Toddy Forth 79

TODDY SOFTWARE *2022*

KELLY ABRANTES MURTA

ZX81 Toddy Forth-79

Version 1.13

Toddy Software 2022

In memoriam of little Toddy, there were countless happy moments that we will keep forever in our memories!

Toddy de Pedra de Guaratiba

(02/03/2008 - 20/06/2019)

PREFACE

Forth is a programming language created by Charles Moore in the 1970s and which became relatively popular on microcomputers in the 1980s. Among the reasons for this popularity, one can cite memory savings and processing speed, especially when compared to BASIC, the predominant language in personal microcomputers.

The first contact I had with Forth was in 1984 through the compiler for ZX81 published by Thomas Löw in issue 39 of the Brazilian magazine Micro Sistemas. Already used to the slowness of ZX BASIC, I was very impressed with the speed of the Forth compiler, but for a teenager living in a small town in the interior of Brazil it was extremely difficult to obtain more information about the language so at this time I advanced no further.

Some time later, with the emergence of interest in retrocomputing (especially the ZX81) it was a matter of time before my attention turned to Forth again, this time with a lot of information available on the Internet. That's when I found a Z80 assembler listing of Fig Forth and comparing it to Thomas Löw's Forth things started to make sense. And when I later discovered Brad Rodriguez's Camel Forth, I was able to incorporate into the Löw's minimalist Forth several of the newly learned techniques, which resulted in the launch of Toddy Forth in 2011.

Over the time that followed, several new releases were shared with the Sinclair Forum community[1] and some well-founded criticisms were put forward, Toddy Forth was still not a consistent product, there were gaps to be filled. And that's what I started doing in mid-2019 when I decided to do a complete overhaul of the Toddy Forth in order to make it conform to an established standard and ended up opting for the Forth 79-Standard. And so we come to the ZX81 Toddy Forth-79, a complete Forth system destined for the ZX81 microcomputer and whose resources and usability will be presented on the following pages.

But first I would like to thank Thomas Löw for introducing me to Forth; Brad Rodriguez, for his Camel Forth and the series of articles "Moving Forth", essential for understanding the "insides" of the Forth; Lennart C. Benschop for the Editor, Double Extension, Floating Point Extension and other word sets, taken from his excellent Forth-83 for ZX Spectrum; Coos Haak Utrecht, for the Assembler Extension Word Set. And a very special thanks to Fred (Moggy), for the criticisms, suggestions, testing, bug reports and text revision. Your enthusiasm with Toddy Forth was a great incentive to continue development.

Good reading, and have fun using the Toddy Forth-79!

Kelly A. Murta
October, 2022

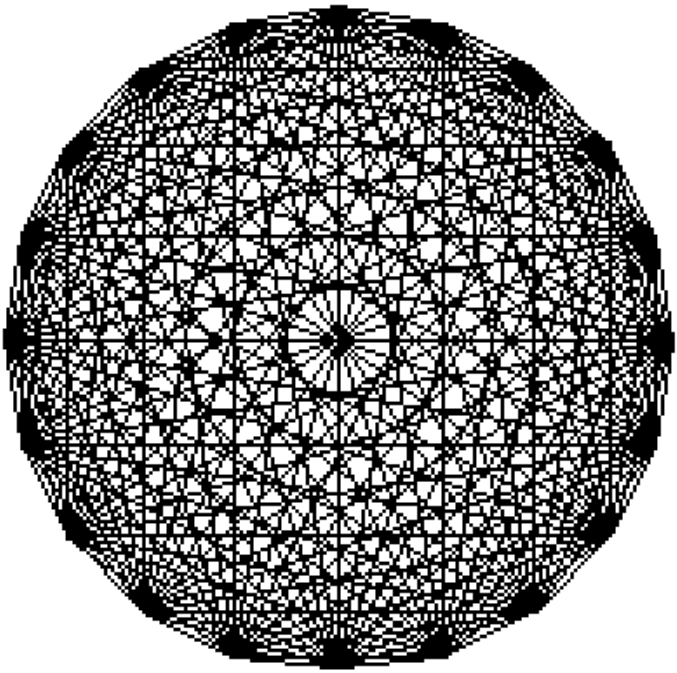
1 https://sinclairzxworld.com/

WHAT'S NEW IN VERSION 1.13

Toddy Forth-79 has undergone a major change since the release of version 1.03 that makes it practically a new product:

- Built-in support for Chroma-81 has been moved to an external extension.
- A new keyboard routine, now independent of the video driver and functional wih external mechanical keyboards.
- The SHIFT+BREAK combination is capable of breaking most programs, either in forth or in machine code.
- Reorganization of system and user variables to make them compatible with the adopted multitasking system.
- Word **PAUSE** renamed to **WAIT**.
- Removed the variable **(WAIT)** and added a new word **PAUSE** with similar function.
- Word **REVERSE** renamed to **INVERSE**.
- **CH128** word renamed to **CHR$128**.
- Removed the words **USER CLEAR** and **COPY**.
- Removed the variable **'ERRNUM** and created the word **ERRNUM** as execution vector.
- **KEY EMIT AT PAGE ABORT** recreated as execution vectors.
- Word **SAVE** renamed to **FSAVE**.
- Added the new words: **FLOAD RAND RND INKEY PLOT MEM EXEC: IS VALUE TO +TO**.
- Extensive code reorganization.
- Added new Extension Word Sets for high-res graphics, multi-tasking and Chroma-81 interface support.
- Added Copy Line functions to the Editor.

Contents



Chapter 1

1 THE ZX81 TODDY FORTH-79

1.1 INTRODUCTION

The Toddy Forth-79 (named TF79 in the rest of this document) is a complete Forth system for the ZX81 microcomputers. It is based on Forth-79 Standard specifications, with the REQUIRED WORD SET fully implemented, as well as some words from EXTENSION WORD SET and from REFERENCE WORD SET. The DOUBLE NUMBER EXTENSION WORD SET and the ASSEMBLER EXTENSION WORD SET are available to be loaded as needed. There are also extensions to support floating point numbers, ZX Printer, multitasking, high-resolution graphics and Chroma-81 color features, among others. Several words specific to the ZX81 hardware are also available.

TF79 is block-oriented, which means that the source code is stored in blocks of 1024 bytes. These blocks (also called Screens) are stored in RAM disk. One or more of these blocks can be saved as a single file on the SD card, to later be reloaded into the RAM disk.

Before proceeding, I would like to point out that this work intends to be a user guide for the Toddy Forth-79, describing its resources, functionalities and technical characteristics. It's not intended to teach programming in Forth, for that there are already excellent documentation available on the internet. I recommend starting with the following two books:

- *The Complete Forth, by Alan Winfield*
- *Starting Forth (First Edition), by Leo Brodie*

Note: *Most web versions of Brodies book has been modernised to ansi standard at the request of Brodie himself and is not so good for Forth-79 so one should seek out the original book itself or the original version in PDF as below.*

https://ia803101.us.archive.org/6/items/winfield_alan_the_complete_forth/winfield_alan_the_complete_forth.pdf

https://www.forth.com/wp-content/uploads/2018/01/Starting-FORTH.pdf

1.2 HARDWARE REQUIREMENTS

The TF79 does not support the use of cassette tapes, so is necessary to have a Zxpand interface (either the classic or plus version) connected to the ZX81 to allow system loading

and also to load and save files, in addition to providing the required RAM for the system (32Kb).

The system kernel is allocated in the lower part of RAM, between addresses 8192 and 16383. The memory area from 16384 to 32767 is dedicated to the system variables, screen file, user dictionary, stacks, terminal buffers and the screen buffer. The memory from 32768 to 40959 is used as a RAM disk for storing Forth screens, with the capacity to store up to 8 screens.

At version 1.12, support for Chroma-81 was removed from the kernel, being provided through an external extension that can be loaded when needed.

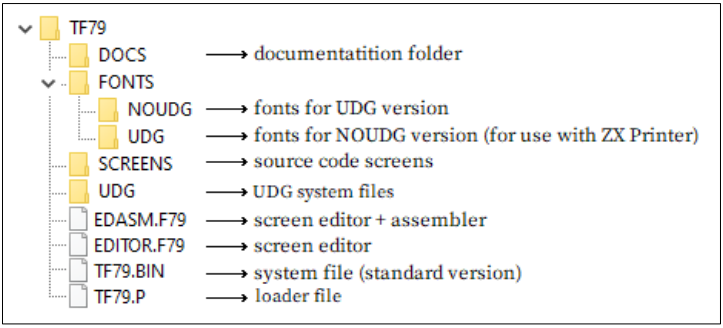
Sound generators compatible with the ZON X-81 continues to be supported, allowing direct PSG programming to generate complex sounds.

TF79 is available in both standard and UDG versions. The standard version only requires the ZXpand and uses conventional characters to represent ASCII characters not available on the ZX81. The UDG version requires in addition to the Zxpand, hardware that allows the redefinition of characters (such as the UDG4ZXPAND board created by Andy Rea or the Chroma-81 interface), thus making available all the ASCII characters used by Forth.

1.3 INSTALLATION

The basic TF79 is made up of two files, TF79.BIN which is the system kernel to be loaded into RAM at address 8192, and TF79.P which is the system loader. In addition, files with a .F79 extension can coexist, they are equivalent to the original TF79.P with the included user vocabulary saved with a proper name, such as EDITOR.F79, for example.

The original system comes with the following files and folders structure:

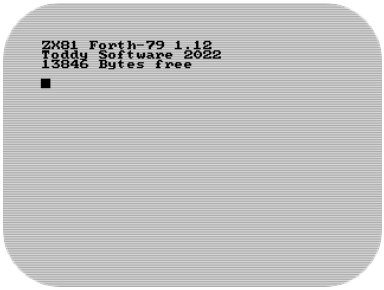


Copy the relevants folders to an SD card and insert it on ZXpand.

To load the system, go to the folder where you saved the files and type from BASIC:

```
LOAD "TF79"
```

The initial screen will be presented with the system identification, available memory information and the prompt with the flashing cursor. You will also hear an audible beep if you have a sound generator connected.



Type **VLIST** and press NEWLINE to see the list of words included in the TF79 (press and hold the SPACE key to pause listing or SHIFT+SPACE to stop the execution).

Note: *The kernel will always be loaded from the current directory. For example, if you want to use the Editor with the UDG version, you can proceed as follows, assuming you are in the /TF79 directory:*

```
CAT ">UDG"
LOAD "../EDITOR.F79"
```

The first sentence move to directory /TF79/UDG; the second loads the Editor from upper directory and then loads the kernel (TF79.BIN) from current directory.

It's also possible to load a ***.F79*** *file directly from the Forth environment using the* **FLOAD** *word, in this case the TF79.BIN will not be reloaded:*

```
FLOAD EDITOR.F79
```

1.4 CHARACTER SET

The TF79 works with ASCII characters that are internally converted to ZX81 codes when sent to the terminal. The 79-Standard defines the ASCII character set (code 32 to 127) as standard, but the TF79 adds an extra character subset to this (codes 128 to 160).

In the NOUDG version, characters missing on the ZX81 are represented by the following reversed characters:

char	ASCII		char	ZX CODE	HOW TO TYPE IT
!	21	→		8D	SHIFT 1
@	40	→		8C	SHIFT 2
#	23	→		95	SHIFT 3
%	25	→		99	SHIFT 4
[	5B	→		90	SHIFT Q
]	5D	→		91	SHIFT W
&	26	→		97	SHIFT E
©	7F	→		9A	–
'	27	→		8F	SHIFT T
_	5F	→		96	SHIFT Y
~	7E	→		94	SHIFT A
|	7C	→		8E	SHIFT S
\	5C	→		98	SHIFT D
{	7B	→		93	SHIFT F
}	7D	→		92	SHIFT G
^	5E	→		8B	SHIFT H

1.4.1 THE CHR$128 WORD

The UDG version uses by default the CHR$128 mode for the character generator. The CHR$128 uses ZX character codes from 128 to 191 to provide 64 new characters, making a total of 128 unique characters available that can be redefined at will. The FONTS folder contains some alternative character sets that can be loaded at address 15360 ($3C00) with

15360 BLOAD ***font.fn***.

To view the full TF79 character set, enter the following definition:

```
: CHARS CR 160 32 DO I EMIT LOOP ;
```

and enter **CHARS** to execute it.

Standard version

UDG Version

The operating mode of the character generator can be changed with the word **CHR$128**:

0 CHR$128 ---> disables CHR$128, returning to traditional operation mode
1 CHR$128 ---> activates CHR$128 mode

The result is instantaneous, experiment to better understand the difference between the two modes.

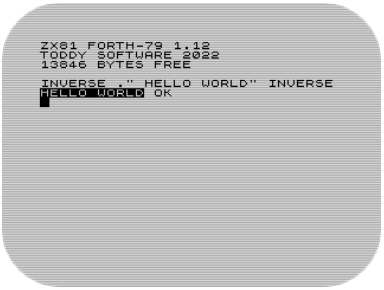
Notes:

1. **CHR$128** *only works with UDG devices that use bit 0 of the Z80's I register to access the extra 64 characters. The UDG4ZXPAND don't respond to it and the switching must be done physically through a jumper on the board.*
2. *TF79 ASCII characters have different encoding depending on the version used: standard or UDG. Hence the need for two versions for each character set in the FONTS/UDG and FONTS/NOUDG folders. You might be wondering why these character sets are needed for the standard version? Simple, the print module uses the character set stored from address 15360, so you can change it to have prints with different characters. And the word CHR$128 also works with the ZX Printer.*

1.4.2 THE INVERSE WORD

The print character routine makes an exclusive-or from the character code with the contents of address 16430 (**INVCHR**) before sending it to the screen. The word **INVERSE** switches the content of this address to 0 or 128, making it possible to switch the print between white background/black char and black char/white background. Try the example:

INVERSE ." HELLO WORLD" INVERSE



The **INVERSE** word is more suitable for use with the standard version of TF79, but it can also be used in the UDG version as long as the CHR$128 mode is disabled. Its use in CHR$128 mode makes no sense.

1.5 THE KEYBOARD

The TF79 comes with a new keyboard routine with auto repeat and audible feeback for key press (requires ZON X-81 or compatible PSG). The key combination SHIFT+SPACE is constantly monitored and its use allows aborting the execution of any program, but only if it is running in SLOW mode. This feature can be changed using **STOPON** / **STOPOFF** words.

The standard ASCII characters are mapped to the keyboard as shown below:

Symbols are accessible by simultaneously pressing the SHIFT key. The SHIFT-0 erases the last character entered and SHIFT-5 erases the entire line being typed. The NEWLINE key informs that the line you typed is finished and the typed commands must be executed (these work differently within the Screen Editor, as will be seen later on).

The SHIFT-9 key combination switches CAPS LOCK for lowercase letters. With the NOUDG version or if CHR$128 mode is inactive, lowercase letters will be represented by reverse capital characters (white letter, black background).

The character © and the extra subset (codes 128 to 160) are only available through their respective character codes for use with the words **EMIT** and **TYPE**.

TF79 has the words **KEY** and **INKEY** for reading the keyboard. **INKEY** reads the keyboard and leaves on the stack the ASCII code of the pressed key or 0 if no key was pressed. **KEY** waits until a valid key is pressed and leaves its ASCII code on the stack.
With the word **P@** it's possible to read directly the port FEh that gives access to the keyboard matrix. Following the keyboard representation with each address associated to five keys and each key is associated with one bit from D0 to D5 of the byte read from the FEh port.

Port Address	D0	D1	D2	D3	D4	D4	D3	D2	D1	D0	Port Address
FBFEh	1	2	3	4	5	6	7	8	9	0	FAFEh
FCFEh	Q	W	E	R	T	Y	U	I	O	P	F9FEh
FDFEh	A	S	D	F	G	H	J	K	L	NL	F8FEh
FEFEh	SH	Z	X	C	V	B	N	M	.	SP	F7FEh

P@ reads the port addressed by the number on the stack, leaving the byte read in the TOS. So to know if the '0' key was pressed, we do

```
64254 P@ \ read port FAFEh
1 AND    \ mask bit 0
```

which will leave 0 on the stack if the key was pressed or 1 otherwise.

To avoid key bouncing on external mechanical keyboards, such as the Memotech, the keyboard reading routine implements a bounce delay of approximately 7ms, which is suitable for most cases. However, if a longer delay is still required, you can change the value

of address 8732 (221Ch) which initially contains the value 30. For a bounce time of 9.6ms type (each additional unit adds approximately 0.24ms):

```
40 8732 C!
```

Note: *It should be noted that external keyboards such as the Fuller or Dk'tronics types which connect to the motherboard in the same manner as the membrane do not suffer debounce problems to the same degree as the Memotech which uses its own box of electronics to connect to the computer.*

Likewise, the keyboard repeat rate can be changed by changing the content of the bytes at addresses 8707 and 8725, respectively the waiting time to start repeating and the repeating time. The default value is 23 and 2 (I know, no much room for decrease the last).

Once you've found the suitable values for you, the changes can be saved permanently by typing:

```
8192 8192 BSAVE >TF79.BIN
```

This will overwrite the current file with the new version. If you want to preserve the current file, change the character **>** to **+** in the filename (refer to the ZXpand manual for details).

1.5.1 THE TERMINAL INPUT BUFFER

With the TF79 in interactive mode, everything typed echoes to the screen and is simultaneously stored in the terminal input buffer (**TIB**) before being interpreted, which occurs when the NEWLINE key is pressed.

The TIB's size is 128 bytes and when it is completely populated, the interpreter automatically takes control and starts to interpret what was typed. If this happens while you are entering a colon definition, the definition is likely to be incomplete. If the last word has not been truncated, Forth will remain in compilation mode and you can complete the definition by typing the remainder ending it with **;** .

But if the last word was truncated, it will likely not be found by the interpreter, generating an error warning and leaving an incomplete definition in the dictionary.

To remove the last incomplete definition from the dictionary, it's necessary to make it findable using the word **SMUDGE** and then remove it with **FORGET**:

SMUDGE FORGET <incomplete word>

1.6 ZONX-81 SOUND GENERATOR SUPPORT

The ZON X-81 was a Programmable Sound Generator produced by BI-PAK Semiconductors, based on the AY-3-8912 chip with 3 sound channels and full control over pitch, volume, tones and noise, all with envelope control. The technical details of PSG programming will not be discussed here, for that, consult the ZON X-81 manual and/or the AY chip datasheet, both available with a quick google search.

TF79 has two words to access PSG: **BELL** and **SOUND**.

BELL (--)
Emits a 1318 Hz sound lasting approximately 40ms. It is used in the word WARM, executed in the initialization of the TF and after the occurrence of an error.

SOUND (d r --)
Used to program the PSG, sends the number d to the register r of the AY-3-8912.

Follow some examples of sound effects that can be generated by PSG. Start defining the following auxiliary words (is not necessary to type the comments in parentheses or after the backslash):

```
16434 CONSTANT SEED

: RANDOMIZE 16436 @ SEED ! ;

: RND ( n1 -- n2) \ generates a random number between 0 and n1-1
  SEED @ 31521 * 6927 + DUP SEED ! U* SWAP DROP ;

: DLY (n -- )      \ delay
  0 DO LOOP ;

: CLPSG 14 0 DO 0 I SOUND LOOP ;

: MSOUND ( dn rn ... d2 r2 d1 r1 n -- ) \ program simultaneously
  0 DO SOUND LOOP ;                     \ n registers
```

Note:

Before running any of the examples it's recommended to execute **CLPSG** *to clear the PSG registers.*

Simple sounds (type in interpretive mode):

```
31 6 55 7 16 8 2 12 14 13 5 MSOUND \ Train
100 0 62 2 45 4 56 7 8 8 8 9 8 10 7 MSOUND \ Discord
210 0 90 2 60 4 56 7 16 8 16 9 16 10 40 12 8 13 9 MSOUND \ Bell
```

Complex sounds:

```
: BIRDS
  RANDOMIZE
  BEGIN
   13 8 254 7 0 1 3 MSOUND 85 RND 15
   DO I 0 SOUND 20 DLY
   LOOP ?TERMINAL
  UNTIL ;

: ADREAM  \ American Dream
  62 7 SOUND
  BEGIN
   2 -1
   DO 1 I - 7 * 2+  I 1+ 7 * 1+
    DO I 8 SOUND  51 100
     DO I 0 SOUND 15 DLY -1 +LOOP
      J NEGATE
     +LOOP 2
   +LOOP ?TERMINAL
  UNTIL ;

: WHISTLING 15 8 62 7 2 MSOUND 193 48 DO I 0 SOUND 30 DLY LOOP 63
7 SOUND ;
: EXPLOSION 31 6 7 7 2 MSOUND 11 8 DO 16 I SOUND LOOP 0 13 56 12 2
MSOUND ;
: LASER 0 13  15 12 16 8 55 7 4 MSOUND 32 1 DO I 6 SOUND 50 DLY 3
+LOOP ;
```

1.7 EXECUTION VECTORS AND VALUES

1.7.1 EXECUTION VECTORS

The action of a dictionary word is usually fixed at the time of its definition. The word can be redefined later and from then on all new references to the word will use its new definition. However, all previous references will still use the old meaning. Through the use of execution vectors it's possible to change the definition of a word in such a way that all references already compiled will also use the new version.

In TF79 an execution vector can be created with the word **EXEC:**. For example:

```
EXEC: CAKE
```

creates the **CAKE** execution vector. Initially this does nothing, but the action of the word can be set at any time with **IS**, as follows:

```
: CREAM-CAKE  CR ." YUMMY!" ;

' CREAM-CAKE  IS  CAKE
```

Now, whenever you execute the word **CAKE**, you'll get the response "**YUMMY!**". Now try:

```
: MUD-CAKE  CR ." YUCK!" ;

' MUD-CAKE  IS  CAKE
```

Now, **CAKE** provokes the response "**YUCK!**". The power of these words is that, unlike simply redefining a word with the same name, the action of the word changes immediately in every definition it has been compiled into.

TF79 comes with important words defined as execution vectors, which means that by substituting your own definitions the behavior of the system can be easily modified.

These are the words defined as execution vectors in TF79:

KEY	**PAGE**
EMIT	**ERRNUM**
AT	**ABORT**

Note: *Every execution vector is a code definition that has the following contents in its code field:*

LD HL,*exec_address*
JP (HL)

So, you should avoid defining execution vectors at memory addresses above 32767.

1.7.2 VALUES

A **VALUE** is a hybrid of **VARIABLE** and **CONSTANT**. We define a **VALUE** just like we define a **VARIABLE**:

```
VALUE THIRTEEN  ok
```

This will create the value **THIRTEEN** initialized to 0.

Just as we do with a **VARIABLE** we can change the value of **THIRTEEN**, for that we will use the word **TO** as follows:

```
13 TO THIRTEEN ok
```

However, we invoke the new **VALUE** the same way we do with a **CONSTANT**:

```
THIRTEEN . 13  ok

49 TO THIRTEEN  ok
```

```
THIRTEEN . 49  ok
```

We can also add a number to a **VALUE** with the word **+TO**:

```
VALUE A  10 TO A  ok

5 +TO A  ok

A . 15  ok
```

The words **TO** and **+TO** also works with words definitions, replacing the **VALUE** that follows it with whatever is currently in the TOS, so it can be dangerous to follow **TO**/**+TO** with anything other than a **VALUE**. It's perfectly safe to use **TO**/**+TO** with **VARIABLE**s and **CONSTANT**s.

You can see examples of using execution vectors and **VALUE**s in the definitions of **LINE** and **CIRCLE** in the next section.

1.8 THE PLOT WORD

TF79 has the **PLOT** word that allows you to manipulate graphic elements commonly called pixels. Each pixel corresponds to 1/4 of a character on the screen and is specified through its coordinates (x,y), on a screen 64 pixels wide and 48 pixels high. The pixel located at the lower left edge of the screen has coordinates (0,0) and the pixel located at the upper right edge has coordinates (63,47).

Once the pixel location is defined, you must decide what to do with it. There are four possibilities, or plotting modes:

0 - Set it white (unplot)
1 - Set it black (plot)
2 - Leave it alone (move)
3 - Change it (invert)

So **PLOT** needs three numbers on the stack: (x-coordinates, y-coordinates, ploting-mode --).

Try:

```
32 24 1 PLOT
```

which will make a little black square appear at the middle of the screen. The best thing to do is experiment with various coordinates and plotting modes to get familiar with it.

Words for plotting lines and circles with PLOT are defined next:

```
\ BRESENHAM'S LINE ALGORITHM
\ FROM ROSETTACODE.ORG
EXEC: STEEP       \ NOOP or SWAP
EXEC: YSTEP       \ 1+ or 1-

VALUE Y         VALUE COLOR
VALUE DELTAX    VALUE DELTAY

: LINE  ( x0 y0 x1 y1 color -- )
  TO COLOR
  ROT SWAP            ( x0 x1 y0 y1)
  2DUP  - ABS >R 2OVER  - ABS R> <
  IF    ' SWAP          \ swap use of
                        \ x and y
  ELSE 2SWAP ' NOOP
  THEN  IS STEEP  ( y0 y1 x0 x1)
  2DUP >
  IF SWAP 2SWAP SWAP    \ ensure
                        \ x1 > x0
  ELSE  2SWAP
  THEN                ( x0 x1 y0 y1)
  2DUP }
  IF    1-   ELSE   ' 1+
  THEN IS YSTEP
  OVER - ABS    TO DELTAY TO Y
  SWAP 2DUP - DUP TO DELTAX
  2/ ROT 1+ ROT ( error x1+1 x0)
  DO   I Y STEEP COLOR PLOT
       DELTAY - DUP 0<
       IF Y YSTEP TO Y DELTAX +
       THEN
  LOOP  DROP ;
```

```
\ MIDPOINT CIRCLE ALGORITHM

VALUE X    VALUE Y     VALUE C
VALUE P    VALUE X1    VALUE Y1
VALUE PY   VALUE PXY

: CIRCLE  ( x y r c -- )
  TO C    TO X1     TO Y    TO X
     0    TO P    0 TO Y1
  BEGIN X1 Y1 < NOT WHILE
    P Y1 2* + 1+   TO PY
    PY X1 2* - 1+  TO PXY
    X X1 +   Y Y1 +   C PLOT
    X X1 -   Y Y1 +   C PLOT
    X X1 +   Y Y1 -   C PLOT
    X X1 -   Y Y1 -   C PLOT
    X Y1 +   Y X1 +   C PLOT
    X Y1 -   Y X1 +   C PLOT
    X Y1 +   Y X1 -   C PLOT
    X Y1 -   Y X1 -   C PLOT
    PY TO P    1 +TO Y1
    PXY ABS PY ABS < IF
        PXY TO P  -1 +TO X1 THEN
  REPEAT ;
```

These words are available in the SCREENS folder, as files LINE.BLK and CIRCLE.BLK. Load them with

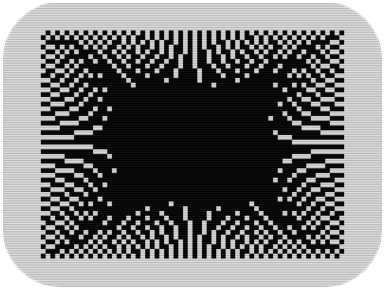
```
RUN SCREENS/LINE.BLK  ok
RUN SCREENS/CIRCLE.BLK  ok
```

Now, try this code:

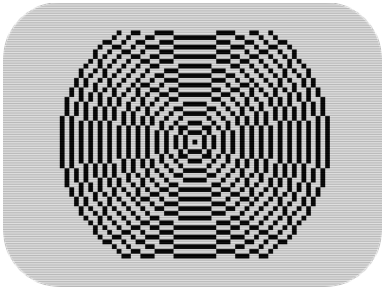
```
: MOIRE  ( step -- )
  PAGE 64 0
  DO I 0 63 I - 47 1 LINE
     I 47 < IF 0 I 63 47 I - 1 LINE THEN
  DUP +LOOP KEY 2DROP PAGE ;
```

and then enter `2 MOIRE` to see the result.

Also try with other numbers (3, 4, 5,…).



2 MOIRE



CIRCLES

The concentric circles in the last image were generated by the following word:

```
: CIRCLES  PAGE
  30 0 DO 32 24 I 1 CIRCLE
  2 +LOOP ;
```

Notes:

1. *The last coordinates (x,y) plotted are stored at addresses 16438 and 16439, respectively.* **PLOT** *accepts off-screen coordinates (x>63 and/or y>47), which allows drawing lines and circles that go beyond the screen boundaries, but values above 255 will produce erroneous results.*
2. *In the UDG version* **PLOT** *only works correctly with CHR$128 mode enabled.*

1.9 PSEUDO-RANDOM NUMBER GENERATOR

Something that is often very useful in certain applications is a random number generator, for example to simulate rolling dice. However, it is very difficult to produce random numbers using an 8-bit computer, but relatively easy to produce a sequence of numbers generated by a mathematical operation that appear to be random. These are commonly known as pseudo-random number generators (PRNG).

TF79 comes with a fast PRNG that uses the 'xor-shift' technique to produce a sequence of numbers with enough characteristics to be considered random for most needs. This generator produces a sequence of numbers from a starting number known as seed.

TF79 uses the words **RAND** and **RND** to generate pseudo-random numbers. **RAND** stores the stack top number in the SEED system variable (16434) to be used by **RND**, but if this number is null, the seed will be taken from the FRAMES system variable (16436). **RND** generates a number between 0 and the number in TOS – 1. Try the example below:

```
: MEASLES  PAGE
  BEGIN
    64 RND 48 RND 1 PLOT
  AGAIN ;
0 RAND  MEASLES
```

For the same root, the sequence of numbers generated by RND will always be the same.

1.10 ERROR CONDITIONS AND THEIR EFFECTS

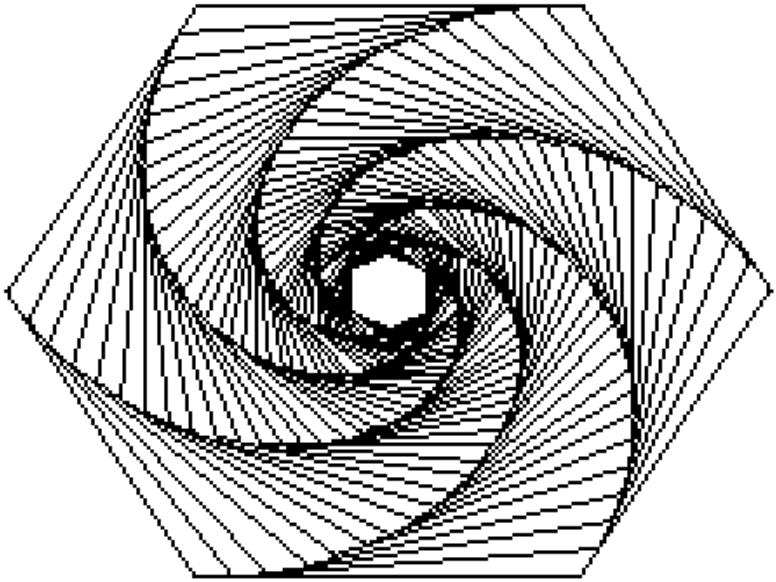
When an error is detected the following will occur:

- An error message will be displayed.
- The name of the Forth word that caused the error will be displayed.
- **ABORT** will be executed, returning the system to the interpreter and clearing the stacks.
- If the error occurs while loading a screen, the error location will be placed on the stack so that when typing **WHERE** the Editor will be started with the cursor positioned to the right of where the error occurred.

Not all error conditions will be detected and no error will be reported under the following conditions:

- Division by zero, the result **-1** will be returned on the stack.
- Negative number when the standard requires a positive number. In this case the number will be treated as an unsigned number.
- Wrong handling of the Return Stack, usually causes the system to crash.
- Invalid address provided for **EXECUTE**.
- Wrong construction of the **IF-ELSE-THEN**, **BEGIN-UNTIL-WHILE-REPEAT** and **DO-LOOP** structures.
- Stack imbalance when defining new words.
- Stack overflow.
- Execution of compilation words during interpretive mode.

Most of these errors have the potential to cause the system to crash.



Chapter 2

2 SCREENS AND FILES

2.1 THE RAM DISK

The source code in Forth is stored in blocks (also called screens) of 1Kb each. For convenience, the screens in Forth are stored in a RAM disk located in memory from the address stored in the **LO** variable until the end of the RAM (by default, from 32768 to 40959). The constant **#SCR** provides the total number of screens available on the RAM disk (8 screens by default).

Before using the RAM disk is advisable to clean it with the **FORMAT** command.

2.2 LOADING AND SAVING FILES

The command

n **GET** name

loads the file with the specified name from the SD card placing them in RAM disk from screen n.

The command

n1 n2 **PUT** name

saves to SD card the screens n1 to n2 with the chosen file name.

The command

n1 n2 **INDEX**

shows the first line of each screen from number n1 to n2. Therefore, it is advisable to store a comment (describing the screen) on the first line of each screen.

The command

DELETE name

deletes the specified file from the SD card.
The command

n **LIST**

displays the contents of screen n. As in the word **VLIST**, the listing is paused while the BREAK key is pressed.

The command

addr n **BSAVE** name

writes to the file with the specified name the block of n bytes located from the address addr.

You can add the characters **+** and **>** at the beginning of the name to force the action in case a file with the same name already exists:

addr n **BSAVE +**name

will rename the target file to name **.BAK** unless a backup already exists, in which case error **ZXP:8** normally occurs.

addr n **BSAVE >**name

will silently overwrite the target file if it already exists.

Note: *If no filename extension is provided, ZXpand will automatically add the* ***.P*** *extension, so always try to use a descriptive extension of the type of file to be created. For screen files the conventional is to use the* ***.BLK*** *extension.*

addr **BLOAD** name

loads the specified file into the addr address.

Note: *In the above commands, where applicable, the name argument can include the path in the referenced file name.*

The command

CAT path

List the contents of the directory specified by path or of the current directory if path is not specified. Examples of use:

CAT	shows contents of current directory
CAT dir	shows contents of directory dir
CAT /	shows contents of root dir
CAT >dir	moves to directory dir
CAT >..	move up one directory level
CAT >/	move to root directory
CAT +dir	creates a new directory dir

etc

When **CAT** fills the screen, the listing is paused until a key is pressed.

2.3 LOADING SCREENS

The command

n **LOAD**

will load the source code of the Forth program from screen n (previously loaded into RAM disk by the word **GET**). All text on the loaded screen will be interpreted as if it was being 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 take place on one screen and means that the next screen must be loaded as well.

The command

RUN name

is equivalent to

1 GET name
1 LOAD
It will load a file onto the RAM disk and immediately load (compile or execute) the source code contained in the screen 1 of that file.
The screens can contain **LOAD** commands, so that other screens can be loaded from one screen.
When an error occurs during loading, **ABORT** will leave the value of **>IN** and the number of the screen where the error occurred on the stack, so **WHERE** can be used to starts the editor on the given screen, with the cursor positioned to the right of the word that caused the error.

2.4 SAVING YOUR CHANGES IN TF79

To save the extended Forth system you must use the word **FSAVE**:

FSAVE name

Ex: **FSAVE EDITOR.F79**

This will generate the EDITOR.F79 file that can be loaded in the usual way from BASIC with the command `LOAD ''EDITOR.F79''` or directly from Forth with **FLOAD EDITOR.F79**. If a file with the same name already exists, the characters **+** or **>** can also be used to direct the **FSAVE** action, as seen in section 2.2.

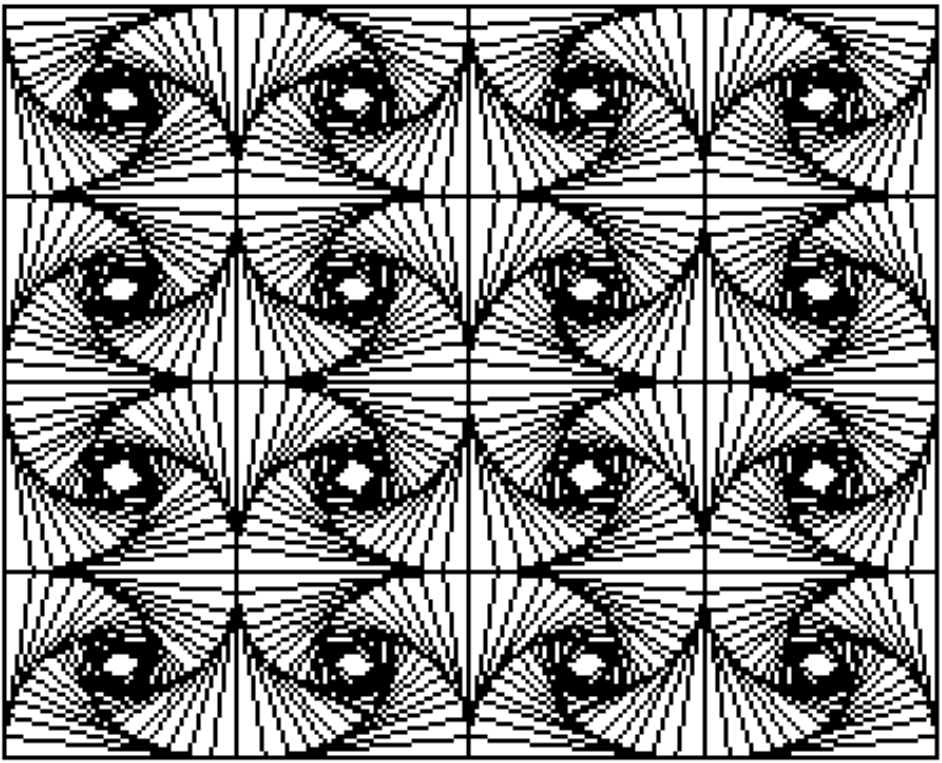
Note: *The use of the **.F79** extension for compiled Forth files is to differentiate them from normal **.P** files. If no extension is provided, ZXpand will automatically add the **.P** extension.*

IF you make any changes to the kernel, such as replacing the character set with another one, you can permanently save changes with the **BSAVE**:

```
8192 8192 BSAVE >TF79.BIN
```

or

```
8192 8192 BSAVE +TF79.BIN
```



Chapter 3

3 THE EDITOR

3.1 INTRODUCTION

The Screen Editor was developed by Lennart Benschop for his ZX Spectrum Forth-83 (https://lennartb.home.xs4all.nl/index.html) and adapted for use with the TF79, including the new copy line and paste line functions. Lennart kindly granted me permission to embed his Editor with the TF79, thank you very much Lennart!

The editor's source code is available in the SCREENS folder, to load it enter the sequence:

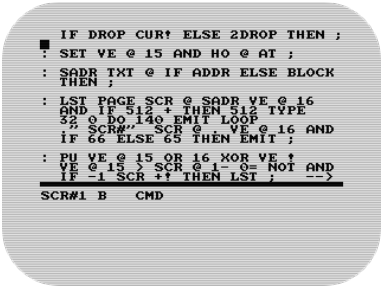
```
RUN SCREENS/EDITOR.BLK
```

Note: *The editor already compiled is available for immediate use, you can load it from BASIC with*

```
LOAD "EDITOR.F79".
```

Traditionally Forth screens are formatted in 16 rows by 64 columns, but in TF79 screens of 32 rows by 32 columns (1024 bytes) are defined.

The screen editor is started with **n EDIT**, where **n** is the number of the screen to be edited. Due to the limitations of the ZX81, the Forth screen will be divided into two parts of 16 rows by 32 columns, being shown one half at a time with the cursor. The screen number will be displayed at the bottom, followed by the letter **A** if the top half is shown or the letter **B** if the bottom half is shown. Further to the right, on the same line, the operating status of the editor (**EDT** or **CMD**) is informed.



3.2 THE CONTROL KEYS

The editor starts in Edit Mode (**EDT**) where the following keys have a special function:

Cursor keys (SHIFT+5 to SHIFT+8): Moves the cursor.
RUBOUT (SHIFT+0): Deletes the character at the cursor location. 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 beginning of the next line, the subsequent lines scroll to the left, preventing those words from being divided.
SHIFT+9: switches the CAPS LOCK
NEWLINE: Moves the cursor to the first position on the next line.
SHIFT+NEWLINE: Switches to command mode (**CMD**).

The Command Mode provides the following function keys:

6: Insert a blank line at cursor location. The lines below move one position down, the last line is deleted.
7: copy line at cursor location to **PAD** and then delete it. The lines below move up one position, a blank line appears at the end.
8: Inserts a space at the cursor location. The rest of the line changes 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 be scrolled along with the first word on the next line, it will move to the next line. One or more subsequent lines can also be shifted to the right.
C: copy line at cursor location to **PAD**.
E: Erases the line at cursor location by filling it with spaces.
H: moves cursor to the first position on first line of current screen.
K (+): moves to next half of screen. From 1A to 1B, from 1B to 2A.
J (-): moves to previous half of the screen. From 2A to 1B, from 1B to 1A. Both keys can be used to scroll through the RAM disk.
R: replaces line at cursor location with content of the **PAD**.
Q: exits the editor. You can undo the edits on the last screen with **EMPTY-BUFFERS**.

To return to Edit Mode, touch the NEWLINE key.

3.3 THE EDITOR MODES

The editor can be operated in two modes: file mode and block mode.

The **FILE** command puts the editor in file mode. The entire RAM disk is treated as a large text file. Insertions and deletions cover the entire RAM disk. The text will be moved across the screens when lines are inserted or deleted. The cursor keys also move across screens.

The **BLOCKS** command puts the editor in block mode. Each screen is treated individually. If a line is inserted, the last line on the current screen will disappear and will not be moved to the next screen. This is the normal way to edit source code in Forth.

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.

CT -- addr
timer for blinking cursor.

CUR c1 c2 --
switch between cursor and text character.

DEL --
erases the character at the cursor position.

DN --
moves the cursor one line down.

EKEY -- c
reads a character from the keyboard and returns the ASCII value as c.

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.

LCLR --
fill current line with blanks.

LCPY --
copy current line to PAD.

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.

LREPL --
replace the current line with the contents of the PAD.

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.

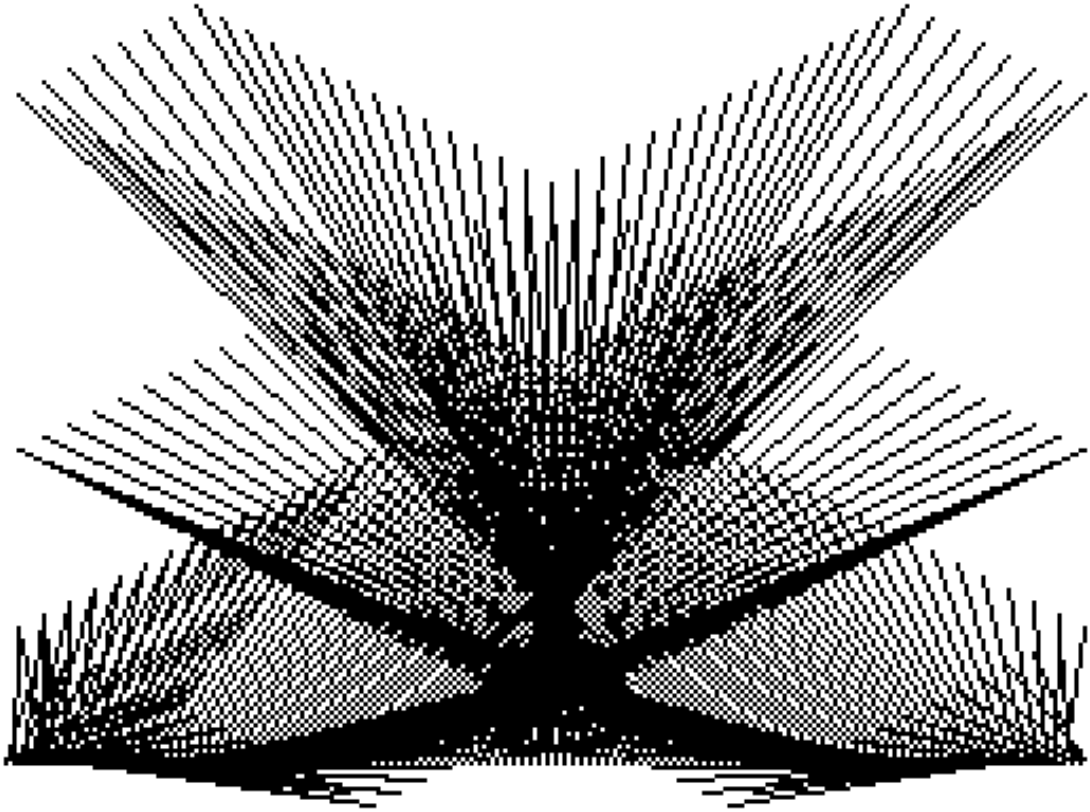
SADR -- addr
returns the start address of the current screen or block buffer.

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

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

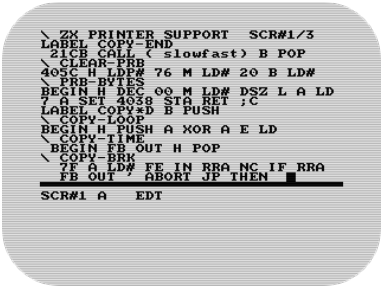
UP --
Moves the cursor one line up.

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



Chapter 4

4 THE ASSEMBLER EXTENSION WORD SET



4.1 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 and to Toddy Forth-79 by myself.

4.2 LOADING THE ASSEMBLER

The assembler consists of three files: TASM.BLK (size 1 screen), ASSEMBL1.BLK and ASSEMBL2.BLK (sizes 7 and 1 screens, respectively). 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 ASSEMBL1.BLK
```

and then you can save the system as described in section 2.4.

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

```
RUN TASM.BLK
```

This way the assembler is loaded at a high memory address outside the dictionary, leaving approximately 10Kb of memory available for use with the Assembler. 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.

4.3 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.

4.4 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,IY
- Use **XH** or **XL** instead of **H** or **L** for instructions that use H or L explicitly.
- For instructions that actually do indexing (IY+12) instead of (HL), use special mnemonics starting with **)**

By default the chosen **X** register is IY. The IX register is tied up by the video routines and will hardly ever be used on the ZX81.
The word **%X** changes the used index register to IX, **%Y** changes it back to IY.

4.5 THE INSTRUCTION SET

We use the same stack notation as in glossary list, 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 `r disp --`
loads r from address IY+disp. Equivalent to LD r,(IY+disp)

)ST r disp --
writes r to address IY+disp. Equivalent to LD (IY+disp),r

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

)LD# 8b disp --
Loads immediate value 8b at address IY+disp. Equivalent to LD (IY+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 IY+disp, B 2)LDP is equivalent to:

```
LD C,(IY+2)
LD B,(IY+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 IY+disp. **B 2)STP** is equivalent to:

```
LD (IY+2),C
LD (IY+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. Equivalent to LD SP,HL

LDAI --
loads A with I. Equivalent to LD A,I

LDIA --
loads I with A. Equivalent to LD I,a

EXAF --
exchanges AF and AF'. Equivalent to EX AF,AF'

EXDE --
exchanges DE and HL. Equivalent to EX DE,HL

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

CLR rps --
Loads rps with 0. Equivalent to 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 IY+disp to accumulator A. Equivalent to ADD A,(IY+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 IY+disp by 1. Equivalent to INC (IY+disp)
The instruction **)DER** works the same way (equivalent to DEC (IY+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 IY+disp one position to the left. Equivalent to RL (IY+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 IY+disp. Equivalent to BIT b,(IY+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 **JPIY** 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 4.6.
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 --
Call to Forth internal routine to print a character.

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

INBC r --
reads register r from port in register C. Equivalent to IN r,(C)
The **OUTBC** instruction works the same way (equivalent to OUT (C),r).

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, **UPTR** 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.

4.6 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 JR C.

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
```

Here are some examples of these structures, conventional Z80 assembler on the left and the conversion to Z80 Forth assembler on the right:

```
        code seg. 1
        JR Z,LBL01
          code seg. 2
LBL01:  code seg. 3
```

```
code seg. 1
NZ IF
  code seg. 2
THEN
code seg. 3
```

```
LBL01:  code seg.
        DJNZ LBL01
```

```
BEGIN
  code seg.
DSZ
```

```
        code seg. 1
        JR NC,LBL01
        code seg. 2
        JR LBL02
LBL01:  code seg. 3
LBL02:  code seg. 4
```

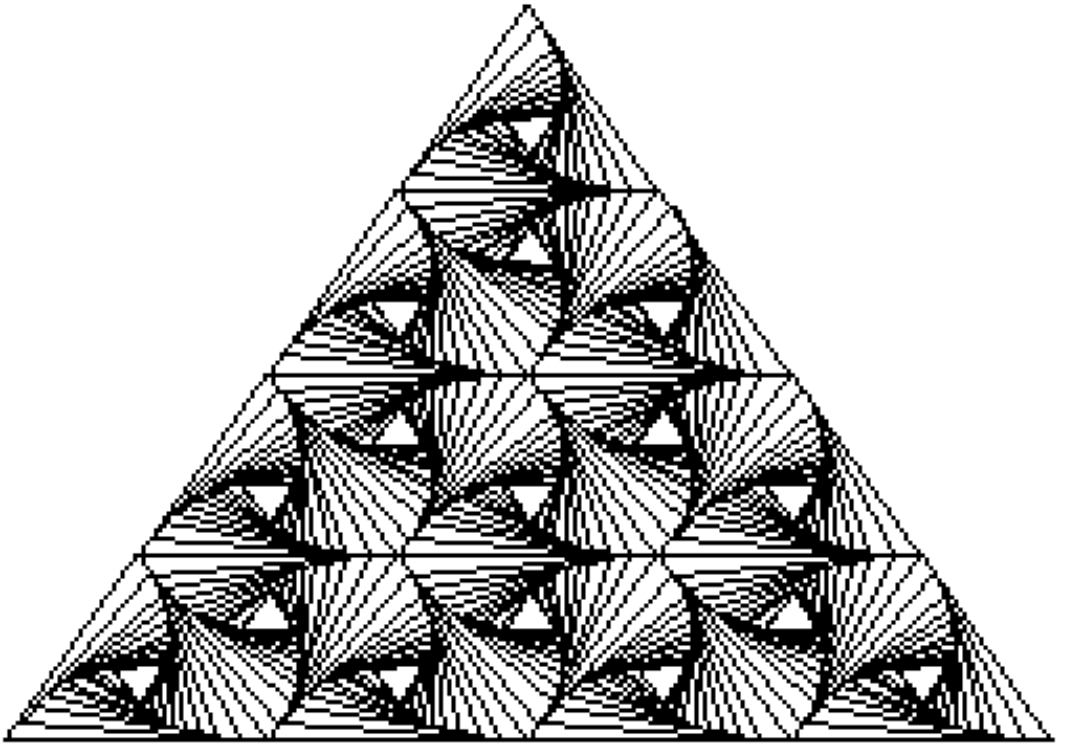
```
code seg.1
CS IF
  code seg. 2
ELSE
  code seg. 3
THEN
code seg. 4
```

```
LBL01:  code seg. 1
        JR Z,LBL02
        code seg. 2
        JR LBL01
LBL02:  code seg. 3
```

```
BEGIN
  code seg. 1
NZ WHILE
  code seg. 2
REPEAT
code seg. 3
```

```
LBL01:  code seg. 1
        JP C,LBL01
```

```
begin
  code seg. 1
cs NOT until
```



Chapter 5

5 THE DOUBLE NUMBER EXTENSION WORD SET

5.1 INTRODUCTION

The file DOUBLE.BLK (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 as explained in section 4.2. Then load **RUN DOUBLE.BLK** and finally (if applicable) **DISPOSE** to remove the assembler.

5.2 EXTRA WORDS

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

2UNDER xd1 xd2 -- xd2
removes the second value from the stack.

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

D* d1 d2 -- d3
multiplies d1 with d2.

D/ d1 d2 -- d3
divides d1 by d2.

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

D2* d1 -- d2
Multiplies d1 by 2.

D2/ d1 -- d2 D
divides wd1 by 2.

D= xd1 xd2 -- f D
f=true if xd1 and xd2 are equal, false otherwise.

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

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.

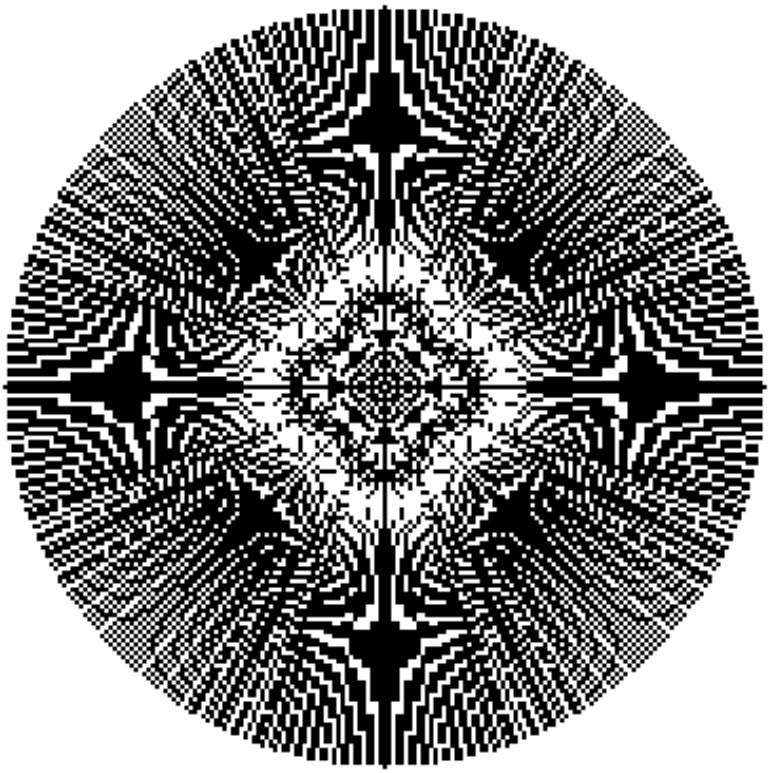
DU.R ud u --
like **D.R** but for unsigned number.

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.



Chapter 6

6 THE FLOATING POINT EXTENSION WORD SET

6.1 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.BLK (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.BLK** or **RUN ASSEMBL1.BLK**. Then the floating point extension can be loaded with **RUN FLOATING.BLK**. Finally the assembler can be removed with **DISPOSE**, if desired.

The second file TRANSCEN.BLK (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 (as described in section 2.4) 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.

6.2 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).

6.3 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 8

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 fp 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** execute 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.

6.4 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 8.

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 computations

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.

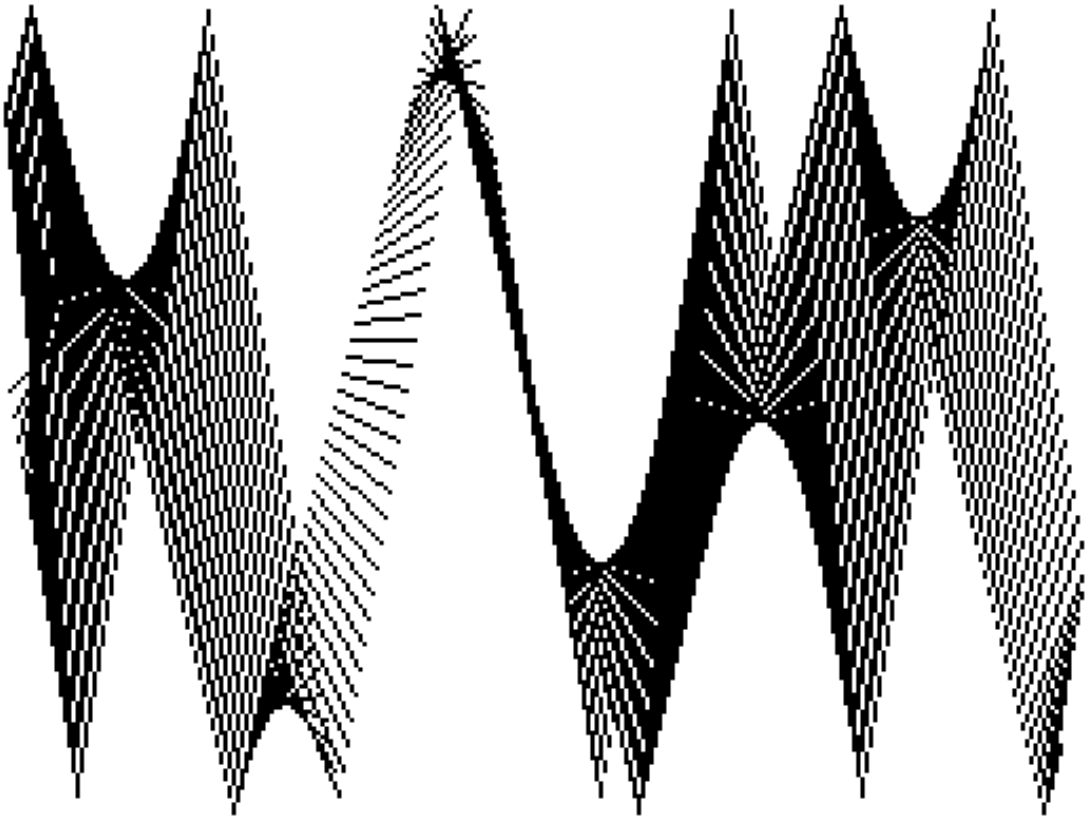
RAD fp1 -- fp2

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.



Chapter 7

7 THE PRINTER EXTENSION WORD SET

The file PRINTER.BLK adds to the ZX81 Toddy Forth-79 the functionality to redirect print output to the ZX Printer. Before loading it, you must first load the assembler as explained in section 4.2

In TF79 the **EMIT** word is an execution vector that by default executes the word **(EMIT)**. To redirect the output to the ZX Printer, **>P** redirect **EMIT** to execute the new word **PRINT**.

The following words are defined by the Printer extension:

>P --
Redirects text output to the printer.

>S --
Directs text output to the screen.

COPYSCR --
Sends a copy of the entire screen to the printer.

LLIST n1 n2 --
Sends the contents of screens n1 to n2 to the printer.

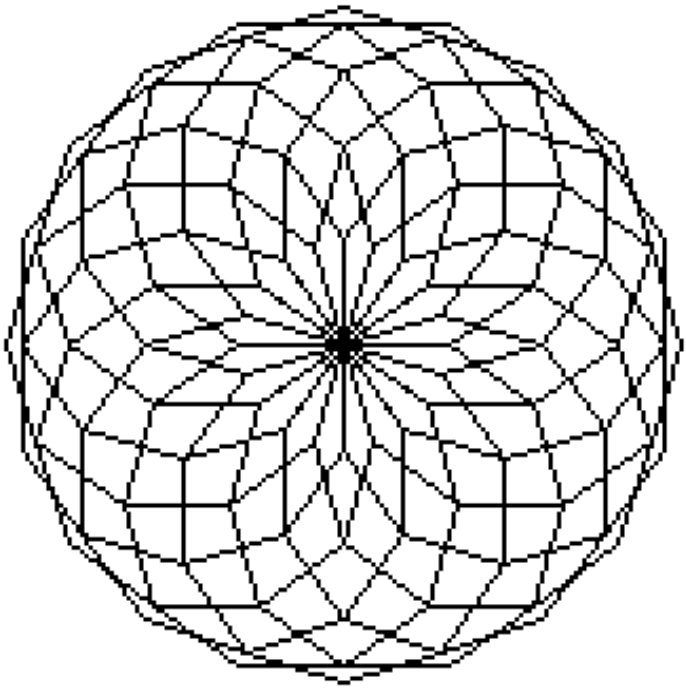
PRINT c --
Sends the character c to the printer.

Type

>P ." HELLO WORLD!" <enter>

Note that after pressing ENTER, all text output will be directed to the printer, including the "**OK**". What is being typed will only be printed after pressing the NEWLINE key or when the printer's buffer is full. Pressing the BREAK key stops printing and performs a **WARM** restart.

Another interesting feature is that printing uses the font of characters stored at address 15360 (1024 bytes), even taking into account bit 0 of the Z80 I register. This bit determines whether character generation will be in CHR$64 mode or in CHR$128 mode (which is the default).



Chapter 8

8 THE CHROMA-81 EXTENSION WORD SET

The file CHROMA.BLK implements all words necessary for the full support to the Chroma-81 interface. Before loading it, you must load the assembler as explained early and then execute **RUN CHROMA.BLK**.

8.1 THE CHROMA-81 INTERFACE

The Chroma-81 was developed by Paul Farrow to be connected to the expansion bus of the ZX81, allowing connection to a TV via a SCART connector to provide a sharp and clear RGB image. It also allows you to add colours to the image generated by the ZX81, supporting two modes of operation: the Character Colour Mode (Mode 0) and the Attribute Colour Mode (Mode 1). To use it with the TF79, set switches 3 and 6 in the ON position.

The character Colour Mode allows you to define two colours (a foreground colour and a background colour) for each scanline of each one of the 128 characters (characters with ZX codes 0-63 and 128-191). The character colour table is positioned from address 49152 ($C000) to address 50175 ($3CFF), 8 bytes for each character.

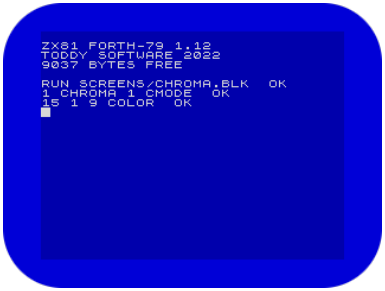
The attribute Colour Mode uses an attribute area to map the foreground and background colours for each character screen position, similar to the attributes area of ZX Spectrum. The attributes area is located from address 49323 ($C0AB) to address 50115 ($C3C3).

It is recommended to read the Chroma-81 documentation for a complete understanding of its features.

8.2 OPERATING THE CHROMA-81 WITH TF79

After loading the Chroma-81 extension, the first thing to do is vectorize the words **EMIT** and **PAGE** to work with the new features, which is done with the word **CHROMA**:

```
1 CHROMA
```



Then define the mode in which you want to work with the word **CMODE**:

```
1 CMODE        ( set mode 1)
```

Now set the screen colours:

```
15 1 9 COLOR
```

(you can experiment with other values from the list below)

```
 0 Black            8 Bright black
 1 Blue             9 Bright blue
 2 Red             10 Bright red
 3 Magenta         11 Bright magenta
 4 Green           12 Bright green
 5 Cyan            13 Bright cyan
 6 Yellow          14 Bright yellow
 7 White           15 Bright white
```

BORDER set screen border colour, **INK** and **PAPER** set foreground and background colours for upcoming screen prints. Try:

```
: HELLO
  16 0 DO
   CR I INK 15 I - PAPER
   I BORDER
   ."Hello World!" 10 WAIT
  LOOP ;
HELLO
```

Note: **INK** *and* **PAPER** *only work in Mode 1 (Attribute Colour Mode).*

For mode 0, the ink and paper attributes of the word **COLOR** are applied to the entire character set:

```
0 CMODE      ( set mode 0)
12 0 4 COLOR
```

DEFCOL is used to define the colours for a given character and **DEFCHR** defines its shape. Both expect 9 values on the stack, the TOS is the ASCII code of the character and the others represent each line of the character with the first line corresponding to the leftmost number. Each colour number in **DEFCOL** represents the background and foreground colours and is encoded as paper*16+ink. So, for paper=yellow and ink=bright red, we have 6*16+10=106. The example below defines the colours and a new shape for the '**>**' character:

```
7 7 6 10 6 13 13 15 62 DEFCOL
24 24 18 126 88 24 100 70 62 DEFCHR
EMIT 62
```

And lastly, to disable Chroma-81 colour features, use

```
0 CHROMA
```

8.3 THE CHROMA-81 WORD SET

BORDER (n --)
Sets the screen border colour.

CHROMA (f –)
Enable/disable the full colour support for Chroma-81. When f is true, the words **EMIT** and **PAGE** are vectored to **cemit** and **cpage**, respectivelly. With f false, **EMIT** and **PAGE** are re-vectored to its default actions **(EMIT)** and **(PAGE)**, and the colour feature of Chroma-81 are disabled.

CMODE (f --)
Defines the operating mode of Chroma-81 according to the value of f:
0 - sets the Character Colour Mode (Chroma Mode 0)
1 - sets the Attribute Colour Mode (Chroma Mode 1)

COLOR (ink paper border --)
In mode 0, sets the border colour and applies ink and paper colours to the entire character set. In mode 1, sets the colours for the entire screen.

DEFCHR (n0 n1 n2 n3 n4 n5 n6 n7 c --)
Defines the shape of a given character c.

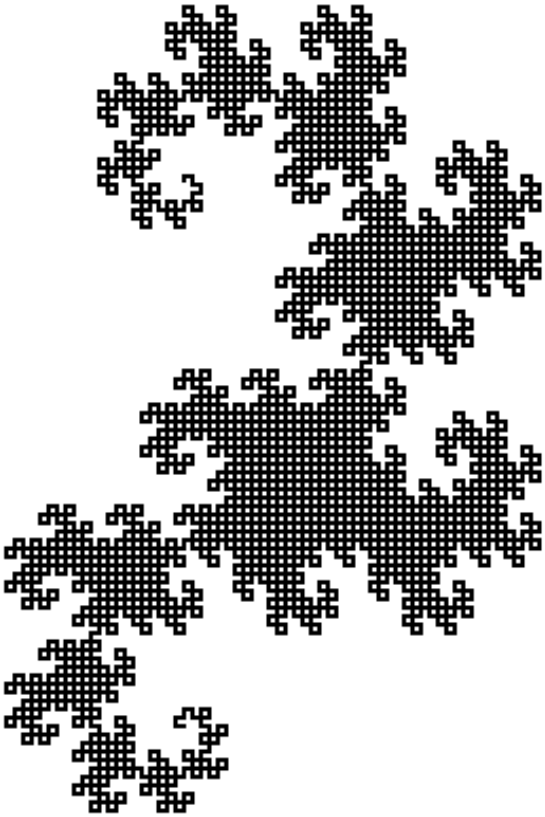
DEFCOL (n0 n1 n2 n3 n4 n5 n6 n7 c --)
Defines the paper and ink colours for each scanline (n0 to n7) of a given character c. The colour are coded as paper*16+ink.

INK (n --)
Sets the foreground colour for the next characters to be sent to the screen.

PAPER (n --)
Sets the background colour for the next characters to be sent to the screen.

cemit (c --)
Sent the character c to the screen updating the colours attributes. It's assigned to **EMIT** by **1 CHROMA**.

cpage (--)
Clear the screen updating the colours attributes. It's assigned to **PAGE** by **1 CHROMA**.



Chapter 9

9 THE HIRES GRAPHICS EXTENSION WORD SET

The file HGR.BLK (7 screens) contains all words required to implement a full high resolution graphic display with 256x192 pixels. It also offers a fully functional graphic terminal that can completely replace the standard text terminal.

Before loading the extension, you must first load the assembler as explained in section 4.2. Then execute **RUN HGR.BLK** and finally (if applicable) **DISPOSE** to remove the assembler.

9.1 THE GRAPHIC SCREEN

The graphic screen occupies 6144 bytes and is located at address 32768 (8000h), therefore overwriting the RAM disk area. If you have disponible more RAM (with Chroma-81 for example) you can move the high-resolution screen to an address with an alignment of 8192 bytes (A000h, C000h or E000h are the options), just edit the value of the constant **hfile** on screen 1 at HGR.BLK.

The graphic screen has a resolution of 256x192 pixels, with the coordinate (0,0) located at the bottom left of the screen, and the coordinate (255,191) at the top right. The graphic words accept coordinates beyond the limits of the screen, pixels outside the range are simply ignored. For text printing, the screen follows the same pattern as the standard terminal, 24 rows by 32 columns with row 0 at the top of the screen.

The graphical display can be manipulated from the text terminal or by activating the graphical terminal with the word **HTERM**. The advantage of the first mode is that the graphic screen is not polluted with the given commands. In the next sections we will see how to use both modes.

9.2 THE TEXT TERMINAL

It's the default terminal we've been using so far, there's not much to add so let's get straight to practical examples:

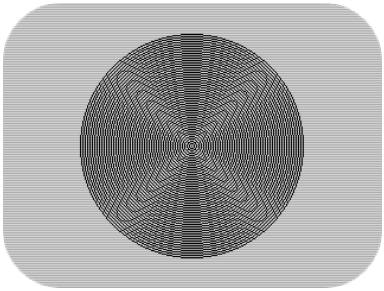
```
GPAGE HGR 1 GPEN 0 0 GSET 255 191 GLINE
```

GPAGE clears the HR screen, **HGR** activates it, **GPEN** sets the action of graphics commands (set pixel in this case), **GSET** sets the pixel at coordinate (0,0) and **GLINE** draws a line from the last coordinate to the point located 255 pixels to the right and 191 pixels above.

Type **TEXT** to return to the default terminal (you won't see what you're typing as the default text output is still the text screen).

Now for a more elaborate example, enter the following definition (or load the GCIRCLE.BLK from SCREENS directory):

```
\ MIDPOINT CIRCLE ALGORITHM
VALUE X    VALUE Y
VALUE X1   VALUE Y1
VALUE P    VALUE PY   VALUE PXY
: GCIRCLE  ( x y r -- )
     TO X1      TO Y     TO X
  0  TO P    0 TO Y1
  BEGIN X1 Y1 < NOT WHILE
     P  Y1 2* + 1+   TO PY
     PY X1 2* - 1+   TO PXY
     X  X1 +   Y Y1 +   GSET
     X  X1 -   Y Y1 +   GSET
     X  X1 +   Y Y1 -   GSET
     X  X1 -   Y Y1 -   GSET
     X  Y1 +   Y X1 +   GSET
     X  Y1 -   Y X1 +   GSET
     X  Y1 +   Y X1 -   GSET
     X  Y1 -   Y X1 -   GSET
     PY TO P    1 +TO Y1
     PXY ABS PY ABS <  IF
        PXY TO P   -1 +TO X1 THEN
  REPEAT ;
```



Then type in direct mode:

```
GPAGE 0 0 GSET
255 0 GLINE 0 191 GLINE
-255 0 GLINE 0 -191 GLINE
20 95 GSET
107 -75 GLINE 107 75 GLINE
-107 75 GLINE -107 -75 GLINE
127 95 45 GCIRCLE HGR
```

Cool, isn't it? ;-)

Finally, an example with **GLINETO**:

```
: GODSEYE  ( -- )
  76 0 DO 75 I -
     128 OVER 2* - 96 GSET
     128 OVER 24 +    GLINETO
     128 OVER 2* + 96 GLINETO
     128 I 96 +       GLINETO
     128 OVER 2* - 96 GLINETO
     DROP 3 +LOOP ;

HGR GPAGE GODSEYE
```

9.3 THE GRAPHIC TERMINAL

The graphic terminal is one of the most exciting features implemented by the HGR extension and once activated, the **EMIT**, **AT**, **PAGE** and **SLOW** actions will take place on

the graphic screen, using the character font stored at 15360. Therefore, it is an excellent option for who do not have available hardware for character redefinition.

To activate the graphical terminal, simply type **HTERM**, and if necessary, **PAGE** to clear the screen. The HGR screen with traditional cursor (static, it does not blink) will be presented and you will be able to act in exactly the same way as in the text terminal. Try typing **VLIST** and you will see that the response will be the same as the one obtained in the standard terminal, although a little slower (remember, each character is 8 bytes transferred to the screen and the scroll moves 6144 bytes against only 793 of the text screen).

Everything that can be done in the standard terminal can also be done in the graphical terminal, but there are some precautions to be taken:

- Words that switch to FAST mode (**BLOAD**, **BSAVE**, **FLOAD**, **FSAVE**, **GET**, **PUT**, etc) do not automatically return to the graphic screen, requiring the use of **SLOW** or **HGR** after use.
- The word **KEY** also loses access to the screen if used in **FAST** mode but don't despair, just type **SLOW** or **HGR** to return.
- It's possible to use the Editor, but only screens 7 and 9 will be available for editing. It's still possible to use the entire RAM disk to load screens from SD card with the words **GET** or **RUN**, the side effect is that the graphic screen will be polluted (nothing that a later **PAGE** won't solve).
- The PRINTER EXTENSION can also be used with the graphical terminal, but you will need to execute **HTERM** after use to return to the HR screen. If you prefer, redefine the **>S** word at PRINTER.BLK file:

```
: >S  HTERM ;
```

- **COLD** returns to text screens but **EMIT**, **PAGE**, **AT** and **SLOW** continue to acting on the graphic display. Again, use **HGR** or **SLOW** to return to the HR screen.

Another nice feature is that the word **INVERSE** presented in section 1.4.2 also works here and without the restrictions presented previously. Try:

```
CHARS INVERSE CHARS INVERSE
```

(The **CHARS** definition was presented in section 1.4.1)

9.4 ADDING SOME COLOURS

For those who have a Chroma-81, the words defined below allow you to add up to three different colours to the graphic terminal: border colour, background colour and foreground colour.

```
: BORDER ( n -- )
  48 + 32751 P! ;

: COLOR  ( n1 n2 -- )
  16 * + 32751 P@ 32 AND NOT IF
  [ coords 32774 + DUP 32 + ]
  LITERAL LITERAL DO DUP I C!
  LOOP THEN DROP ;
```

BORDER expects a number n in the stack that corresponds to the colour number. So, to set the green colour for the screen border we must enter **4 BORDER**. This set the colour border and also set the Chroma-81 to colour attibute mode. To disable the colour feature use **0 32751 P!**.

COLOR expects two numbers on the stack, n1 and n2 that correspond to the foreground colour (ink) and background colour (paper), respectively. Thus, **7 1 COLOR** will assign the colours white to ink and blue to paper.

See the list of colours codes in chapter 8.

Note: *See an example of using* **BORDER** *and* **COLOR** *in file COMPART.BLK available in the SCREENS folder.*

9.5 THE HGR WORDS

HGR (--)
Switches to the HR graphic screen.

TEXT (--)
Switches to the normal text screen.

GINVERT (--)
Invert the graphic screen.

GPAGE (--)
Clear the graphic screen.

GPEN (n --)
Set the behavior of **GSET**, **GLINE** and **GLINETO**:
n = 0 --> clear the pixel
n = 1 --> set the pixel
n = 2 --> invert the pixel

GSET (x y --)
Changes the point with the coordinates (x,y) according to the condition defined by **GPEN**.

GSET? (x y -- flag)
Give a true flag if the point with coordinates (x,y) is set.

GMOVE (x y --)
Change the "last point" coordinates to (x,y), without affecting the pixel.

GPOS? (-- x y)
Get the coordinates of the last point referred to by any other graphics word.

GLINE (dx dy --)
Draw a straight line from the last point, across dx pixels to the right, and up dy pixels. Either dx or dy can be negative, producing lines going left and down.

GLINETO (x y --)
An alternative to GLINE, with this word you specify the destination coordinates of the line, which is drawn to there from the last point.

(SLOW) (--)
Standard slow routine.

hslow (--)
HR slow routine.

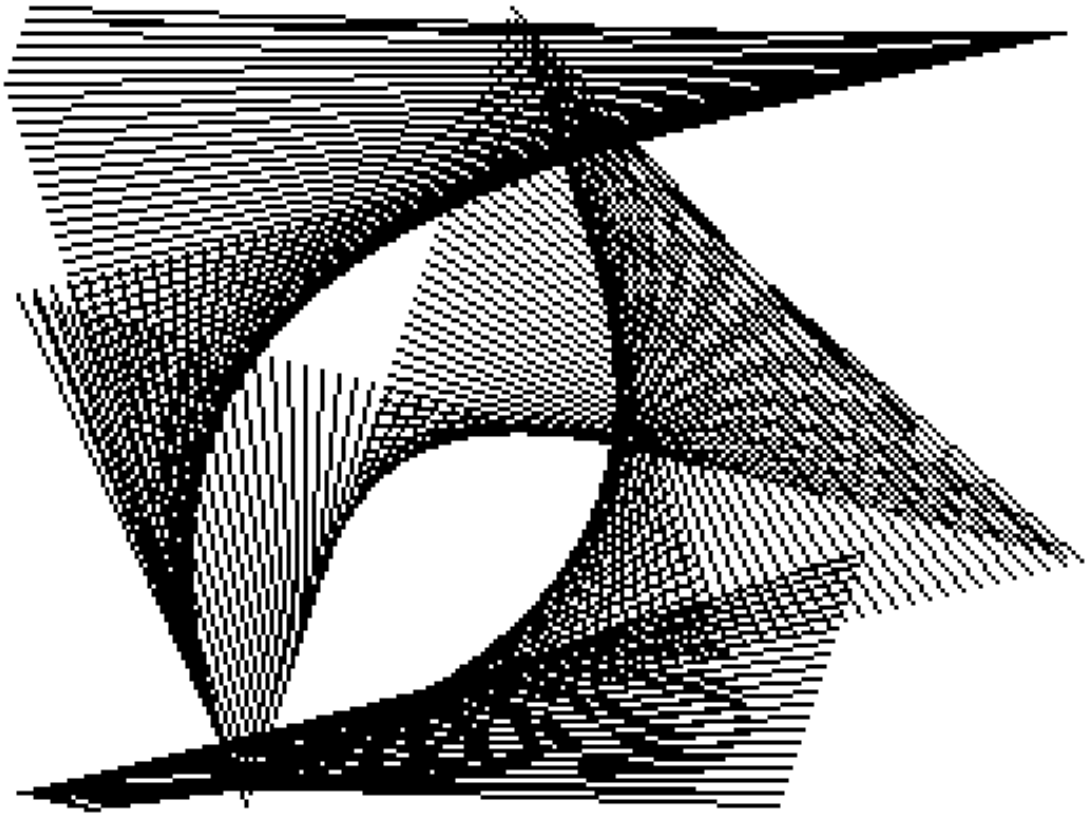
SLOW (--)
An execution vector, executes **(SLOW)** when in the standard terminal or **hslow** when in the graphical terminal.

hat (n1 n2 --)
Sets the print position on the graphic screen for row n1 and column n2.

hemit (c --)
Transmit character c to the graphic terminal.

HTERM (--)
Activates the graphic terminal and redirects the actions of **SLOW**, **AT**, **EMIT** and **PAGE** to the high resolution screen.

TERM (--)
Returns to standard terminal restoring the default actions of **SLOW**, **AT**, **EMIT** and **PAGE**.



Chapter 10

10 MULTI-TASKING SUPPORT

One of the most interesting new features in TF79 1.13 is support for multi-tasking programming, although this is nothing new in the Forth world. Here was adopted a model closely based on that used by Pygmy Forth/Z88 Camel Forth.

The file MULTI.BLK contains all words required to implement the Muti-tasking environment. Before loading it, you must first load the assembler as explained in section 4.2. Then execute **RUN MULTI.BLK** and finally (if applicable) **DISPOSE** to remove the assembler.

The instructions below are largely taken from the Z88 Camel Forth v3.00 documentation and adapted accordingly for the TF79.

10.1 THE MULTI-TASKING ENVIRONMENT

TF79 starts up in single-tasking mode; in this mode a single task (the terminal task) is running, and the task-switching word **PAUSE** is vectored to **NOOP** so that no time is wasted "switching" to the next task.

In multi-tasking mode, **PAUSE** causes the currently active task to switch control to the next "awake" task in the circular list. You can either use **PAUSE** specifically, or rely on one of the following words, which invoke it:

- **KEY** - every keyboard scan loop
- **WAIT** - every 20ms

Only the terminal task should accept input from the keyboard or load other sources. All data areas are shared between tasks (such as PAD, block buffers etc) so if you need to use these areas in more than one task, ensure that you save/restore them as required before and after each possible task switch. Each task has its own stacks and **BASE** settings; everything else is shared.

10.2 DEFINING AND RUNNING TASKS

The first thing you need to do is define your tasks with **TASK:**. Each task requires 266 bytes of RAM to hold its stacks and user variables. Since tasks can be set to run any word at any time, you only need to define as many as you will be running at once (excluding the terminal task, of course). For example, define a task called **A**:

```
TASK: A
```

Executing **A** returns a unique task ID, which is an argument or result for most of the remaining multiprogramming words.

Next set the task to run a particular word with **TASK!**. Normally this will contain an infinite loop with **PAUSE** at some strategic position. As an example:

```
: TEST  BEGIN  BELL PAUSE  AGAIN ;
' TEST A TASK!
```

Finally, you can enable multi-tasking with **MULTI** and set the task running by **WAKE**ing it:

```
MULTI
A WAKE
```

10.3 CONTROLLING TASKS

You can disable multi-tasking at any point with **SINGLE**, which has the effect of locking the system into the currently running task. All tasks previously "awake" are still awake, and will start running again as soon as multi-tasking is re-enabled with **MULTI**.

Any task (except the terminal task) can be stopped with **SLEEP**. For example, to halt task **A**, do:

```
A SLEEP
```

It's possible to re-**WAKE** the task, or use **TASK!** first to set it to run a new word. It's perfectly safe to use **WAKE** on an already awake task, or **SLEEP** on an inactive task; however if you attempt to use **TASK!** on an active task, a crash could occur.

If you wish a task only to run for a specified time, include the word **STOP** somewhere in the word it runs; this will send the current task to sleep and switch to the next.

Finally, you may use **PURGE** to deactivate all running tasks.

10.4 MULTI-TASKING DEMO

As a test, set up two tasks, each of which increments an associated counter. One increments its counter every 40ms and the other once every 400ms.

```
VARIABLE C1  VARIABLE C2
TASK: T1     TASK: T2
```

```
: TST1 BEGIN 2 WAIT ( 40 ms ) 1 C1 +! AGAIN ;
: TST2 BEGIN 20 WAIT ( 400 ms ) 1 C2 +! AGAIN ;
' TST1 T1 TASK!
' TST2 T2 TASK!
T1 WAKE  T2 WAKE  MULTI
```

Ocassionally check the counter values, e.g.

```
C1 ?
C2 ?
C1 ?
C2 ?
```

etc.

Then, as an additional test, set up two more tasks that merely display the values of the counters the tasks on the previous block are incrementing:

```
TASK: T3     TASK: T4
: CUR@ 16398 @ ;
: CUR! 16398 ! ;
: TST3 BEGIN 1 WAIT ( 20 ms ) CUR@ 0 26 AT
  C1 @ 6 .R  CUR! AGAIN ;
: TST4 BEGIN 1 WAIT ( 20 ms ) CUR@ 2 26 AT
  C2 @ 6 .R  CUR! AGAIN ;
' TST3 T3 TASK!
' TST4 T4 TASK!
T3 WAKE   T4 WAKE
```

Notice how the cursor address in the terminal task is preserved in **TST3** and in **TST4**.

10.5 THE WORDS OF MULTI

#TASKS (-- addr)
A variable containing the number of tasks currently active.

(PAUSE) (--)
Switch to next task

AWAKE? (task -- task'|0)
Returns 0 if the task specified is inactive; if it is active, the task ID of the previous task in the chain is returned.

LINK (-- task)
Returns the ID of the currently running task. This ID is actually the address of a variable containing the ID of the next task in the chain.

MULTI (--)
Enable multitasking.

MULTI-ABORT (--)
Special version of **ABORT** so that a non-TERMINAL task will put itself to sleep (with **STOP**) rather than doing **QUIT**.

PURGE (--)
Kill all processes and ret to **TERM**.

SINGLE (--)
Disable multitasking.

SLEEP (task --)
Remove task from chain unless **TERM**.

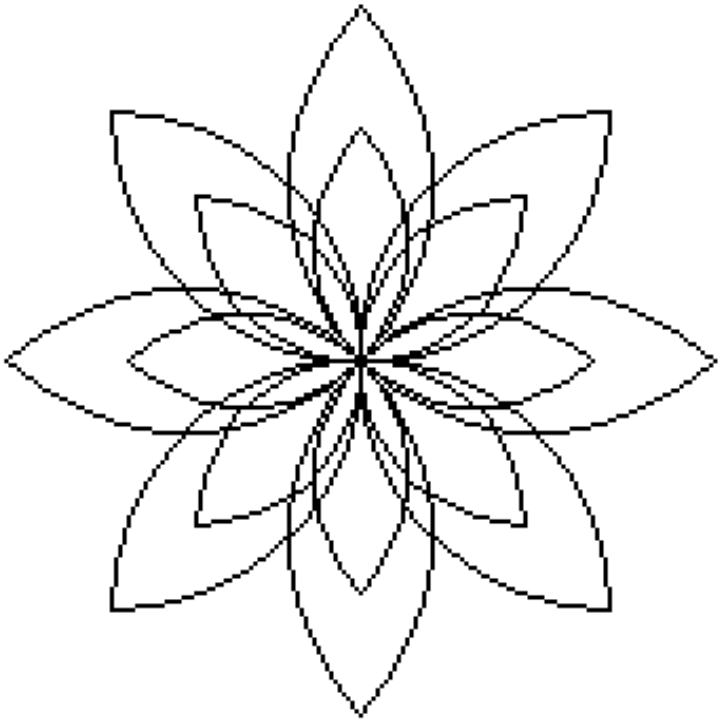
STOP (--)
Puts current task to sleep.

TASK! (addr task --)
Set up a task ready to **WAKE**.

TASK: ("name" --)
Create a new task.

TERM (-- task)
Returns the ID of the terminal task.

WAKE (task --)
Inserts task into chain



Chapter 11

11 INSIDE THE COMPILER

11.1 THE INNER INTERPRETER

The inner interpreter adopts Direct Threaded Code (DTC), which means it jumps directly to the addresses contained in a colon definition. Therefore, the Code Field of each Forth word contains machine instructions. For code definitions these are the machine instructions for the code definition. For constants, variables, colon definitions, and user variables, Code Field contains a subroutine call to the appropriate runtime routine (DOCON for constants, DOCREATE for variables, DOCOLON for colon definitions, DODOES for **DOES>** clause and DOUSER for user variables). For **CREATE DOES>** words the Code Field contains a subroutine call to the address directly after **(;CODE)** in the defining word. A subroutine call to DODOES is compiled at this address. 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 statement causes the Parameter Field to be pushed onto the stack, making it readily available to the handler routine (DOCON, DOCOLON, DODOES, DOUSER), or just leaving it there in case of variables.

Each code definition ends with a jump to the NEXT routine, made with the JP (IY) instruction (for the most important Forth routines, NEXT is built inline instead of having a jumper for it), where the address of the next routine is always stored in the IY register. The NEXT routine is the internal interpreter itself. It reads the address of the next word to be executed (pointed 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 instruction pointer in HL
        LD E,(HL)       ; Read next address, low byte
        INC HL
        LD D,(HL)       ; Read next address, upper byte
        INC HL
        EX DE,HL        ; Put instruction pointer in DE, jump address in HL
        JP (HL)         ; Jump to run address
```

- The item on top of the data stack (TOS) is stored in BC
- The Work register (W) is HL
- Instruction pointer (IP) is DE
- FORTH data stack pointer (PSP) is SP
- The NEXT address is stored in IY

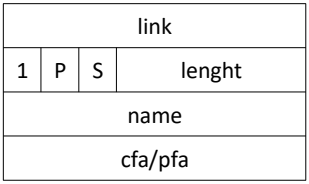
- Return stack pointer (RSP) is stored at address 402BH
- User area pointer (UP) is stored at address 400AH
- The X register is not used in this implementation. It is normally used to access the Parameter Field of the word being executed, but this is pushed on stack by CALL instruction in Code Field.

As with most systems, both the data stack and the return stack grow downwards. Whenever we refer to the top of the stack, it means the lowest memory address and the bottom of the stack is the highest memory address.

11.2 HEADER STRUCTURE

Words in TF79 are composed of four fields, they are:

- **Link Field:** points to the Name Field of the previous word in the dictionary. In the first word, this field contains 0. The size is always 2 bytes.
- **Name Field:**
  First byte:
  bit 7: always 1
  bit 6: Precedence bit, set if it is an immediate word
  bit 5: Smudge bit, set if word is not to be found
  bit 4..0: name length
  The following bytes contain the characters of the name.
- **Code Field:** 3-byte call instruction.
- **Parameter Field:** contains all additional information used by the word, for example, the value stored in a variable, the list of execution addresses in a definition of two points.



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 needed. Execution vectors words are code definitions.

FIND and **'** always return the Code Field address, which is the same as the compilation address.

>NFA converts Code Field address to Name Field address
>PFA converts Code Field address to Parameter Field address
>CFA converts Name Field address to Code Field address

11.3 MEMORY MAP

The TF79 requires the use of the ZXpand interface with the RAM memory mapped in the 8K-40K range. For this configuration the memory usage is as follows (addresses in decimal):

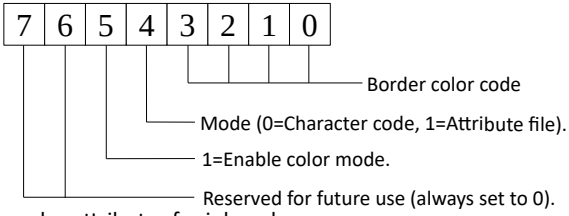
Note: *TF79 is completely independent of the BASIC interpreter, so some BASIC system variables have been replaced by Forth variables.*

8192: Forth kernel
15360: Character definition table
16384: BASIC and Forth system variables:

16391:	RSP	- Return stack pointer
16394:	UP	- user area pointer
16396:	DFILE	- screen address
16398:	CURADD	- screen cursor address
16400:	LKEY	- last key read
16401:	REP	- Key repeat counter
16402:	HCURADD	- Cursor address on HR screen
16417:	FTFLAGS	- Flags of the Forth: CFxxSKxA

C = cursor on/off, 1=on
F = Fast mode flag
S = Stop on/off
K = caps lock flag
A = attribute color mode, 1=active
x = flag not used

16418: CMBDR - Chroma color mode and border color



16419: ATTR - color attributes for ink and paper

16430: INVCHR - character inversion flag
16431: CTIMER - timer for blinking cursor
16444: PRTBUF

USER AREA:
Per task variables:
16477: LINK
16479: ‘SP
16481: S0
16483: R0
16485: BASE
Global variables:
16487: DP
16489: FENCE
16491: STATE
16493: >IN
16495: HLD
16497: VOC-LINK
16499: CONTEXT
16501: CURRENT
16503: DPL
16505: BLK
16507: SCR

16509: BASIC lines to load the Forth kernel

```
1 CONFIG "M=L"
2 LOAD "TF79.BIN;8192"
3 RAND USR 0
```

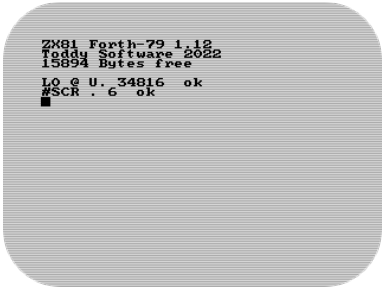
16555: Screen File (D-File)
17350: The **FORTH** word
17371: User dictionary starts here
31217: End of dictionary
31485: Start of Terminal Input Buffer (TIB, 128 bytes) and botton of data stack (S0), which grows downwards and has 256 bytes in size. For safety, there is a gap of 12 bytes between the data stack and the end of the dictionary.
31741: Botton of return stack (R0), grows towards TIB and is 128 bytes.
31743: Block buffer and its markup (which screen is contained in it).
32768: Start of RAM disk, 8192 bytes
40959: Last RAM position

11.4 CHANGING THE MEMORY MAP

The default memory map can be modified as needed by changing the value of RAMTOP and **#SCR** and restarting Forth. For example, if you need more 2Kb of RAM for the dictionary, you can adjust the value of RAMTOP to address 34816, modify the content of the constant **#SCR** to reflect the new RAM disk capacity and finally run **COLD** to restart Forth:

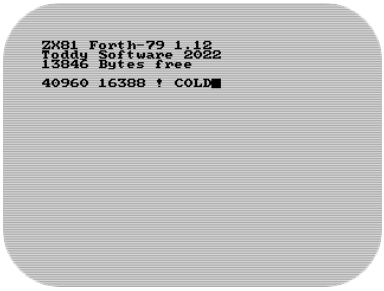
```
34816 16388 !  6 TO #SCR  COLD
```

Consequently, the locations of TIB, the data stack, the return stacks, and the block buffer will also change. **COLD** always sets the variable **LO** (the RAM disk start address) to the address given by RAMTOP.



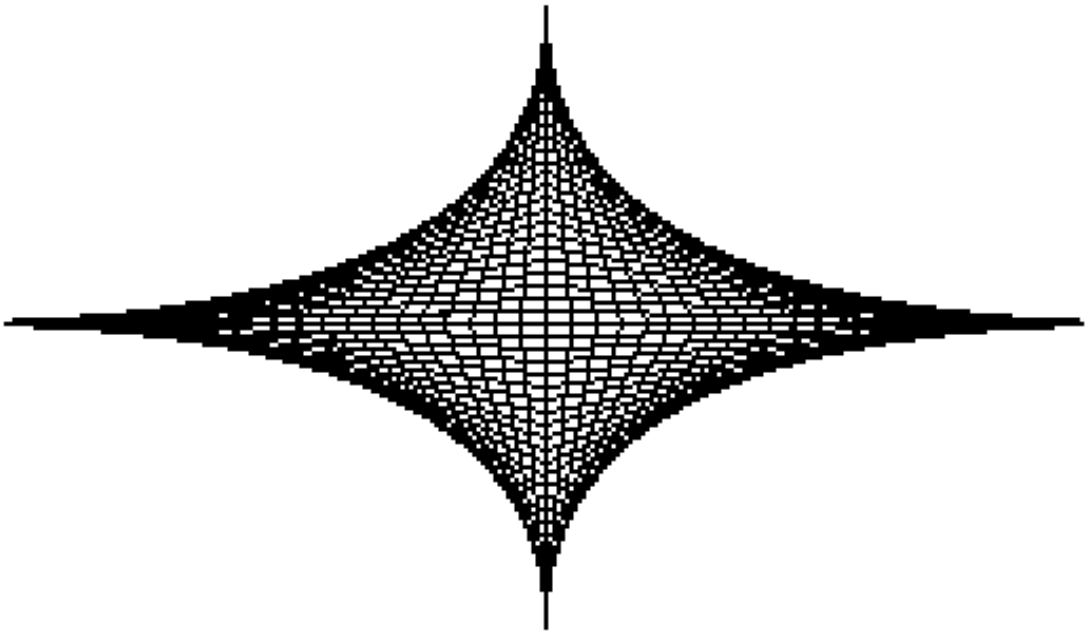
By attaching a Chroma 81 interface to the system, you gain more flexibility in memory usage. In this case, you can transfer the RAM disk to the memory at address 49152 leaving the 8 Kb at 32768 for the dictionary or another purpose:

```
40960 16388 !  COLD  49152 LO !
```



In this case it would not be necessary to change the content of **#SCR** as the RAM disk does not change its original capacity of 8 screens.
There are no restrictions on colon definitions created above address 32767, but code definitions in this area of memory should be avoided, as only a limited number of machine instructions are allowed there. The use of a prohibited machine instruction will fatally cause the system to crash.

One last reminder: if you save a program whose dictionary exceeds address 32767, you must change RAMTOP to the appropriate value before loading it from BASIC.



Appendix A

APPENDIX A - THE IMPLEMENTED WORD SETS

A.1 THE STACK NOTATION

The words in the list are in ASCII order and the word name is followed by any value that the word expects on the stack, followed by two hyphens, followed by any value that is returned on the stack.

After these, one or more capitalized symbols that indicate the word attribute can be followed:

F79 The word is part of the Forth-79 Required Word Set
R The word is part of the Forth-79 Reference Word Set
D The word is part of the Forth-79 Double Number Word Set (not fully implemented)
C The word may only be used during compilation of a colon definition.
I Indicates that the word is IMMEDIATE and will execute during compilation, unless special action is taken.
U A user variable.

The values in the stack (expected and returned) are represented as follows:

addr a memory address
b a value representing an 8 bit byte
c a value representing an 8 bit ASCII character code
d 32 bit signed 'double' number {-2,147,483,648..2,147,483,647}
f a numerical value with two logical states; 0 = false, non-zero = true.
n 16 bit signed integer number {-32,768..32,767}
u 16 bit unsigned integer number {0..65535}
ud 32 bit unsigned integer number {0..4,294,967,295}
x any cell value

A.2 DEFINITION OF TERMS:

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 from 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.

A.3 WORDS IN FORTH VOCABULARY

! n addr -- F79,112 "store"
Store n at addr.

ud1 -- ud2 F79,158 "sharp"
Generate from an unsigned double number d1, the next ASCII character which is placed in an output string. Result d2 is the quotient after division by BASE is maintained for further processing. Used between **<#** and **#>**.

#> d -- addr n F79,190 "sharp-greater"
End pictured numeric output conversion. Drop d, leaving the text address, and character count, suitable for **TYPE**.

#S ud -- 0 0 F79,209 "sharp-s"
Convert all digits of an unsigned 32-bit number ud, adding each to the pictured numeric output text, until remainder is zero. A single zero is added to the output string if the number was initially zero. Use only between **<#** and **#>**.

#SCR -- n
Returns the number of screens in RAM disk.

' -- addr F79,I,171 "tick"

Used in the form:

' <name>

If executing, leave the code field address of the next word accepted from the input stream. If compiling, compile this address as a literal; later execution will place this value on the stack. An error condition exists if not found after a search of the **CONTEXT** and **FORTH** vocabularies. Within a colon-definition ' <name> is identical to **['** <name> **] LITERAL**.

(-- F79,I,122 "paren"

Used in the form:

(ccc**)**

Accept and ignore comment characters from the input stream, until the next right parenthesis. As a word, the left parenthesis must be followed by one blank. It may freely be used while executing or compiling. An error condition exists if the input stream is exhausted before the right parenthesis.

(' -- addr

Reads word from input stream, looks it up in the dictionary and returns the compilation address or error message if the word is not found. Used in definition 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.

(EMIT) c --

Default code executed by **EMIT**.

(ERRNUM) f --

Default code executed by **ERRNUM**.

***** n1 n2 -- n3 F79,138 "times"

Leave the arithmetic product of n1 times n2.

***/** n1 n2 n3 -- n4 F79,220 "times-divide"

Multiply n1 by n2, divide the result by n3 and leave the quotient n4. n4 is rounded toward zero. The product of n1 times n2 is maintained as an intermediate 32-bit value for greater precision than the otherwise equivalent sequence: n1 n2 ***** n3 **/**

***/MOD** n1 n2 n3 -- n4 n5 F79,192 "times-divide-mod"
Multiply n1 by n2, divide the result by n3 and leave the remainder n4 and quotient n5. A 32-bit intermediate product is used as for ***/** . The remainder has the same sign as n1.

+ n1 n2 -- n3 F79,121 "plus"
Leave the arithmetic sum of n1 plus n2.

+! n addr -- F79,157 "plus-store"
Add n to the 16-bit value at the address, by the convention given for + .

+- n1 n2 -- n3
Negate n1 if n2 is negative, leaving the result as n3.

+LOOP n -- F79,I,C,141 "plus-loop"
Add the signed increment n to the loop index using the convention for **+** and compare the total to the limit. Return execution to the corresponding **DO** until the new index is equal to or greater than the limit (n>0), or until the new index is less than the limit (n<0). Upon the exiting from the loop, discard the loop control parameters, continuing execution ahead. Index and limit are signed integers in the range {-32,768..32,767}.

+TO n --
Add n to the value at the parameter field of <name>. Executed in the form:
n **+TO** <name>

, n -- F79,143 "comma"
Allot two bytes in the dictionary, storing n there.

- n1 n2 -- n3 F79,134 "minus"
Subtract n2 from n1 and leave the difference n3.

--\> F79,R,I,131 "next-block"
Continue interpretation on the next sequential block. May be used within a colon definition that crosses a block boundary.

-ROT n1 n2 n3 -- n3 n1 n2
Moves the top of stack to the third position.

-TRAILING addr n1 -- addr n2 F79,148 "dash-trailing"
Adjust the character count n1 of a text string beginning at addr to exclude trailing blanks, i.e., the characters at addr+n2 to addr+n1-1 are blanks. An error condition exists if n1 is negative.

. n -- F79,193 "dot"
Display n converted according to BASE in a free field format with one trailing blank. Display only a negative sign.

." F79,I,133 "dot-quote"
Interpreted or used in the form:
." ccc**"**
Accept the following text from the input stream, terminated by **"** (double-quote). If executing, transmit this text to the selected output device. If compiling compile so that later execution will transmit the text to the selected output device. At least 127 characters are allowed in the text. If the input stream is exhausted before the terminating double-quote, an error condition exists.

.R n1 n2 -- F79,R "dot-r"
Print n1 right aligned in a field of n2 characters, according to BASE. If n2 is less than 1, no leading blanks are supplied.

/ n1 n2 -- n3 F79,178 "divide"
Divide n1 by n2 and leave the quotient n3. n3 is rounded toward zero.

/MOD n1 n2 -- n3 n4 F79,198 "divide-mod"
Divide n1 by n2 and leave the remainder n3 and quotient n4. n3 has the same sign as n1.

0 -- 0
Constant 0

0< n -- f F79,144 "zero-less"
True if n is less than zero (negative)

0= n -- f F79,180 "zero-equals"
True if n is zero.

0> n -- f F79,118 "zero-greater"
True if n is greater than zero.

1 -- 1
Constant 1

1+ n -- n+1 F79,107 "one-plus"
Increment n by one, according to the operation of **+** .

1− n -- n-1 F79,105 "one-minus"
Decrement n by one, according to the operation of − .

2 -- 2
Constant 2

2! d addr – F79,D "two-store"
Store d in 4 consecutive bytes beginning at addr, as for a double number.

2* n1 -- n2 F79,R "two-times"
Leave 2*(n1).

2+ n -- n+2 F79,135 "two-plus"
Increment n by two, according to the operation of **+** .

2− n -- n-1 F79,129 "two-minus"
Decrement n by two, according to the operation of − .

2/ n1 -- n2 F79,R "two-divide"
Leave (n1)/2.

2@ addr -- d F79,D "two-fetch"
Leave on the stack the contents of the four consecutive bytes beginning at addr, as for a double number.

2DROP d -- F79,D "two-drop"
Drop the top double number on the stack.

2DUP d -- d d F79,D "two-dupe"
Duplicate the top double number on the stack.

2OVER d1 d2 -- d1 d2 d1 F79,D "two-over"
Leave a copy of the second double number on the stack.

2SWAP d1 d2 -- d2 d1 F79,D "two-swap"
Exchange the top two double numbers on the stack.

3 -- 3
Constant 3

79-STANDARD F79,119
Execute assuring that a FORTH-79 Standard system is available, otherwise an error condition exists.

: F79,116 "colon"
A defining word executed in the form:
: <name> ... **;**
Select the **CONTEXT** vocabulary to be identical to **CURRENT**. Create a dictionary entry for <name> in **CURRENT**, and set compile mode. Words thus defined are called 'colon-definitions'. The compilation addresses of subsequent words from the input stream which are not immediate words are stored in the dictionary to be executed when <name> is later executed. **IMMEDIATE** words are executed as encountered.
If a word is not found after a search of the **CONTEXT** and **FORTH** vocabularies, conversion and compilation of a literal number is attempted, with regard to the current **BASE**; that failing, an error condition exists .

; F79,I,C,196 "semi-colon"
Terminate a colon definition and stop compilation. If compiling from mass storage and the input stream is exhausted before encountering **;** an error condition exists.

< n1 n2 -- f F79,139 "less-than"
True if n1 is less than n2.

<# F79,169 "less-sharp"
Initialize pictured numeric output. The words:
 # #> #S <# HOLD SIGN
can be used to specify the conversion of a double-precision number into an ASCII character string stored in right-to-left order.

<CMOVE addr1 addr2 n -- F79,R "reverse-c-move"
Copy n bytes beginning at addr1 to addr2. The move proceeds within the bytes from high memory toward low memory.

= n1 n2 -- f F79,173 "equals"
True if n1 is equal to n2.

> n1 n2 -- f F79,102 "greater-than"
True if n1 is greater than n2.

>CFA nfa -- cfa
Convert the name field address of a definition to its code field address.

>IN -- addr F79,U,201 "to-in"
Leave the address of a variable which contains the present character offset within the input stream {{0..1023}}
See: **WORD (." FIND**

>NFA cfa -- nfa
Convert the code field address of a definition to its name field address.

>PFA cfa -- pfa
Convert the code field address of a definition to its parameter field.

>R n -- F79,C,200 "to-r"
Transfer n to the return stack. Every **>R** must be balanced by a **R>** in the same control structure nesting level of a colon-definition.

? addr -- F79,194 "question-mark"
Display the number at address, using the format of "**.**" .

?DUP n -- n (n) F79,184 "query-dupe"
Duplicate n if it is non-zero.

?TERMINAL -- f
Perform a test of the terminal keyboard for actuation of the break key. A true flag indicates actuation.

@ addr -- n F79,199 "fetch"
Leave on the stack the number contained at addr.

ABORT F79,101
Clear the data and return stacks, setting execution mode. Return control to the terminal.

ABORT" f -- F79,R,I,C "abort-quote"
Used in a colon-definition in the form:

```
ABORT" stack empty"
```

If the flag is true, print the following text, till **"** Then execute **ABORT**.

ABS n1 -- n1 F79,108 "absolute"
Leave the absolute value of a number.

ADDR n1 -- addr
Returns the address of screen n in the RAM disk.

AGAIN F79,R,I,C,114
Effect an unconditional jump back to the start of a **BEGIN-AGAIN** loop.

ALLOT n -- F79,154
Add n bytes to the parameter field of the most recently defined word.

AND n1 n2 – n3 F79,183
Leave the bitwise logical 'and' of n1 and n2.

ASCII -- c (executing) F79,R,I,C
-- (compiling)
Leave the ASCII character value of the next non-blank character in the input stream. If compiling, compile it as a literal, which will be later left when executed.

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 F79,R "bytes-per-buffer"
A constant leaving 1024, the number of bytes per block buffer.

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

BASE -- addr F79,U,115
Leave the address of a variable containing the current input-output numeric conversion base. {{2..70}}

BEGIN F79,I,C,147
Used in a colon-definition in the form:
BEGIN ... flag **UNTIL** or
BEGIN ... flag **WHILE** ... **REPEAT**

BEGIN marks the start of a word sequence for repetitive execution. A **BEGIN-UNTIL** loop will be repeated until flag is true. A **BEGIN-WHILE-REPEAT** loop will be repeated until flag is false. The words after **UNTIL** or **REPEAT** will be executed when either loop is finished. flag is always dropped after being tested.

BELL F79,R

Outputs a 1316Hz sound with a duration of 40ms.

BL -- n F79,R,176 "b-l"

Leave the ASCII character value for space (decimal 32).

BLK -- addr F79,U,132 "b-l-k"

Leave the address of a variable containing the number of the mass storage block being interpreted as the input stream. If the content is zero, the input stream is taken from the terminal.

BLANKS addr n -- F79,R,152

Fill an area of memory over n bytes with the value for ASCII blank, starting at addr. If n is less than or equal to zero, take no action.

BLOCK n -- addr F79,191

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.

BLOAD addr --

Reads a word from the input stream and reads the file with that name from mass storage into the RAM at address addr.

BSAVE addr u --

Reads a word from the input stream and saves u bytes from address addr to mass storage.

BUFFER n -- addr F79,130

Creates an empty block buffer for screen n (without reading it from the RAM disk) and returns its address.

C! n addr -- F79,219 "c-store"

Store the least significant 8-bits of n at addr.

C, n -- F79,R "c-comma"

Store the low-order 8 bits of n at the next byte in the dictionary, advancing the dictionary pointer.

C/L -- u

Returns the number of characters per line.

C@ addr -- b F79,156 "c-fetch"

Leave on the stack the contents of the byte at addr (with higher bits zero, in a 16-bit field).

CAT --

Shows the directory list from ZXpand(+).

CHR$128 n --

If n = 1, the CHR$128 character mode will be defined. If n=0, the CHR$64 character mode will be defined. If n<>0 and even, an error condition exists.

CMOVE addr1 addr2 n -- F79,153 "c-move"

Move n bytes beginning at address addr1 to addr2. The contents of addr1 is moved first proceeding toward high memory. If n is zero nothing is moved.

COLD --

Cold start of FORTH. It performs the following actions, in addition to **WARM**:

- Setup the RAM from the value in RAMTOP system variable.
- Removes all words from the dictionary at an address higher than **FENCE**.
- initializes all relevant user variables.

COMPILE F79,C,146

When a word containing **COMPILE** executes, the 16-bit value following the compilation address of **COMPILE** is copied (compiled) into the dictionary. i.e., **COMPILE DUP** will copy the compilation address of **DUP**.

CONSTANT n -- F79,185

A defining word used in the form:

n **CONSTANT** <name>

to create a dictionary entry for <name>, leaving n in its parameter field. When <name> is later executed, n will be left on the stack.

CONTEXT -- addr F79,U,151

Leave the address of a variable specifying the vocabulary in which dictionary searches are to be made, during interpretation of the input stream.

CONVERT d1 addr1 -- d2 addr2 F79,195

Convert to the equivalent stack number the text beginning at addr1+1 with regard to BASE. The new value is accumulated into double number d1, being left as d2. addr2 is the address of the first non-convertible character.

COUNT addr -- addr+1 n F79,159

Leave the address addr+1 and the character count of text beginning at addr. The first byte at addr must contain the character count n. Range of n is {0..255}.

CR F79,160 "c-r"

Cause a carriage-return and line-feed to occur at the current output device.

CREATE F79,239

A defining word used in the form:

CREATE <name>

to create a dictionary entry for <name>, without allocating any parameter field memory. When <name> is subsequently executed, the address of the first byte of <name>'s parameter field is left on the stack.

CURRENT -- addr F79,U,137

Leave the address of a variable specifying the vocabulary into which new word definitions are to be entered.

D+ d1 d2 -- d3 F79,241 "d-plus"

Leave the arithmetic sum of d1 plus d2.

D+- d1 n -- d2

Negate d1 if n is negative, leaving the result as d2.

D- d1 d2 -- d3 F79,D,129 "d-minus"

Subtract d2 from d1 and leave the difference d3.

D. d -- F79,D,129 "d-dot"

Display d converted according to **BASE** in a free field format, with one trailing blank. Display the sign only if negative.

D.R d n -- F79,D "d-dot-r"

Display d converted according to **BASE**, right aligned in an c character field. Display the sign only if negative.

D< d1 d2 -- f F79,244 "d-less-than"

True if d1 is less than d2.

DABS d1 -- d2 F79,D "d-absolute"

Leave as a positive double number d2, the absolute value of a double number, d1. {0..2,147,483,647}

DECIMAL F79,197

Set the input-output numeric conversion base to ten.

DEFINITIONS F79,155

Set **CURRENT** to the **CONTEXT** vocabulary so that subsequent definitions will be created in the vocabulary previously selected as **CONTEXT**.

DELETE

Reads a word from the input stream and deletes the file with that name.

DEPTH -- n F79,238

Leave the number of the quantity of 16-bit values contained in the data stack, before n added.

DLITERAL d -- d (executing) FIG
d -- (compiling)

If compiling, compile a stack double number into a literal. Later execution of the definition containing the literal will push it to the stack. If executing, the number will remain on the stack.

DNEGATE d -- -d F79,245 "d-negate"

Leave the two's complement of a double number.

DO n1 n2 – F79,I,C,142

Used in a colon-definition:

DO ... **LOOP** or

DO ... **+LOOP**

Begin a loop which will terminate based on control parameters. The loop index begins at n2, and terminates based on the limit n1. At **LOOP** or **+LOOP**, the index is modified by a positive or negative value. The range of a **DO-LOOP** is determined by the

terminating word. **DO-LOOP** may be nested. Capacity for three levels of nesting is specified as a minimum for standard systems.

DOES> F79,I,C,168 "does"

Define the run-time action of a word created by a high-level defining word. Used in the form:

: <name> ... **CREATE** ... **DOES>** ... **;**

and then <namex> <name>

Marks the termination of the defining part of the defining word <name> and begins the defining of the run-time action for words that will later be defined by <name>. On execution of <namex> the sequence of words between **DOES>** and **;** are executed, with the address of <namex>'s parameter field on the stack.

DP -- addr FIG,U

A user variable, the dictionary pointer, which contains the address of the next free memory above the dictionary. The value may be read by **HERE** and altered by **ALLOT**.

DPL -- addr F79,R "d-p-l"

A user variable containing the number of digits to the right of the decimal point in double integer input. A value of -1 for **DPL** indicates there is no decimal point and the number is recognized as a 16-bit integer.

DROP n -- F79,233

Drop the top number from the stack.

DUP n -- n n F79,205 "dupe"

Leave a copy of the top stack number.

ELSE -- F79,I,C,167

Used in a colon-definition in the form:

IF ... **ELSE** ... **THEN**

ELSE executes after the true part following **IF**. **ELSE** forces execution to skip till just after **THEN**. It has no effect on the stack. (see **IF**)

EMIT c -- F79,207

Transmit character to the current output device.

EMPTY-BUFFERS F79,145

Marks the block buffer as empty. **FLUSH** will not copy it to the RAM disk.

ERASE addr n -- F79,R,182

Fill an area of memory over n bytes with zeros, starting at addr. If n is zero or less, take no action.

ERRNUM -- addr

Executed if **NUMBER** encounters an error. It's an execution vector, so this way the word **NUMBER** can be extended to parse other data types, for instance floating point numbers.

EXEC:

A defining word executed in the form:

EXEC: <name>

to create an execution vector word, initially assigned to execute **NOOP**. The action of <name> can then be assigned to the word <action> with the use of word **IS**, as follow:

' <action> **IS** <name>

EXECUTE addr -- F79,163

Execute the dictionary entry whose compilation address is on the stack.

EXIT F79,C,117

When compiled within a colon-definition, terminate execution of that definition, at that point. May not be used within a **DO**..**LOOP**.

EXPECT addr n -- F79,189

Transfer characters from the terminal beginning at addr, upward, until a "return" or the count of n has been received. Take no action for n less than or equal to zero. One or two nulls are added at the end of text.

FAST --

Sets the FAST mode. Processing takes place without generating video, but **KEY** automatically turns to SLOW mode to allow the visualization of what is typed, returning to FAST mode when finished.

FENCE -- addr FIG,U

A user variable containing an address below which **FORGET**ting is trapped. To forget below this point the user must alter the contents of **FENCE**.

FILL addr n b -- F79,234

Fill memory beginning at address with a sequence of n copies of byte. If the quantity n is less than or equal to zero, take no action.

FIND -- addr F79,203

Leave the compilation address of the next word name, which is accepted from the input stream. If that word cannot be found in the dictionary after a search of **CONTEXT** and **FORTH** leave zero.

FIRST -- n FIG

A constant that leaves the first address of the block buffer.

FLOAD --

Reads a word from the input stream and loads from the SD card the file with that name. Executed in the form:

FLOAD <name.F79>

FLUSH

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 F79,186

Execute in the form:

FORGET <name>

Delete from the dictionary <name> (which is in the **CURRENT** vocabulary) and all words added to the dictionary after <name>, regardless of their vocabulary. Failure to find <name> in **CURRENT** or **FORTH** is an error condition.

FORMAT --

Fills the entire RAM disk with blank spaces.

FORTH -- F79,I,187

The name of the primary vocabulary. Execution makes **FORTH** the **CONTEXT** vocabulary. New definitions become a part of the **FORTH** until a differing **CURRENT** vocabulary is established. User vocabularies conclude by 'chaining' to **FORTH**, so it should be considered that **FORTH** is 'contained' within each user's vocabulary.

FSAVE --

Reads a word from the input stream and saves on the SD card with that name a copy of the user's vocabulary. Executed in the form:

FSAVE <name.F79>

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.

H. n --

Output n as a hexadecimal integer with one trailing blank. The current base is unchanged.

HERE -- addr F79,188

Return the address of the next available dictionary location.

HEX -- F79,R,162

Set the numeric input-output conversion base to sixteen.

HLD -- addr U

A user variable that holds the address of the latest character of text during numeric output conversion.

HOLD c -- F79,175

Insert c into a pictured numeric output string. May only be used between **<#** and **#>** .

I -- n F79,C,136

Copy the loop index onto the data stack. May only be used in the form:

DO ... **I** ... **LOOP** or

DO ... **I** ... **+LOOP**

IF f -- F79,I,C,210

Used in a colon-definition in the form:

flag **IF** ... **ELSE** ... **THEN** or

flag **IF** ... **THEN**

If flag is true, the words following **IF** are executed and the words following **ELSE** are skipped. The **ELSE** part is optional. If flag is false, words between **IF** and **ELSE**, or between **IF** and **THEN** (when no **ELSE** is used), are skipped. **IF-ELSE-THEN** conditionals may be nested.

IMMEDIATE F79,103

Marks the most recently made dictionary entry as a word which will be executed when encountered during compilation rather than compiled.

INDEX n1 n2 -- F79,R

Print the first line of each screen over the range {n1..n2}. This displays the first line of each screen of source text, which conventionally contains a title.

INKEY -- c

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

INTERPRET F79,R

Begin interpretation at the character indexed by the contents of **>IN** relative to the block number contained in **BLK**, continuing until the input stream is exhausted. If **BLK** contains zero, interpret characters from the terminal input buffer.

INVERSE --

Toggles bit 7 of the next characters to be sent to the terminal.
(See CH128 word).

IS --

Used to assign an action to an exection vector word. See **EXEC:**.

J -- n F79,C,225

Return the index of the next outer loop. May only be used within a nested **DO-LOOP** in the form:
DO ... **DO** ... **J** ... **LOOP** ... **LOOP**

KEY -- c F79,100

Leave the ASCII value of the next available character from the current input device.

LATEST -- addr FIG

Leave the name field address of the topmost word in the **CURRENT** vocabulary.

LEAVE F79,C,213

Force termination of a **DO-LOOP** at the next **LOOP** or **+LOOP** by setting the loop limit equal to the current value of the index. The index itself remains unchanged, and execution proceeds normally until the loop terminating word is encountered.

LIMIT -- addr

The address just above the block buffer.

LIST n -- F79,109

List the ASCII symbolic contents of screen n on the current output device, setting **SCR** to contain n. n is unsigned.

LITERAL n -- F79,I,215

If compiling, then compile the stack value n as a 16-bit literal, which when later executed, will leave n on the stack.

LO -- addr

Variable containing the bottom address of the RAM disk.

LOAD n -- F79,202

Begin interpretation of screen n by making it the input stream; preserve the locators of the present input stream (from **>IN** and **BLK**). If interpretation is not terminated explicitly it will be terminated when the input stream is exhausted. Control then returns to the input stream containing **LOAD**, determined by the input stream locators **>IN** and **BLK**.

LOOP F79,I,C,124

Increment the **DO-LOOP** index by one, terminating the loop if the new index is equal to or greater than the limit. The limit and index are signed numbers in the range {-32,768 ..32,767}.

M/MOD ud1 u2 -- u3 ud4 FIG "m-divide-mod"

An unsigned mixed magnitude math operation which leaves a double quotient ud4 and remainder u3, from a double dividend ud1 and single divisor u2.

MAX n1 n2 -- n3 F79,218 "max"

Leave the greater of two numbers.

MEM --

Reports the amount of memory available between the top of the data stack and the dictionary pointer.

MIN n1 n2 -- n3 F79,127 "min"

Leave the lesser of two numbers.

MOD n1 n2 -- n3 F79,104

Divide n1 by n2, leaving the remainder n3, with the same sign as n1.

MOVE addr1 addr2 n -- F79,113

Move the specified quantity n of 16-bit memory cells beginning at addr1 into memory at addr2. The contents of addr1 is moved first. If n is negative or zero, nothing is moved.

NEGATE n -- -n F79,177

Leave the two's complement of a number, i.e., the difference of zero less n.

NOOP --

No operation (do nothing).

NOT f1 -- f2 F79,165

Reverse the boolean value of f1. This is identical to **0=**.

NUMBER addr -- n F79,R

Convert the count and character string at addr, to a signed 32-bit integer, using the current base. If numeric conversion is not possible, an error condition exists. The string may contain a preceding negative sign.

OR n1 n2 -- n3 F79,223

Leave the bitwise inclusive-or of two numbers.

OVER n1 n2 -- n1 n2 n1 F79,170

Leave a copy of the second number on the stack.

P! b addr --

Writes b to the output port at address addr.

P@ addr -- b

Reads b from the input port address addr.

PAD -- addr F79,226

The address of a scratch area used to hold character strings for intermediate processing. The minimum capacity of PAD is 64 characters (addr through addr+63).

PAGE F79,R

Clear the terminal screen or perform an action suitable to the output device currently active.

PAUSE --

An execution vector that switch to next task in multi-tasking mode. In single-tasking mode execute **NOOP**.

PICK n1 -- n2 F79,240

Return the contents of the n1-th stack value, not counting n1 itself. An error condition results for n less than one.

2 PICK is equivalent to **OVER**. {1..n}

PLOT n1 n2 n3 --

Plots a single point with x-coordinate n1 and y-coordinate n2 with mode n3 (0 = unplot, 1=plot, 2=move, 3=invert). Coordinates are from (0,0) to (63,47).

PUT n1 n2 --

Reads a word from the input stream and writes the screens n1 through n2 to the mass storage media in a file with that name.

QUERY F79,235

Accept input of up to 80 characters (or until a 'return') from the operator's terminal, into the terminal input buffer. **WORD** may be used to accept text from this buffer as the input stream, by setting **>IN** and **BLK** to zero.

QUIT F79,211

Clear the return stack, setting execution mode, and return control to the terminal. No message is given.

R> -- n F79,C,110 "r-from"

Transfer n from the return stack to the data stack.

R0 -- addr U "r=zero"

A user variable containing the initial location of the return stack.

R@ -- n F79,C,228 "r-fetch"

Copy the number on top of the return stack to the data stack.

RAND u --

Set the random number seed. If u is zero then use the FRAMES counter.

REPEAT -- F79,I,C,120

Used in a colon-definition in the form:

BEGIN ... **WHILE** ... **REPEAT**

At run-time, **REPEAT** returns to just after the corresponding **BEGIN**.

RND u1 -- u2

Generates a pseudo-random number between 0 and u1-1, using as root the value stored at address 16434. See **RAND** .

ROLL n -- F79,236

Extract the n-th stack value to the top of the stack, not counting n itself, moving the remaining values into the vacated position. An error condition results for n less than one. {1..n}

3 ROLL = **ROT**

1 ROLL = null operation

ROT n1 n2 n3 -- n2 n3 n1 F79,212 "rote"

Rotate the top three values, bringing the deepest to the top.

RP! -- "r-p-zero"
Initialize the return stack pointer from **R0**.

RP@ -- addr "r-p-fetch"
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 FIG
Sign extend a single number to form a double number.

S0 -- addr F79,R "s-zero"
Returns the address of the bottom of the stack, when empty.

SAVE-BUFFERS F79,221
Copies the contents of the block buffer to the RAM disk if it contains a valid screen.

SCR -- addr F79,U,217
Leave the address of a variable containing the number of the screen most recently listed.

SIGN n -- F79,C,140
Insert the ASCII "-" (minus sign) into the pictured numeric output string, if n is negative.

SLOW
Sets the SLOW mode.

SMUDGE -- FIG
Used during word definition to toggle the "smudge bit" in a definitions name field. This prevents an uncompleted definition from being found during dictionary searches, until compiling is completed without error.

SOUND n1 n2 --
Writes n1 to register n2 of the AY sound chip (ZONX-81 compatible).

SP! --
Initialize the stack pointer from **S0**.

SP@ -- addr F79,R,214 "s-p-fetch"
Return the address of the top of the stack, just before **SP@** was executed.

SPACE -- F79,232
Transmit an ASCII blank to the current output device.

SPACES n -- F79,231
Transmit n spaces to the current output device. Take no action for n of zero or less.

STATE -- addr F79,U,164
Leave the address of the variable containing the compilation state. A non-zero content indicates compilation is occurring, but the value itself may be installation dependent.

SWAP n1 n2 -- n2 n1 F79,230
Exchange the top two stack values.

THEN F79,I,C,161
Used in a colon-definition in the form:
IF ... **ELSE** ... **THEN** or
IF ... **THEN**
THEN is the point where execution resumes after **ELSE** or **IF** (when no **ELSE** is present).

TIB -- addr
The address of the text input buffer. This buffer is used to hold characters when the input stream is coming from the current input device. In this system the capacity of **TIB** is 128 characters.

TO n --
Store n in the parameter field of <name>. Executed in the form:
n **TO** <name>

TOGGLE addr b -- FIG
Complement the contents of addr by the bit pattern b.

TYPE addr n -- F79,222
Transmit n characters beginning at address to the current output device. No action takes place for n less than or equal to zero.

U* un1 un2 -- ud3 F79,242 "u-times"

Perform an unsigned multiplication of un1 by un2, leaving the double number product ud3. All values are unsigned.

U. un -- F79,106 "u-dot"

Display un converted according to **BASE** as an unsigned number, in a free-field format, with one trailing blank.

U/MOD ud1 un2 -- un3 un4 F79,243 "u-divide-mod"

Perform the unsigned division of double number ud1 by un2, leaving the remainder un3, and the quotient un4. All values are unsigned.

U< un1 un2 -- f F79,150 "u-less-than"

Leave the flag representing the magnitude comparison of un1 < un2 where un1 and un2 are treated as 16-bit unsigned integers.

UNDER x1 x2 -- x2

Removes the second value from the stack.

UNTIL f -- F79,I,C,237

Within a colon-definition, mark the end of a **BEGIN-UNTIL** loop, which will terminate based on flag. If flag is true, the loop is terminated. If flag is false, execution returns to the first word after **BEGIN**. **BEGIN-UNTIL** structures may be nested.

UPDATE F79,229

Mark the most recently referenced block as modified. The block will subsequently be automatically transferred to mass storage should its memory buffer be needed for storage of a different block, or upon execution of **SAVE-BUFFERS**. As this system has a fast RAM disk, buffers will always be written back.

VALUE F79,227

A defining word executed in the form:

VALUE <name>

to create a dictionary entry for <name> and allot two bytes for storage in the parameter field, initializing it to zero. When <name> is later executed, the value stored in its parameter field will be left on the stack. This value can also be changed using the words **TO** and **TO+**.

VARIABLE F79,227

A defining word executed in the form:

VARIABLE <name>

to create a dictionary entry for <name> and allot two bytes for storage in the parameter field, initializing it to zero. When <name> is later executed, it will place the storage address on the stack.

VLIST F79,R

List the word names of the **CONTEXT** vocabulary starting with the most recent definition.

VOC-LINK -- addr U

A user variable containing the address of a field in the definition of the most recently created vocabulary. All vocabulary names are linked by these fields to allow control for FORGETting thru multiple vocabularies.

VOCABULARY F79,208

A defining word executed in the form:

VOCABULARY <name>

to create (in the **CURRENT** vocabulary) a dictionary entry for <name>, which specifies a new ordered list of word definitions. Subsequent execution of <name> will make it the **CONTEXT** vocabulary. When <name> becomes the **CURRENT** vocabulary (see **DEFINITIONS**), new definitions will be created in that list.

In lieu of any further specification, new vocabularies 'chain' to **FORTH**. That is, when a dictionary search through a vocabulary is exhausted, **FORTH** will be searched.

WAIT u --

Waits until u timer interrupts have occurred. While waiting, execute the word **PAUSE**.

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**.

WHILE f -- F79,I,C,149

Used in the form:

BEGIN ... flag **WHILE** ... **REPEAT**

Select conditional execution based on flag. On a true flag, continue execution through to **REPEAT**, which then returns back to just after **BEGIN**. On a false flag, skip execution to just after **REPEAT**, exiting the structure.

WORD c -- addr F79,181

Receive characters from the input stream until the non-zero delimiting character is encountered or the input stream is exhausted, ignoring leading delimiters. The characters are stored as a packed string with the character count in the first character position. The actual delimiter encountered (char or null) is stored at the end of the text but not included in the count. If the input stream was exhausted as **WORD** is called, then a zero length will result. The address of the beginning of this packed string is left on the stack.

XOR n1 n2 -- n3 F79,174 "x-or"

Leave the bitwise exclusive-or of two numbers.

[I,125 "left-bracket"

End the compilation mode. The text from the input stream is subsequently executed. See **]**.

[COMPILE] F79,I,C,179 "bracket-compile"

Used in a colon-definition in the form:

[COMPILE] <name>

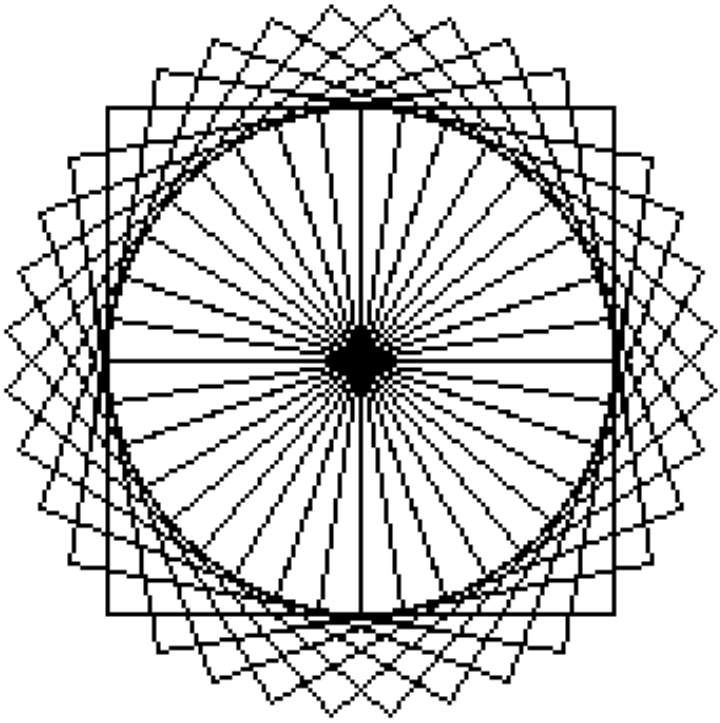
Forces compilation of the following word. This allows compilation of an **IMMEDIATE** word when it would otherwise be executed.

] F79,126 "right-bracket"

Sets the compilation mode. The text from the input stream is subsequently compiled. See **[**.

**** "backslash"

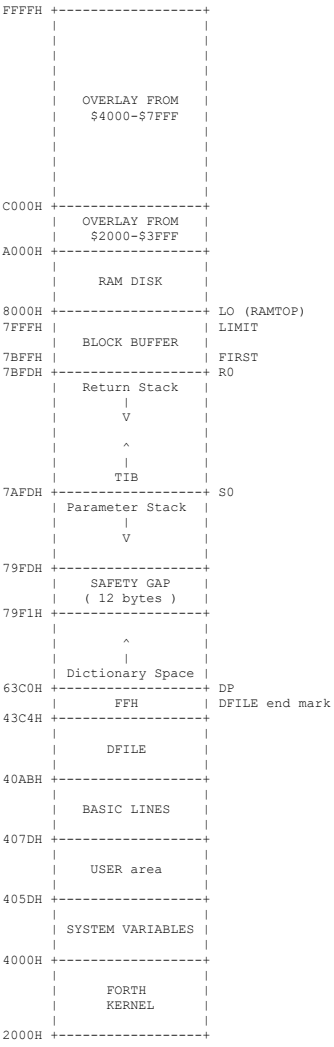
Coment to the end of line. Only be used on a screen.



Appendix B

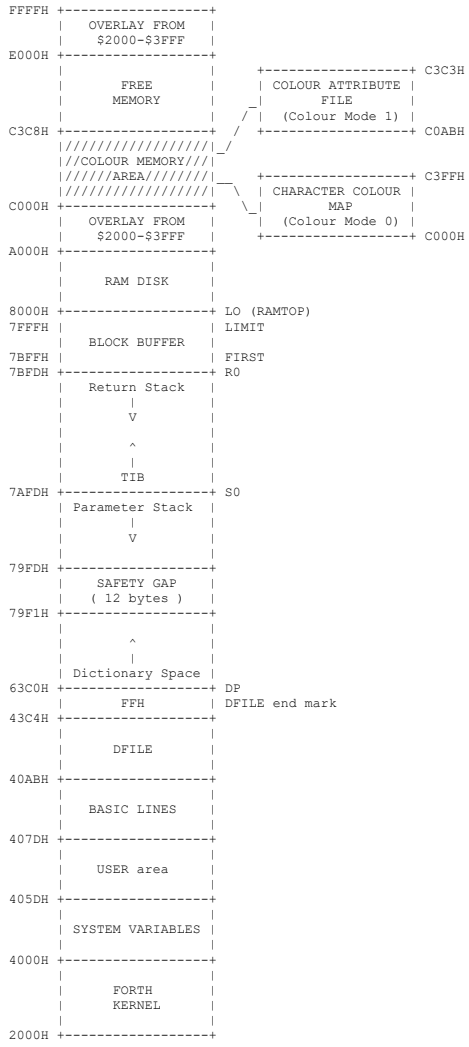
APPENDIX B - THE MEMORY DIAGRAMS

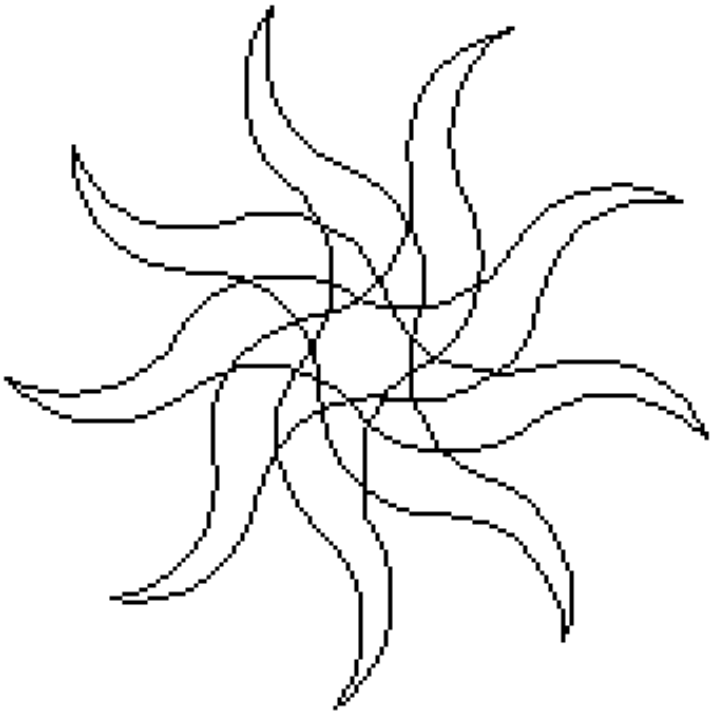
ZXPand



ZXPand + Chroma 81

(Switch 3 + Switch 6 ON)

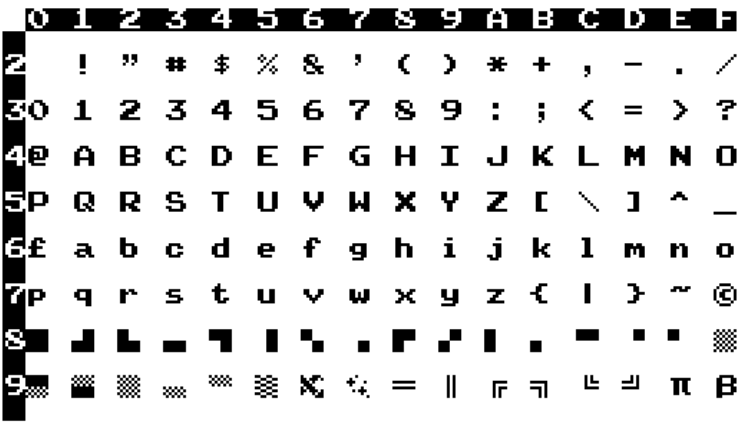




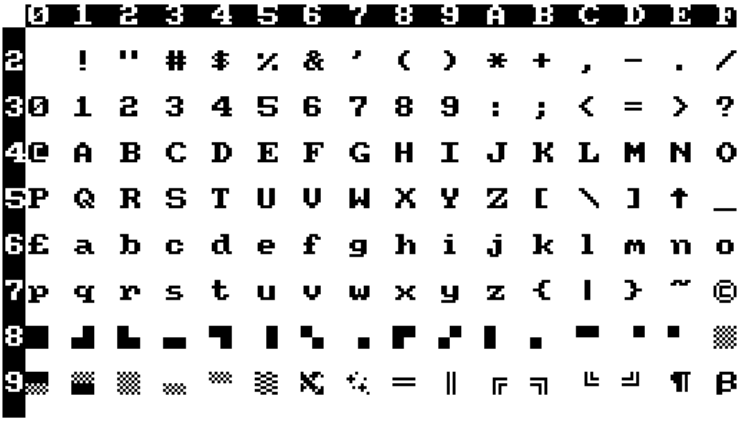
Appendix C

APPENDIX C – CHARACTER SETS

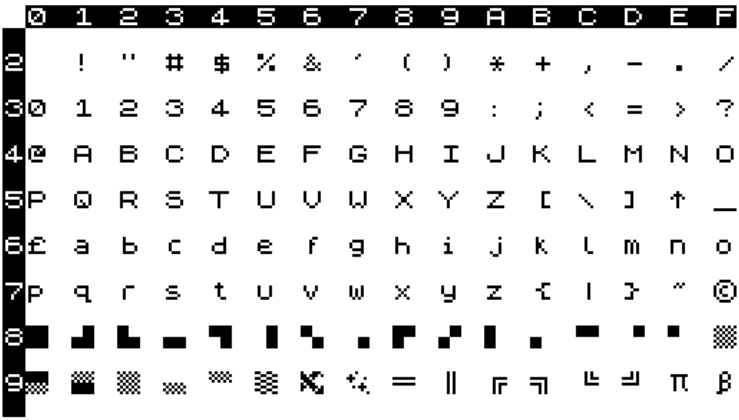
ARCADE.FN



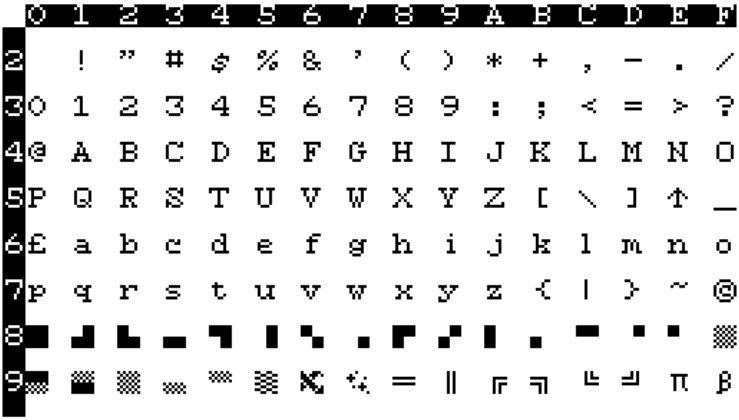
CPC.FN



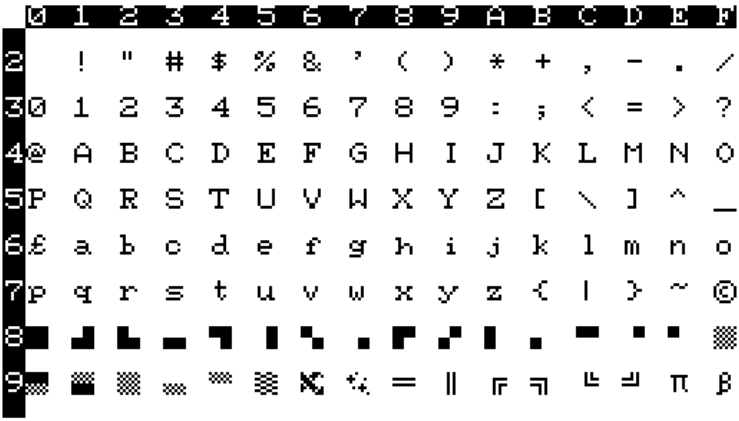
SINCLAIR.FN



SINSERF.FN



TF79.FN



TF79B.FN