Nextor BASIC

Nextor

Nextor BASIC = MSX BASIC + X-BASIC + много дополнительных полезных подпрограмм.

Nextor BASIC

Файлы

NestorBASIC 1.11

User's manual in english

NestorBASIC version 1.11
By Nestor Soriano (Konami Man), December 2004

1. WHAT IS NESTORBASIC?

NestorBASIC is a set of machine code routines, integrated in a single file.
It is intended for being used within MSX-BASIC programs. Without losing Turbo-BASIC compatibility, NestorBASIC gives you the following capabilities:

- Full access to all te available memory of the computer (all existing memory in the case of DOS 1, all free memory in the case of DOS 2), up to 4 Mb.
- Access to VRAM, with data block exchange between VRAM and between RAM and VRAM feature.
- Storage of BASIC programs in the mapped RAM, which can be executed without losing the existing variables.
- Access to disk files and direct access to physical sectors, with read/write to RAM/VRAM feature. File search, directory management.
- Graphic compression and decompression.
- Moonblaster music replay. Samplekit load.
- PSG sound effects replay.
- Execution of machine code routines placed on BIOS, SUB-BIOS, normal BASIC memory, system work area, or a mapped RAM segment.
- Machine code routines execution, from BIOS or from any RAM segment. User defined interrupts.
- NestorMan functions, InterNestor Suite and InterNestor Lite routines execution.

All of these functions are available through a single USR and an integer parameter array, therefore they are fully TurboBASIC compatibles. In fact, the TurboBASIC compiler itself is included into the NestorBASIC file, and it is automatically loaded when NestorBASIC is installed.

NestorBASIC is loaded in a RAM segment not used by BASIC, so only a small amount of the BASIC main RAM (aprox. 500 bytes) is needed for a jump routine. The rest of the BASIC main RAM remains available for the BASIC program.

```
2. SYSTEM REQUIREMENTS. LOADING NESTORBASIC

NestorBASIC works in any MSX2/2+/Turbo-R with at least 128K of mapped RAM.
When DOS 2 is installed, at least one free segment is needed in the primary
mapper (at least two if the music replayer will be used. See section 8 for
details).

To load NestorBASIC, just do BLOAD"NBASIC.BIN",R. No previous nor further
CLEAR nor DEFUSR is needed. If the installation is completed successfully,
the following has then happened:

- NestorBASIC and TurboBASIC have been loaded, each one in a different RAM
segment. Both are ready for use.

- The amount of free BASIC main RAM has been decreased in aprox. 500 bytes,
which are occuped by a routine that jumps to the NestorBASIC segment.
```

```
- The first USR [USR0(parameter) or just USR(parameter)] points to the jump
routine mentioned above. This will be the entry point for using the
NestorBASIC functions.

- The integer array P has been created. This array will be used for passing
parameters to, and returning results from the NestorBASIC functions. The
error codes will be returned by the USR command. (The file access function
and other functions like the ones for string processing need its own string
array, which must be defined separately. See section 4 for details). The
first five elements of P have been initialised as follows:

P(0) = Number of available RAM segments, or error code
P(1) = NestorBASIC main version
P(2) = NestorBASIC secondary version, in BCD format (must be shown in
       hexadecimal format)
P(3) = MSX-DOS main version
P(4) = MSX-DOS secondary version, in BCD format (must be shown in
       hexadecimal format)

The number of available RAM segments will be always at least 5. A smaller
number in P(0) means an error code that has aborted the installation of
NestorBASIC:

0: The computer has not mapped RAM, or has only 64K of mapped RAM.
1: Disk error when reading NestorBASIC or TurboBASIC from NBASIC.BIN.
2: No free segments are available in the primary mapper. This error may
   appear only under DOS 2.
3: NestorBASIC was already installed. All variables have been initialized.
4: Undefined in this version.

After NestorBASIC installation, the CLEAR statement can be freely used in
order to reserve memory for loading other machine code routines or user data
in the BASIC main RAM. However, due to the fact that NestorBASIC performs
slot/segment switchings in page 2, and the BASIC interpreter puts the stack
in the BASIC main RAM below the area reserved with CLEAR, there is a lower
limit for the address that can be specified in the CLEAR statement.
Concretely, the lowest address that can be specified in the CLEAR statement
when NestorBASIC is installed is given by:

&HC000 + (MAXFILES+1)*267 + FRE("") + 100

Also, remember that the CLEAR statement, as well as the
load/erase/modification of any BASIC program, delete all the variables. In
such case, the P array must be redefined with DEFINT P:DIM P(15), and this
must be done out of turbo-blocks. Also, the F array must be redefined if
needed (see section 4 for details about the array F). Watch out: if you need
to define the F array inside of a turbo-block, you must do it in the first
line of the turbo-block:

10 'save"autoexec.bas"
20 BLOAD"nbasic.bin",R:IF P(0)<5 THEN PRINT "Error!":END
30 CLEAR 100:DEFINT P:DIM P(15)
40 _TURBO ON(P())
50 DIM F$(1) 'See section 4
...
65000 _TURBO OFF
65010 RUN"next.bas"

10 'save"next.bas"
20 DEFINT P:DIM P(15)
30 _TURBO ON(P())
40 DIM F$(0) 'If F$(1) is not needed. See section 4.1
...
65000 _TURBO OFF
```

Also, remember that the first USR is always reserved for NestorBASIC. USR1
to USR9 remain available for the user.

3. LOGICAL SEGMENTS

3.1. WHAT IS A LOGICAL SEGMENT?

The mapped RAM of the MSX computers is structured in 16K segment. Each RAM slot contains a given number S of segments (numbered 0 to S-1), which are accessibles when they are connected to the addressing space of the active RAM slot through ports &HFC to &HFF.

When NestorBASIC is installed, all the exisiting slots are scanned for available RAM (all the existing RAM in the case of DOS 1, all the free RAM in the case of DOS 2), and a segment table is built. In this table, all the found segments are registered as a couple slot+segment number (under DOS 2, all the segments are first allocated). NestorBASIC identifies each of these couples with his index number in the table. This number is named logical segment number, and allows the user to manage all the RAM segments in an easy and ordered way, without having to care about the slot(s) in which the RAM is located nor about the physical segment numbers.

For example, let's suppose a MSX with 128K of internal RAM (8 segments) and an external mapper of 1024K (64 segments). Then, when NestorBASIC is installed, and supposing DOS 1, the user has 72 logical segments available for its own use. To use these segments, just specify a logical segment number between 0 and 71 in the appropriate NestorBASIC functions, and don't worry about RAM slots nor about physical segments number.

The address range of a logical segment (just "segment" from now on) is &H0000 to &H3FFF. If higher addresses are specified in the NestorBASIC functions, they will be converted. That is, addresses &H4000-&H7FFF, &H8000-&HBFFF and &HC000-&HFFFF are the same as &H0000-&H3FFF when accessing to RAM segments through NestorBASIC.
All the segments are readable and writable, but there are important restrictions that apply to the first six ones:

- Segment 0 contains NestorBASIC itself, and only a small amount of RAM remains available for the user at the end of the segment. Use function 1 to obtain the beginning address of this free area (see section 10 for the function description).

- Segment 1 contains the TurboBASIC compiler. You can overwrite this segment only if you will not use the compiler.

- Segment 2 is always connected to page 2 (addreses &H8000 to &HBFFF), and contains the BASIC program being executed and maybe some variables.

- Segment 3 is always connected to page 3 (addresses &HC000 to &HFFFF), and contains the MSX work area and part of the BASIC program variables. Be careful when writing here.

- Segment 4 is used as an internal buffer by some NestorBASIC functions. You can use this segment to store your own data as long as you don't use these NestorBASIC functions. See section 10 or appendix 1 to know which functions use this segment.

- If it exists, segment 5 is initially free, and is not used by NestorBASIC. But when the music replayer is loaded, it is stored in this segment. See section 8 for details.

All other existing segments are completely available for the user.

WARNING: The roles of logical segments 2 and 4 have been swapped when stepping from version 0.07 to 1.00. In version 0.07 and previous versions, segment 2 was NestorBASIC internal buffer, and segment 4 was BASIC page 2 RAM.

When using the data block exchange functions, be careful for not surpassing the address &H3FFF when adding the block length to the destination address (for example, don't try to transfer &H2000 bytes specifying address &H3000 as destination address). In such a case, segment 0 or segment 3 is overwrited, and then the result is unpredictable.

Note: using function 80 it is possible to force NestorBASIC to allocate less memory segments than the total amount available. This is useful under DOS 2 when other programs which also need to allocate memory (for example RAM disk or NestorMan) are used simultaneously with NestorBASIC. See section 10 for the functions' description.

3.2. USING VRAM AS LOGICAL SEGMENTS

NestorBASIC allows you to use the VRAM for emulate extra RAM segments. If NestorBASIC founds a number S of segments, segment numbers 0 to S-1 refers to RAM segments, as explained above. But segments S to S+7 or S+3 (according to the VRAM capacity, 128K or 64K) are also available, and they refer to VRAM.

Let's return to the previos sample. When installing NestorBASIC, the number of available segments is 72. Then, segments 72 to 79 (if the computer has 128K VRAM) or 72 to 75 (if the computer has 64K VRAM) are available for VRAM access.

The correspondence between segments and VRAM address is reversed: the segments with the lowest number refers to the highest VRAM addresses. This is done in this way in order to help the user when discarding the VRAM segments corresponding to the screen visualization: these segments will be the last ones when working in text mode, or when using the page 0 in graphic modes.

Let's return to the previous sample: the computer with 72 RAM segments and 128K VRAM. If the program works in text mode, only the first 2000 bytes of VRAM are used: this VRAM area corresponds to the last VRAM segment, with number 79. Then, the user can use freely the segments 72 to 78, and the range of available segments becomex 0 to 78, without having to discard any intermediate segment. If the program works in SCREEN 5 and uses only graphic page 0, the range is 0 to 77; in SCREEN 7, it is 0 to 75, and so on.

Of course, NestorBASIC also includes functions for accessing VRAM with direct addressing specification, useful when the VRAM must be treated as VRAM and not as emulated RAM.

WARNING: VRAM segments can be used as normal RAM segments in order to store data and BASIC programs; also, data block exchange functions and disk access funtions are available when using VRAM segments. But note that the VRAM segments can't be used for the following purposes:

- Graphic compression/decompression
- Machine code routines execution
- User defined interrupt execution
- PSG sound effects replay
- Music replay

If you try to use VRAM segments for any of these purposes, the called function will return the "non existing segment" error.

3.3. THE SEGMENT 255

```
The logical segment number 255 has a special meaning. It does not refer to a
concrete RAM or VRAM segment, but to the BASIC main RAM, that is, the RAM
used by BASIC at addresses &H8000 to &HFFFF (in this case, addresses are not
converted to the range &H0000-&H3FFF). This segment number is useful for
example to exchange data between any segment and a BASIC variable or array:

 1 'Copy 10 bytes from segment 7, begin address &H1000,
 2 'to integer array D.
 3 '(See section 10 for detailed functions specification)
 4 '
10 DEFINT D:DIM D(4) '5 integer data = 10 bytes
20 P(0)=7        'Source segment
30 P(1)=&H1000       'Source begin address
40 P(2)=255      'Destination segment=BASIC main RAM
50 P(3)=VARPTR(D(0)) 'Destination begin address=D array
60 P(4)=10       'length
70 J=USR(10)         'Call to function 10 (block transfer between segments)


3.4. SEGMENTS MAP

As a summary of this section, here is a list of the available segments and
its description. S is the number of available segments, given by NestorBASIC
when it is installed or when using function 1.

0: NestorBASIC segment
1: Turbo-BASIC segment
2: BASIC main RAM, page 2 (&H8000-&HBFFF)
3: BASIC main RAM, page 3 (&HC000-&HFFFF)
4: Internal buffer segment
5 to S-1: Available RAM for the user (if S>5)
   If the music replayer is loaded,
   it is stored in segment 5. See section 8.
S:   VRAM, addresses &H1C000-&H1FFFF (64K VRAM: &HC000-&HFFFF)
S+1: VRAM, addresses &H18000-&H1BFFF (64K VRAM: &H8000-&HBFFF)
S+2: VRAM, addresses &H14000-&H17FFF (64K VRAM: &H4000-&H7FFF)
S+3: VRAM, addresses &H10000-&H13FFF (64K VRAM: &H0000-&H3FFF)
S+4: VRAM, addresses &H0C000-&H0FFFF (not available with 64K VRAM)
S+5: VRAM, addresses &H08000-&H0BFFF (not available with 64K VRAM)
S+6: VRAM, addresses &H04000-&H07FFF (not available with 64K VRAM)
S+7: VRAM, addresses &H00000-&H03FFF (not available with 64K VRAM)

S+8 to 254: Not available (if S+8<255)
255: BASIC main RAM (&H8000-&HFFFF)


3.5. ERRORS

All the functions which perform RAM and/or VRAM access returns the error code
-1 if any non existing RAM segment or VRAM address (addresses above &HFFFF in
computers with 64K VRAM) is specified in any input parameter, or if any VRAM
segment or the 255 segment is specified and the function supports only normal
RAM segments.


4. DISK ACCESS

NestorBASIC has functions for disk files management and for the use of other
MSX-DOS capabilities:

* Create/delete/rename/search files.
* Read/write files to/from RAM or VRAM.
* Under DOS 2, move files and get/set file attributes.
* Read/write disk sectors to/from RAM or VRAM.
```

```
* Get/set the default drive/directory.
* Obtain the disk size and free space.
* Under DOS 2, get size/set the RAM disk.
```

4.1. THE ARRAY F$

For passing filenames and pathnames to these functions, a string array named F$ is used. The functions for search, rename and move files, and for parse a pathname need two strings are needed, so F$ must be defined with two elements [DIM F$(1)]. The rest of disk access functions need only one string; therefore, if none of the four functions mentioned above are needed, it is recommended to define the array with only one element [DIM F$(0)], due to the string storage method of TurboBASIC: each string uses 256 bytes in RAM, regardless of its real length.

Also, remember that if you define a turbo-block, the F$ array must be defined in the first line:

```
1000 _TURBO ON(P())
1010 DIM F$(1) or DIM F$(0)
...
65000 _TURBO OFF
```

IMPORTANT: String variables defined outside a turboblock, as well as the other variables, are not usable from inside the turboblock, and are restored when it finishes. For example:

```
10 A$="Outside"
20 _TURBO ON
30 A$="Inside":PRINT A$
40 _TURBO OFF
50 PRINT A$
run
Inside
Outside
```

With NestorBASIC this still being true, but F$() strings need an additional caution. If F$ array has been used outside the turboblock and it will be used again inside, the following line must be placed before CALL TURBO ON:

```
F$(0)=F$(0)+"":F$(1)=F$(1)+""
```

This caution is not needed if you don't care about the old contents of F$, or if F$ will not be defined inside the turboblock.

Another restriction of F$(0) and F$(1) is that they are limited in length to 80 characters; exceeding characters will just be ignored by NestorBASIC functions having these strings as input parameters. All these restrictions are not present for indexes higher than 1 if F$ is defined with more elements; that is, you can define F$ with more than two elements and use F$(2), F$(3)... normally.

4.2. ERRORS

In addition to error -1, described in section 3, the disk access functions have its own error codes.

The following errors can appear only under DOS 1:

1: General error code of MSX-DOS 1. It can be caused by the following error conditions:

```
   - File not found.
   - Invalid filename.
   - File already exists (when renaming).
   - Invalid drive (when being part of a filename).
   - End of file found when reading from a file.
   - Disk full.
   - Root directory full.
   - Function not available under DOS 1.

Under DOS 2, each one of the errors mentioned above have its own error codes,
which is never 1.

2: Invalid file number (no opened file has this number assigned).
3: Too many opened files. The maximum number of files that can be opened
   simultaneously under DOS 1 can be checked by using function 1.

The following errors can appear under DOS 1 and DOS 2, and have the same
error code that their BASIC equivalents:

60: Bad FAT.
62: Invalid drive (when changing the default drive).
68: Write protected disk.
69: Physical disk error.
70: Disk offline.

The DOS 2 error codes are the following:

222: Not enough free DOS 2 internal memory to create the RAM disk or
     to open a file.
219: Invalid drive (when being part of a filename or pathname).
218: Invalid filename.
217: Invalid pathname.
215: File not found.
214: Directory not found.
213: Root directory full.
212: Disk full.
211: File already exists (when renaming or moving a file).
210: Invalid directory movement operation (a directory can't be moved into
     one of its descendants).
209: Read only file (when writing into a file).
208: The directory is not empty (when deleting a directory).
207: Invalid attributes (when changing the attributes of a file/directory).
206: Invalid "." or ".." operation.
205: System file exists (when creating a file, any previous existing file is
     automatically deleted, except the files having the system
     attribute set).
204: System directory exists (same as above).
203: File already exists (when creating a directory).
202: File is open (when deleting, renaming or moving a file, or when changing
     the file attribute with direct filename specification).
199: End of file is found (when reading from a file).
196: Too many opened files (when opening a file).
195: Invalid file number (greatest than 63).
194: Invalid file number (never assigned to an opened file).
188: RAM disk already exists (when creating RAM disk).
187: RAM disk does not exist (when removing RAM disk).


5. GRAPHIC COMPRESSION AND DESCOMPRESSION

NestorBASIC includes functions for the compression of graphic data from VRAM
to RAM, and decompression from RAM to VRAM. The compression format is the
same as the one used by Sunrise in the logo appearing routine, it is
byte-based and is as follows:
```

- Unrepeated bytes (up to 63):

  &B00nnnnnn &Hdd .. &Hdd

  &Bnnnnnn is the number of bytes, &Hdd are the bytes

- Repeated byte (up to 63 times):

  &B01nnnnnn &Hdd

  &Bnnnnnn is the number of times that the byte is repeated, &Hdd is the
  repeated byte

- Repeated byte (up to 16383 times):

  &B10nnnnnn &Bnnnnnnnn &Hdd

  &Bnnnnnnnnnnnnnn is the number of times that the byte is repeated, &Hdd is
  the repeated byte

- End of data mark:

  &B11000000 = &HC0

The (de)compression is performed through consecutive segments; that is, after
(de)compressing from/to address &H3FFF of segment S, the process continues in
address &H0000 of segment S+1.

5.1 ERRORS

The following error codes can be returned by the graphic compresssion and
decompression functions:

-1: Error when compressing or decompressing. The specified segment does not
    exist, refers to VRAM or is the 255. Also, this error appears when an
    invalid VRAM address is specified in any input parameter.
 5: Error when compressing. Segments have been exhausted before having
    compressed all the required graphic data.
 6: Error when decompressing. An invalid data is found, or segments have been
    exhausted before finding the end of data mark.

6. BASIC PROGRAMS STORAGE AND EXECUTION

NestorBASIC includes functions that allows the storage of BASIC programs in
RAM or VRAM segments, and its activation or execution, without losing the
existing variables generated by the original program.

WARNING: For using these functions, the begin address of the BASIC programs
in the BASIC main RAM must be changed, from &H8000 to &H8003; this operation
must be done only once, and BEFORE NestorBASIC is installed. You can
do this in two different ways:

- In direct mode, enter the following commands:

```
POKE &HF676,4
POKE &H8003,0
NEW
```

- From a BASIC program. This is the best way, because you can use the same
program for performing this change and, after that, for loading NestorBASIC.
The first line of the program must be as follows:

```
1 'program.bas
10 IF PEEK(&HF676)<>4 THEN POKE &HF676,4:POKE &H8003,0:RUN"program.bas"
20 'From here, you can load NestorBASIC
```

When a BASIC program stored in a segment will be activated or executed, NestorBASIC stores all the existing variables in segment 4; after that, the new program is copied from the specified segment to the BASIC main RAM, all variables are placed after the program, and the appropriate pointers in the BASIC work area are updated. The last step is the execution of the new BASIC program from the first line, or a jump to the direct mode, depending on the called function (execution or activation of the program).

If a BASIC program is intended for being stored in a segment and executed or activated with these functions, it must be stored with a special header with information about its length, which is needed in the process of concatenating the existing variables to the new program. There is a function that saves the current BASIC program with this header; later, just load this generated file in any segment with the disk access functions, and then you have the program stored and ready for being executed or activated.

6.1 ERRORS

Obviously, if the BASIC commands placed after the USR that calls the execution or activation functions are executed, an error has occurred. Two possible error codes can then be returned:

- Error -1, if the specified segment does not exist. These functions work with VRAM segments, but not with segment 255.
- Error -2, if the BASIC main RAM has not enough size for storing the new progam and the existing variables. Such situation can arise if the new program is biggest than the previous one.

7. MISCELLANEOUS FUNCTIONS

In this group there are various functions for the following purposes:

- Machine code routines execution. Routines placed in BIOS, SUB-BIOS, BASIC main RAM, system work area or any segment (called then "user routines") may be executed.
- Storage of string variables in RAM segments.
- Screen print of a string variable in graphic mode.
- SCREEN 0 blink mode management.
- User defined interrupts definition (machine code routines which will be executed, from any RAM segment, in each 50/60 Hz timer interrupt).
- SEE 3.xx PSG sound effects replay.

Some of these functions need a srtrings array, named F$, for passing parameters and returning results, in addition to the integer array P. See section 4 for more details about array F$.

Some of the NestorBASIC internal machine code routines can be used by the user machine code routines and by the user defined interrupt. See appendix 2 for a detailed description about the use of these routines.

The PSG sound effects editor SEE was created by Fuzzy Logic, and the use of the sound effects created with this editor in commercial programs implies the pay of a small amount of money to the authors. See appendix 4 for more details.

8. THE MUSIC REPLAYER

8.1. INITIALIZATION OF THE REPLAYER

NestorBASIC includes Moonblaster 1.4 and Moonblaster for MoonSound Wave version 1.05 music replayers.

These replayers are not automatically loaded when NestorBASIC is loaded: due to its big length, they don't fit in the NestorBASIC RAM segment, and must be loaded in a separate segment. Thus, for using a replayer, it must be explicitly loaded. Only one replayer can be loaded at the same time.

Function 71 loads and initializes the desired replayer, leaving it ready for use. This function checks if segment 5 exists and belongs to the primary mapper: in this case, the replayer is loaded in this segment, which is no longer available for the user. If segment 5 does not exist or does not belong to primary mapper, the replayer will not be loaded, and an error will be returned by the function.

NBASIC.BIN file contains two versions of the Moonblaster Wave replayer: one is for MSX2/2+ and Turbo-R in Z80 mode, and the other is for Turbo-R in R800 mode. If the computer is a Turbo-R, NestorBASIC decides the version to be loaded according to the processor switched when function 71 is called. Note that if a processor change is performed, this function must be called again in order to load the appropriate replayer: Z80 version does not work in R800 mode, and R800 version makes the system slower in Z80 mode.

When the replayer is installed, a sound chips search is done, and all found chips are marked as enabled. See funtion 73 description for more details about enabling and disabling the sound chips.

8.2. REPLAYER CAPABILITIES

Once the replayer is loaded, you can use the functions that NestorBASIC includes for the following purposes:

- Start playing a music stored in any RAM segment (VRAM segments are not supported).
- Stop playing a music being played.
- Pause/continue the music being played.
- Fade out the music being played, with specification of the fading speed.
- Obtain various information about the music being played (segment and begin address in which the music is stored, songname and samplekit or wavekit, current position and step).
- Obtain the sound chips found.
- Disable the sound chips, which in this case will not be used even if they have been detected.
- Load a Music Module samplekit or a Moonsound wavekit from a file.

While a music is being replayed, all the NestorBASIC functions can be used normally, including the PSG sound effects replay and the use of user defined interrupts. If NestorBASIC is uninstalled, the music being replayed will be automatically stopped.

Moonblaster 1.4 replayer fits in 4.5K, therefore the space between addresses &H1200 and &H3FFF on segment 5 remains free and can be used, for example, for storing the music to be replayed. This does not occur with Moonblaster Wave replayer, which uses itself the whole segment 5.

Moonblaster 1.4 musics must be stored in a single segment, so its maximum length is 16K. Moonblaster Wave musics can be stored through consecutive segments, maximum 3: when reading music data during the replaying, if address &H3FFF of a segment is reached, the process continues in address 0 of the next segment.

In order to load a music through consecutive segments, the following sample code can be used:

```
1000 'Loading a music through consecutive segments,
1010 'starting in segment S, address A
1020 F$(0)="music.mwm":P(2)=S:P(3)=A
1030 E=USR(31):IF E<>0 THEN 10000
1040 P(4)=&H4000:E=USR(33)
1050 IF (E<>0 AND E<>1 AND E<>199) THEN 10000
1060 IF E=0 THEN P(2)=P(2)+1:P(3)=0:GOTO 1040
1070 E=USR(32):IF E<>0 THEN 10000
...
10000 'Disk error E handling routine
...
```

WARNING: A Moonblaster Wave music can start at any address of a segment, if it continues at the beginning of the next segment. However the first 800 bytes of the music, containing pattern table and various pointers, must be entirely stored in the first segment.

8.3 ERRORS

The following error codes can be returned by the music replay functions:

- Error 7, returned by the sound chips activation function, pause function and fade out function, if any of the input parameters is invalid.
- Error 12, returned by the play start function and sound chips enabling function, if the replayer is not installed. The rest of the functions will simply do nothing in such case.

The following errors can be returned by the replay start function:

```
-1: The specified segment does not exist, refers to VRAM or is the 255.
12: The replayer is not installed.
13: The music was saved in EDIT mode and can't be replayed
    (Moonblaster 1.4 replayer).
    In the specified address there is no Moonblaster Wave music,
    or there is a Moonblaster Wave music not saved in USER mode.
    (Moonblaster Wave replayer).
14: Other music is currently being replayed.
```

MoonSound wavekit load function can return the following error, apart from the disk access errors:

```
15: In the current position of the specified file there is not placed a
    Moonblaster wavekit, or there is placed a wavekit not saved in USER mode.
```

NOTE: Moonblaster 1.4 replayer can't detect if the data starting in the specified address is actually a Moonblaster 1.4 music, and relies only in the first byte for deciding if data is a Moonblaster EDIT music. Moonblaster Wave replayer has not these restriction.

The replayer initialization function returns the same error codes as the disk access functions, and the error code -1 if the segment 5 does not exists or does not belong to the primary mapper.

9. INTERACTION WITH NESTORMAN AND INTERNESTOR SUITE/LITE

NestorBASIC has special functions that allow interaction with NestorMan (resident dynamic memory manager for MSX-DOS 2), InterNestor Suite (TCP/IP stack for MSX-DOS 2) and InterNestor Lite (TCP/IP stack for MSX-DOS 1/2), if these programs are installed. Therefore it is

possibole to develop BASIC applications that use dynamic memory blocks and chained lists, as well as Internet based applications. To check if NestorMan and InterNestor Suite are installed, use function 81. The procedure for checking if InterNestor Lite is installed is detailed in section 9.4.

Note: NestorMan and InterNestor Suite/Lite have their own manuals describing the
functions and routines provided by each one. These programs are available for download at http://msx.konamiman.com

9.1. NESTORBASIC SEGMENTS AND NESTORMAN SEGMENTS

NestorMan uses a logical segments system very similar to the system used by NestorBASIC. Both segments space, NestorBASIC's and NestorMan's, are independent to each other, with the following exceptions:

- Segments 0, 1, 2 and 3 are common to NestorBASIC and NestorMan (in NestorMan's manual these segments are referred to as "TPA segments").

- If NestorMan is present at NestorBASIC installation time, NestorBASIC logical segment 4 is not allocated using DOS 2 mapper support routines, as are the other segments. Instead, NestorMan function 7 is used for this; this way, NestorBASIC segment 4 has a NestorMan segment number assigned as well (this number can be obtained using NestorBASIC function 81). This segment is reserved with the "exclusive" attribute set, therefore it is not used by NestorMan for memory block reservations.

If NestorMan and/or InterNestor Suite will be used together with NestorBASIC, it is recommended to limit the amount of RAM segments that NestorBASIC will use (use function 80 for this). Otherwise NestorBASIC will allocate for itself all the available RAM segments, and therefore NestorMan will not be able to perform RAM segments allocation (this is needed for memory block reservations, chained lists creation, and for sending/receiving data to/from Internet).

To call NestorMan functions you can use function 82; or you can use function 58 specifying the extended BIOS hook (&HFFCA) as the calling address, &H2202 in the registers pair DE, and the functin number in register C. See section 10 for the functions de
scription.

9.2. DATA EXCHANGE BETWEEN NESTORBASIC AND NESTORMAN

When using NestorMan through NestorBASIC, normally you will need to perform data transfers between a NestorBASIC segment and a NestorMan segment. There are three ways to do this:

1) If the source or destination segment is a TPA segment (segment number between 0 and 3), it is sufficient to use the appropriate NestorMan function for data block transfer (function number 14), since NestorMan can see these segments.

2) NestorBASIC segment 4 can be used as an intermediate buffer for the transfer. For example, suppose a transfer from NestorBASIC to NestorMan. S1 is NestorBASIC source segment, S2 is NestorMan destination segment, and S3 is NestorMan segment number of NestorBASIC segment 4. Then, first you do a transfer S1->4 using NestorBASIC data transfer function (function number 10), and then you do a transfer S3->S2 using NestorMan data transfer function (function number 14).

3) NestorBASIC functions 83 and 84 are for data transfer from a NestorMan segment to a NestorBASIC segment and vice-versa. To read or write only one

byte of data on a NestorMan segment, the easiest way is to use the functions that NestorMan provides for this purpose (function numbers 12 and 13).

Remember that some NestorBASIC functions use segment 4 as a temporary data storage buffer. Refer to section 10 or appendix 1 to know which are these functions.

9.3. USING INTERNESTOR SUITE

The procedures for using InterNestor Suite from NestorBASIC are similar to those for using NestorMan explained above. However the following should be noted:

- Function 85 is for executing InterNestor Suite routines.

- To read and write data on the InterNestor Suite segments (configuration constants and variables), first use function 81 to obtain the segment numbers of the InterNestor Suite modules (each module resides on a NestorMan segment), and from this point, use the data exchange procedures explained in previous section.

- InterNestor Suite routines for reading/writing TCP data or UDP datagrams only allow the use of TPA segments as the source/destination for the data or the datagrams. For this purpose you can use the final portion of NestorBASIC segment (segment number 0) as intermediate storage buffer. This space will always be at least 600 bytes long, regardless of the NestorBASIC version being used; this is enough to store a standard datagram of up to 576 bytes (or to store up to 600 bytes of TCP data).

The sample program TCPCON-S.BAS, supplied with NestorBASIC, illustrates the ensemble use of NestorBASIC and InterNestor Suite.

9.4. USING INTERNESTOR LITE

InterNestor Lite is much simpler than InterNestor Suite, hence using it from NestorBASIC is also a simpler task. Only one function is provided (function 86), allowing the execution of any routine of the InterNestor Lite code segment. To read or write in its data segment, you must use the GET_VAR, SET_VAR and COPY_DATA routines of the code segment.

To know whether InterNestor Lite is installed or not, use function 58 issue a call to the extended BIOS hook (address &HFFCA), passing A=0 and DE=&H2203. If A<>0 at return, then InterNestor Lite is installed. See the description of function 86 for details.

Many of the InterNestor Lite routines use TPA addresses as the source or destination when exchanging data with applications. For these routines you can specify addresses above &H8000, that will refer to the BASIC main memory and the system work area; or addresses below &H4000, that will refer to NestorBASIC segment (segmento number 0).

At the end of the NestorBASIC segment there is a free area, whose size depends on the NestorBASIC version but will never be smaller than 600 bytes, which can be used as a temporary buffer for exchanging data with InterNestor Lite. In other words, you can use the space between H3DA8 and &H3FFF as the source or destination TPA area for a data exchange between NestorBASIC and InterNestor Lite.

The sample program TCPCON-L.BAS, supplied with NestorBASIC, illustrates the ensemble use of NestorBASIC and InterNestor Lite.

9.5. ERRORS

Function 86 (InterNestor Lite routine execution)returns error -1 if
InterNestor Lite is not installed.

Function 85 (InterNestor Suite routine execution) returns error -1 if
InterNestor Suite is not installed, or if an invalid module number is
specified in P(0) (valid segment numbers are 1 to 4).

Functions 83 and 84 (data transfer between a NestorMan segment and a
NestorBASIC segment) return error -1 if a non existing NestorMan or
NestorBASIC segment number is specified. VRAM segments and segment 255 can be
used with these functions.

Functions 80, 81 and 82 never return error.

10. NESTORBASIC FUNCTIONS

10.1. GENERAL DESCRIPTION

All the NestorBASIC functions available for the user are identified by the
number passed as a parameter to the USR command. That is, for using the
function with number F you just do USR(F)M remember that if F is a variable
and not an immediate number, it must be variable of integer type. Parameters
for the function must be set in the array P (and, in some cases, also in the
array F$) before calling the function. Once executed, the results of the
function will be returned in the same arrays, P and/or F.

Items of P and F$ not explicitly mentioned in the results list of each
funtion are not modified, except the segment addresses; these are converted
to the range &H0000-&H3FFF (except in the case of the segment 255). When a
function returns an error, the results will not be valid (P and F$ are not
modified), except if otherwise is specified in the description of the
funtion.

"VRAM block" refers to 64K lower VRAM (block 0) or to 64K upper VRAM (block
1). VRAM addresses have the range &H0000-&HFFFF. If any VRAM address surpass
&HFFFF after an autoincrement, the new address will be &H0000 and the VRAM
block will be reversed (from 0 to 1, or from 1 to 0).

The USR command will return an error code, or 0 if there is not error.
Specific error codes for each functions group are detailed in each section.

Appendix 1 contains a list of all available functions. Use it as a quick
reference.

Functions that use segment 4 have a "(S4)" mark after their name. These
functions are: 0, 26-28, 30, 33-41, 55-57, 71, 78, and 79.

10.2. GENERAL FUNCTIONS

* Function 0: NestorBASIC uninstallation (S4)

Input: P(0) = 0 -> Don't free BASIC main RAM area reserved by NestorBASIC
     P(0) <>0 -> Free the BASIC main RAM area reserved by NestorBASIC
Out: -

This function uninstalls NestorBASIC: makes the USR unusable and, in the case
of DOS 2, frees all the reserved segments. Also, all interrupt process (user
defined interrupt, PSG sound effects replay and music replay) are stopped.
NestorBASIC should always be uninstalled before returning to DOS; otherwise,
all the allocated RAM segments will not be freed, and will therefore be

```
unusable until the computer is reset.

Before uninstalling NestorBASIC, check that no files still open. Else, if you
have done file writings, you can lost some data in the internal DOS buffers.

If P(0)=0 is specified, the BASIC main RAM zone ocuped by the jump routine
(aprox. 500 bytes) is not freed; then, any memory reservation done with the
CLEAR statement after the installation of NestorBASIC remains valid. If
P(0)<>0 is specified, a new CLEAR statement is automatically executed, and
the highest RAM address usable for BASIC programs and variables becomes the
same as before the NestorBASIC installation; that is, FRE(0) returns the same
value as before the installation of NestorBASIC. Note that in this case,
variables are initialized.

This function never returns error.


* Function 1: General information about NestorBASIC and about a segment

Input:   P(10)= Logical segment to investigate
Output:  P(0) = Number of available RAM segments
     P(1) = NestorBASIC main version
     P(2) = NestorBASIC secondary version, in BCD format
        (must be shown in hexadecimal format)
     P(3) = MSX-DOS main version
     P(4) = MSX-DOS secondary version, in BCD format
        (must be shown in hexadecimal format)
     P(5) = Amount of memory occuped in the BASIC main RAM
        by the jump routine
     P(6) = VRAM size in K (64 or 128)
     P(7) = Begin address of the free area in segment 0
                (&H3DA8 at most)
     P(8) = Number of last function called
     P(9) = Number of currently opened files
     P(10)= Maximum number of simultaneously opened files
        (valid only under DOS 1)
     P(11)= Slot that contains the logical segment specified in P(0)
        (255 if this segment does not exist or refers to VRAM)
     P(12)= Physical segment number of the logical segment
        specified in P(0)
     F$(0)= Complete path of NBASIC.BIN file

In the case of DOS 2, the maximum number of simultaneously opened files
depends on the state of the DOS internal memory, but will never be bigger
than 63.

This function returns an error code of -1 if the logical segment specified in
P(0) does not exist, refers to VRAM or is the 255. However, even in this case
the results returned in P(0) to P(10) will be valid.

The number returned in P(0) by this function is the same as the number
returned in P(0) by function 81; that is, it is the number of segments
allocated by NestorBASIC. By default (if function 81 has never been called
after installing NestorBASIC), NestorBASIC allocates for itself all the
available RAM segments (up to 247) at installation time.

F$(0) will be just a drive letter and a colon under DOS 1 (for example "A:"),
and a drive+directory finished with "\" under DOS 2 (for example
"C:\UTILS\AMAZING\").

Address returned in P(7) dependes on NestorBASIC version, but it will be
always equal to or smaller than &H3DA8; that is, at least 600 bytes will be
always available at the end of the NestorBASIC segment. This area may be
used, for example, for temporary storage of TCP data or a UDP datagram when
```

```
using InterNestor Suite with NestorBASIC.

P(8) returns the number of the last function called, but function 1 itself is
not counted. So if you call, for example, functions 64, 3, 10, 1, 1, 1 in
sequence, you will obtain P(8)=10 at the end. A value of zero means that no
functions were called other than function 1 since NestorBASIC was installed.


10.3. FUNCTIONS FOR LOGICAL SEGMENTS ACCESS

* Function 2: Read a byte from a segment

Input:   P(0) = Segment
     P(1) = Address
Output:  P(2) = Readed byte


* Function 3: Read a byte from a segment with address autoincrement

Input:   P(0) = Segment
     P(1) = Address
Output:  P(2) = Readed byte
     P(1) = P(1) + 1


* Function 4: Read an integer (2 bytes) from a segment

Input:   P(0) = Segment
     P(1) = Address
Output:  P(2) = Readed integer

Low byte is read from address P(1), and high byte from address P(1)+1.


* Function 5: Read an integer (2 bytes) from a segment
          with address autoincrement

Input:   P(0) = Segment
     P(1) = Address
Output:  P(2) = Readed integer
     P(1) = P(1) + 2

Low byte is read from address P(1), and high byte from address P(1)+1.


* Function 6: Write a byte to a segment

Input:   P(0) = Segment
     P(1) = Address
     P(2) = Byte to be written
Output:  -


* Function 7: Write a byte to a segment with address autoincrement

Input:   P(0) = Segmento
     P(1) = Address
     P(2) = Byte to be written
Output:  P(1) = P(1) + 1


* Function 8: Write an integer (2 bytes) to a segment

Input:   P(0) = Segment
```

```
     P(1) = Address
     P(2) = Integer to be written
Output:  -

Low byte is written to address P(1), and high byte to address P(1)+1.


* Function 9: Write an integer (2 bytes) to a segment
          with address autoincrement

Input:   P(0) = Segment
     P(1) = Address
     P(2) = Integer to be written
Output:  P(1) = P(1) + 2

Low byte is written to address P(1), and high byte to address P(1)+1.


* Function 10: Data block transfer between segments

Input:   P(0) = Source segment
     P(1) = Source start address
     P(2) = Destination segment
     P(3) = Destination start address
     P(4) = Data length
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(1) = P(1) + P(4) if P(5)<>0
     P(3) = P(3) + P(4) if P(6)<>0

P(3)+P(4) must be lower than &H4000, otherwise the result is unpredictable.


* Function 11: Fill a RAM zone with a byte

Input:   P(0) = Segment
     P(1) = Begin address
     P(2) = Byte
     P(3) = Zone length
Output:  -

P(3)+P(4) must be lower than &H4000, otherwise the result is unpredictable.


* Function 12: Fill a RAM zone with a byte with address autoincrement

Input:   P(0) = Segment
     P(1) = Begin address
     P(2) = Byte
     P(3) = Zone length
Output:  P(1) = P(1) + P(3)

P(3)+P(4) must be lower than &H4000, otherwise the result is unpredictable.


10.4. FUNCTIONS FOR VRAM ACCESS

* Function 13: Read a byte from VRAM

Input:   P(0) = VRAM block
     P(1) = Address
Output:  P(2) = Readed byte
```

```
* Function 14: Read a byte from VRAM with address autoincrement

Input:   P(0) = VRAM block
     P(1) = Address
Output:  P(2) = Readed byte
     P(0):P(1) = P(0):P(1) + 1


* Function 15: Read an integer (2 bytes) from VRAM

Input:   P(0) = VRAM block
     P(1) = Address
Output:  P(2) = Readed integer

Low byte is read from address P(1), and high byte from address P(1)+1.


* Function 16: Read an integer (2 bytes) from VRAM
           with address autoincrement

Input:   P(0) = VRAM block
     P(1) = Address
Output:  P(2) = Readed integer
     P(0):P(1) = P(0):P(1) + 2

Low byte is read from address P(1), and high byte from address P(1)+1.


* Function 17: Write a byte to VRAM

Input:   P(0) = VRAM block
     P(1) = Address
     P(2) = Byte to be written
Output:  -


* Function 18: Write a byte to VRAM with address autoincrement

Input:   P(0) = VRAM block
     P(1) = Address
     P(2) = Byte to be written
Output:  P(0):P(1) = P(0):P(1) + 1


* Function 19: Write an integer (2 bytes) to VRAM

Input:   P(0) = VRAM block
     P(1) = Address
     P(2) = Integer to be written
Output:  -

Low byte is written to address P(1), and high byte to address P(1)+1.


* Function 20: Write an integer (2 bytes) to VRAM
           with address autoincrement

Input:   P(0) = VRAM block
     P(1) = Address
     P(2) = Integer to be written
Output:  P(0):P(1) = P(0):P(1) + 2

Low byte is written to address P(1), and high byte to address P(1)+1.
```

```
* Function 21: Data block transfer from VRAM to RAM

Input    P(0) = Source VRAM block
     P(1) = Source start address (VRAM)
     P(2) = Destination segment
     P(3) = Destination start address (RAM)
     P(4) = Block length
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(1) = P(1) + P(4) if P(5)<>0
     P(2):P(3) = P(2):P(3) + P(4) if P(6)<>0

P(1)+P(4) must be lower than &H4000, otherwise the result is unpredictable.


* Function 22: Data block transfer from RAM to VRAM

Input:   P(0) = Source segment
     P(1) = Source start address (RAM)
     P(2) = Destination VRAM block
     P(3) = Destination start address (VRAM)
     P(4) = Block length
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(1) = P(1) + P(4) if P(5)<>0
     P(2):P(3) = P(2):P(3) + P(4) if P(6)<>0


* Function 23: Data block tranfer between VRAM

Input:   P(0) = Source VRAM block
     P(1) = Source start address
     P(2) = Destination VRAM block
     P(3) = Destination start address
     P(4) = Block length (maximum &H4000)
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(0):P(1) = P(0):P(1) + P(4) if P(5)<>0
     P(2):P(3) = P(2):P(3) + P(4) if P(6)<>0

If P(4) is greatest than &H4000 only &H4000 bytes will be transferred, and
the addresses increment will be also limited to &H4000, but P(4) will not be
modified. Thus, for transferring largest blocks (up to 64K) in a easy way you
can do something like the following:

10 'P(0) to P(3) already set. Data block length set in L.
20 P(5)=1:P(6)=1
30 P(4)=VAL("&H"+HEX$(L)):J=USR(23):L=L-&H4000:IF L>0 THEN 30

The decimal to hexadecimal and further hexadecimal to decimal conversion of L
is needed due to the integer variables range in BASIC: -32768 to 32767; if a
biggest number is directly assigned to P(4), an overflow error will be
generated.

Since TurboBASIC manages floating point variables in a special way, if you
use this method inside a turbo-block, L must be an integer multiply of 256.


* Function 24: Fill a VRAM zone with a byte

Input:   P(0) = VRAM block
     P(1) = Start address
     P(2) = Byte
     P(3) = Zone length (maximum 16K)
```

```
Output:  -

For fill a zone largest than 16K, use the method described in function 23.


* Function 25: Fill a VRAM zone with a byte with address autoincrement

Input:   P(0) = VRAM block
     P(1) = Start address
     P(2) = Byte
     P(3) = Zone length (maximum 16K)
Output:  P(0):P(1) = P(0):P(1) + P(3)

For fill a zone largest than 16K, use the method described in function 23.


10.5. DISK ACCESS FUNCTIONS

* Function 26: Search for a file (S4)

Input:   F$(1) = Searching mask (empty string="*.*")
     P(0)  = 0 -> Seacrh for first mactching filename
     P(0)  = 1 -> Seacrh for next matching filename
     P(1)  = Search attributes: 2*H + 4*S + 8*V + 16*D
         H = 1 -> Include hidden files in the search
         S = 1 -> Include system files in the search
         V = 1 -> Search only the volume label
         D = 1 -> Include subdirectories in the search
Output:  F$(0) = Found filename
     P(0)  = 1
     P(1)  = File attributes (always 0 under DOS 1):
         R + 2*H + 4*S + 8*V + 16*D + 32*A
         R = 1 -> Read only
         H = 1 -> Hidden
         S = 1 -> System
         V = 1 -> Volume label
         D = 1 -> Directory
         A = 1 -> Archive
     P(2)  = Hour of creation/last modification (0 to 23)
     P(3)  = Minute of creation/last modification (0 to 59)
     P(4)  = Day of creation/last modification (1 to 31)
     P(5)  = Month of creation/last modification (1 to 12)
     P(6)  = Year of creation/last modification (1980 to 2079)
     P(7)  = First cluster (2 to 4095)
     P(8)  = Logical drive (0=A:,...,7=H:)
     P(9)  = File length (lower part)
     P(10) = File length (higher part)
     P(11) = 0, if searching first no file is found
         1, if searching first a file is found
         P(11), if searching next no file is found
         P(11)+1, if searching next a file is found

F$(1) can include a drive letter and, in the case of DOS 2, a pathname; F$(0)
will contain only the filename found.

P(0) must be set to 0 to search the first filename matching the specified
mask, and must be left to 1 to search the following filenames. This is done
automatically by the function (P(0) is always set to 1). Besides, P(11) is
set to 1 when the first filename is searched (0 if none is found), and is
increased automatically in each successive search. Thus, to search all the
matching filenames you can use a loop like the following one:

10 'Search of all the filenames in drive B: (default directory)
20 F$(1)="B:":P(0)=0:P(1)=2+16 'Include directories and hidden files
```

```
30 IF USR(26)<>0 THEN PRINT:PRINT"Total found:";P(11):END
40 PRINT F$(0),CHR$(-ASC("h")*((P(1)AND2)<>0));
           CHR$(-ASC("d")*((P(1)AND16)<>0));
           CHR$(-ASC("a")*((P(1)AND32)<>0)):GOTO 30
```

When there is no more matching files/directories remaining, the function will return the "File not found" error. Under DOS 1, an error when searching the first matching filename may also mean an invalid drive or filename specification in the input mask (under DOS 2, these errors have its own error code).

To obtain the file length, use the expression P(9)+65536*P(10). When printing these lengths in the screen, remember that the numeric variables screen representation is different for BASIC and TurboBASIC when the number is big; therefore, it is better to print the file size in K: INT(P(9)/1024)+64*P(10).

* Function 27: Rename a file (S4)

```
Input:   F$(0) = File name
     F$(1) = New file name
Output:  -
```

F$(0) can include a drive letter and, in the case of DOS 2, also a pathname; F$(1) must contain only the new filename. Under DOS 1 you can rename several files at the same time by using wildcards; under DOS 2 you can rename only one file at the same time.

* Function 28: Delete a file

```
Input:   F$(0) = File name
Output:  -
```

F$(0) can include a drive letter and, in the case of DOS 2, a pathname. Under DOS 2 this function will return an error if you try to delete an open file. Under DOS 1 you can delete opened files, but this is not advisable because further attempts to access this file may cause unpredictable results.

Under DOS 1 you can delete several files at the same time by using wildcards; under DOS 2 you can delete only one file at the same time.

* Function 29: Move a file (DOS 2)

```
Input:   F$(0) = File name
     F$(1) = New file location
Output:  -
```

This function is only available under DOS 2. Under DOS 1 will always return the error code 1.

F$(0) can include a drive letter and a pathname. F$(1) can not include a filename (the file is always moved with the same name) nor a drive letter (a file can be moved only to other directory on the same drive). If you specify a directory instead of a file in F$(0), all its subdirectories and contained files will also be moved.

* Function 30: Create a file or directory

```
Input:   F$(0) = Name of the file or subdirectory
     P(0)  = Creation attributes (ignored under DOS 1):
         R + 2*H + 4*S when creating a file;
```

```
         2*H + 16 when create a directory.
         R = 1 -> Read only
         H = 1 -> Hidden
         S = 1 -> System

F$(0) can include a drive letter and a pathname. The file will be created
with a length of 0 bytes, and will not remain open: for accessing the file,
it must be first explicitly opened (use function 31 to open a file).

Under DOS 1 you can only create files, and P(0) is ignored. Under DOS 2 the
files are always created with the "Archive" attribute set, in addition to the
attributes specified in P(0).


* Function 31: Open a file

Input:   F$(0) = File name
Output:  P(0)  = Number assigned to the file

F$(0) can contain a drive letter and, in the case of DOS 2, also a pathname.
The number returned in P(0) identifies the file, and must be specified in
further calls to the file access functions. The concrete value of this number
depends on the DOS version and the number of previously opened and closed
files: don't rely on it as a reference of the number of currently opened
files (use function 1 to obtain the number of currently opened files).

Under DOS 1 an error code of 3 will be returned if there is already too many
opened files. Use function 1 to obtain the maximum number of simultaneously
opened files under DOS 1.


* Function 32: Close a file

Input:   P(0) = File number
Output:  -

It is very important to close a file that will no longer be accessed:
otherwise, if the file has been written you can lose some data in the DOS
internal buffers; besides, in this case the directory entry of the file will
not be updated.


* Function 33: Read from a file (S4)

Input:   P(0) = File number
     P(2) = Destination segment
     P(3) = Destination start address
     P(4) = Number of bytes to read
     P(6) = Increase P(3) if <>0
Output:  P(7) = Number of bytes actually read
     P(3) = P(3)+P(4) if P(6)<>0

P(3)+P(4) must be lowest than &H4000, otherwise the result is unpredictable.

The file is read from the position indicated by its file pointer; this
pointer is automatically increased after the read. To examinate or modify
a file pointer, use function 42.

To read till the end of the file, or to read the whole file if it is shorter
that 16K, just try to read 16K (set P(4)=&H4000), and ignore the returned
error if it is 1 or 199: for this function, these errors mean only
that the end of the file has been reached.

If the end of the file is reached before having read P(4) bytes an error will
```

```
be returned; to obtain the number of read bytes look at P(7). If no error is
returned, P(7) will have the same value as P(4); in the case of a physical
error (bad FAT, physical disk error or disk offline), P(7) will always be 0.


* Function 34: Read from a file to VRAM (S4)

Input:   P(0) = File number
     P(2) = Destination VRAM block
     P(3) = Destination start address
     P(4) = Number of bytes to read (maximum &H4000)
     P(6) = Increase P(3) if <>0
Output:  P(7) = Number of bytes actually read
     P(2):P(3) = P(2):P(3)+P(4) if P(6)<>0

If P(4) is greater than &H4000, only &H4000 bytes will be read, and the
address increment will also be limited to &H4000, but P(4) will not be
modified. Thus, to read larger blocks (up to 64K) you can use the method
described in function 23.

See function 33 for explanations about the file pointer and the value
returned in P(7).


* Function 35: Read disk sectors (S4)

Input:   P(0) = Drive (0=A:,...,7=H:)
     P(1) = First sector number
     P(2) = Destination segment
     P(3) = Destination start address
     P(4) = Number of sectors to read (maximum 32)
     P(6) = Increase P(3) if <>0
Output:  P(7) = P(4)*512 (0 in error condition)
     P(3) = P(3)+P(4)*512 if P(6)<>0

P(3)+P(4)*512 must be lower than &H4000, otherwise the result is
unpredictable.

This function does not support partial readings: if there is no error, P(7)
will return the number of read bytes, that is, P(4)*512; but in case of error
P(7) will be always 0. If P(6)<>0, P(3) will be increased by the number of
readed bytes (P(7) or 0).


* Function 36: Read disk sectors to VRAM (S4)

Input:   P(0) = Drive (0=A:,...,7=H:)
     P(1) = First sector number
     P(2) = Destination VRAM block
     P(3) = Destination begin address
     P(4) = Number of sectors to read (maximum 32)
     P(6) = Increase P(3) if <>0
Output:  P(7) = P(4)*512 (0 in error condition)
     P(3) = P(3)+P(4)*512 if P(6)<>0

If P(4) is greater than 32 only 32 sectors (&H4000 bytes) will be read, and
the address increment will also be limited to &H4000, but P(4) will not be
modified. Thus, to read larger blocks (up to 64K) you can use the method
described in function 23.

See function 35 for an explanation about the value returned in P(7).


* Function 37: Write to a file (S4)
```

```
Input:   P(0) = File number
     P(2) = Source segment
     P(3) = Source start address
     P(4) = Number of bytes to be written
     P(6) = Increase P(3) if <>0
Output:  P(7) = Number of bytes actually written
     P(3) = P(3)+P(4) si P(6)<>0

P(3)+P(4) must be lower than &H4000, otherwise the data written to the file
is unpredictable.

Data is written to the file from the position indicated by its file pointer;
this pointer is automatically increased after the writing. For examinate or
modify a file pointer use function 42. In the case of a physical error (bad
FAT, physical disk error or disk offline) P(7) will always be 0.


* Function 38: Write to a file from VRAM (S4)

Input:   P(0) = File number
     P(2) = Source VRAM block
     P(3) = Source start address
     P(4) = Number of bytes to be written (maximum &H4000)
     P(6) = Increase P(3) if <>0
Output:  P(7) = Number of bytes actually written
     P(2):P(3) = P(2):P(3)+P(4) if P(6)<>0

If P(4) is greater than &H4000, only &H4000 bytes will be written, and the
address increment will also be limited to &H4000, but P(4) will not be
modified. Thus, to write larger blocks (up to 64K) you can use the method
described in function 23.

See function 37 for an axplanation about the file pointer.


* Function 39: Write to disk sectors (S4)

Input:   P(0) = Drive (0=A:,...,7=H:)
     P(1) = First sector number
     P(2) = Source segment
     P(3) = Source start address
     P(4) = Number of sectors to be written (maximum 32)
     P(6) = Increase P(3) if <>0
Output:  P(7) = P(4)*512 (0 in error condition)
     P(3) = P(3)+P(4)*512 if P(6)<>0

P(3)+P(4)*512 must be lowest than &H4000, otherwise the written data is
unpredictable.

This function does not supports partial writes: if there is no error, P(7)
will return the number of written bytes, that is, P(4)*512; but in case of
error P(7) will be always 0. If P(6)<>0, P(3) will be increased by the number
of written bytes (P(7) or 0).


* Function 40: Write to disk sectors from VRAM (S4)

Input:   P(0) = Drive (0=A:,...,7=H:)
     P(1) = First sector number
     P(2) = Source VRAM block
     P(3) = Source start address
     P(4) = Number of sectors to be written (maximum 32)
     P(6) = Increase P(3) if <>0
Output:  P(7) = P(4)*512 (0 in error condition)
```

```
     P(3) = P(3)+P(4)*512 si P(6)<>0

If P(4) is greatest than 32 only 32 sectors (&H4000 bytes) will be written,
and the address increment will also be limited to &H4000, but P(4) will not
be modified. Thus, to write largest blocks (up to 64K) you can use the method
described in function 23.

See function 39 for an explanation about the value returned in P(7).


* Function 41: Fill a file with one byte (S4)

Input:   P(0) = File number
     P(1) = Byte
     P(4) = length (maximum &H4000)
Output:  P(7) = Number of bytes actually written

This function just writes P(4) bytes equal to the value indicated in P(1) in
segment 4, beginning at address 0, and then calls function 37 with P(2)=4 and
P(3)=0. Thus, all the explanations given in function 37 are also valid here.


* Function 42: Move a file pointer

Input:   P(0) = File number
     P(1) = Moving method:
        0 -> Relative to the beginning of the file
        1 -> Relative to the current position of the pointer
        2 -> Relative to the end of the file
     P(2) = Signed offset (lower part)
     P(3) = Signed offset (higher part)
Output:  P(4) = New file pointer position (lower part)
     P(5) = New file pointer position (higher part)

TO set P(2) and P(3) with an offset D greater than 32768 or lower than
0 you must use the followin formula:

P(3)=INT(D/65536):P(2)=VAL("&H"+HEX$(D-(P2*65536)))

and to obtain the new pointer P:

P=P(4)-(65536*(P(4)<0))+65536!*P(5)

Since TurboBASIC manages floating point variables in a special way, if you
use these expressions inside a turbo-block, D must be an integer multiply of
256.

To obtain the current file pointer position, just use this function with
P(1)=1, P(2)=0 and P(3)=0; to obtain the file length, use it with P(1)=2,
P(2)=0 and P(3)=0. In the first case the file pointer remains unmodified.

Rememeber that the offset is interpreted as a signed number. Therefore, when
using this function note with the following:

- If you use this function with P(1)=0, the offset must be positive.
Otherwise, further access to the file will have unpredictable results.

- If you use this function with P(1)=1, a positive offset will cause the file
pointer to advance; a negative offset will cause the file pointer to move
back.

- If you use this function with P(1)=2, the offset should normally be
negative; otherwise, the file pointer will be placed beyond the file length.
In this case, a further read of the file will cause an "End of file" error (0
```

```
bytes will be read), and a write to the file will cause the space between the
end of the file and the position indicated by the file pointer to be filled
with zeros.


* Function 43: Obtain the default drive and the available drives vector

Input:   -
Output:  P(0) = Default drive (0=A:,...,7=H:)
     P(1) = Available drives vector

The available drives vector is a byte that contains, in each bit, information
about the existence of each drive letter (bit=1: exists, bit=0: does not
exist); the lower bit refers to drive A: and the upper bit refers to drive
H:. For example, if the function returns P(1)=&B01000011, then the available
drives are A:, B: and G:.


* Function 44: Set the default drive

Input:   P(0) = Drive to be set (0=A:,...,7=H:)
Output:  -

This function returns the error code 62 to an attempt of setting an invalid
drive (not available or higher than 7).


* Function 45: Obtain disk space information

Input:   P(0) = Drive (0=Default drive, 1=A:,...,8=H:)
Output:  P(1) = Sectors per cluster
     P(2) = Total number of clusters (maximum 4096)
     P(3) = Number of free clusters

The free space in bytes can be obtained with the expression P(1)*P(3)*512, or
in K with P(1)*P(3)/2; and the same for the total disk capacity, using P(2)
instead of P(3).

This function returns the error code 62 to an attempt of obtaining
information for an invalid drive (not available or higher than 8).


* Function 46: Obtain the default directory (DOS 2)

Input:   P(0) = Drive (0=Default drive, 1=A:,...,8=H:)
Output:  F$(0)= Default directory

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

The returned string will not include drive letter, nor "\" at the beginning
nor at the end; thus, the root directory will be represented as an empty
string.


* Function 47: Set the default directory (DOS 2)

Input:   F$(0) = Drive (optional) + directory
Output:  -

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

This function will not change the default drive, to do this you must use
```

```
function 44.


* Function 48: Obtain the RAM disk size (DOS 2)

Input:   -
Output:  P(0) = RAM disk size in 16K segments

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

To obtain the RAM disk size in K just do P(0)*16. A size of zero means that
the RAM disk dones not exist.


* Function 49: Create the RAM disk (DOS 2)

Input:   P(0) = Size required in 16K segments
Output:  P(0) = Size of the RAM disk created in 16K segments

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

Since by dafualt NestorBASIC allocates all the available free RAM segments
when it is installed, before this function is called function 80 should be
used to reduce the number of segmens allocated by NestorBASIC; otherwise, no
free segments for creating the RAM disk will be available (unless the
computer has more than 4M RAM).

If there are not P(0) free segments but it is possible to create a smaller
RAM disk, the function will do it and will not return error; P(0) will return
the size of the RAM disk created. But if there are no available segments at
all and the RAM disk has not been created, the function will return the
appropriate error code.


* Function 50: Obtain the attributes byte for a file (DOS 2)

Input:   P(0) = File number, or
     P(0) = 255 and F$(0) = File name
Output:  P(1) = Attributes byte:
        R + 2*H + 4*S + 8*V + 16*D + 32*A
        R = 1 -> Read only
        H = 1 -> Hidden
        S = 1 -> System
        V = 1 -> Volume label
        D = 1 -> Directory
        A = 1 -> Archive

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

If P(0)=255, the attributes byte for the file with the name specified in
F$(0) will be obtained (F$(0) can contain a drive letter and a pathname);
otherwise, the attributes byte for the file previously opened and with the
number P(0) assigned will be obtained.


* Function 51: Set the attributes byte for a file (DOS 2)

Input:   P(0) = File number, or
     P(0) = 255 and F$(0) = File name
     P(1) = New attributes byte:
        R + 2*H + 4*S + 8*V + 16*D + 32*A
```

```
        R = 1 -> Read only
        H = 1 -> Hidden
        S = 1 -> System
        V = 1 -> Volume label
        D = 1 -> Directory
        A = 1 -> Archive
Output:  P(1) = New attributes byte actually set for the file

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

If P(0)=255, the attributes byte for the file with the name specified in
F$(0) will be set (F$(0) can contain a drive letter and a pathname);
elotherwise, the attributes byte for the file opened and with the P(0) number
assigned will be set.

It is not possible to set the attributes byte of an opened file by specifying
its filename in F$(0); if you try to do this, an error will be generated. If
the file is open, you must specify its file number in P(0).

The only attributes than can be modified for a file are the system, hidden,
read only and archive; and for a directory, only the hidden. If you try to
change any other attribute, you will obtain an error. You can't change the
attributes for the "." and ".." entries.


* Function 52: Parse pathname (DOS 2)

Input:   F$(0) = Pathname to be parsed
     P(10)<>0 if the pathname refers to a volume label
Output:  F$(1) = Last item of the pathname
     P(8)  = Logical drive number (1=A:,...,8=H:)
     P(9)  = Position in the pathname string of the last item
         (0 if there is not last item)
     The following results set to -1 if the specified condition mets,
     otherwise set to 0:
     P(0)     The pathname contains characters other than the drive letter
     P(1)     A directory is specified in the pathname
     P(2)     A drive letter is specified in the pathname
     P(3)     A file name is specified in the pathname
     P(4)     A file extension is specified in the pathname
     P(5)     The last element is ambiguous
     P(6)     The last element is "." or ".."
     P(7)     The last element is ".."

This function is only available under DOS 2. Under DOS 1 will always return
the error code 1.

This function just treats the input string returning the specified results.
No disk accesses are done, and the default drive/directory are not modified.

If only a drive letter is specified, or if the last character of the string
is the directory separator character "\", then there is no "last item" on the
string; then P(9) will be 0 and F$(1) will be an empty string. Example: for
F$(0) = "A:\DIR\FILES\ONEFI.LE", the last element, returned in F$(1), will be
"ONEFI.LE", and P(9) will be 14.

If the pathname does not contain a drive letter, the current default drive
will be returned in P(2). Therefore, P(2) will never be zero.

If P(10)<>0 is specified, P(1) and P(4) to P(7) will always be zero.


10.6. FUNCTIONS FOR GRAPHIC COMPRESSION AND DECOMPRESSION
```

```
* Function 53: Compression of graphic data

Input:   P(0) = Source VRAM block
     P(1) = Source start address (VRAM)
     P(2) = First destination segment
     P(3) = Destination start address (RAM)
     P(4) = Size of the VRAM data to be compressed
     P(5) = Increase P(1) if <>0
     P(6) = Increase P(3) if <>0
Output:  P(7) = Size in RAM of the compressed data
     P(8) = Number of used RAM segments
     P(0):P(1) = P(0):P(1) + P(4) if P(5)<>0
     P(2):P(3) = RAM address next to the last one used, if P(6)<>0

Compression is done through consecutive segments: after writting compressed
data in the address &H3FFF of the segment S, the writting of the generated
data continues on address &H0000 of the segment S+1. If this situation is
done but the S+1 segment does not exists, an error 5 will be generated. This
function does not supports VRAM segments, nor the segment 255.

If the last used RAM address is the &H3FFF of the segment S and P(6)<>0, then
P(2) will be S+1 and P(3) will be 0 at the end. In such situation, if S was
the last available RAM segment, P(2) will be zero at the end.


* Function 54: Decompression of graphic data

Input:   P(0) = Destination VRAM block
     P(1) = Destination start address (VRAM)
     P(2) = First source segment
     P(3) = Source start address (RAM)
     P(5) = Increase P(1) if <>0
     P(6) = Increase P(3) if <>0
Output:  P(7) = Size in VRAM of the decompressed image
     P(8) = Number of used RAM segments
     P(0):P(1) = P(0):P(1) + P(7) if P(5)<>0
     P(2):P(3) = RAM address next to the last one used, if P(6)<>0

Decompression process is done through consecutive segments: after
decompressing data stored in the address &H3FFF of segment S, the reading of
the compressed data continues from the address &H0000 of segment S+1. If this
situation is done but segment S+1 does not exist, or if invalid data is found
in the compressed data, error 6 will be generated. This function does not
support VRAM segments, nor the segment 255.

If the last used RAM address is the &H3FFF of the segment S and P(6)<>0, then
P(2) will be S+1 and P(3) will be 0 at the end. In such situation, if S was
the last available RAM segment, P(2) will be zero at the end.


9.6. FUNCTIONS FOR THE EXECUTION OF BASIC PROGRAMS

* Function 55: Execution of a BASIC program stored in a segment (S4)

Input:   P(0) = Segment
     P(1) = Start address
Output:  -

VERY IMPORTANT: For using this function you must change the BASIC programs
begin address. See section 6 for more details.

This function executes the BASIC program stored in the specified segment.
The program must be stored beginning in the P(1) specified address of the
P(0) segment, with the following format:
```

```
+&H0000: Not used by NestorBASIC. You can use it, for example, for inserting
     an identification byte.
+&H0001: Last program address +1, in the range &H8000-&HBFFF; low byte.
+&H0002: Idem, high byte.
+&H0003: Here starts the program data.

All the exisiting numeric variables of the caller BASIC program will be
transferred to the new program. You can use this function from inside a
turbo-block, but in such case the only transferred variables will be the
integer variables defined outside of the turbo-block and passed to it with
CALL TURBO ON (variable, ..., array(), ...).

The string variables stored in the BASIC string area will also be
transferred, but the strings defined directly in the BASIC text reside inside
the own BASIC program data, and will be lost. For being sure that a string is
stored inside the BASIC string area, do the following operation: C$=C$+"". Of
course, the most practical way for managing it is to group all the string
variables into a string array (for example F$), then this operation can be
done with a loop.

This function will return the error code -1 if a non existing segment is
specified, or if the segment 255 is specified, and error code -2 if the BASIC
main RAM is not large enough to store the new program and the existing
variables. This situation may arise if the new program is larger than the
caller program, but if the lengths difference is not very big and the number
of variables is reasonably small you should not reach this situation.


* Function 56: Activation of a BASIC program stored in a segment (S4)

Input:   P(0) = Segment
     P(1) = Begin address
Output:  -

VERY IMPORTANT: For using this function you must change the BASIC programs
begin address. See section 6 for more details.

This function acts like function 55, with the following difference: once the
new program is placed in RAM an the variables are placed after it, the new
program is not executed, and the BASIC interpreter returns to the direct
mode. Except for this, the functionality is the same, as well as the error
conditions and error codes.


* Function 57: Save a BASIC program with a special header (S4)

Input:   P(0)  = Byte to be saved in the first header position
         (not used by NestorBASIC)
     F$(0) = Pathname + filename
Output:  P(1)  = -1 -> No error
          <> -1 -> A disk error occurred, and the file remains open
               with this number; look at the value returned by the
               USR command for the error code, as usual.

VERY IMPORTANT: For using this function you must change the BASIC programs
begin address. See section 6 for more details.

This function saves the active BASIC program in the specified file, with the
appropriate header for being executed or activated with functions 55 or 56
(see function 55 for details about this header).

The use of this function simplifies the conversion of BASIC normal programs
to the appropriate format required by function 55; this conversion is then
reduced to this:
```

```
load"prog.bas"                  'Load the BASIC program
F$(0)="prog.nba":?USR(57)       'Save the BASIC program with the header
                '(setting P(0) is optional)

Once saved in this way, loading and executing the program is easy:

1000 'Open prog.nba, read to segment S and close
1001 F$(0)="prog.nba":?USR(31):P(2)=S:P(3)=0:P(4)=&H4000:?USR(33):?USR(32)
...
10000 'Execution of the program (error treatment is optional)
10001 P(0)=S:P(1)=0:IF USR(55)=-1 THEN PRINT "ERROR: Non existing segment!"
      ELSE PRINT "ERROR: Out of memory!"
10002 END

Using function 56 instead of function 55 you can load programs saved with the
header if you don't have a copy in normal BASIC format (LOADable).

Of course, you can also use the normal disk access functions to save the
program, but in this case you must create the header manually:

POKE &H8000,P(0) 'Data not used by NestorBASIC
POKE &H8001,PEEK(&HF6C2)
POKE &H8002,PEEK(&HF6C3)
'Save from &H8000 to PEEK(&H8001)+256*PEEK(&H8002) to the file

The error codes returned by this function are the same as the ones returned
by the disk access functions (see section 4). If a disk error caused the file
to remain open, P(1) will contain the associated file number; otherwise, P(1)
will be -1.


10.7. MISCELLAENOUS FUNCTIONS

* Function 58: Execution of a machine code routine placed on BIOS, SUB-BIOS,
               BASIC main RAM or system work area

Input:   P(0) = 0 to execute a routine placed on BIOS,
                 BASIC main RAM or system work area
         <> 0 to execute a SUB-BIOS routine
     P(1) = Begin address of the routine
     P(2) to P(11) = Input registers for the routine:
             P(2) = AF     P(8) = AF'
             P(3) = BC     P(9) = BC'
             P(4) = DE     P(10)= DE'
             P(5) = HL     P(11)= HL'
             P(6) = IX
             P(7) = IY
Output:  P(2) to P(12) = Output registers of the routine,
             with the same assignment as in the input,
             plus the following ones:
             P(12) = A
             P(13) = Cy flag (-1 if set, 0 otherwise)
             P(14) = Z flag (-1 if set, 0 otherwise)

With this function you can execute machine code routines placed on BIOS,
SUB-BIOS, BASIC main RAM or system work area, having defined previously the
registers input state using the array P; after the routine execution, the
registers output contents will also be set in array P in the same way.

If P(0)=0 at input, this function will execute directly the machine code
routine placed at the address specified in P(1), without doing any slot or
segment switching. This means that this way you can execute BIOS routines (if
the routine address is in the range 0-&H3FFF), routines loaded in BASIC RAM
by user (for example with BLOAD), and routines placed in page 3 system work
```

```
area (for example the extended BIOS hook, at address &HFFCA). Note however
that when this function is executed, NestorBASIC segment will be switched on
page 1. This implies that it is not possible to directly execute ruotines
placed at the BASIC interpreter ROM. To execute such routines, use the BIOS
routine CALBAS (&H0159), which accepts in IX the address of the BASIC
interpreter routine to be called.

If P(0)<>0 at input, the SUB-BIOS routine whose address is specified in P(1)
will be executed; in this case, an error code -1 will be returned if an
address higher than &H3FFF is specified. To perform the SUB-BIOS call, the
BIOS routine EXTROM is used, so the input contents of the IX register
[variable P(6)] will be lost.

The value returned in P(12) will be the same as the value returned in the
high byte of P(2). Likewise, the values returned in P(13) and P(14) will
indicate the state of bits 0 and 6, respectively, of the low byte of P(2).
Since normally one wants to retrieve the value returned in A register by
itself or the output value of the flags Cy and Z by themselves, rather than
the complete AF pair, using P(12) to P(14) will be more convenient than using
P(2) when looking at the output registers. However, when establishing the
input value for A and the flags you need to use P(2), you can't use P(12).
You can calculate the value to set up in P(2) with the following formula:
P(2) = Cy + 2*N + 4*P/V + 16*H + 64*Z + 128*M + 256*A


* Function 59: User machine code routine execution
           (machine code routine stored in a RAM segment)

Input:   P(0) = Segment in which the routine is stored
     P(1) = Start address of the routine
     P(2) to P(11) = Input registers for the routine:
             P(2) = AF     P(8) = AF'
             P(3) = BC     P(9) = BC'
             P(4) = DE     P(10)= DE'
             P(5) = HL     P(11)= HL'
             P(6) = IX
             P(7) = IY
Output:  P(2) to P(12) = Output registers of the routine,
             with the same assignment as in the input,
             plus the following ones:
             P(12) = A
             P(13) = Cy flag (-1 if set, 0 otherwise)
             P(14) = Z flag (-1 if set, 0 otherwise)

Using this function you can execute a machine code routine previously
loaded in a RAM segment, having defined the registers input state using the
array P; after the execution, the registers output contents will also be set
in the array P in the same way.

The specified segment will be switched in page 2 for the execution of the
routine, therefore the routine must have been assembled in the addresses
range &H8000-&HBFFF. When the routine is executed, BIOS is connected and
available in page 0, and NestorBASIC segment is available in page 1. This
state must be restored by the routine itself before returning, if
segment/slot switchings are done.

Some of the internal NestorBASIC routines and variables (for RAM and VRAM
management, and for obtaining information about the interrupt process,
amongst others) can be used by the user machine code routines. In appendix 2
you can find the list and detailed description of these routines and
variables.

See also the function 58 description for considerations about the output
values of P(12) to P(14).
```

```
This function does not support VRAM segments nor the segment 255, and will
return the error -1 if a non existing segment is specified in P(0).


* Function 60: Print a string in graphic mode

Input:   F$(0) = String
Output:  -

This function works only in SCREEN 5 to 11 modes, and is equivalent to the
PRINT#1,F$(0) command, having executed previously OPEN"GRP:"AS#1. The
difference is that this function is TurboBASIC compatible. If executed in
other graphic modes this function will do nothing, but will never return an
error.

The maximum length of the string is 80 characters; if F$(0) is larger, only
the first 80 characters will be printed. The string can't contain any
character 0, because this code is interpreted as the end of the string.


* Function 61: Store a string in a segment

Input:   F$(0) = String
     P(0)  = Segment
     P(1)  = Begin address
     P(2)  = Increase P(1) if <>0
Output:  P(1)  = P(1) + LEN(F$(0)) + 1 if P(2)<>0

This function copies the string F$(0) to the specified address of the
specified segment. The string is stored with a character 0 at the end, so
LEN(F$(0))+1 bytes will be occuped.

The maximum length of the string is 80 characters; if F$(0) is largest, only
the first 80 characters will be copied. The string can't contain any
character 0, because this code is interpreted as the end of the string.

This function will return the error -1 if a non exisiting segment is
specified in P(0).


* Function 62: Restore a string stored in a segment

Input:   P(0)  = Segment
     P(1)  = Begin address
     P(2)  = Increase P(1) if <>0
Output:  F$(1) = String
     P(1)  = P(1) + LEN(F$(1)) + 1 if P(2)<>0

This function sets in F$(1) the string stored from the specified address of
the specified segment. The string is considered finished when a 0 character
is found, or when 80 characters have been copied to F$(1).

If this function is executed from normal BASIC, the string can't contain the
character 255, because if found, this character will be converted to the
character 34 (quotation marks); this is due to the strings assignation method
of BASIC, that requires a special treatment for the quotation marks. This
does not happen when using TurboBASIC.

This function will return the error -1 if a non exisiting segment is
specified in P(0).


* Function 63: Initialization of the SCREEN 0 blink mode
```

```
Input:   P(0) = Foreground color of the blinking text
     P(1) = Background color of the blinking text
     P(2) = Time ON (0 to 15)
     P(3) = Time OFF (0 to 15)
        IF P(2) and P(3) are both 0, this function
        will only clear the blink zone in VRAM
Output:  -

This function will initialize the colors and times for the SCREEN 0 blink
mode, and will clear the VRAM blink zone. If P(2) and P(3) are both zero,
only this last action will be done.

This function can only be used in SCREEN 0 mode with at least 41 columns;
otherwise, nothing will be done, and the error -1 will be returned.

Actions done by this function can alse be done separately from BASIC, in the
following way:

- Colors setting:

VDP(13) = Foreground color+ 16*Background color

- Times setting:

VDP(14) = Time ON + 16*Time OFF

- Clear VRAM:

Fill with zeros the 256 bytes VRAM zone beginning at &H0A00

System variables FORCLR (&HF3E9) and BAKCLR (&HF3EA) contain respectively the
current foreground color and background color previously set with the COLOR
statement. Thus, for obtain an inverse video effect just do:

P(0)=PEEK(&HF3EA):P(1)=PEEK(&HF3E9):P(2)=15:P(3)=0:?USR(63)

or

VDP(13)=PEEK(&HF3EA)+16*PEEK(&HF3E9):VDP(14)=&HF0


* Function 64: Make or erase a blinking characters block

Input:   P(0) = 0 to erase block
         <> 0 to create block
     P(1) = Begin X coordinate (column) (0 to 79)
     P(2) = Begin Y coordinate (line) (0 to 27)
     P(3) = X length (columns)
     P(4) = Y length (lines)
     P(5) = Increase P(1) if <>0
     P(6) = Increase P(2) if <>0
Output:  P(1) = P(1) + P(3) if P(5)<>0
     P(2) = P(2) + P(4) if P(6)<>0

This function makes or erases a blinking characters block with the specified
dimensions, in the specified coordinates. It is recommended to execute
function 63 before use this function.

This function can only be used in SCREEN 0 mode with at least 41 columns;
otherwise nothing will be done, but no error will be returned. If P(1)>79
and/or P(2)>27, an error -1 will be returned.

This function does not check if the block surpasses the screen limits. If
the block surpasses column 79, it will continue at the column 0 of the
```

```
next line. But if a part of the block surpasses line 27, this part will
not be shown.


* Function 65: Obtain information about interrupts

Input:   -
Output:  P(0) = -1 if any interrupt process is active
               (user defined interrupt, PSG sound effect or music replay)
     P(1) = -1 if an user defined interrupt is active
     P(2) = User defined interrupt segment
     P(3) = User defined interrupt begin address
     P(4) = -1 if a PSG sound effect is being replayed
     P(5) = -1 if a music is being replayed

P(2) and P(3) will return the user interrupt definition, even if it is
stopped. If no user interrupt has never been defined, both results will be
zero.

To obtain more information about the active sound effect use function 67.
For the music, use function 72.


* Function 66: Define or stop an user defined interrupt routine

Input:   P(0) =  0 to stop the current user interrupt
         1 to define and activate an user interrupt
        -1 to reverse the current status of the user interrupt
           (actived <--> stopped)
     P(1) = User interrupt segment (ignored if P(0)<>1)
     P(2) = User interrupt begin address (ignored if P(0)<>1)
Output:  -

This function defines or stops an user defined interrupt routine, that is, a
machine code routine that will be executed in each timer interrupt, 50 or 60
times per second. The specified segment will be switched in page 2 for the
execution of the routine, therefore the routine must have been assembled in
the addresses range &H8000-&HBFFF.

If P(0)=1, the interrupt is defined by P(1) and P(2), and is activated (from
this moment, it is executed to 50 or 60 Hz). If P(0)=0, the interrupt is
stopped (it is no longer executed), but its definition is not changed. If
P(0)=-1, the current status is reversed (actived to stopped and vice-versa),
and its definition is not changed.

Interrupt segment is switched in page 2 for execution, so the code must be
assembled in the address range &H8000-&HBFFF. BIOS is switched and available
in page 0, and NestorBASIC segment is available in page 1. This state must be
restored by the routine itself before returning, if segment/slot switchings
are done. Also, the interrupts will be disabled and must be in this state
during the routine execution. No register preservations are needed, because
NestorBASIC handles this.

Some of the internal NestorBASIC routines and variables (for RAM and VRAM
management, and for obtaining information about the interrupt process,
amongst others) can be used by the user interrupt routines. In appendix 2 you
will found the list and detailed description of these routines and variables.

You can't define an user interrupt routine in a VRAM segment, nor in the
segment 255. If you specify a non existing or invalid segment in P(1) the
function will return an error -1. If you specify a parameter other than 0, 1
or -1 in P(0), an error 7 will be returned.
```

```
10.8. PSG SOUND EFFECTS

* Function 67: Obtain information about the PSG sound effects

Input:   P(0) = New maximum volume (0 to 15, -1 to leave it unchanged)
Output:  P(1) = -1 if a sound effect is being replayed
     P(2) = Number of the SFX being replayed, or the last one replayed
     P(3) = Priority of the SFX being replayed, or the last one replayed
     P(4) = Segment of the sound effects set
     P(5) = Start address of the sound effects set
     P(6) = Highest defined sound effect number
     P(7) = Maximum volume

The results returned by this function will only be valid if the sound effects
set has been initialized with function 68. The sound effects set must have
been created with the editor SEE v3.xx, developped by Fuzzy Logic.

This function never returns error. If the value specified in P(0) is greater
than 15, it will be interpreted as 15.


* Function 68: Initialization of a PSG sound effects set

Input:   P(0) = Segment
     P(1) = Address
Output:  -

This function initializes the sound effects set stored in the specified
segment and address, leaving it ready for use with function 69.

The sound effects set must have been created with the editor SEE v3.xx, by
Fuzzy Logic; if the set format is invalid, the function will return the error
code 8.

You can't use a sound effects set stored in a VRAM segment, nor in the
segment 255. If you specify a non existing segment in P(0), an error -1 will
be generated.


* Function 69: Play a PSG sound effect

Input:   P(0) = Effect number
     P(1) = Priority (0: low, <>0: high)
Output:  -

This function begins the replay of the specified sound effect; the sound
effects set must have been created with the editor SEE v3.xx, by Fuzzy Logic.
If the sound effects set has not been initialized with function 68, the
result is unpredictable.

P(1) handles the priority in the case of executing this function when
another sound effect is being replayed, in the following way:

- If the SFX being replayed and the new one have the same priority, the SFX
being replayed is stopped, and the new one is played.
- If the SFX being replayed has low priority and the new one has high
priority, same as above occurs.
- If the SFX being replayed has high priority and the new one has low
priority, the old SFX stills being replayed, and the function returns the
error code 11.

If the sound effect specified in P(0) is not defined (the first track is left
to "OFF" in the editor), the function will return the error code 9. If the
effect does not exist (its number is higher than the maximum effect number),
```

```
the function will return the error code 10.


* Function 70: Stop PSG sound effect replay

Input:   -
Output:  -

This function stops the PSG sound effect being replayed, and silences the
PSG. It never returns error.


10.9. FUNCTIONS FOR MOONBLASTER MUSIC REPLAY

* Function 71: Load and initialization, or uninstall,
           of the Moonblaster replayer (S4)

Input:   P(0)  = Replayer to be loaded:
         0: Moonblaster 1.4
         1: Moonblaster Wave 1.05
         3: Autodetect:
            Load Moonblaster Wave replayer if MoonSound is present,
            otherwise load Moonblaster 1.4 replayer.
                -1: Uninstall the loaded replayer
Output:  P(0)  = Replayer loaded (0, 1 or -1, as above).
Output:  P(1)  = -1 -> No error
          <> -1 -> A disk error occurred, and NBASIC.BIN remains open
                with this number; look at the value returned by the
                USR command for the error code, as usual.

This function loads the specified Moonblaster replayer in segment 5, leaving
it ready for use. Also, searches for all the existing sound chips in the
system, and leaves all the found chips enabled (see function 73 for more
details about the sound chips activation).

The replayer can only be installed if the segment 5 exists and belongs to the
primary mapper; otherwise, the function will return the error -1. If a disk
error occurs, the function will return the appropriate error code (see
section 4). If NBASIC.BIN remains opened due to a disk error, its associated
file number will be returned in P(1); otherwise, P(1) will be -1.

NBASIC.BIN file contains two versions of the Moonblaster Wave replayer: one
is for MSX2/2+ and Turbo-R in Z80 mode, and the other is for Turbo-R in R800
mode. If the computer is a Turbo-R, NestorBASIC decides the version to be
loaded according to the processor switched when function 71 is called. Note
that if a processor change is performed, this function must be called again
in order to load the appropriate replayer: Z80 version does not work in R800
mode, and R800 version makes the system slower in Z80 mode.

To uninstall the replayer, use this function with P(0)=-1 at input; you don't
need to previously stop any music being replayed, since the function itsef
will do this task. After uninstallation (which just consists on erasing the
sting "NestorPlayer" placed at the start of segment 5), segment 5 becomes
available again, just like any other segment. If you try to uninstall the
replayer when no replayer is actually loaded, the function will do nothing,
and no error will be returned.

You can load, uninstall, and load again the Moonblaster replayer as many
times as necessary within a program's lifetime. If there is a replayer loaded
and you want to load another one, it is convenient to first uninstall the
loaded replayer.


* Function 72: Obtain information about the music being replayed
```

```
Input:   -
Output:  P(0) = -1 if a music is being replayed or is paused
     P(1) = -1 if a Moonblaster 1.4 music is being replayed or paused
     P(2) = -1 if a Moonblaster Wave music is being replayed or paused
     P(3) = 0
     P(4) = -1 if the music is paused
     P(5) = Segment of the music being replayed,
        or segment of the last replayed music
     P(6) = Begin address of the music being replayed,
        or begin address of the last replayed music
     P(7) = Current position
     P(8) = Current step (0 to 15)
     P(9) = -1 if a MSX-MUSIC was detected
     P(10)= -1 if a MSX-AUDIO was detected
     P(11)= -1 if a OPL4 was detected
     P(12)= -1 if the replayer is initialized
     P(13)= Replayer installed (only if P(12)=-1):
        0: Moonblaster 1.4
        1: Moonblaster Wave 1.05
     F$(0)= Songname (empty string if no music is being replayed)
     F$(1)= Samplekit or wavekit (empty string if no music is being
        replayed)

If no Moonblaster replayer is loaded, P(12) will be zero the other results
will not be set (except for P(9) to P(11); the function will not return any
error. Use function 71 to initialize the replayer.

If no music is being replayed, P(5) and P(6) will return the position and the
step in which the last replayed music was stopped, and F$(0) and F$(1) will
be empty strings. Otherwise, F$(0) will return the songname (always 40
characters for Moonblaster 1.4 musics, 50 for Moonblaster Wave musics) and
F$(1) will return the samplekit or wavekit that was loaded in Moonblaster
when the music was saved, in capital letters and without extension ("NONE" if
there was no samplekit loaded).

P(9) to P(11) return information about the detected sound chips, even when no
replayer is loaded. To obtain information about the enabled sound chips, use
function 73.


* Function 73: Activation and deactivation of the sound chips

Input:   P(0) = 0 -> Request for information only
         1 -> Enable or disable chips according to P(1) to P(3)
         2 -> Enable all the found sound chips
     P(1) = 0 -> Don't modify MSX-MUSIC status
        1 -> Disable MSX-MUSIC
        2 -> Enable MSX-MUSIC
           -1 -> Reverse MSX-MUSIC status
     P(2) = 0 -> Don't modify MSX-AUDIO status
        1 -> Disable MSX-AUDIO
        2 -> Enable MSX-AUDIO
           -1 -> Reverse MSX-AUDIO status
     P(3) = 0 -> Don't modify OPL4 status
        1 -> Disable OPL4
        2 -> Enable OPL4
           -1 -> Reverse OPL4 status
Output:  P(4) = 0 -> MSX-MUSIC not found
         1 -> MSX-MUSIC found but disabled
         2 -> MSX-MUSIC enabled
     P(5) = 0 -> MSX-AUDIO not found
        1 -> MSX-AUDIO found but disabled
        2 -> MSX-AUDIO enabled
     P(6) = 0 -> OPL4 not found
```

```
        1 -> OPL4 found but disabled
        2 -> OPL4 enabled

This function allows to disable and enable MSX-MUSIC, MSX-AUDIO and OPL4,
which in such state will not be used, even if they are found in the system.
This feature can be useful, for example, for earing only the MSX-AUDIO part
of a music in a computer with internal MSX-MUSIC.

If P(0)=0, the function will only return information about the enabled sound
chips in P(4) to P(5), and will not change the current status of the chips.
If P(0)=2, all the found chips will be enabled (this is the default state
when the replayer is initialized). To obtain information about the found
sound chips use function 72.

If P(0)=1, the different sound chips will be enabled or disabled according to
the values passed in P(1) to P(3); attempts to enable chips that are not
present will be ignored. If MSX-AUDIO and MSX-MUSIC are both disabled
(Moonblaster 1.4 replayer) or if OPL4 is disabled (Moonblaster wave
replayer), function 74 (start the playing of a music) will do nothing but
will not return error.

The changes done with this function will have not effect until the currently
being replayed music is stopped (function 75 or 77) and restarted (function
74). If no music is being replayed when you use this function, just start the
music and you will note the changes.

This function will return the error code 7 if any of the input parameters in
invalid (but if P(0)<>1 you don't need to set P(1) to P(3)), and error code
12 if the replayer is not installed.


* Function 74: Start playing a Moonblaster music

Input:   P(0) = Segment in which the music is stored
     P(1) = Music begin address
Output:  -

This function begins the replay of the specified Moonblaster music. If the
specified segment does not exist, refers to VRAM or is the 255, the error
code -1 will be returned. If the replayer has not been initialized, the error
code 12 will be returned (use function 71 to initialize the music replayer).
Moonblter Wave musics can be stored through consecutive segments; see section
8.2 for more details.

In the case of Moonblaster 1.4 music, this function examines the first byte
of the music in the specified address; if this byte is &HFF it means that the
music was saved in EDIT format, and therefore can't be replayed; in such case
the error code 13 will be returned. Otherwise, NestorBASIC assumes that in
the specified address a Moonblaster music saved in USER format begins, and
the replay starts without further checkings. If the specified RAM area does
not contain a Moonblaster music, the result is unpredictable.

Moonblaster Wave rapleyer has not these restrictions: it can detect if data
in the specified address is really a Moonblaster music, and if it is saved in
USER mode. If not, error 13 is returned.

If MSX-MUSIC and MSX-AUDIO were both disabled (Moonblaster 1.4 music) or OPL4
was disabled (Moonblaster Wave music) with function 73, this function will do
nothing, but no error will be returned. If another music is already being
replayed, an error code 14 will be returned.


* Function 75: Stop the music replay
```

```
Input:   -
Output:  -

This function stops the replay of the being replayed music and silences the
sound chips. If no music is being replayed or if no music replayer was
initialized, the function will do nothing. It never returns error.


* Function 76: Pause and continue the music

Input:   P(0) = 0 -> Pause music
        1 -> Continue music
           -1 -> Reverse current status (replay <--> pause)
Output:  -

This function pauses or continues the replay of a Moonblaster music. If no
music is being replayed or paused, or if the replayer was not initialized,
the function will do nothing, but no error will be returned. If the specified
parameter is invalid, an error code 7 will be returned.


* Function 77: Fade out the music

Input:   P(0) = 0      -> Request for information only
        -1     -> Pause fade out
        1..254 -> Start/continue fade out with the specified delay
Output:  P(1) = -1 if a music is being faded
     P(2) = Current fade delay (-1 if the fade is paused)

This function begins to fade out the music being replayed. The delay means
the number of clock cycles (of 1/50 or 1/60 seconds each) between fade
steps, so the smallest is the delay, the faster is the fade. Once the fade is
completed (the music main volume decreases to zero), the music will be
automatically stopped: you don't need to stop it by yourself with function
75.

If P(0)=-1 the fade will be paused, that is, the music will remain being
normally replayed with the reached volume. To continue the fade, just execute
again this function, specifying a new delay in P(0).

If P(0)=0, this function will only set P(1) and P(2) with the specified
information. If no music is being replayed, or if the replayer was not
initialized, this function will do nothing. It never returns error.


* Function 78: Load a Music Module samplekit (S4)

Input:   P(0) = File number
Output:  P(7) = Number of readed bytes

This function loads from the specified opened file a Music Module samplekit,
which must be saved in the Moonblaster format: 56 header bytes, which will be
copied into the appropriate zone of the replayer; and 32K of samples, which
will be transferred to the Music Module sample RAM.

Data will be read from a previously opened file. This feature allows you to
concatenate the samplekit with other useful data in a single data file: in
such case, just move the file pointer to the appropriate point, and execute
this function. However, in the most common case you will need to load a MBK
file generated with Moonblaster. In such case, you can load the samplekit by
just doing the following:

F$(0)="sampkit.mbk":?USR(31):?USR(78):?USR(32)
```

```
If a disk error occurs, the function will return the appropriate error code,
and you can look for the number of bytes that have been read from the file in
P(7). If this number is greatert than 16K, only the first 16K of the
samplekit will have been transferred to the sample RAM. Otherwise, no data
will have been transferred to the sample RAM.

If the replayer was not initialized or if no Music Module is found, this
functon will do nothing, but will not return error. If the whole samplekit is
successfully loaded, P(7) will not return 32824 but -32712, due to the BASIC
integer variables range (-32768 to 32767).


* Function 79: Load a MoonSound wavekit (S4)

Input:   P(0) = File number
Output:  P(6):P(7) = Number of readed bytes

This function loads from the specified open file a Moonsound wavekit, which
must be saved in USER format (otherwise, an error 15 is returned).

Data will be read from a previously opened file. This feature allows you to
concatenate the wavekit with other useful data in a single data file: in
such case, just move the file pointer to the appropriate point, and execute
this function. However, in the most common case you will need to load a MWK
file generated with Moonblaster. In such case, you can load the wavekit by
just doing the following:

F$(0)="wavkit.mwk":?USR(31):?USR(79):?USR(32)

If a disk error occurs, the function will return the appropriate error code,
and you can look for the number of bytes that have been readed from the file
in P(6):P(7). In this case MoonSound sample RAM can have been modified, but
work area will not have been updated, so a partial wavekit load can't be
supposed.

If the Moonblaster Wave replayer was not initialized or if no MoonSound is
found, this functon will do nothing, but will not return error. If the whole
samplekit is successfully loaded, P(6):P(7) will contain the wavekit size
(same as the file size if you just load a .MWK file).


10.10. FUNCTIONS FOR CONTROLLING THE USAGE OF SEGMENTS

* Function 80: Obtain and set the number of segments
           allocated for NestorBASIC

Input:   P(0) = Number of segments to be allocated for NestorBASIC
        (0: Do not modify, only obtain the number of segments
         curently allocated)
        This number includes NestorBASIC, TurboBASIC
        and BASIC main RAM segments
Output:  P(0) = Number of segments allocated for NestorBASIC
        after the function execution
     P(1) = Maximum number of segments that can be allocated
        for NestorBASIC

This function works both in DOS 1 and DOS 2, however it is intended for being
used in DOS 2.

Under DOS 2, when NestorBASIC is installed it allocates for itself all the
free RAM segments available in the system, up to 247. This implies that no
free segments remain available for other resident programs that also perform
segment allocation, like the DOS 2 RAM disk, NestorMan or InterNestor Suite.
To solve this problem, use this function to let NestorBASIC know how many
```

```
segments we really need; the other segments will be freed and therefore will
become available for other programs. For example, if you will use a music
replayer (which will be installed on segment 5) and you need only one extra
segment for data, then you should execute this function with P(0)=7; the
extra data segment will have number 6.

If a number of segments lower than 6 is specified in P(0), NestorBASIC will
allocate 5 segments (see section 2), or 6 if a music replayer has been
installed. If a number of segments higher than the total number of available
segments is specified, then as much segments as possible will be allocated
(up to 247), and P(0) will return the number of segments that have been
actually allocated. Whatever finally happens, this function will never return
error.

Under DOS 1, this function just modifies some NestorBASIC internal variables;
the value returned in P(1) is fixed and it is computed at NestorBASIC
installation time. Under DOS 2, contrarywise, actual segment allocations or
freeings are performed using DOS 2 mapper support routines; in this case, the
value returned in P(1) is re-computed on each execution of the function, and
depends on how many segments have been allocated by other resident programs.


10.11. FUNCTIONS FOR NESTORMAN AND INTERNESTOR SUITE/LITE INTERACTION

* Function 81: Obtain information about NestorMan
           and InterNestor Suite availability

Input:   -
Output:  P(0) = 0 if NestorMan is not installed
                1 if NestorMan is installed
                3 if NestorMan and InterNestor Suite are installed
         P(1) = NestorMan segment of InterNestor Suite level 1 module
         P(2) = NestorMan segment of InterNestor Suite level 2 module
         P(3) = NestorMan segment of InterNestor Suite level 3 module
         P(4) = NestorMan segment of InterNestor Suite level 4 module
         P(5) = NestorMan segment number of NestorBASIC segment 4

The value returned in P(5) will be valid only if P(0) is 1 or 3 at output.
This value is necessary if a data transfer between a NestorMan segment and a
NestorBASIC segment will be performed using segment 4 (common to both
programs) as an intermediate buffer, as described in section 9.2.

The values returned in P(1) to P(4) will be valid only if P(0) is 3 at
output. It is not necessary to know the InterNestor Suite modules location in
order to execute their routines (using function 85), but it is necessary in
order to read and write the modules configuration constants and variables, as
detailed in section 9.3.


* Function 82: Execute a NestorMan function

Input:   P(0) = Function number
     P(2) to P(11) = Input registers for the function:
             P(2) = AF     P(8) = AF'
             P(3) = BC     P(9) = BC'
             P(4) = DE     P(10)= DE'
             P(5) = HL     P(11)= HL'
             P(6) = IX
             P(7) = IY
Output:  P(2) to P(12) = Output registers of the function,
             with the same assignment as in the input,
             plus the following ones:
             P(12) = A
             P(13) = Cy flag (-1 if set, 0 otherwise)
```

```
             P(14) = Z flag (-1 if set, 0 otherwise)

This function executes the NestorMan function specified in P(0) using then
indirect calling method, that is, using the extended BIOS hook. It is
therefore equivalent to the execution of function 58 having established
previously P(1)=&HFFCA, P(3)=Function+256*B, and P(4)=&H2202.

An error code -1 will be returned by this function if NestorMan is not
installed. Note however that it does not check if the specified function does
really exist in NestorMan.

See also the function 58 description for considerations about the output
values of P(12) to P(14).

Section 9 describes some procedures for exachanging data between NestorMan
and NestorBASIC.


* Function 83: Data block transfer from a NestorMan segment
           to a NestorBASIC segment

Input:   P(0) = NestorMan source segment
     P(1) = Source start address
     P(2) = NestorBASIC destination segment
     P(3) = Destination start address
     P(4) = Data length
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(1) = P(1) + P(4) if P(5)<>0
     P(3) = P(3) + P(4) if P(6)<>0

This function is identical to function 10, except that the value passed on
P(0) refers to a NestorMan segment. As in the case of function 10, P(3)+P(4)
must be lower than &H4000, a VRAM segment or the segment 255 may be specified
in P(2), and an error -1 is returned if any of the specified segments does
not exist.

This function has one limitation: it is not possible to use it specifying
segment 1 as the NestorBASIC destination segment. However, this will normally
not be a problem, because segment 1 contains the TurboBASIC compiler, and
usually one does not want to modify its contents. If anyway a transfer to
segment 1 is needed, you must first copy the data to any other segment as an
itermediate step, for example segment 4.


* Function 84: Data block transfer from a NestorBASIC segment
           to a NestorMan segment

Input:   P(0) = NestorBASIC source segment
     P(1) = Source start address
     P(2) = NestorMan destination segment
     P(3) = Destination start address
     P(4) = Data length
     P(5)<> 0 -> Autoincrement of P(1)
     P(6)<> 0 -> Autoincrement of P(3)
Output:  P(1) = P(1) + P(4) if P(5)<>0
     P(3) = P(3) + P(4) if P(6)<>0

This function is identical to function 10, except that the value passed on
P(2) refers to a NestorMan segment. As in the case of function 10, P(3)+P(4)
must be lower than &H4000, a VRAM segment or the segment 255 may be specified
in P(0), and an error -1 is returned if any of the specified segments does
not exist.
```

```
This function has one limitation: it is not possible to use it specifying
segment 1 as the NestorBASIC source segment. However, this will normally not
be a problem, because segment 1 contains the TurboBASIC compiler, and usually
one does not want to read its contents. If anyway a transfer from segment 1
is needed, you must first copy the data to any other segment as an
itermediate step, for example segment 4.


* Function 85: InterNestor Suite routine execution

Input:   P(0) = Module number, 1 to 4
     P(1) = Routine address
     P(2) to P(11) = Input registers for the routine:
             P(2) = AF     P(8) = AF'
             P(3) = BC     P(9) = BC'
             P(4) = DE     P(10)= DE'
             P(5) = HL     P(11)= HL'
             P(6) = IX
             P(7) = IY
Output:  P(2) to P(12) = Output registers of the routine,
             with the same assignment as in the input,
             plus the following ones:
             P(12) = A
             P(13) = Cy flag (-1 if set, 0 otherwise)
             P(14) = Z flag (-1 if set, 0 otherwise)

Using this function it is possible to execute any routine of any of the
InterNestor Suite modules. To read or write the modules configuration
constants and variables it is necessary to first obtain the NestorMan segment
numbers of the modules using function 81, and then use any of the techniques
described in section 9.2 for exchanging data between NetorBASIC and
NestorMan.

This function will return an error -1 if InterNestor Suite is not installed,
or if an invalid module number is passed on P(0).

See also the function 58 description for considerations about the output
values of P(12) to P(14).

Section 9.3 mentions some considerations about the ensemble use of
NestorBASIC and InterNestor Suite.


* Function 86: InterNestor Lite routine execution

Input:   P(1) = Routine address
     P(2) to P(11) = Input registers for the routine:
             P(2) = AF     P(8) = AF'
             P(3) = BC     P(9) = BC'
             P(4) = DE     P(10)= DE'
             P(5) = HL     P(11)= HL'
             P(6) = IX
             P(7) = IY
Output:  P(2) to P(12) = Output registers of the routine,
             with the same assignment as in the input,
             plus the following ones:
             P(12) = A
             P(13) = Cy flag (-1 if set, 0 otherwise)
             P(14) = Z flag (-1 if set, 0 otherwise)

Using this function it is possible to execute any routine of InterNestor
Lite. P(1) must contain the address of any of the InterNestor Lite routines
listed in its programmer's manual, otherwise the results of executing this
function are unpredictable.
```

```
This function will return an error -1 if InterNestor Lite is not installed.
To know in advance whether InterNestor Lite is installed or not, use the
following code, which performs the detection method described in section 9.4:

P(0)=0: P(1)=&HFFCA: P(2)=0: P(4)=&H2203: E=USR(58):
IF P(12)=0 THEN ... (not installed)

In section 9.4 there is a comment about the use of the InterNestor Lite
routines that perform data exchange with TPA from NestorBASIC. See also the
function 58 description for considerations about the output values of P(12)
to P(14).

The following sample program shows the installed InterNestor Lite version
number.

10 BLOAD"nbasic.bin",R
20 IF P(0)<5 AND P(0)<>3 THEN PRINT "Error "+P(0):END
30 '* Check that INL is installed
40 P(0)=0: P(1)=&HFFCA: P(2)=0: P(4)=&H2203: E=USR(58): IF P(12)=0 THEN
   PRINT "InterNestor Lite is not installed.":GOTO 100
50 '* The VERS_PAUSE routine(&H402A) returns the INL version in C.D.E
      and the implementation number in B, it must be called with A=0
60 P(1)=&H402A:P(2)=0:E=USR(86)
70 X=P(3):GOSUB 1000:RB=X '* B register
80 X=P(4):GOSUB 1000:RD=X '* D register
90 PRINT "InterNestor Lite version " ; P(3) AND &HFF ; "."
   ; RD ; "." ; P(4) AND &HFF ; ", implementation "
   ; RB ; " is installed."
100 '* Uninstalls NestorBASIC and finishes
110 P(0)=1:E=USR(0):END
1000 '* This subroutine extracts the high byte of X
1010 X=((X AND &HFF00)/256) AND &HFF
1020 RETURN

The program TCPCON-L.BAS, supplied with NestorBASIC, is a more complex
example that illustrates the ensemble use of NestorBASIC and InterNestor
Lite.


APPENDIX 1 - NESTORBASIC FUNCTIONS LIST

In this appendix, all the NestorBASIC functions are listed sorted by number.
Functions that use segment 4 have a "(S4)" mark after their name. These
functions are: 0, 26-28, 30, 33-41, 55-57, 71, 78, and 79.

--- General functions

* Function 0: NestorBASIC uninstallation (S4)
* Function 1: General information about NestorBASIC and about a segment

--- RAM access functions

* Function 2: Read a byte from a segment
* Function 3: Read a byte from a segment with address autoincrement
* Function 4: Read an integer (2 bytes) from a segment
* Function 5: Read an integr (2 bytes) from a segment
          with address autoincrement
* Function 6: Write a byte to a segment
* Function 7: Write a byte to a segment with address autoincrement
* Function 8: Write an integer (2 bytes) to a segment
* Function 9: Write an integer (2 bytes) to a segment
          with address autoincrement
* Function 10: Data block transfer between segments
* Function 11: Fill a RAM zone with a byte
```

```
* Function 12: Fill a RAM zone with a byte with address autoincrement

--- VRAM access functions

* Function 13: Read a byte from VRAM
* Function 14: Read a byte from VRAM with address autoincrement
* Function 15: Read an integer (2 bytes) from VRAM
* Function 16: Read an integer (2 bytes) from VRAM
           with address autoincrement
* Function 17: Write a byte to VRAM
* Function 18: Write a byte to VRAM with address autoincrement
* Function 19: Write an integer (2 bytes) to VRAM
* Function 20: Write an integer (2 bytes) to VRAM
           with address autoincrement
* Function 21: Data block transfer from VRAM to RAM
* Function 22: Data block transfer from RAM to VRAM
* Function 23: Data block tranfer between VRAM
* Function 24: Fill a VRAM zone with a byte
* Function 25: Fill a VRAM zone with a byte with address autoincrement

--- Disk access functions

* Function 26: Search for a file (S4)
* Function 27: Rename a file (S4)
* Function 28: Delete a file (S4)
* Function 29: Move a file (DOS 2)
* Function 30: Create a file or directory (S4)
* Function 31: Open a file
* Function 32: Close a file
* Function 33: Read from a file (S4)
* Function 34: Read from a file to VRAM (S4)
* Function 35: Read disk sectors (S4)
* Function 36: Read disk sectors to VRAM (S4)
* Function 37: Write to a file (S4)
* Function 38: Write to a file from VRAM (S4)
* Function 39: Write to disk sectors (S4)
* Function 40: Write to disk sectors from VRAM (S4)
* Function 41: Fill a file with one byte (S4)
* Function 42: Move a file pointer
* Function 43: Obtain the default drive and the available drives vector
* Function 44: Set the default drive
* Function 45: Obtain disk space information
* Function 46: Obtain the default directory (DOS 2)
* Function 47: Set the default directory (DOS 2)
* Function 48: Obtain the RAM disk size (DOS 2)
* Function 49: Create the RAM disk (DOS 2)
* Function 50: Obtain the attributes byte for a file (DOS 2)
* Function 51: Set the attributes byte for a file (DOS 2)
* Function 52: Parse pathname (DOS 2)

--- Graphic copression functions

* Function 53: Compression of graphic data
* Function 54: Decompression of graphic data

--- BASIC programs execution

* Function 55: Execution of a BASIC program stored in a segment (S4)
* Function 56: Activation of a BASIC program stored in a segment (S4)
* Function 57: Save a BASIC program with a special header (S4)

--- Miscellaneous functions

* Function 58: Execution of a machine code routine placed on BIOS, SUB-BIOS,
```

```
               BASIC main RAM or system work area
* Function 59: User machine code routine execution
           (machine code routine stored in a RAM segment)
* Function 60: Print a string in graphic mode
* Function 61: Store a string in a segment
* Function 62: Restore a string stored in a segment
* Function 63: Initialization of the SCREEN 0 blink mode
* Function 64: Make or erase a blinking characters block
* Function 65: Obtain information about interruptw
* Function 66: Define or stop an user defined interrupt routine

--- PSG sound effects replay

* Function 67: Obtain information about the PSG sound effects
* Function 68: Initialization of a PSG sound effects set
* Function 69: Play a PSG sound effect
* Function 70: Stop PSG sound effect replay

--- Moonblaster music replay

* Function 71: Load and initialization, or uninstall,
           of the Moonblaster replayer (S4)
* Function 72: Obtain information about the music being replayed
* Function 73: Activation and deactivation of the sound chips
* Function 74: Start playing a Moonblaster music
* Function 75: Stop the music replay
* Function 76: Pause and continue the music
* Function 77: Fade out the music
* Function 78: Load a Music Module samplekit (S4)
* Function 79: Load a MoonSound wavekit (S4)

--- Segments usage control

* Function 80: Obtain and set the number of segments
           allocated for NestorBASIC

--- Interaction with NestorMan and InterNestor Suite/Lite

* Function 81: Obtain information about NestorMan
           and InterNestor Suite availability
* Function 82: Execute a NestorMan function
* Function 83: Data block transfer from a NestorMan segment
           to a NestorBASIC segment
* Function 84: Data block transfer from a NestorBASIC segment
           to a NestorMan segment
* Function 85: InterNestor Suite routine execution
* Function 86: InterNestor Lite routine execution


APPENDIX 2 - USER ACCESSIBLE NESTORBASIC ROUTINES AND VARIABLES

When an user machine code routine or an user interrupt routine is executed,
NestorBASIC segment is SWITCHED in page 1. At the beginning of this segment
there is a table to some internal NestorBASIC routines and variables, which
can be useful for the user routine. In this section, the location of these
routines and variables and its meanings are described.

None of these routines modify the interrupts state, except PUTSLOT0, which
disables interrupts before finishing.

WARNING: Logical segment 255 refers always to the memory switched to pages 2
and 3 when the segment is referred to. While an user machine code routine or
an user interrupt routine is being executed, page 2 DOES NOT contain the
BASIC main RAM segment, but the segment of the routine itself. So, to
```

```
actually convert the segment 255 to the appropriate BASIC segment, do the
following:

    ld    hl,address
    call    CHKSLFF
    cp    3
    jr    z,OKFF
    ld    a,2
OKFF:   ;

Note: variables and data areas mentioned in this section are read-only.
Modifying them may cause unpredictable results.

The contents of the table is as follows:


&H4100: Identification string "NestorBASIC x.xx"


&H4110: TABSEGS

Pointer to the segments table. The format of this table is as follows:

-2: Maximum number of segments that can be allocated for NestorBASIC
    (always valid under DOS 1; under DOS 2 this value is valid
     immediately after NestorBASIC installation, but may vary later)
-1: Number of available RAM segments (same as P(0) returned by
    functions 1 and 80)
+0: Slot of the logical segment 0
+1: Physical segment of the logical segment 0
+2: Slot of the logical segment 1
+3: Physical segment of the logical segment 1
...
+492: Slot of the logical segment 246
+493: Physical segment of the logical segment 246


&H4112: INT_DATA

Contains information about the NestorBASIC active interrupts:

bit 0 = 1: User defined interrupt is active
bit 1 = 1: PSG sound effect is being replayed
bit 2 = 1: Moonblaster 1.4 music is being replayed
bit 3 = 1: Moonblaster Wave music is being replayed


&H4113: PUTSLOT0

Enables the specified slot in page 0, without using ENASLT. Returns with
interrupts disabled.

Input:     A = Slot to switch
Output:    -
Registers: AF


&H4116: CHKSLE

Checks if the specified logical segment exists. DOES NOT recognize VRAM
segments as valid segments, nor the segment 255.

Input:     A = Logical segment
Output:    Cy= 1 -> Logical segment exists
```

```
       Cy= 0 -> Logical segment does not exist
Registers: F


&H4119: CHKSLFF

Checks if the specified logical segment is the 255. If so, the segment
is converted to the appropriate segment number (current page 2 segment if
HL<&HC000, segment 3 if HL>=&HC000)

Input:     A  = Logical segment
       HL = Address
Output:    A  = Segment converted if it was 255; otherwise, unchanged
Registers: F


&H411C: CHKSLV

Checks if the specified logical segment refers to VRAM. If so, the
specified segment address is converted to the appropriate VRAM address.

Input:     A  = Logical segment
       HL = Address
Output:    If segment refers to VRAM:
          A  = VRAM block
          HL = VRAM address
          Cy = 1
       If segment does not refer to VRAM:
          A, HL unchanged
          Cy = 0
Registers: -


&H411F: VTOSL

Converts a VRAM address to the appropriate VRAM segment

Input:     A  = VRAM block
       HL = VRAM address
Output:    A  = Equivalent logical segment
       HL = Equivalent RAM address
       Cy = 1 -> A=1 on input, but only 64K VRAM are available
Registers: F


&H4122: GET_SF

Obtains the physical segment and the slot for a logical segment.

Input:     A = Logical segment
Output:    A = Physical segment
       B = Slot (255 -> Logical segment does not exist)
Registers: -


&H4125: GET_SLT

Obtains the slot connected to page 1 or page 2.

Input:     A  = Page (1 or 2)
Output:    B  = Slot
Registers: F, C
```

```
&H4128: READ_SL

Reads a byte from a logical segment.

Input:     A  = Logical segment
       HL = Address (0-&H3FFF)
Output:    A  = Byte read
Registers: F, AF'


&H412B: WRITE_SL

Writes a byte into a logical segment.

Input:     A  = Logical segment
       E  = Byte to be written
       HL = Address (0-&H3FFF)
Output:    -
Registers: F, AF'


&H412E: LDIRSS

Transfers a data block from a logical segment to another.
Recognizes the segment 255 and the VRAM segments.
Returns with BIOS switched in page 0.
DON'T checks if BC>&H4000.

Input:      IXh = Source logical segment
        IXl = Destination logical segment
        HL    = Source begin address (0..&H3FFF)
        DE    = Destination begin address (0..&H3FFF)
        BC    = Block length (0..&H3FFF)
Output:     A   = 0 -> Data block successfully transferred
        A  <> 0 -> One of the specified logical segments does not exist
Registers:  All


&H4131: CHKBV

Checks if a VRAM address exists.

Input:     A  = VRAM block (0 o 1, 64K low VRAM or high VRAM block)
Output:    Cy = 1 -> The specified VRAM block does not exist
             (A = 1 but computer has only 64K VRAM)
Registers: -


&H4134: SET_RD

Sets VDP for reading VRAM.

Input:     HL = VRAM address, 16 low bits
       CY = VRAM address, bit 17
Output:    -
Registers: AF, HL


&H4137: SET_WR

Sets VDP for writing VRAM.

Input:     HL = VRAM address, 16 low bits
       CY = VRAM address, bit 17
```

```
Output:    -
Registers: AF, HL


&H413A: LDIRVR

Transfers data block from VRAM to RAM.

Input:     VRAM source begin address previously set with SET_RD
       DE = RAM destination begin address
       BC = Block length
Output:    DE = RAM address next to the data block end
Registers: AF


&H413D: LDIRRV

Transfers data block from RAM to VRAM.

Input:     VRAM destination begin address previously set with SET_WR
       HL = RAM source begin address
       BC = Block length
Output:    HL = RAM address next to the data block end
Registers: AF


&H4140: LDIRVV

Transfers data block from VRAM to VRAM through a RAM buffer.

Input:     HL = VRAM source address, 16 low bits
       DE = VRAM destination address, 16 low bits
       BC = Block length
       A  = %000000 D S, bit 17 of Destination and Source
       IX = RAM buffer with a size of at least BC bytes
Output:    -
Registers: AF, HL, DE


&H4143: FILLVR

Fills a VRAM zone with un byte.

Input:     VRAM address previously set with SET_WR
       BC = Zone length
       A  = Byte
Output:    -
Registers: -


&H4146: BLK_CLS

Clears the VRAM zone for the SCREEN 0 blink mode.

Input:     -
Output:    -
Registers: AF


&H4149: BLK_COL

Sets the colors for SCREEN 0 blink mode.

Input:     A = foreground color + 16* background color
```

```
Output:    -
Registers: A


&H414C: BLK_TIM

Sets the times for the SCREEN 0 blink mode.

Input:     A = time ON + 16* time OFF
Output:    -
Registers: A


&H414F: BLK_ON

Makes a blinking block.

Input:     HL = XXYY
       B  = X length
       C  = Y length
Output:    L  = YY next to the last line
       H    unchanged
Registers: AF


&H4152: BLK_OF

Erases a blinking block.

Input:     HL = XXYY
       B  = X length
       C  = Y length
Output:    L  = YY next to the last line
       H    unchanged
Registers: AF


&H4155: C_BLKAD

Calculates the VRAM address in the blink area for the specified coordinates.

Input:     HL = XXYY
Output:    HL = VRAM address
Registers: AF


&H4158: C_STBT

Calculates the appropriate bit of the appropriate VRAM address
for the specified coordinates.

Input:     A = X coordinate
Output:    A has the appropriate bit set to 1
Registers: F


&H415B: GINFOUS

Returns information about the user interrupt definition.

Input:   -
Output:  A  = Segment
     HL = Address
```

```
&H415E: GINFOSFX

Returns information about the PSG sound effects.

Input:   A  = New maximum volume (-1 to leave it unchanged)
Output:  A  = SFX set segment
     HL = SFX set begin address
     B  = Number of the SFX being replayed, or the last one replayed
     C  = Priority of the SFX being replayed, or the last one replayed
     D  = Number of the highest existing sound effect
     E  = Maximum volume


&H4161: GINFOMUS

Returns information about the music being replayed and about the replayer.

Input:   -
Output:  Cy = 1 -> No replayer is initialised,
           the other results are invalid.
     Cy = 0 -> Any replayer is initialised.
           The other results are valid
           if a music is being replayed (see INT_DATA).
     A  = Logical segment of the music being replayed
     HL = Begin address of the music being replayed
     B  = Found sound chips:
          bit 0 = 1 -> MSX-MUSIC found
          bit 1 = 1 -> MSX-AUDIO found
          bit 2 = 2 -> OPL4 found
     C  = Enabled sound chips:
          bit 0 = 1 -> MSX-MUSIC enabled
          bit 1 = 1 -> MSX-AUDIO enabled
          bit 2 = 2 -> OPL4 enabled
     D  = Position
     E  = Step (0 to 15)
     IXl= 255 if the music is paused


&H4164: REPTYPE

This is a data byte that contains the type of the music replayer currently
loaded:

0 = Moonblaster 1.4 replayer
1 = Moonblaster Wave replayer

Note that before looking for this information, you must check that actually
any replayer is loaded. Use function GINFOMUS (&H4161) for this.


&H4165: GETF01

Input:  -
Output: -

This routine copies the contents of the strings F$(0) and F$(1) to two
NestorBASIC buffers placed on page 3; the addresses of these buffers can be
obtained from variables F0BUFADD and F1BUFADD, mentioned forward.

Only the first 80 characters of the strings are copied. A 0 character is
placed at the end of the strings.


&H4168: SETF0
```

```
&H416B: SETF1

Input:  -
Output: -

These routines establish the strings F$(0) and F$(1), respectively, with the
contents of two NestorBASIC buffers placed on page 3; the addresses of these
buffers can be obtained from variables F0BUFADD and F1BUFADD, mentioned
forward.

Strings must be finished with a 0 character in the buffers, and its maximum
length is 80 characters; if they are larger, only the first 80 characters
will be established.


&H416E: F0BUFADD
&H4170: F1BUFADD

These variables store the addresses of the page 3 buffers that are used as
destination by routine GETF01 and as source by routines SETF0 and SETF1,
respectively. The length of these buffers is 80 bytes each one.


&H4172: SL_2

Logical segment switched on page 2 (that is, logical segment where the
routine reading this variable resides).


&H4173: NMAN_SL4

NestorMan segment number of NestorBASIC segment 4.


&H4174: INS_SL1
&H4175: INS_SL2
&H4176: INS_SL3
&H4177: INS_SL4

NestorMan segment numbers of InterNestor Suite modules 1 to 4, respectively.


APPENDIX 3 - INTERRUPTS IN MSX TURBO-R

NestorBASIC works perfectly in all the MSX2/2+/Turbo-R computers, but you may
have problems with the interrupt process if your computer is a Turbo-R and
you are running in DOS 1 mode, especially if you have an external mapper.
This is due to the process needed for the interrupt request handling:
NestorBASIC must check which segment is connected in page 1 before making any
segment switching. In DOS 1 mode, the only way to do this is the use of a IN
A,(&HFD) instruction, which in the Turbo-R computers does not work exactly in
the same way as in the MSX2/2+ computers.

Thus, if you will use user interrupts, sound effects or music replay in your
program using NestorBASIC, it is recommended to do the following: check if
the computer is a Turbo-R (if PEEK(&H2D)=3) running in DOS 1 mode (use
function 1 to check this); in such case, show a message asking to reboot in
DOS 2 mode.


APPENDIX 4 - USAGE CONDITIONS

NestorBASIC is freeware, so you can use and distribute it freely. But if you
use NestorBASIC in your programs you should do the following:
```

```
- Somewhere in your program (when loading, in an "about" option, when showing
the staff, etc) you must place a mention to the use of NestorBASIC, as well
as the version number (use function 1 to obtain NestorBASIC version number).

- If the program is not for author's personal use (that is, if the program
will be distributed, freely or not) you must give a copy of the program to
me.

For any suggestion, question or comment about NestorBASIC, please write to me
at <konamiman@konamiman.com>.

The usage of the sound effects created with the sound effects editor SEE has
its own use conditions. If you will shell your program for more than 15 NLG
you must pay a little amount of money to the authors, never more than 25 NLG.
For more details or for comments about SEE, please contact Fuzzy Logic:

       R. v/d Meulen        A. v/d Wal
       Lijsterstraat 25     Tormentil 15
       8917 CX Leeuwarden   8445 RR Heerenveen
       Holland.         Holland.
```

Загрузка

Для загрузки нужно выполнить в MSX BASIC команду:

```
BLOAD"NBASIC.BIN",R
```

Также можно выполнить вызов из файла автозапуска:

AUTOEXEC.BAS

```
10 BLOAD"NBASIC.BIN",R:NEW
```

Ссылки

Nestor BASIC

Трактат о X-Basic'e

MSX2(2+) Universal font editor v3.3, with drivers (Kun-Basic)

https://sysadminmosaic.ru/msx/nextor_basic/nextor_basic

2021-05-15 14:03