

QNICE monitor function documentation

Auto generated

December 28, 2016

Chapter 1

CHR

1.1 CHR\$TO_LOWER

CHR\$TO_LOWER expects a character to be converted to lower case in R8

1.2 CHR\$TO_UPPER

CHR\$TO_UPPER expects a character to be converted to upper case in R8

Chapter 2

DBG

2.1 DBG\$DISASM

DBG\$DISASM disassembles a single instruction pointed to by R8. R8 will be incremented according to the addressing modes used in the instruction.

R8: Contains the address of the instruction to be disassembled

R8 WILL BE CHANGED BY THIS FUNCTION!

Chapter 3

FAT32

3.1 FAT32\$CALL_DEV

FAT32\$CALL_DEV calls a device management function

INPUT: R8, R9 are the parameters to the function

R10 is the function index

R11 is the mount data structure (device handle)

OUTPUT: R8 is the return value from the function

3.2 FAT32\$CD

FAT32\$CD changes the current directory

This function supports traversing deeply in one call using nested directories like "dir1/dir2/dir3/dir4". Any non-zero separator char can be used. It is important to pass a separator char, even if you do not plan to pass nested directories. If in doubt, just pass 0x002F, which is the ASCII code for / (passing 0x0000 also leads to 0x002F being used).

A separator char as the very first char (e.g. "/dir1/dir2") means, that we start searching from the root directory. No separator char at the beginning means, that we work relative to the current directory. The FAT32 typical "." and ".." work as expected.

FAT32 is case preserving but not case sensitive.

INPUT: R8 points to a valid device handle

R9 points to a zero terminated directory string (path)

R10 separator char (if zero, then "/" will be used)

OUTPUT: R8 still points to the directory handle

R9 0, if OK, otherwise error code

3.3 FAT32\$CD_OR_OF

FAT32\$CD_OR_OF changes the current directory or opens a file
This is an internal help function, because changing a directory and opening a file requires very similar actions. See the documentation for FAT32\$CD and FAT32\$FILE_OPEN for details.

INPUT: R8 points to a valid device handle

R9 points to a zero terminated string (directory or filename)

R10 separator char (if zero, then "/" will be used)

R11 operation mode: 0 = FAT32\$CD, otherwise FAT32\$FILE_OPEN

in the latter case, R11 points to the file handle structure

that will be filled by FAT32\$CDOF.

OUTPUT: R9 0, if OK, otherwise error code

all other registers remain unchanged

3.4 FAT32\$CHECKSUM

FAT32\$CHECKSUM computes a directory entry checksum

Used to confirm the binding between a long filename and its corresponding short filename and used to detect long filename orphans. Algorithm:

Sum = 0;

for (FcbNameLen=11; FcbNameLen!=0; FcbNameLen--)

// NOTE: The operation is an unsigned char rotate right

Sum = ((Sum & 1) ? 0x80 : 0) + (Sum << 1) + *pFcbName++;

INPUT: R8: pointer to the 11 bytes of a short name

OUTPUT: R8: 1 unsigned byte checksum (upper byte of R8 is zero)

3.5 FAT32\$DIR_LIST

FAT32\$DIR_LIST is used to browse the currently active directory

This function is meant to be called iteratively to return one directory entry after the other. It supports long filenames. The implementation supports a maximum of 65.535 files within one folder.

INPUT: R8 points to a valid directory handle (created by FAT32\$DIR_OPEN)

R9 points to an empty directory entry structure, having the size of FAT32\$DE_STRUCT_SIZE, that will be filled by this function.

R10 or-ed list of FAT32\$FA_* flags to filter for certain types:

if the attribute is not set, then an entry having this flag

will not be browsed (i.e. it will be hidden). Use

FAT32\$FA_DEFAULT, if you want to browse for all non hidden

files and directories.

OUTPUT: R8 still points to the directory handle

R9 points to the now filled directory entry structure

R10 1, if the current entry is a valid entry and therefore another iteration (i.e. call to FAT32\$DIR_LIST) makes sense

0, if the current entry is not valid and therefore the end of the directory has been reached (R11 is 0 in such a case)

R11 0, if OK, otherwise error code

3.6 FAT32\$DIR_OPEN

FAT32\$DIR_OPEN opens the current directory for browsing

Call this function, before working with FAT32\$DIR_LIST. Directly after mounting a device, the "current directory" equals the root directory.

INPUT: R8 points to a valid device handle

R9 points to an empty directory handle struct that will be filled (use FAT32\$FDH_STRUCT_SIZE to reserve the memory)

OUTPUT: R8 points to the filled directory handle structure, i.e. it points to where R9 originally pointed to

R9 0, if OK, otherwise error code

3.7 FAT32\$FILE_OPEN

FAT32\$FILE_OPEN opens a file for reading or writing or both

You can either pass a simple file name or a complex nested path where the last path segment is interpreted as a file name.

Consult the documentation of FAT32\$CD to learn more about the meaning and handling of the separator char.

INPUT: R8 points to a valid device handle

R9 points to an empty file handle struct that will be filled (use FAT32\$FDH_STRUCT_SIZE to reserve the memory)

R10 points to a zero terminated filename string (path)

R11 separator char (if zero, then "/" will be used)

OUTPUT: R8 still points to the same handle

R9 points to the same address but this is now a valid/filled hndl

R10 0, if OK, otherwise error code

3.8 FAT32\$FILE_RB

FAT32\$FILE_RB reads one byte from an open file

The read operation takes place at the current internal "seek position" within the file, i.e. subsequent calls to FILE_RB will result in reading byte per byte.

INPUT: R8 points to a valid file handle

OUTPUT: R8 still points to the file handle

R9 low byte = currently read byte; high byte = 0

R10 0, if the read operation succeeded

FAT32\$EOF, if the end of file has been reached; in this case

the value of R9 is 0 (means "undefined" here)

any other error code in case of an error

3.9 FAT32\$FILE_SEEK

FAT32\$FILE_SEEK positions the read/write pointer within an open file

INPUT: R8 points to a valid file handle

R9 LO word of seek position

R10 HI word of seek position

OUTPUT: R8 still points to file handle

R9 0, if OK, otherwise error code

3.10 FAT32\$MOUNT

FAT32\$MOUNT mounts a FAT32 file system on arbitrary hardware

The abstraction requires 5 functions to be implemented: Read and write a 512-byte-sized sector using LBA mode. Read and write a byte from within a buffer that contains the current sector. Reset the device. The function signatures and behaviour needs to be equivalent to the SD card functions that are part of the library `sd_library.asm`. You need to pass pointers to these functions to the mount function call in the mount initialization structure.

All subsequent calls to FAT32 functions expect as the first parameter a pointer to the mount data structure (aka device handle) that is being generated during the execution of this function. With this mechanism, an arbitrary amount of filesystems can be mounted on an arbitrary amount and type of hardware.

INPUT: R8: pointer to the mount initialization structure that is build up in the following form. Important: The structure is 18 words large, that

means that a call to `FAT32$MOUNT` will append more words to the structure than the ones, that have to be pre-filled before calling `FAT32$MOUNT`:

word #0: pointer to a device reset function, similar to `SD$RESET`

word #1: pointer to a block read function, similar to `SD$READ_BLOCK`

word #2: pointer to a block write function, similar to `SD$WRITE_BLOCK`

word #3: pointer to a byte read function, similar to `SD$READ_BYTE`

word #4: pointer to a byte write function, similar to `SD$WRITE_BYTE`

word #5: number of the partition to be mounted (0x0001 .. 0x0004)

word #6 .. word #18 : will be filled by `FAT32$MOUNT`, their layout is as described in the `FAT32$DEV_*` constants beginning from index #7 on.

For being on the safe side: Instead of hardcoding "18" as the size of the whole mount data structure (device handle) use the constant `FAT32$DEV_STRUCT_SIZE` instead.

OUTPUT: R8 is preserved and still points to the structure that has been filled by the function from word #6 on.

R9 contains 0, if everything went OK, otherwise it contains the error code

3.11 `FAT32$MOUNT_SD`

`FAT32$MOUNT_SD` mounts a SD card partition as a FAT32 file system

Wrapper to simplify the use of the generic `FAT32$MOUNT` function. Read the documentation of `FAT32$MOUNT` to learn more.

INPUT: R8 points to a 18 word large empty structure. This structure will be filled by the mount function and it therefore becomes the device handle that you need for subsequent FAT32 function calls. For being on the safe side: Instead of hardcoding "18", use the constant `FAT32$DEV_STRUCT_SIZE` instead.

R9 partition number to mount (1 .. 4)

OUTPUT: R8 points to the handle (identical to the input value of R8)

R9 contains 0 if OK, otherwise the error code

3.12 `FAT32$PRINT_DE`

`FAT32$PRINT_DE` is a pretty printer for directory entries

Uses monitor (system) stdout to print. Allows the configuration of the amount of data that shall be printed: Filename only is the minimum.

Additionally attributes, file sizes, file date and file time can be shown.

The printed layout is as follows:

```
¡DIR¿ HRSA BBBBBBBBBB YYYY-MM-DD HH:MM name...
```


;DIR; means that the entry is a directory, otherwise whitespace
 H = hidden flag
 R = read only flag
 S = system flag
 A = archive flag
 BBBBBBBBBB = decimal size of the file in bytes
 YYYY-MM-DD = file date
 HH:MM = file time
 name... = file name in long file format
 INPUT: R8: pointer to directy entry structure
 R9: print flags as defined in FAT32\$PRINT_SHOW_*

3.13 FAT32\$READ_B

FAT32\$READ_B reads a byte from the current sector buffer
 INPUT: R8: pointer to mount data structure (device handle)
 R9: address (0 .. 511)
 OUTPUT: R10: the byte that was read

3.14 FAT32\$READ_DW

FAT32\$READ_DW reads a double word from the current sector buffer
 Assumes that the buffer is stored in little endian (as this is the case
 for MBR and FAT32 data structures)
 INPUT: R8: pointer to mount data structure (device handle)
 R9: address (0 .. 511)
 OUTPUT: R10: the low word that was read
 R11: the high word that was read

3.15 FAT32\$READ_FDH

FAT32\$READ_FDH fills the read buffer according to the current index in FDH
 If the index within FDH is ; FAT32\$SECTOR_SIZE (512), then it is assumed,
 that no read operation needs to be performed, i.e. another function has
 already filled the 512-byte read-buffer. Otherwise, the sector is
 increased (and the index is reset to 0) and if necessary also the cluster
 is increased (and the sector is reset to 0). The new index, sector and
 cluster values are stored within the FDH (file and directory handle).

In case of an increased index or sector or cluster value, the 512-byte read-buffer is re-read for subsequent read accesses.

The above-mentioned "assumed that no read operation needs to be performed" is only true, if the FDH, which was originally responsible for filling the hardware buffer is the same, as the one who is currently active. This is checked by evaluating FAT32\$DEV_BUFFERED_FDH.

INPUT: R8: FDH

OUTPUT: R8: FDH

R9: 0, if OK, otherwise error code

3.16 FAT32\$READ_SIC

FAT32\$READ_SIC reads a sector within a cluster

INPUT: R8: device handle

R9: LO word of cluster

R10: HI word of cluster

R11: sector within cluster

OUTPUT: R8: device handle

R9: 0, if OK, otherwise error code

3.17 FAT32\$READ_W

FAT32\$READ_W reads a word from the current sector buffer

Assumes that the buffer is stored in little endian (as this is the case for MBR and FAT32 data structures)

INPUT: R8: pointer to mount data structure (device handle)

R9: address (0 .. 511)

OUTPUT: R10: the word that was read

Chapter 4

IO

4.1 IO\$DUMP_MEMORY

IO\$DUMP_MEMORY prints a hexadecimal memory dump of a specified memory region.

R8: Contains the start address

R9: Contains the end address (inclusive)

The contents of R8 and R9 are preserved during the run of this function.

4.2 IO\$GETCHAR

IO\$GETCHAR reads a character either from the first UART in the system or from an

attached USB keyboard. This depends on the setting of bit 0 of the switch register.

If SW[0] == 0, then the character is read from the UART, otherwise it is read from

the keyboard data register.

R8 will contain the character read in its lower eight bits.

4.3 IO\$GETS

IO\$GETS reads a zero terminated string from STDIN and echos typing on STDOUT

ALWAYS PREFER IO\$GETS_S OVER THIS FUNCTION!

It accepts CR, LF and CR/LF as input terminator, so it directly works with various terminal settings on UART and also with keyboards on PS/2 ("USB"). Furtheron, it accepts BACKSPACE for editing the string.
R8 has to point to a preallocated memory area to store the input line

4.4 IO\$GETS_CORE

IO\$GETS_CORE implements the various gets variants.
Refer to the comments for IO\$GETS, IO\$GET_S and IO\$GET_SLF
R8 has to point to a preallocated memory area to store the input line
R9 specifies the size of the buffer, so (R9 - 1) characters can be read;
if R9 == 0, then an unlimited amount of characters is being read
R10 specifies the LF behaviour: R10 = 0 means never add LF, R10 = 1 means: add a LF when the input is ended by a key stroke (LF, CR or CR/LF) in contrast to automatically ending due to a full buffer

4.5 IO\$GETS_S

IO\$GETS_S reads a zero terminated string from STDIN into a buffer with a specified maximum size and echos typing on STDOUT
It accepts CR, LF and CR/LF as input terminator, so it directly works with various terminal settings on UART and also with keyboards on PS/2 ("USB"). Furtheron, it accepts BACKSPACE for editing the string.
A maximum amount of (R9 - 1) characters will be read, because the function will add the zero terminator to the string, which then results in R9 words.
R8 has to point to a preallocated memory area to store the input line
R9 specifies the size of the buffer, so (R9 - 1) characters can be read;
if R9 == 0, then an unlimited amount of characters is being read

4.6 IO\$GETS_SLF

IO\$GETS_SLF reads a zero terminated string from STDIN into a buffer with a

specified
maximum size and echos typing on STDOUT. A line feed character is added to the string in case the function is ended not "prematurely" by reaching the buffer size, but by pressing CR or LF or CR/LF. It accepts CR, LF and CR/LF as input terminator, so it directly works with various terminal settings on UART and also with keyboards on PS/2 ("USB"). Furtheron, it accepts BACKSPACE for editing the string. A maximum amount of (R9 - 1) characters will be read, because the function will add the zero terminator to the string, which then results in R9 words. R8 has to point to a preallocated memory area to store the input line. R9 specifies the size of the buffer, so (R9 - 1) characters can be read; if R9 == 0, then an unlimited amount of characters is being read

4.7 IO\$GET_W_HEX

IO\$GET_W_HEX reads four hex nibbles from stdin and returns the corresponding value in R8. Illegal characters (not 1-9A-F or a-f) will generate a bell signal. The only exception to this behaviour is the character 'x' which will erase any input up to this point. This has the positive effect that a hexadecimal value can be entered as 0x.... or just as

4.8 IO\$PUTCHAR

IO\$PUTCHAR prints a single character.
R8: Contains the character to be printed
The contents of R8 are being preserved during the run of this function.

4.9 IO\$PUTS

IO\$PUTS prints a null terminated string.
R8: Pointer to the string to be printed. Of each word only the lower eight bits will be printed. The terminating word has to be zero.

The contents of R8 are being preserved during the run of this function.

4.10 IO\$PUT_CRLF

IO\$PUT_CRLF prints actually a LF/CR (the reason for this is that curses on the

MAC, where the emulation currently runs, has problems with CR/LF, but not with LF/CR)

4.11 IO\$PUT_W_HEX

IO\$PUT_W_HEX prints a machine word in hexadecimal notation.

R8: Contains the machine word to be printed in hex notation.

The contents of R8 are being preserved during the run of this function.

4.12 IO\$TIL

IO\$TIL

Show a four nibble hex value on the TIL-display

R8: Contains the value to be displayed

Chapter 5

KBD

5.1 KBD\$GETCHAR

KBD\$GETCHAR reads a character from the USB-keyboard. R8 will contain the character read in its lower eight bits and special keys in the upper 8 bits (mutually exclusive)

Chapter 6

MEM

6.1 MEM\$FILL

MEM\$FILL fills a block of memory running from the address stored in R8. R9 contains the number of words to be written. R10 contains the value to be stored in the memory area.

6.2 MEM\$MOVE

MEM\$MOVE moves the memory area starting at the address contained in R8 to the area starting at the address contained in R9. R10 contains the number of words to be moved.

Chapter 7

MISC

7.1 MISC\$EXIT

MISC\$EXIT

Exit a program and return to the QNICE monitor

7.2 MISC\$WAIT

MISC\$WAIT

Waits for several milliseconds, controlled by R8. This is just a stupid wait loop as we as of now do not have hardware timers and interrupts.

R8: Contains delay value

Chapter 8

MTH

8.1 MTH\$DIVS

MTH\$DIVS performs a signed 16 / 16 division of the form
R11 = R8

8.2 MTH\$DIVU

MTH\$DIVU performs an unsigned 16 / 16 division of the form
R11 = R8

8.3 MTH\$DIVU32

MTH\$DIVU32 divides 32bit dividend by 32bit divisor and returns
a 32bit quotient and a 32bit modulo
warning: no division by zero warning; instead, the function
returns zero as result and as modulo
INPUT: R8/R9 = LO—HI of unsigned dividend
R10/R11 = LO—HI of unsigned divisor
OUTPUT: R8/R9 = LO—HI of unsigned quotient
R10/R11 = LO—HI of unsigned modulo

8.4 MTH\$MULS

MTH\$MULS performs a signed 16 x 16 multiplication of the form

$R11(H)/R10(L) = R8 * R9$. It is merely an interface to the EAE.

8.5 MTH\$MULU

MTH\$MULU performs an unsigned 16 x 16 multiplication of the form
 $R11(H)/R10(L) = R8 * R9$. It is merely an interface to the EAE.

8.6 MTH\$MULU32

MTH\$MULU32 multiplies two 32bit unsigned values and returns a 64bit unsigned

INPUT: $R8/R9 = LO/HI$ of unsigned multiplicand 1

$R10/R11 = LO/HI$ of unsigned multiplicand 2

OUTPUT: $R11/R10/R9/R8 = HI .. LO$ of 64bit result

Chapter 9

SD

9.1 SD\$READ_BLOCK

SD\$READ_BLOCK reads a 512 byte block from the SD Card

INPUT: R8/R9 = LO/HI words of the 32-bit block address

OUTPUT: R8 = 0 (no error), or error code

The read data is stored inside 512 byte buffer of the the SD controller memory that can then be accessed via SD\$READ_BYTE.

IMPORTANT: 512-byte block addressing is used always, i.e. independent of the SD Card type. Address #0 means 0..511, address #1 means 512..1023, ..

9.2 SD\$READ_BYTE

SD\$READ_BYTE reads a byte from the read buffer memory of the controller

INPUT: R8 = address between 0 .. 511

OUTPUT: R8 = byte

No boundary checks are performed.

9.3 SD\$RESET

SD\$RESET resets the SD Card

R8: 0, if everything went OK, otherwise the error code

9.4 SD\$WAIT_BUSY

SD\$WAIT_BUSY waits, while the SD Card is executing any command

R8: 0, if everything went OK, otherwise the error code

Side effect: Starts the cycle counter (if it was stopped), but does not reset the value, so that other countings are not influenced.

9.5 SD\$WRITE_BLOCK

SD\$WRITE_BLOCK writes a 512 byte block to the SD Card

@TODO: Implement and document

9.6 SD\$WRITE_BYTE

SD\$WRITE_BYTE writes a byte to the write memory buffer of the controller

@TODO: Implement and document

Chapter 10

STR

10.1 STR\$CHOMP

STR\$CHOMP removes a trailing LF/CR from a string pointed to by R8

10.2 STR\$CMP

STR\$CMP compares two strings

R8: Pointer to the first string (S0),

R9: Pointer to the second string (S1),

R10: negative if (S0 < S1), zero if (S0 == S1), positive if (S0 > S1)

The contents of R8 and R9 are being preserved during the run of this function

10.3 STR\$H2D

STR\$H2D converts a 32bit value to a decimal representation in ASCII;

leading zeros are replaced by spaces (ASCII 0x20); zero terminator is added

INPUT: R8/R9 = LO/HI of the 32bit value

R10 = pointer to a free memory area that is 11 words long

OUTPUT: R10 = the function fills the given memory space with the decimal representation and adds a zero terminator

this includes leading white spaces

R11 = pointer to string without leading white spaces

R12 = amount of digits/characters that the actual number has,

without the leading spaces

10.4 STR\$LEN

STR\$LEN expects the address of a string in R8 and returns its length in R9

10.5 STR\$SPLIT

STR\$SPLIT splits a string into substrings using a delimiter char

Returns the substrings on the stack, i.e. after being done, you need to add the amount of words returned in R9 to the stack pointer to clean it up again and not leaving "memory leaks".

The memory layout of the returned area is:

size of string incl. zero terminator, string, zero terminator,

The strings are returned in positive order, i.e. you just need to add the length of the previous string to the returned string pointer (i.e. stack pointer) to jump to the next substring from left to right.

INPUT: R8: pointer to zero terminated string

R9: delimiter char

OUTPUT: SP: stack pointer pointer to the first string

R8: amount of strings

R9: amount of words to add to the stack pointer to restore it

10.6 STR\$STRCHR

STR\$STRCHR searches for the first occurrence of the character stored in R8 in a

string pointed to by R9.

R8: Character to be searched

R9: Pointer to the string

R10: Zero if the character has not been found, otherwise it contains a pointer to the first occurrence of the character in the string

The contents of R8 and R9 are being preserved during the run of this function

10.7 STR\$TO_UPPER

STR\$TO_UPPER expects the address of a string to be converted to upper case in R8

Chapter 11

UART

11.1 UART\$GETCHAR

UART\$GETCHAR reads a character from the first UART in the system.
R8 will contain the character read in its lower eight bits.

11.2 UART\$PUTCHAR

UART\$PUTCHAR writes a single character to the serial line.
R8: Contains the character to be printed
The contents of R8 are being preserved during the run of this function.

Chapter 12

VGA

12.1 VGA\$CHAR_AT_XY

VGA\$CHAR_AT_XY

R8: Contains character to be printed

R9: X-coordinate (0 .. 79)

R10: Y-coordinate (0 .. 39)

Output a single char at a given coordinate pair.

12.2 VGA\$CLS

VGA\$CLS

Clear the VGA-screen and place the cursor in the upper left corner.

12.3 VGA\$INIT

VGA\$INIT

VGA on, hardware cursor on, large, blinking, reset current character coordinates

12.4 VGA\$PUTCHAR

VGA\$PUTCHAR

Print a character to the VGA display. This routine automatically increments

the
X- and, if necessary, the Y-coordinate. Scrolling is implemented - if the end of
the
scroll buffer is reached after about 20 screen pages, the next character will cause
a CLS and then will be printed at location (0, 0) on screen page 0 again.
This routine relies on the stored coordinates VGA\$X and VGA\$Y which always
contain
the coordinate of the next (!) character to be displayed and will be updated
accordingly. This implies that it is possible to perform other character output
and
cursor coordinate manipulation between two calls to VGA\$PUTC without dis-
turb-
ing
the position of the next character to be printed.
R8: Contains the character to be printed.

12.5 VGA\$SCROLL_DOWN

VGA\$SCROLL_DOWN
Scroll many lines down, smartly: As VGA\$SCROLL_DOWN_1 is used in a loop,
all cases
are automatically taken care of.
R8 contains the amount of lines, R8 is not preserved

12.6 VGA\$SCROLL_DOWN_1

VGA\$SCROLL_DOWN_1
Scroll one line down

12.7 VGA\$SCROLL_HOME_END

VGA\$SCROLL_HOME_END
Uses the "_1" scroll routines to scroll to the very top ("Home") or to the very
bottom("End"). As we are looping the "_1" functions, all special cases are
automatically taken care of.
R8 = 0: HOME R8 = 1: END

12.8 VGA\$SCROLL_UP

VGA\$SCROLL_UP

Scroll many lines up, smartly: As VGA\$SCROLL_UP_1 is used in a loop, all cases

are automatically taken care of.

R8 contains the amount of lines, R8 is not preserved

12.9 VGA\$SCROLL_UP_1

VGA\$SCROLL_UP_1

Scroll one line up - this function only takes care of the display offset, NOT of the read/write offset!

R8 (input): 0 = scroll due to calculations, 1 = scroll due to key press

R8 (output): 0 = standard exit; 1 = clear screen was performed