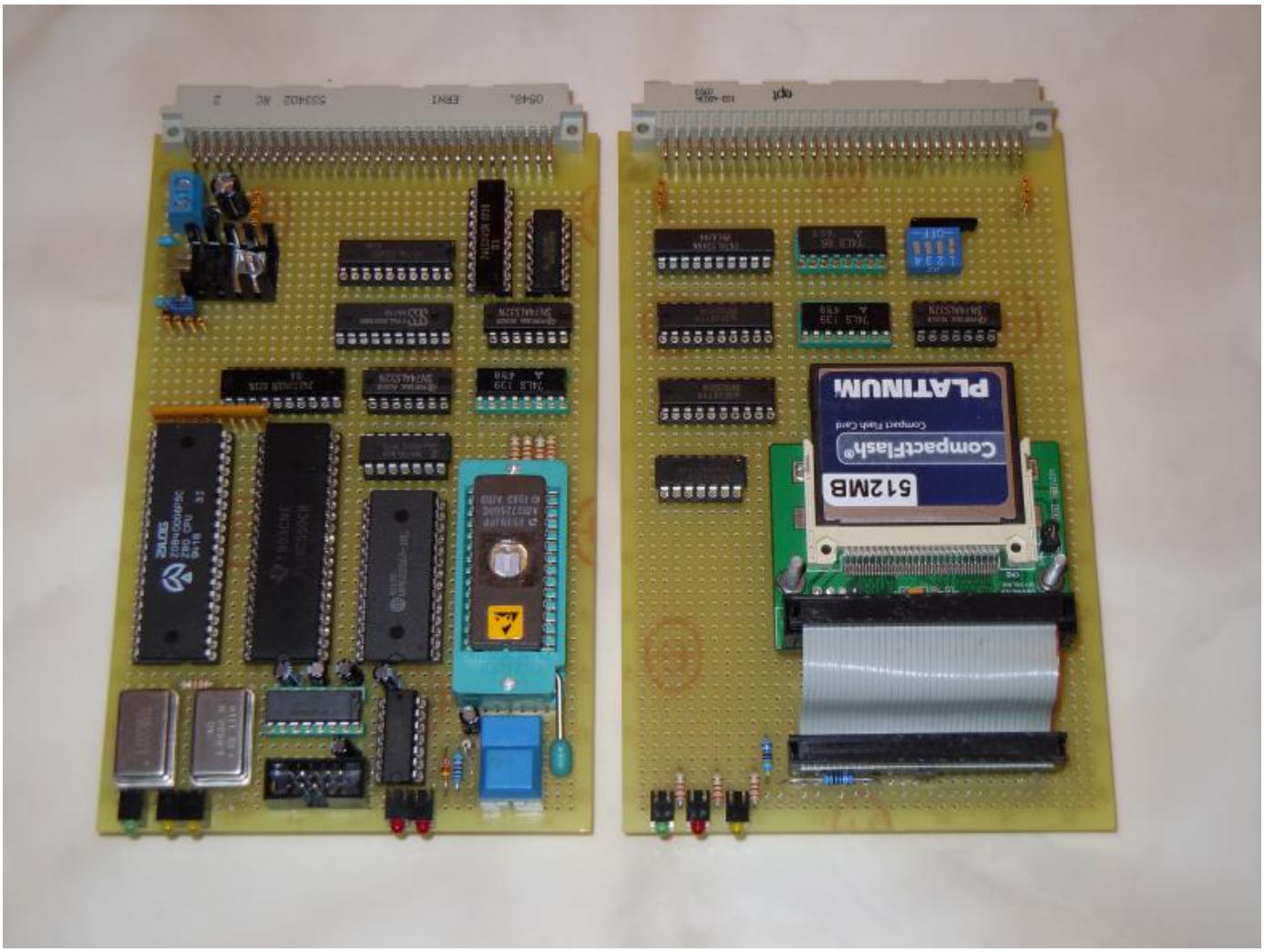
A tiny Z80 based computer

A rather small Z80 based computer with the following features:

- Z80 single board computer
- IDE controller
- FAT file system (currently read only) in Z80 assembler

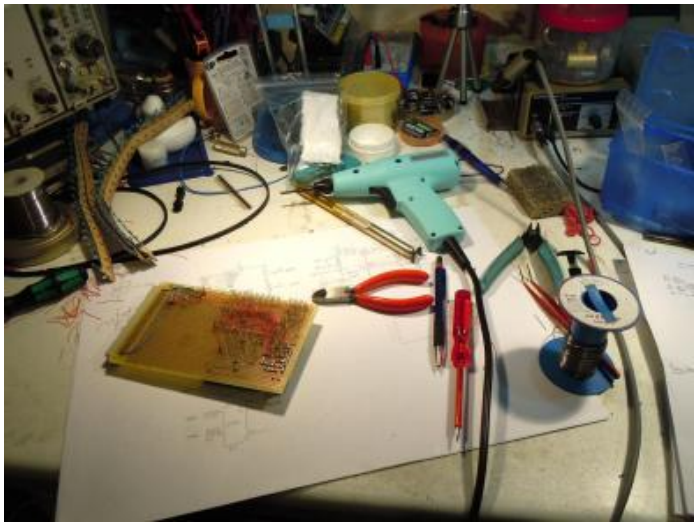
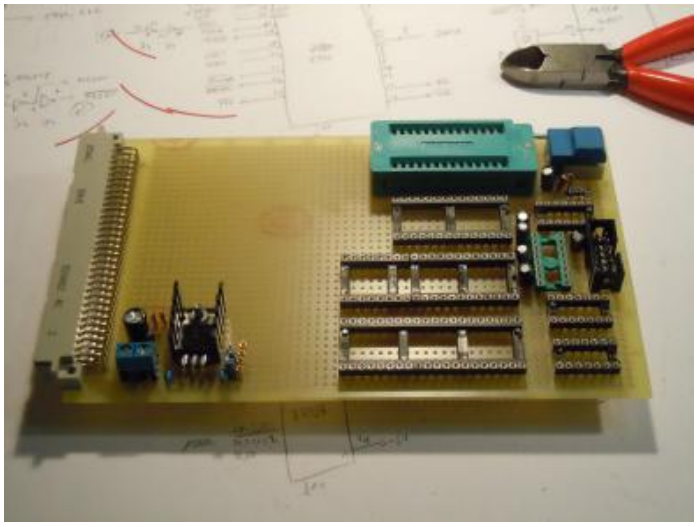
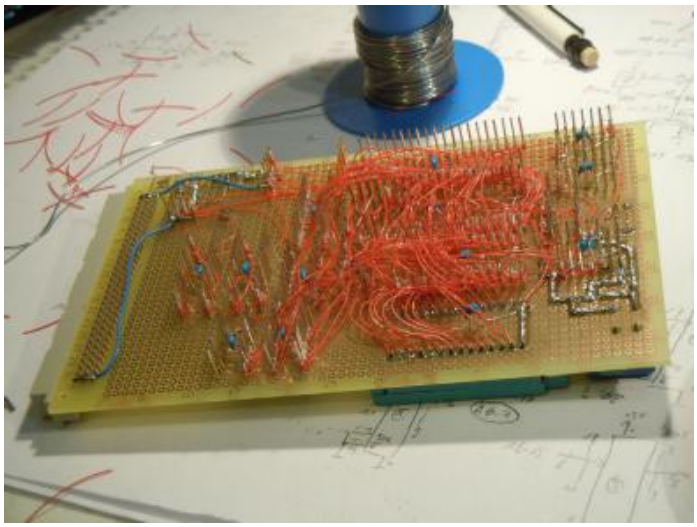
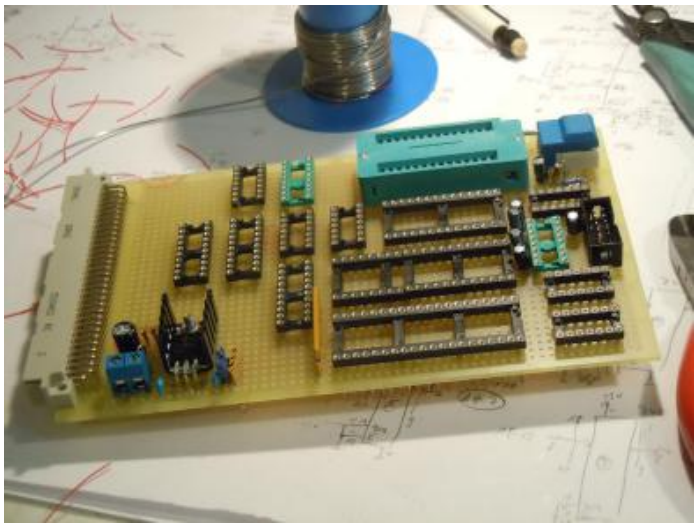
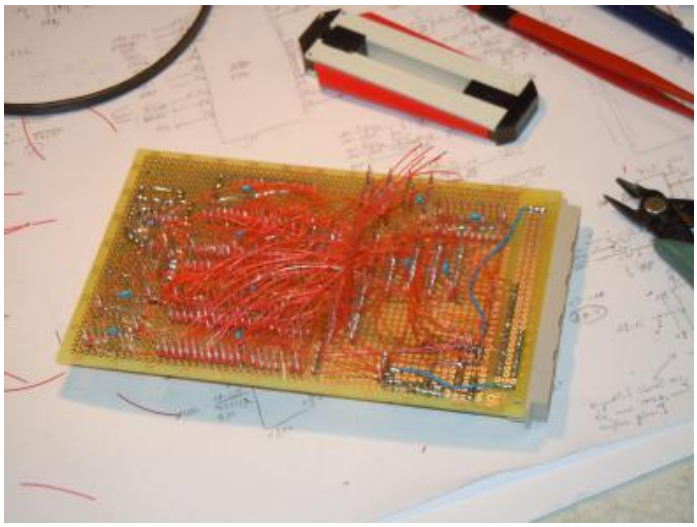
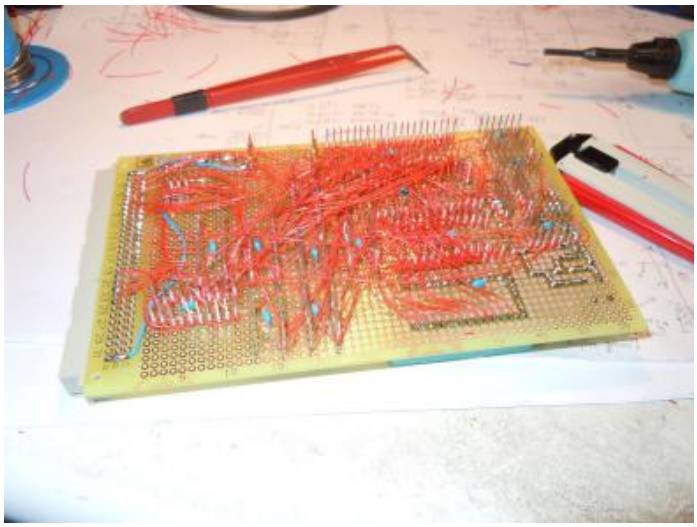


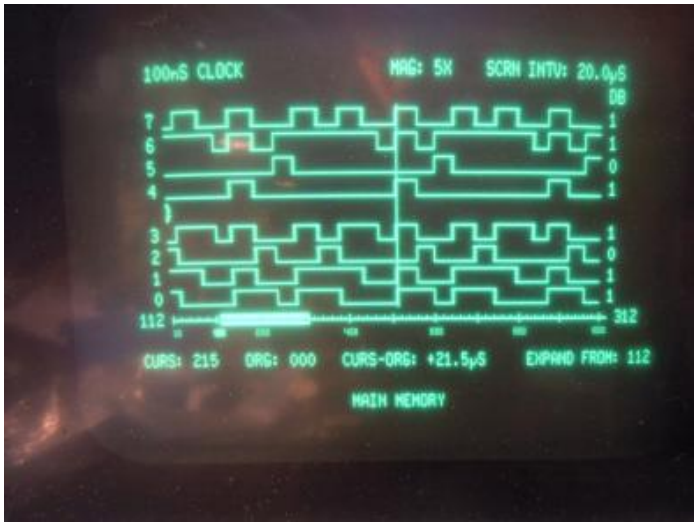
In September 2011 I somehow got the feeling that I just HAD to build a small Z80 based computer again. My last homebrew Z80 was built when I was still in school (more than 20 years ago) and somehow I felt a bit nostalgic and missed the (truly) good old times when computers were small, rather simple, easy to understand and program. Times when writing software meant writing assembler code and not installing several GB of a development package which creates files no smaller than several MB. So I searched for useable parts in my hardware repository and found everything necessary to build a small (tiny, in fact) Z80 based computer.

Since I love wire wrapping, I decided to build the computer on a single Euro card using wire wrap sockets where ever possible (if you happen to have wire wrap sockets or wire etc. and would like to give it to a good home I would love to take care of it - I am running low on wire wrap parts and it is next to impossible to find new sockets etc.). Designing and building the card took two evenings of about 5 hours each, and debugging took another, third evening (there were only four wiring errors but I made a software error in configuring the UART which was a bit hard to find).

The left row of the board shown above contains the TTL clock oscillators for the Z80 CPU (4 MHz) and the 16C550 UART (1.8432 MHz), a MAX232 for the serial line, a 74LS14 for the (simple) reset circuit and the reset push button. The next row contains the Z80 CPU, the 16C550 UART, a 62256 32kB RAM and a 27(C)512 32kB EPROM in a ZIF socket. The remainder of the board contains a simple voltage regulator with a 7805, several bus drivers (74LS245) and some address select logic.

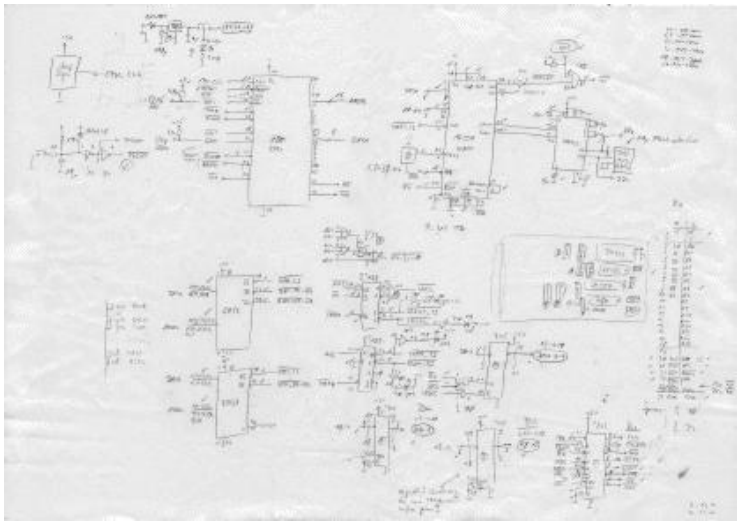
The following pictures give an impression of the process of building this simple computer:



The schematic of the computer is quite simple and straight forward. Since I wanted to build a really simple computer, there is no provision for DMA transfers and the interrupt logic has not been tested by now.

Please note that there is a tiny error in the schematic drawing: The TTL oscillator driving the UART is a 1.8432 MHz type and not an 8 MHz oscillator as noted in the drawing! (Sorry - I forgot to correct this.)

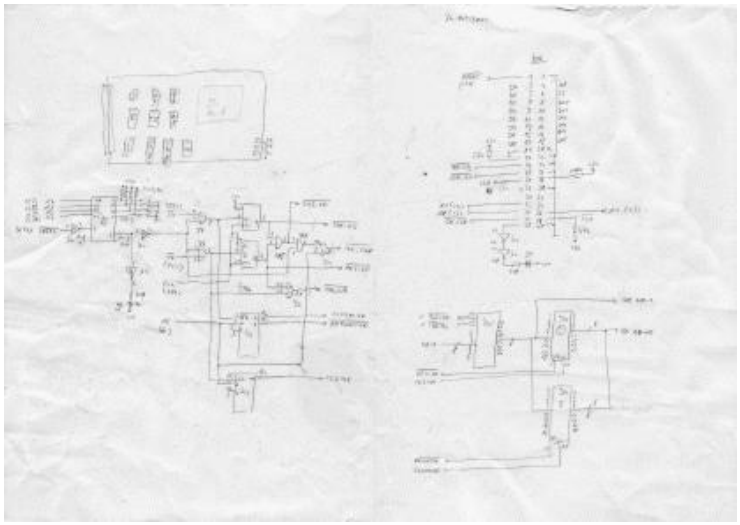


Here is the parts list for the computer (excluding sockets - I recommend to use precision sockets):

- TTL clock oscillator 4 MHz (for Z80 CPU - depending on your model, higher clock rates are possible)
- TTL clock oscillator 1.8432 MHz (for UART)
- MAX232 level converter for the serial line
- 74LS14 (for reset-circuitry)
- Push button for reset
- Z80 CPU (at least "A" variant which is capable of running at 4 MHz)
- 16C550 UART
- 62256 static RAM, 32 kB
- 27(C)256 EPROM (you might want to use a ZIF socket for the EPROM to facilitate insertion and removal of EPROMs during software development)
- 74LS08 (quad AND gate)
- 74LS139 (double 1-out-4 selector)
- 74LS32 two times (quad OR gate)
- 74LS245 four times (buffers for the bus - these are optional and only necessary if you plan to use external devices on the bus)
- 74LS07 (LED driver)
- 100 nF capacitors (16 times - one for each IC)
- 1N4148 diode (reset circuit)
- 10 uF capacitor (reset circuit)
- 10 Ohm resistor (reset circuit)
- 10 kOhm resistor (reset circuit)
- 1 uF capacitor (5 times - needed by the MAX232)
- 4.7 kOhm resistors (four times - needed to pull up some control lines - alternatively you can use a resistor array as I did)
- 10 pin socket or something equivalent for the serial line
- 100 uF capacitor (voltage regulator)
- 100 nF capacitor (voltage regulator)

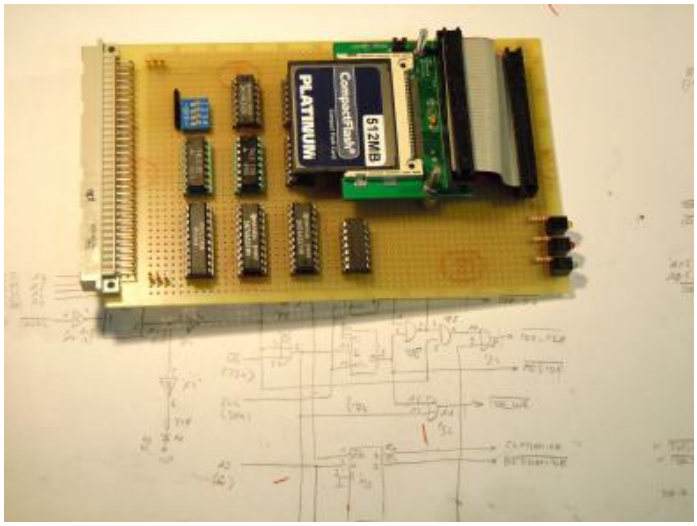
- 7805 linear voltage regulator, 5 V
- 3mm LEDs, 1 green, 2 yellow, 2 red)
- 220 Ohm resistor, 5 times (LEDs)
- Connector for power supply
- Experimental Euro-card (160 mm times 100 mm)
- VG connector, 64 or 96 pins

As stylish as a paper tape reader/puncher or at least a cassette drive would be, I decided to implement a simple IDE interface to use cheap and ruggedized Compact Flash cards for mass storage. After reading quite a bit about the IDE interface and having a look at other people's interfaces, I decided not to reinvent the wheel and implement the controlled developed by Phil from Retroleum since his design is clean and simple and features a really nice flipflop logic to generate the various control signals instead of RC delay circuits etc. as found in other designs.



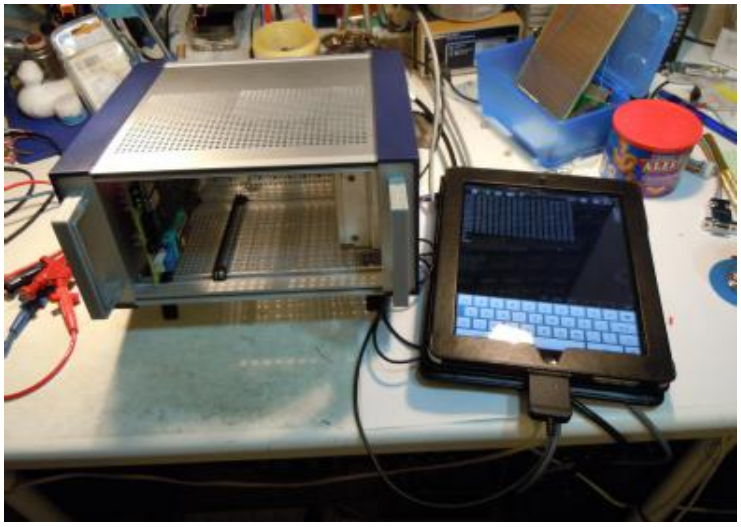
The picture on the right shows the schematic of the IDE controller. Please note that there is an error that I forgot to correct on the schematic: The two eight bit latches on the lower right are of type 74LS374 and NOT (!) 74LS574 as written. (74LS574 would work, too, but have a completely different pinout!)

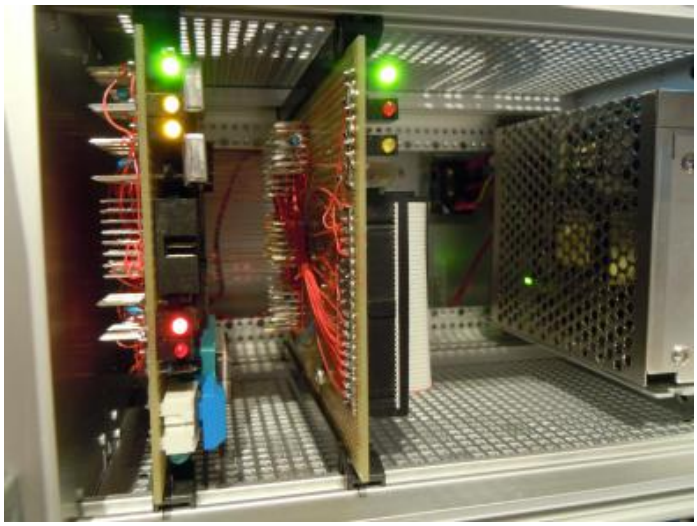
The circuit is described in detail on Phil's page at Retroleum. My implementation only differs with respect to an additional address decoder build around a 74LS85 4 bit comparator and a four place DIP switch. Currently my IDE controller is located at address $10 in the IO address space of the Z80 processor.



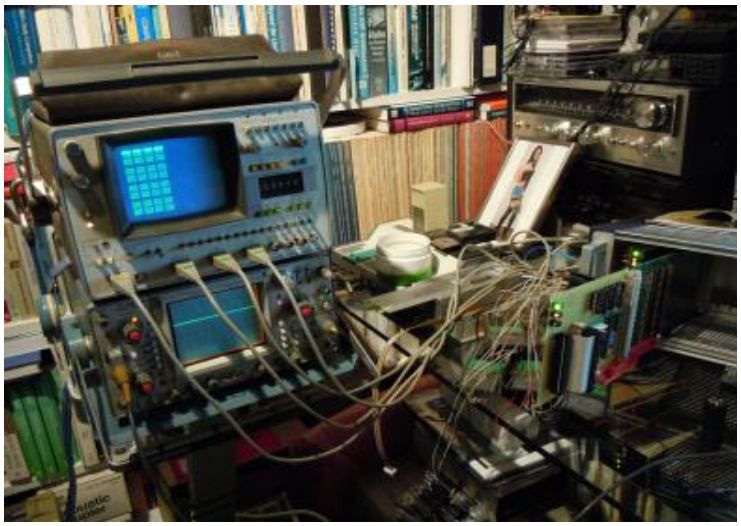
The picture on the left shows the completed IDE controller (which drove me nuts during debugging since I managed to handle the two select signals for the IDE devices incorrectly... *sigh*).

Since the tiny Z80 computer consists of two boards by now, an enclosure was needed. I decided to use a 10 inch standard enclosure (had I known before how much fiddling would be required to build the enclosure out of the many parts delivered by my supplier, I might have settled for something else :-)). The picture on the right shows the first picture of the CPU card in its new housing. The iPad is used as terminal (a very useful feature :-)).





The picture on the left shows both cards, the CPU board on the far left and the IDE controller right next to it, in operation. By the way - at first I thought about building a simple power supply with a linear regulator - then I wondered if I should use an old PC power supply and then I realized that new switching power supplies, capable of delivering 5V with 8A, cost only about 18 EUR... That is the reason for the small and modern power supply on the right hand side of the enclosure.



While the processor card worked quite right out of the box, the IDE controller was bit of a challenge since there were two independent bugs buried in its logic. The first problem was that the 74LS32 quad OR gate chip was defective (this is all the more puzzling since it was a new device). But more difficult to find was a mistake I did concerning the two select lines of the IDE device. I "thought" that I had understood the IDE logic and wired one of the /CS-lines to ground as I would have done with any TTL chip requiring two chip select lines. I did not realize that the two /CS lines act in fact as yet another address line and allow access to the upper registers of the IDE device. Fixing this bug was extremely simple, but finding it was hard since I was so sure that grounding the second /CS line was the right thing to do. :-)

In order to make any use at all of such a small homebrew computer one needs a small operating system, a so called "monitor" in fact. Writing the monitor took another couple of evenings up to now. Currently it occupies $166C bytes of memory (5740 bytes), so the 32 kB EPROM is not really well utilized at the moment. :-) The monitor is quite simple: All commands are collected into functional groups like CONTROL, DISK etc. The first letter typed at the command prompt selects the desired group while the second letter selects the command to be executed. The FAT file system implementation currently only supports reading files and displaying directory listings but this simplifies development a lot since one can now cross assemble/compile on another system and transfer the binaries by means of a CompactFlash card. :-)

Currently the monitor supports the following commands:

- **C**ontrol group:

    - **C**old start: Restart the computer, clearing memory.
    - **I**nfo: Print monitor version.
    - **S**tart: Start a machine program.
    - **W**arm start: Restart the computer without clearing memory.
- **D**isk group:
    - **I**nfo: Print disk information.
    - **M**ount: Mount the disk.
    - **T**ransfer: Initiate a transfer from or to disk:
        - **R**ead: Read sectors from disk into memory.
        - **W**rite: Write sectors from memory to disk.
    - **U**nmount: Unmount the disk.
- **F**ile group:
    - **C**at: Print a file's contents to STDOUT.
    - **D**irectory: Print the disk directory.
    - **L**oad: Load a file's contents into memory.
- **H**elp: Print help.
- **M**emory group:
    - **D**ump: Dump a memory region - the user is asked to enter a start and end address.
    - **E**xamine: Examine individual memory locations. After entering an address, the user can press the space bar to read the next memory cell or press any other key which will terminate the command.
    - **F**ill: Fill a memory area with a constant value.
    - **I**ntel hex load: Load the contents of an Intel Hex file into memory. This is really useful for testing newly developed software.
    - **L**oad: Load memory locations manually - after entering a start address, bytes in hexadecimal notation are read and stored in successive memory cells until a non-hexadecimal character will be entered.
    - **R**egister dump: Dump the current contents of both register banks.

Working with this simple monitor looks like this:

```
Simple Z80-monitor - V 0.8 (B. Ulmann, Sep. 2011 - Jan. 2012)
Cold start, clearing memory.

Z> DISK/MOUNT

        FATNAME:        MSDOS5.0
        CLUSIZ: 40
        RESSEC: 0008
        FATSEC: 00F0
        ROOTLEN:        0200
        PSIZ:           003C0030
        PSTART: 00001F80
        FAT1START:      00001F88
        ROOTSTART:      00002168
        DATASTART:      00002188

Z> FILE/DIRECTORY
Directory contents:
-----------------------------------------
FILENAME.EXT  DIR?   SIZE (BYTES)  1ST SECT
-----------------------------------------
_~1     .TRA            00001000        000021C8
TRASHE~1.       DIR     00000000        00002188
SPOTLI~1.       DIR     00000000        00002208
FSEVEN~1.       DIR     00000000        00002408
TEST    .TXT            00000013        00002688
FAT_1   .ASM            0000552C        000026C8
MEMO    .TXT            00000098        00002748
TEST210 .TXT            00000210        00002788
TEST1F0 .TXT            000001F0        00002DC8
TEST200 .TXT            00000200        00002E08
GROSS   .TXT            0000CB5A        00003488

Z> FILE/CAT: FILENAME=TEST.TXT
123 ich bin so frei
Z> FILE/LOAD FILE: ADDR=C000 FILENAME=TEST.TXT

0013 bytes loaded.
```

```
Z> MEMORY/DUMP: START=C000 END=C01F
C000: 31 32 33 20 69 63 68 20 62 69 6E 20 73 6F 20 66   123 ich bin so f
C010: 72 65 69 00 00 00 00 00 00 00 00 00 00 00 00 00   rei.............

Z>
```

The source of the monitor can be found here:

```
;******************************************************************************
;
;  Small monitor for the Z80 single board computer consisting of 32 kB ROM
; ($0000 to $ffff), 32 kB RAM ($8000 to $ffff) and a 16c550 UART.
;
; B. Ulmann, 28-SEP-2011, 29-SEP-2011, 01-OCT-2011, 02-OCT-2011, 30-OCT-2011,
;            01-NOV-2011, 02-NOV-2011, 03-NOV-2011, 06/07/08-JAN-2012
; I. Kloeckl, 06/07/08-JAN-2012 (FAT implementation for reading files)
; B. Ulmann, 14-JAN-2011,
;
; Version 0.8
;
;******************************************************************************
;
; TODO:
;       Read and print IDE error status codes in case of error!
;
; Known issues:
;       Memory Dump has a problem when the end address is >= FF00
;
;******************************************************************************
;
;  RST $00 will enter the monitor (do not care about the return address pushed
; onto the stack - the stack pointer will be reinitialized during cold as well
; as during warm starts.
;
;  Monitor routines will generally called by placing the necessary parameters
; into some processor registers and then issuing RST $08. More about this later.
;
;  Memory layout is as follows:
;
;  +-------+
;  ! $FFFF !    General purpose 512 byte buffer
;  !  ---  !
;  ! $FE00 !
;  +-------+
;  ! $DFFF !    FAT control block
;  !  ---  !
;  ! $FDDC !
;  +-------+
;  ! $FDDB !    File control block
;  !  ---  !
;  ! $FBBE !
;  +-------+
;  ! $FBBD !    81 byte string buffer
;  !  ---  !
;  ! $FB6D !
;  +-------+
;  ! $FB6C !    12 byte string buffer
;  !  ---  !
;  ! $FB61 !
;  +-------+
;  ! $FB60 !    Buffers for various routines
;  !  ---  !
;  ! $FB4D !
;  +-------+
;  ! $FB4C !    Cold/warm start control (1 byte)
;  +-------+
;  ! $FBBD !    Stack
;  !  ...  !
;  ! $8000 !    Begin of RAM
;  +-------+
;  ! $7FFF !    ROM area
;  !  ---  !    RST $08 calls a system routine
;  ! $0000 !    RST $00 restarts the monitor
```

```
;  +-------+
;
;
monitor_start   equ     $0000           ; $0000 -> ROM, $8000 -> Test image
;
                org     monitor_start
;
rom_start       equ     $0
rom_end         equ     $7fff
ram_start       equ     $8000
ram_end         equ     $ffff
buffer          equ     ram_end - $1ff  ; 512 byte IDE general purpose buffer
;
; Define the FAT control block memory addresses:
;
datastart       equ     buffer - 4      ; Data area start vector
rootstart       equ     datastart - 4   ; Root directory start vector
fat1start       equ     rootstart - 4   ; Start vector to first FAT
psiz            equ     fat1start - 4   ; Size of partition (in sectors)
pstart          equ     psiz - 4        ; First sector of partition
rootlen         equ     pstart - 2      ; Maximum number of entries in directory
fatsec          equ     rootlen - 2     ; FAT size in sectors
ressec          equ     fatsec - 2      ; Number of reserved sectors
clusiz          equ     ressec - 1      ; Size of a cluster (in sectors)
fatname         equ     clusiz - 9      ; Name of the FAT (null terminated)
fatcb           equ     fatname         ; Start of the FATCB
;
; Define a file control block (FCB) memory addresses and displacements:
;
file_buffer     equ     fatcb - $200            ; 512 byte sector buffer
cluster_sector  equ     file_buffer - 1         ; Current sector in cluster
current_sector  equ     cluster_sector - 4      ; Current sector address
current_cluster equ     current_sector - 2      ; Current cluster number
file_pointer    equ     current_cluster - 4     ; Pointer for file position
file_type       equ     file_pointer - 1        ; 0 -> not found, else OK
first_cluster   equ     file_type - 2           ; First cluster of file
file_size       equ     first_cluster - 4       ; Size of file
file_name       equ     file_size - 12          ; Canonical name of file
fcb             equ     file_name               ; Start of the FCB
;
fcb_filename            equ     0
fcb_file_size           equ     $c
fcb_first_cluster       equ     $10
fcb_file_type           equ     $12
fcb_file_pointer        equ     $13
fcb_current_cluster     equ     $17
fcb_current_sector      equ     $19
fcb_cluster_sector      equ     $1d
fcb_file_buffer         equ     $1e
;
; We also need some general purpose string buffers:
;
string_81_bfr   equ     fcb - 81
string_12_bfr   equ     string_81_bfr - 12
;
;  A number of routines need a bit of scratch RAM, too. Since these are
; sometimes interdependent, each routine gets its own memory cells (only
; possible since the routines are not recursive).
;
load_file_scrat equ     string_12_bfr - 2       ; Two bytes for load_file
str2filename_de equ     load_file_scrat - 2     ; Two bytes for str2filename
fopen_eob       equ     str2filename_de - 2     ; Eight bytes for fopen
fopen_rsc       equ     fopen_eob - 4
fopen_scr       equ     fopen_rsc - 2
dirlist_scratch equ     fopen_scr - 2           ; Eight bytes for fopen
dirlist_eob     equ     dirlist_scratch - 2
dirlist_rootsec equ     dirlist_eob - 4
;
start_type      equ     dirlist_rootsec  - $1   ; Distinguish cold/warm start
;
uart_base       equ     $0
ide_base        equ     $10
;
uart_register_0 equ     uart_base + 0
```

```
uart_register_1 equ     uart_base + 1
uart_register_2 equ     uart_base + 2
uart_register_3 equ     uart_base + 3
uart_register_4 equ     uart_base + 4
uart_register_5 equ     uart_base + 5
uart_register_6 equ     uart_base + 6
uart_register_7 equ     uart_base + 7
;
eos             equ     $00             ; End of string
cr              equ     $0d             ; Carriage return
lf              equ     $0a             ; Line feed
space           equ     $20             ; Space
tab             equ     $09             ; Tabulator
;
; Main entry point (RST 00H):
;
rst_00          di                      ; Disable interrupts
                jr      initialize      ; Jump over the RST-area
;
;  RST-area - here is the main entry point into the monitor. The calling
; standard looks like this:
;
; 1) Set register IX to the number of the system routine to be called.
; 2) Set the remaining registers according to the routine's documentation.
; 3) Execute RST $08 to actually call the system routine.
; 4) Evaluate the values returned in the registers as described by the
;    Routine's documentation.
;
;  (Currently there are no plans to use more RST entry points, so this routine
; just runs as long as necessary in memory. If more RSTs will be used, this
; routine should to be moved to the end of the used ROM area with only a
; simple jump at the RST $08-location.)
;
;  This technique of calling system routines can be used as the following
; example program that just echos characters read from the serial line
; demonstrates:
;
;        org     $8000           ; Start in lower RAM
; loop   ld      ix, 5           ; Prepare call to getc
;        rst     08              ; Execute getc
;        cp      3               ; CTRL-C pressed?
;        jr      z, exit         ; Yes - exit
;        ld      ix, 6           ; Prepare call to putc
;        rst     08              ; Execute putx
;        jr      loop            ; Process next character
; exit   ld      ix, 4           ; Exit - print a CR/LF pair
;        rst     08              ; Call CRLF
;        ld      hl, msg         ; Pointer to exit message
;        ld      ix, 7           ; Prepare calling puts
;        rst     08              ; Call puts
;        rst     00              ; Restart monitor (warm start)
; msg    defb    "That's all folks.", $d, $a, 0
;
;  Currently the following functions are available (a more detailed description
; can be found in the dispatch table itself):
;
;       0:      cold_start
;       1:      is_hex
;       2:      is_print
;       3:      to_upper
;       4:      crlf
;       5:      getc
;       6:      putc
;       7:      puts
;       8:      strcmp
;       9:      gets
;       A:      fgetc
;       B:      dump_fcb
;       C:      fopen
;       D:      dirlist
;       E:      fatmount
;       F:      fatunmount
;
;                org     monitor_start + $08
```

```
                nop                     ; Beware: zasm is buggy concerning
                nop                     ; the org pseudo-statement. Therefore
                nop                     ; The displacement to the RST $08
                nop                     ; entry point is generated by this
                nop                     ; NOP-sequence.
rst_08          push    bc              ; Save bc and hl
                push    hl
                push    ix              ; Copy the contents of ix
                pop     hl              ; into hl
                add     hl, hl          ; Double to get displacement in table
                ld      bc, dispatch_table
                add     hl, bc          ; Calculate displacement in table
                ld      bc, (hl)        ; Load bc with the destination address
                push    bc
                pop     ix              ; Load ix with the destination address
                pop     hl              ; Restore hl
                pop     bc              ; and bc
                jp      (ix)            ; Jump to the destination routine
dispatch_table  defw    cold_start      ; $00 = clear etc.
                ; Parameters:    N/A
                ; Action:        Performs a cold start (memory is cleared!)
                ; Return values: N/A
                ;
                defw    is_hex
                ; Parameters:    A contains a character code
                ; Action:        Tests ('0' <= A <= '9) || ('A' <= A <= 'F')
                ; Return values: Carry bit is set if A contains a hex char.
                ;
                defw    is_print
                ; Parameters:    A contains a charater code
                ; Action:        Tests if the character is printable
                ; Return values: Carry bit is set if A contains a valid char.
                ;
                defw    to_upper
                ; Parameters:    A contains a character code
                ; Action:        Converts an ASCII character into upper case
                ; Return values: Converted character code in A
                ;
                defw    crlf
                ; Parameters:    N/A
                ; Action:        Sends a CR/LF to the serial line
                ; Return values: N/A
                ;
                defw    getc
                ; Parameters:    A contains a character code
                ; Action:        Reads a character code from the serial line
                ; Return values: N/A
                ;
                defw    putc
                ; Parameters:    A contains a character code
                ; Action:        Sends the character code to the serial line
                ; Return values: N/A
                ;
                defw    puts
                ; Parameters:    HL contains the address of a 0-terminated
                ;                string
                ; Action:        Send the string to the serial line (excluding
                ;                the termination byte, of course)
                ; Return values: N/A
                ;
                defw    strcmp
                ; Parameters:    HL and DE contain the addresses of two strings
                ; Action:        Compare both strings.
                ; Return values: A contains return value, <0 / 0 / >0
                ;
                defw    gets
                ; Parameters:    HL contains a buffer address, B contains the
                ;                buffer length (including the terminating
                ;                null byte!)
                ; Action:        Reads a string from STDIN. Terminates when
                ;                either the buffer is full or the string is
                ;                terminated by CR/LF.
                ; Return values: N/A
                ;
```

```
                defw    fgetc
                ; Parameters:    IY (pointer to a valid FCB)
                ; Action:        Reads a character from a FAT file
                ; Return values: Character in A, if EOF has been encountered,
                ;                the carry flag will be set
                ;
                defw    dump_fcb
                ; Parameters:    IY (pointer to a valid FCB)
                ; Action:        Prints the contents of the FCB in human
                ;                readable format to STDOUT
                ; Return values: N/A
                ;
                defw    fopen
                ; Parameters:    HL (points to a buffer containing the file
                ;                file name), IY (points to an empty FCB)
                ; Action:        Opens a file for reading
                ; Return values: N/A (All information is contained in the FCB)
                ;
                defw    dirlist
                ; Parameters:    N/A (relies on a valid FAT control block)
                ; Action:        Writes a directory listing to STDOUT
                ; Return values: N/A
                ;
                defw    fatmount
                ; Parameters:    N/A (needs the global FAT control block)
                ; Action:        Mounts a disk (populates the FAT CB)
                ; Return values: N/A
                ;
                defw    fatunmount
                ; Parameters:    N/A (needs the global FAT control block)
                ; Action:        Invalidates the global FAT control block
                ; Return values; N/A
;
;  The stackpointer will be predecremented by a push instruction. Since we need
; a 512 byte buffer for data transfers to and from the IDE disk, the stack
; pointer is initialized to start at the beginning of this buffer space.
;
initialize      ld      sp, start_type - $1
;
; Initialize UART to 9600,8N1:
;
                ld      a, $80
                out     (uart_register_3), a
                ld      a, $c           ; 1843200 / (16 * 9600)
                out     (uart_register_0), a
                xor     a
                out     (uart_register_1), a
                ld      a, $3           ; 8N1
                out     (uart_register_3), a
;
; Print welcome message:
;
                ld      hl, hello_msg
                call    puts
;
;  If this is a cold start (the location start_type does not contain $aa)
; all available RAM will be reset to $00 and a message will be printed.
;
                ld      a, (start_type)
                cp      $aa             ; Warm start?
                jr      z, main_loop    ; Yes - enter command loop
                ld      hl, cold_start_msg
                call    puts            ; Print cold start message
                ld      hl, ram_start   ; Start of block to be filled with $00
                ld      de, hl          ; End address of block
                inc     de              ; plus 1 (for ldir)
                ld      bc, ram_end - ram_start
                ld      (hl), $00       ; Load first memory location
                ldir                    ; And copy this value down
                ld      hl, start_type
                ld      (hl), $aa       ; Cold start done, remember this
;
; Read characters from the serial line and send them just back:
;
```

```
main_loop       ld      hl, monitor_prompt
                call    puts
; The monitor is rather simple: All commands are just one or two letters.
; The first character selects a command group, the second the desired command
; out of that group. When a command is recognized, it will be spelled out
; automatically and the user will be prompted for arguments if applicable.
                call    monitor_key     ; Read a key
; Which group did we get?
                cp      'C'             ; Control group?
                jr      nz, disk_group  ; No - test next group
                ld      hl, cg_msg      ; Print group prompt
                call    puts
                call    monitor_key     ; Get command key
                cp      'C'             ; Cold start?
                jp      z, cold_start
                cp      'W'             ; Warm start?
                jp      z, warm_start
                cp      'S'             ; Start?
                jp      z, start
                cp      'I'             ; Info?
                call    z, info
                jr      z, main_loop
                jp      cmd_error       ; Unknown control-group-command
disk_group      cp      'D'             ; Disk group?
                jr      nz, file_group  ; No - file group?
                ld      hl, dg_msg      ; Print group prompt
                call    puts
                call    monitor_key     ; Get command
                cp      'I'             ; Info?
                call    z, disk_info
                jr      z, main_loop
                cp      'M'             ; Mount?
                call    z, mount
                jr      z, main_loop
                cp      'T'             ; Read from disk?
                call    z, disk_transfer
                jr      z, main_loop
                cp      'U'             ; Unmount?
                call    z, unmount
                jr      z, main_loop
                jr      cmd_error       ; Unknown disk-group-command
file_group      cp      'F'             ; File group?
                jr      nz, help_group  ; No - help group?
                ld      hl, fg_msg      ; Print group prompt
                call    puts
                call    monitor_key     ; Get command
                cp      'C'             ; Cat?
                call    z, cat_file
                jr      z, main_loop
                cp      'D'             ; Directory?
                call    z, directory
                jr      z, main_loop
                cp      'L'             ; Load?
                call    z, load_file
                jr      z, main_loop
                jr      cmd_error       ; Unknown file-group-command
help_group      cp      'H'             ; Help? (No further level expected.)
                call    z, help         ; Yes :-)
                jp      z, main_loop
memory_group    cp      'M'             ; Memory group?
                jp      nz, group_error ; No - print an error message
                ld      hl, mg_msg      ; Print group prompt
                call    puts
                call    monitor_key     ; Get command key
                cp      'D'             ; Dump?
                call    z, dump
                jp      z, main_loop
                cp      'E'             ; Examine?
                call    z, examine
                jp      z, main_loop
                cp      'F'             ; Fill?
                call    z, fill
                jp      z, main_loop
                cp      'I'             ; INTEL-Hex load?
```

```
                call    z, ih_load
                jp      z, main_loop
                cp      'L'             ; Load?
                call    z, load
                jp      z, main_loop
                cp      'M'             ; Move?
                call    z, move
                jp      z, main_loop
                cp      'R'             ; Register dump?
                call    z, rdump
                jp      z, main_loop
                jr      cmd_error       ; Unknown memory-group-command
group_error     ld      hl, group_err_msg
                jr      print_error
cmd_error       ld      hl, command_err_msg
print_error     call    putc            ; Echo the illegal character
                call    puts            ; and print the error message
                jp      main_loop
;
; Some constants for the monitor:
;
hello_msg       defb    cr, lf, cr, lf, "Simple Z80-monitor - V 0.8 "
                defb    "(B. Ulmann, Sep. 2011 - Jan. 2012)", cr, lf, eos
monitor_prompt  defb    cr, lf, "Z> ", eos
cg_msg          defb    "CONTROL/", eos
dg_msg          defb    "DISK/", eos
fg_msg          defb    "FILE/", eos
mg_msg          defb    "MEMORY/", eos
command_err_msg defb    ": Syntax error - command not found!", cr, lf, eos
group_err_msg   defb    ": Syntax error - group not found!", cr, lf, eos
cold_start_msg  defb    "Cold start, clearing memory.", cr, lf, eos
;
; Read a key for command group and command:
;
monitor_key     call    getc
                cp      lf              ; Ignore LF
                jr      z, monitor_key  ; Just get the next character
                call    to_upper
                cp      cr              ; A CR will return to the prompt
                ret     nz              ; No - just return
                inc     sp              ; Correct SP to and avoid ret!
                jp      main_loop
;
;*****************************************************************************
;***
;*** The following routines are used in the interactive part of the monitor
;***
;*****************************************************************************
;
; Print a file's contents to STDOUT:
;
cat_file        push    bc
                push    de
                push    hl
                push    iy
                ld      hl, cat_file_prompt
                call    puts
                ld      hl, string_81_bfr
                ld      b, 81
                call    gets            ; Read the filename into buffer
                ld      iy, fcb         ; Prepare fopen (only one FCB currently)
                ld      de, string_12_bfr
                call    fopen
cat_file_loop   call    fgetc           ; Get a single character
                jr      c, cat_file_exit
                call    putc            ; Print character if not EOF
                jr      cat_file_loop   ; Next character
cat_file_exit   pop     iy
                pop     hl
                pop     de
                pop     bc
                ret
cat_file_prompt defb    "CAT: FILENAME=", eos
;
```

```
;  directory - a simple wrapper for dirlist (necessary for printing the command
; name)
;
directory       push    hl
                ld      hl, directory_msg
                call    puts
                call    dirlist
                pop     hl
                ret
directory_msg   defb    "DIRECTORY", cr, lf, eos
;
; Get and print disk info:
;
disk_info       push    af
                push    hl
                ld      hl, disk_info_msg
                call    puts
                call    ide_get_id      ; Read the disk info into the IDE buffer
                ld      hl, buffer + $13
                ld      (hl), tab
                call    puts            ; Print vendor information
                call    crlf
                ld      hl, buffer + $2d
                ld      (hl), tab
                call    puts
                call    crlf
                pop     hl
                pop     af
                ret
disk_info_msg   defb    "INFO:", cr, lf, eos
;
; Read data from disk to memory
;
disk_transfer   push    af
                push    bc
                push    de
                push    hl
                push    ix
                ld      hl, disk_trx_msg_0
                call    puts            ; Print Read/Write prompt
disk_trx_rwlp   call    getc
                call    to_upper
                cp      'R'             ; Read?
                jr      nz, disk_trx_nr ; No
                ld      ix, ide_rs      ; Yes, we will call ide_rs later
                ld      hl, disk_trx_msg_1r
                jr      disk_trx_main   ; Prompt the user for parameters
disk_trx_nr     cp      'W'             ; Write?
                jr      nz, disk_trx_rwlp
                ld      ix, ide_ws      ; Yes, we will call ide_ws later
                ld      hl, disk_trx_msg_1w
disk_trx_main   call    puts            ; Print start address prompt
                call    get_word        ; Get memory start address
                push    hl
                ld      hl, disk_trx_msg_2
                call    puts            ; Prompt for number of blocks
                call    get_byte        ; There are only 128 block of memory!
                cp      0               ; Did the user ask for 00 blocks?
                jr      nz, disk_trx_1  ; No, continue prompting
                ld      hl, disk_trx_msg_4
                call    puts
                jr      disk_trx_exit
disk_trx_1      ld      hl, disk_trx_msg_3
                call    puts            ; Prompt for disk start sector
                call    get_word        ; This is a four byte address!
                ld      bc, hl
                call    get_word
                ld      de, hl
                pop     hl              ; Restore memory start address
                ; Register contents:
                ;       A:  Number of blocks
                ;       BC: LBA3/2
                ;       DE: LBA1/0
                ;       HL: Memory start address
```

```
disk_trx_loop   push    af              ; Save number of sectors
                call    disk_trampoline ; Read/write one sector (F is changed!)
                push    hl              ; Save memory address
                push    bc              ; Save LBA3/2
                ld      hl, de          ; Increment DE (LBA1/0)
                ld      bc, $0001       ; by one and
                add     hl, bc          ; generate a carry if necessary
                ld      de, hl          ; Save new LBA1/0
                pop     hl              ; Restore LBA3/2 into HL (!)
                jr      nc, disk_trx_skip
                add     hl, bc          ; Increment BC if there was a carry
disk_trx_skip   ld      bc, hl          ; Write new LBA3/2 into BC
                pop     hl              ; Restore memory address
                push    bc              ; Save LBA3/2
                ld      bc, $200        ; 512 byte per block
                add     hl, bc          ; Set pointer to next memory block
                pop     bc              ; Restore LBA3/2
                pop     af
                dec     a               ; One block already done
                jr      nz, disk_trx_loop
disk_trx_exit   pop     ix
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
disk_trampoline jp      (ix)
disk_trx_msg_0  defb    "TRANSFER/", eos
disk_trx_msg_1r defb    "READ: ", cr, lf, "    MEMORY START=", eos
disk_trx_msg_1w defb    "WRITE: ", cr, lf, "    MEMORY START=", eos
disk_trx_msg_2  defb    " NUMBER OF BLOCKS (512 BYTE)=", eos
disk_trx_msg_3  defb    " START SECTOR=", eos
disk_trx_msg_4  defb    " Nothing to do for zero blocks.", cr, lf, eos
;
; Dump a memory area
;
dump            push    af
                push    bc
                push    de
                push    hl
                ld      hl, dump_msg_1
                call    puts            ; Print prompt
                call    get_word        ; Read start address
                push    hl              ; Save start address
                ld      hl, dump_msg_2  ; Prompt for end address
                call    puts
                call    get_word        ; Get end address
                call    crlf
                inc     hl              ; Increment stop address for comparison
                ld      de, hl          ; DE now contains the stop address
                pop     hl              ; HL is the start address again
                ; This loop will dump 16 memory locations at once - even
                ; if this turns out to be more than requested.
dump_line       ld      b, $10          ; This loop will process 16 bytes
                push    hl              ; Save HL again
                call    print_word      ; Print address
                ld      hl, dump_msg_3  ; and a colon
                call    puts
                pop hl                  ; Restore address
                push    hl              ; We will need HL for the ASCII dump
dump_loop       ld      a, (hl)         ; Get the memory content
                call    print_byte      ; and print it
                ld      a, ' '          ; Print a space
                call    putc
                inc     hl              ; Increment address counter
                djnz    dump_loop       ; Continue with this line
                ; This loop will dump the very same 16 memory locations - but
                ; this time printable ASCII characters will be written.
                ld      b, $10          ; 16 characters at a time
                ld      a, ' '          ; We need some spaces
                call    putc            ; to print
                call    putc
                pop     hl              ; Restore the start address
dump_ascii_loop ld      a, (hl)         ; Get byte
```

```
                call    is_print        ; Is it printable?
                jr      c, dump_al_1    ; Yes
                ld      a, '.'          ; No - print a dot
dump_al_1       call    putc            ; Print the character
                inc     hl              ; Increment address to read from
                djnz    dump_ascii_loop
                ; Now we are finished with printing one line of dump output.
                call    crlf            ; CR/LF for next line on terminal
                push    hl              ; Save the current address for later
                and     a               ; Clear carry
                sbc     hl, de          ; Have we reached the last address?
                pop     hl              ; restore the address
                jr      c, dump_line    ; Dump next line of 16 bytes
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
dump_msg_1      defb    "DUMP: START=", eos
dump_msg_2      defb    " END=", eos
dump_msg_3      defb    ": ", eos
;
; Examine a memory location:
;
examine         push    af
                push    hl
                ld      hl, examine_msg_1
                call    puts
                call    get_word        ; Wait for a four-nibble address
                push    hl              ; Save address for later
                ld      hl, examine_msg_2
                call    puts
examine_loop    pop     hl              ; Restore address
                ld      a, (hl)         ; Get content of address
                inc     hl              ; Prepare for next examination
                push    hl              ; Save hl again for later use
                call    print_byte      ; Print the byte
                call    getc            ; Get a character
                cp      ' '             ; A blank?
                jr      nz, examine_exit; No - exit
                ld      a, ' '          ; Print a blank character
                call    putc
                jr      examine_loop
examine_exit    pop     hl              ; Get rid of save hl value
                call    crlf            ; Print CR/LF
                pop     hl
                pop     af
                ret
examine_msg_1   defb    "EXAMINE (type ' '/RET): ADDR=", eos
examine_msg_2   defb    " DATA=", eos
;
; Fill a block of memory with a single byte - the user is prompted for the
; start address, the length of the block and the fill value.
;
fill            push    af              ; We will need nearly all registers
                push    bc
                push    de
                push    hl
                ld      hl, fill_msg_1  ; Prompt for start address
                call    puts
                call    get_word        ; Get the start address
                push    hl              ; Store the start address
                and     a               ; Clear carry
                ld      bc, ram_start
                sbc     hl, bc          ; Is the address in the RAM area?
                jr      nc, fill_get_length
                ld      hl, fill_msg_4  ; No!
                call    puts            ; Print error message
                pop     hl              ; Clean up the stack
                jr      fill_exit       ; Leave routine
fill_get_length ld      hl, fill_msg_2  ; Prompt for length information
                call    puts
                call    get_word        ; Get the length of the block
                ; Now make sure that start + length is still in RAM:
```

```
                ld      bc, hl          ; BC contains the length
                pop     hl              ; HL now contains the start address
                push    hl              ; Save the start address again
                push    bc              ; Save the length
                add     hl, bc          ; Start + length
                and     a               ; Clear carry
                ld      bc, ram_start
                sbc     hl, bc          ; Compare with ram_start
                jr      nc, fill_get_value
                ld      hl, fill_msg_5  ; Print error message
                call    puts
                pop     bc              ; Clean up the stack
                pop     hl
                jr      fill_exit       ; Leave the routine
fill_get_value  ld      hl, fill_msg_3  ; Prompt for fill value
                call    puts
                call    get_byte        ; Get the fill value
                pop     bc              ; Get the length from the stack
                pop     hl              ; Get the start address again
                ld      de, hl          ; DE = HL + 1
                inc     de
                dec     bc
                ; HL = start address
                ; DE = destination address = HL + 1
                ;      Please note that this is necessary - LDIR does not
                ;      work with DE == HL. :-)
                ; A  = fill value
                ld      (hl), a         ; Store A into first memory location
                ldir                    ; Fill the memory
                call    crlf
fill_exit       pop     hl              ; Restore the register contents
                pop     de
                pop     bc
                pop     af
                ret
fill_msg_1      defb    "FILL: START=", eos
fill_msg_2      defb    " LENGTH=", eos
fill_msg_3      defb    " VALUE=", eos
fill_msg_4      defb    " Illegal address!", cr, lf, eos
fill_msg_5      defb    " Block exceeds RAM area!", cr, lf, eos
;
; Help
;
help            push    hl
                ld      hl, help_msg
                call    puts
                pop     hl
                ret
help_msg        defb    "HELP: Known command groups and commands:", cr, lf
                defb    "         C(ontrol group):", cr, lf
                defb    "             C(old start), I(nfo), S(tart), "
                defb    "W(arm start)", cr, lf
                defb    "         D(isk group):", cr, lf
                defb    "             I(nfo), M(ount), T(ransfer),"
                defb    "         U(nmount)", cr, lf
                defb    "                                   R(ead), W(rite)"
                defb    cr, lf
                defb    "         F(ile group):", cr, lf
                defb    "             C(at), D(irectory), L(oad)", cr, lf
                defb    "         H(elp)", cr, lf
                defb    "         M(emory group):", cr, lf
                defb    "             D(ump), E(xamine), F(ill), "
                defb    "I(ntel Hex Load), L(oad), R(egister dump)"
                defb    cr, lf, eos
;
; Load an INTEL-Hex file (a ROM image) into memory. This routine has been
; more or less stolen from a boot program written by Andrew Lynch and adapted
; to this simple Z80 based machine.
;
; The INTEL-Hex format looks a bit awkward - a single line contains these
; parts:
; ':', Record length (2 hex characters), load address field (4 hex characters),
; record type field (2 characters), data field (2 * n hex characters),
; checksum field. Valid record types are 0 (data) and 1 (end of file).
```

```
;
; Please note that this routine will not echo what it read from stdin but
; what it "understood". :-)
;
ih_load         push    af
                push    de
                push    hl
                ld      hl, ih_load_msg_1
                call    puts
ih_load_loop    call    getc            ; Get a single character
                cp      cr              ; Don't care about CR
                jr      z, ih_load_loop
                cp      lf              ; ...or LF
                jr      z, ih_load_loop
                cp      space           ; ...or a space
                jr      z, ih_load_loop
                call    to_upper        ; Convert to upper case
                call    putc            ; Echo character
                cp      ':'             ; Is it a colon?
                jr      nz, ih_load_error
                call    get_byte        ; Get record length into A
                ld      d, a            ; Length is now in D
                ld      e, $0           ; Clear checksum
                call    ih_load_chk     ; Compute checksum
                call    get_word        ; Get load address into HL
                ld      a, h            ; Update checksum by this address
                call    ih_load_chk
                ld      a, l
                call    ih_load_chk
                call    get_byte        ; Get the record type
                call    ih_load_chk     ; Update checksum
                cp      $1              ; Have we reached the EOF marker?
                jr      nz, ih_load_data; No - get some data
                call    get_byte        ; Yes - EOF, read checksum data
                call    ih_load_chk     ; Update our own checksum
                ld      a, e
                and     a               ; Is our checksum zero (as expected)?
                jr      z, ih_load_exit ; Yes - exit this routine
ih_load_chk_err ld      hl, ih_load_msg_3
                call    puts            ; No - print an error message
                jr      ih_load_exit    ; and exit
ih_load_data    ld      a, d            ; Record length is now in A
                and     a               ; Did we process all bytes?
                jr      z, ih_load_eol  ; Yes - process end of line
                call    get_byte        ; Read two hex digits into A
                call    ih_load_chk     ; Update checksum
                ld      (hl), a         ; Store byte into memory
                inc     hl              ; Increment pointer
                dec     d               ; Decrement remaining record length
                jr      ih_load_data    ; Get next byte
ih_load_eol     call    get_byte        ; Read the last byte in the line
                call    ih_load_chk     ; Update checksum
                ld      a, e
                and     a               ; Is the checksum zero (as expected)?
                jr      nz, ih_load_chk_err
                call    crlf
                jr      ih_load_loop    ; Yes - read next line
ih_load_error   ld      hl, ih_load_msg_2
                call    puts            ; Print error message
ih_load_exit    call    crlf
                pop     hl              ; Restore registers
                pop     de
                pop     af
                ret
;
ih_load_chk     ld      c, a            ; All in all compute E = E - A
                ld      a, e
                sub     c
                ld      e, a
                ld      a, c
                ret
ih_load_msg_1   defb    "INTEL HEX LOAD: ", eos
ih_load_msg_2   defb    " Syntax error!", eos
ih_load_msg_3   defb    " Checksum error!", eos
```

```
;
; Print version information etc.
;
info            push    hl
                ld      hl, info_msg
                call    puts
                ld      hl, hello_msg
                call    puts
                pop     hl
                ret
info_msg        defb    "INFO: ", eos
;
; Load data into memory. The user is prompted for a 16 bit start address. Then
; a sequence of bytes in hexadecimal notation may be entered until a character
; that is not 0-9 or a-f is encountered.
;
load            push    af
                push    bc
                push    de
                push    hl
                ld      hl, load_msg_1  ; Print command name
                call    puts
                call    get_word        ; Wait for the start address (2 bytes)
                push    hl              ; Remember address
                and     a               ; Clear carry
                ld      bc, ram_start   ; Check if the address is valid
                sbc     hl, bc          ; by subtracting the RAM start address
                pop     hl              ; Restore address
                ld      de, 0           ; Counter for bytes loaded
                jr      nc, load_loop   ; OK - start reading hex characters
                ld      hl, load_msg_3  ; Print error message
                call    puts
                jr      load_exit
                ; All in all we need two hex nibbles per byte. If two characters
                ; in a row are valid hexadecimal digits we will convert them
                ; to a byte and store this in memory. If one character is
                ; illegal, the load routine terminates and returns to the
                ; monitor.
load_loop       ld      a, ' '
                call    putc            ; Write a space as byte delimiter
                call    getc            ; Read first character
                call    to_upper        ; Convert to upper case
                call    is_hex          ; Is it a hex digit?
                jr      nc, load_exit   ; No - exit the load routine
                call    nibble2val      ; Convert character to value
                call    print_nibble    ; Echo hex digit
                rlc     a
                rlc     a
                rlc     a
                rlc     a
                ld      b, a            ; Save the upper four bits for later
                call    getc            ; Read second character and proceed...
                call    to_upper        ; Convert to upper case
                call    is_hex
                jr      nc, load_exit
                call    nibble2val
                call    print_nibble
                or      b               ; Combine lower 4 bits with upper
                ld      (hl), a         ; Save value to memory
                inc     hl
                inc     de
                jr      load_loop       ; Get next byte (or at least try to)
load_exit       call    crlf            ; Finished...
                ld      hl, de          ; Print number of bytes loaded
                call    print_word
                ld      hl, load_msg_2
                call    puts
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
load_msg_1      defb    "LOAD (xx or else to end): ADDR=", eos
load_msg_2      defb    " bytes loaded.", cr, lf, eos
```

```
load_msg_3      defb    " Illegal address!", eos
;
; Load a file's contents into memory:
;
load_file       push    af
                push    bc
                push    de
                push    hl
                push    iy
                ld      hl, load_file_msg_1
                call    puts            ; Print first prompt (start address)
                call    get_word        ; Wait for the start address (2 bytes)
                ld      (load_file_scrat), hl
                and     a               ; Clear carry
                ld      bc, ram_start   ; Check if the address is valid
                sbc     hl, bc          ; by subtracting the RAM start address
                jr      nc, load_file_1
                ld      hl, load_file_msg_2
                call    puts
                jr      load_file_exit  ; Illegal address - exit routine
load_file_1     ld      hl, load_file_msg_4
                call    puts            ; Prompt for filename
                ld      hl, string_81_bfr
                ld      b, 81           ; Buffer length
                call    gets            ; Read file name into bfr
                ld      iy, fcb         ; Prepare open (only one FCB currently)
                ld      de, string_12_bfr
                call    fopen           ; Open the file (if possible)
                ld      hl, (load_file_scrat)
                ld      de, 0           ; Counter for bytes loaded
load_file_loop  call    fgetc           ; Get one byte from the file
                jr      c, load_file_exit
                ld      (hl), a         ; Store byte and
                inc     hl              ; increment pointer
                inc     de
                jr      load_file_loop  ; Process next byte
load_file_exit  call    crlf
                ld      hl, de          ; Print number of bytes loaded
                call    print_word
                ld      hl, load_file_msg_3
                call    puts
                pop     iy
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
load_file_msg_1 defb    "LOAD FILE: ADDR=", eos
load_file_msg_2 defb    " Illegal address!", eos
load_file_msg_3 defb    " bytes loaded.", cr, lf, eos
load_file_msg_4 defb    " FILENAME=", eos
;
; mount - a wrapper for fatmount (necessary for printing the command's name)
;
mount           push    hl
                ld      hl, mount_msg
                call    puts
                call    fatmount
                pop     hl
                ret
mount_msg       defb    "MOUNT", cr, lf, cr, lf, eos
;
; Move a memory block - the user is prompted for all necessary data:
;
move            push    af              ; We won't even destroy the flags!
                push    bc
                push    de
                push    hl
                ld      hl, move_msg_1
                call    puts
                call    get_word        ; Get address of block to be moved
                push    hl              ; Push this address
                ld      hl, move_msg_2
                call    puts
```

```
                call    get_word        ; Get destination start address
                ld      de, hl          ; LDIR requires this in DE
                ; Is the destination address in RAM area?
                and     a               ; Clear carry
                ld      bc, ram_start
                sbc     hl, bc          ; Is the destination in RAM?
                jr      nc, move_get_length
                ld      hl, move_msg_4  ; No - print error message
                call    puts
                pop     hl              ; Clean up stack
                jr      move_exit
move_get_length ld      hl, move_msg_3
                call    puts
                call    get_word        ; Get length of block
                ld      bc, hl          ; LDIR requires the length in BC
                pop     hl              ; Get address of block to be moved
                ; I was lazy - there is no test to make sure that the block
                ; to be moved will fit into the RAM area.
                ldir                    ; Move block
move_exit       call    crlf            ; Finished
                pop     hl              ; Restore registers
                pop     de
                pop     bc
                pop     af
                ret
move_msg_1      defb    "MOVE: FROM=", eos
move_msg_2      defb    " TO=", eos
move_msg_3      defb    " LENGTH=", eos
move_msg_4      defb    " Illegal destination address!", eos
;
; Dump the contents of both register banks:
;
rdump           push    af
                push    hl
                ld      hl, rdump_msg_1 ; Print first two lines
                call    puts
                pop     hl
                call    rdump_one_set
                exx
                ex      af, af'
                push    hl
                ld      hl, rdump_msg_2
                call    puts
                pop     hl
                call    rdump_one_set
                ex      af, af'
                exx
                push    hl
                ld      hl, rdump_msg_3
                call    puts
                push    ix
                pop     hl
                call    print_word
                ld      hl, rdump_msg_4
                call    puts
                push    iy
                pop     hl
                call    print_word
                ld      hl, rdump_msg_5
                call    puts
                ld      hl, 0
                add     hl, sp
                call    print_word
                call    crlf
                pop     hl
                pop     af
                ret
rdump_msg_1     defb    "REGISTER DUMP", cr, lf, cr, lf, tab, "1st:", eos
rdump_msg_2     defb    tab, "2nd:", eos
rdump_msg_3     defb    tab, "PTR: IX=", eos
rdump_msg_4     defb    " IY=", eos
rdump_msg_5     defb    " SP=", eos
;
rdump_one_set   push    hl              ; Print one register set
```

```
                ld      hl, rdump_os_msg_1
                call    puts
                push    af              ; Move AF into HL
                pop     hl
                call    print_word      ; Print contents of AF
                ld      hl, rdump_os_msg_2
                call    puts
                ld      hl, bc
                call    print_word      ; Print contents of BC
                ld      hl, rdump_os_msg_3
                call    puts
                ld      hl, de
                call    print_word      ; Print contents of DE
                ld      hl, rdump_os_msg_4
                call    puts
                pop     hl              ; Restore original HL
                call    print_word      ; Print contents of HL
                call    crlf
                ret
rdump_os_msg_1  defb    " AF=", eos
rdump_os_msg_2  defb    " BC=", eos
rdump_os_msg_3  defb    " DE=", eos
rdump_os_msg_4  defb    " HL=", eos
;
; Start a program - this will prompt for a four digital hexadecimal start
; address. A program should end with "jp $0" to enter the monitor again.
;
start           ld      hl, start_msg
                call    puts
                call    get_word        ; Wait for a four-nibble address
                call    crlf
                jp      (hl)            ; Start program (and hope for the best)
start_msg       defb    "START: ADDR=", eos
;
;  unmount - simple wrapper for fatunmount (necessary for printing the command
; name)
;
unmount         push    hl
                ld      hl, unmount_msg
                call    puts
                call    fatunmount
                pop     hl
                ret
unmount_msg     defb    "UNMOUNT", cr, lf, eos
;
;***************************************************************************
;***
;*** String routines
;***
;***************************************************************************
;
; is_hex checks a character stored in A for being a valid hexadecimal digit.
; A valid hexadecimal digit is denoted by a set C flag.
;
is_hex          cp      'F' + 1         ; Greater than 'F'?
                ret     nc              ; Yes
                cp      '0'             ; Less than '0'?
                jr      nc, is_hex_1    ; No, continue
                ccf                     ; Complement carry (i.e. clear it)
                ret
is_hex_1        cp      '9' + 1         ; Less or equal '9*?
                ret     c               ; Yes
                cp      'A'             ; Less than 'A'?
                jr      nc, is_hex_2    ; No, continue
                ccf                     ; Yes - clear carry and return
                ret
is_hex_2        scf                     ; Set carry
                ret
;
; is_print checks if a character is a printable ASCII character. A valid
; character is denoted by a set C flag.
;
is_print        cp      space
                jr      nc, is_print_1
```

```
                ccf
                ret
is_print_1      cp      $7f
                ret
;
; nibble2val expects a hexadecimal digit (upper case!) in A and returns the
; corresponding value in A.
;
nibble2val      cp      '9' + 1         ; Is it a digit (less or equal '9')?
                jr      c, nibble2val_1 ; Yes
                sub     7               ; Adjust for A-F
nibble2val_1    sub     '0'             ; Fold back to 0..15
                and     $f              ; Only return lower 4 bits
                ret
;
; Convert a single character contained in A to upper case:
;
to_upper        cp      'a'             ; Nothing to do if not lower case
                ret     c
                cp      'z' + 1         ; > 'z'?
                ret     nc              ; Nothing to do, either
                and     $5f             ; Convert to upper case
                ret
;
;  Compare two null terminated strings, return >0 / 0 / <0 in A, works like
; strcmp. The routine expects two pointer in HL and DE which will be
; preserved.
;
strcmp          push    de
                push    hl
strcmp_loop     ld      a, (de)
                cp      0                       ; End of first string reached?
                jr      z, strcmp_exit
                cp      (hl)                    ; Compare two characters
                jr      nz, strcmp_exit         ; Different -> exit
                inc     hl
                inc     de
                jr      strcmp_loop
strcmp_exit     sub     (hl)
                pop     hl
                pop     de
                ret
;
;******************************************************************************
;***
;*** IO routines
;***
;******************************************************************************
;
; Send a CR/LF pair:
;
crlf            push    af
                ld      a, cr
                call    putc
                ld      a, lf
                call    putc
                pop     af
                ret
;
; Read a single character from the serial line, result is in A:
;
getc            call    rx_ready
                in      a, (uart_register_0)
                ret
;
; Get a byte in hexadecimal notation. The result is returned in A. Since
; the routine get_nibble is used only valid characters are accepted - the
; input routine only accepts characters 0-9a-f.
;
get_byte        push    bc              ; Save contents of B (and C)
                call    get_nibble      ; Get upper nibble
                rlc     a
                rlc     a
                rlc     a
```

```
                rlc     a
                ld      b, a            ; Save upper four bits
                call    get_nibble      ; Get lower nibble
                or      b               ; Combine both nibbles
                pop     bc              ; Restore B (and C)
                ret
;
; Get a hexadecimal digit from the serial line. This routine blocks until
; a valid character (0-9a-f) has been entered. A valid digit will be echoed
; to the serial line interface. The lower 4 bits of A contain the value of
; that particular digit.
;
get_nibble      call    getc            ; Read a character
                call    to_upper        ; Convert to upper case
                call    is_hex          ; Was it a hex digit?
                jr      nc, get_nibble  ; No, get another character
                call    nibble2val      ; Convert nibble to value
                call    print_nibble
                ret
;
; Get a word (16 bit) in hexadecimal notation. The result is returned in HL.
; Since the routines get_byte and therefore get_nibble are called, only valid
; characters (0-9a-f) are accepted.
;
get_word        push    af
                call    get_byte        ; Get the upper byte
                ld      h, a
                call    get_byte        ; Get the lower byte
                ld      l, a
                pop     af
                ret
;
;  Read a string from STDIN - HL contains the buffer start address,
; B contains the buffer length.
;
gets            push    af
                push    bc
                push    hl
gets_loop       call    getc                    ; Get a single character
                cp      cr                      ; Skip CR characters
                jr      z, gets_loop            ; only LF will terminate input
                call    to_upper
                call    putc                    ; Echo character
                cp      lf                      ; Terminate string at
                jr      z, gets_exit            ; LF or
                ld      (hl), a                 ; Copy character to buffer
                inc     hl
                djnz    gets_loop
gets_exit       ld      (hl), 0                 ; Insert termination byte
                pop     hl
                pop     bc
                pop     af
                ret
;
; print_byte prints a single byte in hexadecimal notation to the serial line.
; The byte to be printed is expected to be in A.
;
print_byte      push    af              ; Save the contents of the registers
                push    bc
                ld      b, a
                rrca
                rrca
                rrca
                rrca
                call    print_nibble    ; Print high nibble
                ld      a, b
                call    print_nibble    ; Print low nibble
                pop     bc              ; Restore original register contents
                pop     af
                ret
;
; print_nibble prints a single hex nibble which is contained in the lower
; four bits of A:
;
```

```
print_nibble    push    af              ; We won't destroy the contents of A
                and     $f              ; Just in case...
                add     '0'             ; If we have a digit we are done here.
                cp      '9' + 1         ; Is the result > 9?
                jr      c, print_nibble_1
                add     'A' - '0' - $a  ; Take care of A-F
print_nibble_1  call    putc            ; Print the nibble and
                pop     af              ; restore the original value of A
                ret
;
; print_word prints the four hex digits of a word to the serial line. The
; word is expected to be in HL.
;
print_word      push    hl
                push    af
                ld      a, h
                call    print_byte
                ld      a, l
                call    print_byte
                pop     af
                pop     hl
                ret
;
; Send a single character to the serial line (a contains the character):
;
putc            call    tx_ready
                out     (uart_register_0), a
                ret
;
; Send a string to the serial line, HL contains the pointer to the string:
;
puts            push    af
                push    hl
puts_loop       ld      a, (hl)
                cp      eos             ; End of string reached?
                jr      z, puts_end     ; Yes
                call    putc
                inc     hl              ; Increment character pointer
                jr      puts_loop       ; Transmit next character
puts_end        pop     hl
                pop     af
                ret
;
; Wait for an incoming character on the serial line:
;
rx_ready        push    af
rx_ready_loop   in      a, (uart_register_5)
                bit     0, a
                jr      z, rx_ready_loop
                pop     af
                ret
;
; Wait for UART to become ready to transmit a byte:
;
tx_ready        push    af
tx_ready_loop   in      a, (uart_register_5)
                bit     5, a
                jr      z, tx_ready_loop
                pop af
                ret
;
;*****************************************************************************
;***
;*** IDE routines
;***
;*****************************************************************************
;
ide_data_low    equ     ide_base + $0
ide_data_high   equ     ide_base + $8
ide_error_code  equ     ide_base + $1
;
;       Bit mapping of ide_error_code register:
;
;               0: 1 = DAM not found
```

```
;               1: 1 = Track 0 not found
;               2: 1 = Command aborted
;               3: Reserved
;               4: 1 = ID not found
;               5: Reserved
;               6: 1 = Uncorrectable ECC error
;               7: 1 = Bad block detected
;
ide_secnum      equ     ide_base + $2
;
;       Typically set to 1 sector to be transf.
;
ide_lba0        equ     ide_base + $3
ide_lba1        equ     ide_base + $4
ide_lba2        equ     ide_base + $5
ide_lba3        equ     ide_base + $6
;
;       Bit mapping of ide_lba3 register:
;
;               0 - 3: LBA bits 24 - 27
;               4    : Master (0) or slave (1) selection
;               5    : Always 1
;               6    : Set to 1 for LBA access
;               7    : Always 1
;
ide_status_cmd  equ     ide_base + $7
;
;       Useful commands (when written):
;
;               $20: Read sectors with retry
;               $30: Write sectors with retry
;               $EC: Identify drive
;
;       Status bits (when read):
;
;               0 = ERR:  1 = Previous command resulted in an error
;               1 = IDX:  Unused
;               2 = CORR: Unused
;               3 = DRQ:  1 = Data Request Ready (sector buffer ready)
;               4 = DSC:  Unused
;               5 = DF:   1 = Write fault
;               6 = RDY:  1 = Ready to accept command
;               7 = BUSY: 1 = Controller is busy executing a command
;
ide_retries     equ     $ff                     ; Number of retries for polls
;
;
;  Get ID information from drive. HL is expected to point to a 512 byte byte
; sector buffer. If carry is set, the function did not complete correctly and
; was aborted.
;
ide_get_id      push    af
                push    bc
                push    hl
                call    ide_ready               ; Is the drive ready?
                jr      c, ide_get_id_err       ; No - timeout!
                ld      a, $a0                  ; Master, no LBA addressing
                out     (ide_status_cmd), a
                call    ide_ready               ; Did the command complete?
                jr      c, ide_get_id_err       ; Timeout!
                ld      a, $ec                  ; Command to read ID
                out     (ide_status_cmd), a     ; Write command to drive
                call    ide_ready               ; Can we proceed?
                jr      c, ide_get_id_err       ; No - timeout, propagate carry
                call    ide_error_check         ; Any errors?
                jr      c, ide_get_id_err       ; Yes - something went wrong
                call    ide_bfr_ready           ; Is the buffer ready to read?
                jr      c, ide_get_id_err       ; No
                ld      hl, buffer              ; Load the buffer's address
                ld      b, $0                   ; We will read 256 words
ide_get_id_lp   in      a, (ide_data_low)       ; Read high (!) byte
                ld      c, a
                in      a, (ide_data_high)      ; Read low (!) byte
                ld      (hl), a
```

```
                inc     hl
                ld      (hl), c
                inc     hl
                djnz    ide_get_id_lp           ; Read next word
                jr      ide_get_id_exit         ; Everything OK, just exit
ide_get_id_err  ld      hl, ide_get_id_msg      ; Print error message
                call    puts
ide_get_id_exit pop     hl
                pop     bc
                pop     af
                ret
ide_get_id_msg  defb    "FATAL(IDE): Aborted!", cr, lf
;
;  Test if the buffer of the IDE disk drive is ready for transfer. If not,
; carry will be set, otherwise carry is reset. The contents of register A will
; be destroyed!
;
ide_bfr_ready   push    bc
                and     a                       ; Clear carry assuming no error
                ld      b, ide_retries          ; How many retries?
ide_bfr_loop    in      a, (ide_status_cmd)     ; Read IDE status register
                bit     3, a                    ; Check DRQ bit
                jr      nz, ide_bfr_exit        ; Buffer is ready
                push    bc
                ld      b, $0                   ; Wait a moment
ide_bfr_wait    nop
                djnz    ide_bfr_wait
                pop     bc
                djnz    ide_bfr_loop            ; Retry
                scf                             ; Set carry to indicate timeout
                ld      hl, ide_bfr_rdy_err
                call    puts
ide_bfr_exit    pop     bc
                ret
ide_bfr_rdy_err defb "FATAL(IDE): ide_bfr_ready timeout!", cr, lf, eos
;
;  Test if there is any error flagged by the drive. If carry is cleared, no
; error occured, otherwise carry will be set. The contents of register A will
; be destroyed.
;
ide_error_check and     a                       ; Clear carry (no err expected)
                in      a, (ide_status_cmd)     ; Read status register
                bit     0, a                    ; Test error bit
                jr      z, ide_ec_exit          ; Everything is OK
                scf                             ; Set carry due to error
ide_ec_exit     ret
;
;  Read a sector from the drive. If carry is set after return, the function did
; not complete correctly due to a timeout. HL is expected to contain the start
; address of the sector buffer while BC and DE contain the sector address
; (LBA3, 2, 1 and 0). Register A's contents will be destroyed!
;
ide_rs          push    bc
                push    hl
                call    ide_ready               ; Is the drive ready?
                jr      c, ide_rs_err           ; No - timeout!
                call    ide_set_lba             ; Setup the drive's registers
                call    ide_ready               ; Everything OK?
                jr      c, ide_rs_err           ; No - timeout!
                ld      a, $20
                out     (ide_status_cmd), a     ; Issue read command
                call    ide_ready               ; Can we proceed?
                jr      c, ide_rs_err           ; No - timeout, set carry
                call    ide_error_check         ; Any errors?
                jr      c, ide_rs_err           ; Yes - something went wrong
                call    ide_bfr_ready           ; Is the buffer ready to read?
                jr      c, ide_rs_err           ; No
                ld      b, $0                   ; We will read 256 words
ide_rs_loop     in      a, (ide_data_low)       ; Read low byte
                ld      (hl), a                 ; Store this byte
                inc     hl
                in      a, (ide_data_high)      ; Read high byte
                ld      (hl), a
                inc     hl
```

```
                djnz    ide_rs_loop             ; Read next word until done
                jr      ide_rs_exit
ide_rs_err      ld      hl, ide_rs_err_msg      ; Print error message
                call    puts
ide_rs_exit     pop     hl
                pop     bc
                ret
ide_rs_err_msg  defb    "FATAL(IDE): ide_rs timeout!", cr, lf, eos
;
;  Write a sector from the drive. If carry is set after return, the function did
; not complete correctly due to a timeout. HL is expected to contain the start
; address of the sector buffer while BC and DE contain the sector address
; (LBA3, 2, 1 and 0). Register A's contents will be destroyed!
;
ide_ws          push    bc
                push    hl
                call    ide_ready               ; Is the drive ready?
                jr      c, ide_ws_err           ; No - timeout!
                call    ide_set_lba             ; Setup the drive's registers
                call    ide_ready               ; Everything OK?
                jr      c, ide_ws_err           ; No - timeout!
                ld      a, $30
                out     (ide_status_cmd), a     ; Issue read command
                call    ide_ready               ; Can we proceed?
                jr      c, ide_ws_err           ; No - timeout, set carry
                call    ide_error_check         ; Any errors?
                jr      c, ide_ws_err           ; Yes - something went wrong
                call    ide_bfr_ready           ; Is the buffer ready to read?
                jr      c, ide_ws_err           ; No
                ld      b, $0                   ; We will write 256 word
ide_ws_loop     ld      a, (hl)                 ; Get first byte from memory
                ld      c, a
                inc     hl
                ld      a, (hl)                 ; Get next byte
                out     (ide_data_high), a      ; Write high byte to controller
                ld      a, c                    ; Recall low byte again
                out     (ide_data_low), a       ; Write low byte -> strobe
                djnz    ide_ws_loop
                jr      ide_ws_exit
ide_ws_err      ld      hl, ide_ws_err_msg      ; Print error message
                call    puts
ide_ws_exit     pop     hl
                pop     bc
                ret
ide_ws_err_msg  defb    "FATAL(IDE): ide_ws timeout!", cr, lf, eos
;
;  Set sector count and LBA registers of the drive. Registers BC and DE contain
; the sector address (LBA 3, 2, 1 and 0).
;
ide_set_lba     push    af
                ld      a, $1                   ; We will transfer
                out     (ide_secnum), a         ; one sector at a time
                ld      a, e
                out     (ide_lba0), a           ; Set LBA0, 1 and 2 directly
                ld      a, d
                out     (ide_lba1), a
                ld      a, c
                out     (ide_lba2), a
                ld      a, b                    ; Special treatment for LBA3
                and     $0f                     ; Only bits 0 - 3 are LBA3
                or      $e0                     ; Select LBA and master drive
                out     (ide_lba3), a
                pop     af
                ret
;
;  Test if the IDE drive is not busy and ready to accept a command. If it is
; ready the carry flag will be reset and the function returns. If a time out
; occurs, C will be set prior to returning to the caller. Register A will
; be destroyed!
;
ide_ready       push    bc
                and     a                       ; Clear carry assuming no error
                ld      b, ide_retries          ; Number of retries to timeout
ide_ready_loop  in      a, (ide_status_cmd)     ; Read drive status
```

```
                and     a, $c0                  ; Only bits 7 and 6 are needed
                xor     $40                     ; Invert the ready flag
                jr      z, ide_ready_exit       ; Exit if ready and not busy
                push    bc
                ld      b, $0                   ; Wait a moment
ide_ready_wait  nop
                djnz    ide_ready_wait
                pop     bc
                djnz    ide_ready_loop          ; Retry
                scf                             ; Set carry due to timeout
                ld      hl, ide_rdy_error
                call    puts
                in      a, (ide_error_code)
                call    print_byte
ide_ready_exit  pop     bc
                ret
ide_rdy_error   defb    "FATAL(IDE): ide_ready timeout!", cr, lf, eos
;
;*****************************************************************************
;***
;*** Miscellaneous functions
;***
;*****************************************************************************
;
; Clear the computer (not to be called - jump into this routine):
;
cold_start      ld      hl, start_type
                ld      (hl), $00
warm_start      ld      hl, clear_msg
                call    puts
                ld      a, $00
                ld      (ram_end), a
                rst     $00
clear_msg       defb    "CLEAR", cr, lf, eos
;
;*****************************************************************************
;***
;*** Mathematical routines
;***
;*****************************************************************************
;
; 32 bit add routine from
;       http://www.andreadrian.de/oldcpu/Z80_number_cruncher.html
;
; ADD ROUTINE 32+32BIT=32BIT
; H'L'HL = H'L'HL + D'E'DE
; CHANGES FLAGS
;
ADD32:  ADD     HL,DE   ; 16-BIT ADD OF HL AND DE
        EXX
        ADC     HL,DE   ; 16-BIT ADD OF HL AND DE WITH CARRY
        EXX
        RET
;
; 32 bit multiplication routine from
;       http://www.andreadrian.de/oldcpu/Z80_number_cruncher.html
;
; MULTIPLY ROUTINE 32*32BIT=32BIT
; H'L'HL = B'C'BC * D'E'DE; NEEDS REGISTER A, CHANGES FLAGS
;
MUL32:  AND     A               ; RESET CARRY FLAG
        SBC     HL,HL           ; LOWER RESULT = 0
        EXX
        SBC     HL,HL           ; HIGHER RESULT = 0
        LD      A,B             ; MPR IS AC'BC
        LD      B,32            ; INITIALIZE LOOP COUNTER
MUL32LOOP:
        SRA     A               ; RIGHT SHIFT MPR
        RR      C
        EXX
        RR      B
        RR      C               ; LOWEST BIT INTO CARRY
        JR      NC,MUL32NOADD
        ADD     HL,DE           ; RESULT += MPD
```

```
        EXX
        ADC     HL,DE
        EXX
MUL32NOADD:
        SLA     E               ; LEFT SHIFT MPD
        RL      D
        EXX
        RL      E
        RL      D
        DJNZ    MUL32LOOP
        EXX
        RET
;
;*****************************************************************************
;***
;*** FAT file system routines
;***
;*****************************************************************************
;
;  Read a single byte from a file. IY points to the FCB. The byte read is
; returned in A, on EOF the carry flag will be set.
;
fgetc           push    bc
                push    de
                push    hl
;  Check if fcb_file_pointer == fcb_file_size. In this case we have reached
; EOF and will return with a set carry bit. (As a side effect, the attempt to
; read from a file which has not been successfully opened before will be
; handled like encountering an EOF at the first fgetc call.)
                ld      a, (iy + fcb_file_size)
                cp      (iy + fcb_file_pointer)
                jr      nz, fgetc_start
                ld      a, (iy + fcb_file_size + 1)
                cp      (iy + fcb_file_pointer + 1)
                jr      nz, fgetc_start
                ld      a, (iy + fcb_file_size + 2)
                cp      (iy + fcb_file_pointer + 2)
                jr      nz, fgetc_start
                ld      a, (iy + fcb_file_size + 3)
                cp      (iy + fcb_file_pointer + 3)
                jr      nz, fgetc_start
; We have reached EOF, so set carry and leave this routine:
                scf
                jp      fgetc_exit
;  Check if the lower 9 bits of the file pointer are zero. In this case
; we need to read another sector (maybe from another cluster):
fgetc_start     ld      a, (iy + fcb_file_pointer)
                cp      0
                jp      nz, fgetc_getc          ; Bits 0-7 are not zero
                ld      a, (iy + fcb_file_pointer + 1)
                and     1
                jp      nz, fgetc_getc          ; Bit 8 is not zero
; The file_pointer modulo 512 is zero, so we have to load the next sector:
; We have to check if fcb_current_cluster == 0 which will be the case in the
; initial run. Then we will copy fcb_first_cluster into fcb_current_cluster.
                ld      a, (iy + fcb_current_cluster)
                cp      0
                jr      nz, fgetc_continue      ; Not the initial case
                ld      a, (iy + fcb_current_cluster + 1)
                cp      0
                jr      nz, fgetc_continue      ; Not the initial case
; Initial case: We have to fill fcb_current_cluster with fcb_first_cluste:
                ld      a, (iy + fcb_first_cluster)
                ld      (iy + fcb_current_cluster), a
                ld      a, (iy + fcb_first_cluster + 1)
                ld      (iy + fcb_current_cluster + 1), a
                jr      fgetc_clu2sec
; Here is the normal case - we will check if fcb_cluster_sector is zero -
; in this case we have to determine the next sector to be loaded by looking
; up the FAT. Otherwise (fcb_cluster_sector != 0) we will just get the next
; sector in the current cluster.
fgetc_continue  ld      a, (iy + fcb_cluster_sector)
                jr      nz, fgetc_same          ; The current cluster is valid
;  Here we know that we need the first sector of the next cluster of the file.
```

```
; The upper eight bits of the fcb_current_cluster point to the sector of the
; FAT where the entry we are looking for is located (this is true since a
; sector contains 512 bytes which corresponds to 256 FAT entries). So we must
; load the sector with the number fatstart + fcb_current_cluster[15-8] into
; the IDE buffer and locate the entry with the address
; fcb_current_cluster[7-0] * 2. This entry contains the sector number we are
; looking for.
                ld      hl, (fat1start)
                ld      c, (iy + fcb_current_cluster + 1)
                ld      b, 0
                add     hl, bc
                ld      de, hl                  ; Needed for ide_rs
                ld      bc, 0
                ld      hl, (fat1start + 2)
                adc     hl, bc
                ld      bc, hl                  ; Needed for ide_rs
                ld      hl, buffer
                call    ide_rs
;  Now the sector containing the FAT entry we are looking for is available in
; the IDE buffer. Now we need fcb_current_cluster[7-0] * 2
                ld      b, 0
                ld      c, (iy + fcb_current_cluster)
                sla     c
                rl      b
; Now get the entry:
                ld      hl, buffer
                add     hl, bc
                ld      bc, (hl)
                ld      (iy + fcb_current_cluster), c
                ld      (iy + fcb_current_cluster), b
; Now we determine the first sector of the cluster to be read:
fgetc_clu2sec   ld      a, (clusiz)             ; Initialize fcb_cluster_sector
                ld      (iy + fcb_cluster_sector), a
                ld      l, (iy + fcb_current_cluster)
                ld      h, (iy + fcb_current_cluster + 1)
                call    clu2sec                 ; Convert cluster to sector
                jr      fgetc_rs
fgetc_same      and     a                       ; Clear carry
                ld      bc, 1                   ; Increment fcb_current_sector
                ld      l, (iy + fcb_current_sector)
                ld      h, (iy + fcb_current_sector + 1)
                add     hl, bc
                ld      (iy + fcb_current_sector), l
                ld      e, l                    ; Needed for ide_rs
                ld      (iy + fcb_current_sector + 1), h
                ld      d, h                    ; Needed for ide_rs
                ld      l, (iy + fcb_current_sector + 2)
                ld      h, (iy + fcb_current_sector + 3)
                ld      bc, 0
                adc     hl, bc
                ld      (iy + fcb_current_sector + 2), l
                ld      c, l                    ; Needed for ide_rs
                ld      (iy + fcb_current_sector + 3), h
                ld      b, h                    ; Neede for ide_rs
fgetc_rs        ld      (iy + fcb_current_sector), e    ; Now read the sector
                ld      (iy + fcb_current_sector + 1), d
                ld      (iy + fcb_current_sector + 2), c
                ld      (iy + fcb_current_sector + 3), b
; Let HL point to the sector buffer in the FCB:
                push    iy                      ; Start of FCB
                pop     hl
                push    bc
                ld      bc, fcb_file_buffer     ; Displacement of sector buffer
                add     hl, bc
                pop     bc
                call    ide_rs                  ; Read a single sector from disk
; Since we have read a sector we have to decrement fcb_cluster_sector
                dec     (iy + fcb_cluster_sector)
; Here we read and return a single character from the sector buffer:
fgetc_getc      push    iy
                pop     hl                      ; Copy IY to HL
                ld      bc, fcb_file_buffer
                add     hl, bc                  ; HL points to the sector bfr.
; Get the lower 9 bits of the file pointer as displacement for the buffer:
```

```
                ld      c, (iy + fcb_file_pointer)
                ld      a, (iy + fcb_file_pointer + 1)
                and     1                       ; Get rid of bits 9-15
                ld      b, a
                add     hl, bc                  ; Add byte offset
                ld      a, (hl)                 ; get one byte from buffer
; Increment the file pointer:
                ld      l, (iy + fcb_file_pointer)
                ld      h, (iy + fcb_file_pointer + 1)
                ld      bc, 1
                add     hl, bc
                ld      (iy + fcb_file_pointer), l
                ld      (iy + fcb_file_pointer + 1), h
                ld      bc, 0
                ld      l, (iy + fcb_file_pointer + 2)
                ld      h, (iy + fcb_file_pointer + 3)
                adc     hl, bc
                ld      (iy + fcb_file_pointer + 2), l
                ld      (iy + fcb_file_pointer + 3), h
;
                and     a                       ; Clear carry
fgetc_exit      pop     hl
                pop     de
                pop     bc
                ret
;
;  Clear the FCB to which IY points -- this should be called every time one
; creates a new FCB. (Please note that fopen does its own call to clear_fcb.)
;
clear_fcb       push    af                      ; We have to save so many
                push    bc                      ; Registers since the FCB is
                push    de                      ; cleared using LDIR.
                push    hl
                ld      a, 0
                push    iy
                pop     hl
                ld      (hl), a                 ; Clear first byte of FCB
                ld      de, hl
                inc     de
                ld      bc, fcb_file_buffer
                ldir                            ; And transfer this zero byte
                pop     hl                      ; down to the relevant rest
                pop     de                      ; of the buffer.
                pop     bc
                pop     af
                ret
;
; Dump a file control block (FCB) - the start address is expected in IY.
;
dump_fcb        push    af
                push    hl
                ld      hl, dump_fcb_1
                call    puts
                push    iy                      ; Load HL with
                pop     hl                      ; the contents of IY
                call    print_word
; Print the filename:
                ld      hl, dump_fcb_2
                call    puts
                push    iy
                pop     hl
                call    puts
; Print file size:
                ld      hl, dump_fcb_3
                call    puts
                ld      h, (iy + fcb_file_size + 3)
                ld      l, (iy + fcb_file_size + 2)
                call    print_word
                ld      h, (iy + fcb_file_size + 1)
                ld      l, (iy + fcb_file_size)
                call    print_word
; Print cluster number:
                ld      hl, dump_fcb_4
                call    puts
```

```
                ld      h, (iy + fcb_first_cluster + 1)
                ld      l, (iy + fcb_first_cluster)
                call    print_word
; Print file type:
                ld      hl, dump_fcb_5
                call    puts
                ld      a, (iy + fcb_file_type)
                call    print_byte
; Print file pointer:
                ld      hl, dump_fcb_6
                call    puts
                ld      h, (iy + fcb_file_pointer + 3)
                ld      l, (iy + fcb_file_pointer + 2)
                call    print_word
                ld      h, (iy + fcb_file_pointer + 1)
                ld      l, (iy + fcb_file_pointer)
                call    print_word
; Print current cluster number:
                ld      hl, dump_fcb_7
                call    puts
                ld      h, (iy + fcb_current_cluster + 1)
                ld      l, (iy + fcb_current_cluster)
                call    print_word
; Print current sector:
                ld      hl, dump_fcb_8
                call    puts
                ld      h, (iy + fcb_current_sector + 3)
                ld      l, (iy + fcb_current_sector + 2)
                call    print_word
                ld      h, (iy + fcb_current_sector + 1)
                ld      l, (iy + fcb_current_sector)
                call    print_word
                call    crlf
                pop     hl
                pop     af
                ret
dump_fcb_1      defb    "Dump of FCB at address: ", eos
dump_fcb_2      defb    cr, lf, tab, "File name      : ", eos
dump_fcb_3      defb    cr, lf, tab, "File size      : ", eos
dump_fcb_4      defb    cr, lf, tab, "1st cluster    : ", eos
dump_fcb_5      defb    cr, lf, tab, "File type      : ", eos
dump_fcb_6      defb    cr, lf, tab, "File pointer   : ", eos
dump_fcb_7      defb    cr, lf, tab, "Current cluster: ", eos
dump_fcb_8      defb    cr, lf, tab, "Current sector : ", eos
;
;  Convert a user specified filename to an 8.3-filename without dot and
; with terminating null byte. HL points to the input string, DE points to
; a 12 character buffer for the filename. This function is used by
; fopen which expects a human readable string that will be transformed into
; an 8.3-filename without the dot for the following directory lookup.
;
str2filename    push    af
                push    bc
                push    de
                push    hl
                ld      (str2filename_de), de
                ld      a, ' '                  ; Initialize output buffer
                ld      b, $b                   ; Fill 11 bytes with spaces
str2filiniloop  ld      (de), a
                inc     de
                djnz    str2filiniloop
                ld      a, 0                    ; Add terminating null byte
                ld      (de), a
                ld      de, (str2filename_de)   ; Restore DE pointer
; Start string conversion
                ld      b, 8
str2filini_nam  ld      a, (hl)
                cp      0                       ; End of string reached?
                jr      z, str2filini_x
                cp      '.'                     ; Dot found?
                jr      z, str2filini_ext
                ld      (de), a
                inc     de
                inc     hl
```

```
                dec     b
                jr      nz, str2filini_nam
str2filini_skip ld      a, (hl)
                cp      0                       ; End of string without dot?
                jr      z, str2filini_x         ; Nothing more to do
                cp      '.'
                jr      z, str2filini_ext       ; Take care of extension
                inc     hl                      ; Prepare for next character
                jr      str2filini_skip         ; Skip more characters
str2filini_ext  inc     hl                      ; Skip the dot
                push    hl                      ; Make sure DE points
                ld      hl, (str2filename_de)   ; into the filename buffer
                ld      bc, 8                   ; at the start position
                add     hl, bc                  ; of the filename extension
                ld      de, hl
                pop     hl
                ld      b, 3
str2filini_elp  ld      a, (hl)
                cp      0                       ; End of string reached?
                jr      z, str2filini_x         ; Nothing more to do
                ld      (de), a
                inc     de
                inc     hl
                dec     b
                jr      nz, str2filini_elp      ; Next extension character
str2filini_x    pop     hl
                pop     de
                pop     bc
                pop     af
                ret
;
;  Open a file with given filename (format: 'FFFFFFFFXXX') in the root directory
; and return the 1st cluster number for that file. If the file can not
; be found, $0000 will be returned.
;  At entry, HL must point to the string buffer while IY points to a valid
; file control block that will hold all necessary data for future file accesses.
;
fopen   push    af
                push    bc
                push    de
                push    ix
                ld      (fopen_scr), hl
                ld      hl, fatname             ; Check if a disk has been
                ld      a, (hl)                 ; mounted.
                cp      0
                jp      z, fopen_e1             ; No disk - error exit
                call    clear_fcb
                push    iy                      ; Copy IY to DE
                pop     de
                ld      hl, (fopen_scr) ; Create the filename
                call    str2filename            ; Convert string to a filename
                ld      hl, buffer              ; Compute buffer overflow
                ld      bc, $0200               ; address - this is the bfr siz.
                add     hl, bc                  ; and will be used in the loop
                ld      (fopen_eob), hl         ; This is the buffer end addr.
;
                ld      hl, (rootstart)         ; Remember the initial root
                ld      (fopen_rsc), hl         ; sector number
                ld      hl, (rootstart + 2)
                ld      (fopen_rsc + 2), hl
; Read one root directory sector
fopen_nbf       ld      bc, (fopen_rsc + 2)
                ld      de, (fopen_rsc)
                ld      hl, buffer
                call    ide_rs                  ; Read one sector
                jp      c, fopen_e2             ; Exit on read error
fopen_lp        ld      (fopen_scr), hl
                xor     a                       ; Last entry?
                cp      (hl)                    ; The last entry has first
                jp      z, fopen_x              ; byte = $0
                ld      a, $e5                  ; Deleted entry?
                cp      (hl)
                jr      z, fopen_nxt            ; Get next entry
;                ld      (fopen_scr), hl
```

```
                ld      ix, (fopen_scr)
                ld      a, (ix + $b)            ; Get attribute byte
                cp      $0f
                jr      z, fopen_nxt            ; Skip long name
                bit     4, a                    ; Skip directories
                jr      nz, fopen_nxt
; Compare the filename with the one we are looking for:
                ld      (ix + $b), 0            ; Clear attribute byte
                ld      de, (fopen_scr)
                push    iy                      ; Prepare string comparison
                pop     hl
                call    strcmp                  ; Compare filename with string
                cp      0                       ; Are strings equal?
                jr      nz, fopen_nxt           ; No - check next entry
                ld      a, (ix + $1a + 1)       ; Read cluster number and
; Save cluster_number into fcb_first_cluster:
                ld      (iy + fcb_first_cluster + 1), a
                ld      a, (ix + $1a)
                ld      (iy + fcb_first_cluster), a
                ld      a, (ix + $1c)           ; Save file size to FCB
                ld      (iy + fcb_file_size), a
                ld      a, (ix + $1d)           ; Save file size to FCB
                ld      (iy + fcb_file_size + 1), a
                ld      a, (ix + $1e)           ; Save file size to FCB
                ld      (iy + fcb_file_size + 2), a
                ld      a, (ix + $1f)           ; Save file size to FCB
                ld      (iy + fcb_file_size + 3), a
                ld      (iy + fcb_file_type), 1 ; Set file type to found
                jr      fopen_x                 ; Terminate lookup loop
fopen_nxt       ld      bc, $20
                ld      hl, (fopen_scr)
                add     hl, bc
                ld      (fopen_scr), hl
                ld      bc, (fopen_eob)         ; Check for end of buffer
                and     a                       ; Clear carry
                sbc     hl, bc                  ; ...no 16 bit cp :-(
                jp      nz, fopen_lp            ; Buffer is still valid
                ld      hl, fopen_rsc           ; Increment sector number
                inc     (hl)                    ; 16 bits are enough :-)
                jp      fopen_nbf               ; Read next directory sector
fopen_e1        ld      hl, fopen_nmn           ; No disk mounted
                jr      fopen_err               ; Print error message
fopen_e2        ld      hl, fopen_rer           ; Directoy sector read error
fopen_err       call    puts
fopen_x         pop     ix
                pop     de
                pop     bc
                pop     af
                ret
fopen_nmn       defb    "FATAL(FOPEN): No disk mounted!", cr, lf, eos
fopen_rer       defb    "FATAL(FOPEN): Could not read directory sector!"
                defb    cr, lf, eos
;
;  Convert a cluster number into a sector number. The cluster number is
; expected in HL, the corresponding sector number will be returned in
; BC and DE, thus ide_rs or ide_ws can be called afterwards.
;
; SECNUM = (CLUNUM - 2) * CLUSIZ + DATASTART
;
clu2sec         push    af                      ; Since the 32 bit
                push    hl                      ; multiplication routine
                exx                             ; needs shadow registers
                push    bc                      ; we have to push many,
                push    de                      ; many registers here
                push    hl
                ld      bc, 0                   ; Clear BC' and DE' for
                ld      de, bc                  ; 32 bit multiplication
                exx
                ld      bc, 2                   ; Subtract 2
                sbc     hl, bc                  ; HL = CLUNUM - 2
                ld      bc, hl                  ; BC = HL; BC' = 0
                ld      a, (clusiz)
                ld      d, 0                    ; CLUSIZ bits 8 to 15
                ld      e, a                    ; DE = CLUSIZ
```

```
                call    MUL32                   ; HL = (CLUNUM - 2) * CLUSIZ
                ld      de, (datastart)
                exx
                ld      de, (datastart + 2)
                exx
                call    ADD32                   ; HL = HL + DATASTART
                exx
                push    hl
                exx
                pop     bc
                ld      de, hl
                exx
                pop     hl
                pop     de
                pop     bc
                exx
                pop     hl
                pop     af
                ret
;
; Print a directory listing
;
dirlist         push    af
                push    bc
                push    de
                push    hl
                push    ix
                ld      hl, fatname
                ld      a, (hl)
                cp      0
                jp      z, dirlist_nodisk
                ld      ix, string_81_bfr
                ld      (ix + 8), '.'           ; Dot between name and extens.
                ld      (ix + 12), 0            ; String terminator
                ld      hl, dirlist_0           ; Print title line
                call    puts
                ld      hl, buffer              ; Compute buffer overflow
                ld      bc, $0200               ; address - this is the bfr siz.
                add     hl, bc
                ld      (dirlist_eob), hl       ; This is the buffer end addr.
;
                ld      hl, (rootstart)         ; Remember the initial root
                ld      (dirlist_rootsec), hl   ; sector number
                ld      hl, (rootstart + 2)
                ld      (dirlist_rootsec + 2), hl
; Read one root directory sector
dirlist_nbfr    ld      bc, (dirlist_rootsec + 2)
                ld      de, (dirlist_rootsec)
                ld      hl, buffer
                call    ide_rs
                jp      c, dirlist_e1
dirlist_loop    xor     a                       ; Last entry?
                cp      (hl)                    ; The last entry has first
                jp      z, dirlist_exit         ; byte = $0
                ld      a, $e5                  ; Deleted entry?
                cp      (hl)
                jr      z, dirlist_next
                ld      (dirlist_scratch), hl
                ld      ix, (dirlist_scratch)
                ld      a, (ix + $b)            ; Get attribute byte
                cp      $0f
                jr      z, dirlist_next         ; Skip long name
                ld      de, string_81_bfr       ; Prepare for output
                ld      bc, 8                   ; Copy first eight characters
                ldir
                inc     de
                ld      bc, 3                   ; Copy extension
                ldir
;                ld      hl, de
;                ld      (hl), 0                 ; String terminator
                ld      hl, string_81_bfr
                call    puts
                ld      hl, dirlist_NODIR       ; Flag directories with "DIR"
                bit     4, a
```

```
                jr      z, dirlist_prtdir
                ld      hl, dirlist_DIR
dirlist_prtdir  call    puts
                ld      h, (ix + $1c + 3)       ; Get and print file size
                ld      l, (ix + $1c + 2)
                call    print_word
                ld      h, (ix + $1c + 1)
                ld      l, (ix + $1c)
                call    print_word
; Get and print start sector
                ld      a, tab
                call    putc
                ld      h, (ix + $1a + 1)       ; Get cluster number
                ld      l, (ix + $1a)
                ld      bc, 0                   ; Is file empty?
                and     a                       ; Clear carry
                sbc     hl, bc                  ; Empty file -> Z set
                jr      z, dirlist_nosize
                call    clu2sec
                ld      hl, bc
                call    print_word
                ld      hl, de
                call    print_word
dirlist_nosize  call    crlf
                ld      hl, (dirlist_scratch)
dirlist_next    ld      bc, $20
                add     hl, bc
                ld      bc, (dirlist_eob)       ; Check for end of buffer
                and     a
                sbc     hl, bc
                jp      nz, dirlist_loop        ; Buffer is still valid
                ld      hl, dirlist_rootsec
                inc     (hl)
                jp      dirlist_nbfr
dirlist_e1      ld      hl, dirlist_1
                jr      dirlist_x
dirlist_nodisk  ld      hl, dirlist_nomnt
dirlist_x       call    puts
dirlist_exit    pop     ix
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
dirlist_nomnt   defb    "FATAL(DIRLIST): No disk mounted!", cr, lf, eos
dirlist_0       defb    "Directory contents:", cr, lf
                defb    "------------------------------------------", cr, lf
                defb    "FILENAME.EXT  DIR?   SIZE (BYTES)"
                defb    "  1ST SECT", cr, lf
                defb    "------------------------------------------", cr, lf
                defb    eos
dirlist_1       defb    "FATAL(DIRLIST): Could not read directory sector"
                defb    cr, lf, eos
dirlist_DIR     defb    tab, "DIR", tab, eos
dirlist_NODIR   defb    tab, tab, eos
;
; Perform a disk mount
;
fatmount        push    af
                push    bc
                push    de
                push    hl
                push    ix
                ld      hl, buffer              ; Read MBR into buffer
                ld      bc, 0
                ld      de, 0
                call    ide_rs
                jp      c, fatmount_e1          ; Error reading MBR?
                ld      ix, buffer + $1fe       ; Check for $55AA as MBR trailer
                ld      a, $55
                cp      (ix)
                jp      nz, fatmount_e2
                ld      a, $aa
                cp      (ix + 1)
```

```
                jp      nz, fatmount_e2
                ld      bc, 8                   ; Get partition start and size
                ld      hl, buffer + $1c6
                ld      de, pstart
                ldir
                ld      hl, buffer              ; Read partition boot block
                ld      de, (pstart)
                ld      bc, (pstart + 2)
                call    ide_rs
                jp      c, fatmount_e3          ; Error reading boot block?
                ld      bc, 8                   ; Copy FAT name
                ld      hl, buffer + 3
                ld      de, fatname
                ldir
                ld      ix, buffer
                ld      a, 2                    ; Check for two FATs
                cp      (ix + $10)
                jp      nz, fatmount_e4         ; Wrong number of FATs
                xor     a                       ; Check for 512 bytes / sector
                cp      (ix + $b)
                jp      nz, fatmount_e5
                ld      a, 2
                cp      (ix + $c)
                jp      nz, fatmount_e5
                ld      a, (buffer + $d)        ; Get cluster size
                ld      (clusiz), a
                ld      bc, (buffer + $e)       ; Get reserved sector number
                ld      (ressec), bc
                ld      bc, (buffer + $16)      ; Get FAT size in sectors
                ld      (fatsec), bc
                ld      bc, (buffer + $11)      ; Get length of root directory
                ld      (rootlen), bc
                ld      hl, (pstart)            ; Compute
                ld      bc, (ressec)            ; FAT1START = PSTART + RESSEC
                add     hl, bc
                ld      (fat1start), hl
                ld      hl, (pstart + 2)
                ld      bc, 0
                adc     hl, bc
                ld      (fat1start + 2), hl
                ld      hl, (fatsec)            ; Compute ROOTSTART for two FATs
                add     hl, hl                  ; ROOTSTART = FAT1START +
                ld      bc, hl                  ;             2 * FATSIZ
                ld      hl, (fat1start)
                add     hl, bc
                ld      (rootstart), hl
                ld      hl, (fat1start + 2)
                ld      bc, 0
                adc     hl, bc
                ld      (rootstart + 2), hl
                ld      bc, (rootlen)           ; Compute rootlen / 16
                sra     b                       ; By shifting it four places
                rr      c                       ; to the right
                sra     b                       ; This value will be used
                rr      c                       ; for the calculation of
                sra     b                       ; DATASTART
                rr      c
                sra     b
                rr      c
                ld      hl, (rootstart)         ; Computer DATASTART
                add     hl, bc
                ld      (datastart), hl
                ld      hl, (rootstart + 2)
                ld      bc, 0
                adc     hl, bc
                ld      (datastart + 2), hl
                ld      hl, fatmount_s1         ; Print mount summary
                call    puts
                ld      hl, fatname
                call    puts
                ld      hl, fatmount_s2
                call    puts
                ld      a, (clusiz)
                call    print_byte
```

```
                ld      hl, fatmount_s3
                call    puts
                ld      hl, (ressec)
                call    print_word
                ld      hl, fatmount_s4
                call    puts
                ld      hl, (fatsec)
                call    print_word
                ld      hl, fatmount_s5
                call    puts
                ld      hl, (rootlen)
                call    print_word
                ld      hl, fatmount_s6
                call    puts
                ld      hl, (psiz + 2)
                call    print_word
                ld      hl, (psiz)
                call    print_word
                ld      hl, fatmount_s7
                call    puts
                ld      hl, (pstart + 2)
                call    print_word
                ld      hl, (pstart)
                call    print_word
                ld      hl, fatmount_s8
                call    puts
                ld      hl, (fat1start + 2)
                call    print_word
                ld      hl, (fat1start)
                call    print_word
                ld      hl, fatmount_s9
                call    puts
                ld      hl, (rootstart + 2)
                call    print_word
                ld      hl, (rootstart)
                call    print_word
                ld      hl, fatmount_sa
                call    puts
                ld      hl, (datastart + 2)
                call    print_word
                ld      hl, (datastart)
                call    print_word
                call    crlf
                jr      fatmount_exit
fatmount_e1     ld      hl, fatmount_1
                jr      fatmount_x
fatmount_e2     ld      hl, fatmount_2
                jr      fatmount_x
fatmount_e3     ld      hl, fatmount_3
                jr      fatmount_x
fatmount_e4     ld      hl, fatmount_4
                jr      fatmount_x
fatmount_e5     ld      hl, fatmount_5
fatmount_x      call    puts
fatmount_exit   pop     ix
                pop     hl
                pop     de
                pop     bc
                pop     af
                ret
fatmount_1      defb    "FATAL(FATMOUNT): Could not read MBR!", cr, lf, eos
fatmount_2      defb    "FATAL(FATMOUNT): Illegal MBR!", cr, lf, eos
fatmount_3      defb    "FATAL(FATMOUNT): Could not read partition boot block"
                defb    cr, lf, eos
fatmount_4      defb    "FATAL(FATMOUNT): FAT number not equal two!"
                defb    cr, lf, eos
fatmount_5      defb    "FATAL(FATMOUNT): Sector size not equal 512 bytes!"
                defb    cr, lf, eos
fatmount_s1     defb    tab, "FATNAME:", tab, eos
fatmount_s2     defb    cr, lf, tab, "CLUSIZ:", tab, eos
fatmount_s3     defb    cr, lf, tab, "RESSEC:", tab, eos
fatmount_s4     defb    cr, lf, tab, "FATSEC:", tab, eos
fatmount_s5     defb    cr, lf, tab, "ROOTLEN:", tab, eos
fatmount_s6     defb    cr, lf, tab, "PSIZ:", tab, tab, eos
```

```
fatmount_s7     defb    cr, lf, tab, "PSTART:", tab, eos
fatmount_s8     defb    cr, lf, tab, "FAT1START:", tab, eos
fatmount_s9     defb    cr, lf, tab, "ROOTSTART:", tab, eos
fatmount_sa     defb    cr, lf, tab, "DATASTART:", tab, eos
;
;  Dismount a FAT volume (invalidate the FAT control block by setting the
; first byte (of fatname) to zero.
;
fatunmount      push    af
                push    hl
                xor     a
                ld      hl, fatname
                ld      (hl), a
                pop     hl
                pop     af
                ret
;
                defb    "THE MONITOR ENDS HERE...", eos
```

ulmann@vaxman.de

webmaster@vaxman.de

02-OCT-2011, 08-JAN-2012, 20-FEB-2012