File: API-C-EN.DOC (59 pages) February 22, year of the CPC 2019

Documentation of FutureOS functions located in ROM C

All OS functions are described in the following form:

1. Short description: describes an OS function in brief.

2. Label: this label labels the OS function in the (Z80-)source code. The actual label-library #EQU-API.ENG (delivered with the OS) contains the correct and newest address for every OS function. Please use EVER the appropriate LABEL when using an OS function or system-variable! Never use the direct address! Addresses can change in further versions of FutureOS, but the features of OS functions and system-variables should remain in future.

3. ROM-number: gives the logical number of the ROM in which the appropriate OS function is located. More numbers are possible, if the function is present in different ROMs. This logical ROM number doesn't have to be equivalent with the physical number of the ROM. Only older versions of FutureOS have identical logical and physical ROM numbers (&0A, &0B, &0C and &0D). Now ROM expansions are bigger.

4. Start address: gives you the entry address of the current OS function in it's ROM. If the OS function exists in more than one ROM there may be more addresses given.
This address is the same you'll find in the file #EQU-API.ENG for every label.

5. Jump in conditions: the conditions when the OS function get called are given here. Registers and RAM-variables must be load with the correct values. Which they are is described here.

6. Jump out conditions: describes registers and RAM-variables at the time when the OS function returns. Often registers give information if the OS function did return successful or not.

7. Manipulated: all the manipulations which was done through the OS function are given. Registers and RAM-variables are most interesting.
But changes in memory or what ever are described, too.

8. Description: this is the complete description of the functions of the entire OS function.

9. Attention: crucial details are described here. All the OS functions are trimmed to high-speed. Therefore you have to deal with them in a correct way. If not something unmeant could happen.

The FutureOS ROM C provides a variety of different OS functions. For example OS functions to move data very fast, the DIRectory management, to load and save data, rename, and and and ... Much high-level OS functions are located in this ROM C.

The newest version of this file can be found in the internet:

http://www.FutureOS.de

FILL A MEMORY REGION EXTREME FAST

Short description: Fills a part of the RAM with a byte or a word. This OS function is the fastest way to fill memory with data.

Label: F_FILL8 (fill memory with a byte, 8 bit) or
F_FILL6 (fill memory with a two byte word, 16 bit)

ROM-number: C

Start address: &C01E (F_FILL8) or &C01F (F_FILL6)

Jump in conditions: BC = Length of the region to be filled.
D = Value that should be written to RAM using 8 bit version F_FILL8
... or ...
DE = Value that should be written to RAM using 16 bit vers. F_FILL6.
HL = Startaddress in RAM, where the filling should start.

Jump back conditions: A part of the RAM has been filled.

Manipulated: AF, B, HL, HL' and when using F_FILL8 additional E.

Description: This OS function(s) fill part of the RAM in a short time. The bigger the space to be filled, the faster this function. It needs about 1.5 µs to fill one byte in RAM. For example you can use this functions(s) to fill the 16 KB V-RAM, that means to clear the video screen. Before you call one of these two functions you must load register HL with a pointer to the first address in RAM that should be filled.
Register BC must contain the length (in bytes) of the RAM block that should be filled. This functions fill the memory ascending. So the memory between HL and HL + BC will be filled. You can fill RAM with an 8 bit value, therefore use label F_FILL8. In this case you must load register D with the desired byte. Else you can fill the RAM with an 16 bit value. In this case use label F_FILL6 and load register DE with the 16 bit value. The low byte of the 16 bit value will be written to RAM before the high byte.

Examples:

```
TEST_FILL_8     LD BC,&4000       ;&4000 bytes of RAM will be
                LD D,&09          ;filled with the value &09,
                LD HL,&2000       ;starting at address &2000
                CALL F_FILL8

TEST_FILL_16    LD BC,&4000       ;&4000 bytes of RAM will be
                LD DE,&1234       ;filled with the value &1234,
                LD HL,&2000       ;starting at addresse &2000
                CALL F_FILL6
```

If ROM C is not switched on, you have to insert the following two lines just before the above code:

```
LD BC,(&FF0D)    ;get physical number of ROM C
OUT (C),C        ;and switch ROM C on!
```

Attention: This functions overwrite any kind / space of RAM. So if you load a register with the wrong value and call the function, then the stability of the system could be endangered. When using 16 bit filling (F_FILL6), then the low byte (register E) is written to RAM before the high byte (register D).

COPY A MEMORY AREA BIGGER THAN 255 BYTES

Short description: Copies a memory area of more than 255 bytes fast.

Label: F_MOVE

ROM-number: C

Start address: &C0C8

Jump in conditions: BC = Length of the memory area, that should be copied. This length must be at least &100 / 256 bytes or bigger. That means that register B isn't allowed to be zero.
DE = Target address of the memroy area that should be copied.
HL = Source address of the memroy area that should be copied.

Jump back conditions: The memory area has been copied.

Manipulated: AF, BC, DE and HL.

Description: A memory area of at least 256 bytes will be copied with highest possible speed. You can copy the memory up and down. Source- and target-areas can overlap.

Before calling OS function F_MOVE the register HL must point to the first byte of the source block of RAM, register DE must point to the first byte of the target block of RAM. And register BC must contain the length of the block of RAM that should be copied. BC must be &100 / 256 bytes at least or longer. Memory blocks fewer than 256 bytes must be copied with the Z80 commands LDIR or LDDR (slower than F_MOVE).

Example:

```
HOLE_BILD   LD DE,&C000       ;Target address (of memory area)
            LD HL,&4000       ;Source address (of memory area)

            LD B,H            ;Length of block / memory area
            LD C,L            ;BC equals HL, so both contain &4000

            CALL F_MOVE       ;copy block of BC bytes from HL to DE
```

Attention: If the register B contains zero and you call F_MOVE, then this OS function will copy a block of 64 KB, so the system will crash.
Therefore always load BC with a value of 256 bytes or more.

COPY A MEMORY AREA BIGGER THAN 255 BYTES UPWARDS

Short description: This OS function is a part of F_MOVE, it's used to copy RAM blocks of 256 bytes or bigger upwards in memory.

Label: LDD_256

ROM-number: C

Start address: &C2E8

Jump in conditions: BC = Length of the memory area, that should be copied. This length must be at least &100 / 256 bytes or bigger. That means that register B isn't allowed to be zero.
DE > HL
DE = Target address of the memroy area that should be copied.
HL = Source address of the memroy area that should be copied.

Jump back conditions: The memory area has been copied.

Manipulated: AF, BC, DE and HL.

Description: A memory area of at least 256 bytes will be copied with highest possible speed. Source- and target-areas can overlap, but in this case the source address must be lower than the target address.
Before you call OS function LDD_256 the register HL must point to the first byte of the source block of RAM, register DE must point to the first byte of the target block of RAM. And register BC must contain the length of the block of RAM that should be copied. BC must be &100 / 256 bytes at least or longer. Memory blocks fewer than 256 bytes must be copied with the Z80 commands LDIR or LDDR (slower).

Attention: If the register B contains zero and you call LDD_256, then the OS function will copy a block of 64 KB, so the system will crash.
Always load BC with a value of 256 bytes or more.

COPY A MEMORY AREA BIGGER THAN 255 BYTES DOWNWARDS

Short description: This OS function is a part of F_MOVE, it's used to copy RAM blocks of 256 bytes or bigger downwards in memory.

Label: LDI_256

ROM-number: C

Start address: &C0D0

Jump in conditions: BC = Length of the memory area, that should be copied. This length must be at least &100 / 256 bytes or bigger. That means that register B isn't allowed to be zero.
DE < HL
DE = Target address of the memroy area that should be copied.
HL = Source address of the memroy area that should be copied.

Jump back conditions: The memory area has been copied.

Manipulated: AF, BC, DE and HL.

Description: A memory block of at least 256 bytes will be copied with highest possible speed. Source- and target-areas can overlap, but in this case the source address must be higher than the target address.
Before you call OS function LDI_256 the register HL must point to the first byte of the source block of RAM, register DE must point to the first byte of the target block of RAM. And register BC must contain the length of the block of RAM that should be copied. BC must be &100 / 256 bytes at least or longer. Memory blocks fewer than 256 bytes must be copied with the Z80 commands LDIR or LDDR (slower).

Attention: If the register B contains zero and you call LDI_256, then the OS function will copy a block of 64 KB, so the system will crash.
Always load BC with a value of 256 bytes or more.

COPY CHARACTER SET FROM ROM TO RAM

Short description: OS function copies ROM character set to RAM.

Label: INRZ

ROM-number: C

Start address: &C9AD

Jump in conditions: RAMCHAR defines the screen mode.

Jump back conditions: Character set was copies from ROM to RAM.

Manipulated: AF, BC, DE, HL RAM &3800-&3FFF, lower RAM active.

Description: This OS function is used to copy the character set (all 256 characters) from ROM addresses &3800 to &3FFF into the same memory are in RAM.
After the return of INRZ the lower RAM is banked in. Therefore this OS function can also be called from the lower RAM area.
The screen mode will be set according to the OS variable RAMCHAR.

Attention: The memory area between &3800 and &3FFF was overwritten.

SORT A PREVIOUSLY READ DIRECTORY

Short description: Sorts a previously read DIRectory of a floppy disc.

Label: TS_D_IN

ROM-number: C

Start address: &FDA7

Jump in conditions: D = rounded up high-byte of the upper end of the memory block, where the unsorted DIR is located.
REG16_1 = Length of the DIRectory in bytes, for example &800 2048 when using Data formatted discs. Or &1000 4096 bytes when using a disc with Vortex format.
It doesn't matter how much the DIR is used, the complete lenght (with all unused entries) is needed.
REG16_2 = Pointer to the start (first byte) of the DIR (to be sorted).
REG16_3 = REG16_2. Both system variables must contain the same value.
REG16_5 = Pointer to the start address of a buffer, that is as big as the DIR that should be sorted.
REG16_6 = REG16_5. Both system variables must contain the same value.
REG16_7 = Pointer to the end of the buffer defined by REG16_5. The programmer can compute this value by adding the values of REG16_5 and REG16_1. But you have to write this calculated value yourself to this system variable REG16_7.

Jump back conditions: The DIRectory is now sorted.

Manipulated: AF, BC, DE, HL, REG16_0, REG16_3, REG16_4 and REG16_6.

Description: This OS function performs multiple actions:
- First all empty DIRectory entries are fltered out.
- Then the valid entries will be sorted numero-alphabetical.
- And the empty space in the DIR will be filled with the byte &E5.

Let's talk about a Data formatted disc as example. Let's say that the DIR has been read from disc to RAM between &6000 and &6800. So its length is &800 bytes (write that value to system variable REG16_1).
The system variables REG16_2 and REG16_3 will get the value &6000, that's the start address of the DIR in RAM. Further a buffer (in the size of the DIR, also &800 bytes) is needed, we can use the region between &7000 and &7800. So we write the value &7000 to REG16_5 and to REG16_6 (start address of buffer). And to system variable REG16_7 we write the value &7800. Now the Z80 register D needs the high-byte of the unsorted DIR. This value is calculated by start_of_DIR + length.
The high-byte of this calculation (&6000 + &800 &6800) is &68. And now you can call TS_D_IN to sort the DIR!
After the DIR was sorted it's located at its old address in RAM.

Attention: A buffer - as long as the DIR itself - must be available. All the addresses you load to the sysstem variables must be &XX00, that means that you can divide them by 256 (&100). Good examples of valid addresses are for example &100, &200, &300, ..., &8600, &8700, 8800. This OS function is used by the Turbo Desktop of FutureOS.

CONVERT ALL 32 BYTE ENTRIES TO 16 BYTE ENTRIES

Short description: This OS function is used by the Turbo Desktop: It converts all enties (32 byte) of the DIRectories to 16 byte enties.

Label: DIRWA

ROM-number: C

Start address: &FD9E

Jump in conditions: TURBO_X must contain a valid value.
The first four bytes (of the eight bytes) of system variables TURBO_A ... TURBO_M must contain valid values. If a drive is inaktive, it's enough when the drive tagging byte is set correct.
Half the amount of expansion memory used for buffered DIRectories must be free and useable.

Jump back conditions: A 16_byte_entries version of every DIRectory was generated, which can be used to be displayed on the screen.
The system variables TURBO_A .. TURBO_M describe the location and length in RAM.
Further the variable TURBO_X was adapted.

Manipulated: AF, BC, DE, HL, DE', HL', IY and XH.

Description: If you use the DIR function of the Desktop, then first all DIRectories of all tagged drives were read and buffered in RAM.
The buffered DIRs must be converted, so that they can be shown on the screen. This is done by this OS function DIRWA.
DIRWA uses the system variables intensively. So the variables TURBO_A ... TURBO_M (each consist of eight bytes) must contain correct values in their first four bytes. Especially their drive tagging bytes must be valid. Also variable TURBO_X must contain a correct value. That means it must be enough free expansion RAM for the 16 byte versions of the new generated 16 byte DIRectories.
The function DIRWA switches on the expansion RAM of every used drive, converts their 32 byte entries to 16 byte entries and writes the 16 byte version of the DIR back to free expansion RAM. Informations about this process were written to the system variables TURBO_A .. TURBO_M.
Further TURBO_X is actualized. It points now to the highest free byte in the highest free 16 KB expansion RAM.
The 16 byte version of the DIR is now ready to be displayed on screen.

Attention: Use this OS function only if enough free expansion RAM is available. Because it will break if E-RAM is too small.

CLEAR THE SCREEN

Short description: Clears the screen (16 KB video memory) very fast.

Label: LESC

ROM-number: C

Start address: &C017

Jump in conditions: V-RAM must begin at address &C000 (just normal).

Jump back conditions: The screen RAM between &C000 and &FFFF is now cleared with byte &00.

Manipulated: AF, BC, DE, HL, HL' and &4000 bytes video RAM.

Description: This OS function LESC resembles the BASIC command CLS, because it clears the complete video RAM between &C000 and &FFFF. The screen is filled with the byte &00.

What makes this OS function special? Well, it's very fast, it only needs about 2 us to clear one byte. So it takes about 0.03 seconds to clear the complete 16 KB screen RAM.

Attention: There is no faster way to clear the 16 KB video screen RAM.

CLEAR THE LOWER PART OF THE SCREEN, BEGINNING AT 15. LINE

Short description: When using 32 lines with 64 colums (Mode 2) the lower half of the screen RAM (beginning at 15. line) will be cleared.

Label: LEDA

ROM-number: C

Start address: &C4F5

Jump in conditions: V-RAM must start at &C000 (that is just normal).

Jump back conditions: Lower half of the screen is cleared to &00.

Manipulated: AF, BC, DE, HL, HL' and &480 bytes V - RAM are cleared.

Description: This OS function LEDA is used to clear the lower half of the screen (fill with byte &00).

It starts at the 15. line and ends at the bottom of the screen.

Only use LEDA if the screen is set to 32 lines and 64 columns, else it could be that the cleaning process doesn't match the start of line 15.

Since this OS function uses the FutureOS F_FILL function it is very fast. There is no faster way to do this in software.

LEDA is used for example by the Desktop to clear the region of the screen where the DIRectories were usually displayed.

Attention: The screnn must have 32 lines and 64 columns in each line.

Generate TABLE to LOAD FILES from DATA, SYSTEM or IBM formatted DISCS

Short description: A Track-Sector-Table / Loading-Table will be generated. Usable for discs in IBM, Data or System format.

Label: LTAI (IBM format) // LTAB (DATA format) // LTAS (SYSTEM format)

ROM-number: C

Start address: &C664 (LTAI) // &C539 (LTAB) // &C710 (LTAS)

Jump in conditions:
HL = 1. Byte of the 1. Extent of the file, in the 32 byte DIRectory
HL' = 1. Byte of the Track-Sector-Table (target address)

Jump back conditions: Track-Sector-Table / Loader-Table was generated.

Manipulated: AF, BC, DE, HL, BC', DE' and HL'

Description: To load a file there must be first generated a so called Loader-Table. This Track-Sector-Table can be made by this OS function, but only for files on a disc in IBM, System or Data Format.

First you have to switch on the E-RAM which contains the 32 byte DIR of the souce file. Then you have to load HL with the source address of the first extent of the source file (that you want to load). Further you have to load HL' with the target address where the Loader-Table should be generated (saved) in RAM.

Construction of the Table using discs with IBM, System or Data Format:

The table begins with a "Track Number", then the "Number of Sectors" (which should be loaded from this track) is provided. Then the numbers of the single Sectors follow.

Then the next Track number follows and so on ...

- If the "Number of Sectors" is &FF, then the complete Track must be loaded. In this case the Track-Number is ONLY followed by &FF, but there are NO Sector-Numbers provided. The next Track-Number follows directly.

- If the Track-Number is &FF, then the End of the Table is reached. So the "End of Table" is marked by a Track-Number of &FF.

Attention: HL must contain a correct value, else if at HL no extent begins this can lead to unknown errors or even a crash.

GENERATE TABLE TO LOAD FILES FROM VORTEX FORMATTED DISCS

Short description: A Track-Sector-Table / Loading-Table will be generated. Usable for discs in VORTEX Format.

Label: LTAV

ROM-number: C

Start address: &C7FB

Jump in conditions:
HL = 1. Byte of the 1. Extent of the source-file, 32 byte DIRectory
HL' = 1. Byte of the Track-Sector-Table/Loader-Table (target address)

Jump back conditions: Track-Sector-Table / Loader-Table was generated.

Manipulated: AF, BC, DE, HL, BC', DE' and HL'
Two bytes located in RAM before HL' have also been manipulated, because they were shortly used as pseudo table.

Description: To load a file there must be first generated a so called Loader-Table. The Track-Sector-Table for file on discs in VORTEX format can be made by this OS function. The generated table has NOT the same construction as Data/System/IBM tables.

First you have to switch on the E-RAM which contains the 32 byte/extent DIR of the souce file. Then you have to load HL with the source address of the first extent of the source file (that you want to load). Further you have to load HL' with the target address where the Loader-Table should be generated (saved) in RAM.

Please continue to read on the next page!

Construction of the Table using discs with Vortex Format:

For Vortex Format: Every track number also contains the head number!!!

The table begins with a "Track / Head Number", that value contains the number of the track (0..80) multiplicated by two - AND it contains the Head-Number (0 = upper side, 1 = lower side) in Bit 0. The upper-side and the lower-side of a track therefore have foreign Track-Numbers.

You can calculate the physical Track-Number by a shift right: SRL A. Then the Carry contains the Head-Number.

After the Track-Head-Byte the "Number of Sectors" (which should be loaded from this track) is provided.

Then the numbers of all the single Sectors of the Track follow.

Then the next Track-Head-Number follows and so on ...

- If the "Number of Sectors" is &FF, then the complete Track must be loaded. In this case the Track-Head-Number is ONLY followed by &FF, but there are NO Sector-Numbers provided. The next Track-Head-Number follows directly.

- If the Track-Head-Number is &FF, then the End of the Table is reached. So the "End of Table" is marked by a Track-Hd-Number of &FF.

Attention: HL must contain a correct value, else if at HL no extent begins this can lead to unknown errors or even a crash.
BEWARE: Two bytes before HL' will be manipulated, eventually stored data will be lost.

COMPARE TWO EXTENTS OF ONE DIRECTORY

Short description: This OS function checks if two extents are part of the same file. User number, name and extension will be compared.

Label: EXCP

ROM-number: C

Start address: &C94D

Jump in conditions: DE = &XXX0 = 1st extent, that should be compared.
HL = &YYY0 = 2nd extent, that should be compared.
The addresses of both extents must be dividable by 16. That means that the lower four bits of register E and L must contain zero bits: &XXX0.

Jump back conditions:
Z-FLAG set (Z): both extents are part of the same file.
Z-FLAG cleared (NZ): the two extents are unequal and they are NOT part of the same file.
Contents of registers DE and HL (pointers to extents) are unchanged.

Manipulated: AF and BC

Description: Sometimes it is necessary to compare two file extents, for example to investigate if a file has one more extent or so ... For such questions this OS function EXCP is made.

When calling EXCP the registers DE and HL must contain the RAM address of the two extents that should be compared. But beware both addresses must be divideable by 16, that means they must fit into &XXX0, where X can be every number.

This function compares user-number, file-name and the file-extent of both extents. All other bytes of the extent will NOT be compared. That means that from the 32 bytes of an extent only the first 12 bytes will be used for comparison. If the first 12 bytes of two extents are equal (the rest doesn't matter), then both extents are part of the same file (in one DIRectory). In this case EXCP returns with a set zero flag. If the two compared extents don't belong to the same file, then EXCP will return with a cleared zero flag.

The registers DE and HL (pointers to two extents) were saved. Only the registers AF and BC will be changed through EXCP.

Attention: The start address of both used extents must be dividable by 16 (format: &XXX0), also errors can occur which set the zero flag in a wrong way.
This OS function is used for normal 32 byte extents of a file, like they were used in normal DIRectories. But only the first 12 bytes will be compared.

SORT AND CONVERT DIRECTORIES FROM 32 BIT TO 16 BIT VERSIONS

Short description: All marked (and manipulate) DIRectories will be new sorted and converted from the 32 byte/extent version to the 16 byte/name version.

Label: ISWS

ROM-number: C

Start address: &FDA1

Jump in conditions: RAM variables TURBO_A..M must be valid.

Jump back conditions: All concerned DIRs have been sorted and converted. The lower ROM is switched off.

Manipulated: All register but AF' and XL. Further the RAM variables REG16_0..9 and so on...
Beware: The RAM between &0000 and &3FFF is also manipulated. That 16 KB were used as buffer.

Description: If some DIRectories have been changed (for example by using a file operation like ERA, REN or something similar), then the DIRectories aren't sorted any longer. So also the 16 bit versions of that DIRs aren't valid any longer.

The OS function ISWS tests which drives / hard-disc partitions are active and which of the corresponding DIRs have been changed (bit 3 set in the TURBO_A..M system variables). First all changed DIRs (32 bit version) will be sorted. Then the new 32 bit DIRs will be used to generate new 16 bit DIRs. (Now it is like that DIRs would have been loaded again). The old expansion RAM buffers are used, so no RAM is wasted.Well, what is a manipulated / changed DIR??? The status of a drive or partition will be set to "manipulated" when its 32 bit DIRectory has been changed. Example: If you change the name of a file with the REN icon, then the DIRectory has been manipulated, the status of that drive is automatically set to "manipulated". That way this OS function can sense that there is work to do.

Attention: When ISWS returns, then the lower ROM is switched off. The "print on screen" functions will only work if you have previously load a character set to &3800. Else you can switch the lower ROM on.
The 16 KB RAM between &0000 and &3FFF was used as buffer.

TEST EXPANSION RAM AND SET SYSTEM VARIABLES

Short description: All 16 KB expansion RAMs will be tested if they're connected. The corresponding system variables will set.

Label: RANI or RAMI

ROM-number: C

Start address: &C8EF (RANI) or &FD35 (RAMI)

Jump in conditions: -

Jump back conditions: The 32 system variables XRAM_C4 ... XRAM_FF have been set to correct values:
Bit 0 = 0 => 16 KB RAM block is NOT connected.
Bit 0 = 1 => 16 KB RAM block IS connected.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL' and the RAM state is set to main memory (&7FC0) in case of RAMI
RANI: The first byte (&4000) of every 16 KB exp. RAM block is changed
RAMI: Exp-RAMs have been NOT changed, but RAM &B7C0-&B7DF is changed

Description: The OS must know how much expansion RAM is connected and where (which RAM select) the exp. RAM is located.

Both OS functions test all 32 16 KB expansion RAM blocks that can be switched in between &4000 and &7FFF throug port &7Fxx. The system RAM provides a system RAM variable for every one of this 16 KB expansion RAM blocks. This bytes are the 32 XRAM_?? bytes, where ?? can contain the characters C4, C5, C6, C7, CC, CD, CE, CF, D4, D5, .. ,FF.

This OS functions will set all 32 XRAM system variables correctly.

If the first bit (bit 0) is set, then the corresponding 16 KB RAM is connected to the CPC actually. If this bit 0 is cleared to zero, then the corresponding exp. RAM block is NOT connected.

The FutureOS uses all connected expansion RAM in blocks of 16 KB and switch that 16 KB blocks into the main memory between &4000 and &7FFF.

Attention: Only OS function RANI manipulates the first byte of every 16 KB block of the expansion RAM (at address &4000).
OS function RAMI does NOT change the expansion RAM, but it uses the RAM between &B7C0 and &B7DF as buffer.

COMPUTE FREE EXPANSION RAM AND SET SYSTEM VARIABLES

Short description: Computes free, usable expansion RAM and sets system variables to correct values.

Label: FESB

ROM-number: C

Start address: &C998

Jump in conditions: The 32 system variables XRAM_C4 .... XRAM_FF must contain correct values.

Jump back conditions:
D = Number of free 16 KB (short time buffer) expansion RAMs.
E = &09
BC = &00F7
HL = &B9EF = Pointer to XRAM_FF system variable.

Manipulated: AF, BC, DE, HL and maybe some XRAM_?? system variables.

Description: For all 32 expansion RAM blocks (of 16 KB each) that can be accessed through port &7Fxx there is a system variable XRAM_??.

Every one of the 32 system variables informes you if the corresponding 16 KB RAM is connected and in which way it is used.

The OS function FESB counts all existing and free/unused or as "short time buffer" defined 16 KB expansion RAM blocks. When returning FESB provides the number of free 16 KB blocks in the Z80 register D. Take a look at the XRAM system variables to know which blocks are free. And this is the second thig FESB does: It sets all free / unused exp. RAMs to status "short time buffer". The XRAM_?? variable of a free 16 KB RAM block will be set to &09 to define it free ("short time buffer").

Attention: When calling FESB, the contents of all 32 system variables (XRAM_C4 to XRAM_FF) must contain valid values.
Please have a look at the FutureOS memory map to learn more about the single bits of the XRAM_?? system variables.

SEARCH FIRST TAGGED FILE

Short description: Search all file-tagging-bytes for a tagged file.

Label: FMD32

ROM-number: C

Start address: &FD98

Jump in conditions: TURBO_A ... M and all file-tagging-bytes must hold correct values.
A < &FF => normal function
A = &FF => file-tagging-status will NOT be affected!

Jump back conditions:
HL = &0000 => NO file is tagged.
HL = Pointer to the address of the first extent of the first tagged file inside the 32 byte DIRectory.
REG08_1 = Drive / Partition on which the file is located.
REG08_2 = Physical RAM number &C4, &C5, &C6 ... &FF of the 32 byte DIR in which the tagged file is located. This expansion RAM is already switched on.
REG16_6(...REG32_1) = 16 byte file-name, like it is written in the 16 byte DIR.
The drive-motor of all floppy disc drives is switched on, also when no file is tagged.
When you call FMD32 with register A UNequal &FF:
The file-tagging-status will be set from "tagged" to "retagged/old tagged". Such a file-name is shown streaked out when browsing the DIR.

Manipulated: All registers but IY. RAM status, file-tagging-byted, the drive motor status and system variables REG08_1..2, REG16_6..REG32_1.

Description: The OS function FMD32 is used to search for a tagged file in all active drives or partitions. When calling FMD32 the system variables must contain valid contents.

FMD32 provides register HL with a pointer to the address of the first extent of the first found tagged file. If HL contains &0000 then no tagged file has been found.
If a tagged file has been found (HL not 0) the system variable REG08_1 contains the number of the drive / partition where the tagged file is located.
Further system variable REG08_2 contains the physical expansion RAM number (&C4 .. &FF) in which the corresponding 32 byte DIRectory is located. This expansion RAM is already switched on between &4000 and &7FFF.
FMD32 starts the floppy disc drive motors, if any drive is marked. So if a tagged file has been found, the drive motors are running. But you still have to wait until the drive is "Ready" before loading the file.

Attention: The system variables must contain correct values, else the OS function will provide corrupt data. Error!
Beware: If register A contains the value &FF before calling FMD32, the OS function will NOT change the status of the tagged file to not tagged. In this case you cann call FMD32 again to find the same tagged file again.

GENERATE BLOCK-USAGE-TABLE FOR DISCS IN DATA/IBM/SYSTEM/VORTEX FORMAT

Short description: This OS function generates the Block-Usage-Table for a floppy disc drive with Data, IBM, System or Vortex format. But the DIRectory must NOT be sorted.

Label: BBTG (XL contains drive number), BBTH (A contains drive number)

ROM-number: C

Start address: &FD95 (BBTG), &FD53 (BBTH)

Jump in conditions: A or XL = Number of drive from 0 to 7 (A to H).
The DIRectory of the drive must be read from disc and buffered in RAM.
The system variables must contain correct values.

Jump back conditions: The Block-Usage-Table has been generated and saved to RAM between &B600 and &B6FF. Block &00 (address &B600) is never used.
XL = Number of drive from whose DIR the table was generated.
XH = Drive-Tagging-Byte of drive previously numbered in A or XL.
The expansion RAM of the 32 bytes DIR of the drive previously numbered in register A or XL is switched on (&4000-&7FFF).

Manipulated: AF, BC, DE, HL, AF', (XL), XH and the RAM configuration.

Description: If the OS wants to save a file it must know which blocks on a floppy disc are free for usage. To investigate which blocks are still free on a floppy disc with Data, IBM, System or Vortex format this OS function can be used.

When you call the OS function you have to provide the number (0..7) of the drive (A..H) which shall be investigated. You can write this number in register A (use entry point BBTH) or in register IX low (entry BBTG).

BBTH / BBTG will generate a Block-Usage-Table beginning at RAM address &B600 and ending at address B6FF (latest). Address &B601 contains the status of block &01, address &B602 contains the status of block &02, and so on ... until the last block is reached. If such a Block-Byte contains the value &00 then the corresponding block is FREE for usage.

But if the Block-Byte contains the number &FF then this block is used by another file and you can't use it for saving a new file. Further the expansion RAM of the DIR is switched on (between &4000 and &7FFF) when the OS function jumps back.

When using BBTH / BBTG to generate a Block-Usage-Table the DIR can be unsorted, this doesn't matter.

BEWARE: The blocks of the DIR will NOT be marked as used by this OS function! It only generates the table.

The different floppy disc drive formats have different maximum block numbers. When using Vortex format one block has 4 KB (NOT 1 KB). Also the DIR will use different blocks.

Drive with Data format has 180 blocks of 1 KB, DIR in block 0,1.
Drive with ibm format has 156 blocks of 1 KB, DIR in block 0,1.
Drive with System format has 171 blocks of 1 KB, DIR in block 0,1.
Drive with Vortex format has 177 blocks of 4 KB, DIR in block 0.

Attention: The DIR of the source drive must be buffered in the E-RAM.

The DIR is NOT changed in any way, no blocks will be marked.

When using Data, IBM or System format the blocks &00 and &01 are used for the DIR. Vortex discs only use block &00 for the DIR.

At jump back register XH contains the drive-tagging-byte, which also contains the format information.

LOAD A FILE OF DEFINED NAME

Short description: Loads a file of defined name from defined drive.

Label: LADE_N

ROM-number: C

Start address: &FD5C

Jump in conditions: The DIRectory which contains the name of the file must have been loaded previously (tagged drive, DIR icons was used).

A = Drive (0..12) from which the file should be loaded.
If the eight bit of A is set then a file header will be ignored.

DE = Pointer to User, Name and Extension of the file. In detail one byte for the User-number, 8 bytes for the name and 3 bytes for the file-extension. This address must be dividable be 16, its format is: DE = &???0.

If the file that should be loaded has NO file-header, then additional information is needed (write them into the system variables):

REG08_4 = Type of Loading (0, 1, 2 or 3) - look at OS function LADEN.

REG16_3 = Target address in RAM, 16 bit.

AKT_RAM = Physical expansion RAM number &7FC4, &7FC5..&7FFF .. &78FF.

Jump back conditions: The Z80 register A will contain a status byte which informs about the success of the operation:

A = &00 => NO DIRectroy has been read at all, so NO file can be loaded from any drive.

A = &01 => The source drive is not tagged, its DIR is NOT read, no file can be loaded.

A = &02 => The named file doest NOT exist in the DIR of the named drive. The RAM configuration (&400-&7FFF) is unknown!

A = &FF => The file has been loaded. But you can additionally take a look at the 7 result bytes beginning at FDC_RES.

(REG_PC+1) = Code (from 0 up to 12) of the source drive (A..M) from which the file has been loaded.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the system RAM-variables REG08_0-3,5,6 REG16_0,1 REG_PC. Further the RAM regions &B000-&B7FF and &BC00-&BC7F, the RAM state, the FDC, the drive motor, the file-tagging-bytes. On error the RAM &4000-&7FFF is unknown.

Description: This OS function is used to load any existing file. The file is defined through its source-drive, user-number, file-name and file-extension. It doesn't matter if the file is tagged or not. Its tagging state will not be influenced.

The drive to load the file from is given in A, if the high bit is set, then the file-header will be ignored. Else the file-header decides where the file will be loaded to. If the file has no header (or the header is ignoren), then the following RAM variables decide loading:

REG08_4 = Type of loading:

Value &02 loads a file direct into the main memory, the first 64 KB of the CPC.

Value &03 loads a file into the expansion RAM of the CPC.

REG16_3 = Target address in RAM, thats an 16 bit value.

AKT_RAM = Physical expansion RAM number from &7FC4 up to &78FF.

When a file was loaded without an error, then this OS function jumps back with the value &FF in register A. Else A contains an error code.

Further it can be usefull to test the result bytes of the FDC, they can be found in RAM beginning at system variable FDC_RES.

The system variable REG_PC+1 contain the number of the drive (source of the file) after the return of the OS function. 0-12 equals A-M.

Attention: Check the 7 result bytes of the FDC for maximum security.

LOAD A TAGGED FILE TO MAIN MEMORY AT ADDRESS &0000

Short description: Load a tagged file to the main memory (first 64 KB) at address &0000 (standard start address of many programs).

Label: LADEN

ROM-number: C

Start address: &FD8F

Jump in conditions: Write value &00 to the system variable REG08_3.
Other system variables must contain correct values, DIRectories must have been read. At least one file must be tagged.
A < &FF => normal function.
A = &FF => status of file will NOT be changed (remains tagged)!

Jump back conditions: File has been loaded at address &0000 in the first 64 KB RAM (main memory). Its 128 byte header is written to &BC00 in RAM.

System variable REG_PC+1 contains the number of the source drive (from 0 to 12 which equals drive A ... M).

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the system RAM variables REG08_0, 1, 5, REG16_0, REG_PC. Further the RAM regions &B000-&B7FF and &BC00-&BC7F, the RAM state, the FDC, the drive motor, the file-tagging-bytes.

Description: This OS function is used to load a tagged file into the main memory. It you have tagged a file in the Desktop (see drive-tagging-bytes), this OS function can load the first tagged file. The file will be automatically loaded at the RAM address &0000. But don't load a file which is longer than 40 KB (maximal 44 KB!). Longer files will overwrite important parts of the OS. Have a look at the memory map in files #OS-VAR.ENG and #EQU-API.ENG. The file header will be placed at address &BC00 in the RAM.

If an error occurs (for example if there is no tagged file), this OS function displays an error message and jumps back to the Desktop.

Attention: The DIR of the source drive must have been read & buffered in RAM. A file must have been marked. Don't load files greater than 40 KB (maximum 44 KB).
Beware: If Z80 register A contains the value &FF, before you call this OS function, then the file-tagging-byte / the status of the source file will NOT be changed. So the same source file can / will be used again for the next file operation.

LOAD A TAGGED FILE TO MAIN MEMORY AT ANY ADDRESS

Short description: Load a tagged file to main memory (first 64 KB).

Label: LADEN

ROM-number: C

Start address: &FD8F

Jump in conditions: Write value &02 to the system variable REG08_3.
REG16_1 = Target address of file in the first 64 KB, must be dividable by 256. Format of target address &XX00. Examples &2300, &7800, &C000.
Other system variables must contain correct values, DIRectories must have been read. At least one file must be tagged.
A < &FF => normal function.
A = &FF => status of file will NOT be changed (remains tagged)!

Jump back conditions: File has been loaded beginning at target address from REG16_1 in the first 64 KB RAM (main memory). Its 128 byte header is written to &BC00 in RAM.
System variable REG_PC+1 contains the number of the source drive (from 0 to 12 which equals drive A ... M).

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the system RAM variables REG08_0, 1, 5, REG16_0, REG_PC. Further the RAM regions &B000-&B7FF and &BC00-&BC7F, the RAM state, the FDC, the drive motor, the file-tagging-bytes.

Description: This OS function is used to load a file to the main memory. The target address can be choosen relatively freely, it must be provided in the system RAM variable REG16_1. But it must be dividable by 256, that means it has the format &XX00. (Look before).

If you load a file at address &C000 - into the screen memory - then it can contain up to 60 KB. Because if LADEN reads over address &FFFF it will continue at address &0000. You should know which RAM regions are used by the OS and which of them you can overwrite. Have a look at the memory map in files #OS-VAR.ENG and #EQU-API.ENG. The file header will be placed at address &BC00 in the RAM.

If an error occurs (for example if there is no tagged file), this OS function displays an error message and jumps back to the Desktop.

Attention: The DIR of the source drive must have been read & buffered in RAM. A file must have been marked. Don't load files greater than 40 KB (maximum 60 KB).
Beware: If Z80 register A contains the value &FF, before you call this OS function, then the file-tagging-byte / the status of the source file will NOT be changed. So the same source file can / will be used again for the next file operation.

LOAD a TAGGED FILE to EXPANSION RAM at BLOCK &7FC4 - Address &4000

Short description: Load a tagged file to the expansion RAM block &7FC4 at address &4000 (standard start address of many E-RAM programs).

Label: LADEN

ROM-number: C

Start address: &FD8F

Jump in conditions: Write value &01 to the system variable REG08_3.
Other system variables must contain correct values, DIRectories must have been read. At least one file must be tagged.
A < &FF => normal function.
A = &FF => status of file will NOT be changed (remains tagged)!

Jump back conditions: File has been loaded at block &7FC4 at address &4000 in the expansion RAM.
System variable REG_PC+1 contains the number of the source drive (from 0 to 12 which equals drive A ... M).

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the system RAM variables REG08_0, 1, 5, REG16_0, REG_PC. Further the RAM region from &B000 to &B7FF, the RAM state, the FDC, the drive motor, the file-tagging-bytes.

Description: OS function loads a file to the expansion RAM of the CPC.

Loading begins at the first 16 KB expansion RAM block &7FC4 of the CPC and its target address is &4000. Any file-header will NOT be removed, it will remain at the beginning of the file (E-RAM &7FC4, &4000).

The User must care if the target expansion RAM is existing, that means if a RAM expansion is connected. So investigate the system variables XRAM_C4 .. XRAM_FF and X4RAM.

If an error occurs (for example if there is no tagged file), this OS function displays an error message and jumps back to the Desktop.

Attention: The DIR of the source drive must have been read & buffered in RAM. A file must have been marked. Don't load files greater than 40 KB (maximum 60 KB).

Beware: If Z80 register A contains the value &FF, before you call this OS function, then the file-tagging-byte / the status of the source file will NOT be changed. So the same source file can / will be used again for the next file operation.

LOAD A TAGGED FILE TO THE EXPANSION RAM AT ANY ADDRESS

Short description: Load a tagged file to the expansion RAM. Start block and address can be choosen freely.

Label: LADEN

ROM-number: C

Start address: &FD8F

Jump in conditions: Write value &03 to the system variable REG08_3.
REG16_1 = Target address of file in expansion RAM, must be dividable by 256. Format of target address &XX00. Examples: &4000, &4300, &7800.
AKT_RAM = Target expansion RAM block from &7FC4 up to &78FF.
Other system variables must contain correct values, DIRectories must have been read. At least one file must be tagged.
A < &FF => normal function.
A = &FF => status of file will NOT be changed (remains tagged)!

Jump back conditions: File has been loaded beginning at expansion RAM block from AKT_RAM and target address from REG16_1 in the expansion RAM.
System variable REG_PC+1 contains the number of the source drive (from 0 to 12 which equals drive A ... M).

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the system RAM variables REG08_0, 1, 5, REG16_0, REG_PC. Further the RAM region between &B000 and &B7FF, the RAM state, the FDC, the drive motor, the file-tagging-bytes.

Description: OS function loads a file to the expansion RAM of the CPC at any expansion RAM block(s) and any address (between &4000 and &7FFF).
Any file-header will NOT be removed, it will remain at the start of the file.
The User must care if the target expansion RAM is existing, that means if a RAM expansion is connected. So investigate the system variables XRAM_C4 .. XRAM_FF and X4RAM.
If an error occurs (for example if there is no tagged file), this OS function displays an error message and jumps back to the Desktop.
If you want to load bigger files (some 100 KB) you must use the expansion RAM of the CPC. To define the target address of such a file you have to select the first physical target 16 KB exp. RAM block in system variable AKT_RAM. Use values between &7FC4, &7FC5 .. &78FF. This covers a region of 4 MB. Further system variable REG16_1 must contain the target address (from (&0000)/&4000 up to &7F00). The target address must be dividable by 256, so its format is &XX00. Else errors will occur. If the target address is between &0000 and &3FFF then first the main memory will loaded up to &3FFF, then the exp. RAM will be loaded.

Attention: The DIR of the source drive must have been read & buffered in RAM. A file must have been marked. Don't load files greater than 40 KB (maximum 60 KB).
Beware: If Z80 register A contains the value &FF, before you call this OS function, then the file-tagging-byte / the status of the source file will NOT be changed. So the same source file can / will be used again for the next file operation.

SAVE ALL Z80 REGISTERS AND JUMP TO FutureOS MONITOR

Short description: This is a part of the FutureOS system monitor. All Z80 registers will be saved in RAM. OS function doesn't jump back. Use it for debugging purposes.

Label: CARET

ROM-number: C

Start address: &FD62

Jump in conditions: -

Jump back conditions: NO return.

Manipulated: The system RAM variables: REG_AF1, REG_BC1...REG_R, REG_I will get the contents of the Z80 processor registers.

Description: This OS function is a part of the FutureOS system monitor. It is usefull when you want to debug a program. If you jump to CARET the contents of all Z80 processor registers (but PC) will be saved to the corresponding system RAM variables REG_AF1, REG_BC1 ... REG_R, REG_I. The content of Z80 register R will be corrected in that way, that an interrupted program will NOT sense the jump to CARET (when you use the system monitor to continue the original program).

CARET doesn't jump back, it jumps directly to the FutureOS system monitor. From that point you can continue the program which jumps to CARE.

If you want to test a OS function, you can PUSH the address of CARET onto the stack. Then call the OS function by a JP instruction. If the OS function ends with an RET instruction, this will automatically lead you to the system monitor (entry point CARET). There you can investigate all Z80 registers.

Attention: Also the Z80 register R will be handled correct.
The OS function CARET donesn't jump back.

DELETE FILE FROM DIR, SET ENTRY TO &E5

Short description: Deletes file in buffered DIRectory.

Label: EWEG

ROM-number: C

Start address: &FD7A

Jump in conditions:
REG08_1 = Drive-number, from whose DIR the file should be erased.
Beginning at system RAM variable REG16_8 12 bytes contain the User number (1 byte), file-name (8 bytes) and the extension (3 bytes) of the file that should be eliminated.

Jump back conditions:
File has been deleted in the RAM buffered DIRectory.
The DIR is now unsorted!
TURBO_? of the corresponding drive is marked as MANIPULATED (bit 3).
The 32 byte DIR of the drive in banked in (&4000-&7FFF), but system variable AKT_RAM is NOT changed.

Manipulated: AF, BC, DE, HL, XH and the RAM state.

Description: If you (or the OS) want to save a file to disc, maybe an older version (same user, name) of the file already exists on disc.

This OS function deletes a file from the (RAM buffered) DIRectory, its defined by its drive, user, name and extension.

When calling EWEG you have to provide the drive-number (0..12 = A..M) and the UNE (User,Name,Extension) of the file.

When the EWEG has deleted the entries of a file, the DIR is unsorted.

There are some entries overwritten with byte &E5, that has been the deleted file.

Attention: EWEG marks the system variable TURBO_? of the corresponding file in bit 3. That means that the DIR of that drive is NOT sorted. So the OS function ISWS can sense such an MANIPULATED DIR and can write it back to the original disc.

GENERATE TABLE OF FREE BLOCKS (TO SAVE A FILE)

Short description: If you (or the OS) want to save a file free blocks are needed. This OS function generates a table of free blocks, there you can save data. Only for floppy disc!

Label: BBTT

ROM-number: C

Start address: &FD86

Jump in conditions: A table generated by BBTG must begin at &B600.
B = Number of wantd free blocks 1..255.
C' = Number of total blocks of a unused disc, number depends on the format of the disc.
HL = Target address for the table of free blocks.
HL should begin at the start of a 256 byte page, all block must fit into that page. HL should have the format &XX00 !!!

Jump back conditions:
Carry-Flag is set to 1 => File is too BIG.
Carry-Flag is cleared to 0 => All went well, disc has enough free space. In this case: ...
HL = End of the table of free blocks.
The end of the table is marked by a &00 byte.

Manipulated: AF, B, L and HL'

Description: If you (or the OS) want to save a file of defined lenght to disc, you need enough free blocks on the floppy disc. This OS function BBTT will create a table of a defined number of free blocks of the floppy disc.
Before you can use BBTT you have to create a block usage table, which tells which blocks of a disc are free (or used). This is done by the OS function BBTG. The block usage table must be generated beginning at address &B600. If the block &01 is part of the DIRectory, you have to mark it self as used. This is the case for DATA, SYSTEM and IBM disc formate. Block &00 is not used by BBTT.
After calling BBTG you can use this OS function BBTT. But you have to write the number of desired free blocks in register B (that depends on the file lenght). Take care: Data, System and IBM formats use 1 KB block size, but Vortex format use 4 KB block size. Further you have to fill Z80 register C' (NOT C) with the number of total blocks of the disc. This value depends on the floppy disc format and the used tracks of the disc. Further you have to load register HL with the target address of the table of free blocks. But beware! The complete table must fit into one 256 byte page, so HL should have the format &XX00.
Now you can call this OS function BBTT. The tabel will be generated.
And the success of the operation is written to the Carry flag of the Z80. If the Carry flag is set, then the floppy disc has NOT enough space for the file. Else if the Carry flag is cleared to zero, the file will fit on the disc. In this case HL points to the end of the table of free blocks.
The end of the table is marked by a &00 byte.

Attention: If you write byte &00 to Z80 register B this means that you search 256 blocks. So B shouldn't contain &00 as number of desired blocks.
Before you use BBTT you MUST have use BBTG to generate a block usage table at address &B600.
This OS function BBTT is ONLY usable for floppy disc formats.

CALCULATE NUMBER of FREE ENTRIES in a DIRECTORY (FLOPPY DISK only)

Short description: Calculates the number of free 32 byte DIRectory entries of a floppy disc. This is needed for saving a file.

Label: EFED

ROM-number: C

Start address: &FD83

Jump in conditions:
REG08_1 = Number of drive from 0 to 7 (=A-H), whose DIR should be investigated.
System RAM variables must contain valid values.
The DIRectory is allowed to be NOT sorted.

Jump back conditions:
BC = &0020
E = Numbre of free 32 byte entries of the disc.
HL = Start of the 32 byte DIRectory of the drive (data from REG08_1).
The 32 byte DIR of the drive is switched on (between &4000 and &7FFF), but the system RAM variable AKT_RAM is NOT changed.

Manipulated: AF, BC, DE, HL and the RAM state.

Description: If a file should be saved to disc, an corresponding entry must be written to the DIRectory. Bigger files need more entries.
This OS function calculates how many 32 byte entries are free in the DIR of a defined floppy disc drive.
Before you call EFED you have to load system RAM variable REG08_1 with the number (0..7) of the disc drive (A..H) where the file should be saved.
EFED will provide the number of free 32 byte DIR entries in register E and register HL will point to the start of the 32 byte DIR or the disc drive. The corresponding expansion RAM, that buffers the DIRectory, is switched on.

Attention: When the OS function jumps back the register BC contains the value &0020. This is important if you want to use the OS function EGEN afterwards. Hey, this saves 3 µs.
The changed RAM select is NOT written to system RAM variable AKT_RAM.
Beware: EFED can only used for floppy disc drives. For Dobbertin HD20 hard disc partitions you must use OS function EFED_HD (located in ROM A).

GENERATE DIRECTORY ENTRY FROM TABLE OF FREE BLOCKS (SAVE)

Short description: Generates DIRecotry entr(y/ies) from a table of free blocks, includes entr(y/ies) in DIR. Floppy disc only!

Label: EGEM or EGEN (BC must contain value &0020)

ROM-number: C

Start address: &FD80 (EGEM) bzw. &FD7D (EGEN)

Jump in conditions:
A = Number of 32 byte entries that should be generated.
BC = &0020 (ONLY if entry EGEN is used),
DE = Start of table of free blocks (made by OS function BBTT).
HL = Start of the 32 byte DIR of the target floppy disc drive.
HL'= 12 bytes: User(1),Name(8),Extension(3) of file, that should be saved. ALL 12 bytes mus be in one page (&NN??)!
YL = &00 => 1 KB block size selected (Data, System, ibm format) or
YL > &00 => 4 KB block size selected (Vortex format).
The 32 byte DIR RAM must be switched on (between &4000 and &7FFF).
The 32 byte DIR must contain enough space for all entries. (=> EFED).
The 32 byte DIR is allowed to be unsorted.

Jump back conditions: DIRectory entries have been generated.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL' and 32 byte DIR.

Description: When you want to save a file, you have to write the file name in the DIRectory. This OS function EGEM (or EGEN) is used to generate 32 byte entries for a file that has to be saved. The entries will be written in the RAM buffered 32 byte DIRectory.
Before you can call OS function EGEM you have to write the number of 32 byte entries to Z80 register A. Register DE must contain a pointer to a table consisting of free blocks of the disc. HL must point to the start of the DIRectory, where the new entries should be introduced.
That 32 byte DIR must already be banked in. Further HL' (NOT HL) must point to 12 bytes (in ONE page &XX??), that contain User, Name, and Extension of the file, whose entries should be generated. The block size of the target floppy disc format in encoded in register YL, it can be 0 (1 KB blocks) or &FF (4 KB blocks).
After calling EGEM the 32 byte DIR contains the new entries. After sorting the DIR (use ISWS) the 32 byte DIR can be written to disc.

Attention: You can use two entry points for this OS function EGEM or EGEN. EGEN is 3 ys faster, but it needs the value &0020 in register BC. When you use entry point EGEM the register BC must not contain a defined value.
BEWARE: EGEM or EGEN can only be used for floppy disc DIRectories. For the HD20 hard-disc you must use the OS function EGEN_HD of ROM A.

LEAVE APPLICATION AND RETURN TO DESKTOP

Short description: This is an entry (return) point of the OS. Use it only to leave an application with intact OS icons.

Label: FORA

ROM-number: C

Start address: &FD77

Jump in conditions: ALL icons of the Desktop must be intact, the screen format must be still 64 * 32.

Jump back conditions: NO jump back.

Manipulated: doesn't matter.

Description: If an application / program will give control back to the OS, then it can use this OS entry point. But it can use it ONLY if the icons of the upper half of the Desktop are still intact and viewable.

The screen format must still be set to 64 columns and 32 lines.

The following tasks / restaurations are done by entry point FORA:
* Switch to screen mode 2.
* Switch on both ROMs (ROM charset!).
* Show the first page of the actual 16 byte DIR buffers (if existing).
* Jump to OS entry point KLICK in ROM D and in that way the Desktop.

Attention: There is NO return / NO jump back. But you can load the RAM variables of the OK icon to have the possibility to jump back to the program. In this case the user must click the OK icon.

LOAD A FILE (PARTIALLY) TO SHORT-TIME-BUFFERS

Short description: Loads a file (if possible complete) into the short-time-buffers of the expansion RAM. Uses free 16 KB E-RAMs and buffers.

Label: TEILA (load first part of file) // TEILB (reload rest of file)

ROM-number: C

Start address: &FD74 (TEILA) // &FD71 (TEILB)

Jump in conditions:
TEILA: System variables for discs and drives must be valid!
TEILB: REG08_2,4, REG_PC+1, REG_IY, REG_SP must be UNCHANGED!!!

Jump back conditions: valid for TEILA and TEILB:
A = &00 => File has been loaded complete in the short-time-buffer.
A = &F0 => File has been loaded partially in the short-time-buffer, all such 16 KB expansion RAMs are fully loaded. Reload rest of file using OS function TEILB.
REG08_2 = 32 byte DIR RAM block of the remaining file.
REG_IY = Start of residual file in the 32 byte DIR, if exists.
REG_PC+1 = Source drive 0-12 (Floppy,hard-disc, memory drive) of file.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY, and further the system RAM variables: REG08_0....5, REG16_0,_1, REG16_6....REG32_1 (only TEILA), AKT_RAM, REG_IY, REG_SP, REG_PC and XRAM_C4..FF. Further the RAM configuration, floppy-motor-status and the file-tagging-bytes.

Description: This OS function(s) enable you to load any tagged file of any length to the short-time-buffers of the free expansion RAM of the CPC. If the short-time-buffers are not enough, only the part of the file is being loaded which will fit into all free 16 KB expansion RAM short-time-buffers. The residual part of the file can be loaded afterwards, if needed in a couple of steps.
Example: If you want to show a very long file on the screen, maybe the RAM is not sufficient to buffer the complete file. The solution of such problems is called TEILA / TEILB.
First you have to tag the desired file in file-tagging-bytes (do that in the Desktop). Then you call OS function TEILA. All free 16 KB expansion RAM blocks will be marked as short time buffers and the file will be loaded in the free expansion RAM.
If the file has been loaded completely, the OS function will returen with register A = &00. You don't have to load another part of the file. If NO tagged file was found the OS function will returen with A = &FF. If the file was to big for the RAM, only the first part of the file has been loaded and the OS function returns with A = &F0. In this case it is very important to leave the RAM variables REG08_2, _4, REG_IY, REG_PC+1 and REG_SP unchanged. So the rest of the file can be reloaded with OS function TEILB. Now you have to check register A again, to investigate if the file now has been loaded completely or if there is still a file-rest.

Attention: To use TEILB correct the system RAM variables REG08_2, 4, REG_IY and REG_PC+1 must be left UNCHANGED after the return of TEILA.

SAVE A FILE (PARTIALLY) FROM THE SHORT-TIME-BUFFERS TO DISC

Short description: One file will be saved (partially?) from the short-time-buffers.

Label: TEISI (save first part of file) // TEISK (save rest of file)

ROM-number: C

Start address: &FD56 (TEISI) // &FD59 (TEISK)

Jump in conditions:
TEISI: A = Target drive for file 0..7 for drive A..H (no HD or MM).
REG_BC1 = Number of blocks of complete file (depends on disc-format).
REG16_8 = 12 bytes: User(1), Name(8) and Extension(3) of the file.
The floppy disc motor must be running.

TEISK: The RAM variables REG_AF1, REG_DE1, REG_HL1, REG_IX and REG16_8 up to REG16_8+12 must stay unchanged since calling TEISI.

Jump back conditions: Register A informs about success of operation:
A = &FF => The file was written completely to the disc. Ok, thats it.
A = &F0 => A part of the file was written to the disc. The rest of the file must be saved using OS function TEISK.
A = &00 => No DIRectory was read from disc, ERROR!
A = &01 => The selected drive is NOT tagged, ERROR!
A = &02 => The length of the file is of 0 KB. So forget it.
A = &03 => The file is bigger than the free space on disc, ERROR!
A = &04 => The target DIRectory has not enough space for the extents.
A = &05 => No free space in expansion RAM for short-time-buffering.
A = &06 => The floppy disc is "NOT READY".

TEISI and TEISK **manipulate**: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY, the RAM variables FDC_RES, REG08_0,1, REG_DE1(low), REG_PC(low).
TEISI will also manipulate: RAM variables TURBO_A - _M, REG16_0 - _9, REG32_1, 2, REG_AF1, REG_HL1, REG_IX, the RAM between &B000 and &BFFF.
Further changed: RAM configuration, the step-rate-time and the DIRs.

Description: The OS function TEISI and TEISK are the counterpart to the OS functions TEILA and TEILB. Two for partially file loading, and two for save a file partially. TEISI / TEISK will save a file in one piece or in some pieces, depanding on the free expansion RAM.

First you have to call TEISI. It generates an entry in the DIRectory and saves the content of all short-time-buffers to floppy disc. If an error occurs, it will be reportet in register A.
If TEISI returns with value &FF in register A, then the complete file was saved. But if A holds the value &F0 then the file is longer than all existing short-time-buffers. Now a variety of system RAM variables must be conserved (look before). You can now load the rest of the file with OS function TEILB. Then this part of the file can be saved with TEISK to disc. So the "Copy" of a file can take some cycles - depending on the length of the file and the amount of expansion RAM.

OS functions TEISI and TEISK will not SAVE the DIR to disc, this can be an speed advantage when transferring more than one file.

Attention: When calling TEISI the motor of the floppy disc drive must be running. Before you call TEISI it can make sense to mark a free 16 KB E-RAM block (below the DIR buffer) as DIR buffer, because new extents will increase the amount of used DIR buffers.
After using this OS function the new DIR must be saved to disc using OS function SIDIR.

DISPLAY A PAGE OF TEXT ON THE SCREEN

Short description: One page of text will be displayed on the screen.

Label: TYSAZ(normal) / TYSZA(variable numb.of RETURNs) / TYYZ(HL!,DE!)

ROM-number: C

Start address: &FD6E (TYSAZ) // &FD6B (TYSZA) // &FD68 (TYYZ)

Jump in conditions: valid for all labels:
MAX_CRX = Number of columns (MODE 2).
MAX_CRY = Number of lines.
REG08_6 = physic.RAM-block (&7FC0,&7FC4,&7FC5...&78FF), where the text is located. This 16 KB RAM block must be banked in and must be marked as short-time-buffer.
REG16_0 = Start address of the text page (in the expansion RAM block)
REG16_1 = End of text page regarding actual RAM configuration, this means the byte after the short-time-buffer or text buffer.

also valit for TYSZA and TYYZ:
XL = Number of fetched RETURNs, means number of "used" lines in this page.

Also valid for TYYZ:
You have to load HL from RAM variable REG16_0, and DE from REG16_1.

Jump back conditions: Valid for all labels:
A = &00 => Text page was displayed properly.
A = &FE => Text page was displayed. End of text is reached!
A = &FF => Part of text page was displayed, please reload rest of the text. Then call f.e. TYSZA, which shows rest of text.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', the RAM variables REG08_6, REG16_0,1, C_POS and the RAM configuration.

Description: These OS functions are used to display a complete page of text on the screen. This text can be located everywhere in the RAM or the expansion RAM (short-time-buffers). Use the label TYSAZ as entry point to display the first page of text. If the text is located in the E-RAM (short-time-buffers), then if the text reaches the end of the block, the next block will be banked in automatically. And the rest of the text will be displayed.

Normally this OS function will return value &00 in register A, that means the actual text page was displayed completely on screen, but the end of the text was NOT reached. If the OS function reaches the absolute end of the text, then register A will return value &FE.

But if the OS function returns &FF in Z80 register A the end of the short-time-buffers (end of the expansion RAM) was reached, BUT NOT the end of the text (or end of the file). Now you have to load the rest of the file / text into the E-RAM f.e. using OS function TEILB (look before). Then you can display the rest of the actual text page on screen using OS function TYSZA.

Instead of TYSZA you can use OS function TYYZ, but in this case the content of RAM variable REG16_0 must have been written to register HL and the contento of REG16_1 must be in DE.

When using entry points TYSZA or TYYZ the register XL must contain the number of RETURNs already done by OS function TYSAZ. Normally TYSAZ loads XL automatically with the correct value, so just don't change it ;-)

Attention: Please take time and have a CLOSER look at the jump in and jump out conditions of the three OS functions.
When using TYYZ it needs more parameters than TYSZA.

MULTIPLICATION OF TWO 8 BIT VALUES WITH 16 BIT RESULT

Short description: Two 8 bit values will be multiplicated.

Label: MUL88

ROM-number: C

Start address: &FD65

Jump in conditions: The two registers H and L both contain an 8 bit value, they will be multiplicated.

Jump back conditions: HL = 16 bit result.

Manipulated: DE, HL and the flags.

Description: Since the Z80 has no direct 8 bit * 8 bit multiplication command, the programmer has to do that in software.

This OS function provides a very fast 8 bit multiplication OS function. Both 8 bit values have to be in the registers H and L. Then call MUL88, the result will be provided as 16 bit value in register HL.

This OS function doesn't care about algebraic signs.

Attention: Only multiplicate positive values between 0 and 255.

SAVE A FILE TO DISC

Short description: A defined part of the memroy will be saved to disc.

Label: SICHRE

ROM-number: C

Start address: &FD8C

Jump in conditions:
The drive motor must be spinning (OUT &FA7E,&FF). The SAVE-mode must be provided in system RAM variable REG08_3: (use ASCII char + 1)
REG08_3 = "1"+1 = &32 => Save Foreground program, data from RAM.
REG08_3 = "2"+1 = &33 => Save Background program, data from RAM.
REG08_3 = "3"+1 = &34 => Save from main memory (first 64 KB).
REG08_3 = "4"+1 = &35 => Save from expansion RAM.
REG16_6+1 = Target drive, to which data should be saved. It is defined through the corresponding ASCII character. Example: Use value &41 = "A" (or &61 = "a") for drive A.

When using SAVE-mode 3 or 4 the additional information is needed:
REG16_8 = 12 bytes, provide User-number (1 byte), File-name (8 bytes) and Extension (3 bytes).
REG_IX = Source address at which data begins (16 bit).
AKT_RAM = Source RAM block, from which should be saved: &7FC0...&78FF.
REG_IY = Length of file in KB (16 bit). Depends on RAM, up to 4 MB.

Jump back conditions: File has been saved to disc, DIRectory has been updated.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY, and the RAM variables REG08_0,1,4, REG16_0,....., REG32_1, AKT_RAM, REG_R, the RAM from &8000 up to &9FFF and from &B000 to &BFFF.

Further the RAM configuration, the step-rate-time and the DIRectories.

Description: The OS function SICHRE allows to save any part of the memory to a disc drive (and update the DIRectory).
The system RAM variable REG08_3 must contain a value that defines the SAVE mode, which defines if a previously read program or a part of the RAM should be saved.
In every case RAM variable REG16_6+1 must contain the target drive from A to L in the ASCII format, use values &41 to &4D. Look before!
If you use SAVE mode 3 or 4 additional data must be provided to define the part of the memory that should be saved.

Attention: You can only save to drives, whose DIR has been read before and whose discs are readable / WRITEable. The system variables must be correct.
The drive has to be defined as ASCII character in variable REG16_6+1.

SHOW A FutureOS FILE HEADER

Short description: Displays a FutureOS file header on screen.

Label: DHED

ROM-number: C

Start address: &FD5F

Jump in conditions: At least one file should be tagged.

Jump back conditions: File header informations were shown!
XL = &8C => The used file has no header!
XL = &B3 => File has a file header.
The screen mode was switched to 1. The screen has 32 columns and 32 lines. The lower ROM (charset!) is switched on.
The 128 byte file header is located at &B400 in RAM.
The status of the used file is NOT changed, means file-tagging-bytes.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the RAM variables REG08_0,1,5, REG16_0, REG_PC and C_POS. Further the RAM from &B000 to &B7FF and from &BC00 to &BC7F, the configuration of RAM, the screen mode(1), the FDC, the file-tagging-bytes and the drive motors.

Description: The 128 byte header of a file contains additional data under FutureOS (head line or icon, please look there).
This OS function is used to display the header of the first tagged file on the screen. This covers the following information:

- Head line, semigraphic icon or icon.
- Target address and target RAM block of file.
- Length of file.
- Start address and start RAM block of file.

To display this informations, the screen mode 1 will be used. All four colors are used, so they should be different.

Attention: If the source file has NO file header this OS function will display an error message and return to the OS.

LOAD THE FILE HEADER OF A TAGGED FILE

Short description: Load file header of first tagged file.

Label: LADAH

ROM-number: C

Start address: &FD92

Jump in conditions: At least one file should be tagged.

Jump back conditions: 512 bytes of the first tagged file are loaded to &B400 in RAM. So the file header starts exactly at this address.

Manipulated: see OS function LADEN, further variables REG08_3, REG16_1.

Description: This OS function LADAH loads the first sector (512 bytes) of a file, which contains the file header at its beginning.

The first tagged file will be used. If no file is tagged, this OS function will break with an error message and jumps then to the OS.

If you have called LADAH correctly, it will return the first sector of the tagged file at RAM address &B400. Further User number, Name and Extent of the file have been written to RAM address &B7E0.

This OS function can be used if you want to investigate the header of the first tagged file. The file status (file tagging bytes) will NOT be changed.

Attention: At least one file should be tagged.

SAVE LOWER RAM TO EXPANSION RAM (LONG TIME BUFFER)

Short description: The RAM between &0000 and &3FFF will be saved to a 16 KB expansion RAM, which will be marked as "long time buffer" (LZS).

Label: SSB0

ROM-number: C

Start address: &FD4D

Jump in conditions: One 16 KB expansion RAM block should be free.

Jump back conditions: The Z flag reports about the success:
Zero flag cleared => NO free expansion RAM, L-RAM was NOT saved!
Zero flag set => Lower RAM was correctly saved in expansion RAM
RAM variable L_RAM contains the phys. RAM block or &00 (no E-RAM free)
The lower RAM is switched on between &0000 and &3FFF

Manipulated: AF, BC, DE, HL and the variables L_RAM and XRAM_??.

Description: FutureOS often uses the lower RAM block (&0000 to &3FFF) as buffer (for example DIRWA and TS_D_IN).OS function SSB0 saves the L-RAM block to the expansion RAM if there is a 16 KB block free (or marked as short time buffer). Then this free ERAM block will be marked as "long time buffer". Further the physical RAM configuration of this free 16 KB expansoin RAM buffer will be saved to the system RAM variable L_RAM. Use RRB0 to restaurate the lower RAM (see below).

Attention: If the Z flag is returned zero, the L RAM was NOT saved.

RESTAURATE LOWER RAM FROM EXPANSION RAM (LONG TIME BUFFER)

Short description: The old content of the lower RAM (between &0000 and &3FFF) will be restaurated from the expansioin RAM. Look OS function SSB0.

Label: RRB0

ROM-number: C

Start address: &FD4A

Jump in conditions: The L-RAM should have been saved before with SSB0.

Jump back conditions: The Z flag reports about the success:
Zero flag set => Error! L-RAM wasn't saved before!
Zero flag cleared => L-RAM has been restaurated correctly.
The 16 KB expansion RAM "long time buffer" was released and marked as free E-RAM. RAM variable L_RAM was set to &00.

Manipulated: AF, BC, DE, HL, the RAM variables L_RAM and XRAM_??.

Description: The previously through SSB0 in the expansion RAM saved lower RAM (&0000-&3FFF) will be restaurated by this OS function RRB0.

The 16 KB "long time buffer" of the E-RAM will be marked as free 16 KB "short time buffer". Further the system RAM variable L_RAM will be set to the value &00.

If the zero flag is cleared when the OS function RRB0 returns, then the old L-RAM was restaurated correctly. Else if the zero flag is set, then the L-RAM hasn't been saved before (with OS function SSB0).

Attention: L-RAM should have been saved before (using SSB0).

PROTECT THE CONTENTS OF THE RAM-DISC - ADAPT XRAM_??

Short description: The part of the expansion RAM, which is used by the RAM-DISC will be protected.

Label: PRO_MD

ROM-number: C

Start address: &FD50

Jump in conditions: -

Jump back conditions: Expansion RAM blocks used by files of the RAM-DISC are marked in the XRAM_?? varibles as USER-RAM (&11).
The RAM configuration &7FCC (fifth 16 KB E-RAM block) is aktive, this block contains the DIR of the RAM disc (beginning at &4000).

Manipulated: AF, BC, DE, HL, RAM status, RAM variables XRAM_CC...FF.

Description: The FutureOS used dynamic memory management, it uses all free expansion RAM blocks.

AmsDOS and CP/M can use a part of this E-RAM for a RAM-disc. The DIR of this RAM-disc is located in block &7FCC (fifth 16 KB block) of the E-RAM at address &4000.

This OS function protects the files of the RAM-disc. This is done by calculation the highest E-RAM block used by a file of the RAM-disc.

All 16 KB E-RAM blocks below will be marked in the XRAM_?? Variables as an USER-RAM (value &11). The FutureOS doesn't access the so called USER-RAMs, therefore RAM-disc files will no be altered.
If the RAM-disc is completely filled with files the RAM variables from XRAM_CC up to XRAM_FF will contain the value &11 (USER-RAM). In this case FutureOS can only work with the 64 KB of the RAM configurations &C4, &C5, &C6 and &C7.

The OS function PRO_MD is called by the OS at the start of the system.
So the files of the RAM-disc will be protected automatically.

If you have not enough RAM, then you can erase the DIR of the RAM disc by hand and restart the system. Now FutureOS will feel quite well again, but don't fortet to backup all your files from the RAM-disc before.

Attention: Erased files of the RAM-disc - located at user &E5 – will also be protected by this OS function. If you want to discard this trash finally you can erase the files by hand (setting all block numbers to &00). Or you copy all important files to disc, then erase the DIR of the RAM-disc with byte &E5. (Then copy important files back).

CALCULATE THE NUMBER OF BLOCKS OF THE FIRST TAGGED FILE

Short description: The number of blocks of the first tagged file will be calculated.

Label: G_BLK

ROM-number: C

Start address: &FD41

Jump in conditions: At least one file should be tagged.
The system RAM variables must contain valid values.

Jump back conditions: If a tagged file was found its number of blocks will be provided in register BC. If no tagged file was found then the register HL will contain value &0000.
REG_BC1 = BC = Number of block of first tagged file.
HL = &0000 => There was NO tagged file.

Manipulated: see FMD32

Description: This OS function is used to calculate the number of blocks of the first tagged file.

When calling G_BLK its important that the system RAM variables contain valid values, especially the variables TURBO_?. If the programmer will use the conventions, then this is no problem.

This OS function will work with floppy discs in Data, System, ibm or Vortex format, also files from the HD20 hard disc are supported.

After the return of the OS function the register BC and the RAM variable REG_BC1 will both contain the number of blocks of the first tagged file.

But beware: If the register HL containt value &0000 (when the OS function returns), then there was NO tagged file.

Take care: The same file can have a different number of blocks on different formats.

An example: The file MULTITAS.X16 contains 7 KB. In this case the file has seven blocks on discs with Data, System or ibm format. But if the file is located on a disc with Vortex format it has only two blocks.

Attention: The RAM variables must contain correct values.

Calculate the NUMBER of BLOCKS of a File. DATA, SYSTEM, IBM or VORTEX

Short description: The number of blocks of a file will be calculated. Its disc must have Data, System, ibm or Vortex format.

Label: BLK_FD

ROM-number: C

Start address: &FD47

Jump in conditions: HL = Pointer to the first entry of the file.
The system RAM variables must contain valid values.

Jump back conditions: BC = &00?? = Number of blocks of source file.

Manipulated: AF, BC, DE, HL, IX and IY.

Description: This OS function is used to calculate the number of blocks of a file, whose address of its first file extent is known in the 32 byte DIR. The source file must be located on a floppy disc with Data, System, ibm or Vortex format.

Before calling BLK_FD the register HL must be loaded with the address of the first extent of the file of interest. The expansion RAM where the 32 byte DIR is buffered MUST be switched in (&4000-&7FFF).

When the OS function returns the register BC will contain the number of blocks of the file.

Take care: The same file can have a different number of blocks on different formats.

An example: File JACKDANI.ELS contains 9 KB. In this case the file has also nine blocks on discs with Data, System or ibm format. But if the file is located on a Vortex formatted disc it has only three blocks.

Attention: The file of interrest must be located on a floppy disc with Data, System, ibm or Vortex format.

CALCULATE THE NUMBER OF BLOCKS OF A FILE ON HD20 HARD DISC

Short description: The number of blocks of a file on the HD20 hard disc will be calculated.

Label: BLK_HD

ROM-number: C

Start address: &FD44

Jump in conditions: HL = Pointer to the first entry of the file.
The system RAM variables must contain valid values.

Jump back conditions: BC = &???? = Number of blocks of source file.

Manipulated: AF, BC, DE, HL, IX and IY.

Description: This OS function is used to calculate the number of blocks of a file, whose address of its first file extent is known in the 32 byte DIR. The source file must be located on the HD20 hard disc.

Before calling BLK_HD the register HL must be loaded with the address of the first extent of the file of interest. The expansion RAM where the 32 byte DIR is buffered MUST be banked in (&4000-&7FFF).

When the OS function returns the register BC will contain the number of blocks of the file.

Take care: If the same file would be located on a floppy disc, its number of blocks may differ!

Attention: The source file must be located on the HD20 hard disc.

WAIT UNTIL A KEY IS PRESSED

Short description: This OS function waits until any key is pressed.

Label: WATA

ROM-number: C

Start address: &FD38

Jump in conditions: -

Jump back conditions: The user had pressed a key. Its ASCII value + 1 is provided in Z80 register A (look H_ALLET).

Manipulated: AF, BC, DE, HL, IX, PIO and PSG.

Description: This OS function is used to wait until the user will press a key. The ASCII value + 1 will be provided in the Z80 register A. If the user press NO key, this OS function will wait forever.

But if you want to scan the keyboard you should use H_XALLET.

Attention: OS function returns AFTER a keypress.

CONVERT TWO ASCII CHARS TO ONE 8 BIT VALUE

Short description: Converts two ASCII chars into an 8 bit value.

Label: CC2N

ROM-number: C

Start address: &FD3B

Jump in conditions: HL points to an address in RAM where two ASCII chars (0-9 and A-F) are located. At (HL) there is the "Highbyte", at HL+1 is the "Lowbyte".

Jump back conditions: A = 8 bit value &00-&FF
HL = HL + &0001

Manipulated: AF, B and L

Description: This OS function CC2N is used to konvert two ASCII chars (which are written to RAM) to one 8 bit value. When calling CC2N the register HL must contain the RAM address where the first of the two ASCII chars is located (highbyte). The lowbyte is located in RAM at (HL + 1).
After the return of the OS function the register HL is increased by one and register A will contain an 8 bit value which was calculated by that formula:

```
A = (HL)*16 + (HL+1)
```

Example:

Address &4000 contains the char "7" and &4001 contains char "B".

```
LD   HL,&4000
CALL CC2N
```

Now register A contains &7B.

This OS function is only able to convert chars from "0" up to "9" and from "A" up to "F" into one 8 bit value.
Under FutureOS this OS function is for example used to convert user numbres (entered by human, consists of two ASCII chars) into one byte.

Attention: Valid ASCII chars - that can CC2N convert - are located between "0" and "9" or between "A" and "F".

ONE BYTE WILL BE DISPLAYED ON SCREEN IN FORM OF TWO ASCII CHARS

Short description: One byte will displayed on screen in hexadecimal.

Label: HAUT

ROM-number: C

Start address: &FD3E

Jump in conditions: A = Byte, that should be displayed on screen.

Jump back conditions: Two hexadicimal chars has been written on the screnn.

Manipulated: AF, BC, DE, HL, AF', IX, variables CUR_POS and REG08_0.

Description: This OS function is used to display an hexadecimal 8 bit value on the screen. The byte that should be displayed must be written to register A. The OS function then displays two hexadicimal chars on the screnn.

It is important that the screen is switched to MODE 1. Both chars will be displayed with PEN 1 (BB!) at the actual CURsor-POSition.

Attention: This OS function is made for screen mode 1. The byte will be displayed in hexadecimal numbers.

TEST IF 4 MB RAM EXPANSION CONNECTED, SET SYSTEM RAM VARIABLES

Short description: Tests how much expansion RAM (0,5 MB up to 4 MB) is connected to the CPC. Ignores the first 512 KB expansion RAM.

Label: TXR4M

ROM-number: C

Start address: &FD32

Jump in conditions: -

Jump back conditions: System variables X4RXE, X4RDC, X4RBA and X4R98 have been set according to the amount of connected expansion RAM.
The register BC contains &7FC0, the first 64 KB are switched in.

Manipulated: AF, BC, DE, HL, AF', IX, address &4000 of all 16 KB expansion RAM blocks behind RAM configuration &7EC4. Further the RAM configuration itself: First 64 KB are selected.
The first 512 KB of expansion RAM remain untouched.

Description: You can connect a 4.0 MB RAM expansion to your CPC. This expansion RAM can be used as 16 KB blocks between RAM configuration &7FC4 and &78FF.

This OS function tests which 512 KB blocks of the maximal possible 4096 KB are connected / accessible. But the first 512 KB (port &7Fxx) will be ignored.

For every connected 512 KB block the corresponding bits of the system variables X4RXE, X4RDC, X4RBA or X4R98 will be set (see file #OS-VAR.ENG). These 512 KB blocks are accessed through I/O ports &7Exx, &7Dxx, &7Cxx, &7Bxx, &7Axx, &79xx or &78xx. The first 512 KB (port &7Fxx) will be ignored, because they are managed through the XRAM_?? variables.

After the return of the OS function TXR4M all the RAM variables, concerned about managing 4 MB expansion RAM, have been set to valid values. (FutureOS calls TXR4M at the start of the OS once). Further the address &4000 of each 16 KB block, abouve the first 512 KB expansion RAM, has been manipulated: this equals the RAM configurations &7Exx, &7Dxx, &7Cxx, &7Bxx, &7Axx, &79xx and &78xx.

Attention: The first 512 KB expansion RAM (port &7Fxx) will be ignored and its contents will NOT be changed.
The expansion RAM accessible between ports &7Exx and &78xx has been manipulated in every 16 KB block at address &4000.

CONVERT PHYSICAL E-RAM SELECT IN POINTER TO XRAM-VARIABLE

Short description: Converts ths physical E-RAM selection (first 512 KB expansion RAM) in a pointer to the corresponding XRAM_C4, C5, ... , FF variable.

Label: E2XRAM

ROM-number: C

Start address: &C9D4

Jump in conditions: A = physical E-RAM select &C4, &C5, ..., &FF

Jump back conditions: HL = pointer to system variable XRAM_C4 ... FF

Manipulated: AF and HL

Description: The CPC can manage up to 4 MB expansion RAM (E-RAM). The first 512 KB play a special role. They are accessed by the I/O address &7Fxx. To switch an 16 KB E-RAM block in between &4000 and &7FFF you have to send one of 32 values to port &7Fxx (&C4, &C5, &C6, &C7, &CC, &CD, ... &FF). This value is the physical E-RAM select. For every one of this 32 E-RAM blocks the FutureOS provides its own XRAM variable, which informs about the existance and usage of this 16 KB E-RAM block.

Details can be found in the FutureOS handbook and file #OS-VAR.ENG.

Now, to convert such a physical E-RAM select in a pointer to its corresponding XRAM variable you can use this OS function E2XRAM.

To do so load the physical E-RAM select into register A, then call the OS function. After its returen the register HL points to the desired XRAM variable.

Attention: This OS function deals only the first 512 KB of the E-RAM, accessed by I/O port &7Fxx. Only for the first 32 blocks of E-RAM the system RAM provides XRAM variables. For the other 3.5 MB of possible E-RAM the FutureOS provides other variables (see #OS-VAR.ENG).

FREE A 16 KB E-RAM BLOCK OF THE FIRST 512 KB E-RAM

Short description: Free a 16 KB block of E-RAM (of first 512 KB E-RAM)

Label: FER7F

ROM-number: C

Start address: &C9CB

Jump in conditions: A = physical E-RAM select (&C4, &C5, ..., &FF)

Jump back conditions: HL = Pointer to corresponding variable XRAM_??

Manipulated: AF and HL

Description: The OS function FER7F serves to free a 16 KB block of expansion ROM (E-RAM), which was previously allocated by another OS function. It's one out of the 32 blocks, being part of the first 512 KB E-RAM (access via I/O port &7Fxx). Only for the 32 blocks (16 KB) of the first 512 KB E-RAM the FutureOS provides XRAM variables, which give information if a block is connected, if it is used and for what. They are called: XRAM_C4, _C5, up to _FF.

To physically address one of this 32 blocks the I/O port &7Fxx is used to bank in such a block between &4000 and &7FFF. Therefore you send one out of 32 values to port &7Fxx: &C4, &C5, &C6, &C7, &CC, &CD, ... &FF. This value is the physical E-RAM select. Details: See file #OS-VAR.ENG.

Now, to free previously allocated E-RAM again you place the value of the physical E-RAM select to register A, then you call this OS function FER7F. After it's returen the corresponding E-RAM is defined as free its XRAM variable and register HL also points to that system variable.

Attention: This OS function deals only with the first 512 KB E-RAM, because only for this region (&7Fxx) the system RAM provides XRAM variables. For the other possible 3.5 MB of E-RAM other variables exist.

SEARCH XRAM VARIABLES for FREE E-RAM -> CONVERT to PHYSICAL E-RAM

Short description: Search the 32 XRAM_C4..FF variables for an E-RAM which is free or marked as short time memory (KZS). Subsequently convert it to physical E-RAM select (works with first 512 KB E-RAM).

Label: KZS2E

ROM-number: C

Start address: &C9EA

Jump in conditions: The variables XRAMC4-FF must contain valid values
At least one E-RAM is free or marked as KZS

Jump back conditions: A = Physical number/select of E-RAM (&C4-&FF)
B = &7F (allows direct 'OUT (C),A' instruction)
HL = Pointer to corresponding variable XRAM_??

Manipulated: AF, B and HL

Description: The OS function KZS2E serves the first 512 KB E-RAM (access via I/O port &7Fxx) of the 4 MB max. E-RAM. For every one of the 32 blocks (16 KB in size) of the first 512 KB E-RAM the FutureOS provides an 8 bit XRAM variable. The XRAM variable provide information if a block is connected, if it is free or used and for what. They are called: XRAM_C4, C5, C6, C7, CC, CD...FF.
To physically address one of this 32 blocks the I/O port &7Fxx is used to bank in such a block between &4000 and &7FFF. Therefore you send one out of 32 values to port &7Fxx: &C4, &C5, &C6, &C7, &CC, &CD, ... &FF. This value is the physical E-RAM select. For details please see file #OS-VAR.ENG.
This OS function searches the XRAM?? system variable for either a free E-RAM or an E-RAM marked as short-time-memory (KZS). After its return the register HL points to that XRAM variable. In addition register A will contain the corresponding physical E-RAM IO select and register B will contain the value &7F to allow the instructin 'OUT (C),A'.

An example:

```
CALL KZS2E    ;Search for free (or KZS) E-RAM
OUT  (C),A    ;and bank it in between &4000 and &7FFF
```

Since this OS function will not check if E-RAM is free in general the application has to do that. See handbook in chapter E-RAM management.
If no E-RAM is free or marked as KZS an incorrect value will be provided (HL will be bigger than XRAM_FF).

Attention: There MUST be at least one free E-RAM block or one short-time-memory (KZS) E-RAM.
This OS function works with the first 512 KB E-RAM, because only for this region (&7Fxx) the system RAM provides XRAM variables.

LOAD AND DISPLAY A 17 KB CPC SCREEN OR AN COMPRESSED OCP SCREEN

Short description: A regular CPC screen (17 KB) or an picture from OCP will be loaded and displayed on screen. Screen format and mode can be dynamically changed (joystick or cursor keys).

Label: ZEIBI bzw. ZEIBJ

ROM-number: C

Startaddress: &FD2F (ZEIBI) or &FD2C (ZEIBJ)

Jump in conditions: For both entry points a file must be tagged in the file-tagging-bytes (FTB). The first tagged file will be used.
Using ZEIBI a file-header must be present (loaded) at address &B400
Using ZEIBJ register H contains the High-Byte (HH) of a pointer to the file-header of a picture file. The low-byte is always considered &00.
So the address is &HH00.

Jump back conditions: The Zero Flag will report about the success of the OS function. If the zero flag is set (Z = 1) then the picture was discplayed successfully.
But if the zero flag is cleared (Z = 0) then an error eccured or the file just was no picture.

Manipulated: AF, BC, DE, HL, AF', BC', DE', HL', IX, IY and the RAM-variables REG08_0,1,5, REG16_0, REG_PC, L_RAM and XRAM_??. Furthermore the areas of RAM from &B000 to &B7FF and from &BC00 to &BC7F, the FDC, the PIO, the PSG, the file-tagging-bytes, the Video-RAM (&C000-&FFFF) and the motor status of the floppy disc drives.

Descripton: The OS functions ZEIBI and ZEIBJ are used to load and display a picture on the screen. The joystick or the cursor keys allow to alter the screen mode (right, left) or the screen format (up, down) in a dynamic way.
Before you use this OS functions there must either be a file-header be loaded to address &B400 (ZEIBI) or the register H must contain the High-Byte of an address of such an header (ZEIBJ). When loading the file-header using OS function 'LADEN' the register A should contain &FF to conserve the FTB. The file-header has to be loaded to the beginning of a 256 bytes page, so its address is &XX00. Furthermore the screen file must be tagged in the FTBs as first file (please see handbook).
When calling one of these OS functions they will first check if the file is a picture. If yes, it will be loaded and if needed it will be decompressed (OCP Advanced Art Studio screen), then it will be shown on screen. While the screen is shown you can change the screen MODE (move right or left) and the screen format (move up and down). The screen format can be either 80 x 25 or 64 x 32 characters (reference to MODE 2). You can use a joystick or the cursor keys. If you press Fire 1 or Copy this will end the presentation.
Both OS functions will set the zero flag to show that everything went well. Or they will clear the zero flag in case the file was no screen or in case of any other error.

Attention: There must be a file-header at address &B400 and the corresponding file must be tagged. Else there will be an error message provided in F (zero flag cleared).

The newest version of this file (API-C-EN.DOC) can be downloaded at:

http://www.FutureOS.de