
```

The NEW command deletes the command line, and loading a program (using LOAD or RUN command) changes its contents to reflect new program name and its arguments, if any.

III. Editing commands

AUTO [start][,increment]

NUM [start][,increment]

Enable the line-autonumber mode. After a program line was typed, the editor will automatically put a number for a new line onto the screen, and then wait for new contents. You exit this pressing the Return key directly after the line number appears on screen, or hitting the Break key. The autonumber mode is exited automatically, if there exist a line with the number that is about to be displayed, or the number exceeds the allowed range (i.e. it is greater than 32768).

The AUTO/NUM command accepts up to two optional arguments: the first is the starting line number, and the other is the increment. **AUTO 0,0** or **NUM 0,0** will reset the counter to defaults. The default values for these two can be read using SYS(22) and SYS(23), respectively, and changed by appropriate SET commands.

DEL line_1[,line_2]

Delete a part of BASIC program that resides in the memory, from line number 'line_1' to line number 'line_2', inclusive. Neither the first not the second line have to exist, these numbers are only limits, and beyond that no line will be deleted.

Alternatively you can specify only one number – this will be interpreted as the number of the line to be deleted, just like in BASIC XE.

DIR ["filename"]

Outputs the disk directory to the screen. With SpartaDOS the "long" directory listing is generated, and standard one otherwise. This can be controlled with SET 20,x, where x = 0 disables the long listing under SpartaDOS, and x = 128 enables it back. Don't change the default value if not using SpartaDOS: the SET 20 sets a value that will be passed to the OS as 'icax2' in the call opening the directory for reading.

DUMP ["filename"]

Displays a list of variables used in the program or, if filename is given, writes it to the specified file. The variables are displayed as name-type-value in the following manner:

"A =150" - numeric variable of value 150

"A(0,0" - undeclared numeric table

"A(6,1" - numeric table DIM A(5)

"A(6,6" - numeric table DIM A(5,5)

"A$ 0,0" - undeclared string

"A$ 3,8" - string DIM A$(8), LEN(A$) = 3

"A # 100" - label in line 100

”A PROC 100” - procedure in line 100

”A ?” - unknown label or procedure

IV. Memory reads and writes

DPEEK(address)

Returns a 16-bit word value found at address ‘address’. The ‘address’ value may specify addresses in the entire 16 MB address space. Also, in MultiBASIC DPEEK() does an atomic 16-bit read, so that if you happen to fetch with it the value of a 16-bit, interrupt driven counter, you may be sure that no interrupt has occurred between fetching the low and the high byte.

DPOKE adress,value

Puts a 16-bit word ‘value’ to address ‘address’. The ‘address’ range is 0-16777215, or $000000-$FFFFFF. As a counterpart to DPEEK(), it does a 16-bit atomic write – you can safely change VBL or DLI vectors with it.

MEMSET begin,length,pattern

Fills the memory block starting at address ‘begin’ and being of ‘length’ bytes long with the byte value given as ‘pattern’. The first two arguments are ranged within 0-16777215. A sum of ‘begin’+’length’ may not be bigger than 16777215, or a value error will occur. Unlike POKE and DPOKE, MEMSET does not do destination address sanity check – so vital system areas can be overwritten if you’re not careful enough.

MOVE src,dest,how_much

-MOVE src,dest,how_much

MOVE src,dest,-how_much

Copies a block of memory starting at ‘src’ and being ‘how_much’ bytes long to the address ‘dest’. The arguments may be ranged from 0 to 16777215. Neither source nor destination block may span the 16 MB boundary (i.e. neither a sum of ‘src’+’how_much’ nor ‘dest’+’how_much’ may be greater than 16777216). If this is the case, error 3 occurs and nothing is copied. Unlike POKE and DPOKE, MOVE/-MOVE does not do destination address sanity check – so vital system areas can be overwritten if you’re not careful enough.

The difference between MOVE and -MOVE is that the former copies bytes in ascending order (starting at ‘src’ and proceeding towards higher addresses), and the latter in descending order (starting at ‘src+how_much-1’ and proceeding towards lower addresses). The interpreter does not decide itself which method is to be used to avoid destroying the data, if ‘src’ and ‘dst’ areas overlap – you have to

determine that in your program. Instead of -MOVE, which follows the Turbo BASIC XL style, you can use MOVE with third argument being a negative value (BASIC XE style).

It may be useful to know, that even if data can be copied freely across the entire 16 MB address space, the copying speed may depend of the actual location of both src and dest areas. The fastest copying occurs when neither area spans any of the 64k boundaries, i.e. both are completely contained in a 64k segment (either in the same one or in two different ones, this doesn't matter). With CPU clocked at stock rate 1,773 MHz you can expect about 160 kB/s. In this case there is no speed difference between MOVE and -MOVE.

However, when the src block or dest block or both span a 64k boundary, the copying is much slower and rates about 90 kB/s at 1,773 MHz for MOVE, and about 108 kB/s for -MOVE.

The MOVE instruction is sometimes used in Turbo BASIC XL to fill the memory quickly with the given pattern. It is usually done with the following instruction sequence: **POKE adr,pattern:MOVE adr,adr+1,size-1**. This should still work in Turbo BASIC XL programs loaded into MultiBASIC, in new programs you should however use MEMSET in its stead, as MOVE is not guaranteed to work so if used on addresses past the first 64k of memory.

PEEK(address)

Returns a byte value found at 'address'. The address range is 0-16777215.

POKE address,value

Stores a byte value 'value' at address 'address'. The address range is 0-16777215.

V. Advanced memory management

MALLOC(size)

This is a numeric function that tries to allocate 'size' bytes in the memory past the first 64k, and returns 24-bit address of the allocated block, if successful. It is guaranteed that the allocated block wouldn't be smaller than 'size' bytes. The underlying system function called is kmalloc(). If the memory can't be allocated, an error -74 is reported. Possible reasons are: out of memory, no more entries in the allocation table (there's room for 42 entries in current implementation).

The 'size' parameter must be non-zero; if it is zero, a value error is reported. Values greater than a zero are understood as an amount of bytes to allocate. A value of -1 does not allocate anything, but returns the amount of available memory instead.

An allocated memory block may explicitly be freed with MFREE after use. All allocated blocks belong to the BASIC program and are freed whenever it is removed from the memory (NEW) or replaced by another program (LOAD). Also RUN frees all the memory blocks, so that after a program

was interrupted you can run it immediately from the beginning without a need to reload or explicit release.

All the allocated memory is also freed when the interpreter is told to exit (with CP or DOS command), and also when the user hits the RESET key causing the OS to do a warm start.

MFREE address

Releases the block of memory that had previously been allocated by MALLOC(). The argument is the address as returned by MALLOC().

MATTRIB atr

Sets the memory attributes for the kmalloc() system call invoked by MALLOC(). The 'atr' is a 16-bit attribute word that will be passed by the interpreter to the kmalloc() whenever MALLOC() is executed. The default value is zero. It is also zeroed whenever a BASIC program terminates and the interpreter goes to the 'direct' mode (i.e. waiting for user commands). For the possible values of the 'atr' argument and their meaning refer to the OS documentation.

VI. Machine code support

ADR("string")

This is identical as in Turbo BASIC XL, except that it may return 24-bit addresses.

SYSCALL value

Call a system routine specified by 'value'. A value from 0 to 2 specify a system routine, one of the following:

0 – RESETCD

1 – RESETWM

2 – SIOINT

The call is done in emulation mode. Values bigger than 2 and lesser than $0400 are reserved and will cause value errors.

A value equal to or bigger than $0400 is interpreted as a long address to be called. A long direct JSR (JSL) is done to that address and the routine is expected to end with an RTL. Contrary to the behaviour described above, this call is done in native mode. Since no direct ROM calls can be done with that, values equal to or greater than $00E000 and lesser than $010000 will cause value errors.

USR(address[,arg1[,arg2 ...]])

Identical as in Turbo BASIC XL except that SET 8,0 prevents the number of arguments to be

stacked. The function being called may then return with normal RTS (not PLA/RTS); this means that you can call a ROM function and yet hope for a successful return. This behaviour is compatible with BASIC XE. You can restore the default behaviour using SET 8,1.

The call is always done in 6502 emulation mode and the address specified is always expected to be within the range of $000000-$00FFFF (segment 0). When such a construct is used: X=USR(ADR(A$)), and the program is executing outside the bank 0, the string A$ is copied to bank 0 before being jumped to. Otherwise, if an address is given above $FFFF, a value error is reported. See also LUSR.

WAIT [address,and_mask,compare]

Suspends the program execution until the byte value found at the 'address', when AND-ed with 'and_mask', equals the third parameter. The 'address' can be ranged from 0 to 16777215, the rest of arguments are byte values. The desired value is waited for in a busy loop, you can abort it with the Break key.

When WAIT is used without arguments, it suspends the program execution (halting the CPU) until an interrupt occurs. Useful in busy-wait loops.

VII. Audio control

AUDCTL #stereo-channel[,mask,value]

This keyword allows an access to the global audio control in Pokey. On computers with stereo-Pokey the first argument selects one of the stereo channels (left or right). The 'mask' value is AND-ed with the current value of the Pokey's AUDCTL register (or better to say – with its shadow, as the AUDCTL register is write-only), and the result of this operation is the OR-ed with the 'value', and stored into the appropriate register. Availability of the second Pokey can be proven using SYS(28).

The arguments 'mask' and 'value' can be omitted (AUDCTL #0 is perfectly valid), both are assumed zeros in that case.

DSOUND [chn,freq,pitch,volume]

DSOUND #stereo-channel[,chn,freq,pitch,volume]

This is similar to SOUND, except that a pair of audio channels is used to generate sound. The 'chn' value may be 0 or 1. The 'freq' range is 0-65535. For 'pitch' and 'volume' see SOUND. The four arguments may be preceded with an additional one specifying one of the two Pokeys in computers equipped with stereo extension (if not specified, 0 is assumed). DSOUND #0 or DSOUND #1 (without the following four arguments) disable the sound on the specified stereo-channel only. Availability of the second Pokey can be proven using SYS(28).

SOUND [chn,freq,pitch,volume]

SOUND #stereo-channel[,chn,freq,pitch,volume]

Identical to Atari BASIC's SOUND keyword, except that the usual four arguments may be preceded with an additional one specifying one of the two Pokeys on computers equipped with stereo extension (if not specified, 0 is assumed). SOUND #0 or SOUND #1 (without the following four arguments) disable the sound on the specified stereo-channel only. Availability of the second Pokey can be proven using SYS(28).

VIII. Numeric conversions and arithmetics

%

This is the same as EXOR. This is a BASIC XE compatibility stuff.

%0, %1, %2, %3

Like in Turbo BASIC XL, these are constant values. They're faster than actual variables or regular numeric constants, and also take less memory (only 1 byte).

DEC("hex string")

Converts "hex string" into numeric value. The hex number being converted may have up to 6 hex digits (24-bit value), if it is longer, last six digits are taken into account. Unlike in Turbo BASIC XL, the hex string may be lower case, upper case, inverse video or preceded with spaces.

HEX$(n)

Returns a hex string corresponding to the decimal value passed as a parameter. The range of the numeric value is 0-16777215.

PI

Holds the value of 3,14159265.

RANDOMIZE [value]

On Atari this is more like VOID: it calculates the 'value' and does nothing with the result. Atari has the random number generator implemented in hardware and so RANDOMIZE command lacks sense. This keyword has been introduced solely for better compatibility with Microsoft BASIC programs.

STR$(n)

Returns a string that represents the passed numeric value as an ASCII string of decimal digits.

VAL(”text string”)

Returns a value of the ‘text string’ interpreted as an ASCII string of decimal digits. There is an additional behaviour compatible with BASIC XE: namely, the VAL function can calculate a value of a hex string, much like the DEC() function. Unlike that, the hex string must be preceded with a $ sign to be calculated correctly. You may switch off this with SET 13,0 (default is 1).

VOID numeric_expression

Calculates the ‘numeric_expression’ and ignores the result of the calculation. This is useful when calling numeric functions that perform some actions and return a value, when you’re only interested in the action and not in the returned result. E.g. VOID USR(address) spares you assigning a random value to a variable, if the called function returns nothing.

XOR

This is the same as EXOR in Turbo BASIC XL. Just a more standard name.

IX. Input/output

BSAVE begin,end,”filename”

Saves a memory block specified by ‘begin’ and ‘end’ addresses. The block is saved as a standard DOS binary file (header $FFFF). The addresses must fit within the range 0-65535.

FILE

This is a pseudovariable that holds the number (ranged from 0 to 7) of the first available, i.e. closed, IOCB channel. Usage: **FD=FILE:OPEN #FD,4,0,”D:FILENAME”**. If there are no more free channels, reading FILE causes error -95 (”too many channels open”). Remember that reading this doesn’t ”allocate” the IOCB for you, so a statement like: **F1=FILE:F2=FILE:OPEN #F1,4,0,”D:FOO”:OPEN #F2,4,0,”D:BAR”** – won’t work.

FLEN #iocb,variable

The ‘variable’ is assigned with the length of the file opened on the ‘iocb’ IOCB channel. The underlying OS function called is XIO 39 – the system must have it implemented for the FLEN to work.

GET [#iocb,]var[,var[,var...]]

Gets a number of bytes from the specified channel and assigns their values to specified numeric variables. If the channel number is not specified, the data is read from the keyboard (”K:” device).

PUT [#iocb,]val[,val[,val...]]

The specified values are interpreted as byte values and sent out to the specified channel as single data bytes. If the channel number is not specified, the data is sent to the screen editor (”E:” device).

SEEK #iocb,value

Do a seek in the file opened on ‘iocb’ IOCB to the position specified by ‘value’. The underlying OS function called is XIO 37 – the system must have it implemented in identical manner as in SpartaDOS X for the SEEK to work correctly. It’s counterpart is TELL.

STATUS #iocb,var

The variable ‘var’ is assigned a value equal to the status code of last I/O operation done on the specified IOCB channel. The only difference is that the error number is negative (e.g. -120 instead of traditional 136). This is controlled by SET 19,1 – which changes error numbers into positive values (default is 0).

TELL #iocb,variable

The ‘variable’ is assigned a value which specifies the current read/write position in the file opened as ‘iocb’ IOCB. The underlying OS function called is XIO 38 – the system must have it implemented in identical manner as in SpartaDOS X for the TELL to work correctly. TELL’s counterpart is SEEK.

X. Disk file management

DELETE ”filename”

ERASE ”filename”

Deletes the specified file from the disk.

LOCK ”filename”

PROTECT ”filename”

Sets the write-protection flag on the specified file.

UNLOCK ”filename”

UNPROTECT ”filename”

Clears the write-protection flag on the specified file.

XI. Error handling

ERL

This is a pseudovariable that holds the number of the line, where an error occurred.

ERR

This is a pseudovariable that holds the last error number similarly as in Turbo BASIC XL. The only difference is that the error number is negative (e.g. -120 instead of traditional 136). This is controlled by SET 19,1 – which changes error numbers into positive values (default is 0). PEEK(195) works no more in MultiBASIC.

REPORT [code]

Prints the message associated with the given error code or, if no code was given, the last occurred error code. This keyword is inspired by Acorn BBC BASIC.

TRAP [line]

TRAP #label

This sets a 'trap' for an error, which may occur in the program. When it occurs, the control is given over via a GOTO-like jump to the specified place. If the label used is not defined, an error number 30 is generated immediately and the program execution aborts. MultiBASIC adds own syntax modification here: the argument is optional. Using the TRAP instruction without an argument cancels the trap set previously (this works like TRAP 32768 in Atari BASIC).

XII. Labels

var

Declares a symbolic name (label) for the program line where it is put into. It can only be put at the beginning of the line, as its first instruction. Lines marked so can be later referenced through labels defined so using instructions GO#, GOSUB, GOTO, RESTORE and TRAP. Such references (calls) are done much faster than when using the regular line number.

To accomplish this effect the interpreter does some sort of precompilation, so that labels are assigned physical addresses and are referenced by these. Commands such as RUN and CONT explicitly run the precompilation stage before proceeding with BASIC program execution. Getting back to the direct mode (after the program has been finished or stopped) cancels the precompilation. So does the DEL instruction. If program execution is resumed using anything else than RUN or

CONT, the precompilation gets invoked by the first reference to a labelled program line.

GO# label

Makes a GOTO-like jump to the program line, which is labelled with the variable name ‘label’ (see # instruction). Such a jump is made much faster than a regular GOTO with line number.

GOSUB line

GOSUB #label

This works exactly as in Atari BASIC except that the destination place of the jump can be defined with a label (see #).

GOTO line

GOTO #label

This works exactly as in Atari BASIC except that the destination place of the jump can be defined with a label (see #).

RESTORE [line]

RESTORE #label

This works as in Turbo BASIC XL. Using the label makes this instruction execute faster.

XIII. Comments and separators

--

Identical as in Turbo BASIC XL. SET 18,x controls whether the lines beginning with this are subject to normal indentation (1) or not indented (0).

REM

Identical as in Turbo BASIC XL. SET 17,x controls whether the lines beginning with the REM are subject to normal indentation (1) or remain not indented (0).

XIV. Structural instructions

MultiBASIC provides the same sort of structural instructions as Turbo BASIC XL does. These are:

DO: ... :LOOP

Keep executing instructions found between these two keywords in a loop. The loop is an unconditional one, it never ends.

EXIT

EXIT line_number

EXIT #label

Exit a loop cleaning up the runtime stack. EXIT without an argument jumps to the first instruction past the loop, or to the specified location in the program otherwise.

IF cond: ... :ENDIF

If the condition 'cond' is true (i.e. non-zero), execute the code between the IF and ENDIF, or ignore it otherwise.

IF cond: ... :ELSE: ... :ENDIF

If the condition 'cond' is true, execute the code between IF and ELSE, and between the ELSE and ENDIF otherwise.

PROC name: ... :ENDPROC

Define a procedure identified by 'name'. The 'name' undergoes the same restrictions as a variable name. The PROC keyword must be the first one in a program line, and it can never be executed by other means than with the EXEC keyword. The **EXEC name** calls a procedure named 'name', execution of the instructions appearing after the EXEC keyword is resumed when the interpreter meets the ENDPROC instruction. In other words, the sequence **EXEC name / ENDPROC** works more like **GOSUB line_number / RETURN**, it is however faster, because the address of the procedure is stored in the variable table and used to find the procedure in the memory. A conditional form of this call, i.e. ON ... EXEC is also possible.

REPEAT: ... :UNTIL cond

Keep looping and executing the code between REPEAT and UNTIL, as long as the condition 'cond' is false.

WHILE cond: ... :WEND

WHILE cond: ... :ENDWHILE

Keep looping and executing the code inside the loop as long as the condition 'cond' is true. ENDWHILE is an alias for WEND and the operation of both is identical, only the latter however is token-compatible with Turbo BASIC XL.

XV. Interpreter control

SET num,value

Controls internal variables of the MultiBASIC interpreter. The ‘num’ values from range 0-15 have meaning compatible with BASIC XE (not that everything of that is currently implemented).

0 – the Break key behaviour. The default is 0, in this mode the program gets aborted and the ”Stopped at line” message is printed on the screen; 1 causes an error 1 to be generated instead, which can be caught with TRAP. This is BASIC XE compatible behaviour. 2 is the mode that is not present in BASIC XE, hitting Break in this mode causes error 128 to be generated. This is what Turbo BASIC XL does after *B+ is executed. In fact, *B+ and SET 0,2 do exactly the same thing. *B- is the same as SET 0,0.

1 – the number of TAB positions a comma skips in PRINT instruction. Minimum is 3.

2 – ASCII code for a character displayed as INPUT prompt. Default is $3F (‘?’)

3 – unimplemented

4 – unimplemented

5 – accept (1 – default) or don’t accept (0) keywords typed in lower case.

6 – generate (0 – default) or don’t generate (1) error descriptions in error messages.

7 – unimplemented

8 – 1 (default) causes the USR and LUSR functions to put the number of arguments on top of the stack, 0 disables this.

9 – 1 causes error 32 to be generated after ENTER; 0 is the default.

10 – unimplemented

11 – unimplemented

12 – enables (1 – default) and disables (0) listing indentation.

13 – enables (1 – default) and disables (0) the VAL function to calculate values of hex strings.

14 – unimplemented

15 – unimplemented

--- The ‘num’ values above 15 have the following meanings:

16 – number of spaces added in an indentation step; the default is 2.

17 – lines beginning with REM are not indented (0 – default), or they are (1)

18 – lines beginning with – are not indented (0 – default), or they are (1)

19 – error codes are negative (0 – default), or positive (1)

20 – ICAX2 for the ”open dir” call in DIR; default value is 128 under SpartaDOS or 0 otherwise.

21 – listing style: BASIC XE (1) or Turbo BASIC XL (0 – default). BASIC XE style means that keywords are listed in lower case except the first character (Graphics 8:Color 1:Plot 0,0:Drawto

319,191), while Turbo BASIC XL style lists everything in upper case (GRAPHICS 8:COLOR 1:PLOT 0,0:DRAWTO 319,191).

22 – starting value for the AUTO command; default is 10

23 – increment value for the AUTO command; default is 10

24-28 – these values are read-only, can't be changed

All the values can be examined with SYS function (see there).

SYS(num)

Returns a value of a MultiBASIC internal variable. The 'num' values from range 0-15 have meaning compatible with BASIC XE (not that everything of that is currently implemented):

0 – the Break key behaviour. The default is 0, in this mode the program gets aborted and the "Stopped at line" message is printed on the screen; 1 causes an error 1 to be generated instead, which can be caught with TRAP. This is BASIC XE compatible behaviour. 2 is the mode that is not present in BASIC XE, hitting Break in this mode causes error 128 to be generated. This is what Turbo BASIC XL does after *B+ is executed. In fact, *B+ and SET 0,2 do exactly the same thing. *B- is the same as SET 0,0.

1 – the number of TAB positions a comma skips in PRINT instruction. Minimum is 3.

2 – ASCII code for a character displayed as INPUT prompt. Default is $3F ('?')

3 – unimplemented

4 – unimplemented

5 – accept (1 – default) or do not accept (0) keywords typed in lower case.

6 – generate (0 – default) or do not generate (1) error descriptions in error messages.

7 – unimplemented

8 – 1 (default) causes the USR and LUSR functions to put the number of arguments on top of the stack, 0 disables this.

9 – 1 causes error 32 to be generated after ENTER; 0 is the default.

10 – unimplemented

11 – unimplemented

12 – enables (1 – default) and disables (0) listing indentation.

13 – enables (1 – default) and disables (0) the VAL function to calculate values of hex strings.

14 – unimplemented

15 – unimplemented

--- The 'num' values above 15 have the following meanings:

16 – number of spaces added in an indentation step; the default is 2.

17 – lines beginning with REM are not indented (0 – default), or they are (1)

18 – lines beginning with – are not indented (0 – default), or they are (1)

19 – error codes are negative (0 – default), or positive (1)

20 – ICAX2 for the ”open dir” call in DIR; default value is 128 under SpartaDOS or 0 otherwise.

21 – listing style: BASIC XE (1) or Turbo BASIC XL (0 – default). BASIC XE style means that keywords are listed in lower case except the first character (Graphics 8:Color 1:Plot 0,0:Drawto 319,191), while Turbo BASIC XL style lists everything in upper case (GRAPHICS 8:COLOR 1:PLOT 0,0:DRAWTO 319,191).

22 – starting value for the AUTO command; default is 10

23 – increment value for the AUTO command; default is 10

24 – MultiBASIC interpreter/compiler major version number

25 – MultiBASIC interpreter/compiler minor version number

26 – major version number of the underlying SpartaDOS. If the underlying DOS is not SpartaDOS, there is 0 here.

27 – minor version number of the underlying SpartaDOS. If the underlying DOS is not SpartaDOS, there is 0 here.

28 – if 1, stereo Pokey is available, 0 otherwise.

Most values can be changed with SET.