# Table des matières

# USER MANUAL OF SPECTRUM FORTH83

## 1. GENERAL

This manual is a translation of the user manual that I wrote originally in Dutch for the SPECTRUM FORTH83 system in 1988. This is not a literal translation and it contains some updates and clarifications.

**This FORTH system is written by**
**L.C. Benschop in Vught The Netherlands** (formerly living in Eindhoven).

I made use of the assembler and some code definitions designed by Coos Haak in Utrecht, The Netherlands. Some high-level definitions were derived from FIG-FORTH, the rest is my original work, in particular the screen editor and all Spectrum-specific words. This manual is not meant
to be an exact definition of the FORTH-83 standard. For this, please refer to:

**FORTH-83 STANDARD**
**FORTH STANDARDS TEAM**
**P.O. BOX 4545**
**MOUNTAIN VIEW CA94040 USA**

Paper copies used to be available from FIG or its local chapters. Currently it can be found online here:

http://forth.sourceforge.net/standard/fst83/

## 1.2 INTRODUCTION

This Forth compiler is suitable for the ZX-Spectrum 48 and 128, with or without Microdrives or floppy disk drives. The program contains the full FORTH-83 Required Word Set and System Extension Word Set. The Double Number Extension Word Set and Assembler Word Set can be loaded as extensions. Further there are extensions for floating point numbers, strings and graphics. The complete system with or without any of the above-mentioned extension word sets can be saved to tape, Microdrive of floppy disk.

This system is block-oriented, which means that source code is stored in fixed-sized blocks. These blocks (also called screens) are stored in a RAM disk and they can be saved to tape, Microdrive or floppy disk. One or more of these blocks can be saved to a single file. These files can later be reloaded into the RAM disk. This RAM disk is located at the top part of the RAM on the Spectrum 48 and its size can be varied. On the Spectrum 128 on the other hand, it is always 80 kilobytes in size and it is located in the extra RAM that is not available on the Spectrum 48.

The system comes standard with a screen-editor that can also edit large contiguous texts that are stored in multiple blocks in the RAM disk. The original Dutch version of this manual was edited this way.

The BREAK key is capable of interrupting almost any program, be it FORTH or machine code.

## 1.3 LOADING AND SAVING FORTH.

On the Spectrum 128, you must use 128K Basic or Tape Loader to load Forth from cassette. On the Spectrum 48 you must type LOAD "" from Basic.
Forth will start automatically when loaded from tape.

If you have trouble running FORTH on a Spectrum 48, it may be caused by hardware extensions that use port address 7FFDH. In this case, do the following:

- Load the Basic program from tape with LOAD "".
- Press BREAK to interrupt the loading procedure.
- Manually load the rest of Forth with LOAD "" CODE 27028
- Patch the system with

```
 POKE 30388,250
 POKE 30389,111
```

This patch replaces the P! address in the >BANK word with 2DROP, so this word will no longer write to the bank switching port.
- Start with RUN 20.

Once you are in Forth, you can save a copy of the program on cassette tape with:

```
 0 DRIVE 60 BCAL
```

You can save a copy of the program on Microdrive or floppy disk with:

```
 1 DRIVE 60 BCAL
```

But when there is already a version on disk or Microdrive, you have to delete the old files first with:

```
 1 DRIVE DELETE run DELETE FORT83.BIN
```

It will be stored on drive no.1. Once it is stored on disk or Microdrive you can later load it by typing just the RUN command immediately after starting the machine. On the Spectrum 128 this has to be done from 128K Basic.

If you use a floppy disk, it may be necessary to edit the SAVE and LOAD commands in the Basic program, depending on the disk interface you use.

From Forth you can return to Basic by pressing SYMBOL-SHIFT-W.

From Basic you can return to Forth by typing the RUN command.

If you have loaded extension word sets and you want to save the extended Forth system, you have to type the following commands first before you can actually save the system with 60 BCAL:

**HERE FENCE !**

On the Spectrum 48 it is possible to change the size of the RAM disk by entering the following commands:

**N ' #B >BODY ! COLD**

Instead of N, type the desired number of screens, for instance 8 or 16 (default is 10).

## 1.4 SCREEN, KEYBOARD AND PRINTER

In FORTH the screen uses 24 lines of 32 characters each. As opposed to Basic, all 24 lines are used and the bottom two lines are not reserved for error messages and keyboard input. As opposed to Basic, Forth will never interrupt screen output with the "Scroll?" message.

The Extended mode of the keyboard is not used. You type the characters that you have to enter normally in Extended Mode (like \ { and [) by typing the corresponding keys just with SYMBOL-SHIFT. You cannot enter Basic keywords in Forth, as you would do in Basic.

Special keys (outside the editor):
- With **DELETE** (CAPS-SHIFT-0) you **erase** the last character typed.
- With **ENTER** you indicate that the line you just typed is finished and then the typed commands will be executed.
- **CAPSLOCK** (CAPS-SHIFT-2) works normally. The system starts in caps-lock mode, because the interpreter is case-sensitive and all standard Forth words are upper case.
- With SYMBOL-SHIFT-W or with the **BYE** command you can return to Basic. SYMBOL-SHIFT-W works even if the Forth dictionary is corrupted and no Forth words can be found by the interpreter. From BASIC you may be able to save your RAM disk or reload the Forth system while preserving the RAM disk. Return to Forth with the RUN command.
- **BREAK** (CAPS-SHIFT-SPACE) interrupts just about any program and returns to the Forth command line.
- **EDIT** (CAPS-SHIFT-1) can be used to stop DUMP, VLIST and LIST, much more cleanly than with BREAK.

The word **>P** redirects all output to the printer until the next command line has to be entered. The word >S directs the output to the screen again.

If you use a ZX-Printer on the Spectrum 128, you have to type the word ZX-PRINT first. The ZX-Printer is not normally supported by 128K Basic, but I had this (in fact a Timex work-alike) as my printer, so I found out what it took to re-enable the ZX-Printer output on the Spectrum 128.

If you use a serial printer on Interface-1, you have to edit the Basic program:

- Leave Forth with SYMBOL-SHIFT-W.
- Edit Line 20. Just before RANDOMIZE, add the following commands:
  FORMAT "t";9600:OPEN #3,"t"
- Do a cold start of Forth with RUN 20.
- You can save the Forth system as described in section 1.3.

## 1.5 ERROR CONDITIONS AND THEIR EFFECTS

Whenever in chapter 3 an error or error message is mentioned, the following will happen:

- An error message is printed.
- The name of the forth word causing the error is printed.
- ABORT is executed (which causes the system to return to the command interpreter and to clear the stacks).
- If the error occurs during loading of a screen, the location of the error is pushed on the stack, so that you can type WHERE to
  start the editor right at the location in the source where the error occurred.

Not all error conditions are detected and reported though, No error is reported in the following cases :

- Division overflow or division by zero, A result of -1 is returned.
- Negative number where the standard requires a positive number. In this case the number is treated as an unsigned number.
- Wrong manipulations of the return stack, EXIT does not check whether the return address is valid and the system will likely crash if it is not valid.
- An invalid address supplied to EXECUTE.
- WORD does not find an end delimiter.

# 2. SCREENS AND FILES

## 2.1 THE RAM DISK

Forth source code is stored in blocks (also called screens) of 1kB each. As the Spectrum has to work with slow cassette tapes or with not very fast Microdrives, the FORTH screens are stored in a RAM disk. On the Spectrum 48 this RAM disk is located in memory from the address stored in the LO variable (normally 55296) to the address 65535. On the Spectrum 128 it is located in five extra RAM banks, which are normally not accessible. It consists of 5 groups of 16 screens each: 1-16, 17-32, 33-48, 49-64 and 65-80. #SCR is the total number of screens.

After loading FORTH, you can empty the RAM disk with the FORMAT command, which is advisable. The command

**N1 N2 INDEX**

shows the first line of every screen from number N1 through N2. Therefore it is wise to store a comment (describing the screen) in the first line of each screen.

## 2.2 LOADING AND SAVING SCREENS

With the command

**N DRIVE**

you can select the drive to use for saving and loading screens. N=0 means cassette tape, N=1..8 means Microdrive or floppy disk no. N. The CAT command shows the names of all available files. For Microdrives or disks it invokes the regular CAT command, for tapes it attempts to load a file named "$$$$$$$$$$", which likely does not exist, so it will show the names of all tape blocks that are encountered. The CAT command on tape has to be terminated with the BREAK key.

The command

**N GET filename**

loads the file with the specified name into the RAM disk starting at screen N.

The command

**N1 N2 PUT filename**

saves screens N1 through N2 with the chosen file name, as a "Bytes" block saved with the Basic command SAVE..CODE.

The command

```
DELETE filename
```

deletes the specified file. With Microdrives it is necessary to delete the old file first, before saving another file with the same name.

IMPORTANT: On the Spectrum 128, all screens loaded or saved in the same file have to be part of the same group of 16 screens:

```
15 17 PUT FOO
```

is illegal, as screen 15 and 16 are part of the first group and screen 17 is part of the second group.

The following commands are allowed.

```
15 16 PUT FOO1
17 17 PUT FOO2
```

To copy a file, proceed as follows:
1 Type the FORMAT command

2 Type the command

```
  1 GET filename
```

3 Type

**1 N INDEX**

where N is the highest screen number in the RAM disk (or 16 on the 128K Spectrum).

4 Check the highest screen number with a non-blank line in it.

5 Select another drive if needed, insert the correct cassette, Microdrive or floppy disk. If required, delete any old file with the same name.

6 Type

**1 M PUT**

where M is the highest non-empty screen, as found in step 4.

## 2.3 THE EDITOR

With N EDIT you start the screen editor. You will see half of screen N with a blinking cursor. At the bottom you seen the screen number, followed by the letter A if the top part is shown or the letter B if the bottom part is shown.

You can move the cursor with the cursor keys (CAPS-SHIFT-5 through CAPS-SHIFT-8 on the classic Spectrum keyboard). You always see one half of the screen, which is the part where the cursor is currently located.

As opposed to standard Forth systems, each screen contains 32 lines of 32 characters each (standard is 16 lines of 64 characters each).

The following keys have a special function in the editor:

**EDIT = CAPS-SHIFT-1 exits the editor**. You can undo the edits in the last screen with EMPTY-BUFFERS.

**CAPS LOCk = CAPS-SHIFT-2 CAPS LOCK as in BASIC**.

**TRUE VIDEO = CAPS-SHIFT-3 Inserts a blank line** at the location of the cursor. The lines below shift one position down, the last line is deleted.

**INV VIDEO = CAPS-SHIFT-4 deletes the line** at the location of the cursor. The lines below shift one position up, a blank line appears at the end.

**GRAPHICS = CAPS-SHIFT-9 Inserts a space** at the location of the cursor. The rest of the line shifts one position to the right. If the rightmost character at the end of the line is not a space or if the last word on the current line would run together with the first word on the next line, it will move to the next line. One of more subsequent lines may be shifted to the right as well.

**DELETE = CAPS-SHIFT-0 Deletes the character** at the location of the cursor. The rest of the line shifts one position to the left. If a word is divided between the end of the current line and the start of the next line, subsequent lines will shift to the left, avoiding these words to be split.

**SYMBOL-SHIFT-Q moves to the next half screen.** From 1A to 1B, from 1B to 2A.
**SYMBOL-SHIFT-E moves to the previous half screen.** From 2A to 1B, from 1B to 1A. Both of these keys can be used to page through the RAM disk.

**SYMBOL-SHIFT-W moves to the first position** at the first line of the current screen.

**ENTER moves to the first position** on the next line.

The editor can be in two modes: **file mode an blocks mode**.

The FILE command puts the editor in file mode. The entire RAM disk (or on the Spectrum 128K the entire group of 16 screens) is treated as one large text file. Insertions and deletions span the entire RAM disk or the entire group of 16 screens. Text will be moved across screens when lines are inserted or deleted. Cursor keys will also move across screens.

The BLOCKS command puts the editor in blocks mode. Each screen is treated individually. If a line is inserted, the last line of the current screen will disappear and will not be moved to the next screen. This is the normal way of editing FORTH source code.

## 2.4 PROGRAM LOADING

The command

**N LOAD**

will load Forth program source code from screen N. All text on a the loaded screen will be interpreted as if it was typed on the command line.

The word '****' (a backslash followed by a space) means a comment until the end of the current line.

The command **-->** can occur on a screen and it means that the next screen shall be loaded as well.

The command
**RUN filename**

is equivalent to

**1 GET filename**
**1 LOAD**

It will load a file into the RAM disk and immediately load (compile or execute) the source code contained in that file.

Screens can contain LOAD commands, so that other screens can be loaded from one screen.

## 3 IMPLEMENTED WORD SETS

### 3.1 STACK NOTATION

The words in the list are in ASCII order. The name of the word is followed by any values that the word expects on the stack, followed by three hyphens, followed by any values the word returns on the stack.

After that come the following optional indicators:
- I means that the word is immediate.

- 83 means the word is part of the FORTH-83 Required Word Set.
- S means that the word is part of the System Extension Word Set.
- C means that the word may only be used in the compilation state.
- D means that the word is part of the Double Number Extension Word Set, which is not fully implemented in the bare system (but the remaining words can be loaded as extensions).

The stack values (expected and returned) are denoted as follows:

The following stack values occupy one stack location each:

**f**: flag 0 is false, any other value means true.

**true**: -1, standard true value.

**false**: 0, standard false value.

**c**: ASCII character

**8b**: byte

**16b**: 16-bit word

**n**: signed number between -32768 and +32767.

**u**: unsigned number between 0 and +65535.

**w**: number between -32768 and +65535, signed or unsigned depends on context.

**addr**: memory address.

The following stack values occupy two stack locations each:

**32b**: 32-bit word

**d**: signed double number between -2147483648 and +2147483647.

**ud**: unsigned double number between 0 and 4294967295.

**wd**: number between -2147483648 and +4294967295, signed or unsigned depends on context.

## 3.2 DEFINITION OF TERMS:

block buffer: buffer of 1024 bytes that contains the screen that is currently accessed. This FORTH system contains a single block buffer. Screens are moved between the block buffer and the RAM disk.

**colon-definition:** FORTH word whose definition was started by ':'. When the colon definition is executed, the inner interpreter executes the words contained in the colon definition in succession. The address from which the colon definition was invoked gets stored on the return stack.

**compilation:** building a colon definition by adding compilation addresses and literals to the dictionary and by executing immediate words.

**compilation address:** the address of a FORTH word that will be added to the dictionary during compilation.

**counted string:** string of ASCII characters stored in memory, preceded by a single byte containing the length of the string.

**dictionary:** collection of vocabularies, which contains all FORTH words. The dictionary occupies a contiguous memory area, which is extended or reduced in size from the top end.

**immediate word:** FORTH word that is always executed, even if the text interpreter is in compilation state.

**inner interpreter:** piece of machine code that executes the words contained in a colon definition.

**input buffer:** buffer containing the line that is typed on the keyboard.

**input stream:** text read by the text interpreter, either form the input  buffer or from a block buffer.

**interpretation:** looking up words from the input stream in the dictionary and immediately executing them. When a word is not found in the dictionary, an attempt is made to convert it to a number and the resulting value is pushed on the stack if this succeeds.

**literal:** a special word within a colon definition that, when executed, pushes the value immediately following it on the stack.

**loop:** repetition structure used within a colon definition, using a terminal value (limit) and a counter (index) that are both stored on the return stack.

**numeric conversion:** conversion of a number from internal binary representation to a string of ASCII characters representing the number in human readable form.

**return stack:** LIFO (last in first out) stack containing return addresses of colon definitions, as well as other values, such as loop counters
and limits.

**runtime part:** word that is added to colon definition by an immediate word during compilation. When the colon definition is executed, the run time part will be executed.

**screen:** block of 1024 bytes usually containing FORTH source program text. The screens are stored in a RAM disk

**stack:** LIFO (last in first out) data stack central to the operation of FORTH, on which values are passed between words.

**text interpreter**: FORTH word that reads words from the input stream and interprets or compiles them, depending on the STATE variable.

**user variable**: Variable stored in a memory area whose start address can be changed. With multi tasking, each task has its own version of the user variables.

**vocabulary**: list of FORTH words.

## 3.3 WORDS IN FORTH VOCABULARY

**!** 16b addr ---      83
writes 16b in memory at address addr.

**!CSP** ---
stores the stack pointer in the CSP variable. Used internally by the compiler for checking that constructs like IF..THEN are complete.

**#** ud1 --- ud2     83
used between <# and #> (numeric conversion). Divides ud1 by BASE and obtains the rightmost digit as the remainder. The ASCII code of that digit is added to the numeric string using HOLD. The result is ud1 divided by BASE, which is the original number without the rightmost digit.

**#>** ud --- addr n    83
ends numeric conversion, returns address and length of the converted string.

**#B** --- n
constant, number of screens in the RAM disk  on the Spectrum 48.

**#S** ud1 --- ud2      83
Converts all remaining digits to ASCII using #, ud2 is 0.

**#SCR** --- n
returns number of screens in RAM disk.

**#TIB** --- addr      83
user variable containing the number of characters in the input buffer.

**'**    --- addr      83
reads word from input stream, looks it up in the dictionary and returns the compilation address, error message if the word is not found.

**'ERRNUM** --- addr
user variable containing the address of the word to be executed if NUMBER encounters an error. This way the word NUMBER can be extended so it can parse other data types,

for instance floating point numbers.

**(**    ---          I83
skips the input stream until ), is used for comment.

**(+LOOP)**   w ---
runtime part of +LOOP.

(.") ---
runtime part of  ."

**(;CODE)** ---
runtime part of DOES> Can only occur in a colon definition.
When executed, the colon definition is exited and the address following (;CODE) is put in the code field of the last created definition.

**(?DO)**   w1 w2 ---
runtime part of ?DO

**(ABORT")** f ---
runtime part of ABORT"

**(DO)**    w1 w2 ---
runtime part of DO

**(EMIT)**  --- addr
user variable that contains the address of the word to be executed by EMIT.

**(ERRNUM)** f ---
Causes an error of f is true. Normally 'ERRNUM points to this.

**(FIND)** addr1 addr2 --- addr3 n
This is the internal routine of FIND. addr2 is the name field address of the last word in the vocabulary. Otherwise identical to FIND.

**(FORGET)** addr ---
addr is the link field address of the word to be forgotten.
Removes this word and any words defined later from the dictionary.

**(KEY)** --- addr
user variable containing the address of the word to be executed by KEY.

**(LOOP)** ---
runtime part of LOOP

**(WAIT)** --- addr
variable containing the address of the word to be executed while waiting in KEY or PAUSE. A multitasker could be hooked up to this word.

**(WORD)** c addr1 --- addr2
Main word parsing routine of text interpreter.
addr1 is the address in the input stream where the search for a word must start. The first character unequal to c is the first character of the word, any consecutive characters all unequal to c are taken to be part of the word. addr2 is the address in the dictionary after the character c that terminates the word. A byte 0 marks the end of the input stream, addr2 will not increment beyond the end of the input stream.

The parsed word is stored as a counted string at the address in DP.
The counted string is followed by a space. The counted string will have length 0 if no word was parsed from the input stream.

***** w1 w2 --- w3     83
multiplies w1 by w2.

***/** n1 n2 n3 --- n4 83
multiplies n1 by n2 and divides by n3. The intermediate result has double precision.

***/MOD** n1 n2 n3 --- n4 n5 83
like */ but returns n4 as the remainder and n5 as the quotient of the division.

**+** w1 w2 --- w3     83
adds w1 an w2.

**+!** w addr ---      83
adds w to the contents of address addr and stores the result there.

**+-** n1 n2 --- n3
returns  n1 negated if n2 is negative, otherwise returns n1.

**+LOOP** addr 3 ---   IC83
 (runtime) w ---
terminates a DO LOOP. adds w to the index and terminates the loop if the addition crosses the boundary between limit-1 and limit. Jumps to the start of the loop if the

loop is not terminated.

**,** 16b --- 83
increases the size of the dictionary by 2 bytes and stores 16 at the end of the dictionary.

**-** w1 w2 --- w3 83
subtracts w2 from w1

**-->** --- I
may only occur in a screen, causes the text interpreter to continue on the next screen.

**-1** --- -1
constant -1

**-ROT** 16b1 16b2 16b3 --- 16b3 16b1 16b2
moves the top of stack to the third position.

**-TRAILING** addr n1 --- addr n2 83
if the string with address addr and length n1 contains spaces at the end, the length is decreased until there are no spaces at the end anymore.

**.** n1 --- 83
prints the number n1.

**."** --- IC83
(runtime) ---

reads the characters from the input stream until " and causes this text to be printed at runtime.

**.(** ---            I83
reads the input stream until ) and prints this text immediately.

**.R** n1 n2 ---
prints n1 right-justified to a length of at least n2 characters. Spaces are added to the left if necessary.

**.S** ---
prints the contents of the stack without changing it.

**/** n1 n2 --- n3    83
divides n1 by n2. n3 is always rounded down (floored division).

**/MOD** n1 n2 --- n3 n4 83
divides n1 by n2. n3 is the remainder with the same size as n2. n4 is the quotient, which is rounded down. This means rounded towards -infinity, not towards zero and it is called floored division.

**0**  --- 0
constant 0

**0<** n1  --- f         83
f=true if n1 is negative, false otherwise.

**0=** n1 --- f 83
f=true if n equals 0, false otherwise.

**0>** n1 --- f 83
f=true if n is positive, false otherwise.

**1** --- 1
constant 1

**1+** w1 --- w2 83
adds 1 to w1.

**1-** w1 --- w2 83
subtracts 1 from w1

**2** --- 2
constant 2

**2!** 32b addr --- 83
writes 32b at address addr.

**2*** w1 --- w2
multiplies w1 by 2

**2+** w1 --- w2 83
adds 2 to w1

**2-**  w1 --- w2       83
subtracts 2 from w1

**2/**  n1 --- n2       83
divides n1 by 2.

**2@**  addr --- 32b     83
reads 32b from memory address addr

**2DROP** 32b ---       D
removes 32b (the top two cells) from the stack

**2DUP**  32b ---  32b 32n D
duplicates 32b (the top two cells) on the stack,

**2OVER**  32b1 32b2 --- 32b1 32b2 32b1 D
duplicates the second 32-bit entry on the stack.

**2SWAP**  32b1 38b --- 32b2 32b1 D
swaps the two top most 32-bit entries on the stack,

**2ROT**  32b1 32b2 32b3 --- 32b2 32b3 32b1 D
moves the third 32-bit element on the stack to the top.

**3** --- 3
constant 3

**:** ---            I83

this word is not allowed during compilation. It performs the following actions:
- reads a word from the input stream.
- creates a new colon definition in the dictionary, which is not finished and which cannot be found yet.
- switches the text interpreter to compilation state.
- CONTEXT vocabulary is set to CURRENT.

**;** ---            I83
this word is only allowed during compilation. It performs the following actions:
- Finishes the colon definition currently being compiled in the dictionary by adding EXIT to it.
- Removes the 'smudge' bit from the header of the last defined word, so it can be found.
- switches the text interpreter to interpretation state.

< n1 n2 --- f     83
f=true if n1 is less than n2, false otherwise.

**<MARK** --- addr     S
Used by the compiler to compile BEGIN, returns the target address for a backward branch.

**<RESOLVE** addr ---   S
Used by the compiler to compile words like UNTIL. adds the backward branch address to the dictionary, previously returned by <MARK.

**<#** ---            83
Starts a numeric conversion operation by initializing the HLD variable.

**=** 16b1 16b2 --- f    83
f=true if n1 is equal to n2, false otherwise.

**>** n1 n2 --- f        83
f=true if n1 is greater than n2, false otherwise.

**><**  16b1 --- 16b2
swaps the bytes of 16b1

**>BODY** addr1 --- addr2 83
converts a compilation address to the word's parameter field address.

**>IN**  --- addr        83
user variable containing the position where the text interpreter will read the next word. It is an offset from the start of either the input buffer or the block buffer.

**>MARK** --- addr       S
used for compiling words like IF. It reserves space for a forward branch address and returns the address of that location.

**>NAME** addr1 --- addr2
converts a compilation address to the word's name field address.

**>P**    ---
causes all output to be directed to the printer until either >S is executed or a new command line is read.

**>R**    16b ---        83

pushes 16b on the return stack,

**>S** ---
causes all output to be redirected to the screen.

**>RESOLVE** addr --- S
used for compiling words like THEN. It fills the current dictionary address HERE as a forward branch address into the location previously returned by >MARK.

**?** addr ---
reads and prints the 16-bit number at address addr.

**?BRANCH** f --- S
can only be used inside a colon definition. It is followed by a branch target address. It branches to the target address if f=false.

**?COMP** ---
checks that the text interpreter is in compilation state, reports an error otherwise.

**?CSP** ---
reports an error if the contents of the CSP variable are not equal to the current stack pointer. Used by ; to check that there are no unfinished control structures.

**?DO** --- addr 3
(runtime) w1 w2 ---

marks the start of a loop. The index will be w2, the limit will be w1. If w1 is equal to w2, the loop will be skipped completely.

**?DUP** 16b --- 16b 16b or 16b 83
duplicates 16b on the stack, but only if it is not equal to zero.

**?EXEC** ---
checks that the text interpreter is in interpretation state, reports and error otherwise.

**?LOADING** ---
reports an error if the current input stream is in the block buffer.

**?PAIRS** 16b1 16b2 ---
reports an error if 16b1 is not equal to 16b2, used internally by the compiler to check proper nesting of control structures like IF..THEN.

**?STACK** ---
reports an error if too much data is on the stack or if the stack is empty and more has been popped from the stack than was pushed.

**?TERMINAL** --- f
f=true if the EDIT key (CAPS-SHIT-1) was pressed, false otherwise.

**@** addr --- 16b 83
reads 16b from memory at address addr.

**ABORT** --- 83
clears all stacks and restarts forth in a well-defined state by calling WARM.

**ABORT"** --- IC83
(runtime) f ---
reads the input stream until the " character and prints this text as an error message at runtime if f=true.

**ABS** n --- u 83
returns the absolute value of n

**ADDR** n1 --- addr
returns the address of screen n in the RAM disk. On the Spectrum 128 it switches to the correct memory bank.

**ALLOT** n --- 83
enlarges or the dictionary by n bytes of reduces it by n bytes if n is negative. reports an error if there is insufficient memory to enlarge it or if the base system would be lost by reducing the size of the dictionary.

**AND** 16b1 16b2 --- 16b3 83
returns the bitwise AND of 16b1 and 16b2.

**AT** u1 u2 ---
positions the cursor at line u1 and column u2. 0 0 AT is the topmost line and the leftmost column.

**B/BUF** --- 1024
constant, the number of bytes in a block buffer.

**B/SCR** --- 1
constant, the number of block buffers per screen.

**BANK** n ---
switches the memory bank at address C000H-0FFFFH on the Spectrum 128. n=0 for the normal RAM, 1-5 for the extra five memory banks containing the RAM disk.

**BASE** --- addr 83
user variable containing the numeric base, used by numeric conversion, printing of numbers or reading numbers from the input stream.

**BCAL** n ---
jumps to line n in the BASIC program, The BASIC statement RANDOMIZE USR 27036 returns to Forth and Forth, continuing just after the BCAL call.
If the BASIC program has used the bottom two lines on the screen, the BASIC program must execute PRINT; thereafter (this is required to set the output channel to the normal screen output).

**BEGIN** --- addr 1 IC83
starts a BEGIN..UNTIL loop or BEGIN..WHILE..REPEAT loop.

**BL** --- 32
constant, ASCII code for blank space

**BLK** --- addr 83
user variable that contains the screen number from which the input stream is read. This variable contains 0 if the input stream is read from the input buffer and not from a screen. The position in the input stream is fully defined by BLK and >IN

together.

**BLANK** addr u ---
fills the memory region starting at addr and u bytes in size with blank spaces.

**BLOCK** n --- addr 83
causes screen n to be loaded into the block buffer and returns the address of that block buffer. Any screen contained in the block buffer that has been changed, will be copied back to the RAM disk first.

**BLOCKS** --- 83
puts the editor in a state in which each screen is considered a separate text.

**BRANCH** --- S
like ?BRANCH, but jumps unconditionally. will be compiled by ELSE and REPEAT.

**BS** ---
prints a backspace.

**BUFFER** n --- addr 83
creates an empty block buffer for screen n (without reading it from the RAM disk) and returns its address.

**BYE** ---
returns to BASIC. This is always 48k BASIC, but on the 128k Spectrum the bank switch port will not be disabled.

**C!** 8b addr --- 83

writes 8b to memory at address addr.

**C,** 8b ---
extends the dictionary by 1 byte and puts 8b at that new location.

**C/L** --- u
returns the number of characters per line.

**C@** addr --- 8b 83
reads 8b from memory address addr.

**CAP** ---
switches CAPS LOCK on or off.

**CAT** ---
shows the names of all files on the disk, Microdrive cartridge or tape.
If used with tape, the command has to be terminated by pressing the BREAK key.

**CHAN** n ---
selects the output channel, 2 is screen, 3 is printer.

**CLEAR** n ---
fills screen n with blank spaces.

**CLS** ---
clears the video screen.

**CMOVE** addr1 addr2 u --- 83

copies memory area starting at addr1, size u bytes to address addr2. The byte at addr1 will be copied first.

**CMOVE>** addr1 addr2 u --- 83
Like CMOVE, but copying the bytes in reverse order, so the byte at addr1 will be copied last. If the source block and destination block do not overlap, CMOVE and CMOVE> will have an identical result.

**COLD** ---
cold start of FORTH. It performs the following actions, in addition to WARM.
- Removes all words from the dictionary at an address higher than FENCE>
- Checks for the presence of the extra RAM of the 128k and sets LO accordingly.
- initializes all relevant user variables.

**COMPILE** --- 83
may only occur inside an immediate colon definition. Compiles the word immediately following it in the colon definition by adding its address to the dictionary.

**CONSTANT** 16b --- 83
(runtime) --- 16b
reads a word from the input stream and creates a constant with that name and value 16b

**CONTEXT** --- addr S
user variable containing the address of the vocabulary that will be searched first.

**CONVERT** ud1 addr1 --- ud2 addr2 83
converts ASCII string to number. reads ASCII characters from address addr1+1.

If the character is a digit, ud1 will be multiplied by the value of BASE and the digit will be added to it. CONVERT will terminate on the first character that is not a digit. addr2 is the address of the first character that is not a digit. ud2 is the result of converting the ASCII digits to a binary number.

**COPY** n1 n2 ---
copies screen n1 to n2.

**COUNT** addr1 --- addr2 n 83
n is the byte at address addr1, addr2 is addr1+1. This is used to convert the address of a counted string to address and length.

**CR** ---        83
prints a carriage return and newline.

**CREATE** ---          83
(runtime) --- addr
reads a word from the input stream and creates a new word in the dictionary without allocating any space for data. This can be done with ALLOT. At runtime the word returns the end address of the dictionary at the time CREATE was called, this will be the start of the area added with ALLOT.

**CSP** --- addr
user variable containing the value of the stack pointer when the last colon definition was started. Will be checked when the colon definition is finished.

**CURRENT** --- addr      S
user variable containing the address of the vocabulary to which new words will be

added.

**D+** wd1 wd2 --- wd3   83
adds wd1 and wd2.

**D+-** d1 n --- d2
d2 is d1 negated if n is negative, otherwise it is d1.

**D-**  wd1 wd2 --- wd3  D
subtracts wd2 from wd1.

**D.**   d ---          D
prints the double number d.

**D.R**  d u ---        D
like .R, but with a double precision number.

**D0<**  d --- f        D
f=true if d is negative, false otherwise.

**D0=**  32b -- f       D
f=true if 32b equals 0, false otherwise.

**D<**   d1 d2 --- f    83
f=true if d1 is less than d2, false otherwise.

**DABS** d1 --- ud1     D
returns the absolute value of d1.

**DECIMAL** ---         83
sets the BASE variable to 10.

**DEFINITIONS** ---     83
sets the CURRENT vocabulary to the CONTEXT vocabulary.

**DELETE** ---
reads a word from the input stream and deletes the file with that name.

**DEPTH** --- n         83
returns the number of items on the stack.

**DIGIT** c --- u true  or  false
converts ASCII digit c to digit value u with a true flag.
return only a false flag if c does not contain a valid digit.

**DLITERAL** 32b ---    I
like LITERAL, but with a double number instead. It compiles LIT twice.

**DNEGATE** d1 --- d2   83
returns d1 negated.

**DO**    --- addr 3    83
(runtme) w1 w2 ---
starts a loop. w2 is the index, w1 is the limit. If w1 is equal to w2, the loop will be iterated 65536 times, as opposed to ?DO that will skip the loop (iterate 0 times) in this case.

```
DOES>   ---         IC83
(runtime) ---
(runtime of created word)
        --- addr
```
can only occur in a colon definition that contains CREATE.

- When compiling, the word will compile the word (;CODE) followed by a machine code CALL instruction to DOCOL (code field for colon definitions).
- At runtime, (;CODE) will change the code field of the word just created by CREATE, so it will call the part after DOES>.
- At runtime of the created word, the parameter field address will be pushed on the stack and the part of the colon definition following DOES> will be called.

**DP** --- addr       83

variable containing 1 more than the highest address of the dictionary.

**DPL** --- addr

user variable containing the location of the decimal point in the number last read by NUMBER. 0 means that the decimal point is at the right of the number, 1 means that 1 digit follows the decimal point etc.
Contains -1 if the number does not contain a decimal point at all.

**DRIVE** n ---

selects the mass storage medium to use. n=0 for tape, n=1..8 to select drive no. n.

**DROP** 16b ---     83

removes 16b from the stack.

**DUMP** addr u ---
shows the contents of the memory area starting at addr and u bytes in size.
The contents will be shown both in hexadecimal and in ASCII form.

**DUP**   16b --- 16b 16b  83
duplicates 16b on the stack.

**EDIT**  n ---
starts the editor on screen n.

**EDITOR** ---
vocabulary containing words that are used only by the editor,

**ELSE** addr1 2 --- addr2 2  IC83
used in IF..ELSE..THEN structure. Words between ELSE and THEN will be executed if and only if the flag checked by IF is false.

**EMIT**  c ---        83
prints the ASCII character c, using the word contained in (EMIT).

**EMPTY-BUFFERS**
marks the block buffer as empty. FLUSH will not copy it to the RAM disk.

**ERASE** addr u ---
fills memory area starting at addr and with a size of u bytes with byte 0.

**EXECUTE** addr ---   83
executes the word with compilation address addr.

**EXIT** ---          83
exits the colon definition containing this word. In interpretation, it exits interpretation of the screen containing the word.

**EXPECT** addr u ---  83
reads u characters from the keyboard and stores them in memory starting at addr, Typed characters are echoed to the screen. Special keys have the same functions as on the command line. Exits when either ENTER is typed or when u characters are typed.

**FENCE** --- addr
variable containing the address below which the dictionary will be preserved. COLD will delete everything above this address. This must contain a link field address.

**FILE** ---
puts the editor in a state in which the entire RAM disk (or a group of 16 screens on the Spectrum 128) is considered one large text file.

**FILL** addr u 8b ---      83
Fill memory area starting at addr, u bytes in size with byte 8b.

**FIND** addr1 --- addr2 n  83
addr1 is the start address of a counted string containing the word to be found in the dictionary. FIND searches the CONTEXT vocabulary first, the CURRENT vocabulary next. addr2 is the compilation address of the word found or it will be equal to addr1 if the word was not found.

n is as follows:

0 if the word was not found
-1 if the word is found and not immediate.
1 if the word is found and immediate.

**FIRST** --- addr     83
returns the first address of the block buffer.

**FLUSH** ---          83
saves the contents of the block buffer to the RAM disk if it is not empty.
Next it marks the block buffer as empty.

**FORGET** ---         83
reads a word from the input stream and removes the word with that name from the dictionary, plus all words defined later. Reports an error if the word is not found or if a word below FENCE would be deleted.

**FORMAT** ---
fills the entire RAM disk with blank spaces.

**FORTH** ---         I83
vocabulary that contains all words from this list. All other vocabularies will be linked to the FORTH vocabulary.

**FORTH-83** ---      83
shows that this system is FORTH-83.

**GET** n ---

reads a word from the input stream and reads the file with that name from mass storage into the RAM disk at screen n and any following screens.

**GETFN** u1 u2 ---
reads a word from the input stream and stores that in the BASIC variable A$.
n1 is stored in BASIC variable I and n1 is stored in BASIC variable J. Is used by the mass storage commands like GET and PUT before calling a BASIC routing with BCAL.

**H.** u ---
prints u as four hexadecimal digit.

**HERE** --- addr 83
returns the first address after the end of the dictionary.

**HEX** ---
sets BASE to 16.

**HLD** --- addr
user variable containing the address, just below which the next digit will be stored during numeric conversion.

**HOLD** c --- 83
adds ASCII character c to the string created during numeric conversion.

**I** --- w 83
returns the index of the innermost loop.

**I'** --- w

returns the limit of the innermost loop.

**ID.** addr ---
addr is a name field address of a FORTH word. Prints the name of that word,

**IF** --- addr 2 IC83
(runtime) f ---
used in IF..THEN and IF..ELSE..THEN. The words between IF and ELSE or between IF and THEN are only executed if flag=true.

**IMMEDIATE** --- 83
turns the word that was defined latest into an immediate word.

**INDEX** n1 n2 ---
shows the first lines of the screens n1 through n2

**INKEY** --- c
c=0 if no key is pressed, else c is the ASCII code of the key being pressed.

**INTERPRET** ---
The text interpreted. It reads words from the input stream until then end and interprets or compiles. Numbers containing a . will be considered 32-bit numbers.

**J** --- w
returns the index of the loop immediately outside the innermost loop.

**KEY** --- c 83
reads a character from the keyboard and returns the ASCII value as c. Uses (KEY).

**LATEST** --- addr
returns the address of the word that was defined latest.

**LEAVE** ---          83
leaves the innermost loop, removes values associated with the loop (index, limit) from the return stack.

**LIMIT** --- addr       83
the address just above the block buffer.

**LIST** n ---
prints the contents of screen n.

**LIT** --- 16b
can only occur inside a colon definition and is followed by a number.
pushes the number onto the stack. Will be compiled by LITERAL.

**LITERAL** 16b --- nothing or 16b I83
of the interpreter is in compilation state, 16b will be compiled as a literal, when it is interpreting, nothing will happen.

**LLIST** n1 n2 ---
prints the contents of screens n1 through n2 on the printer.

**LO** --- addr
variable containing the bottom address of the RAM disk. Contains 0 on the Spectrum 128.

**LOAD** n ---          83
submits screen n to the text interpreter as input stream. After the screen is loaded, it returns to the point in the input stream just after LOAD.

**LOOP** addr 3 ---    IC83
marks the end of a loop. The index will be incremented by 1 and the loop will terminate if it becomes equal to the limit, otherwise the next iteration of the loop will be started.

**M*** n1 n2 --- d
multiplies n1 and n2, giving a 32-bit product.

**M/** d n1 --- n2 n3
divides d by n1 giving n2 as a remainder and n3 as a quotient, using the same floored division rule as /MOD.

**M/MOD** ud1 u1 --- u2 ud2
divides ud1 by u1 giving u2 as a remainder and ud2 as a quotient.

**MAX** n1 n2 --- n3    83
n3 is the maximum of n1 and n2.

**MIN** n1 n2 --- n3     83
n3 is the minimum of n1 and n2.

**MIR** 16b1 16b2 16b3 --- 16b3 16b2  16b1
exchanges the top of stack with the number at the third position.

**MOD** n1 n2 --- n3     83
computes the remainder of the division of n1 by n2.

**MTYPE** addr u ---
fast version of TYPE that does not use (EMIT) for each character. Is primarily used by the editor.

**NAME>**  addr1 --- addr2
converts a name field address to the word's compilation address.

**NEGATE** n1 --- n2     83
computes n2 negated (n2 = -n1).

**NOOP** ---
no operation (do nothing).

**NOT** 16b1 --- 16b2     83
returns the ones' complement (all bits inverted).

**NUMBER** addr --- wd
converts a string starting at address addr+1 and ending in a blank space to a double precision number. The DPL variable will contain the position of any decimal point in the number (counted from the rightmost position).
DPL contains -1 if the number does not contain a decimal point.
As a special case, an & character followed by a single character is converted to the ASCII code of that character.
Executes the word contained in the 'ERRNUM variable if the string does not represent a

number, this will normally report an error message.

**OR** 16b1 16b2 --- 16b3 83
returns the bitwise OR of 16b1 and 16b2.

**OUT** --- addr
user variable containing the position on the line where the next character will be put by EMIT. Will be set to 0 by CR and be incremented by EMIT.

**OVER** 16b1 16b2 --- 16b1 16b2 16b1 83
duplicates the second value on the stack.

**P!** 8b addr ---
writes 8b to the output port at address addr.

**P@** addr --- 8b
reads 8b from the input port at address addr.

**PAD** --- addr 83
returns the address of a 64 byte buffer for string processing. This is at a certain offset above the dictionary.

**PAUSE** u ---
waits until u timer interrupts have occurred. The Spectrum is hardwired to generate 50 timer interrupts per second. While waiting, execute the word whose address is contained in the (WAIT) variable.

**PICK** u --- 16b 83

reads a value from the stack at position u, counted from the TOP. So:
0 PICK is equivalent to DUP
1 PICK is equivalent to OVER
The value is copied from the original stack position and not removed.

**PKEY** --- c
waits until a key is pressed and returns the ASCII code.

**PUT** n1 n2 ---
reads a word from the input stream and writes the screens n1 through n2 to the mass storage medium in a file with that name.

**QUERY** ---
reads a maximum of 128 characters from the keyboard into the input buffer using EXPECT and clears BLK and >IN to zero. Make sure to undo any input and output redirections before reading the line. This is word used by FORTH to read command lines.

**QUIT** --- 83
clears the return stack and reads command lines from the keyboard and interprets them.
**R>** --- 16b 83
pops 16b from the return stack.

**R0** --- addr
user variable containing the highest address (bottom address) of the return stack.

**R@** --- 16b 83
reads 16b from the top of the return stack without removing it.

**REPEAT** addr1 addr2 4 --- 83
marks the end of a BEGIN..WHILE..REPEAT loop. Returns to the BEGIN position.

**ROLL** u --- 83
moves the value at position u in the stack to the top (somewhat like PICK), but it removes the value from the original location.
1 ROLL is equivalent to SWAP.
2 ROLL is equivalent to ROT.

**ROT** 16b1 16b2 16b3 --- 16b2 16b3 16b1 83
moves the third value on the stack to the top.

**RP!** ---
initializes the return stack pointer.

**RP@** --- addr
returns the value of the return stack pointer.

**RUN** ---
reads a word from the input stream and loads a file with that name into the RAM disk starting from screen 1. Next loads the first screen.

**S->D** n --- d
converts a single precision number into a double precision number, sign extension.

**S0** --- addr
user variable that contains the bottom of the stack.

**SAVE-BUFFERS** ---     83
copies the contents of the block buffer to the RAM disk if it contains a valid screen.

**SCR** --- addr
user variable containing the number of the screen last displayed with LIST.

**SIGN** n ---          83
used in numeric conversion. Adds a - to the numeric conversion string if n is negative.

**SMUDGE** ---
Changes the 'smudge' bit in the header of the word that is defined latest.
If this bit is set the word cannot be found with FIND, if it is clear it can be found.

**SP!** ---
initializes the stack pointer.

**SP@** --- addr
returns the value of the stack pointer.

**SPACE** ---          83
prints a single blank space.

**SPACES** u ---        83
prints u black spaces.

**SPAN** --- addr     83
user variable containing the number of characters read by the latest

call to EXPECT.

**STATE** --- addr      83
user variable containing the state of the text interpreter, 0 is interpretation, otherwise it is compilation.

**STOPOFF** ---
Disables the BREAK key.

**STOPON** ---
Enables the BREAK key, so it will interrupt any running program.

**STYPE** addr u ---
Prints a string starting at addr and u characters long, thereby clearing the highest bit of each character and printing control characters as dots.
Used by DUMP.

**SWAP** 16b1 16b2 --- 16b2 16b1 83
swaps the top two values on the stack.

**TCH**  c ---
prints the character with ASCII code c on the screen or printer.

**TERMINAL** ---
sets the relevant variables, so subsequent input will be read from the keyboard and output will be printed on the screen.

**THEN** addr 2 ---    IC83

marks the end of an IF..THEN or IF..ELSE..THEN structure.

**TIB** --- addr        83
constant, the start address of the input buffer.

**TOGGLE** addr 8b ---
changes the byte stored at address addr, by exclusive-ORing it with 8b.

**TRAVERSE** addr1 n --- addr2
adds n to addr until bit 7 of the byte at the address is set. So with "addr 1 TRAVERSE" you scan memory at increasing addresses to find a byte with bit 7 set, with "addr -1 TRAVERSE" you can memory at decreasing addresses.
Is used by >NAME and NAME> to find the opposite end of the name field.

**TYPE** addr u         83
prints u characters, starting at address addr, using (EMIT) for each character.

**U.** u ---         83
prints the unsigned number u.

**U<** u1 u2 ---f    83
f=true if u1 is less than u2 (unsigned numbers), false otherwise.

**UM*** u1 u2 --- ud  83
multiply unsigned numbers u1 and u2, giving a double precision result.

**UM/MOD** ud u1 --- u2 u3 83
divides unsigned number ud by u1, giving u2 as a remainder and u3 as a quotient.

**UNDER** 16b1 16b2 --- 16b2
removes the second value from the stack.

**UNTIL** addr 1 --- IC83
(runtime) f ---
mars the end of a BEGIN..UNTIL loop. If f=false, return to the start of the loop, otherwise terminate.

**UPDATE** --- 83
indicates that the contents of the block buffer have been modified and hence must be written back with SAVE-BUFFERS. As this system has a fast RAM disk, buffers will always be written back.

**USER** n ---
(runtime) --- addr
reads a word from the input stream and creates a user variable with that name, stored at offset n in the user area. At runtime the address is returned.

**VARIABLE** --- 83
(runtime) --- addr
reads a word from the input stream and creates a variable with that name with storage for a single 16-bit value. At runtime the address is returned.

**VLIST** ---
shows all words in the CONTEXT vocabulary, preceded by their name field addresses.

**VOC-LINK** --- addr

user variable containing a pointer to a linked list of all vocabularies.

**VOCABULARY** ---      83
reads a word from the input stream and creates a vocabulary with that name, which is linked to the CONTEXT vocabulary, so the words from the linked vocabulary can also be found when this vocabulary is searched. Each vocabulary will ultimately be linked to the FORTH vocabulary. At runtime set the CONTEXT vocabulary to the selected vocabulary.

**WARM** ---
Warm start of Forth. It performs the following actions:
- Reset CONTEXT and CURRENT to the FORTH vocabulary.
- Reset some user variables.
- Execute the command loop via QUIT.

**WHERE** n1 n2 ---
n1 is the screen number, n2 the value of >IN at the time an error was detected while a screen was loaded. These are put on the stack by ABORT. Start the editor at the given screen with the cursor at the given location.

**WHILE** addr1 1 --- addr2 4 IC83
(runtime) f ---
occurs inside a BEGIN..WHILE..REPEAT loop and marks the decision point to exit the loop. If f=false, jump to the point beyond REPEAT to terminate the loop.

**WIDTH** --- addr
user variable containing the maximum number of characters that will be stored of the name of a newly created word. Normally this is 31, but it could be reduced to save memory or speed up compilation. The original length of the name will be stored and

FIND will only match words with the same length.

**WORD** c ---addr 83
reads a words from the input stream, starting at the first character that is not equal to c and ending at a character that is equal to c. It uses (WORD).
The word will be stored at the end of the dictionary (at the HERE address) as a counted string. This address is returned.

**XOR** 16b1 16b2 --- 16b3 83
bitwise exclusive-OR of 16b1 and 16b2.

**ZX-PRINT** ---
Enables the ZX-Printer on the Spectrum 128.

**[** --- I83
switches the text interpreter to the interpretation state.

**[']** --- IC83
reads a word from the input stream, looks it up in the dictionary and compiles the compilation address as a literal.

**[COMPILE]** --- IC83
reads a word from the input stream, looks it up in the dictionary and compiles it, even if the word in immediate.

**]** --- 83
switches the text interpreter to the compilation mode.

\ ---
may only be used on a screen. Skips to the end of the current line.
Used as a comment.

## 3.4 THE EDITOR VOCABULARY

Words from this vocabulary are used internally by the editor and are not normally used by other programs, so the descriptions are very short. The editor works on data in the block buffer when it is in BLOCKS mode and directly on the RAM disk in FILE mode.

**CUR** ---
Shows or hides the blinking cursor on the screen.

**DEL** ---
erases the character at the cursor position,

**DN** ---
moves the cursor one line down.

**HO** --- addr
variable containing horizontal cursor position.

**HOME** ---
moves cursor to the start of the current screen.

**INS** ---
inserts a space character at the cursor position.

**LDEL** ---
deletes the current line.

**LE** ---
move the cursor one position to the left..

**LIM** --- addr
One more than the last address of the text. It is the address beyond the block buffer in BLOCKS mode, it is 0 in FILE mode, as the RAM disk always extends to the end of RAM

**LINS** ---
inserts a blank line.

**LPOS** --- addr
start address of the current line.

**LREST** --- u
number of characters to be displayed on the screen from the current line..

**LST** ---
prints the current half screen on the video screen with a line below it showing the screen number and A or B.

**PD** ---
moves half a screen down.

**POS** --- addr
returns the address in the text.

**PU** ---
moves half a screen up.

**REST** --- u
number of characters to be displayed on the screen after the cursor position.

**RI** ---
moves the cursor one position to the right.

**SADR** --- addr
returns the start address of the current screen or block buffer.

**SET** ---
Set print position on video screen to cursor location in text.

**TXT** --- addr
variable containing the editor mode (BLOCKS or FILE).

**UP** ---
Moves the cursor one line up.

**VE** --- addr
variable containing the vertical position of the cursor from the start of the current screen (range 0..31)
.

# 4 COMPILER INTERNALS

## 4.1 THE INNER INTERPRETER.

The inner interpreter works with direct threaded code, this means: the addresses contained in a colon definition are directly jumped to. Therefore the code field of each FORTH word contains machine instructions. For code definitions these are the machine instructions of the code definition. For constants, variables, colon definitions and user variables, the code field contains a subroutine call to the appropriate runtime routine (DOCON for constants, NEXT for variables, DOCOL for colon definitions and DOUSER for user variables).

For CREATE DOES> words, the code field contains a subroutine call to the address directly following (;CODE) in the defining word. At this address a subroutine call to DOCOL is compiled. The first call puts the parameter field on the stack, the second call starts execution of the DOES> part of the defining word as a colon definition.

In all cases, the CALL instruction causes the parameter field to be pushed on the stack, so it is readily available to the handling routine (DOCON, DOCOL, DOUSER), or it just is there in case of variables.

Each code definition ends with a jump to the NEXT routine. This is a JP (IX) instruction, whereby the address of the next routine is always stored in the IX register. The NEXT routine is the inner interpreter itself. It reads the address of the next word to execute (pointed to by the instruction pointer, which is incremented) and jumps to it.

The **NEXT** routine consists of 7 instructions, as follows:

```
NEXT:
    EX DE,HL     ; Store the instruction pointer in HL
    LD E,(HL)    ; Read next address, lower byte
    INC HL
    LD D,(HL)    ; Read next address, higher byte
    INC HL
    EX DE,HL     ; Put incremented instruction pointer in DE, jump address in HL
    JP (HL)      ; Jump to the execution address.
```

- The instruction pointer is DE.
- The FORTH data stack pointer is SP.
- The address of NEXT is stored in IX.
- The return stack pointer is stored at address 69A7H,
- The user area pointer is stored at address 6994H.
- The W register is not used in this implementation. This is normally used to access the parameter field of the word being executed, but this is pushed on the stack by the CALL instruction in the code field.

Like in most systems, both the data stack and the return stack grow downward. Whenever we refer to the top of stack, this means the lowest memory address and the bottom of stack is the highest memory access.

The FORTH system runs a custom interrupt handler in Z-80 Interrupt Mode 2, which monitors the BREAK key and aborts execution if it is pressed.

## 4.2 HEADER STRUCTURE

The words in this Forth system consist of four fields, they are:

- **Link Field:** points to the name field of the previous word in the vocabulary. In the very first word, this field contains 0. Size is always 2 bytes.

- **Name Field**:
  First byte:
    bit 7: always 1
    bit 6: set if the word is immediate
    bit 5: smudge bit, set if the word shall not be found.
    bit 4..0: length of the name.

  The following bytes contain the characters of the name, whereby the final character has bit 7 set to 1. There may be fewer bytes than indicated by the length. See WIDTH variable for explanation.
  The end of the name is marked by setting bit 7.

- **Code Field**, call instruction of 3 bytes.

- **Parameter field**, contains all additional information used by the word, e.g. The value stored in a variable, the list of execution addresses in a colon definition.

Note: code definitions do not have a separate code field and parameter field. Machine instructions start at the code field address and the code definition contains as many bytes of machine code as required.

FIND and ' always return the code field address and this is the same as the compilation address.

>NAME converts the code field address to the name field address.
>BODY converts the code field address to the parameter field address.

NAME> converts the name field address to the code field address.

## 4.3 MEMORY MAP

All addresses are in decimal.

```
25580: highest address (bottom) of the data stack. The data stack has around 1kB
       free space to grow downward. The data stack lives in the memory area used by
       BASIC. The top part of this area is used by BASIC itself to store the stack.
       There are 20 bytes above the stack to allow for stack underflows without
       corrupting other memory areas.

25599: Top of RAM address for BASIC. Any addresses above it are not used
       by BASIC.

25600: input buffer, 128 bytes.

26000: highest address (bottom) of return stack. Grows toward input buffer, 272 bytes
       of space.
26000-27027: block buffer and its markings (which screen is contained in it).
27028: user area pointer.
27030: cold start entry
27033: warm start entry
27036: BCAL return.
27039: start of user area, 60 bytes in size. Locations 48..59 are still unallocated.
27047: return stack pointer, contained in user area.
27099: inner interpreter (=NEXT, value stored in IX instruction). The dictionary
       starts after this.
```

33792: is address 8400H. 257 bytes filled with 85H, interrupt vector table.
The Z-80 expects to read a an interrupt number from the bus in Interrupt Mode 2, but the ZX Spectrum does not contain hardware to provide it. Hence the Z-80 reads an undefined byte from the bus (usually 0FFH, but there is no guarantee). By filling these 257 bytes with 85H, we make sure that the CPU will jump to 8585H regardless of the byte read.

The Z-80 I register is set to 84H. Due to hardware restrictions, the interrupt vector table must be above 8000H.

34181: is address 8585H, interrupt routine. dictionary continues after this.

**HERE** = 36989: The dictionary will be extended starting here. The word read by WORD will be stored here too.
**PAD** = HERE+56: The numeric conversion buffer extends down from this address. The address at PAD and above (at least 64 bytes) can be used as a scratchpad.
**LO @** = 55296 on 48K Spectrum: start of RAM disk.

On the Spectrum 128 LO contains 0 and all addresses till 65535 can be used by FORTH. The RAM disk is located at addresses 49152 and 65535 in five additional banks. Words in the dictionary above 49152 are not allowed to access the RAM disk directly. The editor and some words in the base system do directly access the RAM disk, but they are all located below 49152.

## 4.4 BASIC PROGRAM

All tape and disk operations are carried out via BASIC. The variable I contains the start of the block, J the length of the block in bytes and D the drive number (0 for tape). A$ contains the file name.

The BASIC program creates these variables in a particular order (including DIM A$(10) to create A$ of size exactly 10) and it is important that this order is not changed. The FORTH system writes to fixed offsets into the BASIC variable area. See section 3.3 for details on GETFN (to store values into the BASIC variables) and BCAL (to invoke a BASIC routine).

Lines in the BASIC program:

line 1: warm start  
line 10: start after loading. The BASIC program will be saved such that it automatically starts at line 10 after loading. Reserves memory and loads FORTH83.BIN as data block. Next continues into cold start.  
line 20: cold start.  
line 40: DELETE routine.  
line 45: CAT routine.  
line 50: PUT routine.  
line 55: GET routine.  
line 60: saving the FORTH system.

The BASIC program temporarily stores the value of D (drive number) at address 23728 and reads it back after CLEAR, so this values will be preserved.

Note that variables are also preserved across saving and loading, so D will contain the drive number when the program is started at line 10.

## 4.5 META COMPILING.

The files META1 through META4 contain the meta compiler with which the FORTH system can be compiled from source. Proceed as follows:

- When using Microdrives or disks, **DELETE FORT83.BIN**
- **RUN META1**
- **RUN META2**
- **RUN META3**
- **RUN META4**
  This will save FORT83.BIN on tape or disk. When using tape, save it on a temporary tape.
- **BYE**
- **From BASIC, GO TO 10**. This will load and run the newly generated FORT83.BIN file. When using tape, you have to rewind the tape to get the file just saved.
- **RUN EDITOR**. When using tape, you will have to switch tapes to get to the tape that contains that file.
- When using Microdrives or disks:
  **DELETE run**
  **DELETE FORT83.BIN**
- **60 BCAL**
  This will save the new FORTH system, complete with BASIC part and editor. When using tape, you may have to switch to the correct tape of course.

Note: the BASIC part will not be regenerated by the meta compiler. It is already source and it can be edited.

The target system is the end product of the meta compiler. It contains all machine code and the entire dictionary of the Forth system being built. It is however located at a different address from which it will later run. The target system resides in a special memory area, outside the dictionary of the running Forth system.

Code in the target system will never be run during meta compilation. It is possible that a target system contains code that is to be run on an entirely different computer, possibly having a different CPU architecture. This meta compiler however, is intended for generating a new Forth system for the system on which it runs.

The file META1 contains the meta compiler itself. It consists of:
- words to access the memory area where the target system will be built. These are words like !-T @-T and ,-T that are equivalent to ! @ and ,
- The META vocabulary. Most words have identical names to words in the FORTH vocabulary, but they compile to the target system.
- The TARGET vocabulary. This contains the immediate words of the target system.
- CREATE-T. It creates headers in the target system. Further it creates definitions in the META vocabulary
- The meta assembler. It generates machine code in the target system.
- Words that perform the actions of immediate words in FORTH, but that act on the target system. These are located in META.

## Examples of vocabulary usage, the word IF.

- The FORTH vocabulary contains the word IF used by FORTH. It is used when FORTH compiles normal words in its own dictionary.
- The META vocabulary contains the word IF used by FORTH when meta compiling. It is used when the meta compiler compiles colon definitions into the target system.
- The TARGET vocabulary contains the word IF that will be used by the target system when it will itself run.

The files META2, META3 and META4 use the meta compiler to construct a target system. The file EDITOR will be used by the newly built system, to build its own editor. Using the meta compiler you can make changes to the FORTH system at the source level and any part of it can be modified. It is also possible to create special versions of Forth that do not contain headers or compilation words, that contains special application. Projects like this must only be carried out by experienced FORTH programmers.

The file meta_annotated.txt contains an annotated meta source listing that documents the entire meta compiler and the FORTH system generated with it.

# 5 THE ASSEMBLER

## 5.1 INTRODUCTION

Using the assembler you can create words in machine code. This manual assumes that you are familiar with the Z80 and its standard mnemonics. There are two reasons why this assembler uses mnemonics that are different from the standard:

- The standard assembler uses one and the same word as a mnemonic for instructions that must be translated differently into machine code.

  For instance LD is used for

  ```
  LD A,B
  LD (IX+12),23
  LD HL,(RPTR)
  LD A,I
  LD SP,HL
  ```

and all these use different opcodes.

- Forth is suitable for expressions in postfix notation. Therefore it is straightforward to write an assembler that expects opcode mnemonics after the operands and the use of expressions in operands is straightforward as well.

Labels are hardly ever used in a FORTH assembler. Instead we make use of control structures like IF..THEN, as is done in FORTH as well.

This assembler was designed by Coos Haak in Utrecht, The Netherlands and he used it in his own FORTH. It was ported to FORTH83 by L.C. Benschop.

## 5.2 LOADING THE ASSEMBLER

The assembler consists of two files: TASM (size 1 screen) and ASSEMBLER (size 8 screens). If you desire to have the assembler as a permanent part of FORTH (for instance if you desire to save the system complete with the assembler), you type:

**RUN ASSEMBLER**

You can type **HERE FENCE !** to make it a permanent part of the system and then you can save the system as described in section 1.3.

If you use the assembler temporarily, for instance just to load other extensions like the Double Number Extension Word Set, you type:

**RUN TASM**

This way the assembler is loaded at a high memory address outside the dictionary. With the command DISPOSE you can remove the assembler from the system after use and any words defined later (including code definitions created with the assembler) will remain in the dictionary.

## 5.3 CREATING CODE WORDS

For the creation of code definitions, the following words are available.
If the word is followed by the letter A, the word is part of the Assembler Extension Word Set. If the word is followed by the letter F, it is located in the FORTH vocabulary, otherwise it is located in the ASSEMBLER vocabulary. Assembling state means: interpretation state with ASSEMBLER as the CONTEXT vocabulary and BASE set to 16.

**;C** ---          F
synonym for END-CODE

**;CODE** ---          FICA
Assembler version of DOES>. Used in a colon-definition containing CREATE. Compiles (;CODE) and puts FORTH in the assembling state. The colon definition will at runtime modify the code field of the defined word, so it calls the machine code assembled after ;CODE. The created word will in turn call the machine code assembled after ;CODE with the parameter field address on the stack.

**ASSEMBLER** ---     FA
Vocabulary containing all Assembler mnemonics, register definitions and the like.

**CODE** ---          FA
reads a word from the input stream and creates a code definition with that name. Puts

FORTH in the assembling state and sets the 'smudge' bit, so the word will not be found.

**END-CODE** ---        FA
ends a code definition. Clears the 'smudge' bit, so the word defined latest can be found. CONTEXT is set to CURRENT and BASE is set to 10. Use this word to terminate an assembler definition that was started with CODE, ;CODE or LABEL.

**ENDM** ---          IC
synonym for ; ends a macro.

**LABEL** ---         F
reads a word from the input stream and creates a word with that name, which will return its parameter field address at runtime. Next it puts FORTH in the assembling state. A word created with LABEL can be used as a jump or call address in the definition of other code words.

**MACRO** ---         F
Like : but makes ASSEMBLER the context vocabulary and sets BASE to 16. When the macro is executed, it will assemble (add to a new code definition) the instructions that are contained in it.

**XY** --- addr      F
variable indicating which of the two index registers will be used. Values are 0DDH for IX, 0FDH for IY, just like the opcode prefixes used by the Z80.

## 5.4 THE REGISTERS

The assembler uses the names A, B, C, D, E, H and L for the 8-bit Z80 registers. In addition it uses the name M, which can be used in most places where a register is allowed. M is the memory address pointed to by HL (this is like the 8080 assembler that uses M instead of (HL)).

Register pairs are named B, D and H to indicate the BC, DE and HL register pairs respectively. In addition SP an AF are register pair names. AF is only allowed with PUSH and POP. B, D, H and SP are allowed in many instructions using register pairs.

The order in two operand instructions is source followed by destination, which is the reverse of standard Z80 or 8080 notation.

Hence
```
B C LD
```
is equivalent to
```
LD C,B
```

Use of index registers IX and IY:
- Use the word X in front of instructions where use of the HL register is implicit, For instance X LDSP to indicate LD SP,IX
- Use XH or XL instead of H or L for instructions that use H or L explicitly.
- For instructions that actually do indexing (IX+12) instead of (HL), use special mnemonics starting with )

By default the chosen X register is IX. The IY register is tied up by the ROM routines and will hardly ever be used on the ZX-Spectrum. The word %Y changes the used index register to IY, %X changes it back to IX.

## 5.5 THE INSTRUCTION SET

We use the same stack notation as in chapter 3, but with the following additions to denote values (all values are a single stack entry):

**b**: bit 0-7
**cc**: condition code: z, cs, pe, m, v or their negations obtained with the word NOT
**r**: register A, B, C, D, E, H, L or M
**rp**: register pair B, D or H
**rps**: register pair or SP.
**rpa**: register pair or AF.
**disp**: displacement between -128 and 127 for use with indexed addressing.

**LD** r1 r2 ---
copies r1 to r2. Equivalent to LD r2, r1

**)LD** r1 disp ---
loads r1 from address IX+disp. Equivalent to LD r1,(IX+disp)

**)ST** r disp ---
writes r1 to address IX+disp. Equivalent to LD (IX+disp),r1

**LD#** 8b r ---
loads r with immediate value 8b. Equivalent to LD r1,8b

**)LD#** 8b disp ---
Loads immediate value 8b at address IX+disp. Equivalent to LD (IX+disp),8b

**MOV** rp1 rp2 ---
copies register pair rp1 to rp2. Assembles to two Z80 instructions. For example D B MOV is equivalent to
```
LD B,D
LD C,E
```

**LDP#** 16b rps ---
loads register pair rps with immediate value 16b. Equivalent to LD rps,16b

**LDP** addr rps ---
loads register pair rps from address addr. Equivalent to LD rps,(addr)

**)LDP** rps disp ---
loads register pair rps from address IX+disp, B 2 )LDP is equivalent to:
```
LD C,(IX+2)
LD B,(IX+3)
```

**STP** addr rps ---
writea register pair rps to address addr. Equivalent to LD (addr),rps

**)STP** rps disp ---
writes register pair rps to address IX+disp. B 2 )STP is equivalent to:
```
LD (IX+2),C
LD (IX+3),B
```

**LDHL** addr ---
like LDP, but for HL register. Uses shorter opcode form.

**STHL** addr ---
like STP, but for HL register. Uses shorter opcode form.

**LDA** addr ---
loads A from address addr. Equivalent to LD A,(addr)

**STA** addr ---
writes A to address addr. Equivalent to LD (addr),A

**LDAP** rp ---
rp=B or D only! Loads A from address in rp.
Equivalent to  LD A,(BC) or LD A,(DE)
But use M A LD  for LD A,(HL) !!!

**STAP** rp ---
rp=B or D only! Writes A to adres in rp.
Equivalent to LD (BC),A or LD (DE),A
But use A M LD   for LD (HL),A !!!

**LDSP** ---
loads SP with HL. LD SP,HL

**LDAI** ---
loads A with I. LD A,I

**LDIA** ---
loads I with A. LD I,a

**EXAF** ---
exchanges AF and AF'. EX AF,AF'

**EXDE** ---
exchanges DE and HL. EX DE,HL

**EXSP** ---
exchanges HL with top element on stack. EX (SP),HL

**CLR** rps ---
Loads rps with 0. LD rps,0

**ADD** r ---
adds r to accumulator A. Equivalent to ADD A,r
The instructions ADC, SUB, SBC, AND, XOR, OR and CP work the same way.

**)ADD** disp ---
adds contents of address IX+disp to accumulator A. Equivalent to ADD A,(IX+disp)
The instructions )ADC, )SUB, )SBC, )AND, )XOR, )OR and )CP work the same way.

**ADD#** 8b ---
adds constant 8b to accumulator A. Equivalent to ADD A,8b
The instructions ADC#, SUB#, SBC#, AND#, XOR#, OR# and CP# work the same way.

**ADDP** rps ---
adds rps to HL register. Equivalent to ADD HL,rps

The instructions ADCP, SUBP and SBCP work the same way.
Note that SUBP is a macro composed of A AND and rps SBCP

**INC** rps ---
increments register pair rps by 1. Equivalent to INC rps
The instruction DEC works the same way.

**INR** r ---
increments register r by 1. Equivalent to INC r
The instruction DER works the same way (equivalent to DEC r).

**)INR** disp ---
increments the byte at address IX+disp by 1. Equivalent to INC (IX+disp)
The instruction )DER works the same way (equivalent to DEC (IX+disp)

**RL** r ---
Rotates register r one position to the left. Equivalent to RL r
The instructions RR, RLC, RRC, SRL, SRA and SLA work the same way.

**)RL** disp ---
Rotates the byte at address IX+disp one position to the left.
Equivalent to RL (IX+disp).
The instructions )RR, )RLC, )RRC, )SRL, )SRA and )SLA work the same way.

**BIT** b r ---
test bit b of register r. Equivalent to BIT b,r
The instructions RES and SET work the same way.

**)BIT** b disp ---
test bit b at address IX+disp. Equivalent to BIT b,(IX+disp).
The instructions )RES and )SET work the same way.

**TST** rp ---
test if register pair rp is zero. B TST expands to the instructions.
LD A,B
OR A,C

**JP** addr ---
jumps to absolute address addr Equivalent to JP addr.
The instructions JPNZ, JPZ, JPNC, JPC, JPPO, JPPE, JPP, JPM, CALL, JR, JRNZ, JRZ, JRNC, JRC, DJNZ en RST work the same way. Relative branch instructions (JR etc) take absolute addresses as inputs, but are assembled with relative branch offsets.
Example: addr JPNZ is equivalent to JP NZ,addr

**JPHL** ---
jumps to the address in HL. Equivalent to JP (HL).
The instruction JPIX works the same way.

**CALLC** addr cc ---
conditional call to address addr. cc is one of the condition codes that can be specified in lowercase letters as described in 5.6
Equivalent to CALL cc,addr

**RETC** cc ---
conditional return. cc is specified the same way as in CALLC.
Equivalent to RET cc.

**PUSH** rpa ---
pushes rpa onto the stack. Equivalent to PUSH rpa
The POP instruction works the same way.

**PRT** ---
RST 10H to print a character.

**HOOK** 8b ---
RST 8 DEFB 8b to report a BASIC error message or to call an Interface 1 function.

**IN** 8b ---
reads accumulator A from port 8b. Equivalent to IN A,(8b)
The OUT instruction works the same way.

The following instructions have no operands and use the exact same mnemonics as standard Z80 assembler: NOP, RLCA, RRCA, RLA, RRA, HALT, RET, DAA, CPL, SCF, CCF, DI, HALT, EXX, LDIR, LDDR, CPIR, IM1, EI, IM2 and NEG.

Note that not all Z80 instructions are implemented by this assembler, but the missing instructions are unlikely to be ever used. They can be added to the assembler if required or their opcodes can be assembled directly with , or C,

RPTR, DPTR and NEXT are 3 address constants. They denote the address of the return stack pointer, the address of the user area pointer and the address of the inner interpreter.

## 5.6 CONTROL STRUCTURES

The assembler can use the IF..THEN, IF..ELSE..THEN, BEGIN..UNTIL and BEGIN..WHILE..REPEAT control structures from FORTH and they are implemented with relative jumps. IF, WHILE and UNTIL are preceded by a condition code, which can be Z, NZ, CS or NC.

```
BEGIN
..
NC UNTIL
```

denotes a loop that must repeat until the carry bit is clear. Hence the relative jump instruction at the end will be JRC.

Further we have BEGIN..AGAIN for an infinite loop and BEGIN..DSZ for a down counting loop with a DJNZ instruction at the end.

The same control structures, except BEGIN..DSZ can also be used with absolute jumps. To specify absolute jumps, specify the relevant words in lowercase. Permissible condition codes are z, cs, pe, m and v (where v is the same as pe). Each of these condition codes can be followed by NOT to indicate the opposite condition.

The same BEGIN..UNTIL construct specified with absolute jumps will become:

```
begin
..
cs NOT until
```

# 6 THE DOUBLE NUMBER EXTENSION WORD SET

## 6.1 INTRODUCTION

The file DOUBLE (3 screens in size) contains all words required to implement the full Double Number Extension Word Set and additionally all words necessary to perform all operations with double precision, including multiplication, division and square root.

Before loading DOUBLE, you must first load the assembler using either RUN TASM or RUN ASSEMBLER. Then load RUN DOUBLE and finally (if applicable) DISPOSE to remove the assembler.

## 6.2 EXTRA WORDS

**2CONSTANT** 32b ---   D
(runtime) --- 32b
like CONSTANT, but for a 32-bit value.

**2UNDER** 32b1 32b2 --- 32b2
removes the second value from the stack.

**2VARIABLE** ---       D
(runtime) --- addr
like VARIABLE, but with 4 bytes of space.

**D*** wd1 wd2 ---wd3
multiplies wd1 with wd2.

**D/** d1 d2 --- d3
divides d1 by d2, rounding down (floored division).

**D/MOD** d1 d2 --- d3 d4
divides d1 by d1. d3 is the remainder and d4 the quotient.

**D2*** wd1 --- wd2
Multiplies wd1 by 2.

**D2/** wd1 --- wd2    D
divides wd1 by 2.

**D=** 32b1 32b2 --- f  D
f=true if 32b1 and 32b2 are equal, false otherwise.

**D>** d1 d2 --- f      D
f=true if d1 is greater than d2, false otherwise.

**DIVISOR** --- addr
double number variable to store the divisor for D/

**DMAX** d1 d2 --- d3   D
d3 is the maximum of d1 and d2.

**DMIN** d1 d2 --- d3   D
d3 is the minimum of d1 and d2.

**DMOD** d1 d2 --- d3

d3 is the remainder of the division of d1 by d2.

**DU.** ud ---
like D. but for unsigned number.

**DU.R** ud u ---
like D.R but for unsigned number.

**DU<** ud1 ud2 --- f   D
f=true if unsigned ud1 is less than ud2, false otherwise.

**SQRT** ud --- u
u is the square root of ud, rounded down.

**U.R** u1 u2 ---
like .R but for unsigned number.

**UD/MOD** ud1 ud2 --- ud3 ud4
divides ud1 by ud2. ud3 is the remainder and ud4 is the quotient..

# 7 THE FLOATING POINT EXTENSION

## 7.1 INTRODUCTION

With this extension you can use floating point numbers in Forth. These numbers are 32 bits in size and occupy two entries on the data stack each, hence you can use 2@ and 2! to read and write floating point numbers in memory and all double precision stack words, such as 2DUP and 2SWAP, to manipulate them on the stack.

The word NUMBER is extended with a means to process floating point numbers. Now you can enter floating point numbers from the text interpreter by immediately following a number (that may or may not contain a decimal point) with an & sign. The & sign can optionally be followed by a + or - sign and then a small integer that represents the exponent. The exponent is the power of BASE with which the number must be multiplied. This way we can enter floating point numbers in scientific notation in any number base.

Examples

```
     2&    the number 2
-2.718&    the number -2.718
    12&+3  the number 12000
-0.041&    the number -0.041
  -4.1&-2  the number -0.041
```

The floating point words are contained in two files. The first file, FLOATING (8 screens in size) contains the basic operators, words to print floating point numbers, the extension of NUMBER to allow floating point input and a square root function. The assembler must be loaded first. This can be loaded with RUN TASM or RUN ASSEMBLER. Then the floating point extension can be loaded with RUN FLOATING. Finally the assembler can be removed with DISPOSE, if desired.

The second file TRANSCEN (size 4 screens) contains trigonometric and logarithmic functions. This file does not require the assembler, but it does require FLOATING to be loaded.

If the system is saved (using HERE FENCE ! 60 BCAL) and later reloaded or if it is restarted via a cold start, the word FLOAT must be executed to re-activate the extension to NUMBER that allows floating point input.

## 7.2 FORMAT AND PRECISION.

A floating point number consists of two parts. The three least significant bytes form the fractional part (also called mantissa), a 24-bit number between 1 and 2. The leading bit represents 1, the second bits represents 1/2, the third bit represents 1/4 and so on. As the leading bit is always 1, this is omitted from the number and this position is used for the sign of the number (0 for positive, 1 for negative). The most significant byte is the exponent offset by 128. This represents the power of two (in the range -128..127) with which the fractional part is multiplied.

The largest negative number (closest to zero) and the smallest positive number both represent the number zero. The largest positive number and the smallest negative number represent + or - infinity. These numbers are the result of arithmetic overflow or of illegal operations.

The floating point numbers have a precision of around 7 decimal digits.
The following number ranges can be represented:
- -3*10**38 .. -3*10**-39
- 0
- +3*10**-39 and 3*10**38

The floating point format is similar to, but not identical to the IEEE-754 single precision binary floating point format. It does not have subnormal numbers and no NaN as a value different from infinity. Also the exponent offset is different (127 for IEEE-754, 128 for the Forth format).

## 7.3 WORDS FROM FLOATING.

The stack notation is the same as in previous chapters, but fp is added to represent a 32-bit floating point number. Words only used internally (such as assembler labels) are not listed here.

**1&** --- fp
The constant 1.

**10&** --- fp
The constant 10.

**2CONSTANT** 32b ---
 (runtime) --- 32b
See chapter 6

**D->F** d --- fp
converts a double precision integer d to a floating point number with the same value (which cannot be represented exactly for large integers).

**F*** fp1 fp2 --- fp3
floating point multiplication

**F+** fp1 fp2 --- fp3
floating point addition,

**F-** fp1 fp2 --- fp3
subtracts fp2 from fp1

**F->D** fp --- d
converts the floating point number f to a double precision integer, rounding toward zero.

**F.** fp ---
prints fp in scientific notation using the & symbol as "ten to the power" symbol.

**F.R** fp n1 n2 ---
prints fp right-justified in a field of at least n1 characters wide, inserting spaces to the left if required. n2 digits are printed after the decimal point. Any number base can be used.

**F/** fp1 fp2 --- fp3
divides fp1 by fp2

**F0<** fp --- f
f true if fp is less than zero, otherwise false.

**F0=** fp --- f
f true if fp is equal to zero, otherwise false.

**F2*** fp1 --- fp2
multiplies fp1 by 2.

**F2/** fp1 --- fp2

divides fp1 by 2.

**F<** fp1 fp2 --- f
f is true if fp1 is less than fp2, false otherwise.

**F=** fp1 fp2 --- f
f is true if fp1 is equal to fp2, false otherwise.

**F>** fp1 fp2 --- f
f is true if fp1 is greater than fp2, false otherwise.

**FABS** fp1 --- fp2
fp2 is the absolute value of fp1.

**FERRNUM** f ---
The extension of NUMBER for floating point input. The 'ERRNUM variable points to this. Parses the number as a floating point number and converts to a floating point value if f is true. If conversion fails, an error is reported.

**FI**** fp1 n --- fp2
Raises fp1 to the power n. n is an integer.

**FLOAT** ---
Activates the floating point extension to NUMBER. Must be executed after a cold start.

**FNEGATE** fp1 --- fp2
subtracts fp1 from 0.

**FSQRT** fp1 --- fp2
computes the square root of fp1.

**NAN** --- fp
Constant, infinite value.

**X!** fp1 8b --- fp2
Replaces the exponent byte of fp1 with 8b.

**X@** fp --- fp 8b
8b is the exponent byte from fp.

## 7.4 WORDS FROM TRANSCEN.

**180/PI** --- fp
constant, 180 divided by pi.

**2,** 32b ---
appends 32b to the end of the dictionary, 32-bit version of ,

**2VARIABLE** ---
 (runtime) --- addr
See chapter 6.

**ATN2** --- fp
Computes the angle in radians relative to the X-axis, represented by the vector contained in the FX and FY variables.

**ATNTAB** --- addr
addr is the start address of a table containing the arc-tangents of negative powers of 2. Used internally by trigonometric computations.

**ATNTAB@** u --- fp
fp is the arc-tangent of 2 to the power of -u.

**DEG** fp1 --- fp2
converts angle fp1 in radians to angle fp2 in degrees.

**F**** fp1 fp2 --- fp3
raises fp1 to the power fp2. fp1 must be nonnegative. Raising to an integer power with FI** can have a negative first argument.

**F*2**** fp1 n  --- fp2
divides fp1 by 2 to the power of n.

**F10**** fp1 --- fp2
computes 10 to the power of fp1.

**FARCCOS** fp1 --- fp2
computes the arc-cosine of fp1 in radians.

**FARCSIN** fp1 --- fp2
computes the arc-sine of fp1 in radians.

**FARCTAN** fp1 --- fp2
computes the arc-tangent of fp1 in radians.

**FCOS** fp1 --- fp2
fp1 is in radians, computes the cosine.

**FEXP** fp1 --- fp2
raises e to the power of fp1.

**FLN** fp1 --- fp2
computes the natural logarithm of fp1.

**FLOG** fp1 --- fp2
computes the base 10 logarithm of fp1.

**FSIN** fp1 --- fp2
fp1 is in radians, computes the sine.

**FTAN** fp1 --- fp2
fp1 is in radians, computes the tangent.

**FX** --- addr
**FY** --- addr
Two floating point variables containing a vector used by trigonometric computations.

**LN10** --- fp
constant, the natural logarithm of 10.

**LN2** --- fp
constant, the natural logarithm of 2.

**LNTAB** --- addr
addr is the start address of a table containing the natural logarithms of 1+2**-i. Used by logarithmic computation.s

**LNTAB@** u --- fp
fp is the natural logarithm of 1+2**-u

**PI** --- fp
constant, the number pi.

**PI/180** --- fp
constant, pi divided by 180.

**RAD** fp1 --- fp2
converts angle fp1 in degrees to angled fp2 in radians. Required if you want to compute sine, cosine or tangent of angles expressed in degrees. Example:
To compute the sine of 45 degrees, type:

```
45& RAD FSIN F.
```

**TAN2** fp ---
fp is positive. Creates a vector in the variables FX and FY that has an angle of fp radians relative to the x-axis.

# 8 GRAPHICS AND SOUND

## 8.1 GRAPHICS ROUTINES.

The file GRAPHIC (2 blocks in size) contains routines to draw points and lines on the screen and further it contains commands to set colors. GRAPHIC depends on the assembler and therefore the assembler has to be loaded first, the same way as described in chapter 6 for DOUBLE.

**PLOT** n1 n2 ---
Plots a single point with x-coordinate n1 and y-coordinate n2. Coordinates are the same as in BASIC.

**DRAW** n1 n2 ---
Draws a line from the current plot position, n1 points to the right and n2 points upward. If n1 or n2 are negative, the directions are to the left and downward respectively.

**POINT** n1 n2 --- f
Reads the value 0 or 1 of the point on the screen with x-coordinate n1 and y-coordinate n2.

**ATTR** n1 n2 --- c
returns the value of the attribute byte at line n1 and column n2.

**HARDCOPY** ---
prints a copy of the screen to the ZX-Printer.

**SCREENLOAD** ---

Reads a file name from the input stream and reads the file with that name into the screen buffer.

**SCREENSAVE** ---
Reads a file name from the input stream and writes the contents of the screen buffer to a file with that name.

**INK** n ---
**PAPER** n ---
**BORDER** n ---
**BRIGHT** n ---
**FLASH** n ---
work the same way as the BASIC commands with the same name.

**INV** n ---
works the same way as the basic command INVERSE.

**GOVER** n ---
works the same way as the BASIC command OVER.

**NORMAL** ---
Resets the colors and related settings back to normal.

**COLOR** c ---
 (runtime) n ---
Defining word used to implement commands like IN and PAPER.

## 8.2 USER DEFINED GRAPHICS.

The file UED (4 blocks in size) can be loaded with RUN UED after GRAPHIC has been loaded. It contains an editor for User Defined Graphics.

Codes of User Defined Graphics can be entered in FORTH with ^ (up-arrow) directly followed by a letter. ^A represents the graphic A.
For example: type ^A EMIT to print the character GRAPHICS-A.

**CHADDR** n --- addr
returns the address of the character bitmap of the character with code n.

**UED** ---
Starts the UDG editor. The UDG Editor shows an 8x8 field in which you can edit a graphic character. Within the UDG editor the keys have the following functions:
- Cursor keys: move the graphic cursor in the 8x8 field defining the bit pattern.
- ENTER go to the start of the next line in the 8x8 field.
- @ Reads a character into the 8x8 field. By typing a letter, the corresponding UDG will be read. By typing an & followed by any other character, this character will be read.
- ! Writes the bit pattern in the 8x8 field into memory where the graphic representation of a character. Selecting the character can be done as described for the @ command. Normally only UDG's can be written back to memory as ASCII characters are stored in ROM, but the system variable CHARS (at 23606) can be modified, so these characters are stored in RAM as well.
- A do a bitwise AND on the bit pattern in the 8x8 field with any other selected character.

- O do a bitwise OR on the bit pattern in the 8x8 field with any other selected character.
- E do a bitwise Exclusive-OR on the bit pattern in the 8x8 field with any other selected character.
- X Mirror the bit pattern in the 8x8 field in the X-axis.
- Y Mirror the bit pattern in the 8x8 field in the Y-axis.
- R Rotate the bit pattern in the 8x8 field 90 degrees counterclockwise.
- I Inverts the bit pattern in the 8x8 field.
- SPACE inverts the pixel at the location of the cursor.
- EDIT (CAPS-SHIFT-1) exits the graphics editor.

## 8.3 SOUND.

The file MUSIC (two blocks in size) contains routines to make sound, both for the Spectrum 48 and for the Spectrum 128. MUSIC depends on the assembler and therefore the assembler has to be loaded first, the same way as described in chapter 6 for DOUBLE.

**BEEPER** u1 u2 ---
produces a tone consisting of u1 pulses of u2 machine cycles each on the beeper of the Spectrum 48.

**TONE** u1 u2 ---
produces a tone with a duration of u1 milliseconds and a frequency of u2 Hz on the beeper of the Spectrum 48.

**MS** n1 ---
Delay of n1 milliseconds.

The following commands only work on the Spectrum 128.

**SOUND** 8b n ---
Writes 8b to register n of the sound chip.

**FREQ** u n ---
Sets channel n (n=0, 1 or 2) to a frequency of u Hz.

**VOL** u n ---
Sets channel n (n=0, 1 or 2) to volume u (u in range 0..15).

**TONES** ---
Enables the three tone channels.

**NOISE** ---
Enables the three noise channels. With n 6 SOUND one can change the noise frequency.

**SHUTUP** ---
Disables all sound channels.

The following commands work on the internal beeper or on the sound chip, depending on the Spectrum model. Note: on the Spectrum 128 you have to type 15 1 VOL before hearing the musical notes.

**NOOT** u ---
(runtime) ---
defining word to define a note with frequency u.

**A A# B C C# D D# E F F# G G#**

Play the corresponding note.

**L** u ---
Sets the length of following notes to u beats.

**O+** O- ---
Move an octave up or down.

**>>** << ---
increases or decreases the volume.

**R** ---
Rest (silence) for one beat.

**TEMP** --- addr
variable indicating the tempo, milliseconds per beat.

**OCT** --- addr
variable containing the octave (as a factor with which the not frequency) is multiplied.

**VOLUME** --- addr
variable containing the volume.

**LEN** --- addr
variable containing the number of beats per note.

# 9 THE DECOMPILER

The file UTIL (2 blocks in size) contains the utility to decode the internal structure of a fFORTH word, a decompiler.

**SEE** name shows the following information about a FORTH word:

- the compilation address.
- IMMEDIATE if the word is immediate.
- The word type: CONSTANT, USER, : CREATE or CODEWORD or the name of the defining word if the word was created with CREATE..DOES> (e.g. VOCABULARY).
- The contents of the word:
  - the value for constants and user variables.
  - for colon definitions, a list of words from which the colon definition is compiled. This is not the same as the source code as runtime parts of immediate words are shown (e.g. LIT and BRANCH). Parameters of well-known words are shown: the value following LIT, the branch offset for BRANCH, the string following (.") etc.
  - a Hex and ASCII dump for any other words.
- The size in bytes of the parameter field. This length is incorrect if the next word in the CONTEXT vocabulary does not immediately follow this word in the dictionary.

Decompilation of a colon definition stops when EXIT is encountered.
Words that contain EXIT halfway are not decompiled completely.

Newly defined words that are compiled with operands in the colon definition (like LIT and BRANCH) are not handled by the decompiler. The decompiler will have to be extended to handle any such words.

Besides SEE, the following words are relevant:

**(SEE)** addr ---
Performs the action of SEE on the word with compilation address addr.

**HIGH-NFA** addr1 --- addr2
addr2 is the name field address sin the CONTEXT vocabulary closest above addr1.

**#WORDS** --- u
returns the number of words in the CONTEXT vocabulary and any vocabularies linked to it.

**DOCON** --- addr
**DOCOL** --- addr
**DOUSER** --- addr
constants that return the runtime addresses of CONSTANT, : and USER.

**NEXT** --- addr
Address of the inner interpreter. Is used in the code field of variables.

# 10 MULTI-TASKING.

## 10.1 INTRODUCTION.

Multi-tasking is the execution of multiple programs, seemingly at the same time. Every program has its own stack and user variables and the system switches between the various programs very fast.

Applications:
- control of hardware devices, for instance model trains.
- video games.
- polyphonic music. (128k)
- background music.(128k)
- continuing to work with FORTH while a background task is running for a long time.

The file TASKS (size 3 blocks) contains all basic routines necessary for multi-tasking. It depends on the assembler and therefore the assembler has to be loaded first, the same way as described in chapter 6 for DOUBLE.

## 10.2 CREATING A TASK.

Creating a task is illustrated by means of an example. We create a task that prints a number in the upper left corner of the screen. The number keeps incrementing. First we define a word that returns the current cursor position, so AT can restore it later.

```
: UNAT 24 23689 C@ - 33
  23688 C@ - DUP 32 = IF
  DROP 1+ 0 THEN ;
```

Next we define the task, like a colon definition, but with TASK: and ;TASK

```
TASK: TASK1 DECIMAL 0 BEGIN
 UNAT 0 0 AT 2 PICK 6 .R AT
 1+ DUP 0= SWITCH UNTIL ;TASK
```

Next start the multi-tasking system with STARTUP. The task is started by typing its name: TASK1.

You should now see a number rapidly incrementing in the upper left corner of the screen, while you can continue to work with the Forth system. When you type ' TASK1 STOP, the task stops and the number on the screen stops incrementing. When you type ' TASK1 START, the task continues to run and the incrementing resumes. When you type TASK1, the task starts over from the start and you see the number incrementing from zero again. When you wait long enough and the task has printed all positive and negative 16-bit integers, the task will stop automatically.

If an error message occurs in the normal FORTH task, the multi-tasking system will stop. Restart it again with STARTUP.

## 10.3 INTERNALS.

Every time you create a new task, a new task area is created, which consists of the following parts:

- 2 bytes, start address of the FORTH code.
- 2 bytes, address of the previously defined task. The first task points to the last task, so all tasks are on a circularly linked list.
- 1 byte, status byte, which can have the following values:
    0 is runnable (active).
    1 is new
    2 is finished
    3 is sleeping
    4 is stopped.

  The task will only run when the status byte is 0.

- 2 bytes, the task's stack pointer.
- 60 bytes, user variables.
- 256 bytes, return stack
- 256 bytes, data stack.

The fORTH code of the task will follow this task area.

On the Spectrum 128 all task areas have to be located below address 49152.

Task switching is performed with the word SWITCH. This word saves the return stack pointer and instruction pointer on the task's stack and saves the stack pointer on its location in the current task area. Via the circularly linked list, the next runnable task is selected. The stack pointer is loaded from the newly selected task area, the instruction pointer and return stack pointer are loaded from the stack and the new task will continue.  This way all tasks are executed round-robin.  The user area pointer points to the current task's user area.

To make multi-tasking work, all tasks have to contain the word SWITCH in their loops, so this word will be executed frequently. Otherwise one task may execute for a long time without giving other tasks time to run. If a task never executes the word SWITCH, it will run indefinitely without giving other tasks time to run.

The words KEY and PAUSE both execute switch while they wait. They execute the word whose address is contained in the (WAIT) variable.

## 10.4 WORDS.

**SWITCH** ---
Switches to the next runnable task.

**UPTR** --- addr
Constant, the address of the user area pointer.

**TASK-LINK** --- addr
Variable containing the address of the last created task area.

**FIRST-TASK** --- addr
Variable containing the address of the first created task area.

**TASK:** ---
Defining word to create a new task. Puts FORTH in the compilation state.
At runtime the task will be started from the beginning.

**;TASK** ---
Finishes the definition of a new task. Puts FORTH in the interpretation state.

**TERMINATE** ---
Finishes the current task. The task cannot be restarted with START.

**SLEEP** ---
Puts the current task in the sleep state. START can wake up the task to make it runnable again.

**STOP** addr ---
addr is the code field address of a task. The task is stopped and can be resumed with START.

**START** addr ---

addr is the code field address of a task. The task is resumed from the point where it was stopped.

**MTSK** ---
The task containing the FORTH interpreter. The > prompt is shown when the multi-tasking system is active.

**(START)** ---
Code word used in STARTUP.

**STARTUP** ---
Starts the multi-tasking system

**TLIST** ---
Shows a list of all tasks with their status. The status ACTIVE is shown for the task calling TLIST, all other runnable tasks are listed as RUNNBALE.

# 11 STRINGS AND OTHER DATA TYPES.

## 11.1 INTRODUCTION

The file OBJECT

The file OBJECT (6 screens in size) contains everything that is required to work with strings and several words to create structured data types, such as arrays. Many of the operations described in this chapter can be carried out at a lower level, which is faster. These routines are designed for ease of use, mot for speed.

## 11.2 THE STRING STACK

Strings are arrays of ASCII characters of a variable length. They are stored in a separate stack. Every string consists of a length byte (0 to 255) followed by as many characters as specified by the length. Strings are stored contiguously in the string stack, the last byte of one string is immediately followed by the length byte of the next string.

The top element of the string stack is at the lowest memory address. The string stack pointer points to the length byte of the top element.  By adding the length byte plus 1 to this address, you get the address of the second element on the stack.  The string stack (which grows towards lower memory addresses) is located at the top of free RAM, immediately below the address contained in the LO variable.
The string stack pointer is contained in the variable $SP.

The string stack supports the operations $DUP, $SWAP, $OVER and $DROP.
Strings can be stored in memory using $! and $!LIM and they can be read from memory into the string stack using $@.

The string stack supports the same string operations as most BASIC dialects, like LEFT$, MID$, RIGHT$, STR$, VAL, ASC and CHR$. Further they can be compared with $=, $< and $>, they can be concatenated with $+ and they can be printed with $.

String constants work the same way in the compiler and the interpreter.
A string constant starts with the " character followed by a space, followed by the characters contained in the string. The string constant is terminated by another " character. Example:

```
" ABCD"
```

is a string of length four, containing the ASCII characters A, B, C and D.

```
" This is a string" $.
```

is equivalent to

```
." This is a string"
```

in the compiler or

```
.( This is a string)
```

in the interpreter, but ." and .( do not use the string stack.

Another example:

```
1234. STR$ 1 RIGHT$ $.
```

This converts the double-precision number 1234 into the string " 1234", next the rightmost single character string " 4" is extracted with RIGHT$ and this single character string is printed.

In most BASIC versions you can achieve this with:

```
   PRINT RIGHT$(STR$(1234),1).
```

The same effect can be obtained in pure FORTH with

```
   1234. <# # #> TYPE
```

which is more efficient and does not require the string stack routines.

## 11.3 THE 'TO' CONCEPT.

Standard variables in FORTH-83 always return their addresses and the user is responsible to use the correct fetch or store operation to read or write the value. The user has to choose the correct word @, C@, 2@, $@, !, C!, 2!, $! or $!LIM. Omitting the fetch operation to read the value of a variable is a frequently occurring source of errors.

When using the TO concept, the variable itself takes care of reading or writing its value. A state variable tells the variable whether it should read or write this value. The default state is 'reading', but by using the word TO you switch the state to 'writing' for the next variable access. The variable resets the state variable to 'reading' after writing the value.

Now it is possible to type:

```
   A B + C D + * TO E
```

instead of:

```
   A @ B @ + C @ D @ + * E !
```

For double precision numbers, you can now type:

```
   A B D+ C D D+ D* TO E
```

instead of:

```
   A 2@ B 2@ D+ C 2@ D 2@ D+ D* E 2!
```

So with the TO concept, you only have to change the arithmetic operations, without it you have to change all fetch and store operations.

This implementation maintains the state at runtime. This makes it machine independent, but not very efficient. In some other systems the state is maintained at compile time and the compiler inserts the fetch and store operations, so the state does not have to be checked and maintained at runtime.

## 11.4 STRUCTURED DATA TYPES

When you want to create a variable using the TO-concept, you can type:

```
<type> VALUE <name>
<name> is the name of the variable to be created.
<type> is one of the following:
BOOL for Boolean variables (storage size 1 byte).
CHAR for single characters (storage size 1 byte).
INT  for integers (storage size 2 bytes)
LONG for double precision numbers (storage size 4 bytes)
REAL for floating point numbers (storage size 4 bytes).
n STR for strings, where n is a number indicating the maximum string length
      storage size is n+1 bytes, but at least 5.
```

INT values support +TO in addition to TO to add a number to the variable.

String values support >ELEM and ELEM> operations in addition to TO to read and write single characters in the string.

Example:

```
INT    VALUE #ITEMS
REAL   VALUE SIZE
```

```
10 STR VALUE NAME
```

creates three variables: one integer, one floating point and one string of 10 characters maximum.

You can create a one-dimensional array with

```
   n <type> ARRAY <name>
```

where n is the number of elements contained in the array.

```
3 LONG  ARRAY VECTOR
9 5 STR ARRAY NAMES
```

creates two arrays: one with 3 double precision numbers and one with 9 names of at most 5 characters each.
You can set the value of the first element in VECTOR with

```
1234. TO 0 VECTOR.
```

Note: if A is an integer variable (using the TO-concept), you have to type

```
1234. A TO VECTOR
```

instead of

```
1234. TO A VECTOR
```

because in the latter case you are changing the value of A instead of VECTOR. The array element is selected by an integer in the range 0 to n-1, where n is the number of elements in the array.

You can create a two-dimensional array with:

```
   n1 n2 <type> MATRIX <name>
```

You then get an array with n1 times n2 elements. You select one element with two integers, the first one in the range 0 to n1-1 and the second one in the range 0 to n2-1.

Internally the element type is distinguished by the storage size.
- Elements of size 1 are accessed with C! and C@ and are considered
  to be of type BOOL or CHAR.
- Elements of size 2 are accessed with ! and @ and are considered
  to be of type INT
- Elements of size 4 are accessed with 2! and 2@ and are considered
  to be of type LONG or REAL.
- Elements of size 5 of higher are considered to be of type string
  and are accessed with $!LIM and $@.

It is possible to build other structured data types using the CREATE..DOES> mechanism, like records containing elements of different types. queues, stacks, lists and so on. The new defining word has to store the element size somewhere in the data structure. The runtime part can use this to compute the address of the selected data element. The word (VAL) accesses any single data element, using the element address, the size byte and the state contained in the variable %TO.

## 11.5 THE WORDS

We use the same stack notation as in chapter 3, but with the following additions:

$  is a string on the string stack.

" ---    (compilation)
  --- $  (interpretation)
Reads a string from the input stream until the next " and compiles it into a string literal when the text interpreter is in compilation state. In interpretation state it pushes the string onto the string stack.

"" --- $
string constant, the empty string.

**$!**  $ addr ---
Stores the string $ at address addr.

**$!LIM** $ addr u ---
Stores the string $ at address addr, with a maximum of u characters. The string will be truncated if it is longer.

**$+**   $1 $2 --- $3
Concatenates the strings $1 and $2

**$-**   $1 $2 --- n
compares two strings in ASCII lexicographic order. n is negative if $1 is less than $2, n=0 if the two strings are equal and n is positive if $1 is greater than $2.

**$.** $ ---
prints the string $.

**$<** $1 $2 --- f
f is true is $1 is less than $2, false otherwise.

**$=** $1 $2 --- f
f is true is $1 is equal to $2, false otherwise.

**$>** $1 $2 --- f
f is true is $1 is greater than $2, false otherwise.

**$@** addr --- $
Reads the string $ from memory at address addr.

**$CONSTANT** $ ---
 (runtime)  --- $
defining word. Creates a new word that pushes the constant string on the string stack at runtime.

**$DROP** $ ---
removes a string from the stack.

**$DUP** $ --- $ $
duplicates string on stack.

**$OVER** $1 $2 --- $1 $2 $1
duplicates second string on stack.

**$S0** --- addr
The highest address of the string stack.

**$SP** --- addr
variable containing the string stack pointer.

**$SP!** ---
resets the string stack pointer to the topmost address, thereby
emptying the string stack.

**$SWAP** $1 $2 --- $2 $1
swaps the top two string stack elements.

**%TO** --- addr
variable containing the state used by any TO-variable to select its action.

0 for fetch.
1 for store, set with TO.
2 for add to variable, set with +TO
4 for writing a single character of a string, set with >ELEM
5 for reading a single character of a string, set with ELEM>

**(")** --- $
runtime part of "

**(CHAR) addr --- c %TO=0**
      c addr ---    %TO=1
internal word to access values of type CHAR and BOOL.

**(INT)** addr --- n %TO=0

```
      n addr ---    %To=1 of 2
internal word to access values of type INT.
```

**(LONG)**
```
        addr --- d  %TO=0
      d addr ---    %TO=1
internal word to access values of type LONG and REAL.
```

**(STR)**
```
        addr u --- $ %TO=0
      $ addr u ---   %TO=1
    c n addr u ---   %TO=4
      n addr u --- c %TO=5
internal word to access values of string type.
     addr is the address where the string is stored.
        u is the maximum string length.
    c is the single character read or written.
        n is the index in the string to read or write a single character.
```

**(VAL)**
```
  addr u ---
accesses a variable of one of the above-mentioned types
     addr is the address where the values is stored.
        u is the size in bytes, also indicating the type.
```

**+$** `$1 $2 --- $3`
```
concatenates strings in reverse order, $2 is followed by $1.
```

**+TO** `---`
```
specifies that the next variable operation will be addition to value.
```

**-RIGHT$** $1 u --- $2
$2 is the rightmost part of $1, from the u'th character.

**>ELEM** ---
specifies that the next variable operation is a single character write.
Example: if NAME is a string variable, &A 1 >ELEM NAME sets the first character to 'A'.

**?RANGE** u1 u2 ---
reports an error if U1>u2

**ARRAY** u1 u2 ---
creates an array of type u2 containing u1 elements.

**ASC** $ --- c
c is the first character of $

**BOOL** --- 1
indicates variable type.

**CHAR** --- 1
indicates variable type.

**CHR$** c --- $
$ is a single-character string containing the character c.

**ELEM>** ---
specifies that the next variable operation is a single character read.

Example: if NAME is a string variable, 3 ELEM> NAME returns the third character in the string.

**ER** ---
Reports the error "INVALID OP".

**GET$** --- $
Reads a string from the keyboard with EXPECT.

**INT** --- 2
indicates variable type.

**LEFT$** $1 u ---$2
$2 is the leftmost part of $1, u characters in length.

**LEN** $ --- u
u is the length of string $

**LONG** --- 4
indicates variable type.

**MATRIX** u1 u2 u3 ---
creates a matrix of type u3 with u1 rows and u2 columns.

**MID$** $1 u1 u2 --- $2
$2 is the substring of $1 starting at the u1'th position and u2 character long.

**REAL** --- 4
indicates variable type.

**RIGHT$** $1 u1 --- $2
$2 is the rightmost part of $1, u1 characters in size.

**RV** ---
Returns %TO to the read state.

**STR** u1 --- u2
indicates variable type, string with u1 characters maximum.

**STR$** d --- $
converts double precision integer to string.

**TO** ---
specifies that the next variable access will be a store.

**VAL** $ --- d
converts string to double precision integer.

**VALUE** u1 ---
Creates variable (single value) of type u1.

**WORD$** --- $
Converts the next word of the input stream to a string.

## 11.6 ADAPTATION TO MULTI-TASKING

In order to make these routines suitable for multi-tasking, you must define the $SP and %TO variables as USER variables, except when one of the following conditions is true:
- Only one task makes use of these routines.
- Every SWITCH occurs when the string stack is empty and the %TO variable is in the read state.

If you make $SP a user variable, you have to reserve a separate string stack area for each task. For each task the string stack pointer has to be initialized to the end of its string stack area.

-- End of document -

# ANNOTATED META SOURCE CODE OF SPECTRUM FORTH-83.

```
\ Written by L.C. Benschop in 1988, annotations written in 2015.

\ This is a text file that contains the source code from the files META1,
\ META2, META3 and META4 from the Spectrum FORTH-83 distribution.
\
\ These files have to be run in succession from Spectrum FORTH-83.
\ RUN META1 RUN META2 RUN META3 RUN META4
\
\ Together these files rebuild the original FORTH system from source.
\ The end product is the file FORT83.BIN. See also section 4.5 of the manual.
\ When the newly built FORTH system loads the file EDITOR and is saved,
\ it will be identical to the base FORTH system as distributed.
\
\ Code has been reformatted and comments are added.
\ But all source instructions themselves have been preserved, including
\ bugs and things that are so awfully ugly after all these years.
\
\ Note: this text file in this form cannot be loaded as source code
\ into the Spectrum FORTH-83 system. Use the original files META1..META4
\ instead. This source code has never compiled on any system other than
\ Spectrum FORTH-83 and will not work unmodified.


\ FILE META1
\ Scr#1
\ Metaforth 83 COMPILER DEF'S
```

```
HEX \ We will enter all numbers in hex, except when switching briefly to
    \ decimal for some numbers.

\ The meta compiler will build a FORTH system image at a different address
\ from where the image will later be loaded and run. The newly built
\ image will be called the TARGET system.
\
\ The HOST system is the FORTH system that runs the meta compiler.
\
\ As the newly built system resides at a different address, FORTH cannot
\ EXECUTE any words it has built in the target system. The target system
\ and the host system can in
\ principle be totally different FORTH systems, for different CPUs,
\ with different threading style and with a different dictionary format.
\

B400 CONSTANT VIRTSTART
\ Start address of the target system, where it will be built by the meta
\ compiler. This address is far enough above HERE, so the host dictionary
\ will not run into it when it is extendend and far enough below the RAM
\ disk. Ugly to steal a memory area in this way, but those were the days.
\ CREATE VIRTSTART 2800 ALLOT would have been better.

DECIMAL 26000 1030 + CONSTANT ORIGIN HEX
\ Address where the target system will be loaded and run when it is finished.
\ This is the COLD start entry point. See memory map in section 4.3 of the
\ manual.
\
\ Uglyness alert: the saved image will be from address 27028, 2 bytes
\ below the ORIGIN address, so it includes the user area pointer as well.
```

```
VIRTSTART ORIGIN - CONSTANT OFFSET
\ Address difference between where the target system is stored now
\ and where it will be loaded once it is finished.

\ Operations on target system.

\ The usual fetch and store operations. With these the meta compiler
\ can pretend it reads and writes data at the addresses in the finished
\ system.
: C!-T OFFSET + C! ;
: !-T  OFFSET + !  ;
: C@-T OFFSET + C@ ;
: @-T  OFFSET + @  ;

VARIABLE LINK-T 0 LINK-T !          \ Link between words in the target directory.
VARIABLE DP-T   ORIGIN DP-T !       \ Dictionary pointer in the target system.

\ More operations on the target system.
: HERE-T DP-T @ ;
: ALLOT-T DP-T +! ;
: ,-T HERE-T !-T 2 ALLOT-T ;
: C,-T HERE-T C!-T 1 ALLOT-T ;

: NMOVE ( src dst n --- )
\ Move a block of n bytes to the target system. Source is a host address,
\ destination is a target address.
        SWAP OFFSET + SWAP
        CMOVE ;
```

```
\ This word creates a header for a new word in the target system (only link and
\ name fields), code field not created. See section 4.2 in manual for header
\ structure.
\ For each word in the target system, a word is also created in the
\ host dictionary.
: CREATE-T
  CREATE                     \ Create a word in the host dictionary as well.
                             \ will end up in the META vocabulary of host system.
  LINK-T @ ,-T               \ Put link field in target system (= link to previous word)
  HERE-T LINK-T !            \ Update target link variable, so it points to the new word
  LATEST HERE-T              ( host-nfa target-nfa )
  OVER C@ 01F AND 1+         ( host-nfa target-nfa namelength )
  DUP >R NMOVE R> ALLOT-T    \ Copy the name of the latest word from the
                             \ host dictionary to the target and allocate space.
  HERE-T 1- DUP C@-T 80 OR SWAP C!-T \ Set bit 7 in final character of name.
                             \ This is not even required as this bit was
                             \ already set in the name copied from the host.
                             \ but can be required when running on different host.
  IMMEDIATE                  \ Make the word in the host dictionary immediate.
;


VOCABULARY META
\ The META vocabulary contains all words that the meta compiler uses
\ when compiling for the target system. Most words are created by
\ CREATE-T and will add their own code field address to the target dictionary.
\ Some words (for instance IF) are special definitions that run on the
\ host system, but compile code for the target system.

VOCABULARY TARGET
```

```
\ The TARGET vocabulary contains all words that are immediate in the
\ target system. They shall not be used by the meta compiler, but instead
\ the meta compiler specific counterparts must be used.

\ It's so ugly to move the immediate words from the target system from
\ the META to the TARGET vocabulary, but that's how it's done here.
\ Later meta compilers written by me use TRANSIENT and TARGET
\ vocabularies, where TRANSIENT is always searched first and all target
\ words are created in TARGET. In later FORTHs the search order can
\ be specified more easily.
\
\ This meta compiler was started on a FORTH where VOCABULARY was buggy
\ and it might not even be possible to have three vocabularies in the
\ search order. So that may be why it was done this way.

\ Scr#2
\ METAFORTH83 ASSEMBLER #1

\ The Meta FORTH assembler was copied from the normal assemler (see file
\ ASSEMBLER) and adapted to put the generated code in the target system.
\ For assembler syntax see chapter 5 of the manual.
\
\ With a sligtly better design we could have used the exact same source
\ code for both.

VOCABULARY ASSEMBLER IMMEDIATE \ Assembler words get their own vocabulary.
HEX

VARIABLE XY
\ Variable that contains the index register opcode prefix byte,
\ DD for IX, FD for IY.

\ These definitions for CODE and ;C are for generating code definitions
\ on the host system. Just copied from normal assembler and not used by
\ meta compiler.
: CODE CREATE
  -3 ALLOT                    \ Remove the code field of the newly created word.
  !CSP [COMPILE] ASSEMBLER
  HEX DD XY ! ;
: ;C DECIMAL  CURRENT @
  CONTEXT ! ?CSP
  SMUDGE ;

\ MACRO and ENDM are not used in the meta compiler either.
: MACRO [COMPILE] ASSEMBLER
  [COMPILE] : ;
ASSEMBLER DEFINITIONS
: ENDM [COMPILE] ; ; IMMEDIATE

: %X DD XY ! ;             \ Select IX for index register operations.
: %Y FD XY ! ;             \ Select IY for index register operations.

: CON CONSTANT ;           \ CONSTANT is such a long word, so make a shorter synonym.

\ These definitions provide shorter synonyms for operations on
\ the target system. They allowed turning the assembler into a meta
\ assembler with minimal editing.
: HERE HERE-T ;
: C_ C,-T ;
: _   ,-T ;
: C!_ C!-T ; : !_ !-T ;

: 8* 2* 2* 2* ;

\ The following constants specify register names.
0 CON B 1 CON C
2 CON D 3 CON E
4 CON H 5 CON L
6 CON M 7 CON A                 \ M is memory byte addressed by (HL).
6 CON SP 6 CON AF

\ Specify use of IX/IY register (instead of HL) in some instructions.
\ X JPHL  becomes for JP (IX).
: X XY @ C_ ;

\ Allow use of lower and upper half of IX/IY register, not used.
: XL X L ;
: XH X H ;


: ?PAGE  ( offs --- offs)
\ Check that offs is in range -128..127, for use in relative branches.
   DUP 80 + 00FF SWAP U<
   ABORT" BRANCH TOO LONG" ;

\ Scr#3
\ METAFORTH83 ASSEMBLER #2

\ The words M1..MC are simple defining words, one for each category
\ of instructions.
```

```
: M1 CREATE C, DOES> C@ C_ ;         \ Single opcode byte, no operands.
: M2 CREATE C, DOES> C@ + C_ ;       \ Single opcode byte, 8-bit register operand.
: M3 CREATE C, DOES> C@ SWAP
  8* + C_ ;                          \ Single opcode byte, 16-bit register operand.
: M4 CREATE C, DOES> C@ C_ C_ ;      \ Opcode byte plus 8-bit immediate.
: M5 CREATE C, DOES> C@ C_ _ ;       \ Opcode byte plus 16-bit immediate.
: M6 CREATE C, DOES> 00CB C_
  C@ + C_ ;                          \ CB prefix opcode, register operand.
: M7 CREATE C, DOES> 00CB C_
  C@ + SWAP 8* + C_ ;                \ BIT/SET/RES instructions, register operand.
: M8 CREATE , DOES> @ _ ;            \ Two byte opcode, no operands.
: M9 CREATE C, DOES> C@ C_ HERE
  1+ - ?PAGE C_ ;                    \ Relative branches.
: MA CREATE C, DOES> X C@ C_ C_
;                                    \ Indexed instructions.
: MB CREATE C, DOES> X 00CB C_
  C@ SWAP C_ C_ ;                    \ Indexed instructions with CB prefix.
: MC CREATE C, DOES> X 00CB C_
  C@ ROT ROT C_ 8* + C_ ;            \ Indexed BIT, SET and RES.

\ Some instructions get their own definition, no common defining word,
: LDP# 8* 1+ C_  _ ;                 \ Load 16-bit immediate.
: LD# 8* 06 + C_ C_ ;                \ Load 8-bit immediate.
: LD 8* 40 + + C_ ;                  \ Register-to-register move.
: SBCP 00ED C_ 8* 42 + C_ ;          \ 16-bit SBC
: ADCP 00ED C_ 8* 4A + C_ ;          \ 16-bit ADC
: STP 00ED C_ 8* 43 + C_ _ ;         \ 16-bit store to absolute address.
: LDP 00ED C_ 8* 4B + C_ _ ;         \ 16-bit load from absolute address.
```

```
\ Scr#4
\ METAFORTH83 ASSEMBLER #3

\ Instructions, ordered by Z80 opcode.
00 M1 NOP  02 M3 STAP
03 M3 INC  04 M3 INR        \ INC is increment 16-bit register, INR is increment 8-bit
05 M3 DER  07 M1 RLCA       \ DER is decrement 8-bit register.
08 M1 EXAF 09 M3 ADDP
0A M3 LDAP 0B M3 DEC        \ DEC is decrement 16-bit register
0F M1 RRCA 10 M9 DJNZ
17 M1 RLA  18 M9 JR
1F M1 RRA  20 M9 JRNZ
22 M5 STHL 27 M1 DAA
28 M9 JRZ  2A M5 LDHL
2F M1 CPL  30 M9 JRNC
32 M5 STA  37 M1 SCF
38 M9 JRC  3A M5 LDA
3F M1 CCF  76 M1 HALT
80 M2 ADD  88 M2 ADC
90 M2 SUB  98 M2 SBC
B8 M2 CP
\ Note that XOR AND and OR are missing here.
\ They are defined much later when the original FORTH words with the same
\ names are no longer needed in the ASSEMBLER vocabulary.
C1 M3 POP
C2 M5 JPNZ C3 M5 JP
C5 M3 PUSH C6 M4 ADD#
C7 M2 RST  C9 M1 RET
CA M5 JPZ  CD M5 CALL
CE M4 ADC# D2 M5 JPNC
```

```
D3 M4 OUT  D6 M4 SUB#
D9 M1 EXX  DA M5 JPC
DB M4 IN   DE M4 SBC#
E2 M5 JPPO E3 M1 EXSP
E6 M4 AND# E9 M1 JPHL
EA M5 JPPE EB M1 EXDE
EE M4 XOR# F2 M5 JPP



\ Scr#5
\ METAFORTH83 ASSEMBLER #4
F3 M1 DI   F6 M4 OR#
F9 M1 LDSP FA M5 JPM
FB M1 EI   FE M4 CP#

\ CB prefix instructions.
00 M6 RLC  08 M6 RRC
10 M6 RL   18 M6 RR
20 M6 SLA  28 M6 SRA
38 M6 SRL  40 M7 BIT
80 M7 RES  C0 M7 SET

\ ED prefix instructions.
B0ED M8 LDIR B8ED M8 LDDR
44ED M8 NEG  57ED M8 LDAI
47ED M8 LDIA 56ED M8 IM1
5EED M8 IM2  B1ED M8 CPIR

\ Indexed instructions.
```

```
86 MA )ADD 8E MA )ADC
96 MA )SUB 9E MA )SBC
A6 MA )AND AE MA )XOR
B6 MA )OR  BE MA )CP
34 MA )INR 35 MA )DER
06 MB )RLC 0E MB )RRC
16 MB )RL  1E MB )RR
26 MB )SLA 2E MB )SRA
3E MB )SRL 46 MC )BIT
86 MC )RES C6 MC )SET

\ Indexed instructions directly defined without a common defining word.
: )LD X SWAP 8* 46 + C_ C_ ;
: )ST X SWAP 70 + C_ C_ ;
: )LD# X 36 C_ C_ C_ ;
: )LDP OVER 1+ OVER )LD 1+ )LD ;
: )STP OVER 1+ OVER )ST 1+ )ST ;


\ Scr#6
\ METAFORTH83 ASSEMBLER #5
: CLR 0 SWAP LDP# ; \ Clear 16-bit register.
: MOV 2DUP LD 1+ SWAP 1+ SWAP LD
 ; \ Macro to copy 16-bit register H B MOV


: NOT 08 + ;
\ Condition codes (Capitalized) for the relative jump instructions.
\
\ Specific condition code values are Z80 opcodes for required instructions.
\ Note that condition codes in the instructions are the opposite of
\ the condition code mnemonics here.
\ Example: Z IF requires an instruciton JR NZ, so Z specifies the JR NZ opcode.
20 CON Z  28 CON NZ
30 CON CS 38 CON NC

\ Condition codes (lowercase) for the absolute jump instructions, can
\ be followed by NOT for inverse condition codes.
\
\ Specific condition code values are Z80 opcodes for required instructions.
\ Example z specifies JP NZ opcode.
C2 CON z  D2 CON cs
E2 CON pe E2 CON v
F2 CON m

\ Control structures generating relative jumps, capitalized, use
\ capitalized conditions.
\ NC IF ... THEN   or BEGIN ...  Z UNTIL
\
\ These definitions look very terse and cryptic,
\
\ But they work similarly to the FORTH control structure words,
\ except they lay out Z80 opcodes instead of the FORTH BRANCH/?BRANCH words.
\ (opcodes mostly supplied in condition codes) and single byte addresses follow
\ the opcodes.

: THEN 000A ?PAIRS HERE 1- OVER
 - ?PAGE SWAP C!_ ;
: IF _ HERE 1- 000A ;
: ELSE 000A ?PAIRS 0018 IF ROT
```

```
  SWAP THEN 000A ;
: UNTIL _ 000B ?PAIRS 1- HERE 1-
  SWAP OVER - ?PAGE SWAP C!_ ;
: BEGIN HERE 000B ;
: AGAIN 0018 UNTIL ;  \ Unconditional loop BEGIN..AGAIN.
: DSZ 0010 UNTIL ;    \ BEGIN..DSZ makes a loop with DJNZ instruction.
: REPEAT 2SWAP AGAIN 2- THEN ;
: WHILE IF 2+ ;

\ Control structures generating absolute jumps, lowercase, use
\ lowercase conditions.
\ cs NOT if ... then   or begin ...  z until
: then 08 ?PAIRS HERE SWAP !_ ;
: if C_ HERE 0 _ 0008 ;
: else 08 ?PAIRS C3 if ROT
  SWAP then 08 ;
: begin HERE 09 ;
: until C_ 09 ?PAIRS _ ;
: while if 2+ ;
: again C3 until ;
: repeat 2SWAP again 2- then ;

\ Scr#7
\ METAFORTH83 ASSEMBLER #6

\ The missing AND, OR and XOR are defined at the end.

: RETC 8 XOR 2- C_ _ ;                \ Conditional return
: CALC 8 XOR 2+ C_ _ ;                \ conditional call.
A0 M2 AND B0 M2 OR
```

```
: SUBP A AND SBCP ;                    \ Macro, 16-bit subtract.
: TST DUP A LD 1+ OR ;
\ Macro to test 16-bit regsiter for zero H TST tests HL
A8 M2 XOR
CF M4 HOOK \ RST 8 followed by byte: error code or Interface 1 system call.
D7 M1 PRT  \ RST 16 = print character.
FORTH DEFINITIONS DECIMAL

\ Assembler ends here.

\ Scr#8
\ Metaforth 83 compiler
META DEFINITIONS HEX

\ Word to create code definitions on the target system.
: CODE
  CREATE-T               \ Create new word in target system.
  [OMPILE] ASSEMBLER     \ select assembler vocabulary.
  SMUDGE                 \ Make word unfindable in host system.
  !CSP                   \ Mark stack
  HERE-T ,               \ Store target CFA into host word.
  DOES>
  @ ,-T ;                \ runtime: add CFA to compiled target colon definition.

ASSEMBLER DEFINITIONS
: ;C
  CURRENT @ CONTEXT !    \ Restore vocabulary.
  ?CSP                   \ Check that stack pointer is the same as when CODE
                         \ definition was started.
  SMUDGE                 \ Make word findable in host system.
```

```
;

\ Variables used by assembler to store jump addresses
VARIABLE L1 VARIABLE L2
META DEFINITIONS

\ The word [COMPILE] for the meta compiler. Will look up the immediate
\ word in the TARGET vocabulary and compile its address.
: [COMPILE]
  TARGET '             \ Look up in TARGET vocabulary.
  3 +                  \ Convert CFA to PFA.
  @ ,-T                \ Read target CFA from parameter field and compile to
                       \ target.
  META                 \ Return to META vocabulary.
; IMMEDIATE

: IMM IMMEDIATE ;      \ Create a synonym for the original IMMEDIATE.

\ IMMEDIATE moves the latest word from the META vocabulary to the TARGET
\ vocabulary, so another word in the META vocabulary with the same name
\ will be found instead.
: IMMEDIATE
  LATEST DUP 2-           (latest-nfa latest-lfa)
  DUP @ CURRENT @ !       \ Adjust CURRENT vocabulary to point to previous
                          \ definition, removing the word from current (=META).
  TARGET                  \ Set TARGET to context.
  CONTEXT @ @ SWAP !      \ Make link field in latest def point to previous
                           \ def in TARGET.
  CONTEXT @ !              \ Make target point to latest def.
  META                     \ Switch back to META
  LINK-T @ DUP C@-T 40 OR
  SWAP C!-T               \ Set bit 6 in first name field byte in target system
                          \ to mark the word immediate there.
;

FORTH DEFINITIONS

VARIABLE 'LIT             \ Variable to contain the address of LIT primitive in
                          \ target system.
\ Meta interpreter: Execute words and take care to compile literals.
\      Execution of a normal word from the META vocabulary means compiling
\      it to the target system.
\      Some words will execute special actions.
: META-INTERPRET
  BEGIN
    BL WORD DUP C@        \ Read next word until input stream exhausted.
  WHILE
    FIND
    IF
      EXECUTE             \ Execute any word found.
    ELSE
      NUMBER DROP STATE @ \ Forget about double precision literals.
      IF
        'LIT @ ,-T ,-T    \ If compiling, add literal to target system.
      THEN
    THEN
  REPEAT DROP ;

\ Scr#9
\ Metaforth 83 compiler
\ Variables to contain the addresses of several FORTH primitives.
\ Will be filled in when these words are defined later in the target system.
\ DOCOL, DOCON, DOUSER and NEXT are constants, will be patched
\ later when the corresponding primitives are defined.
VARIABLE 'BRANCH VARIABLE 'LOOP
VARIABLE '+LOOP  VARIABLE '."
VARIABLE '?DO    VARIABLE '?BR
0 CONSTANT DOCOL                \ Runtime part for colon definitions.
VARIABLE 'EXIT
0 CONSTANT DOCON                \ Runtime part for constants.
VARIABLE 'DO
0 CONSTANT DOUSER               \ Runtime part for user variables.
VARIABLE 'ABORT
0 CONSTANT NEXT                 \ Address of inner interpreter.

VARIABLE VL                     \ Pointer to fix up FORTH vocabulary in target system.
VARIABLE 'CODE                  \ Runtime part for DOES>

META DEFINITIONS

\ Next create the words that will perform special actions in the
\ meta compiler in the META vocabulary. When executed, they will
\ do something different than just adding the code address to the
\ current colon definition.

: ."
  '." @ ,-T                 \ Compile runtime for ."
  22 WORD                   \ Read word up to next "
  HERE-T HERE C@ 1+ NMOVE   \ Copy counted string to target.
  HERE C@ 1+ ALLOT-T          \ Allocate space for it.
; IMM

: DO
  'DO @ ,-T                   \ Compile runtime
  HERE-T                      \ Put current address on stack for forward ref.
  0 ,-T                       \ Add extra cell for forward address.
  3                           \ 3 indicates loop structure.
; IMM

: LOOP
  3 ?PAIRS                    \ Check for nesting consistency.
  'LOOP @ ,-T                 \ Compile runtime part.
  HERE-T  SWAP !-T            \ Fill in forward reference at address after DO.
; IMM

: +LOOP
  3 ?PAIRS                    \ Check for nesting consistency.
  '+LOOP @ ,-T                \ Compile runtime part.
  HERE-T  SWAP !-T            \ Fill in forward reference at address after DO.
; IMM

: ?DO
  '?DO @ ,-T                  \ Compile runtime
  HERE-T                      \ Put current address on stack for forward ref.
  0 ,-T                       \ Add extra cell for forward address.
  3                           \ 3 indicates loop structure.
; IMM

: IF
```

```
  '?BR @ ,-T                  \ Compile conditional branch
  HERE-T                      \ Put current address on stack for forward ref.
  0 ,-T                       \ Add extra cell for forward address.
  2                           \ 2 indicates if structure.
; IMM

: ELSE
  2 ?PAIRS                    \ Check for nesting consistency.
  'BRANCH @ ,-T               \ Compile unconditional branch.
   HERE-T                     \ Put current address on stacl
   0 ,-T                      \ Add extra cell for forward address.
   HERE-T ROT !-T             \ Resolve original forward reference originating at IF,
                              \ will jump past unconditional branch at ELSE.
   2                          \ 2 indicates if structure.
; IMM

: THEN
  2 ?PAIRS                    \ Check for nesting consistency.
  HERE-T SWAP !-T             \ Fill in forward reference.
; IMM

: BEGIN
  HERE-T                       \ Put current address on stack for backward reference.
  1                            \ 1 indicates BEGIN structure.
; IMM

: UNTIL
  '?BR @ ,-T                   \ Compile conditional branch.
  1 ?PAIRS                     \ Check for nesting consistency.
  ,-T                          \ Add backward address.
```

```
; IMM

: WHILE
  1 ?PAIRS                    \ Check for nesting consistency.
  '?BR @ ,-T                  \ Compile conditional branch.
  HERE-T                      \ Put current address on stack for forward reference.
  0 ,-T                       \ Add cell for forward address.
  4                           \ 4 indicates WHILE
; IMM

\ Scr#10
\ Metaforth 83 compiler
: REPEAT
  4 ?PAIRS                    \ Check for nesting consistency.
  'BRANCH @ ,-T               \ Compile unconditional branch.
  HERE-T 2+ SWAP !-T          \ Resolve the forward reference originating in WHILE.
  ,-T                         \ Add the backward address to BEGIN.
; IMM

: ABORT"
  'ABORT @ ,-T                \ Compile runtime.
  22 WORD                     \ Read word until next "
  HERE-T HERE C@ 1+ NMOVE     \ Move counted string to target.
  HERE C@ 1+ ALLOT-T          \ Allocate space.
; IMM

: CONSTANT
  CREATE-T                    \ Create new target word.
  HERE-T ,                    \ Store target compilation address into host word.
  CD C,-T DOCON ,-T ,-T       \ Compile a CALL to DOCON, followed by the constant.
  DOES> @ ,-T ;              \ At runtime, compile to target system.

: CREATE
  CREATE-T                   \ Create new target word.
  HERE-T ,                   \ Store target compilation address into host word.
  CD C,-T NEXT ,-T           \ Compile a call to NEXT.
  DOES> @ ,-T ;              \ At runtime, compile to target system.

: VARIABLE CREATE            \ Create word in target system, just as CREATE.
  0 ,-T ;                    \ Allocate space for variable
.
: DOES>
  'CODE @ ,-T                \ Compile the (;CODE) word.
  CD C,-T DOCOL ,-T          \ Compile a call to DOCOL.
; IMM

: USER
  CREATE-T                   \ Create new target word.
  HERE-T ,                   \ Store target compilation address into host word.
  CD C,-T DOUSER ,-T C,-T    \ Compile a call to DOUSER followed by byte offset.
  FORTH                      \ Select the FORTH vocabulary now to get the correct
                             \ version of DOES>
  DOES> @ ,-T ;              \ At runtime, compile to target system.

\ Things get fairly ugly towards the end of this section, as more and
\ more common words get a new definition in the META vocabulary, hiding
\ their normal FORTH definitions, so we have to switch vocabularies
\ quite a lot.
  META DEFINITIONS
: ;
```

```
  'EXIT @ ,-T                 \ Compile EXIT to the current colon definition.
  CURRENT @ CONTEXT !         \ Set context vocabulary.
  ?CSP                        \ Check stack balancing.
  SMUDGE                      \ Make word findable.
  0 STATE !                   \ Switch to interpretation state.
;  IMM

: LITERAL
  'LIT @ ,-T ,-T              \ Compile a literal.
  FORTH ;
  META DEFINITIONS IMM

: :
  CREATE-T                    \ Creat new target word
  HERE-T ,                    \ Store target compilation address into host word.
  CD C,-T DOCOL ,-T           \ Compile a call to DOCOL.
  SMUDGE                      \ Make the word unfindable.
  !CSP                        \ Mart start stack pointer.
  FORTH 0BF STATE !           \ Switch to compilation state.
  DOES> @ ,-T                 \ At runtime compile to target system
; META DEFINITIONS IMM
\ This is the last colon definition running on the host system that
\ will be defined in the META vocabulary. The original FORTH : is now
\ hidden.

FORTH DEFINITIONS

\ Create a few aliases for words that we need in their oriignal FORTH
\ meaning, but that will be shadowed in the META vocabulary.
: } ] ;
: O+ + ;
: DEF DEFINITIONS ;

DECIMAL

\ FILE META2
\ Scr#1
\ METAFORTH83 INNER INTERPRETER

\ Until now, nothing has been compiled in the target image.

HEX ASSEMBLER

 \ The following three entry points are the first instructions to add to
 \ the target system.
 \ Jump target addresses will be patched later.
 0 JP                  \ Cold entry point
 0 JP                  \ Warm entry point
 0 JP                  \ BCAL return entry point.
FORTH          DEFINITIONS

\ Constants that define addresses of variables in the user variable area.
HERE-T CONSTANT 'BASE
HERE-T 2 + CONSTANT 'DP
HERE-T 4 + CONSTANT 'KEY
HERE-T 6 + CONSTANT 'EMIT
HERE-T 8 + CONSTANT RPTR
HERE-T A + CONSTANT 'WAIT
HERE-T C + CONSTANT 'FENCE
HERE-T E + CONSTANT 'S0
```

```
HERE-T 10 + CONSTANT 'R0
ORIGIN 2- CONSTANT UPTR

3C ALLOT-T              \ Allocate 60 bytes in the target system for the
                        \ user variables.

VARIABLE 'I             \ Address of the code definition for I, will be
                        \ used as a jump target by the assembler.

ASSEMBLER DEFINITIONS
: JPIX %X X JPHL ;      \ Add JPIX instruction to the assembler.

\ The inner interpreter starts here.
\ See section 4.1 of the manual.
HERE-T ' NEXT >BODY  ! \ Patch the NEXT constant, used by CREATE/VARIABLE

\ DE contains the instruction pointer.
   EXDE                 \ Instruction pointer to HL.
 M E LD
  H INC
 M D LD
  H INC                 \ Read word into DE, incrementing HL
   EXDE                 \ Instrucion pointer again in DE.
   JPHL                 \ Jump to the address just read.

\ DOCON, runtime part of CONSTANT.
HERE-T ' DOCON >BODY ! \ Patch the DOCON constant.
  H POP                 \ Pop parameter field address from stack, put there by CALL
 M C LD
  H INC
 M B LD                \ Read word from address into BC.
 B PUSH                \ Push constant to stack.
   JPIX                \ IX always contains the NEXT address, JPIX short jump to NEXT.

\ DOCOL, runtime part of colon defintiions.
HERE-T ' DOCOL >BODY ! \ Patch the DOCOL constant.
RPTR LDHL              \ Read return stack pointer to HL
   H DEC
  D M LD
   H DEC
  E M LD               \ Write DE (instruciton pointer) to return stack,
                       \ thereby decrementing pointer.
RPTR STHL              \ Store return stack pointer.
   D POP               \ Pop instruction pointer from stack, put there by CALL DOCOL.
    JPIX

\ DOUSER, runtime part of user variables.
HERE-T ' DOUSER >BODY !
   H POP               \ Pop parameter field address from stack, put there by CALL
  M C LD               \ Load single byte offset into C.
 0 B LD#               \ Clear B.
UPTR LDHL
  B ADDP               \ Add to UPTR variable.
  H PUSH               \ Push variable addres on stack.
    JPIX

\ Scr#2
\ META PRIMITIVES LIT,BRANCH,DO
META DEFINITIONS
```

```
\ LIT is the first FORTH word defined. The previous code (NEXT, DOCOL)
\ had no headers.
\ Primitives used in compiling literals and control structures.
\ When these words are defined, the corresponding variables in the meta
\ compiler are filled in.
\ This filling in is an action performed by the meta compiler when this code
\ is assembled, it is not part of the generated code.
CODE LIT ( --- 16b )
   HERE-T 'LIT !        \ Fill in the 'LIT variable.
   EXDE                 \ Instruction pointer now in HL
   M C LD
   H INC
   M B LD
   H INC                \ Read word at instruction pointer in BC, increment HL.
   B PUSH               \ Push literal.
   NEXT 1+  JP          \ Skip the first EXDE in NEXT.
;C

CODE BRANCH  ( --- )
   HERE-T 'BRANCH !     \ Fill in the 'BRANCH variable.
   EXDE
 M E LD
  H INC
 M D LD
   JPIX ;C

CODE ?BRANCH ( f --- )
   HERE-T '?BR !        \ Fill in the '?BR variable.
  B POP                 \ Pop the flag
 B A LD
```

```
   C OR                 \ Test if it is zero.
'BRANCH @ JPZ           \ If zero, do the branch.
  D INC
  D INC                \ Not zero, Skip the branch address.
   JPIX ;C

CODE EXECUTE ( addr --- )
  H POP                \ Pop execution address
    JPHL ;C            \ Jump to it. Note: just RET would have worked too.

CODE EXIT ( --- )
   HERE-T 'EXIT !      \ Fill in the 'EXIT variable.
RPTR LDHL              \ Load return stack pointer into HL
   M E LD
    H INC
   M D LD
    H INC              \ Load DE from return stack pointer, incrementin HL
RPTR STHL              \ Store updated return stack pointer.
     JPIX ;C

\ Return stack for loop. A do loop pushes three items on the return stack.
\ Limit
\ reverse branch address (jump to here if loop repeats).
\ current index (represented as (index-limit) xor 0x8000. This is at the top.
\              The current index is represented this way so it is easier
\              to check whether index has crossed the boundary between
\              limit-1 and limit, acoording to
\              the rules of Forth-83, even with negative increment in +LOOP.
CODE (DO) ( w1 w2 --- )
   HERE-T 'DO !        \ Fill in the 'DO variable.
   H POP
   B POP        \ Initial value in HL, limit in BC.
  H PUSH        \ Push initial value back.
RPTR LDHL       \ Return stack pointer in HL.
   H DEC
  B M LD
   H DEC
  C M LD        \ Push limit value on return stack.
   D INC
   D INC        \ Increment the instruction pointer, skip branch address.
   H DEC
  D M LD
   H DEC
  E M LD        \ Push current instruction pointer on return stack.
    EXSP        \ initial value now in HL, Return stack pointer on stack.
   A AND
  B SBCP        \ Subtract limit value.
  H A LD
 80 XOR#
  A B LD
  L C LD        \ Flip most significant bit, move to BC.
   H POP        \ Get return stack pointer from stack,
   H DEC
  B M LD
   H DEC
  C M LD        \ Push (initial - limit) XOR 0x8000 onto return stack.
RPTR STHL       \ Save return stack pointer.
    JPIX ;C

CODE (?DO) ( w1 w2 --- )
```

```
 HERE-T '?DO !    \ Fill in the '?DO variable.
   H POP
   B POP          \ initial value in HL, limit in BC,
  B SUBP          \ Compare.
  NZ IF
    B ADDP        \ Not equal: Reverse the subtraction.
    B PUSH
    H PUSH        \ Push operands back onto stack.
    'DO @ JR      \ Perform regular DO.
  THEN
  'BRANCH @ JP ;C \ Equal: Branch past the end of the loop.

\ Scr#3
\ META PRIMITIVES LOOP,+LOOP,I
CODE (LOOP)
  HERE-T 'LOOP !  \ Fill in the 'LOOP variable.
RPTR LDHL         \ Read return stack pointer into HL
  M C LD
   H INC
  M B LD          \ Read current index value.
   B INC          \ Increment it.
  B A LD
 80 XOR#
    C OR          \ Was it equal to 0x8000 ?
    Z IF
     5 B LDP#     \ If so, the real index has reached limit, terminate loop.
       B ADDP
     RPTR STHL     \ Increment ret stack pointer by 5 (1 increment already done)
                   \ and store updated ret stack pointer back.
    ELSE
```

```
       B M LD
        H DEC
       C M LD      \ Store updated index.
        H INC
        H INC
       M E LD
        H INC
       M D LD      \ Read loop start address into instruction pointer, repeat loop.
    THEN
    JPIX ;C

CODE (+LOOP) ( w --- )
   HERE-T '+LOOP ! \ Fill in '+LOOP variable.
 RPTR LDHL         \ Read return stack pointer into HL.
   M C LD
    H INC
   M B LD          \ Read Current index.
     EXSP          \ HL now contains w, the increment value.
    A AND
   B ADCP          \ Add increment to index.
   v if
                   \ If overflow, then boundary between limit-1 and limit is
                   \ crossed, terminate loop.
     H POP         \ Get return stack pointer.
  5 B LDP#
    B ADDP
 RPTR STHL     \ Increment ret stack pointer by 5 (1 increment already done)
               \ and store updated ret stack pointer back.
   else
    H B LD
    L C LD     \ Move updated index to BC.
     H POP     \ Get return stack pointer.
    B M LD
     H DEC
    C M LD     \ Store updated index.
     H INC
     H INC
    M E LD
     H INC
    M D LD     \ Read loop start address into instruction pointer, repeat loop.
   then
 JPIX ;C

CODE LEAVE
 RPTR LDHL     \ Read return stack pointer into HL.
    H INC
    H INC
   M E LD
    H INC
   M D LD      \ Get start address into DE.
    H INC
    H INC
    H INC
RPTR STHL             \ Write updated return stack pointer (6 was added).
    D DEC
    D DEC             \ DE (instruction pointer) now points to forward branch address
  'BRANCH @ JP ;C     \ continue into BRANCH.

CODE I  ( --- w)
  HERE-T 'I !         \ Fill in 'I variable.
```

```
RPTR LDHL                 \ Read return stack pointer into HL.
                          \ J jumps here.
  M C LD
   H INC
  M B LD                  \ Read current index. (which is (index-limit) xor 0x8000.
   H INC
   H INC
   H INC
  M A LD                  \ Read limit and add to index
   C ADD
  A C LD
   H INC
  M A LD
   B ADC
 80 XOR#                  \ and flip most significant bit, getting true index value.
  A B LD
  B PUSH                  \ Push result.
JPIX ;C

CODE J ( --- w)
  RPTR LDHL               \ Read return stack pointer into HL
  6 B LDP#
    B ADDP                \ Add 6 to it, to get to next inner loop parameters.
 'I @ 3 + JR ;C           \ Continue into I.

CODE I' ( --- w)
  RPTR LDHL               \ Read return stack pointer into HL
     H INC
     H INC
     H INC
     H INC
    M C LD
     H INC
    M B LD              \ Read limit value.
    B PUSH              \ Push result
    JPIX ;C

\ Scr#4
\ META PRIMITIVES FIND,WORD
CODE (FIND) ( addr1 addr2 --- addr3 n )
    EXX                 \ Use shadow register set, so original DE is preserved.
  D POP                 \ Get word list address, points to last defined name.
  BEGIN
    E A LD
      D OR
  NZ WHILE              \ Terminate loop if name field address is zero.
     H POP
    H PUSH              \ address of name to find.
    D PUSH              \ Save address of name in dictionary.
    D LDAP           \ Get length byte from name in dictionary.
   3F AND#           \ Mask off top 2 bits. Bit 5 is not masked, so a name
                     \ with 'smudge' bit set will not match.
      M CP           \ Is length byte equal?
      Z IF
      BEGIN
         D INC
         H INC
        D LDAP       \ DE points to name in dict, HL points to name to find.
          M CP       \ Compare names until bytes not equal.
      NZ UNTIL
     7F AND#      \ Check if last byte of name in dict with bit 7 masked off
                  \ is equal to corresponding byte in name to search.
     M CP
     Z IF
                  \ Names are equal, found.
         H POP    \ Get start address of name in dictionary.
        AF POP    \ Remove next stack item (address of name to find).
         D INC
        D PUSH    \ Push code field address (final name address + 1).
      1 D LDP#    \ Set DE to 1.
        M A LD
       40 AND#    \ Is word not immediate.
       Z IF
           D DEC
           D DEC  \ Change DE to -1.
       THEN
         D PUSH   \ Push the flag word.
         EXX      \ Go back to normal register set.
         JPIX     \ Next.
       THEN
     THEN
    D POP         \ Get start address of name in dictionary.
    D DEC
     EXDE
   M D LD
    H DEC
   M E LD         \ Access link field, just below name and read link.
 REPEAT
D PUSH            \ Push zero flag value.
   EXX            \ Go back to normal register set.
   JPIX ;C

CODE (WORD)  ( c addr1 --- addr2)
    EXX              \ Use shadow register set, so original DE is preserved.
  H POP              \ Input stream address in HL
  B POP              \ Delimiter character in C
 C A LD              \ Copy to A.
  H DEC
  BEGIN
      H INC
      M CP
  NZ UNTIL           \ Scan input stream until character unequal to delimiter is
                     \ found.
0 B LD#              \ B contains length byte, initialize to zero.
'DP D LDP
  D INC              \ Initialize DE to HERE+1.
  BEGIN
      M A LD         \ Read next character from input stream.
      A OR
      NZ IF
        C CP         \ Compare with delimiter if character is nonzero.
      THEN
  NZ WHILE           \ Terminate loop if character read is 0 (input stream ends)
                     \ or character is equal to delimiter.
      D STAP         \ Store character at destination.
       D INC
       H INC
       B INR         \ Increment source and destination pointers and byte count.
  REPEAT
20 A LD#
```

```
  D STAP           \ Store a blank space just after word in destination.
  M A LD
    A OR
    NZ IF
       H INC       \ Increment source pointer beyond delimiter, but only if
                   \ last character read from source is the delimiter and not
                   \ a zero byte.
    THEN
  H PUSH           \ Push input stream address.
'DP LDHL
  B M LD           \ Store length byte at HERE.
     EXX           \ Switch to normal register set.
    JPIX ;C

CODE R@  ( --- 16b)
 RPTR LDHL          \ Read return stack pointer into HL
    M C LD
     H INC
    M B LD          \ Read value into BC
    B PUSH          \ Push value.
      JPIX ;C

\ Scr#5
\ META PRIMITIVES DIGIT,>R,TCH
CODE DIGIT ( c --- u true  or  false)
   B POP            \ Get ASCII character, stored in C.
UPTR LDHL           \ Read user pointer to HL
  M B LD            \ Read BASE variable to B, stored at offset 0 in user area.
  C A LD            \ Move character to A.
0 H LDP#            \ Initialize HL to 0 (false flag).
```

```
 30 SUB#              \ Subtract ASCII '0'.
 NC IF
   0A CP#
   NC IF              \ Is digit value 10 or higher?
     7 SUB#           \ Subtract 'A' - '9', so character 'A' gets value 10.
     0A CP# CS
     IF               \ If value lower than 10, it was one of the characters between
                      \ '9' and '9'
       H PUSH         \ push false flag and exit.
       JPIX
     THEN
   THEN
 ELSE
   H PUSH             \ Lower then ASCII 0, no valid digit, push false flag and exit.
   JPIX
 THEN
 B CP
 NC IF                \ Digit value greater or equal to BASE?
   H PUSH             \ Then no valid digit, push false flag and exit.
   JPIX
 THEN
 H B LD
 A C LD               \ Move digit value to BC.
 H DEC
 B PUSH               \ Push Digit value.
 H PUSH               \ Push true flag.
 JPIX ;C

CODE >R  ( 16b --- )
    B POP             \ Pop value to be pushed on return stack.
```

```
RPTR LDHL            \ Read return stack pointer into HL
    H DEC
   B M LD
    H DEC
   C M LD            \ Store value, decrementing the pointer.
RPTR STHL            \ Write updated return stack pointer.
JPIX ;C

CODE R> ( --- 16b)
RPTR LDHL            \ Read return stack pointer into HL
   M C LD
    H INC
   M B LD
    H INC            \ Read value, incrementing the pointer.
RPTR STHL            \ Write updated return stack pointer.
   B PUSH            \ Push value.
   JPIX ;C

CODE TCH ( c --- )
    H POP
   L A LD            \ Pop character and move to A.
%Y FF 52 )LD#        \ Set SCR_CT to 255 to prevent 'Scroll?' message.
      PRT            \ Print value via RST 16 ROM call.
     JPIX ;C

CODE CHAN ( n --- )
    H POP            \ Get channel number.
   D PUSH            \ Save instruction pointer.
   L A LD
1601 CALL            \ Call ROM CHAN-OPEN routine to select output channel.
```

```
   D POP            \ Restore instruction pointer.
   JPIX ;C

\ Scr#6
\ META PRIMITIVES PKEY,PAUSE
CODE PKEY ( --- c)
5 %Y 1 )RES         \ Clear bit 5 of FLAGS, key press available.
  BEGIN
     5 %Y 1 )BIT     \ Test bit 5 of FLAGS, is key press now available?
                     \ Will be set by interrupt routine in ROM.
  Z WHILE            \ Call the WAIT routine while no key press.
                     \ All trickery within the WHILE loop is for calling a
                     \ FORTH word from assembler code and returning.
     'WAIT LDHL
     D PUSH          \ Save current instruction pointer.
     HERE 4 + D LDP# \ Set instruction pointer to XXX, NEXT will fetch word
                     \ at XXX
     JPHL            \ Execute WAIT routine.
    \ XXX
     HERE 2+ _       \ Word fetched by NEXT, will execute at YYY.
    \ YYY
     D POP           \ Restore instruction pointer.
   REPEAT
   5C08 LDA          \ Read LAST K variable, ASCII code of key pressed.
   HERE L1 !         \ Mark corrent address in L1, INKEY will jump here.

   \ The following compares and IFs will convert certain key codes
   \ yielded by just SYMBOL-SHIFT-KEY to ASCII codes that are normally obtained
   \ with EXTENDED-MODE SYMBOL-SHIT-KEY,
   \ so we don't have to use EXTENDED mode.
  C6 CP# Z IF 5B A LD# THEN \ [
  C5 CP# Z IF 5D A LD# THEN \ ]
  E2 CP# Z IF 7E A LD# THEN \ ~
  C3 CP# Z IF 7C A LD# THEN \ |
  CD CP# Z IF 5C A LD# THEN \ \ backslash.
  CC CP# Z IF 7B A LD# THEN  \ {
  CB CP# Z IF 7D A LD# THEN  \ }
  C7 CP# Z IF 1  A LD# THEN  \ Convert '<=' to special Previous screen code.
  C9 CP# Z IF 2  A LD# THEN  \ Convert '<>' to special HOME code.
  C8 CP# Z IF 3  A LD# THEN  \ Convert '>=' to special Next screen code.
 A L LD
0 H LD#                      \ Move key code to HL
 H PUSH                      \ and push.
%X JPIX ;C

CODE INKEY  ( --- c )
 \ Same sequence of KEY-SCAN, K-TEST and K-DECODE calls is used in BASIC INKEY$
 \ handler.
 D PUSH                      \ Save instruction pointer.
 28E CALL                    \ Call KEY-SCAN in ROM.
 Z IF                        \ If key press valid?
   31E CALL                  \ Call K-TEST in ROM.
   CS IF                     \ Valid code if carry set.
     0 C LD#
       D DER
      A E LD
    333 CALL                 \ Call K-DECODE in ROM.
       D POP                 \ Restore instruction pointer.
     L1 @ JR                 \ Continue into PKEY routine to process key code in A.
   THEN
  THEN
     D POP                    \ No valid key pressed. Restore instruction pointer.
  0 H LDP#
    H PUSH Â            \ Push zero value.
    JPIX ;C

CODE PAUSE ( u --- )
    B POP               \ Pop number of ticks.
    BEGIN
       B PUSH           \ Save number of ticks,
       D PUSH           \ Save instruction pointer.
       \ Use the same trick to call FORTH from assembler as was used in PKEY.
   'WAIT LDHL
   HERE 4 + D LDP#      \ Setup instruction pointer to XXX.
         JPHL           \ Execute WAIT function.
   \ XXXX
   HERE 2+ _            \ Gets fetched by NEXT, causes execution of code to resume
                        \ at YYY.
   \ YYY
       D POP
       B POP            \ Restore instruction pointer and number of ticks,
       HALT             \ Wait for next timer interrupt.
      B DEC             \ Decrement number of ticks.
     B A LD
      C OR              \ Test for zero.
   Z UNTIL
   JPIX ;C

CODE NOOP ( --- )
 JPIX ;C
```

```
\ Scr#7
\ META PRMITIVES MTYPE,CMOVE
CODE MTYPE ( addr u ---)
FF %Y 52 )LD#     \ Set SCR_CT variable to 255 to prevemt "Scroll?" message.
    D PUSH        \ Save instruction pointer
   2 A LD#
 1601 CALL        \ Select Channel 2, screen output.
     D POP        \ Restore instruction pointer.
     B POP
     H POP        \ Address in HL, count in BC
     BEGIN
       B A LD
       C OR       \ As long as count nonzero.
     NZ  WHILE
       M A LD
       PRT        \ Print next character.
       B DEC
       H INC      \ Decrement count, icrement pointer.
     REPEAT
     JPIX ;C

CODE CMOVE ( addr1 addr2 u ---)
    EXX           \ Use the shadow registers.
  B POP
  D POP
  H POP           \ Get the operands.
  B A LD
  C OR
  NZ IF           \ Skip if u is zero.
```

```
    LDIR          \ Z80 does block move in one instruction.
  THEN
  EXX             \ Back to normal registers.
 JPIX ;C

CODE CMOVE>  ( addr1 addr2 u ---)
    EXX           \ Use the shadow registers.
  B POP
  D POP
  H POP           \ Get the operands.
 B A LD
 C OR
 NZ IF            \ Skip if u is zero.
   B ADDP
   H  DEC         \ Add u-1 to source address.
     EXDE
   B ADDP
    H DEC         \ Add u-1 to destination address.
     EXDE
     LDDR         \ Single instruction for block move in reverse order.
 THEN
 EXX              \ Back to normal registers.
 JPIX ;C

CODE FILL ( addr u 8b ---)
    EXX           \ Use shadow registers.
  D POP
  B POP
  H POP           \ Get the operands.
  BEGIN
```

```
    B A LD
    C OR
  NZ WHILE        \ While count is not yet zero.
    E M LD        \ Store byte
    H INC         \ increment address
    B DEC         \ decrement count.
  REPEAT
    EXX           \ Back to normal registers.
   JPIX ;C

CODE SP@  ( --- addr)
 0 H LDP#
  SP ADDP         \ Get stack pointer to HL
   H PUSH         \ Push it.
     JPIX ;C

CODE SP!  ( --- )
  'S0 SP LDP      \ Load stack pointer from initial variable.
  JPIX ;C

CODE RP@  ( --- addr)
 RPTR LDHL
   H PUSH         \ Push return stack pointer.
   JPIX ;C

CODE RP!  ( --- addr)
 'R0 LDHL
RPTR STHL           \ Load return stack pointer from initial variable.
     JPIX
   SMUDGE           \ Ugly! An extra ;C later on will reverse the effect of
```

```
                    \ this SMUDGE.
;C

\ 16x16 to 32 bit multiplication subroutine.
\ A is loop counter.
\ Operand 1 in DE (will be shifted out to the left).
\ Operand 2 in BC (will be added to DE:HL)
\ Result in DE (msw) and HL (lsw)
\ During operation, result bits shift into DE from the right,
ASSEMBLER HERE L1 ! \ L1 is label for this subroutine.
   0 H LDP#
   10 A LD#         \ Iterate for 16 bits.
   BEGIN
     H ADDP
       E RL
       D RL         \ Shift entire result 1 bit to the left, shift
                    \ next bit of operand1 out onto carry.
       CS IF        \ If operand1 bit was set.
        B ADDP      \ Add operand2 bit into result.
         CS IF
           D INC    \ If carry, add 1 to msw of result.
         THEN
       THEN
      A DER
   Z UNTIL
   RET
( MULTIPLY)

\ Scr#8
\ META PRIMITIVES UM*,M*,UM/MOD
\ 32 / 16 bit unsigned division subroutine. 16-bit result and remainder.
\ A is a loop counter.
\ 32-bit dividend is in HL (msw) and DE (lsw)
\ During operation, dividend will shift to the left and divisor is
\ trial-subtracted from HL. Quotient bits will shift in from the right.
\ divisor in BC (will be subtracted from HL)
\ Result: HL is remainder, DE is quotient.
HERE L2 !           \ L2 is label for this subroutine.
   10 A LD#         \ Iterate for 16 bits.
   BEGIN
      E SLA
       D RL
     H ADCP         \ Shift dividend 1 bit to the left
      CS IF         \ If most significant bit of dividend was 1, do the
                    \ subtraction anyway.
         B SUBP     \ Subtract divisor from most significant word.
      ELSE
         B SBCP     \ Trial subtraction.
         CS IF
           B ADDP   \ Add back if subtraction result is negative.
            D DEC   \ Compensate the effect of later D INC, so
                    \ last significant quotient bit will not be set.
         THEN
      THEN
     D INC          \ Set least significant bit of quotient if subtraction
                    \ succeeded.
     A DER
  Z UNTIL
     RET
```

```
( DIVIDE)

!CSP  ;C             \ Finish code definition. Side effect is toggling the SMUDGE bit
                     \ of LATEST definition, RP! in this case.

CODE UM* ( u1 u2 --- ud)
     EXX             \ Use shadow registers.
   B POP
   D POP             \ Get operands
L1 @ CALL            \ Do the multiplication.
  H PUSH
  D PUSH             \ Push result
     EXX             \ Back to normal registers.
    JPIX ;C

CODE UM/MOD ( ud u1 --- u2 u3 )
     EXX             \ Use the shadow registers.
   B POP             \ Divisor into BC.
   H POP
   D POP             \ Dividend into HL:DE
  L A LD
   C SUB
  H A LD
   B SBC             \ If HL >= BC, then division will overflow.
   NC IF
     -1 H LDP#       \ Division overflows, Set quotient and remainder both
        L E LD       \ to 0ffffh.
        H D LD
   ELSE
      L2 @ CALL      \ Do the division.
    THEN
   H PUSH
   D PUSH            \ Push results.
      EXX            \ Back to normal registers
 JPIX ;C

CODE M* ( n1 n2 --- d)
      EXX            \ Use shadow registers.
 0 H LDP#
    B POP            \ Pop operand 2.
  7 B BIT
    NZ IF
      A XOR          \ If negative, negate operand2
      C SUB
     A C LD
     H A LD
      B SBC
     A B LD
      H DER          \ H is now -1, to indicate negative operand.
    THEN
    D POP            \ Pop operand 1.
  7 D BIT
    NZ IF
      A XOR          \ If negative, negate operand1.
      E SUB
     A E LD
     L A LD
      D SBC
     A D LD
      L DER          \ L is now -1, to indicate negative operand.
    THEN
   H A LD
    L XOR           \ Exor the signs, sign of result.
     EXAF           \ Save result sign.
L1 @ CALL           \ Do unsigned multiplication.
     EXAF
     A OR
    NZ IF           \ If result sign is negative.
      A XOR         \ Negate and push result.
      L SUB
     A L LD
    0 A LD#
      H SBC
     A H LD
     H PUSH
   0 H LDP#
     D SBCP
     H PUSH
   ELSE
     H PUSH         \ No negation, just push the result.
     D PUSH
   THEN
     EXX            \ Back to normal registers.
    JPIX ;C

\ OVER ended up here, probably because I forgot to include this word
\ when I wrote the stack words. This screen still had free space.
CODE OVER ( 16b1 16b2 --- 16b1 16b2 16b1)
   B POP
   H POP            \ Pop two numbers.
   H PUSH
   B PUSH          \ Push them back
   H PUSH          \ plus the second number.
   JPIX ;C

\ Scr#9
\ META PRIMITIVES M/,AND,OR,XOR
\ Floored division is long and hairy.
CODE M/ ( d n1 --- n2 n3)
    EXX            \ Use shadow registers.
   B POP           \ Divisor in BC
   H POP
   D POP           \ Dividend in HL (msw) and DE (lsw)
   B PUSH
   D PUSH          \ Push divisor and dividend (lsw)
 0 D LDP#          \ DE will be used to record operand signs.
  7 B BIT          \ Is divisor negative?
   NZ IF
       A XOR       \ Negate divisor.
       C SUB
      A C LD
      D A LD
       B SBC
      A B LD
       E DER       \ E is now -1 to indicate negative divisor
    THEN
  7 H BIT          \ Is dividend negative?
    EXSP           \ Get dividend LSW into HL
   NZ IF
      A XOR
    L SUB
   A L LD
   D A LD
    H SBC
   A H LD        \ Negate it
     EXSP        \ Get dividend MSW into HL, store LSW on stack.
   D A LD
    L SBC
   A L LD
   D A LD
    H SBC
   A H LD          \ Negate that too.
    D DER          \ D is now -1 to indicate negative dividend.
     EXSP          \ Swap back,
THEN
EXDE
EXSP               \ Dividend MSW in HL, LSW in DE, operand signs on stack.
                   \ Both operands are now nonnegative.
L A LD
 C SUB
H A LD
 B SBC             \ If HL >= BC, then division will overflow.
 NC IF
  \ 0 H LDP        \ Suggested fix for bug, insert thsese two instructions
  \    EXSP        \ clear operand signs on stack.
  -1 H LDP#        \ Set both quotient and remainder to -1.
    L E LD         \ BUG: if operands are negative, we still negate the
                   \  result(s) and correct for  floored division.
    H D LD
 ELSE
```

```
   L2 @ CALL       \ Divde the numbers
  THEN
  B POP            \ Get signs from stack.
 B A LD
  A AND            \ Test dividend sign.
  NZ IF
     A XOR         \ Negate remainder if dividend is negative.
     L SUB
    A L LD
   0 A LD#
     H SBC
   A H LD
  THEN
 B A LD
  C XOR
  NZ IF            \ If operands have different signs.
     A XOR         \ Negate quotient.
     E SUB
    A E LD
   0 A LD#
     D SBC
   A D LD
  THEN
  H A LD
    L OR
   NZ IF           \ If remainder is nonzero.
     B A LD
      C XOR
      NZ IF        \ And operands have differnt signs.
        B POP
```

```
          B PUSH
          B ADDP      \ Add original divisor to remainder
           D DEC      \ decrement quotient, required for floored division.
       THEN
      THEN
    B POP             \ Remove original divisor
    H PUSH            \ Push remainder
    D PUSH            \ Push quotient
      EXX             \ Back to normal registers.
      JPIX ;C

CODE STOPON ( --- )
      IM2             \ Switch to Interrupt mode 2, so our own interrupt handler
                      \ with the BREAK key test is used.
     JPIX ;C
CODE STOPOFF ( --- )
      IM1             \ Switch to Interrupt mode 1, so the ROM interrupt handler
                      \ is used.
     JPIX ;C

CODE XOR ( 16b1 16b2 --- 16b3)
   B POP
   H POP              \ Get operands
  C A LD
   L XOR
  A L LD
  B A LD
   H XOR              \ XOR them.
  A H LD
  H PUSH              \ Push result
```

```
    JPIX ;C

CODE AND ( 16b1 16b2 --- 16b3)
   B POP
   H POP             \ Get operands
  C A LD
   L AND
  A L LD
  B A LD
   H AND             \ AND them.
  A H LD
  H PUSH             \ Push result
    JPIX ;C

CODE OR ( 16b1 16b2 --- 16b3)
   B POP
   H POP             \ Get operands.
  C A LD
    L OR
  A L LD
  B A LD
    H OR             \ OR them.
  A H LD
  H PUSH             \ Push result.
    JPIX ;C

\ Scr#10
\ META PRIMITIVES ARITHMETIC
CODE 0=  ( n1 --- f)
   H POP
  H A LD
    L OR              \ Check that operand is equal to 0.
0 B LDP#              \ Prepare false flag in BC
    Z IF
      B DEC           \ Change to true -1 is operand was 0.
    THEN
  B PUSH              \ Push result.
    JPIX ;C

CODE 0< ( n1 --- f)
  AF POP             \ Only need to check most significant byte (in A).
    RLCA             \ Shift most significant bit into carry.
 0 A LD#             \ A=0
  0 SBC#             \ If carry set, change it to 0ffh
  A L LD
  A H LD             \ Flag in HL
  H PUSH             \ Push result.
    JPIX ;C

CODE <  ( n1 n2 --- f)
   B POP
   H POP             \ Get operands.
   A XOR             \ A contains flag=0, clear carry.
  B SBCP             \ Subtract operands
    m if
      FF A LD#       \ If subtraction is negative, set flag to 0ffh
    then
    v if
      CPL            \ If overflow, complement flag in A.
    then             \ sign flag XOR overflow flag is proper result for
```

```
                     \ signed compare, works throughout integer range.
  A L LD
  A H LD             \ Flag in HL
  H PUSH             \ Push result.
    JPIX ;C

CODE + ( w1 w2 --- w3)
   B POP
   H POP             \ Get operands.
  B ADDP             \ Add
  H PUSH             \ Push result.
    JPIX ;C

CODE - ( w1 w2 --- w3)
   B POP
   H POP             \ Get operands
  B SUBP             \ Subtract
  H PUSH             \ Push result
    JPIX ;C

CODE NEGATE ( n1 --- n2)
   B POP             \ Get operand
   A XOR             \ A=0, clear carry.
  A L LD
  A H LD             \ HL=0
  B SBCP             \ Subtract BC from 0
  H PUSH             \ Push result.
    JPIX ;C

CODE D+ ( wd1 wd2 --- wd3 )
```

```
     EXX          \ Use shadow register set.
   B POP
   D POP          \ wd2 in BC (msw) and DE (lsw)
   H POP          \ get msw of wd1
    EXSP          \ swap with lsw of wd1 on stack.
  D ADDP          \ Add least significant words.
    EXSP          \ Store result on stack, get msw of wd1 in HL
  B ADCP          \ Add most significant words
  H PUSH          \ Push result.
     EXX          \ Back to normal registers.
    JPIX ;C

CODE D- ( wd1 wd2 --- wd3 )
     EXX          \ Use shadow register set.
   B POP
   D POP          \ wd2 in BC (msw) and DE (lsw)
   H POP          \ get msw of wd1
    EXSP          \ swap with lsw of wd1 on stack.
  D SUBP          \ Subtract least significant words.
    EXSP          \ Store result on stack, get msw of wd1 in HL
  B SBCP          \ Subtract most significant words
  H PUSH          \ Push result.
     EXX          \ Back to normal registers.
    JPIX ;C

CODE DNEGATE ( d1 --- d2)
   H POP
   B POP          \ Get number in HL (msw) and BC (lsw)
  H PUSH          \ Push msw.
0 H LDP#
```

```
  B SUBP         \ subtract least sigificant word from 0.
   B POP         \ Get msw of input
  H PUSH         \ Push lsw of result
0 H LDP#
  B SBCP         \ subtract most significant word from 0.
  H PUSH         \ push msw of result.
    JPIX ;C

CODE U< ( u1 u2 --- f)
   B POP
   H POP         \ Get operands
   A XOR         \ A=0 (prepare flag), clear carry.
  B SBCP         \ Subtract operands.
  0 SBC#         \ Subtract carry of subtraction from A, A=0ffh if u1<u2
  A L LD
  A H LD         \ Flag to HL
  H PUSH         \ Push
JPIX ;C

\ Most stack operations are very straightforward to implement.
CODE DROP ( 16b --- )
   H POP
    JPIX ;C

CODE DUP ( 16b --- 16b 16b)
   H POP
  H PUSH
  H PUSH
    JPIX ;C
DECIMAL FORTH DEFINITIONS
```

```
\ FILE META3
\ Scr#1
\ META PRIMITIVES STACK
META DEFINITIONS HEX
CODE SWAP ( 16b1 16b2 --- 16b2 16b1)
   H POP        \ Get 16b2
    EXSP        \ Swap it with 16b1 on stack.
  H PUSH        \ Push 16b1
    JPIX ;C

CODE ROT ( 16b1 16b2 16b3 --- 16b2 16b3 16b1)
   B POP
   H POP
    EXSP        \ Swap 16b2 in HL with 16b1 on stack.
  B PUSH
  H PUSH
    JPIX ;C

CODE -ROT ( 16b1 16b2 16b3 --- 16b3 16b1 16b2)
   H POP
   B POP
    EXSP
  H PUSH
  B PUSH
    JPIX ;C

CODE UNDER  ( 16b1 16b2 --- 16b2) \ is known as NIP in some FORTH systems.
   H POP
   B POP
  H PUSH
    JPIX ;C

CODE MIR ( 16b1 16b2 16b3 --- 16b3 16b2  16b1)
   H POP
   B POP
    EXSP
  B PUSH
  H PUSH
    JPIX ;C

CODE 2DROP ( 32b ---)
   H POP
   H POP
    JPIX ;C

CODE 2DUP ( 32b --- 32b 32b)
   B POP
   H POP
  H PUSH
  B PUSH
  H PUSH
  B PUSH
    JPIX ;C

CODE PICK ( u --- 16b)
   H POP
  H ADDP              \ Double the operand, word offset to byte offset.
 SP ADDP              \ Add to stack pointer
  M C LD
```

```
     H INC
    M B LD                \ Read the desired number from stack.
    B PUSH
      JPIX ;C

CODE ROLL ( u ---)
       EXX                \ Use shadow registers.
      EXSP                \ Operand now in HL
     H INC
    H ADDP                \ Add 1 and multiply by 2.
    H B LD
    L C LD                \ Byte count in BC = 2*(u+1)
   SP ADDP                \ Add to SP, Address of cell to pick up and move to top.
    M E LD
     H INC
    M D LD                \ Read the cell that must be moved to top.
    D PUSH                \ Save it.
    L E LD
    H D LD
     H DEC
     H DEC                \ Source address is destination address - 2.
    B A LD
      C OR
     NZ IF                \ Test for byte count zero is unnecessary BC=2 even for 0 ROLL.
         LDDR             \ Move the remaining stack cells up.
     THEN
     H POP                \ Get saved top.
     B POP                \ Remove junk cell.
      EXSP                \ Put top back.
       EXX                \ Back to normal registers
    JPIX ;C

CODE 2* ( w1 --- w2)
   H POP
  H ADDP
  H PUSH
    JPIX ;C

CODE 2/  ( n1 --- n2)
   H POP
   H SRA              \ Signed division by 2, so arithmetic shift
    L RR
  H PUSH
    JPIX ;C

\ Scr#2
\ META PRIMITIVES @,!,1+
CODE @ ( addr --- 16b)
   H POP
  M C LD
   H INC
  M B LD
  B PUSH
    JPIX ;C

CODE C@ ( addr --- 8b)
   H POP
  M C LD
 0 B LD#
  B PUSH
```

```
    JPIX ;C

CODE 2@ ( addr --- 32b)
   H POP
   H INC
   H INC
  M C LD        \ Read word at high address first.
   H INC
  M B LD
  B PUSH
   H DEC
   H DEC
  M B LD        \ Read word at low address last.
   H DEC
  M C LD
  B PUSH
    JPIX ;C

CODE ! ( 16b addr --- )
   H POP
   B POP
  C M LD
   H INC
  B M LD
    JPIX ;C

CODE C! ( 8b addr --- )
   H POP
   B POP
  C M LD
```

```
    JPIX ;C

CODE 2! ( 32b addr --- )
   H POP
   B POP
  C M LD
   H INC
  B M LD
   H INC
   B POP
  C M LD
   H INC
  B M LD
    JPIX ;C

CODE +! ( n addr ----)
   H POP
   B POP
  M A LD
   C ADD
  A M LD
   H INC
  M A LD
   B ADC
  A M LD
    JPIX ;C

CODE 1+ ( w1 --- w2)
   H POP
   H INC
  H PUSH
    JPIX ;C

CODE 2+ ( w1 --- w2)
   H POP
   H INC
   H INC
  H PUSH
    JPIX ;C

CODE D< ( d1 d2 --- f)
     EXX          \ Use shadow registers.
   B POP
   D POP
   H POP
   A XOR
  B SBCP
   H POP
  D SBCP          \ BUG!!! subtraction of MSW and LSW is done in wrong order!
    v if
      A DER       \ Change flag to 0ffh if overflow
    then
    m if
      CPL         \ Complement if sign flag, same trick as with < but
                  \ tests are in reverse order.
    then
  A L LD
  A H LD          \ Flag to HL and push it.
  H PUSH
     EXX          \ Back to normal registers.
    JPIX ;C

\ FIX: The part before "v if" should be replaced with this.
\ It adds one EXSP instruction and swaps the order of D SBCP and B SBCP.
\     EXX         \ Use shadow register set.
\   B POP
\   D POP         \ d2 in BC (msw) and DE (lsw)
\   H POP         \ get msw ofwd1
\    EXSP         \ swap with lsw of d1 on stack.
\   A XOR         \ Set flag to zero, clear carry.
\  D SBCP         \ Subtract least significant words.
\   H POP         \ get msw of d1 in HL
\  B SBCP         \ Subtract most significant words


\ Scr#3
\ META PRIMITIVES 1-,P!,BCAL
CODE 1- ( w1 --- w2)
   H POP
   H DEC
  H PUSH
    JPIX ;C

CODE 2- ( w1 --- w2)
   H POP
   H DEC
   H DEC
  H PUSH
    JPIX ;C
```

```
\ The IN/OUT (C) instructions are not in the assembler, so put
\ their opcodes directly.
CODE P@  ( addr --- 8b)
   B POP
   ED C_ 68 C_   \ IN L,(C) instruction, reads from port BC.
0 H LD#          \ Make high byte zero.
 H PUSH
   JPIX ;C

CODE P! ( addr 8b ---)
  B POP
  H POP
  ED C_ 69 C_   \ OUT (C),L instruction, writes to port BC.
  JPIX ;C

CODE 2SWAP ( 32b1 32b2 --- 32b2 32b1)
    EXX
  D POP
  H POP
  B POP
   EXSP
 D PUSH
 H PUSH
 B PUSH
    EXX
   JPIX ;C

\ 2OVER is the only word that in the base system
\ that uses both normal and shadow registers for its operation.
CODE 2OVER ( 32b1 32b2 ---- 32b1 32b2 32b1)
```

```
    EXX
  D POP
  H POP         \ Get 32b2 in DE' and HL'
    EXX
  B POP
  H POP            \ Get 32b1 in BC and HL
 H PUSH
 B PUSH            \ Push it back.
    EXX
 H PUSH
 D PUSH            \ Push 32b2 back.
    EXX
 H PUSH            \ Push 32b1 back once more.
 B PUSH
   JPIX ;C

CODE BCAL ( n --- )
ORIGIN 3 + B LDP#
     H POP         \ Get Line number in HL
    D PUSH         \ Save instruction pointer.
5C3D D LDP
    D PUSH         \ Save old value of ERR_SP.
    B PUSH         \ Push WARM entry point on stack, errors in BASIC
                   \ will execute WARM.
5C3D SP STP        \ Save current stack pointer into ERR_SP.
 5C42 STHL         \ Store line number in NEWPPC, so BASIC will execute
                   \ this line next.
     A XOR
  5C44 STA         \ Set NS_PPC to 0, so BASIC will use first statement in line.
  5C89 LDA         \ Read S_POSN line number.
```

```
     3 CP#
0DAF cs  CALC      \ Clear the screen if we are on the lowest 2 screen lines.
                   \ We can't be on lowest 2 lines if DF_SZ is set to 2.
%Y 2 31 )LD#           \ Set DF_SZ to 2 as is required in BASIC.
 1B76 JP ;C            \ Jump to STMT_RET in ROM, will execute next BASIC statement.
ASSEMBLER

\ BCAL return code. BASIC program will USR 27036 to get to BCAL return point.
HERE-T ORIGIN 7 + !-T  \ Patch jump address at BCAL return entry point.
 5C3D SP LDP           \ Load ERR_SP
       B POP           \ Remove error return address, discarded.
       H POP           \ Get old value of ERR_SP.
   5C3D STHL           \ Restore old value of ERR_SP.
       D POP           \ Restore instruction pointer.
%Y 0 31 )LD#           \ Set DF_SZ to 0, so we can use all 24 screen lines.
NEXT %X XH LDP#        \ Set IX to NEXT address.
        JPIX

\ Scr#4
\ META USER VARIABLES,EMIT,KEY
META DEFINITIONS
CODE BYE ( --- )
5C3D SP LDP            \ Load stack pointer from ERR_SP.
      B POP            \ Remove old error return address.
1303 B LDP#
     B PUSH            \ Push BASIC entry point in ROM onto stack.
                       \ so the next error will return to BASIC command line.
        IM1            \ Interrupt mode 1, normal situation.
%Y 2 31 )LD# %X        \ Set DF_SZ to 2, as it must be in BASIC.
      8 RST FF C_      \ RST-8 followed by byte FFh, generate OK 'error'.
```

```
;C

META DEFINITIONS
\ Now define most user variables and constants of the target system.
\ Some items allocated in the user area are not true user variables,
\ because they do not change per task. Their addresses are defined
\ as constants.

 0 USER BASE
699F 2 O+ CONSTANT DP
 4 USER (KEY)              \ Address of the KEY routine.
 6 USER (EMIT)             \ Address of the EMIT routine.
 \ Position 8 in the user area contains return stack pointer.
699F 0A O+ CONSTANT (WAIT) \ Address of routine to wit in PKEY and PAUSE.
699F 0C O+ CONSTANT FENCE  \ Address below which dictionary is not cleared.
0E USER S0                 \ Initial stack pointer.
10 USER R0                 \ Initial return stack pointer.
12 USER VOC-LINK           \ Linked list of all vocabularies.
14 USER HLD                \ Address to store digit during numeric conversion.
16 USER DPL                \ Decimal point location.
18 USER OUT                \ Output column number.
1A USER >IN                \ position in input stream/
1C USER SCR                \ Screen last listed.
1E USER BLK                \ block number of input stream.
20 USER CURRENT            \ vocabulary to add words to.
22 USER CONTEXT            \ Vocabulary first searched.
24 USER SPAN               \ Number of characters read with EXPECT.
26 USER #TIB               \ Number of characters in Terminal Input Buffer.
2E USER WIDTH              \ Max. number of characters stored in new name.
```

```
\ One code definition thrown in here.
CODE CLS ( --- )
    D PUSH              \ Save instruction pointer.
 0DAF CALL              \ Call CL-ALL routine in ROM to clear the screen.
   2 A LD#
 1601 CALL              \ Select channel 2, screen output.
     D POP              \ Restore instruction pointer.
     JPIX ;C

\ The usual constants in the target system.
  0 CONSTANT 0
 20 CONSTANT BL         \ Blank space.
  1 CONSTANT 1
400 CONSTANT B/BUF      \ Bytes per buffer
  2 CONSTANT 2
 01 CONSTANT B/SCR      \ Buffers per screen.
  3 CONSTANT 3
 -1 CONSTANT -1
DECIMAL
\ Constant addresses of items below FORTH image. See section 4.3 of User Manual.
26003 CONSTANT FIRST    \ Start address of single block buffer.
                        \ Two bytes below is the block number (0 if no block).
                        \ The byte at address 26000 is unused, reserved for
                        \ update flag.
27027 CONSTANT LIMIT    \ Address beyond that buffer.
25600 CONSTANT TIB      \ Address of terminal input buffer.
HEX

\ Only now do we start META-INTERPRET, this is our first colon definition
\ to compile in the target system.
```

```
META-INTERPRET
: EMIT ( c --- )
  (EMIT) @ EXECUTE
  1 OUT +! ;              \ The OUT user variable stores the column position.

: HERE ( --- addr)
  DP @ ;

: CR ( --- )
  0D EMIT
  0 OUT ! ;

: KEY  ( --- c)
  (KEY)  @ EXECUTE ;

: TYPE ( addr u --- )
  0 ?DO
    DUP C@ EMIT 1+
  LOOP DROP ;

\ Scr#5
\ META COUNT,ABORT,ARITHMETIC

: COUNT ( addr1 --- addr2 n)
  DUP 1+ SWAP C@ ;

: (.") ( --- )
  [ FORTH HERE-T 3 - '." ! META ]    \ Fill in the address of the ." runtime part
                                     \ in META compiler.
  R>                                    \ Get return address, where string is stored,
  COUNT 2DUP +                          \ Update return address.
  >R                                    \ Put updated address back.
  TYPE  ;                               \ Type the string.

: ABORT ( --- )
  \ Start executing at WARM entry point.
  [ FORTH ORIGIN 3 + META ] LITERAL EXECUTE ;

: (ABORT") ( f --- )
  [ FORTH HERE-T 3 - 'ABORT ! META ] \ Fill in the address of the ABORT" runtime
                                        \ part in META compiler.
  R> COUNT                              \ Access string at return address.
  ROT IF
     TYPE BL EMIT                       \ Type the error message.
     HERE COUNT TYPE                    \ Type the word last read.
     ABORT                              \ Never return from ABORT
  ELSE
     + >R                               \ Update return address.
  THEN ;

: ?DUP ( 16b --- 16b 16b or 16b)
  DUP IF DUP THEN ;

: 2ROT ( 32b1 32b2 32b3  ---- 32b2 32b3 32b1)
  5 ROLL 5 ROLL ;

: S->D ( n --- d)
  \ Sign-extend the single precision number, if n<0 then add 0xffff else 0
  DUP 0< ;
```

```
: 0> ( n --- f)
  NEGATE 0< ;

: > ( n1 n2 --- f)
  SWAP < ;

: =  ( 16b1 16b2 --- f)
  - 0= ;

: MIN ( n1 n2 --- n3)
  2DUP > IF SWAP THEN DROP ;

: MAX ( n1 n2 --- n3)
  2DUP < IF SWAP THEN DROP ;

: NOT ( 16b1 --- 16b2) \ Bitwise invert, will also invert
                       \ proper flags 0xffff <-> 0x0000
  -1 XOR ;

: +- ( n1 n2 --- n3)
  0< IF NEGATE THEN ;

: ABS ( n --- u)
  DUP +- ;

: D0= ( wd --- f)
  OR 0= ;

: D0< ( d --- f)
```

```
  UNDER 0< ;              \ Only need to check MSW.

: D+- ( d1 --- d2)
  0< IF DNEGATE THEN ;

: DABS ( d --- ud)
  DUP D+- ;

: * ( w1 w2 --- w3)
  UM* DROP ;

: /MOD ( n1 n2 --- n3 n4)
  SWAP S->D ROT           \ Convert n1 to double number.
  M/ ;

: / ( n1 n2 --- n3)
  /MOD UNDER ;

: MOD ( n1 n2 --- n3)
  /MOD DROP ;

\ Scr#6
\ META EXPECT,BANK,ADDR

CODE TOGGLE ( addr 8b --- )
   B POP
   H POP
  C A LD
   M XOR
  A M LD
```

```
    JPIX ;C

: BS ( --- )
  08 EMIT ;

: SPACE ( --- )
  BL EMIT ;

: SPACES ( u ---)
  0 ?DO SPACE LOOP ;

: CAP ( ---)
  5C6A 8 TOGGLE ; Flip bit 8 in FLAGS2 system variable.

\ Expect is one of the most complex FORTH words.
: EXPECT ( addr u ---)
  DUP IF \ Ignore if u=0
    SWAP 0 SPAN !
    BEGIN
       12 EMIT 1 EMIT \ Set FLASHING.
       4C             \ Capital L
       5C6A C@ 8 AND IF 9 - THEN \ If CAPS bit is set, change it into C.
       EMIT           \ Emit the cursor character, flashing L or C.
       12 EMIT 0 EMIT \ Set FLASHING off.
       BS             \ Move cursor position one back.
       KEY            \ Read key.
       DUP 1F > OVER  80 < AND  \ Is it a printable character?
       IF
         DUP EMIT     \ Display it.
         OVER C!      \ Store it.
```

```
        1+            \ Update address.
        1 SPAN +!     \ Increment SPAN.
      ELSE
        DUP 0D = IF  \ is it ENTER?
           DROP 2DROP SPACE EXIT \ Remove stuff from stack and exit.
        ELSE
           DUP 0C = SPAN @ AND   \ Is it DELETE and do we already have
                                 \ characters in the line?
           IF
             -1 SPAN +!          \ Decrement number of characters read.
             SPACE BS BS         \ Overwrite cursor with space, two backspaces.
             DROP                \ Remove DEL charcter from stack
             1-                  \ Decrement address.
           ELSE DUP 2 = IF       \ Is SYM-SHIFT-W pressed?
             BYE
           ELSE
             6 = IF CAP THEN   \ Is CAPS-LOCK pressed, then flip state.
           THEN
        THEN
       THEN
     THEN
     OVER SPAN @ =             \ Exit if number of characters equals max.
   UNTIL
 THEN SPACE                    \ Overwrite cursor with space.
 2DROP ;

: ERASE ( addr u ---)
  0 FILL ;

: BLANK ( addr u ---)
```

```
  BL FILL ;

\ BANKSWITCH SPECTRUM 128
\ Map one of the RAM banks to the address range 0xc000-0xffff.
\ 0 is the normal memory bank used by BASIC.
\ 1-5 are converted to 1,3,4,6 and 7. Banks 2 and 5 are mapped
\ to 0x4000-0xBfff, don't use these for RAM disk.
\ Operation has no effect on Spectrum 48.
: BANK ( n --- )
  DUP 1 > IF 1+ THEN     \ Convert range 0..5 to 0, 1, 3, 4, 6 and 7.
  DUP 4 > IF 1+ THEN
  10 +                   \ Add 0x10 to select normal ROM.
  7FFD P! ;

   VARIABLE LO           \ Hold base address of RAM disk, 0 on Spectrum 128
0A CONSTANT #B           \ Number of screens in RAM disk.

: #SCR ( --- n)
  LO @ IF
     #B                  \ On the 48, number of screens is defined by #B.
  ELSE
     50                  \ On the 128 we have 80 screens.
  THEN ;

\ ADDR yields the address of any given screen in the RAM disk
\ and (on Spectrum 128) switches to the correct bank.
\ Words that directly use this function have to reside below 0xC000.
: ADDR ( n1 --- addr)
  1-                     \ Screen numbers start at 1.
  DUP #SCR U< 0= ABORT" Out of ramdisk" \ Check this is a valid screen number.
  LO @ IF
     \ Spectrum 48
     B/BUF * LO @ +      \ Add to address in LO variable.
  ELSE
     \ Spectrum 128
     10 /MOD             \ Compute bank number and screen within bank.
     1+ BANK             \ Select bank.
     B/BUF * C000 +      \ Add to 0xc000, start of upper 16kB.
  THEN ;

\ Scr#7
\ META SAVE-B,EMPTY-B,BLOCK,WORD

\ This FORTH system has one block buffer, containing a block from
\ the RAM disk. Buffer handling functions are very simple.
\ The word at FIRST - 2 is the block number of the block stored in the
\ buffer, 0 if there is none.

: SAVE-BUFFERS ( ---)            \ Will always copy the buffer to RAM disk, whether
                                 \ it was updated or not.
  FIRST 2- @ ?DUP                \ Do we have a buffer?
  IF
    ADDR FIRST SWAP B/BUF CMOVE \ Copy to RAM disk.
    0 BANK                       \ Select normal memory bank.
  THEN ;

: EMPTY-BUFFERS ( ---)           \ Mark buffer as empty.
  0 FIRST 2- ! ;

: UPDATE ( --- )                 \ Do nothing, SAVE-BUFFERS will always save.
```

```
  ;

: BUFFER ( n --- addr)
  SAVE-BUFFERS
  FIRST 2- !                      \ Mark block buffer number.
  FIRST B/BUF BLANK               \ Fill buffer with spaces.
  FIRST ;

: FLUSH  ( ---)
  SAVE-BUFFERS EMPTY-BUFFERS ;

: BLOCK ( n --- addr)
  DUP FIRST 2- @ -           \ Does the current block buffer contain a different block
                             \ from the one requested?
  IF
    SAVE-BUFFERS             \ Save buffer to RAM disk.
    DUP ADDR FIRST B/BUF CMOVE \ Copy data from RAM disk to buffer.
    FIRST 2- !               \ Mark the block number stored.
    0 BANK                   \ Switch to normal memory bank.
  ELSE
     DROP                    \ Drop block number.
  THEN
  FIRST                      \ Return the start address of the block buffer.
  0 LIMIT C! ;               \ Set byte at LIMIT to 0.

: WORD ( c --- addr)
  0 BANK                     \ Make sure normal RAM is selected.
  BLK @ ?DUP
  IF                         \ If BLK contains nonzero
    BLOCK                    \ Read from block.
  ELSE
    TIB                     \ else read from TIB.
  THEN
  SWAP OVER >IN @ +         \ Add contents of >IN to address.
  (WORD)                    \ Do the hard work in a code definition.
  SWAP - >IN !              \ Subtract input start address and tore updated offset
                            \ into >IN
  HERE ;                    \ Return address where word is stored.

: FIND ( addr1 --- addr2 n)
  CONTEXT @ @ (FIND)        \ Search context vocabulary first.
  CONTEXT @ CURRENT @ -     \ Is CONTEXT different from CURRENT?
  IF
    ?DUP 0= IF              \ If last find returned 0 flag?
      CURRENT @ @ (FIND)    \ Search CURRENT.
    THEN
  THEN ;

: ' ( --- addr)
  BL WORD FIND              \ Read word and find.
  0= ABORT" Not found" ;    \ Complain if not found.

: ALLOT ( n --- )
  HERE +                    \ Add to current dictionary pointer.
  LO @ 1- OVER 80 + U<      \ Check that we have enough space below LO.
      ABORT" Dictionary full"
  DUP FENCE @ U<            \ Check that we are not below FENCE.
      ABORT" Protected dictionary"
  STOPOFF DP ! STOPON ;     \ Prevent break key interrupt while updating DP.
```

```
: PAD ( --- addr)
  HERE 38 + ;

\ Scr#8
\ META ERRORS NUMERIC CONV

\ Some more user variables.
28 USER STATE          \ Outer interpreter state, nonzero for compiling.
2A USER CSP            \ Stack pointer check for compiler.

20 CONSTANT C/L        \ Characters per line (in screen), standard would be 64.

\ The following  words are used internally by the compiler for error checking.
: ?PAIRS ( n1 n2 --- )
  - ABORT" Wrong structure" ;
: ?COMP ( --- )
  STATE @ 0= ABORT" Not compiling" ;
: ?EXEC ( --- )
  STATE @ ABORT" Not executing" ;
: !CSP ( --- )         \ Store stack pointer at start of definition.
  SP@ CSP ! ;
: ?CSP ( --- )         \ Check that the stack pointer is the same as at start.
  SP@ CSP @ - ABORT" Incomplete structure" ;
: ?LOADING  ( ---)
  BLK @ 0= ABORT" Not loading" ;
: ?STACK  ( --- )
  SP@ S0 @ 1+ U< 0= ABORT" Stack empty"
  SP@
  5C65 @ 10 +         \ Read STKEND BASIC system variable, still require 16
                      \ bytes free above that.
  U< ABORT" Stack full" ;

\ This word was used to make the ZX-Printer usable on a Spectrum 128.
\ The Spectrum 128 redirects printer output to the serial port and abuses.
\ the printer buffer for other purposes.
\ The ROM must be switched to 48k mode by clearing bit 4 in FLAGS.
\ This is done in COLD.
: ZX-PRINT
  5B00 100 ERASE       \ Erase printer buffer.
  09F4 5C4F @ 0F + ! ; \ Put original ROM entry point in the CHANS table.

: */MOD ( n1 n2 n3 --- n4 n5)
   >R                  \ Save divisor on return stack.
   M*
   R>  M/ ;

: */  ( n1 n2 n3 ---)
  */MOD UNDER ;        \ Discard modulus.

\ This word is used for numeric conversion, 32 by 16 division giving 32-bit
\ quotient. Implemented in many FORTHs since FIG-Forth.
: M/MOD
  >R                   \ Save divisor on return stack.
  0 R@ UM/MOD          \ Zero-extend MSW of dividend and divide MSW.
  R>                   \ Get divisor back.
  SWAP
  >R                   \ Save quotient MSW on return stack.
  UM/MOD               \ Divide MSWRemainder: LSWDividend by divisor.
  R> ;                 \ GET MSW of quotient.
```

```
\ The following words are for numeric conversion. They have been implemented
\ the same way in many FORTHs since FIG-Forth.
\
\ The output string is produced from rightmost digit to leftmost digit
\ in a memory area just below PAD, ad decreasing addresses.
\
\ At the start of numeric conversion <#, a double number is on the stack.
\ The word # extracts the rightmost digit and puts it into the string.
\ The word #> discards the double number (which is now supposed to be zero)
\ And returns the string.

: HOLD ( c --- )
  -1 HLD +!            \ Decrement character address.
   HLD @ C! ;          \ Store the character.

: <#  ( ud --- ud)
  PAD HLD ! ;

: #> ( ud --- addr c)
  2DROP                \ Discard number being converted.
  HLD @ PAD OVER - ;   \ Return result string.

: # ( ud1 --- ud2)
  BASE @ M/MOD         \ Divide by BASE.
  ROT                  \ Put modulus on top, this is the extracted digit value.
                       \ Double number quotient remains on the stack.
  DUP 9 >
  IF                   \ If digit is above 9, add 'A'-'9', so digit value 10.
                       \ will become A, etc.
    7 +
  THEN
  30 + HOLD ;             \ Add ASCII '0', to convert digit to ASCII.

: #S ( ud --- 0 0)
  BEGIN # 2DUP D0= UNTIL ;

: SIGN ( n --- )
  0< IF 2D HOLD THEN ;    \ Add minus sign to string if n is negative.

\ Scr#9
\ META NUMERIC CONV,MASS STORE
: D.R ( d n --- )
  >R                      \ Save field width on stack.
  SWAP OVER               ( d.msw d.lsw d.msw) \ Keep sign in original msw.
  DABS                    \ Take absolute value.
  <# #S                   \ Convert number to string.
   ROT SIGN               \ Add - if original msw was negative.
   #>
  R> OVER - 0 MAX SPACES \ Print spaces (field width - string length)
                          \ but do not print a negative number of spaces.
  TYPE ;                  \ Print the converted string.

: .R ( n1 n2 --- )
  >R S->D R> D.R ;

: D. ( d --- )
  0 D.R SPACE ;           \ If field width in D.R is 0, no spaces are printed in front.

: . ( n ---)
  S->D D. ;
```

```
: U. ( u ---)
  0 D. ;

: ? ( addr --- )
  @ . ;
: HEX ( --- )
  10 BASE ! ;

: DECIMAL ( --- )
  0A BASE ! ;

: H. ( u --- )
  BASE @            \ Save and restore original base.
  SWAP HEX 0 <# # # # # #> TYPE SPACE
  BASE ! ;

\ These words handle file I/O. They call BASIC routines to do the
\ work for us.

\ GETFN stores three items into BASIC variables, which were pre-initialized
\ by the BASIC program. The same set of variables must be initialized
\ in the same order, otherwise GETFN will no longer work.
\ The offsets with respect to VARS are hard-coded in GETFN.
\
\ A name is read from the input string and stored into A$ (A$ was initialized
\ to length 10 before).
\
\ Two integers are stored in variables I and J.
\ Numeric variables in BASIC are normally floating point, but Spectrum BASIC
\ represents numbers in the range -65535...+65535 as integers in
\ an FP number with exponent byte set to zero. Therefore we can just store
\ the integer at the right offset.

: GETFN ( n1 n2 ---)
  5C4B @                      \ Read VARS.
  DUP 6 + DUP 0A BLANK        \ Clear the A$ variable.
  BL WORD COUNT ROT SWAP CMOVE \ Copy word from input string into A$
                              \ We should have checked word length, we will
                              \ silently clobber the variables if name is too long.
  DUP 19 + ROT SWAP !         \ Store J
  13 + ! ;                    \ Store I.

: DELETE ( --- )              \ Delete a file with the given name.
  0 0 GETFN 28 BCAL ;         \ BASIC line 40.

: CAT    ( --- )              \ Show a list of files.
  2D BCAL ;                   \ BASIC line 45.

: PUT ( n1 n2 --- )
  FLUSH                       \ Synchronize RAM disk with buffer.
  OVER - 1+ B/BUF *           \ Compute file length.
  SWAP DUP ADDR               \ Compute base address ( n1 addr size ---)
  ROT GETFN                 \ Read name into A$, set I and J variables.
  ADDR  DROP                \ Select the correct bank again, was undone by GETFN
  32 BCAL                   \ BASIC line 50.
  0 BANK  ;                 \ Back to normal memory bank.

: GET ( n1 --- )
  FLUSH                     \ Make sure block buffer is empty.
  DUP ADDR 0 GETFN       \ Read name into A$ Address to I, J set to 0 (unused).
  ADDR  DROP             \ Select the correct bank again, was undone by GETFN
  37 BCAL                \ BASIC line 55
  0 BANK ;               \ Back to normal memory bank.

: FORMAT                 \ Clear all buffers in RAM disk.
  #SCR 1+ 1 DO
     I BUFFER DROP
  LOOP ;

: INDEX ( n1 n2 ---)
  1+ SWAP DO
    CR  I 2 .R \ Print   \ Print block number in field of 2 characters.
    I BLOCK 1E TYPE      \ Print top line, only 30 characters of it.
  LOOP ;

\ Note: we use LIT TCH instead of ['] TCH which was a common FORTH idiom.
\ The compiler does not even know it is a literal, it just lays out the
\ compilation address for LIT followed by that of TCH.
: >P ( ---)
  LIT TCH (EMIT) !       \ Set default handler for EMIT.
  3 CHAN ;                \ Select output channel 3, is printer.

: >S ( ---)
  LIT TCH (EMIT) !        \ Set default handler for EMIT.
  2 CHAN ;                \ Select output channel 2, is screen.

: TERMINAL ( --- )
  LIT PKEY (KEY) !        \ Set default handler for KEY.
  >S ;                    \ And initialize screen output.
```

```
\ Scr#10
\ META CONVERT,NUMBER,LITERAL

: ?TERMINAL ( --- f)      \ Check if EDIT (CAPS-SHIT-1) was pressed.
  INKEY 7 = ;


\ The following words are for converting ASCII numbers to binary form,
\ used by the interpreter of the target system.

: CONVERT ( ud1 addr1 --- ud2 addr2)
  BEGIN
     1+                   \ Increment address.
     DUP >R               \ Save it on the return stack
     C@ DIGIT             \ Read a character and convert it to digit value.
  WHILE
     SWAP BASE @ UM* DROP \ Multiply number MSW by BASE. (digitval msw*base)
     ROT BASE @ UM* D+          \ Multiply number LSW by base and add it.
                                \ Result is double number with value ud1*BASE+digitval.
     DPL @ 1+ IF 1 DPL +! THEN \ Increments DPL if it was set earlier..
     R>                         \ Get saved address.
  REPEAT R> ;

2C USER 'ERRNUM                 \ Handler invoked when NUMBER finds an error.
                                \ Allows number to be extended with other
                                \ parsing code, e.g. for floating point.
: NUMBER ( addr --- wd)
  0 0                           \ Start with a double of value 0.
  ROT DUP 1+ C@                  \ Read character at addr+1
  DUP 26 = IF                    \ Is it equal to ASCII '&'?
       DROP 2+ C@                \ Read character following the '&' return that.
       UNDER UNDER 0             \ Remove double value, zero extend char to double.
      -1 DPL !                   \ Set DPL to -1, indicating single number.
  ELSE
      2D = DUP >R                \ Compare character with '-', save to return stack.
       -                         \ Also use the flag to adjust the address
                                 \ (if there was a minus sign, skip to next character.
      -1                         \ Initialize DPL to -1.
      BEGIN
         DPL !                   \ Store DPL value (-1 or 0).
         CONVERT                 \ Convert a string of digits into binary.
         DUP C@ BL -             \ Is the next character a blank space?
      WHILE
         DUP C@ 2E -             \ Compare the next character with '.'
         'ERRNUM @ EXECUTE       \ Error if not equal.
         0                    \ Initialize DPL to 0.
      REPEAT
      DROP                    \ drop address.
      R> IF DNEGATE THEN      \ negate the number if saved sign
  THEN ;

: (ERRNUM) ( f ---)           \ If NUMBER can't convert the word after
                              \ the interpreter could not find it, then abort.
  ABORT" Can't find" ;

\ The following words are part of the compiler on the target system.
: , ( 16b ---)
  HERE ! 2 ALLOT ;
```

```
: C, ( 8b ---)
  HERE C! 1 ALLOT ;

: COMPILE ( --- )
  ?COMP                        \ Check we are compiling.
  R> DUP @                     \ Read word at return address (immediately following
                               \ COMPILE in the colon definition where itt was
                               \ compiled.
  ,                            \ Compile the compilation address to the new def.
  2+ >R ;                      \ Update return address so we skip the next word.

: LITERAL ( 16b --- nothing or 16b)
  STATE @
  IF                           \ If we are compiling
    COMPILE LIT                \ Compile the LIT primitive.
    ,                          \ Put the literal value after it.
  THEN ; IMMEDIATE

: DLITERAL (32b --- nothing or 32b)
  STATE @ IF
    SWAP
    [COMPILE] LITERAL          \ Put LSW first.
    [COMPILE] LITERAL          \ Put MSW last.
  THEN ; IMMEDIATE

: LATEST ( --- addr)
  CURRENT @ @ ;

: SMUDGE ( --- )
```

```
  LATEST 20 TOGGLE ;         \ Set bit 5 in name field of latest def to make it
                             \ findable or unfindable.

: ['] ( --- )
  ' [COMPILE] LITERAL
 ; IMMEDIATE
FORTH DEFINITIONS DECIMAL EXIT \ Exit the META interpreter at end of file.


\ FILE META4
\ Scr#1
\ META CREATE AND BUILDING WORDS
HEX META DEFINITIONS META-INTERPRET \ Start it again for the next file.

: >BODY ( addr1 --- addr2)
  3 + ;                 \ Skip code field.

\ Defining words in the compiler on the target system.

\ All defining words on the target system will call CREATE, which
\ adds headers.
\
\ Note that CREATE uses the WIDTH  variable, so it can store names with
\ fewer characters than the original length. Lengths must still match, but
\ characters beyond the stored length are not checked. An old trick to save
\ space.
\
\ The trick was used occasionally to create a word with one signicicant name
\ character, for instance $XXXX (WIDTH temporarily set to 1) with length 5.
\ Now any word with length 5 and starting with $ will match that name, such as
\ $0000 or $FFFF. When the word is executed it can read characters from
\ the word buffer at HERE and use these, for instance to parse a hex number
\ and put it on the stack/compile it as a literal.
\
: CREATE ( --- )
  LATEST ,                           \ Add link field.
  BL WORD                            \ Read next word from input stream.
  DUP C@ 0= ABORT" Name expected"    \ Abort if no word read.
  DUP C@ WIDTH @ MIN 1+ ALLOT        \ Allocate space for the name.
                                     \ Note that the name was read at HERE,
                                    \ so it is already in its final location.
  HERE 1- 80 TOGGLE  \ Set bit 7 in final character of name.
  STOPOFF            \ No BREAK key while updating current.
  CURRENT @ !        \ Add new definition to CURRENT vocabulary.
  CD C,              \ Add a CALL instruction.
  [ NEXT ] LITERAL , \ to the NEXT address, (, will call ALLOT will call STOPON)
  LATEST 80 TOGGLE ; \ Set bit 7 of first name character.

: VARIABLE ( --- )
  CREATE 2 ALLOT ;

: CONSTANT ( 16b ---)
  CREATE
  [ DOCON ] LITERAL  HERE 2- ! \ Store the DOCON address in the code field.
  , ;                          \ Add the value to the parameter field.

: USER ( n ---)
  CREATE
  [ DOUSER ] LITERAL HERE 2- ! \ Store the DOUSER address in the code field.
  C, ;                         \ Add the variable offset to the parameter field.
```

```
\ Traverse is used by >NAME and NAME> to find the opposite end of the
\ name in a header.
: TRAVERSE ( addr1 n --- addr2)
  SWAP
  BEGIN
    OVER +                      \ Update address by adding n (either 1 or -1)
    7F OVER  C@ <               \ Read character and check it is at least 80h.
  UNTIL
  UNDER ;

: >NAME ( addr1 --- addr2)
  1- -1 TRAVERSE ;

: NAME> ( addr1 --- addr2)
  1 TRAVERSE 1+ ;

: (;CODE)  ( --- )
  [ FORTH HERE-T 3 - 'CODE ! META ]  \ Fill in the 'CODE address of the
                                     \ meta compiler.
  R>                \ Remove return address.
                    \ This is not put back, so (;CODE) will return from
                    \ the containing colon definition.
  LATEST NAME> 1+ ! \ Put the return address in the code field of the
                    \ latest definition.
                    \ The latest definition will now CALL to the CALL DOCOL
                    \ in the DOES> part.
                    \ The first call puts the parameter field on stack, the
                    \ DOCOL will execute the DOES> part as a colon definition.
;
```

```
: DOES>  ( --- )
  COMPILE (;CODE)            \ Add (;CODE) to the defining word, causing it to exit.
                             \ and change the code field in the newly defined word,
                             \ so it will execute the DOES> part.
  CD C, [ DOCOL ] LITERAL ,  \ Add a call to DOCOL to the defining word,
                             \ just after (;CODE).
; IMMEDIATE

: [ ( --- )
  0 STATE ! ; IMMEDIATE

: ] ( --- )
  0BF STATE ! ; \ Note the value 0BF was used traditionally in Fig-FORTH
                \ so the inner interpreter could compare the length byte
                \ of the name with the contents of STATE
                \ In an IMMEDIATE word the length byte would be greater than
                \ 0BF, so the word would always execute.
                \ In a non-immediate word the length byte would be less than
                \ 0BF, but greater than 0, so it would execute if state was 0.
               \ On this system, the VALUE of STATE is NOT used in this way.
: : ( ---)
  ?EXEC
  CREATE SMUDGE \ Make the word unfindable, temporarily until ;.
  [ DOCOL } LITERAL HERE 2- ! \ Store the DOCAL address in the code field.
  \ Note that we have to use } here instead of ]  as ] is now redefined.
  \ And unlike many other compiler words, ] is not immediate.
  !CSP
  ] ;
  CURRENT @ CONTEXT !
```

```
; IMMEDIATE

: ; ( --- )
  ?COMP
  ?CSP
  COMPILE EXIT
  SMUDGE
  [COMPILE] [
; IMMEDIATE

\ Scr#2
\ META OUTER INTERPRETER,STRUCT
: QUERY ( --- )
  TERMINAL                  \ Make sure wa use screen and keyboard.
  TIB 80 EXPECT             \ Read the line.
  0 TIB SPAN @ + C!         \ Add a trailing null byte.
  0 >IN !
  0 BLK !
  SPAN @ #TIB ! ;

: INTERPRET ( --- )
  BEGIN
     BL WORD DUP C@         \ Read next word.
  WHILE                     \ Continue the loop as long as words are available.
    FIND DUP
    IF
      0< STATE @ AND IF     \ Word is found, execute it or compile it.
          ,
      ELSE
          EXECUTE
      THEN
    ELSE
      DROP NUMBER           \ Not found, parse it as a number.
      DPL @ 1+ IF           \ If number contains decimal point.
        [COMPILE] DLITERAL
      ELSE
        DROP [COMPILE] LITERAL
      THEN
    THEN
    ?STACK
  REPEAT DROP ;

 : LOAD ( n ---)
   >IN @ >R BLK @ >R       \ Save old input stream on return stack.
   BLK ! 0 >IN !
   INTERPRET
   R> BLK ! R> >IN ! ;     \ Restore old input stream.

: --> ( ---)
  ?LOADING
  1 BLK +! 0 >IN !
; IMMEDIATE

: (  ( ---)
  29 WORD DROP             \ Read until next ')' character, discard it.
; IMMEDIATE

: \ ( ---)
  ?LOADING
  >IN @
  C/L NEGATE AND C/L +      \ Adjust >IN to start of next line.
  >IN ! ; IMMEDIATE

\ Control structure words on the target system.

: >MARK  ( --- addr)
  ?COMP
  HERE                      \ Put current address on stack for forward reference.
  0 ,                       \ Add extra cell for forward branch address.
;

: >RESOLVE ( addr ---)
  ?COMP
  HERE SWAP !               \ Fill in forward branch address with current location.
;

: <MARK ( --- addr)
  HERE                      \ Put current location on stack as backwards branch address.
  ?COMP
;

: <RESOLVE  ( addr ---)
  ?COMP
  ,                         \ Add add backward branch address.
;

: DO ( --- addr 3)
  COMPILE (DO)              \ Compile runtime part
  >MARK 3                   \ Mark forward ref and push 3 on stack for consistency
check.
```

```
; IMMEDIATE

: ?DO ( --- addr 3)
  COMPILE (?DO)          \ Compile runtime part
  >MARK 3                \ Mark forward ref and push 3 on stack for consistency check.
; IMMEDIATE

: LOOP ( addr 3 --- )
  3 ?PAIRS               \ Do consistency check.
  COMPILE (LOOP)         \ Compile runtime part.
  >RESOLVE               \ resolve forward reference from DO.
; IMMEDIATE

: +LOOP
  3 ?PAIRS               \ Do consistency check.
  COMPILE (+LOOP)        \ Compile runtime part.
  >RESOLVE               \ resolve forward reference from DO.
; IMMEDIATE

\ Scr#3
\ META COMPILING WORDS,QUIT
: BEGIN ( --- addr 1)
  <MARK 1                \ Mark backward ref and push 1 for consistency check.
; IMMEDIATE

: UNTIL ( addr 1 ---)
  1 ?PAIRS
  COMPILE ?BRANCH        \ Compile conditional branch.
  <RESOLVE               \ Resolve backward ref to BEGIN.
; IMMEDIATE
```

```
: WHILE ( addr1 1 --- addr1 addr2 4)
  1 ?PAIRS
  COMPILE ?BRANCH       \ Compile forward branch.
  >MARK 4               \ Mark forward reference and push 4 for consistency check.
; IMMEDIATE

: REPEAT ( addr1 addr2 4 ---)
  4 ?PAIRS
  SWAP
  COMPILE BRANCH        \ Compile unconditional branch
  <RESOLVE              \ fill in backward reference, back to begin.
  >RESOLVE              \ Resolve forward reference, starting at WHILE
; IMMEDIATE

: IF ( --- addr 2)
  COMPILE ?BRANCH       \ Compile conditional branch.
  >MARK 2               \ Mark forward reference and push 2 for consistency check.
; IMMEDIATE

: ELSE ( addr1 2 --- addr2 2)
  2 ?PAIRS
  COMPILE BRANCH        \ Compile onconditional branch, so IF part skips ELSE part.
  >MARK                 \ Mark forward reference for this branch.
  SWAP
  >RESOLVE              \ resolve forward reference originating in IF.
                        \ if conditional branch at IF is taken, jump to ELSE part.
  2                     \ Push 2 for consistency check
; IMMEDIATE
```

```
\ THEN will terminate either IF or IF..ELSE
: THEN ( addr 2 --- )
  2 ?PAIRS
  >RESOLVE                \ Resolve forward reference, either from IF or from ELSE.
; IMMEDIATE

: ."  ( ---)
  COMPILE (.")            \ Compile runtime part.
  22 WORD                 \ Get text delimited by next "
  C@ 1+ ALLOT             \ Allocate space for it. String does not need to be moved, it
                          \ is already at HERE.
; IMMEDIATE

: .( ( ---)
  29 WORD COUNT TYPE      \ Type text delimited by )
; IMMEDIATE

: ABORT"
  COMPILE (ABORT")        \ Compile runtime part.
  22 WORD                 \ Get text delimited by next "
  C@ 1+ ALLOT             \ Allocate space for it. String does not need to be moved, it
                          \ is already at HERE.
; IMMEDIATE

: [COMPILE] ( --- )
  ' ,                     \ Find the word and compile its compilation address.
; IMMEDIATE

: QUIT ( --- )
  RP!
  [COMPILE] [             \ Go to interpretation state.
  BEGIN
    CR QUERY INTERPRET
    STATE @ 0=
      IF ." Ok"
    THEN
  0 UNTIL ;

: DRIVE ( d ---)
  DUP 5C4B @              \ Read VARS system variable, start of BASIC variables.
  1F + C!                 \ Store in D BASIC variable. Offset depends on variables
                          \ having been assigned in a particular order.
  5CB0 C! ;               \ Store it also at 27028, the 'unused' system variable.

\ Scr#4
\ META CLEAR,COPY,RUN
: CLEAR ( n --- )
  BUFFER DROP ;           \ BUFFER clears the buffer.

: COPY ( n1 n2 ---)
  SWAP BLOCK              \ Source screen in block.
  SWAP ADDR               \ Destination is RAM disk, in any bank.
  B/BUF                   \ Copy one blck.
  0 BANK ;                \ Select normal bank.

: RUN
  1 GET 1 LOAD ;          \ Get a file and load from it.

\ Scr#5
\ META INTERRUPT,FORTH-83,VOC
```

```
FORTH DEFINITIONS HEX

\ Create an entiire 256-byte page + 1 byte (257 bytes) all filled with
\ the same byte. The I register will point to this page. When in
\ Interrupt mode 2, the Z80 will read an interrupt vector from this page
\ where the least significant 8 bits of the address are read from the bus.
\
\ Unfortunately no hardware provides the vector address. Normally 0ffh
\ is read from the bus, but you can't be sure. Therefore the same word
\ will be read from anywhere in the page, as the page is filled with the
\ same byte. This will be the adress of the interrupt handler.

HERE-T FF + FF00 AND            \ Go to the next page-aligned address.
DUP 101 + DP-T !                \ Add 257 to it, end of the 257-byte area to fill.
DP-T 1+ C@ DUP 100 * + DP-T !
\ Set DP-T to address where both bytes are the same.
\ Example: Filled page starts at 0x8400 and will be filled with all 0x85 bytes.
\ The interrupt handler will now start at 0x8585.

DUP CONSTANT INTREG             \ Set INTREG to start of page.
OFFSET + 101 HERE-T FF AND FILL \ Fill the page (257 bytes) with the byte.
ASSEMBLER ( INTERRUPT)

\ Interrupt handler starts here.
     38 RST          \ Call the standard ROM interrupt handler, keyboard scan
                     \ and FRAMES increment.
   AF PUSH           \ Save AF
                     \ The following it similar to the BREAK-KEY routine in ROM.
                     \ Do a WARM start if break key pressed.
  7F A LD#
```

```
     FE IN                    \ Read Lower right keys,
       RRA
     NC IF                    \ Rightmost bit cleared, so SPACE pressed.
         FE A LD#
            FE IN             \ Read lower left keys.
              RRA
         NC IF                \ Rightmost bit cleared, so CAPS-SHIFT pressed too.
           0 LDA              \ Read the first ROM byte.
           F3 CP#             \ Check that it's equal to F3h, standard 48k BASIC
                              \ ROM paged in.
                              \ Do not break when one of the other ROMs is paged in.
           Z IF
             14 A LD#
             5C3A STA         \ Store ERR_NR, error code 20 for BREAK.
             ORIGIN 3 + JP    \ Do a WARM start.
           THEN
         THEN
     THEN
   AF POP
      RET                     \ Return from interrupt.

META DEFINITIONS
: FORTH-83 ( --- )
  CR ." Forth-83 Standard System"
  CR ." 1988 L.C. Benschop"
  CR ." Thanks to Coos Haak"
  CR LO @ HERE - 80 - .       \ Compute number of free bytes.
  ." Bytes free, "
  #SCR . ." Screens " ;
```

```
: VOCABULARY
  CREATE              \ Create mew word.
  HERE 6 + ,          \ cell 1, contains latest def in this vocabulary,
                      \          initialized to second dummy name field.
  A081 ,              \ cell 2, first dummy name field, forms dummy
                      \          header with cell 1 as link field, will be
                      \          silently traversed by FIND.
  CURRENT @ 2+ ,      \ cell 3 contains pointer to cell 2 in
                      \         old CURRENT vocabulary,
  A081 ,              \ cell 4  second dummy name field, forms dummy
                      \         header with cell 3 as link field, will be
                      \         silently traversed by FIND.
  HERE VOC-LINK @ , VOC-LINK ! \ Link the new vocabulary in VOC-LINK list.
                               \ cell 5 is link to previous vocabulary/
  [ FORTH HERE-T VL ! META }   \ Mark target address of DOES> part in VL.
  DOES>
    CONTEXT !         \ Set context VOCABULARY
;
\ The two dummy headers (link field + name field) will cause FIND
\ to continue searching from one vocabulary into the next.
\
\ The name field in a dummy header is bytes 81h 0A0h, name of length 1
\ containing a single space. This will never be matched by the compiler
\ but FIND will move on to the next header in the chain.
\
\ When FIND traverses a new vocabulary, it will start at the pointer in
\ cell 1 (CONTEXT @ @). All words will be traversed of this vocabulary,
\ ending in the second dummy header. The link field of the second
\ dummy header points to the first dummy header in the old CURRENT
\ vocabulary, so the old current vocabulary will now be searched in its
\ entirety.

: DEFINITIONS ( --- )
  CONTEXT @ CURRENT ! ;


\ Scr#6
\ META FORTH,FORGET,WARM


FORTH
\ We cannot use our freshly made defining word VOCABULARY in the meta compiler
\ to create the FORTH vocabulary. We have to build it by hand instead.
  CREATE-T FORTH      \ Create header of FORTH vocabulary.

  ORIGIN 9 + UPTR !-T \ Initialize the User Pointer. Ugly!  Why here?

  HERE-T ,            \ Store TARGET compilation address in HOST definition.
                      \ The host word has no DOES> part,
                      \ but [COMPILE] FORTH will work just fine.
  CD C,-T VL @ 2+ ,-T \ Create code field, CALL to DOES> part of VOCABULARY.
  HERE-T VL !         \ Mark address to patch with latest definition.
  0 ,-T               \ cell 1 latest def in this vocabulary, will be
                      \        patched at end of meta compilation.
  A081 ,-T            \ cell 2 First dummy name field
  0 ,-T               \ cell 3 link to old CURRENT, null for FORTH.
  A081 ,-T            \ cell 4 Second dummy name field.
  HERE-T UPTR @-T 12 + !-T 0 ,-T  \ Set VOC-LINK to HERE-T, store
                                  \ null pointer in cell 5.
```

```
META DEF IMMEDIATE

: (FORGET) ( addr --- )
   DUP FENCE @ U<
     ABORT" Protected dictionary" \ Check that we don't forget below FENCE
  STOPOFF                         \ Disable BREAK key while operation in progress.
  VOC-LINK @                      \ Traverse all vocabularies.
  BEGIN
    ?DUP                          \ While more vocabularies in list.
  WHILE
    2DUP U< IF
      @ DUP VOC-LINK !            \ If vocabulary above forget address, unlink it
                                  \ from VOC-LINK list, discard vocabulary completely.
    ELSE
      DUP 8 -
       BEGIN
         @ 2- DUP 3 PICK U<       \ Traverse vocabulary until below forget point
       UNTIL
      2+ OVER 8 - !               \ Update link pointer in vocabulary.
      @                           \ Go to next vocabulary in VOC-LINK list.
    THEN
  REPEAT
  DP !                            \ Update dictionary pointer.
  STOPON ;

: FORGET ( --- )
  '                               \ Find address of word to forget.
  CONTEXT @ OVER U<
  IF
    [COMPILE]                     \ Select FORTH vocabulary if CONTEXT is below forget
```

```
                     \ address, so it will be removed completely.
  THEN
  CURRENT @ OVER U< IF
    DEFINITIONS          \ Make sure CURRENT is not removed completely.
  THEN
  >NAME 2- (FORGET) ;

: WARM
  STOPOFF
  RP! SP! DECIMAL
  LIT NOOP  (WAIT) !
  [COMPILE] FORTH  DEFINITIONS
  BEGIN
   INKEY
  0= UNTIL              \ Wait until no key pressed.
  FF 5C3A C!            \ Set ERR_NR to OK, clear error number.
  STOPON
  0  BANK
  BLK @ ?DUP            \ If we were loading from a screen, push BLK and >IN
                        \ to stack, so WHERE can return to editor.
  IF
   >IN @
  THEN
  1F WIDTH !            \ Set WIDTH to 31, store up to 31 characters for each name.
  QUIT ;

: DEPTH ( --- u )
  S0 @  SP@ -  2/  1- ;
```

```
\ Scr#7
\ META COLD,.S,VLIST,DUMP
: COLD
  STOPOFF
  LIMIT 0C + LIMIT 1+ !      \ Initialize User Area pointer.
  TERMINAL
  DECIMAL CLS
  ED 5C3B !                  \ Clear bit 4 of FLAGS, Causes 128k Spectrum ROM to
                             \ behave like 48k BASIC.
                             \ ! instead of C! so it clears TV-FLAGS too, unintentional
  0 BANK  -1 C@              \ Get old byte at address -1 in bank 0.
  0 -1 C!                    \ Store 0 at address -1.
  1 BANK -1 C@               \ Get old byte at address -1 in bank 1.
  1 -1 C!                    \ Store 1 at address -1.
  0 BANK -1 C@               \ Read byte at address -1 in bank 0.
  IF
                             \ If it was unequal to 0, bank switching did not work
                              \ Assume we are Spectrum 48.
    ." 48"
    #B B/BUF * NEGATE LO !   \ Set LO variable to start of RAM disk.
  ELSE
    ." 128" 0 LO !           \ Set LO to 0, RAM disk will be in different banks on 128.
  THEN
  1 BANK  -1 C!
  0 BANK -1 C!               \ Restore the bytes at -1 in banks 1 and 0.
    ." K Spectrum"
  LIT (ERRNUM) 'ERRNUM !
  RP!
  FENCE @ (FORGET)
  FORTH-83
  EMPTY-BUFFERS
  8 5C6A C!               \ Set bit 3 in FLAGS2, set CAPS-LOCK.
  WARM ;

: .S  ( --- )
  DEPTH
  IF
   DEPTH 0 DO
     DEPTH I - 1- PICK .
   LOOP
  ELSE
   ." Empty "
  THEN ;

: STYPE ( addr u ---)
  \ Like TYPE, but mask off bit 7 and print control chars as .
  0 ?DO
    DUP C@               \ Read byte
    7F AND               \ Mask off bit 7.
    DUP BL < IF DROP 2E THEN \ Control chars replaced by .
    EMIT
    1+                   \ increment address.
  LOOP DROP ;

: ID.  ( addr --- )
  \ Print word name from header, addr is name field address.
  DUP NAME> OVER - 1-    \ Compute name length.
  SWAP 1+  SWAP          \ Increment start address, do not type length byte.
  STYPE
;
```

```
: VLIST
  CONTEXT @ @
  BEGIN
    DUP                 \ As long as address is not zero.
    ?TERMINAL 0= AND    \ And EDIT key is not pressed.
  WHILE
    6 EMIT              \ Go to next TAB position on screen.
    DUP  H. SPACE       \ Print name field in hex.
    DUP ID. SPACE       \ Print word's name.
    2- @                \ Follow link to previous word in list.
  REPEAT DROP ;

CODE ><  ( 16b1 --- 16b2 )
  H POP
  H A LD
  L H LD
  A L LD                                \ Swap H and L registers.
  H PUSH
  JPIX ;C

: DUMP  ( addr u --- (
  7 + -8 AND                            \ Round size up to next multiple of 8 bytes.
  8 / 0 ?DO                             \ Print rows of 8 bytes.
     CR DUP H.                          \ Print current address.
     8 0 DO I OVER + @ >< H. 2 +LOOP    \ Print 4 byteswapped words in hex.
     DUP BS 8 STYPE                     \ Type as 8-byte ASCII
     ?TERMINAL IF LEAVE THEN            \ Check for EDIT key to stop.
     8 +                                \ Add 8 to address, next 8 bytes.
  LOOP DROP ;
\ Scr#8
\ META FINAL STARTUP

\ Define IMMEDIATE last. It will hide the META word IMMEDIATE.
: IMMEDIATE ( --- )
  LATEST 40 TOGGLE            \ Set bit 6 in name length byte.
;

ASSEMBLER
 \ COLD startup code.
 HERE-T ORIGIN 1+ !-T         \ Patch jump address at cold entry point.
 11 C,-T META COLD ASSEMBLER \ LD DE,#'COLD Set entry pointer to COLD
 HERE-T 5 + JR                \ Jump past next LD DE,#xxx instruction.
 \ WARM startup code.
 HERE-T ORIGIN 4 + !-T        \ Patch jump address at warm entry point.
 11 C,-T META WARM ASSEMBLER \ LD DE.#'WARM Set entry pointer to WARM
 \ Common assembly code for COLD and WARM.
 NEXT XH LDP#                 \ Set IX register to NEXT address.
       D PUSH                 \ Save entry pointer.
          IM1                 \ Select Interrupt mode 1.
      2 A LD#
    1601 CALL                 \ Select channel 2, screen output.
        D POP                 \ Restore entry pointer.
 %Y 0 31 )LD#                 \ Set DF_SZ to 0, so we can use all 24 screen lines.
  5C3D SP LDP                 \ Set SP to ERR_SP.
        B POP                 \ Remove old address on stack.
ORIGIN 3 + B LDP#
       B PUSH                 \ Push WARM entry point address on stack, so
                              \ errors in ROM routines will now jump to WARM.
```

```
                             \ instead of the BASIC command line.
     0A B LD#
        BEGIN
           AF PUSH
        DSZ                  \ Push 10 dummy words in the stack to
                             \ protect against stack underflow.
       D PUSH                \ Save entry pointer.
     5C3A LDA                \ Load ERR_NR variable
        A INR
        NZ IF                \ If error is not "OK"
           1391 D LDP#
           0C0A CALL         \ Call PO_MSG in ROM to print error message.
         THEN
        D POP                \ Restore entry pointer.
 INTREG C@-T 1- A LD#
\ This is ugly! We read a byte from the interrupt vector table and use one less.
\ What should have been here:
\ INTREG 0 100 UM/MOD SWAP DROP A LD#
\ We cannot use signed division here, hence the complex expression.
         LDIA                \ Set I register to Interrupt vector table
                             \ when IM2 is activated later, custom interrupt
                             \ handler is used.
         EXDE
         JPHL                \ Jump to FORTH entry pointer, executing
                             \ either WARM or COLD, never return from there.
FORTH

 \ Fix up a few variables in target system.
 HERE-T UPTR @-T 0C + !-T    \ Set FENCE in target system.
 DECIMAL 26000 RPTR !-T      \ Iniitialize return stack pointer.
 LINK-T @ VL @  !-T              \ Store ponter to latest definition in
                                 \ FORTH vocabulary.
 26000 'R0 !-T                   \ Initialize Return stack bottom.
 25580 'S0 !-T                   \ Initialize stack bottom.

 EXIT \ Exit Meta compile loop.

DECIMAL  CR
 \ Save the target image staring at VIRTSTART-2 up to HERE-T
 \ to the file FORT83.BIN
 VIRTSTART 2- ( START)
 HERE-T ORIGIN 2- - ( LENGTH)
 GETFN FORT83.BIN
 50 BCAL
EXIT

\ End of meta source.
```