The FORTH Manual

Editor, Interpreter and Compiler

for Atari ST(e)/TT/Falcon

version 0.8.3

May, 16th, 2026

With the great help of Simon Yardley for the first translation
and typo-tracking by Phil Waller.

(c) 1989-2026 Guillaume Tello

Programmed in assembler 68k
first using MadMac assembler and then
using Brainstorm's Assemble

Table of contents

Overview

The FORTH application is a single program containing an editor and an interpreter. It is a fully GEM compatible, running in any resolution and even supports MultiTOS. There is a separate compiler to create standalone programs.

At launch a window opens and the command interpreter is ready to receive your instructions. FORTH is case sensitive so makes a distinction between upper and lower case. Please use functions as described. Each instruction is separated from the others by one or more spaces. Here are some useful commands to start with:

system

exit.

edit

enter the editor, with ESC key to return back to the interpreter.

desk

returns mouse control to the AES. The FORTH window can be resized or moved, the accessories (or other applications under MultiTOS) can be called. To return to FORTH, choose the "return" option from the menu.

.

the dot displays the contents of the top of the stack (this number is then lost). This word will be used to control the results of your first steps in FORTH.

```
2 3 + .           \ displays the result of the addition 2+3
```

..

same operation as the previous word except that the number is not lost, it remains available on the stack.

```
2 3 + ..          \ prints the result of 2+3, and 5 stays on the stack
6 * .             \ then calculates 6*5 and therefore displays 30.
```

ver

displays the current running **version** (STE or TT, number and date), the **free space** to store the new word's names and their addresses.

Help key (or Shift F1)

reminds you of the main commands.

Shift Ins (or Shift F10)

access to the memory of the four last commands entered. As long as you keep Shift pressed, the lines stored are proposed to you, one second each, just release the Shift key when the desired line appears, you can reuse it, and eventually modify it.

The possibilities of FORTH:

- complete arithmetic on 4 bytes, calculations in 35 different bases.
- real calculations in the DOUBLE format of the MC68882 (also on 68000).
- variables and arrays of numbers or strings.
- structure types (struct from C) or sets (setof from pascal).
- many loops (with counter, while, until ...)
- call of the AES, VDI, TOS functions by their name (the same as in C).
- management (summary) of DMA sound and ST REPLAY16 card.
- management of the Supercharger (Nec V30 card with 1MB)
- management of the M&E PARX modules (import/export image files)
- possible calls to an external assembler

FORTH.INF file

It must be in the current directory when FORTH is launched, otherwise the default values are chosen.

The tool **FORTHINS.PRG** allows you to create and modify your INF files (the one of the interpreter and the one of the compiler). It comes with two ressources files, FORTHINS.RSC (in **french**) and FORTHINS.ENG (in **english**). If you wish to use the english version, please rename it as *.RSC.

It has 5 or 6 lines and is editable with a rudimentary word processor. The lines must be terminated with CR+LF and the information must start from the very first column:

Optional line

Allows you to set certain flags and must start with the sign # and be located at the very beginning of the file. The flags are contiguous.

Wnnn : sets the size of the Forth window to nnn% of the screen. By default it is full screen.

F : fast mode activated if present (see fast and slow)

M : permanently imposes the mouse. By default, the cursor is off if the user does not force it.

B : manages the bomb errors, if B is present then FORTH lets the system do it. (bombs and return to the desktop, useful if you are working with a debugger). If B is absent, then FORTH intercepts those errors and displays an error message.

Lnnn : forces the language code to nnn. By default, the code is found in the system ROM. For now, this is only used for the date format with **datetostr** and **strtodate**. See variable **language**.

S : ignore FORTH standard (impact on **.s**)

Annn : sets how FORTH reacts to some situations, nnn is a bitmask:
bit0 = 1 : alert if a file is overwritten by saveb/append.
bit1 = 1 : alert if a block is marked with saveb/append.
bit2 = 1 : alert if source has changed when quitting.
bit3 = 1 : compilation warnings allowed
The default is to display all alerts.

example:
```
#W80MFA2          ; 80% screen and mouse present, Fast and alert block
```

First line

Allows you to calculate the maximum size of a text in the editor, its format is:
%xxx+yyyyy : i.e. xxx% of free memory plus yyyyy KB
%xxx-yyyyy : i.e. xxx% of free memory minus yyyyyKB
For example, %0+100 will reserve 100KB regardless of free memory.
By default, %17+0 will be taken into account (1/6 of the memory).
Line in conjunction with 'edit', see editor.

Second line

Allows to calculate the maximum size of the dictionary under the interpreter, its format is the same as for the edited text:
For example, %50+100 will reserve half the memory plus 100KB (after the editor is reserved).
By default, %75+0 will be taken into account (3/4 of the memory).
Line in connection with 'free', 'full', see the dictionary and the pre-processor (function >exec).

Third line

Defines the path to the *.FOR source files.
Line related to 'saveb', 'loadb'.

Fourth line

Defines the path to *.TXT files (when exporting or importing ASCII).

Line related to 'export' and 'import'.

Fifth line

Defines the path to FORTH.HLP and FORTH.HYP ended with a slash. It can be completed, after a comma, with the full pathname to run ST-GUIDE or HypView in multitasking mode. If you work with standard TOS, this is useless as you have to load the accessories in memory.

Line related to 'help and 'guide'.

A possible file is:

```
%0+100
%50+100
F:\FORTH\SOURCES\*.FOR
F:\FORTH\TEXTS\*.TXT,F:\TOOLS\HYP105\HYPVIEW.APP
```

AUTOEXEC.FOR file

This is a file that will be loaded and compiled at start provided that FORTH.INI exists and gives the default path to the FOR files.

If you don't want it to be executed, then press one of the Shift keys while running FORTH (for example if the file leads to an error).

Example of an AUTOEXEC.FOR:

```
v_hide_c
." Version    : " ver cr
." RAM FORTH  : " free . cr
." RAM System : " -1 malloc . cr
cr
." Welcome to FORTH !"
0 v_show_c
```

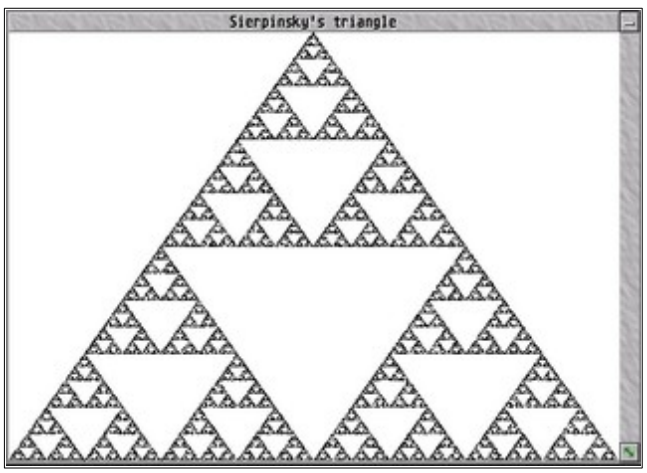
Fast Tutorial

Here is a quick introduction to all the main functions in FORTH. We will see how to use the **editor**, **save** the program, and **run** plus how to create an independent binary file with the **compiler**.

We are going to draw the **Sierpinski's triangle** in a...random way.

Imagine three people arranged in a triangle, calling a very indecisive dog by offering food to eat.

Each time the dog seems to make a choice, it walks towards one of the people, but only goes half the distance and sits down. He then allows himself to be tempted again and randomly redirects towards one of the three people, but stops halfway. If we mark with a point each place where the dog sat, the Sierpinski triangle appears!



Editing

Run FORTH (FORTH.PRG or FORTHSTE.PRG) and type:

```
edit [enter]
```

You have now entered the **editor**.

To start, let's define our variables, the three people will have their coordinates stored in two tables for the X abscissas and the Y ordinates. The dog will have two simple variables for its coordinates.

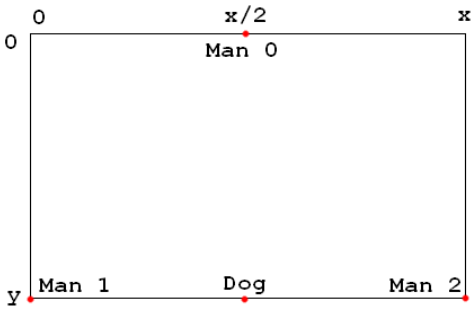
(Please Note: For more information on this mathematical coordination system see https://en.wikipedia.org/wiki/Abscissa_and_ordinate).

```
3 array MenX
3 array MenY

variable DogX
variable DogY

>comp
```

Note: >comp *will be explained later, please put it!*

Then we need to initialise these coordinates according to the size of the interpreter's window.

The AES **wind_get** function associated with **get_xyxy** will give us the coordinates of the extreme points of the window in relative mode (by default), the top point (0,0), the opposite one (x,y).



Man 0 will stand up in the center: (x/2 ; 0)
Man 1, lower left corner: (0 ; y)
Man 2, lower right corner: (x ; y)
The dog, down in the center: (x/2 ; y)

```
: init
   fastopen 4 wind_get \ fastopen returns the handle
   intout 2+ get_xyxy  \ on the stack 0 0 x y
   dup    1 MenY !     \ MenY(1) = y
   dup    2 MenY !     \ MenY(2) = y
            DogY !     \ DogY = y
   dup    2 MenX !     \ MenX(2) = x
   2/ dup 0 MenX !     \ MenX(0) = x/2
            DogX !     \ DogX = x/2
          1 MenX !     \ MenX(1) = 0
          0 MenY !     \ MenY(0) = 0
;
```

You don't have to type the comments that follow the "\", if you do, the slash must be isolated (surrounded by spaces).

Remember:

- word **"!"** is used to store a value in a variable
- **dup** duplicates the top value of the stack to use it without losing it
- **2+** and **2/** are in one word, optimized versions of 2 + or 2 / separated.

Running

It's time to test all of this! Exit the interpreter with **Esc**, then FORTH tells you that no block is marked. Type:

```
compilb [enter]
```

and your new words are ready. To verify this, type:

```
vlist [enter]
```

You should read the following list:

```
ManX ManY DogX DogY init
```

Let's try the program:

```
init [enter]
```

At this point, your variables should be filled in correctly, for example, let's visualize the coordinates of the dog:

```
DogX @ . [enter]
```

Will display the dog's abscissa in the center and:

```
DogY @ . [enter]
```

Will display its ordinate at the bottom.

```
2 MenX @ . [enter]
```

Will display the abscissa of Man 2 which must be twice that of the dog. We can edit the coordinates of men with this loop:

```
3 0 do i MenX @ . space i MenY @ . cr loop [enter]
```

Remember:

- the **"@"** pops the variable and replaces it with its content.
- the **"."** pops the top of the stack and displays it.
- array variables must always be preceded by the desired index.
- **space** displays a space and **cr** goes to the next line

Saving

If everything works, we save our source. Type:

```
saveb [enter]
```

And give the name TUTO.FOR to the program. In the event of a crash at some point, you can always restart FORTH and recover your program with **loadb**.

It's time to go back to the editor to complete the program:

```
edit [enter]
```

We are going to create the **main** procedure main which will wait on the stack for the number of points to draw. This number of points will be the argument of an **ndo..nloop** loop.

The XBIOS provides the **random** function which returns a 24-bit random number. By taking the remainder of the division by 3 with **mod**, we obtain a value 0 or 1 or 2. Exactly the necessary index for the choice of one of the three men.

Then, how to calculate the coordinates of the point where the dog will stop halfway? It's simply the average between his coordinates and the man's, so:

$$Dogx = \frac{MenX(i) + DogX}{2} \text{ and } DogY = \frac{MenY(i) + DogY}{2}$$

```
: main
   cls
   init
   ndo
      random 3 mod
      >r
      r@ MenX @ DogX @ + 2/ DogX !
      r> MenY @ DogY @ + 2/ DogY !
      DogX @ DogY @ 1 v_pmarker
   nloop
;

>comp
```

cls clear screen
init calls our procedure which fills in the coordinates
ndo takes the argument passed to main from the stack to loop
random 3 mod returns a random number between 0 and 2, let's call it i
>r sends i to the return stack (trick to avoid a variable)
r@ recalls i (without deleting it from the return stack) as an index for MenX and calculates an average which is stored again in DogX with "!"
r> recalls i (deleting it from the return stack) as an index for MenY and calculates an average which is stored again in DogY with "!"
This new position serves as coordinates for the **v_pmarker** function which plots a point.

We exit the editor with **Esc** and you can still save your work with:

```
saveb [enter]
```

Note that the program name is retained in the file selector.
We recompile with:

```
compilb [enter]
```

And then you get an error:

```
Attempt to redefine an existing word: ManX
```

FORTH has not forgotten our previous work. If we want to start from scratch, we must ask him to forget with:

```
forget ManX [enter]
```

This time compilb works.
It's time to test the program:

```
1000 main [enter]
```

A silhouette of the triangle appears. Experiment with other larger values to visualize it more accurately.

The compiler

Last step, make an independent program. We can clearly see that this source must be supplemented because for the moment, we are launching the **main** procedure by hand with different arguments.

This system needs to be automated. We will ask the user for the number of points. As long as this one is not zero, we draw a new triangle, otherwise we quit.

Here is the loop to add using the editor after the last **>comp**:

```
fastopen " Sierpinsky's triangle" 2 wind_set
begin
   ." Points : " input
   ?dup
while
   main
repeat
```

The first line gives a title to the window.
Note the **space after "** for the title, it's a standalone word!
Similarly, a **space after ."** for the prompt!
input reads a value "n" from the keyboard.
?dup duplicates n if n is not null.
while pops the top and continues while n is not null.
n is passed as an argument to **main**.

Note: remember to validate the last line so that the cursor is below **repeat**.

Let's save the program one last time with:

```
saveb [enter]
```

And let's exit the interpreter for the compiler with:

```
system [enter]
```

Launch the compiler COMP.PRG or COMPSTE.PRG. In the same folder must be located in the FORTH.LIB or FORTHSTE.LIB library. Remember to adjust your COMP.INF or COMPSTE.INF files (see the compiler at the end of the manual).

In the file selector, find TUTO.FOR and validate.
At the end of compilation, you are asked where to save TUTO.PRG, validate.

All you have to do is run TUTO.PRG from the desktop to draw your triangle, remember to put zero point to exit the program.

A brief history

In the early 80's, FORTH language sounded very interesting to me, it was implemented in the Jupiter Ace that I couldn't afford at this time.

Once independent, with a little money aside, I was able to get an Atari 520 STf with a single-sided floppy disk. It is by chance that I discover in the library of my university the manual of Forth on TO7, one of the computers manufactured by Thomson.

Quickly won over, I set to work and, in 1989, I wrote a first interpreter in GFA Basic. I quickly saw the limits in terms of speed and memory usage and going through the assembler appeared essential.

Except that... I didn't know the 68000 assembler, so far I had only worked on the TMS9900 of the TI-99.

I found a freeware: **MadMac Assembler**. A very good product from Motorola which required the use of a separate editor. My choice fell on **EDIMAX**, a super fast monochrome text editor.

At first, I did not yet have the money to afford the SM124 monitor. So I was working on a 640×200 TV set with **SEBRA**, a monochrome emulator. I must have damaged my eyes for long months.

With the books on the assembler of the Atari, that of the FORTH of the TO7 here I was launched in the programming of this peculiar language. When I reread the blocks programmed at that time, they are very clumsy, poorly optimized, and show that I did not master all the subtleties of the MC68000. I learned assembler and FORTH together.

If at the beginning it was an "old-fashioned" programming, full screen with simply the GEMDOS functions for the inputs/outputs, it was necessary to switch to a window, to use VDI and AES. The interpreter has been expanded with an editor, a compilation in RAM bringing more power, an integrated assembler (which was later replaced by calls to an external assembler), and an external compiler allowing generate a standelone program.

The passage to the Atari TT was important with the new possibilities of the processor, an extended memory and the presence of the FPU. I had done a lot of work creating floating point calculation routines on the 68000, and there everything was getting simple. But this work was not lost since it continues in the STE version.

I also abandoned MadMac for **ASSEMBLE** programmed by Brainstorm. The integrated environment was unrivaled in comfort and the speed of assembly made it possible to quickly test what I programmed.

Even today, with more than 400KB of source, it only takes 5 seconds for the TT to generate the executable.

The pack came with a debugger which quickly became indispensable. Tracing a program step by step makes it possible to remove errors.

Later I had to replace ADEBUG with **PureDebugger** which was running on

graphics card.

What is nice when you code a programming language is that any new knowledge on the machine is turned into instructions!

The discovery of SpeedoGDOS was an opportunity to manage vector fonts, the discovery of the NOVA card was an opportunity to manage TrueColor modes, the Supercharger was an opportunity to work on parallelism and most recently, the management of M&E modules of PARX has brought its own set of instructions for managing images.

We are now in 2022 and I continue my little adventure... Still with my duo ASSEMBLE and PureDebugger mainly on the TT, but also on the Apollo Vampire V4.

The development of this system will certainly continue as long as I have an Atari available.

In-line help

You have two ways to get some help:

- ✔ the **help** instructions displays syntax informations inside the FORTH window
- ✔ the **guide** instruction uses ST-Guide or HypView for a more flexible help.
- ✔ when editing, Shift+F9 displays the help for the word under the cursor

You have to set the path in FORTH.INI and copy FORTH.HLP and FORTH.HYP into this location.

If no help is available for one word, you're told so.

help

'word help : displays a quick help for the specified word. The file FORTH.HLP must be present. If a link is present, for example:

« -47 help » Other modules

Just type -47 help from the command line to go to the specified page.

- ➢ `'>fom help` : is the door to the **Forth Modules** help pages.

guide

'word guide : displays the help into ST-Guide or HypView at the desired page for the specified word. Then you can freely navigate into the whole hypertext. FORTH.HLP and FORTH.HYP must be present.

While you are browsing the help file, FORTH has turned into desktop mode using the **desk** instruction. To go back to the command line, you can top the FORTH window or use the "Return" entry from the menu.

It's better to close the accessory's window before returning to FORTH as this last has a poor window management.

If none of the two accesories is present, then **guide** acts as **help**.

If both accessories are present, then HypView is preferred.

Using **help** or **guide** on **help** or **guide** returns informations about the presence of the HYP reader in memory and how many words are supported in the help file.

Shift+F9

When using the editor, place the cursor on the word you want the help for and use Shift+F9. The help page is displayed, use Esc (or any key except "-") to exit.

If the cursor points to a space or to a number, then nothing occurs.

If there is a link, for example:

« -47 help » Other modules

Just type -47 [Enter] to go to the specified page.

Arithmetic instructions on the stack

FORTH is based on the use of a stack, for each function executed, the input parameters must first be stacked and the output parameters will be recovered from this same stack (if there are any).

1 Stack moves

Each number encountered by the interpreter is simply deposited on the stack, the functions always use as parameters the last values deposited (the top of the stack). In order to bring the desired value to the top, there are several words allowing to modify the order of the values on the stack:

dup

duplicates the value at the top of the stack.

?dup

duplicates the top of the stack if it is non-zero.

drop

discard the top of the pile.

swap

swaps the two values on top of the stack.

dropm

n dropm: throws n values from the top of the stack.

dupm

n dupm: duplicates the top n values of the stack.

rot

a b c rot: performs a circular permutation on 3 values, we obtain c a b.

roll

n roll: performs a circular permutation on n values (the first becomes the last).

over

duplicates the second value on top of the first.

pick

n pick: duplicates the value of rank n above the top of the stack.

sp!

empties the stack completely.

.s

displays the contents of the stack (The top value is indicated according to the **S** flag in FORTH.INI). The display of "*" indicates a stack underflow (too many values popped). You should immediately use sp!.

2 Basic operations

Each of these operations first pops its parameters, only the result remains on the stack.

+

a b + : provides a+b on the stack.

-

a b - : provides ab on the stack.

*

a b * : provides a*b on the stack.

/

a b / : provides a/b on the stack.

mod

a b mod: provides the remainder of the division of a by b.

/mod

a b /mod: first provides the quotient, then at the top the remainder of the division of a by b.

w/

a b w/: divides a by b with the following limits: b and the quotient must be in the interval [-32768,32767], ie fit on 2 bytes. The operation is faster than /.

w*

a b w*: provides a*b faster than * if a and b are in the interval [-32768,32767].

w/ mod

a b w/mod: with the limitations of w/, provides the quotient and remainder of a/b.

isqr

x isqr : return a fast approximation of the square root of x, with a possible error of 1 or 2.

1+ 1-

increases/decreases the value at the top by 1.

2+ 2-

increases/decreases the value at the top by 2.

2* 2/

multiply/divide by 2 the value at the top.

0<

returns 1 (true) if the top value is negative, 0 (false) otherwise.

0>

returns 1 (true) if the top value is positive, 0 (false) otherwise.

0=

returns 1 if the top value is zero, 0 otherwise.

> >=

a b > : returns 1 if a>b, 0 otherwise (idem with >=).

< <=

a b < : returns 1 if a<b, 0 otherwise (idem with <=).

=

a b = : returns 1 if a=b, 0 otherwise.

<>

a b <> : returns 1 if a is different from b, 0 otherwise.

min

a b min : returns the smaller of a and b.

max

a b max : returns the greater of a and b.

and

a b and : performs a logical AND between a and b.

or

a b or : performs a logical OR between a and b.

xor

a b xor : performs a logical EXCLUSIVE OR between a and b.

not

performs a logical NOT on the top of the stack.

negate

returns the opposite of the number at the top of the stack.

+-

a b +- : applies the sign rule of b on a:
if b<0, a is changed to its opposite
if b>0, a remains as it is.
b is popped and only remains a possibly modified.

sign

returns the sign of the to of the stack. 1 if positive, -1 if negative, 0 if zero.

abs

provides the absolute value of the top of the stack.

<seg>

x a b <seg> : returns 1 if x is between a and b (a and b included), 0 otherwise.

>seg<

x a b >seg< : returns 1 if x is between a and b (a and b excluded), 0 otherwise.

irandom

ruturn a 24 bits random number. Much faster that the Xbios' **random** function (×10 on a MegaSTE and ×30 on a TT).

irandomize

n irandomize : reinit the random generator using a value n. With the same n value, you get the same random serie of numbers, this was not possible with the Xbios' **random** function.

For a unpredictable start, you can use: `timer irandomize`

3 Work on multiple stacks

There are two stacks in FORTH, the data stack (the one used for current calculations) and the return stack. This second stack is managed internally and keeps the return addresses of a subroutine, the data of a loop, etc. This second stack can be accessed as long as it returns to its initial state before a word. using it (end of loop, end of word...)

r0

places the base address of the return stack on the data stack. This stack is descending (we stack using -(a4) and we pop with (a4)+), that is to say that the base is above the stack itself.

rp@

places the address of the return stack pointer on the data stack. This address is therefore less than r0.

>r

moves the top of the data stack to the top of the return stack. Useful for temporarily discarding a value.

r@

copies the top of the return stack onto the data stack (so the value still remains on the return stack).

r>

moves the top of the return stack onto the data stack.

rp!

empties the stack of returns (not to be done in the middle of a program, its use is reserved for cleaning if a program is interrupted prematurely during an error).

The use of this second stack is limited and can be fatal for a program. It may be necessary to create other data stacks that will no longer have the constraints of the return stack. A second stack can also be useful when debugging a program to keep values taken during execution without disturbing its progress.

&stack

n &stack XXX: creates a word XXX which will be a stack containing at most n integers of 4 bytes (a real count for 2 integers=8 bytes). On execution XXX leaves the address of its own base on the current stack.

current

XXX current: the stack XXX becomes the current stack, all operations are done in this new stack now.

stack0

FORTH stack, for example: stack0 current , allows you to return to the normal stack. This stack of 4096 bytes contains up to 1024 integers or 512 reals.

>st

x XXX >st : moves the value x from the current stack to the XXX stack.

st>

XXX st> : moves the top of the XXX stack onto the current stack.

st@

XXX st@ : copies the top of the XXX stack onto the current stack.

s0

deposits the base of the current stack onto the stack. The data stacks are descending like the return stack (we stack with -(a6) and we pop with (a6)+).

sp@

deposits the current stack pointer on the current stack , this value is less than s0.

depth

deposits on the current stack the number of integers present on this stack.

4 Real numbers

They are in double IEEE format (that of the MC68882). All calculations will be performed with this precision. To specify in FORTH that a real number is used, it must be preceded by %f. A real number uses 8 bytes on the stack, the room of two integers.

f+ f- f* f/ fneg

replace +, -, *, / and negate for reals.

f= f> f< f>= f<= f<>

replace =, >, <, >=, <= and <> for the reals.

fdup fdrop fswap fover frot fpick f>r r>f

replace dup, drop, swap, over, rot, pick >r, r> for reals.

itof ftoi

transform the integer (int) into a real (fl) or the opposite. The conversion to an integer is done by truncation.

sin cos tan

calculate the sine, the cosine, or the tangent of a real number. Angles are in radians.

sincos

simultaneously calculates the sine and cosine of an angle. The cosine is at the top of the stack.

asin acos atan

calculate the arcsine, arccosine or arctangent or a real. The return angle is expressed in radians.

pi

pushes the value of pi onto the stack.

cosh sinh tanh

calculate the hyperbolic sine, the hyperbolic cosine, the hyperbolic tangent of a real number.

asinh acosh atanh

calculate the hyperbolic arcsine, hyperbolic arccosine or hyperbolic arctangent or a real.

exp log

calculates the exponential or the natural logarithm of a real.

1/x x^2 sqr

calculates the inverse, square, or square root of a number.

x^y

x y x^y: calculates x to the y power.

fabs int frac round

calculates the absolute value, the integer part, the fractional part or the rounding of a real (the result remains in real format).

f*2^n

x n f*2^n : multiplies the real x by 2^n (n being in integer format). To multiply by 4 we use n=2, by 32 it is n=5, to divide by 2 we use n=-1 etc... This operation is extremely fast compared to f* since it suffices to carry out a addition on the exponent of the real.

fsign

returns the sign of a real number. 1 if positive, -1 if negative, 0 if zero. The returned value is an integer.

setdigits

n setdigits: sets, for real numbers, the number of digits to display. This does not affect the precision of the calculations.

fval

string fval : transforms a string of characters into a real number (the %f characters in front of the real are not necessary). Useful function in conjunction with input$ for entering real parameters.

>rad >deg

%fx >rad convert x degrees into radians.
%fx >deg convert x radians into degrees.

_fpu

variable on 1 byte for the STE version
_fpu c@ : 0 if there is no coprocessor
: %hFF if a coprocessor is present in peripheral mode. In this case, we can still set the variable to zero to return to the 68000 routines:
0 _fpu c!

Variables and memory access

1 Memory organisation in FORTH

One can distinguish three essential parts of the memory when one is under the FORTH interpreter:

1: the buffer containing the text entered in the editor (1/6 of the memory limited to 512 KB, or value set in FORTH.INF)

2: the dictionary (3/4 of the memory or value set in FORTH.INF)

3: the memory left to the system (1/4 of the memory or what remains according to FORTH.INF).

It is the dictionary that interests us here more particularly. In this area are stored (from the bottom) the variables (numerical or strings) and the definitions of new words (procedures or functions as desired). There are also rows (from the top) the dynamically allocated variables (created or destroyed during the program and whose number cannot be determined at launch).

Its operation is simple, a variable is created? The necessary space is allocated to it and the dictionary end pointer is increased accordingly. It's the same principle for a new function, a character string, etc...

here

pushes the value of the dictionary end pointer onto the stack.

top

variable containing the value of here. It is handled like a FORTH variable:
top @ : is equivalent to here.
100 top +!: Advance 100 bytes in the dictionary.

, w, c, f,

x , : places the value x on 4 bytes at the end of the dictionary. **w,** performs the same operation but on 2 bytes **c,** on one byte and **f,** on 8 bytes. Attention, with **c,** the pointer only increasing by 1, it risks no longer being even (which is essential if following is the assembled definition of a function).

even

ensures the parity of top adding 1 if it is odd.

align

n align : ensures alignment of top on a multiple of n, power of 2.

allot

n allot : allocates n bytes in the dictionary and returns the address of this zone (the old here value in fact). We could define allot like this:

```
: allot here swap top +! ;
```

allot0

n allot0 : equivalent to allot, except that the zone is initialised to 0.

free

full

return the size in bytes of the free zone (free) of the dictionary or of the zone already occupied (full).

remember

restore

remember keeps in memory the current state of the dictionary, restore allows to restore it to this state after a certain number of modifications. Restore executed without any prior remember restores the dictionary to its initial state. Attention, restore does not erase the variables created after remember, if there are any, they would therefore point to areas considered as free and risk being erased.

Those calls can be nested up to 49 times. Each restore goind back to the state of the last remember.

top>

addr top> : exchanges the value of top with that on the stack. This temporarily moves the "logical" position of the dictionary in memory. Then allows the use of "w," or "c," or "," or "f," on other areas than in the dictionary.

>top

takes the value from the stack and puts it back into top.

2 Read from and write to memory

@

addr @ : returns on the stack the long integer (4 bytes) pointed to by addr.

!

x addr ! : stores the long integer x at the address addr.

?

addr ? : displays the 4-byte value contained at address addr.

w@

c@

same principle as @ but for words (W=2bytes) or bytes (C=1byte). The sign of the values is not extended to 4 bytes, to extend it in you will need to use extwl for a word or extbl for a byte.

w!

c!

same principle as ! but with words or bytes. The values correspond to the least significant word or byte of the 4-byte integer present on the stack.

extbl

extends the sign of one byte to 4 bytes. For example 255 becomes -1.

extwl

extends the sign of a word to a long word, for example 65535 becomes -1.

extbw

extends the sign of a byte to a word. For example 255 becomes 65535.

highw

return the high word, for example %h12345678 will return %h1234.

loww

return the low word, for example %h12345678 will return %h5678.

highb

return the high byte of the low word, for example %h12345678 will return %h56.

lowb

return the low byte of the low word, for example %h12345678 will return %h78.

+!

x addr +! : adds x to the long word pointed to by addr.

-!

x addr -! : subtracts x from the long word pointed to by addr.

1+!

addr 1+! : adds 1 to the long word pointed to by addr.

1-!

addr 1-! : subtracts 1 from the long word pointed to by addr.

2*!

addr 2*! : multiply by 2 the long word pointed to by addr.

2/!

addr 2/! : divide by 2 the long word pointed to by addr.

w*!

x addr w*! : multiplies by x the long word pointed to by addr (with the limitations of w*).

w/!

x addr w/! : divide by x the long word pointed to by addr (with the limitations of w/).

3 Variables and Arrays

Not being able to work with the stack all the time, the movements would become too complex, we have to create variables to keep certain values.

variable

variable XXX : creates a word XXX which will be a variable on 4 bytes. On execution XXX simply leaves the address of a 4-byte area in memory. Since it behaves like an address, all previous words are valid for XXX:

```
variable p          \ creates the variable
50 p !              \ now p=50
21 p +!             \ p=50+21=71
p 2/!               \ p=71/2=35 (in integer)
p ?                 \ displays the value 35.
```

&wvar

&wvar XXX : equivalent to variable, except that the value is on 2 bytes. It is therefore accessed with w@ and w!.

array

n array XXX : creates a word XXX which will be an array of n long integers. The word XXX needs an index on the stack, it returns the address of the element thus pointed:

```
10 array ARRAY      \ creates the array
50 3 ARRAY !        \ ARRAY(3)=50
0 ARRAY ?           \ displays the value of ARRAY(0)
84 5 ARRAY +!       \ add 84 to ARRAY(5)
```

The indices start at 0 and go up to n-1, so in an array of 10 values the indices go from 0 to 9.

&warray

n &warray XXX : equivalent to array, except that the integers are on 2 bytes, they are accessed with w@ and w!.

constant

x constant XXX : creates a word XXX which will systematically have the value x. At runtime, XXX leaves x on the stack.

table

table XXX: creates a table of constants. Note that no dimension is indicated, in fact, this table must be filled in immediately after its definition:

```
TEST table
7 , 9 , 15 , -6 , 3 ,   \ stores 5 values
0 TEST                  \ returns TEST(0), i.e. 7.
3 TEST                  \ returns TEST(3), ie -6.
2 TEST 4 TEST +         \ calculates TEST(2)+TEST(4), so 18.
```

For more details on the word "," refer to the dealing part of the dictionary.

Variables can be used to point to memory areas, here is a series of useful words to scan a block pointed to by a variable:

.b .w .l .f

size

indicates the size (.b=1 byte, .w=2 bytes, .l=4 bytes, .f=8 bytes) of the numbers to be processed with the following words (and with **uh.** and **ub.**). By default, the size is .l.

size?

return current size (0, 1, 2 or 3 for byte, word, long or float).

)@

p)@ : returns the value pointed to by p, p remains unchanged.

-(@

p -(@ : first decrements the variable p of the size specified with size, then returns to the stack the value (1, 2, or 4 bytes) pointed to by p.

+(@

p +(@ : first increments p and then returns the value pointed to by p.

)-@

p)-@ : returns the value pointed to by p and then decrements p.

)+@

p)+@ : Returns the value pointed to by p and then increments p.

)!

x p)! : stores x at the address pointed to by p, p remains unchanged.

-(!

x p -(! : decrements p then stores x at the address pointed to by p.

+(!

x p +(! : increment p then stores x at the address pointed to by p.

)-!

x p)-! : stores x at the address pointed to by p then decrements p.

)+!

x p)+! : stores x at the address pointed to by p then increments p.

Example, we want to copy 100 words from zone1 to zone2:

```
variable p zone1 p!
variable q zone2 q!
.w size
100 ndo p )+@ q )+! nloop
```

cmove

addr addr' n cmove : transfers n bytes, or words or long words or reals (depending on the value passed to size) from addr to addr'. The routine takes into account the fact that the zones can partly overlap and makes the copy either by the end or by the beginning to avoid errors. Copying is slower for bytes, and if you are sure to have even addresses and an even number of bytes to move, you might as well move half as many words (.w).

The **file ARRAYS.INC** contains general difinitions for 2D and 3D arrays of elements whose type is defined by **size**. You must load it with, for example, the default path:

```
>include "%\INC\ARRAYS.INC"
```

&ARRAY2

co li &ARRAY2 XXX
Create a 2D array with co columns and li lines.

&ARRAY3

co li pl &ARRAY3 XXX
Create a 3D array with co columns, li lines and pl planes.

Example:

```
.b size                          \ elements are bytes
10 10 &ARRAY2 MUL                \ table of 10×10 bytes
10 0 do
     10 0 do
          i j w* i j MUL c!      \ element i,j contains i×j
     loop
loop
5 4 MUL c@ .                     \ display 20, 5×4
```

4 Character strings

A problem arises for the management of strings compared to that of numbers. The result of any calculation is anyway a 4 byte integer (fixed size). So putting it on the stack is not a problem. On the other hand, the result of a string operation can be a string of 10000 bytes, the stack would then be quickly overflowed. The solution is to place on the stack only the address of the string (4 bytes), the characters themselves being stored in a string specific to FORTH (pad) or in string variables declared by the user.

FORTH strings can contain up to 32763 bytes, they will always be terminated by a final 0 ensuring compatibility with the functions of the ROM.

string

l string XXX : creates a word XXX which will be a string of maximum l bytes. At runtime, XXX leaves the address of the first character in the string.

array$

l n array$ XXX : creates a word XXX which will be an array of n strings of l bytes each. XXX needs an index on the stack, it returns the address of the first byte of the string thus pointed:

```
14 8 array$ TEST  \ 8 strings of 14 bytes
5 TEST            \ sixth string address
0 TEST 1 TEST $=  \ compares the first two strings.
```

" "

" xxxxx" : puts on the stack the address of the string enclosed by " and ". Notice the space needed between the first quotation mark and the beginning of the text. You will have to access this string only in read mode because it is temporary. If its value must be kept, it must be transferred to a string created with string.

pad

deposits on the stack the address of the string serving as buffer memory for the results of certain operations.

len

string len : returns the current size of the string.

mlen

string mlen : returns the maximum size of the string (the one passed to string for example).

$!

string1 string2 $! : copies string1 into string2.

$+

string1 string2 $+ : adds string1 after string2.

$=

string1 string2 $= : returns 1 if the strings are equal, 0 otherwise.

$<

string1 string2 $< : returns 1 if string1 is before string2 (according to ASCII order), 0 otherwise.

$>

string1 string2 $> : returns 1 if string1 is after string2 (in ASCII order), 0 otherwise.

asc

string asc : returns the ASCII code of the first character of the string.

chr$

c chr$: create in pad a string of 1 character of ASCII code c, pad is left on the stack.

str$

x str$: create in pad a string representing the number x in the current base, pad is left on the stack.

instr

string1 string2 n instr : finds if string2 is a substring of string1 from the nth character (numbering starting at 1). If the search is positive returns the position of string2 in string1, 0 otherwise.

val

string val : returns on the stack the number represented by the string in the current base.

left$

string n left$: sends in pad the first n characters of string and leaves pad on the stack.

mid$

string k n mid$: sends in pad n characters of string starting from the kth and leaves pad on the stack.

right$

string n right$: sends to pad the last n characters of the string and leaves pad on the stack.

puts

addr string puts : sends the content of the FORTH string to the addr address (with a final 0).

gets

addr string gets : retrieves in the FORTH string the bytes present from addr, until a 0 is found.

bstr

string bstr : interprets the input string as a description containing some characters codes that are not supported by the editor (under ASCII 32) or not available via the keyboard. Possible codes are :

:nn : add character with hex code nn (2 digits exactly)
:: : add character « : »
.nnn : add character with decimal code nnn (max 3 digits).
.. : add character« . »
other : just add it.

Returns in pad (put on the stack) the interpreted string and, at the top, the length of this string.

```
" 4:E3R.253" bsrt . cr type
```

Will display 4 (the new lenght) then $4\pi R^2$.
Character %hE3 being Pi and character 253 the square.

5 Sets

A set is an object whose maximum number and value of elements are predefined by the user. The whole object contains a series of indicators attesting to the presence or absence of each element.

There are two kinds of sets in FORTH, the enumerated ones (we will explicitly give the list of elements) or the segments (all the integers included between two values). Segments are created directly, enumerated assemblies are created from a reference model.

As for character strings, sets are not placed on the stack, only their address is there. We therefore once again have recourse to a predefined set "pads" in which the results of certain operations will go.

&segment

sup inf &segment XXX : creates a word XXX which will be a set that can contain all the integers of the interval [inf,sup]. It is absolutely necessary that -32769<inf<sup<32768. At start the set is empty. When executing, XXX leaves the address of the set on the stack:

addr	: inf
addr+2	: n = number of elements (sup-inf+1)
addr+4	: sequence of n bits (0=element absent, 1=present).

&(... & ... &)

&(elem1 & elem2 & ..... & elemn &) XXX: create n words (elem1 to elemn) which will be the names of the elements of the set. The model itself being called XXX. On execution XXX leaves the address of a set header on the stack:

addr	: code of elem1
addr+2	: n, number of elements.

At runtime, each elemi leaves its FORTH code on the stack. Since the words were created one after the other, their codes follow each other, so this set behaves like a segment.

&setof

model &setof XXX: creates a set whose type is modeled on the model, the one created with &(&). Initially, this set is empty and leaves at runtime the address of the following block:

addr	: code of elem1
addr+2	: n=number of elements
addr+4	: sequence of n bits.

pads

predefined set receiving the results of certain operations. Its type (inf value and number of elements) is automatically taken from the operands.

initset

e initset: gives pads the type of set e (e can also be a model), pads then contains no element.

elmt+ elmt-

x e elmt+ : adds element x to the set e.
x e elmt- : removes element x from set e.

elmtin

x e elmtin : returns 1 if x belongs to set e, 0 otherwise.

elmt@

n e elmt@ : returns on the stack the value of the nth element present in the set e, returns inf-1 otherwise (inf being the lower value of the set).

card

e card : returns the number of elements currently present in the set e, so 0 if it is empty.

set!

e e' set! : gives e' the value (and the type) of the set e. If e and e' were not of the same type (who knows why...), they become so. On the other hand, if the set e' used less memory than e, a dangerous overflow occurs.

set=

e e' set= : returns 1 if the sets e and e' are equal, 0 otherwise.

setin

e e' setin : returns 1 if set e is contained in set e', 0 otherwise.

set+ set* set- -set

e e' set+ : calculates the union of the two sets.
e e' set* : calculates the intersection of the two sets.
e e' set- : calculate their difference (elements of e not in e').
e -set : calculates the complement of a set.

For each of these four instructions, the result is sent to pads whose address is left on the stack.

Example for segments:

```
255 0 &segment BYTE           \ BYTE is included in [0,255]
255 0 do
      i BYTE elmt+            \ all even numbers
      2
+loop
255 0 do
      i BYTE elmt-            \ removes multiples of 3
      3
+loop
```

Now BYTE contains all non-multiple even numbers of 3 between 0 and 255 (i.e. 2, 4, 8, 10, ...)

```
255 0 do
      i BYTE elmtin           \ element i in BYTE?
      if
            i. space          \ if yes, we display it
      then
loop
```

This loop displays all BYTE elements.

Example for listed sets:

```
&( Mon & Tue & Wed & Thu & Fri & Sat & Sun &) WEEK
                    \ create the week model
WEEK &setof ME
WEEK &setof HER   \ two sets empty weeks
```

\ this word lists the elements of the set whose address is on the stack

```
variable p

: see
      p!                                \ address in p
      sun 1+ mon do                     \ loop from mon to sun
            i p @elmtin                 \ element present?
            if i word$ type space       \ if yes displays it
            then
      loop
;

list
      Mon Tue Thu Fri Sat
dolist
      i ME elmt+
lloop
ME see                                  \ my working week

list
      Tue Wed Thu Fri Sat
dolist
      i SHE elmt+
lloop
HER see                 \ hers

ME SHE set* see         \ intersection, days when both work
ME SHE set- see         \ I work but she doesn't
ME SHE set+             \ days when one at least works
-set see                \ complementary, days free for both
```

6 Real numbers

&float

&float XXX : Creates a word XXX that will be a variable for a float number in IEEE format. On execution XXX leaves the address of an 8-byte zone on the stack.

f@

addr f@ : puts the real pointed to by the address addr on the stack. Can also be used to return two long integers more quickly.

f!

x addr f! : store the real x at the address addr.

For example:

```
&float REAL                    \ creates a real variable
%f1.23 REAL f!                 \ REAL is 1.23
REAL f@                        \ returns 1.23 on the stack
frac f.                        \ fractional part 0.23 that we display
```

7 Structures

Used to group under a single name several objects of different types which in the application are logically linked. This allows them to be created, destroyed or transferred in bulk. Contrary to the other variables which are created in the lower zone of the dictionary, the structures are created dynamically in the upper zone and can therefore be erased and free up the memory space previously used.

To illustrate the instructions we are going to create a structure corresponding to the information on a person (surname, first name, age).

struct .... &name

prepares the model of a structure. This model will be in low memory, the structures created by it will be in high memory. Between struct and &name are the definitions of the components of the structure:

```
structure
      40 string NAME          \ name over 40 characters
      20 string FIRSTNAME     \ first name out of 20 characters
      variable AGE            \ age is a numeric variable
&name HUMAN                   \global model name.
```

To access a component of the structure , you must first deposit the address of the structure and then the name of the component.

screate

model screate : creates a structure using the model (this model can be replaced by another structure corresponding to the same model). returns the address of this structure on the stack.

```
variable p
HUMAN screate p !             \ p=address of a new human

" DUPONT" p @ NAME $!         \ his name
" PETER" p @ FIRSTNAME $!     \ his first name
34 p @ AGE !                  \ his age

\ this word displays the components of the structure whose address is
\ passed on the stack.

: see
      dup NAME type cr
      dup FIRSTNAME type cr
      AGE ? cr
;

p @ see                       \ displays DUPONT, PETER, 34.

variable q
HUMAN screate q !             \ another human
```

s!

s1 s2 s! : transfers the content of structure s1 to s2.

```
p @ q @ s!          \ now Pierre dupont is duplicated
q @ see             \ displays the same data
15 q @ AGE +!       \ modifies the age by adding 15 years.
p @ see             \ the 34 year old youngster
p @ see             \ the same at 49 years old.
```

sdelete

addr sdelete : frees the memory used by the structure located at address addr. This location can be reallocated later.

```
p @ sdelete         \Peter Dupont no longer exists.
q @ sdelete         \ his old double either
```

sizeof

s sizeof : provides the memory size of a structure.

We see that these structures used in this way are not necessarily simple, in fact, a different variable is needed to keep the address of each structure. However, their interest is to be created or destroyed at will, therefore without prior limitation of number, which is in contradiction with the use of variables which are in known number (since declared by name one by one).

We then resort to chains of structures. Each element containing the address of another element. It is enough to keep in a variable the address of the first element and one can thus traverse a chain of indefinite size by using only one variable!

```
\ this word will be used to fill in the three fields of a human

: fillhuman
      >r            \ structure address on return stack
      r@ AGE !
      r @ FIRSTNAME $!
      r> NAME $!
;

\ this variable will contain the address of the beginning of the chain

variable p
0 p !                   \ p=0, empty chain.
```

pchain

p model pchain : creates and inserts a structure copy of the model at the head of the chain, this chain being pointed to by the variable p. After this function, p points to the new element and this element points to the old start of the chain.

```
p HUMAN pchain      \ p points to a new element, this one
                    \ pointing to 0 (before p contained 0!)
" Smith" " Peter" 34 p @ fillhuman        \ initialise this guy
```

```
p HUMAN pchain    \ p points to a new one, this one points to
                  \ Peter Smith and Smith on 0.

" Tom" "Uncle" 95 p @ fillhuman          \ initialise new
```

next

s next : provides the address of the structure that follows the s structure in the chain. Returns 0 if this is the end.

```
p @ see                 \ shows the first element (Uncle Tom)
p @ next see            \ shows the second (Peter Smith)
p @ next next .\ prints 0, that's the end.

\ this word displays all the elements of the string pointed to by p

: all
      p @               \ the start
      begin
            ?dup        \ duplicate if not null
      while
            dup see cr  \ if not null, print and line break
            next        \ next address
      repeat
;
```

punchain

p addr punchain : removes from the chain pointed to by p the element whose address is specified. As with sdelete, the memory is then freed.

pinchain

p addr pinchain : adds at the head of the chain pointed to by p the element already existing in memory pointed to by the address addr. This will point to the old chain header. (This element will have been created by screate for example).

pempty

p pempty : completely empties the chain pointed to by p, all the elements are destroyed and p=0. If we simply want to create a new string p=0 is enough (moreover pempty would be an error, because p would initially have any value and not a pointer to a structure).

All operations have so far been carried out at the head of the chain. This will not always be desirable, so here are the instructions needed to achieve the same results in the middle of the chain:

chain

addr chain : addr is the address of a structure inserted in a chain. Chain creates a new structure similar to the previous one and inserts it into the chain right after.

This function returns the address of the new element (to initialise it for example).

unchain

addr1 addr2 unchain : removes the addr2 element from the chain, also frees the memory it occupied, for this purpose addr1 must be an element preceding it in the chain (not necessarily the one just before) in order to perform the unchaining correctly.

inchain

addr1 addr2 inchain : inserts the already existing element pointed to by addr2 just after the one pointed to by addr1. The latter must be part of a chain.

empty

addr empty : empties the chain starting from the element pointed to by addr, this one not included, it then becomes the last of the chain (its pointer at 0) and the memory space occupied by its following is again available.

islast

addr islast : defines the element pointed to by addr as being the last one by canceling its pointer. Unlike empty, no element is deleted. This can be used to start a chain of which (addr) would be the first element (without using a variable).

previous

addr1 addr2 previous : search in the string for the element preceding addr2 (therefore the one that points to addr2). For this purpose, addr1 must be the address of an element preceding addr2 (most often the head of the list). If the start of the string is stored in a variable p, we use:

```
p @ addr2 previous      \ p @ giving the address of the first element.
```

To achieve this, the program runs the whole chain until it finds the element whose pointer is addr2. If this element exists, its address is left on the stack, otherwise we obtain 0 (this is the case if addr2 is itself at the head of the chain, this value 0 therefore corresponds to an "end" of the chain traversed at towards).

settable

-1 settable : initialises the table while keeping its size. The upper part of the dictionary is considered free again.

n settable : (n>0), creates a new table of n bytes and initialises it, the upper part of the dictionary is considered free again.

This table is used to manage allocations and deallocations of structures at the top of the dictionary. To illustrate its principle, suppose that we create 1000 structures in a row , they form a compact block. The table will therefore contain a single pointer (4 bytes) to an occupied block, a single pointer (4 bytes) to a free block (the rest of the memory) and an end of table indicator (-1 on 4 bytes). The table will then require only 12 bytes.

Suppose now that we delete a structure in the middle of the block. The table will then be:

pointer to an occupied block	: those before the deleted structure
pointer to a free block	: the one just deleted
pointer to an occupied block :	: those after the structure
pointer to a free block	: the rest of the memory
end of table	: -1

The table will then request 20 bytes with one less structure. If in the end I delete exactly one structure out of two to have a maximum of holes , I will need 500 pointers on occupied blocks, 500 on free blocks and the end of table indicator: 500*4 + 500 *4 + 4, i.e. 4004 bytes.

All this depends on how your program will act. If it releases the structures in the reverse order (or the same order) that it creates them, 20 bytes will be more than enough, if on the other hand the game of memory allocations and releases is wilder it will be necessary to provide at least 4 times the number number of structures you expect to create, plus 4 bytes for the terminator.

By default, the table is 1028 bytes long, which makes it possible to manage up to 256 elements and to arrive at the worst (freeing one out of two). Anyway, when a structure is created, we try to attach it to an already existing block, so this does not increase the size of the table (since the number of blocks is the same!), sometimes this new element allows to glue up two blocks, so the table is even reduced!

*The previous words are adapted to the structures created by FORTH itself (it deals with pointers, allocations and deallocations of memory, etc... But it is possible that the data is already in memory and that we need to access them in a clear way using structures.Let us think in particular of the *.RSC files in which each object is defined by a structure.*

Since in this case the data already exists, we only need a structure header whose data address will be set by the user, which is precisely what &pointer does.

&pointer

model &pointer XXX : creates the word XXX which will be a simple pointer to a structure conforming to the model. No memory space is allocated for data. To set the data address, XXX behaves like a simple variable:

```
addr  XXX !              \ sets the address
```
At runtime, XXX leaves the address of a header.

So let's go back to the example of object trees:

```
struct
      &wvar             ob_next
      &wvar             ob_head
      &wvar             ob_tail
      &wvar             ob_type
      &wvar             ob_flags
      &wvar             ob_state
      variable          ob_spec
      &wvar             ob_x
      &wvar             ob_y
      &wvar             ob_width
      &wvar             ob_height
&name OBJECT
```

Suppose now that following the loading of the file we obtain the address of the tree with rsrc_gaddr which we have stored in the constant TREE.

```
OBJECT &pointer object        \ pointer without data

TREE object!                  \ it points to the first object!
object ob_flags w@ b.         \ displays its flags
0 object ob_state w!          \ null state (normal appearance)

object sizeof 5 *             \ size of 5 objects
object +!                     \ that is added to the address of
                              \ data, it points to the 5th!
object ob_x get_xylh          \ returns its 4 coordinates
```

In practice, if the structures are glued to each other, all of the same size and starting at the ADDR address, the formula to access the data of the object of index i (indexes starting at 0) is this :

```
ADDR object sizeof i * + object !
```

For GEM objects the size is always 24 bytes, so we can replace object sizeof by 24.

8 Create a data type

We sometimes need specific data that cannot be expressed using basic types. We must therefore create a type. This type will be a word equivalent to variable or &float or array etc... which itself will create another word giving it a certain behavior.

For example, the variable word has three successive actions:

1- the following word is new, it must be created

2- reserve an area of 4 bytes

3- Give the variable the behavior of depositing the address of the 4 bytes.

The general syntax of a type will be:

```
: &TYPE <actions2> n :does> <actions3> ;end <others> ;
```

The fact that the name begins with the character & ensures action n°1, ie the inclusion of the next word in the vocabulary (like &float, or &wvar, &name...). The <actions2> are therefore the initializations (memory reservation for example, the part between :does> and ;end will correspond to the behavior of the new word, the rest, <others> will contain additional actions, i.e. nothing more often.

:does> ... ;end

n :does> : n tells how many values coming from <actions2> must be included at the beginning of the new defined word
... : everything inside is added to the definition
;end ; end of the new defined word

Let's create (or recreate) the variable type

```
: &VAR
      4 allot     \ reserve space and return address
      1 :does>    \ the address is placed in the definition
      ;end        \  variable does nothing more than giving
                  \ the address!
;

&VAR a            \ creates the variable a
&VAR b            \ same for b
```

Let's create an **array of real numbers** (like array but with 8 bytes per number), the number of elements will be passed on the stack:

```
: &farray
      8 * allot0        \ reserve 8*number of items and returns address
      1 :does>          \ address passed in definition
            swap 8 * +  \ will calculate addr+8*i
      ;end
;

5 &farray MYARRAY       \ array of 5 reals
%f5.36                  \ 5.36 on the stack
2 MYARRAY               \ address of 3rd element (addr+8*2)
f!                      \ store 5.36 at the given address

5 0 do                        \ loop from 0 to 4
      i MYARRAY f@ f. cr      \ displays element i
loop
```

Let's re-create the constant

```
: &CONST
      1 :does> ;end     \ value in definition
;

20 &CONST twenty        \ twenty will be a constant equal to 20.
0 &CONST zero           \ zero will be a constant equal to 0.
```

Let's create a matrix of integer variables

We'll use it this way: X Y &MAT xxx to create the matrix and i j xxx to access to an element:

```
: &MAT
      over w* 4 w*              \ X  4*X*Y=size to be allocated
      allot0                    \ X   addr
      2 :does>                  \ stack X and addr over i,j -> i j X addr
            >r w* +             \ (addr for later) i+j*X
            4 w*                \ 4*(i+j*X)
            r> +                \ addr+4*(i+j*X) = element address
      ;end
;

10 10 &MAT pythagore
10 0 do
      10 0 do
            i j w*
            i j pythagore !     \ fill the product table
      loop
loop

5 7 pythagore @                 \ return element 5,7 that's to say 35.
```

If you replace 4 by 8, you can create a matrix of real numbers.

Note: *multiple :does> ;end are possible within the same type provided only one pair is executed (with a conditional choice).*

::does> ;end

Same principle as :does> ;end except that with each new word creation, its definition will be assembled and not interpreted. This word will be usable in an assembled (or compiled) word.

The creation of new words

1 Words and the dictionary

A FORTH program can be broken down into several subtasks, each receiving a name. When this new word is created it can be used like a standard word, it takes its parameters on the stack and returns the results there. It can also use variables, it's up to you, or have no parameters.

The definition of each word is stored in the dictionary, like a variable. The dictionary is a simple LIFO (last in, first out) stack . That is to say that a created word can only use those already created before it (or itself, as FORTH is recursive). Second effect, when a word is erased, FORTH has to to erase all the words created after it as well.

There is nevertheless a way (ugly, but what's the point?) of making "forward" calls to words not yet created.

When it is created, a word receives a FORTH code (this is its sequence number in the series of words created), this code will be useful for certain functions but is above all essential for the interpreter.

: ... ;

: XXX ... ; this creates a new word XXX, its definition corresponds to everything between XXX and ;. For example, here is a word that adds 65 to the top of the stack:

```
: plus65
      65 +
;
Therefore:
15  plus65  .           \ displays 80 (65+15)
```

:: ... ;

:: XXX ... ; Same principle as the previous pair, except that the execution of this word will be faster.

When the text of a definition is sent to the interpreter, the latter replaces each FORTH word with its code (on 2 bytes). During execution, the interpreter reads the codes one by one and jumps to the assembler routine (for standard words) or to another definition (for user words). To speed up this process, you can simply replace the sequence of codes by the assembler routines themselves, thus avoiding the cycle of reading-code/searching for the address/jumping to the sub-program. This is what does :: ; thus improving the pair: ;. But then why still use : ;? Firstly because not all the words are usable in the other mode, then for memory size issues. With : ; (interpreted mode) each word uses only 2 bytes, with :: ; (compiled or assembled mode) each word is an assembler routine which often exceeds 2 bytes, and sometimes the speed of a word is irrelevant. For example, imagine a word waiting for a keystroke .

Here are the rules to remember:

An interpreted word can call on any other word.

A compiled (or assembled) word can only call on standard words or other compiled words (the variables are considered as compiled).

To be able to assemble variables/arrays, it is absolutely necessary that they be defined and therefore that the code preceding :: ; is already executed. You must therefore insert >comp *in the source just before an assembly (unless your routine does not use any variables).*

find

find XXX : returns the FORTH code of the word XXX onto the stack .

adress

is an array of type array. Each element contains the address of a FORTH word (arranged in the order of their codes). Examples:

```
find system adress ?    \ display address of word system
find cls adress @       \ atke the address of cls...
find system adress !    \ ... and give it to system.
```

Next time you call system to exit the interpreter, you'll just get a clear screen...!

For standard words, the address points directly to an assembler routine. For a user word, the first code on 2 bytes can indicate its type:

140: variable (variable, &float, &setof, &pointer, &name, &stack)
141: constant
142: array
143: string
144: array$
145: table
437: compiled word (:: ... ;), rest is assembler routine
628: element of an enumerated set (&)
706: warray
Other : this is the first code of the definition of an interpreted word.

forget

forget XXX : erases from the dictionary the word XXX and all those created after it. This frees up the space previously occupied. Used on a standard word, the stability of the interpreter is no longer guaranteed.

For example forget system, no longer allows to exit the interpreter properly.

&f

&f XXX : create a word XXX that, when executed, performs a "forget everything that follows me". Useful when developing before a new compilation.

vlist

displays the list of all words created by the user .

word$

c word$: sends in pad the string of characters corresponding to the code word c, the address pad is left on the stack. This is useful in conjunction with the elements of an enumerated set.

Another example:

```
500 0 do i word$ type space loop
```

Displays the name of all FORTH words up to code 499.

fast slow

In **slow** mode, the execution of interpreted words can be stopped using the Control+Alternate+Both shift combination. The underflow of the stack is also carried out (prohibition of popping more than one had stacked). If a stop is requested, the interpreter deposits on the stack the address of its execution pointer (which points to the code of the instruction that it was going to execute).

The alert box that opens allows you to run an analysis of the return stack to help you locate the break point. As an example, this line:

```
test ⇩ [qt]
```

means that the word test was called and that the interpreter entered its definition. Upon return, [qt] will be eecuted (quit, back to the command line). This other line :

```
begin ⇨ dup
```

means that a loop starting with « begin dup » is opened. Be aware that the analysis can crash the computer if you use assembled words. You decide! Similarly, it's up to you to clean both stacks with **sp!** and **rp!**.

In **fast** mode, these possibilities do not exist, but we gain in speed.

Slow mode is the default mode with the interpreter, this can be changed in the FORTH.INF configuration file.

Fast mode is the default mode for a standelone program created by the compiler.

exec

c exec : executes the c code word. Two possible uses for this word, first the equivalent of an ON .... GOSUB from Basic:

```
table CODE
find sub0 ,
find sub1 ,
find sub2 ,       \ creates a table with the codes of 3 routines

input             \ user types 0, 1 or 2
CODES             \ returns the corresponding code
exec              \ executes sub1, 2 or 3.
```

Second usage, the forward call of a word not yet created:

```
variable CODE           \ will contain the code

: try CODE @ exec ;     \ will execute the word pointed to by CODE

: Hello ." Hi!" ;       \ the second word

find Hello CODE !       \ CODE will point to Hello

try                     \ will run the Hello word created after it.
```

exe+

n exe+: executes the word whose code is equal to that of the current word plus n. For example if n=1, will execute the next word.

```
: try 1 exe+ 2 exe+ ;   \ will execute the next two words.
: Hello1 ." Hel";
: Hello2 ." lo !" ;
try                     \ display Hello !
```

2 Using the editor

To facilitate the debugging of a program, an editor is integrated into the interpreter (you can nevertheless type all its instructions in direct mode following Ok>). The size of the editable text is fixed by the FORTH.INF file, or, failing that, to 1/6 of the free memory.

edit

enters the editor. The window is cut in two parts. A top line indicates the remaining memory size for the text of the program as well as the number of the current line under the cursor (starts at 0). At the bottom, the text of the program is displayed.

Each line can contain up to 80 characters. If the width of the window is insufficient, horizontal scrolling will be done to access the entire line. Lines starting with a definition (: or ::) appear un bold and starting with a comment (\) in light. The edited line is normal and returns to its style once it is validated.

Active keys under the editor:

F1 : start of text
F2 : half screen up
F3 : marks the beginning of the block **Shift F3** : go to block start
F4 : marks the end of the block **Shift F4** : go to block end
F5 : cancel block marking **Shift F5** : delete block
F6 : or **Ctrl+L** go to line... (use info line)
F7 : or **Ctrl+F** search for a string (use the info line)
F8 : or **Ctrl+G** search next
F9 : half screen down
F10 : end of text
Shift F6 : move block before line under cursor.
Shift F7 : copy block before the line under the cursor.
Shift TAB : if a block is marked, shift it left or right according to the Shift key. With both Shifts, this reformats the tabulations to multiples of 3.

The text search function finds only one occurrence of the string per line.
The block marks are shown with the vertical slider of the window. If this one is in the top corner facing the information line, then no block exists or it is not seen in the window. You can reverse the use of F3/F4, FORTH should put things back correctly.
The block functions are irremediable, no UNDO provided.

Help or Shift F1 : reminds you of the key usage.
Shift F9 : displays help for the word under the cursor.
arrows : up, down, right, left.
Shift Left : end of line
Shift Right : start of line
Enter : end of line and creation of a new one (breaking the old line at cursor level)
Undo or Shift F2 : cancels the modifications of the current line.
Backspace : erase while moving backwards and, if enough space, can glue together the current line with the previous one.
Delete : delete under the cursor
ControlDel : delete the current line
Tab : advance right to the next block of 3 characters.
Shift Up : half screen up, equivalent to F2.
Shift Down : half screen down, equivalent to F9.

Identation is automatic.

Esc : exits the editor. A message is then displayed indicating whether there is still a marked block or not. This is essential because the functions **saveb** , **writeb** and **compileb** are limited to the block if it exists.

If a **GEM Clipboard** is defined (generally, a CLIPBRD folder at the root of the boot disk, for example C:\CLIPBRD\ ), then the editor can access the clipboard :

Ctrl+C	: copies the block to a new SCRAP.TXT file.
Shift+Ctrl+C	: adds the block at the end of the current SCRAP.TXT file.
Ctrl+V	: insert the content of the SCRAP.TXT file at the cursor position. Those lines become the new block.

writeb

prints the selected block (if it exists) or all the text.

saveb

saves the selected block (if it exists) or all the text. The file selector is open, just enter the desired name . If a file of the same name already exists, its content is replaced by that of the editor.

According to the settings if FORTH.INF two alert boxes can appear:

- one if the file is about to be overwritten
- one if only the block is about to be saved.

export

like saveb, except that it is sent in simple ASCII format (can be reprocessed by another word processor). Erase **pads** (see sets).

append

Same principle as saveb, except if the file already exists, the editor content is appended to the old content.

loadb

loads a file into the editor via the file picker. If a text is already present, then the file is added after this text. (in mode select+) If the file is not found, a second attempt is performed forcing the extension to ".FOR".

import

like loadb, except that the file is in simple ASCII format and will be recoded before it is copied into the editor. Erase **pads** (see sets). (in mode select+) If the file is not found, a second attempt is performed forcing the extension to ".TXT".

select+ select-

determine the operation mode of the words **loadb**, **saveb, import, export, rsc2for** and **append**. In select+ mode (default), the file selector is open. In select- mode, a string containing the full pathname is expected on the stack, for example:

```
" D:\FORTH\TEST.FOR" saveb
```

compilb

sends the selected block (if it exists) or all the text to the interpreter. Executable parts are executed, word definitions are stored in the dictionary. If an error is detected it is reported and running **edit** will bring the cursor to the faulty line.

If you use assembled definitions with :: xxx ; then compilation statistics are available at global-2 and global-4:

global 4 - w@ . displays the number of optimizations related to the use of a constant as an argument of a function

global 2- w@ . displays the number of optimizations linked to the sequence of words of which one retrieves the result of the other without going through the stack.

>comp

the use of this word is reserved within the editor. As explained previously, FORTH first reads a text, turns it into codes, then executes it. Therefore, it uses an execution buffer in which it stores the instruction codes on first reading. This buffer is limited and sometimes fills up. If FORTH signals this error, it suffices to cut the executable part with a >comp which will send the data in several parts to the interpreter.

>comp cannot be in the middle of a loop or any other control structure (if, case, etc...). Therefore, the executable parts inscribed in a loop (or other) are limited in size. Fortunately, this does not affect the definitions of the new words (between : and ;, or :: and ;) since the execution buffer is not used, everything is sent directly to the dictionary.

> comp is also used in conjunction with **>export** (see pre-processor).

eval

this word behaves like compilb except that instead of fetching the text from the editor, it takes it from a string.

"string" eval : interprets the string as a sequence of FORTH commands, executes what must be executed and stores the definitions encountered in the dictionary. We can thus create a FORTH interpreter in FORTH:

```
begin
      ." Ok>" input$ eval
again
```

This loop does the same job as the direct mode of the interpreter. The string itself can contain the eval function. At each level a new layer of the interpreter is created.

lmax

2-byte variable containing the number of the last line of the editor. Modifying this variable is a little dangerous if you don't know the coding of the lines of the editor.

bstart bsize

These two variables on two bytes each contain respectively the number of the first line of the block and the number-1 of lines of the block. If bstart=-1 (or 65535) no block is defined. If the block is defined and bsize=-1 (or 65535), the block extends to the end of the text.

```
10 bstart w!
5 bsize w!        \ manually marks a block starting at line 10 and 6 lines
long.
writeb            \ prints these 6 lines.
```

buf0 buf

bufo is a constant containing the starting address of the text in the editor. Buf is a variable (4 bytes) containing the address of the end of the text in the editor.

```
buf @ buf0 - .    \ displays the text size in the editor.
```

buf!

empty editor text.

In the editor the lines are coded as follows:
2 bytes : offset for next line or 0 if last line
1 byte : 1+number of spaces at the beginning of the line
then comes the text of the line possibly completed by a space so that the following line begins at an even address.

( )

Parenthesis are only used for readability. They are ignored at compilation and absent when running.

You can use them to show a logical group, for example the parameters of a function.

```
( " D:\FOLDER\FILE.EXT" 0  )
      fopen fhd !
( fhd @ 1000 here )
      fread drop
( fhd @ )
      fclose drop
```

This cleary shows the parameters for fopen, fread, fclose.

****

Only with the editor, this sign allows you to define a string on more than one line. It must be the very last sign on the line, this works with **."** or **"** only.

The next line is appened from its first non empty character, so the tabulation is skipped. If you want a space between the two lines it must be placed before \\.

Example:

```
." This string \\
      ends here !"
```

Will display:

```
This string ends here !
```

Control structures

1 Loops

do ... loop

x y do ... loop : executes the loop with y as the starting index, the index is incremented by 1 at each loop and the loop continues as long as the index is less than x. For example: 10 2 do ... loop, the index will take the values from 2 to 9.

Note: *if x=y, then the loop is ignored.*

do ... +loop

x y do ... +loop : executes the loop with y as the starting index, the index is incremented by the value found on the stack by +loop and continues as long as the index is less than x (if the increment is positive) or as long as the index is greater than or equal to x (if the increment is negative). For example:

```
16 2 do ... 4 +loop          \ index=2, 6, 10, 14
6 18 do ... -3 +loop         \ index=18, 15, ,2, 9, 6.
```

Note: *if x=y, then the loop is ignored.*

do ... ifloop

x y do ... ifloop : executes the loop with y as the starting index. If ifloop encounters a false value on the stack (zero), the loop stops without incrementing and the value of the last index is left on the stack. If, on the contrary, the value encountered is true (other than 0), the incrementation of 1 is done and the loop continues as long as the index is less than x. The returned value is then x if the condition has always been true.

Note: *if x=y, then the loop is ignored and x is returned.*

list ... dolist ... lloop

There are two possible syntaxes:

1) list ... (values) ... dolist ... (loop body) ... lloop: The loop body is executed, the index successively taking the values dropped on the stack between list and dolist. For example:

```
list 12 14 -1 -3 dolist ... lloop   \ index=12, 14, -1, -3.
```

Notes: *if the list is empty, then the loop is ignored.*
You can't nest a 2nd dolist between list and dolist.
If list is absent before dolist, a warning is issued.

2) addr n dolist ... lloop : the index values are stored at the address addr on 4 bytes each (this is the format of an array of integers created with array), n is the number of values to take (i.e. the number of loops).

Note: *if n=0, then the loop is ignored.*

You can get the index of the current value (0 to n-1) by using **r@**, you must be at the loop level without another structure openned inside (loop, case...).

i j k

These are the names of the loop indices. When a loop is opened it is assigned the index i. If a loop is openned inside another, it takes the index i, the one immediately before takes the index j. If a third loop is open in the other two, it takes the index i, the previous one the index j, and the outermost the index k. Example:

```
102 100 do
      i. cr                         \i hundreds
      12 10 do
            i. space                \ here i becomes ten
            j. cr                   \j is the hundred
            2 0 do
                  i. space          \ here i becomes unit
                  j. space          \ j becomes tens
                  k. cr             \ k becomes hundred
            loop
            i. space this           \ i returns to ten again
            j. cr                   \ j returns to hundred again
      loop
      i. cr                         \ i returns to hundred again
loop
```

These indices are applicable to the loops presented previously.

ndo ... nloop

n ndo ... nloop : executes the loop n times without being able to use an index in the body of the loop. This loop is then faster than do/loop.

Note: *if n=0, then the loop is ignored.*

leave

allows you to exit a loop prematurely, the exit will be from the next loop, +loop or nloop.

begin ... again

The loop body will run indefinitely (unless exited with exit or quit).

begin ... until

The loop body will be executed until the value taken from the stack by until is true (other than 0). Example: begin mousek until, waits for a mouse button to be pressed.

begin ... while ... repeat

The loop contains an exit condition in the middle. We exit as soon as the condition taken by while is false (zero).

```
addr p!                 \ p points to any address
.b size                 \ we manipulate bytes
begin
      p )+@             \ returns the value pointed to by p, p incremented
      ?dup              \ duplicated if not zero
while                   \ while non-zero
      emit              \ outputs the corresponding character
repeat                  \ to next
```

This loop therefore displayed a string located at the address addr and ending with a null byte.

2 Testing

if ... then

If the condition found by if on the stack is true (non-zero), then the part between if and then is executed. If not, we jump directly to then (in FORTH "then" means next, not then).

if ... else ... then

If the condition found by if on the stack is true (non-zero) the part between if and else is executed, otherwise, the part between else and then is executed.

```
x 0>
if
      ." Positive"
else
      ." Negative or null"
then
```

Displays the x sign in full.

case of endof endcase

```
x case
      x1 of ... endof
      x2 of ... endof
      ...
      xn of ... endof
      (default part)
endcase
```

The value x found by case is successively compared to x1, x2, ... xn. As soon as there is an equality, the part between the following of and endof is executed then we jump after the endcase. If none of the xi values suits, the default part is executed. It is only in the default part that x is left on the stack (it's up to you to pop it or use it).

```
key %hff and                        \ reads a key and keep the ASCII code
case
      65 of ." This is the A" endof \ the A has ASCII code 65
      66 of ." This is the B" endof \ the B is 66
      drop                        \ other, drop key
      ."Incorrect key"            \error message
endcase
```

>of <of <>of

Allow to replace of (whose only test is equality) by other tests >of: is x greater than xi?, <of: is x less than xi?, <>of: is x different from xi?

```
x case                                  \ memorize x
      0 >of ." Positive" endof          \ x>0?
      0 <of ." Negative" endof          \ x<0?
      drop ." Null"                     \ only remaining case, x=0
endcase
```

-> and-> or-> ->.

This structure makes it possible to carry out complex tests while avoiding unnecessary calculations. For example I want to execute a subroutine only if two conditions are simultaneously true. If the first is false, the second is useless to check. Similarly, I want to execute a subroutine if either of two conditions is true. If the first is already true, the second is useless.

```
-> test1
and-> test2
and-> test3
etc..
->.
```

returns -1 (true) if all conditions are true, 0 if any is false. Each calculation must leave only one value on the stack (0=false, other=true). The most interesting thing is to place first the conditions with the least chances of being verified (according to the program) and thus avoid unnecessary tests.

```
-> test1
or-> test2
or-> test33
etc...
->.
```

returns -1 (true) if any of the conditions is true, 0 if all are false. Each calculation must leave only one value on the stack (0=false, other=true). The most interesting thing is to place the conditions with the greatest chance of occurring first (according to the program) in order to leave the structure as quickly as possible and avoid unnecessary tests.

These two structures are nestable:

```
->
      -> A and-> B ->.
or->
      C not
or->
      -> D and-> E not ->.
->.
```

This tests the condition (A and B) or (not C) or (D and not E).

3 Other commands

exit

allows you to exit a word prematurely. **exit** must not be inside a structure using the return stack (loop with do...).

quit

stops all execution and returns to the interpreter.

system

exit the interpreter.

If you have modified the program since the last **saveb** or **append**, then an alert box warns you and you can cancel this command.

This alert box can be enabled or disabled in the FORTH.INF configuration file.

Keyboard and screen

1 Numbers representation

By default, numbers are written in base ten, but FORTH allows you to modify the base, so you can use base 16 (hexadecimal) or 2 (binary) or other more exotic ones (like base 13...).

%d %b %o %h

These are not strictly speaking FORTH words, but prefixes indicating that the number to follow is in base 10 (%d), 2 (%b), 8, (%o) or 16 (%h). This does not modify the current base, only the following number is concerned:

```
%d15        \ 15
%h15        \ 21 (1*16 + 5)
%o15        \ 13 (1*8 + 5)
%b1011      \ 11 = 8 + 2 + 1
```

base

variable on 4 bytes containing the current base. The sequence:

```
base @ .
```

display 10, since whatever the base, it is written 10 in its own base... Indeed 2 is written 10 in base 2, 23 is written 10 in base 23. If you want to have the value in base 10 we must specify:

```
base @ d.
```

Similarly, if I am in base 2, the sequence is 10 base ! will leave me in base 2 because here 10 is 2. It will therefore be necessary to use:

```
%d10 base !
```

or one of the following words:

bin hex decimal

sets the base to 2, 16 or 10.

For the base to become effective as input, the sequence of instructions modifying the base and the number must be separated by an Enter (in direct mode) or a >comp in the editor. During the program, the base change is immediate (with input in particular).

%f

prefix indicating that the number to follow is a real. The format is:
prefix : %f
sign : - or + (this one is optional)
mantissa : a floating point number or not
exponent : this part is optional, it consists of:
prefix : e or E
sign : - or + (this one is optional)
value : an integer in base 10.

Facility: if a number contains a decimal point, then it is considered as a real number even without the prefix.

For example:

```
%f-2.36e5   = -236000
%f72e-6     = 0.000072
%f.5        = 0.5
8.          = 8
```

2 Keyboard entries

input

waits for a number keystroke at the current cursor position. Enter ends the entry. The number expressed in the current base is converted into an integer and placed on the stack.

For input of **real numbers**, one must use input$ fval.

input$

waits for a character string to be typed on the keyboard, Enter ends the entry. The string is sent to **pad** whose address is left on the stack.

expect

string expect : displays the current content of the string and allows you to modify it. The result string is stored at the same address (the address is popped). Using a string constant with expect is limited since its maximum size corresponds to its initial size, example:

" D:*.*" expect will only allow the introduction of 6 characters!, just enough to change the name of the drive. You must then use a normal string:

```
80 string PATH
" D:\*.*" PATH $!
PATH expect       \ up to 80 characters allowed
```

key

waits for a keystroke and returns a word containing the ASCII code in the lower byte and the SCAN CODE in the upper byte.

```
key %hff and      \ isolate ASCII code
key 256 w/        \ isolates the SCANCODE
key 256 w/mod     \ separates the two (ASCII at the top).
```

inkey

reads the keyboard "on the fly" and returns the ASCII code and the SCANCODE of the key pressed or 0 if no key is detected. Here the ASCII code is in the lower word and the SCANCODE in the upper word:

```
inkey %hff and    \ isolate ASCII code
inkey 65536 /     \ isolate scancode
inkey 65536 /mod  \ separates the two (ASCII on top)
```

?terminal

returns -1 if a character is present on the keyboard port or 0 otherwise.

3 Outputs to the screen

. f.

pops the integer off the top of the stack and displays it in the current base (for .) or the real number (for f.).

.. f..

prints the top integer (..) or real (f..) of the stack (it remains on the stack).

d. b. o. h.

force display in base 10, 2 , 8 or 16. The current base is not modified, only this display is affected.

uh. ub.

As **h.** et **b.** but for unsigned values. For example, -1 appears as FFFFFFFF. The value is displayed using exactly 2, 4, 8 or 16 hexdigits (or 8, 16, 32, 64 bits) according to **size**.

type

string type : displays the string.

." ....."

displays the string between ." and ", this is more convenient than type if the message is a string constant, in fact:

```
" xxx" type
```

is equivalent to

```
." xxx"
```

Note the space required between ." and the first character of the string.

space

displays a space at the current cursor position.

spaces

n spaces : displays n spaces at the current cursor position.

cr

skips a line and scrolls the screen if necessary.

emit

c emit : displays the c character (the control characters are simply displayed too).

cls

clears the current FORTH page and returns the cursor to the top of the screen.

4 Formatted outputs

start

empties the temporary string to prepare a new one. This string is like **pad,** a predefined area of memory.

pushes

string pushs : add the string to the right of the temporary string.

pushv

x pushv : add the number x to the right of the temporary string (it will be expressed in the current base).

pushc

c pushc : add the character c to the right of the temporary string.

display

displays the temporary string on the screen (this one is not emptied, it can still be redisplayed or even completed).

getfmt

copies the temporary string to **pad** and leaves **pad** on the stack. This is useful when this string is intended for a purpose other than screen display.

Let's use this structure to output the values of an array to the printer:

```
5 array SQUARE
5 0 do
      i dup *
      i SQUARE !
loop                    \ fills the array with the squares of the numbers
from 0 to 4 (0, 1, 4, 9, 16).

5 0 do
      start                     \ new line
      "SQUARE[" pushs           \ SQUARE[
      i pushv                   \ SQUARE[i
      " ]=" pushs               \ SQUARE[i]=
      i SQUARE @ pushv          \ SQUARE[i]=i*i
      13 pushc 10 pushc         \ carriage return and line feed
      getfmt ltype              \ send to printer
loop
```

There is another set of words for formatting numeric data (it allows display with a fixed number of digits or even fixed-point display):

<#

x <# : empties the temporary string and prepares the value x to be formatted.

#

performs the Euclidean division of x by the current base. The rest is converted to a character appended to the left of the temporary string. The quotient replaces x on the stack.

hold

c hold : adds the c character to the left of the temporary string (a comma for

example).

#s

converts x to a string and appends it to the temporary string on the left, 0 replaces x on the stack.

#>

ends the formatting, pops x, copies the temporary string into **pad** and leaves **pad** on the stack.

#null_char

modify the behavior of # when the value on the stack is zero, by default # would add a zero, we can set another character to 'fill' on the left of a number:

```
<# 48 #nulchar     \ the zero
#                  \ extracts at least one digit
32 #nul_char       \ space for continuation
# # # # #          \ outputs another 5 digits with a blank as soon as the
                   \ number becomes zero
#>
```

Will avoid writing 000231 for the number 231 while keeping 6 characters (facilitates layout).

Let's assume that we calculate in cents and that we would like to display in dollars (thus with two digits after the comma), here is the procedure to follow:

```
:  dollars
      <#                \ startup
            #           \ strips the units digit from cents
            #           \ that of tens of cents
            44 hold     \ add a comma
            #s          \ the rest is displayed before the comma
      #> type           \ get the string and display it
;
```

4590 dollars will display 45.90
12 dollars will display 0.12

We want to display a 16-bit binary number with all its digits (possibly leading zeros):

```
: 16bit
      <#                \ prepare
      16 ndo # nloop    \ extracts 16 digits
      #> type           \ end and display
;
21 16bit          will show         0000000000010101
-1 16bit          will display      1111111111111111
```

TOS

1 GEMDOS

All GEMDOS functions return a value, if the operation went wrong this value is negative and corresponds to one of the error codes listed below:

0	everything is fine	>0	parameter returned
-32	bad function	-49	no more files
-33	file not found	-58	already locked
-34	path not found	-59	bad unlock request
-35	more handles	-64	limits exceeded
-36	access denied	-65	internal error
-37	bad handle	-66	bad program format
-39	out of memory	-67	block resizing error
-40	bad address block	-80	too many symlinks
-46	bad drive	-200	mount point crossed
-48	cross device rename		

fcreate

" file" attribute fcreate : creates a file and returns the handle.

fopen

" file" mode fopen : opens a file (0=read,1=write,2=r+w), returns handle.

fopensize

" file" mode fopensize : as fopen, but if the operation is ok, then two values are returned: the size and at the top, the handle.

fclose

handle fclose : closes a file.

fread

handle n addr fread : reads n bytes on the file and places them at the address addr. Returns the number of bytes read.

fwrite

handle n addr fwrite : writes n bytes on the file located at address addr. returns the number of bytes written.

int>

x handle int> : writes the integer x to the file (4 bytes).

str>

" string" handle str> : writes the string to the file, with FORTH format.

line>

" string" handle line> : writes the string to a text file, in text format adding CR, LF as the separator.

flt>

%fx handle flt> : writes the real x to the file (8 bytes).

set>

addr handle set> : writes the set pointed to by addr on the file.

struct>

addr handle struc> : writes the structure pointed to by addr on the file.

>int

handle >int : read an integer from the file and push it onto the stack, then at the top 1 (ok) or 0.

>str

string handle > str : reads a string on the file in FORTH format and places it in the specified string, returns 1 (ok) or 0.

>line

string handle > str : reads a text line in text format and places it in the specified string, returns 1 (ok) or 0. (Separator can be CR, LF or zero). If the line doesn't fit in the string then it is cut, you can still get the remaining bytes with another call to >line.

>flt

handle >flt : reads a real from the file and pushes it onto the stack, then returns 1 (ok) or 0.

>set

addr handle >set : reads a set on the file and places it in addr (address of a set of the same type) and returns 1 (ok) or 0.

>structure

addr handle >struc : reads a structure from the file and places it in addr (address of a structure of the same type) and returns 1 (ok) or 0.

fseek

n handle fseek : moves the file pointer by n bytes (relative) and returns the new position relative to the beginning of the file.

fseek>

n handle fseek> : places the pointer n bytes (>0) from the start of the file, returns n.

fseek<

n handle fseek< : places the pointer n bytes (>0) from the end of the file, returns n.

fdelete

" file" fdelete : deletes the file and returns 0 if ok.

fgetdta

fgetdta : returns the address of the current DTA:

addr	: 21 bytes reserved
addr+21	: attribute
addr+22	: hour
addr+24	: date
addr+26	: size
addr+30	: name (14 bytes terminated by a null).

fsetdta

addr fsetdta : sets the DTA to the specified address (does not return anything).

dta

dta : places on the stack the address of a 44-byte zone that serves as a DTA, FORTH sets is by default at start.

ffirst

" path" attribute ffirst : searches for the 1st file corresponding to the given mask. returns 0 if found and fills the DTA, error code otherwise.

fnext

fnext : searches for the next file, returns 0 and fills the DTA.

dir

" mask" dir : displays the files corresponding to the mask. By default, the name and size are displayed. If you want the date and the time, just add d and/or t in brackets at the beginning of the mask.

```
" (dt)*.PRG" dir \ display name, size, date and time of the PRG files
```

dsetdrv

d dsetdrv : sets drive d, returns a bit mask of existing drives.

dgetdrv

dgetdrv : returns the current drive number (A=0).

dsetpath

" path" dsetpath : sets the current directory and returns 0.

dgetpath

string n dgetpath : returns in the string the current path of drive n and returns 0.

dfree

addr n dfree : returns information about the drive n :
- addr : free clusters
- addr+4 : total clusters
- addr+8 : bytes per sector
- addr+12 : sectors per cluster

dcreate

" folder" dcreate : creates a folder and returns 0.

ddelete

" older" ddelete : deletes a folder and returns 0.

frename

0 " old" " new" frename : renames a file and returns 0.

fattrib

" file" f attribute fattrib : modifies (f=1) or reads (f=0) the attribute of the file, the attribute is returned.

fduplic

n fduplic : gives device n a normal handle, returns the handle.

force

n handle fforce : redirects device n to a file.

malloc

n malloc : reserves n bytes and returns the block address. If n=-1 returns the maximum size available.

mfree

addr mfree : frees the block allocated to the address addr, returns 0.

mshrink

0 addr n mshrink : shrinks the existing bloc at addr to n bytes.

cauxis

cauxis : returns 0 if serial buffer empty, %hFFFF otherwise.

cauxin

cauxin : waits for a character on the serial port and returns it on the stack.

cauxos

cauxos : returns 0 if serial port ready, %hFFFF otherwise.

cauxout

c cauxout : write the character c to the serial port.

cprnos

cprnos : returns 0 if printer not ready, %hFFFF otherwise.

cprnout

c cprnout : send the character c to the printer port and return -1 if ok or 0 if error.

cconis

cconis : returns 0 if keyboard buffer is empty, %hFFFF otherwise.

crawio

c crawio : reads (c=255) a character from the keyboard and returns its SCANCODE+ASCII on the stack or (c<255) displays the character c and returns a dummy value.

cconws

" string" cconws : displays the string.

conrs

addr cconrs : enters a keyboard string of n bytes maximum and returns the characters in addr+2:

addr	: n (must be filled before input)
addr+1	: n'=nbr of characters read (output)
addr+2	: n' bytes.

pterm0

ptermo : end a program, this is equivalent to **system**.

pterm

n pterm : call **system** and return n to the parent process.

ptermres

p n ptermres : call **system**, return n to the parent and keep p bytes from the base page in memory.

pexec

mode " name" " command" " envir" pexec : loads and/or executes a program depending on the mode.

super

super : switches to supervisor mode.

user

user : return to user mode.

tgettime

tgettime : returns the current system time with the 16 bits GEMDOS format: hhhh hmmm mmms ssss, the seconds runs from 0 to 29 and must be doubled from 0 to 58.

tgetdate

tgettime : returns the current system date with the 16 bits GEMDOS format: dddd dmmm myyy yyyy, the year from 0 to 63 must be added to 1980.

tsettime

x tsettime : set system time, x is the time with the 16 bits GEMDOS format.

tsetdate

x tsetdate : set system date, x is the date with the 16 bits GEMDOS format.

timetostr

x timetostr : converts a time in the GEMDOS format to a string stored into **pad** (that is left on the stack) using the pattern HH:MM:SS.
the x value can come from **tgettime** or from the time field of the DTA when using **ffirst** and **fnext**.

datetostr

x datetostr : converts a datee in the GEMDOS format to a string stored into **pad** (left on the stack) using the pattern DD/MM/YYYY or, only if the language code is 0=US, the pattern MM/DD/YYYY.
the x value can come from **tgetdate** or from the date field of the DTA when using **ffirst** and **fnext**.
If you set bit#31 of x, then the year will be on two digits only (as the command **dir** does) :
Example :

```
tgetdate %h80000000 or datetostr type
```

strtotime

" hh:mm:ss" strtotime : converts a string to the time GEMDOS format.

strtodate

" dd/mm/yy" strtodate : converts a string to the date GEMDOS format. If the language code is 0=US, then you must provide " mm/dd/yy". If year uses only two digits it is considered as belonging to [1980;2079], but you can indicate the four digits of the year.

Exemples:

```
tgettime timetostr type             \ display system time

DTA fsetdta
" TEST.PRG" 0 ffirst 0=             \ look for TEST.PRG
if                                  \ if found
      DTA 22 + w@ timetostr type cr \ display creation time
      DTA 24 + w@ datetostr type cr \ and date
then
```

langage

word variable containing the language code taken from the ROM of forced by FORTH.INI.

ddate

" d1/m1/y1" " d2/m2/y2" ddate : returns the number of days between those two dates (positive or negative according to date1-date2). If the years are within [1980 ;2079], the century os not necessary. Else, you must provide a four-digits year.

dofw

" dd/mm/yy" dofw : returns the day of week of the given date with the following code: 0=saturday to 6=friday. In addition, **pad** is filled with the name of the day but not put on the stack.

date+

" dd/mm/yy" n date+ : returns in **pad** the date n days after (or before) the given date, n being positive or negative. **pad** is left on the stack.

Examples:

```
tgettime timetostr " 4/7/1969" ddate .
      \ how many days since 4th of july, 1969.
tgetdate datetostr 1000 date+ type
      \ the date 1000 days ahead.
" 8/5/1945" dowf . Space pad type
      \ code and name of the day of march, 8th, 1945.
```

unpackdate

" dd/mm/yy" unpackdate : return day, month and year on the stack (year at the top). The year format is preserved (2 or 4 digits).

packdate

dd mm yy packdate : return in pad (put on the stack) the string corresponding to the specified date.
The year format is preserved (2 or 4 digits).

ARGC

return the number of arguments passed on the commad line. If only the current process'name is available, then ARGC = 0. This value is limited to 63.

ARGV

array of strings containing the different arguments passed on the command line. Especially:

o ARGV returns the full path and name of the current process (except if it is an accessory or a program launched from the AUTO folder).

Example, if a FORTH program is called this way:

```
C:\TEST\MYPROG.PRG -lFILE.TXT -t3
      You'll get: ARGC = 2
      0 ARGV      points to "C:\TEST\MYPROG.PRG"
      1 ARGV      points to "-lFILE.TXT"
      2 ARGV      points to "-t3"
```

FORTH and programs compiled by COMP support both the standard command line (in this case **pads** is used for ARGV[o]) and the extended protocol ARGV= (in this case, **pads** is free).

2 BIOS

As with GEMDOS, the value returned by these functions is either a parameter (positive) or an error code (negative):

0	Ok	-9	nomore paper
-1	general error	-10	write error
-2	drive not ready	-11	read error
-3	unknown command	-12	write-protected device
-4	CRC error	-14	media-change detected
-5	bad request	-15	unknown device
-6	track not found	-16	bad sectors detected
-7	unknown media	-17	insert another disk
-8	sector not found		

Standard I/O devices each have a predetermined handle:

0	PRN: parallel port
1	AUX: serial port
2	CON: keyboard, screen
3	MIDI
4	IKBD intelligent keyboard
5	screen (without control codes

bconstat

n bconstat : returns 0 if no character is available on the device n (input), -1 otherwise.

bcostat

n bcostat : returns 0 if device n is not ready to send, -1 otherwise (output).

bconin

n bconin : reads a character on device n.

bconout

n c bconout : send character c to device n.

rwabs

f addr n s d rwabs : according to f, reads or writes n sectors of unit d from sector s, the exchanges being made at the address addr.

f bit0: 0=read, 1=write
bit1: 0=consider media-change, 1=ignore

Special case addr=0, then simulates (n=0) or erases (n=1) the disk change.

mediach

d mediach : indicates the disk change on drive d, returns 0 (not changed), 1 (maybe), 2 (certainly changed).

drvmap

drvmap : Returns a bitmask indicating the connected drives.

getmpb

addr getmpb : copies the Memory Parameter Block to the address addr (12 bytes).

getbpb

d getbpb : returns the address of a block concerning unit d or 0 on error.

addr	: bytes per sector	addr+10	: 1st sect of 2nd FAT
addr+2	: sectors per cluster	addr+12	: 1st sect of 1st free cluster.
addr+4	: bytes per cluster	addr+14	: data clusters
addr+6	: sectors in directory	addr+16	: FAT 12 bits(0), 16bits(1)
addr+8	: sectors by FAT		

kbshift

m kbshift : reads (m=-1) or sets (another value) the state of the control keys:

bit0	: right shift	bit2	: control
bit1	: left shift	bit3	: alternate

setexec

n vect setexec : gives the value vect to the exception vector n.

3 The XBIOS

physbase

physbase : returns the address of the physical screen.

logbase

logbase : returns the address of the logical screen.

getrez

getrez : returns current resolution:

0: max 320x240 1: 640x200
2: 640x400 or 640x480 (on Falcon)
4: 640x480 (on TT) 6: 1280x960
7: 320x480

setscreen

l p r setscreen : sets the logical base (l), physical (p) and the resolution (r) of the screen, put -1 if a parameter is not to be changed.

setpalette

addr setpalette : sets the color palette (addr points to 16 words).

setcolor

n c setcolor : sets the color (c) of register n.

floprd

addr o d s t f n floprd : reads n sectors of track t on side f of drive d from sector s, the read bytes are stored at the address addr. returns o if Ok.

flopwr

addr o d s t f n flopwr : like floprd but in write mode.

format

un si tr se format
: formats unit un (0=A,1=B) with si sides (1 or 2), tr tracks (80 per example) and se sectors per track (9,10 or 11). With se=-9 we obtain a formatting accelerating disk access.

flopver

addr 0 d s t f n flopver : same parameters as floprd, but compares the contents of the memory to that on disk. Returns 0 if Ok, or an error code and puts the bad sectors list in addr.

mfpint

n addr mfpint : gives vector n of the MFP the address addr.

jenabint

n jenabint : activates interrupt n of the MFP.

jdisint

n jdisint : Disables MFP interrupt n.

xbtimer

tim ctrl dat addr xbtimer : launches an MFP timer.

kbdvbase

kbdvbase : returns the address of an array of keyboard routine addresses:

addr	: MIDI input	addr+20	: time routines
addr+4	: keyboard error	addr+24	: joystick routine
addr+8	: MIDI error	addr+28	: vector MIDI system
addr+12	: IKBD status	addr+32	: vector IKBD system
addr+16	: mouse routines		

midiws

n addr midiws : sends to the MIDI port n+1 bytes found at addr.

giaccess

val reg giaccess : reads (bit7 of reg to 0) or writes (bit7 of reg to 1) a value in the reg register of the sound circuit. If reading then val is without meaning, if written the returned value is meaningless.

dosound

addr dosound : sends a string of commands to the sound circuit, returns previous addr.
-1 dosound: returns current addr or 0 if sound is ended.

offgibit

mask offgibit : cancels certain bits of port A (those at 0 in mask).

ongibit

mask ongibit : sets certain bits of port A (those at 1 in mask).

setptr

mask setprt : reads (mask=-1) or sets the characteristics of the printer, returns the characteristics.

rsconf

baud ctrl ucr rsr tsr scr rsconf : configures the serial port.

iorec

n iorec : returns the address of the receive buffer of device n:

n=0 (serial), 1 (keyboard) or 2 (MIDI):

addr	: buffer address	addr+8	: last position
addr+4	: buffer size	addr+10	: bottom of buffer
addr+6	: first position to read	addr+12	: top of buffer

initmous

type param vect initmous : initialises mouse management.

keytbl

unshift shift capslock keytbl : redefines the mappings between keyboard codes and ASCII code.

bioskeys

bioskeys : restores the standard definition of the keyboard.

kbrate

delay repeat kbrate : sets the time before repeat (delay<256) and that between two repeats (repet<256), returns previous values:
bit0-7: delay bits8-15: repeat.
If both parameters are -1, only the status is returned.

random

random : Returns a 24-bit random number.

supexec

addr supexec : executes a routine in supervisor mode (routine pointed to by addr, ending with RTS, and return the value of register D0).

vsync

vsync : waits for a screen scan to complete .

nvmaccess

op start n buf nvmaccess
: read (op=0) or write (op=1) n bytes starting from the start byte (50 max bytes) from or to memory (address buf). With op=2 the Non Volatile Ram is reset.
returns 0 if Ok or a negative error code.

4 Other Functions

Unimplemented functions can be called "by hand" using the following three words:

gemdos

it is used by stacking the parameters in the dictionary as follows:

```
here              \ keep the start of the area
opcode w,         \ function number always on 2 bytes
param1 w, (if 2 bytes) or , (if 4 bytes)
param2 w, ....
value of D0 ,     \ put 0 if not used
gemdos
```

then returns the value of D0 and restores the dictionary.

bios

xbios

same order of parameters as gemdos.

VDI functions

1 Presentation

For all VDI calls there are different arrays to fill in, in response, besides graphical outputs, some parameters will be returned in the output arrays.

Input tables :

control

pushes the address of the control field onto the stack:

contro l: function code
control+2 : number of coordinate pairs in ptsin
control+4 : number of coordinate pairs in ptsout
control+6 : number of values in intin
control+8 : number of values in intout
control+10 : subfunction code
control+12 : workstation handle
control+14 to +24: depending on the function

intin

puts on the stack the address of the intin field where the input parameters will be passed.

ptsin

puts on the stack the address of the ptsin field where the input coordinates will be passed.

Output tables:

intout

places on the stack the address of the intout field in which the result of the operation is returned.

ptsout

pushes on the stack the address of the ptsout field in which output coordinates are returned.

Using FORTH in normal use, control, intin and ptsin are filled in for you. All that remains is to read any data from intout and ptsout. Explanations on the handle: it is the number allotted to the graphic station (the screen) already opened by FORTH. Again in normal use it will be of no use.

Point coordinates:

The display is done in a window whose position the user cannot predict. Also, instead of forcing the user to perform the calculations himself, there is a relative mode in which all the coordinates passed to the VDI will be shifted to be displayed only in the current FORTH window.

relative

the upper left corner of the current window or page becomes the 0,0 point. All graphic outputs are therefore shifted. This is the default mode.

absolute

the upper left corner of the screen is the 0,0 point. Graphical output may occur outside of the current page.

Manual functions:

Even if FORTH takes care of everything, it can be useful to launch a function manually (if for example it is not implemented):

handle

2-byte variable containing the handle.

vctrl

f p i sf vctrl : fills the control field with f: function code, p: number of ptsin pairs, i: number of intint values, sf: subfunction code. Handle is automatically set to control+12.

vintin!

fills the intin field taking into account the value in control+6. Values are taken from the stack.

ptsin!

fills the ptsin field taking into account the value in control+2. Values are taken from the stack but are not adjusted even in relative mode!

16b$

" string" addr 16b$: copies the string from addr passing the characters from 8 to 16 bits, returns the length of the string.

vdi

runs the function assuming control, intin, and ptsin are properly filled.

vdi!

first fills the ptsin field (with the offsets), then the intin field and finally starts the execution of the function. Control must be completed first. This is a mix of ptsin! vintin! vdi.

2 General functions:

For screen worstations, you shoudn't have to use the calls for opening or closing. Under the interpreter, a worstation is opened for you. If your program is running as a standelone application, then a call to **fastopen** does the job.

Again, if your program is run from the AUTO folder, a call to **fastopen** creates a physical workstation with **v_opnwk** and gives you access to the remaining VDI calls.

v_opnvwk

opens a virtual workstation. FORTH is designed to work with only one open station. To use this function correctly, it is therefore necessary to save the value of handle because it will be modified, then it will be necessary to restore it after the call of v_clsvwk. If you want to write to two stations at the same time, you have to juggle the two handles by correctly feeding the handle variable before each VDI call.

id l lc m mc f tc fs fp fc c v_opnvwk

id	identifier (that of ASSIGN.SYS if external station: printer=21, metafile=31...) or getrez+2 for screen.
l	line type
lc	line color
m	type of marks
mc	color of marks
f	font
tc	text color
fs	filler style
fp	fill pattern
fc	fill color
c	ccoordinate system: 2 for screen, 0 for the external ones (from 0 to 32767).

in return, handle contains the station number or 0 if the opening was not possible.

In fact, for a screen station, the easiest way is to put id=0, in this case FORTH replaces it with getrez+2 and correctly supplies handle before the call with the value returned by graf_handle.

work_out

places on the stack the address of the intout array containing the information returned by **v_opnvwk** during the opening by FORTH.

vq_extnd

m vq_extnd : returns information about the workstation, if m=0 it is the same information as that contained in work_out, if m=1 it is additional information.

screen_info / graphic_card

screen_info returns a long word corresponding to the current screen mode. This is not a VDI call but a FORTH feature.

BPP: high word:
- contains the number of bits per pixel on 16 bits
- if bit 15 is set, there was an error

SUB: low word:
- contains 16-bit bit encoding code
- if bit 15 is set, a VDI error has occurred but SUB has been determined by direct accesses to the screen memory.

graphic_card returns 0 for a mode compatible with Atari standard, else 1.

BPP		SUB	
0001	monochrome	0000	One plane
0002	4 colors	0000	Atari mode, two interlaced planes
		0001	two separate planes
		0002	2 bit/pixel, 4 pixel/byte
0004	16 colors	0000	Atari mode, interlaced planes
		0001	PC mode, separate planes
		0002	4 bits/pixel, 2 pixels/byte
0008	256 colors	0000	Atari mode, interlaced planes
		0001	PC mode, separate planes
		0002	one byte per pixel, NOVA
		0003	one byte per pixel, Matrix
0010	High Color	0000	mode Atari rrrrrggg gggbbbbb
		0001	mode PC gggbbbbb rrrrrggg
		0002	mode PC xrrrrrvv vvvbbbbb
		0003	mode PC gggbbbbb xrrrrrgg
0018	True Color 24 bits	0000	RGB, 3 bytes
		0001	BGR, 3 bytes
0020	True Color 32 bits	0000	xRGB, 4 bytes
		0001	xBGR, 4 bytes
		0002	BGRx, 4 bytes
		0003	RGBx, 4 bytes

If the program is launched from the AUTO folder, screen_info returns zero until fastopen has been executed.

vqt_devinfo

id vqt_devinfo: returns 0 if the corresponding driver is not available and 1 otherwise. id is the number preceding the driver name in the ASSIGN.SYS file.

ptsout : 0 or 1
intout : string corresponding to the name of the driver

v_clsvwk

closes a virtual workstation. You have to set handle back to the previous value.

savevdipal

addr savevdipal : saves the current palette to the VDI format (3 words 0-1000 for each color) at address specified. (Max 256 colors even in TC, so you need a maximum of 1536 bytes at addr).

setvdipal

addr setvdipal : set the palette with data found at addr in VDI format. Same limitations than with savevdipal.

pal2xbios

PALv PALx n pal2xbios
: turns a VDI palette into a XBIOS palette changing the format and the color indexes that are different.

PALv address of the VDI palette (6 bytes per color, red, green, blue from 0 to 1000)

PALx address of the XBIOS palette (2 bytes per color)

n number of colors (2, 4, 16 or 256)

- if n is positive, then STE/TT format is used with R,G,B from 0 to 15 (so colors from %h0000 to %h0FFF, to remain compatible with ST, the bits 3210 are rearranged as 0321)
- if n is negative, then ST format is used with R,G,B from 0 to 7 (so colors from %h0000 to %h0777).

v_opnwk

Same parameters as v_opnvwk. The open station here is the physical station, for the screen this call is already made by the desktop, so we only open virtual stations. On the other hand, for the printer (id=21) or a metafile (id=31) you must open a physical station in which you can work either directly or by opening several virtual stations (with different parameters). For v_opnwk, placing 0 in id is not recognized by FORTH unlike v_opnvwk.

In return, the handle variable contains 0 if there was an error or the handle of the open station if everything went well.

v_clswk

closes a physical station. The handle variable must then be restored.

v_clrwk

clears a workstation. If it is virtual, it also erases all the others depending on the same physical station. If it is a printer type, it performs a page ejection.

On opening, a station is automatically cleared.

vq_gdos

returns a value indicating the presence of GDOS or not:

-2 : not installed

other : installed and

%h5f464e54	: FONTGDOS
%h5f46534 d	: FSMGDOS or SpeedoGDOS
other	: GDOS 1, 1.1 or 1.2

vswr_mode

m vswr_mode: sets the write mode for graphical output. The parameter m can take the following values:

m=1 replacement
m=2 transparent
m=3 XOR
m=4 transparent reverse

vs_color

i r v b vs_color : sets the rates of red, green and blue for the index color i. The rates are values from 0 to 1000, the index depends on the number of colors supported.

vq_color

i f vq_color : requests the composition of the index color i. The parameter f influences the behavior of the function:

f=0 returns the values requested by vs_color.
f=1 the values are those actually taken into account.

Parameters returned in intout:

intout : index i (or -1 if the color does not exist)
intout+2 : red rate (0 to 1000)
intout+4 : green rate

intout+6 : blue rate

v_get_pixel

x y v_get_pixel : returns the color (in 16bit mode or more) or the color index of the point at the x,y coordinates.

intout : hardware index or 0
intout+2 : VDI index or RRRRRGGGGGGBBBBB.

vq_key_s

Returns the status of special keys on the keyboard. The returned value is a bitmask:

intout : bit 0 right shift
: bit 1 left shift
: bit 2 Control
: bit 3 Alternate

vs_clip

Enable or disable clipping:

x1 y1 x2 y2 1 vs_clip turns on clipping, past coordinates are those of the two opposite corners of the rectangle.

0 vs_clip disable clipping, whole screen is again accessible.

vex_timv

addr vex_timv installs a routine at address addr which will be executed 50 times per second. The address of the old routine is returned in control+10. The routine that we install should end by jumping to the old one, without modifying the registers and using only BIOS and XBIOS calls.

v_updwk

outputs the current page on the current station (the one pointed to by handle).

v_output_window

x y x' y' v_output_window : same effect as v_updwk, except that only a portion is affected.

v_form_adv

Same effect as v_updwk but additionally performs a page ejection. The current page is not erased.

v_pgcount

n v_pgcount : produces n+1 copies of the same page on a Laser station.

v_clear_disp_list

clears the current page without outputting it to the open station.

3 Text outputs

vst_load_fonts

loads the fonts listed in ASSIGN.SYS if GDOS is present.
intout: number of fonts loaded.
the first font loaded has an index of 2, the last n+1. Fonts must be loaded for each opened station.

vst_unload_fonts

frees the space occupied by fonts loaded using vst_unload_fonts.

vst_charmap

m vst_charmap: selects the use of ATARI standard strings (m=1) in 8 bits or Bitstream strings (m=0) in 16 bits per character. This call must precede the use of v_ftext16, v_ftext_offset16 and vqt_f_extent16.

vqt_get_table

returns the address of an array of 7 addresses pointing to correspondence tables between the ATARI font and the Bitstream fonts:
intout: address (long word). Each table contains 224 words indicating the 16-bit Bitstream indexes of ASCII characters 32 to 255.

v_gtext

x y " string" v_gtext : outputs the string at the specified x,y position.

v_ftext

x y " string" v_ftext : outputs the string at the specified x,y position using a vector font (SpeedoGDOS).

v_ftext_offset

addr_off x y " string" v_ftext_offset : same principle as v_ftext, except that the address addr_off points to an array of offsets for each character to be output. A couple of offsets is a pair of relative coordinates on 2 bytes each .

v_ftext16

x y addr v_ftext16 : same principle as v_ftext except that the address addr points to a string whose each character uses 16 bits instead of 8 (See vst_charmap).

v_ftext_offset16

addr_off x y addr v_ftext_offset16 : same principle as v_ftext_offset16, except that the characters are on 16 bits each (See vst_charmap).

v_justified

x y "string" l w c v_justified : outputs the string at the specified x,y position. The other 3 parameters influence the length of the text on the screen:

l : imposed total length (in screen points)
w : modify space between words (1:yes, 0:no)
c : modify space between characters (1:yes, 0:no)

vst_height

h vst_height : selects the character size in pixels. In return, we obtain the values actually taken into account:

ptsout : character width
ptsout+2 : character height
ptsout+4 : width of a cell
ptsout+6 : height of a cell.

vst_point

h vst_point : selects the character size in points (1/72 inch). If the requested size is not listed in EXTEND.SYS, the one immediately below is chosen. The parameters returned are the same as for vst_height.

vst_arbpt

h vst_arbpt : selects the size in points of a vectorial font (under SpeedoGDOS). The value of h is not limited to values present in ASSIGN.SYS. Returns the same parameters as vst_height.

vst_arbpt32

hi hf vst_arbpt32 : same effect as vst_arbpt except the size is in fix31 format separated into integer (hi) and fractional (hf) parts.

vst_setsize

l vst_setsize : sets the width in points of a vector font independently of its height (vst_arbpt). Returns the same values as vst_height.

vst_setsize32

li lf vst_setsize32 : same effect as vst_setsize except width is in fix31 format separated into integer part (li) and fractional part (lf).

vst_rotation

a vst_rotation : selects the writing angle in 1/10° of a degree. We find in intout the selected value:

intout : angle actually taken into account.

vst_skew

a vst_skew : sets the skew angle in 1/10° of a degree for a vector font . You can thus obtain italics to the left (values -900 to 0) or italics to the right (0 to +900).

intout : inclination taken into account.

vst_font

id vst_font : selects a font. Number 1 is always available (system font). The identifier is a personal number of the font which is returned by vqt_name.

intout : identifier of the selected font.

vqt_fontheader

addr vqt_fontheader : sends a copy of the header of the current font under Speedo to the addr address (minimum 421 bytes), also returns in **pad** the full name of the file corresponding to this font. **Pad** is placed on the stack.

vst_color

i vst_color : sets the color for text outputs.

intout : selected color index.

vst_effects

m vst_effects : select different text effects depending on bitmask m:

m				
	bit 0	: bold	bit 3	: underscore
	bit 1	: light	bit 4	: outline
	bit 2	: italic	bit 5	: shadow

intout : mask effectively taken into account.

vst_alignment

h v vst_alignment : specifies where in the text the x,y coordinates of the v_gtext functions or its derivatives correspond. By default x points to the left of the text and y to the bottom.

h=0	: x on the left	v=0	: y points to the writing line
h=1	: x in the center	v=1	: y points to the top of the lowercase letters
h=2	: x on the right	v=2	: y points to the top of the capital letters
		v=3	: y points to the bottom of the cell
		v=4	: y points to the bottom of the letters
		v=5	: y points to the top of the cell

intout : selected h
intout+2 : selected v

vqt_attributes

returns the set of parameters for text outputs:

intout	: font id	ptsout	: character width
intout+2	: color index	ptsout+2	: height
intout+ 4	: angle	ptsout+4	: width of a cell
intout+6	: pos. horizontal	ptsout+6	: height
intout+8	: pos. vertical		
intout+10	: writing mode		

vqt_extent

" string" vqt_extent : returns the coordinates of the rectangle containing the string taking into account all text effects:

ptsout	: x top left	ptsout+8	: x bottom right
ptsout+2	: y top left	ptsout+10	: y bottom right
ptsout+4	: x top right	ptsou t+12	: x bottom left
ptsout+6	: y top right	ptsout+14	: y bottom left

vqt_f_extent

" string" vqt_f_extent : same action as vqt_extent, except that the function applies to fonts managed by SpeedoGDOS and takes into account their specificities.

vqt_f_extent16

addr vqt_f_extent16 : same action as vqt_f_extent but the address points to 16-bit Speedo characters instead of 8 bits. (See vst_charmap).

vqt_width

c vqt_width : returns information about character c:
ptsout : cell width
ptsout+4 : available space on the left
ptsout+8 : available space on the right
intout : -1 if the character is not in the font.

vqt_name

n vqt_name : returns in **pad** the name of the font with number n.
n : font number (according to the number returned by vst_load_fonts)
intout : font identifier (keep it to pass it to vst_font).
pad is placed on the stack.

vqt_fontinfo

returns information (in pixels) about the current font:
intout : starting ASCII
intout+2 : end ASCII

ptsout : cell width
ptsout+2 : base/bottom line distance
ptsout+4 : enlargement during effects
ptsout+6 : descent/baseline distance
ptsout+8 : left offset for italics
ptsout+10 : half/base line distance
ptso ut+12 : right offset for italics
ptsout+14 : ascent/base line distance
ptsout+18 : top/base line distance

v_flushcache

empties the cache (this zone is used to store Bitmap images of proportional fonts when a size has been chosen, this avoids recalculating the screen image of a character on each call).

v_savecache

" file" v_savecache : saves the cache on disk, the string contains the full name of the file to be created.

v_loadcache

" file" mode v_loadcache : load a previously saved file into the cache. If

mode=0, the file is added to the existing cache, if mode=1, the cache is first emptied.

vqt_cachesize

n vqt_cachesize : returns in intin the size of one of the two caches. If n=0, it's the Bitmap cache, if n=1, it's the other.

intint : on 4 bytes, cache size.

v_getbitmap_info

c v_getbitmap_info: returns information about the bitmap image of character c taking into account the current text attributes:

intout	: width	intout+12	: xoff
intout+2	: height	intout+16	: yoff
intout+4	: advx	intout+20	: bitmap address
intout+8	: advy		

advx,advy and xoff,yoff are in fix31 format.

v_getoutline

c n addr1 addr2 v_getoutline : returns the information needed to reconstruct character c using Bézier curves. The addresses addr1 and addr2 point to two buffers which will receive the coordinates of the points and their flags (see v_bez), n indicates the maximum number of points that these buffers can contain.

intout : number of points returned.

vqt_advance

c vqt_advance : returns the displacement vectors for character c. The move indicates where to draw the next character from character position c (including text attributes).

ptsout and ptsout+2 : dx and dy in pixels.
ptsout+4 and ptsout+6 : rx and ry. This vector is the remainder (modulo 16384).

For a more precise rendering, it suffices to add these remainders character by character and to carry out a shift of an additional pixel as soon as 16384 is reached.

vqt_advance32

c vqt_advance32 : returns the displacement vector for character c. Compared to vqt_advance, the vector is in fix31 format (so no remainder) over two long words:

ptsout+8 : dx in fix31
ptsout+12 : dy in fix31.

vst_error

addr m vst_error : if m=1 the GDOS errors will be displayed on the screen in text form, if m=0 the messages are replaced by a 2-byte error code stored at the address addr. The application must then consult this value after a VDI call, a zero indicates that everything is fine, the other values are error codes, again the application must reset this value to zero after having read it.

vst_scratch

m vst_scratch : set the buffer allocation mode for SpeedoGDOS according to the values of m:

m=0 : sufficient size for effects on vector and bitmap fonts
m=1 : size sufficient for effects on bitmap fonts only
m=2 : reduced size, no effect on fonts.

vst_kern

tmode pmode vst_kern

vqt_pairkern

c1 c2 vqt_pairkern

vqt_trackkern

returns in fix31 format the x and y coordinates:
ptsout : x
ptsout+4 : y

4 Line functions

v_pline

x1 y1 .... xn yn n v_pline : draws a broken line connecting the n points.

vsl_type

t vsl_type : sets the line type according to the values of t:

```
t=1----------------      t=5--------
t=2------------          t=6---- -- --
t=3--     --             t=7 defined by vsl_udsty
t=4--------    --
```

vsl_udsty

m vsl_udsty : sets the line pattern when type 7 is chosen with vsl_type. For this purpose m is a 16-bit mask (1=the point is set, 0=the point is not set) which will be repeated indefinitely to form the lines.

vsl_width

w vsl_width : sets the width of the lines in pixels. An even value is rounded down to the odd number.

ptsout : selected thickness

vsl_color

i vsl_color : sets the color index for lines.

intout : selected index.

vsl_ends

s e vsl_ends : sets the styles for the beginnings and end of lines according to the values of s (start) and e (end):

0 : rectangular
1 : arrow
2 : rounded

vql_attributes

returns the current lines attributes:

intout	: type	intout+6	: start
intout+2	: color index	intout+8	: end
intout+4	: write mode	ptsout	: thickness

5 Mark functions

v_pmarker

x1 y1 .... xn yn n v_pmarker : draws a mark at each of the n points.

vsm_type

t vsm_type : sets the type of mark to use

t=1	point	t=4	square
t=2	cross (plus)	t=5	cross (multiplied)
t=3	star	t=6	diamond

Any other value will be replaced by 3.

intout : selected mark.

vsm_height

h vsm_height : sets the size of the marks (no effect on the point). All heights are not always possible, so we get in return:

ptsout : width actually taken into account

ptsout+2 : same for height.

vsm_color

i vsm_color : sets the color index for marks.

intout : selected index

vqm_attributes

returns the current mark attributes:

intout	: type	ptsout	: width
intout+2	: color	ptsout+2	: height
intout+4	: write mode		

6 Fill functions

v_contourfill

i x y v_contourfill : starts filling in x,y with the current fill color and stops:
if i=-1 : as soon as a point has another color than that in x,y
if i>0 : as soon as a point has the color i.

v_recfl

x1 y1 x2 y2 v_recfl : fills a rectangular surface according to the current attributes, the coordinates are those of two opposite points of the rectangle.

v_fillarea

x1 y1 .... xn yn n v_fillarea : fills the polygon delimited by the n points. The first and the last are automatically linked.

vsf_interior

t vsf_interior : sets the type of infill to use according to t:
t=0 use background color
t=1 use the color passed to vsf_color
t=2 use patterns (see vsf_style)
t=3 use hatches (see vsf_style)
t=4 use the pattern defined by vsf_udpat.
intout : selected type

vsf_style

i vsf_style : selects the type of hatches (1 to 12) or patterns (1 to 24) depending on the type chosen by vsf_interior.
intout : selected index

vsf_color

i vsf_color : chooses the fill color.
intout : selected color

vsf_perimeter

f vsf_perimeter : enables (f=1) or disables (f=0) contour drawing of filled surfaces. Only v_recfl ignores it, so use v_bar instead.

vsf_udpat

addr n vsf_udpat : defines a fill pattern. The parameter n defines the number of

planes (n=1 in monochrome), addr is the address where the n planes are successively stored. Each plane is a sequence of 16 times 16 bits corresponding to a square grid of 16 by 16 (32 bytes per plane). The first plane contains the most significant bits, the last the least significant bits.

vqf_attributes

returns the current fill attributes:

intout	: type (vsf_interior)	intout+6	: write mode
intout+2	: color	intout+8	: flag vsf_perimeter
intout+4	: pattern index (vsf_style)		

7 Basic objects

v_bar

x1 y1 x2 y2 v_bar : fills a rectangle.

v_pieslice

x y r a0 a1 v_pieslice : fills a portion of circle with center x,y of radius r and start and end angles a0,a1 in 1/10° of degree.

v_circle

x y r v_circle : fills a circle with center x,y and radius r.

v_arc

x y r a0 a1 v_arc : draws the contour of an arc of a circle with center x,y of radius r and limited by the angles a0,a1 in 1/10° of degree.

v_ellpie

x y rx ry a0 a1 v_ellpie : fills a portion of an ellipse with center x,y with radii rx,ry and limited to angles a0,a1 in 1/10° of degree.

v_ellipsis

x y rx ry v_ellipse : fills an ellipse with center x,y and radii rx,ry.

v_ellarc

x y rx ry a0 a1 v_ellarc : draws the contour of an ellipse with center x,y of radii rx,ry and bounded by angles a0,a1.

v_rfbox

x1 y1 x2 y2 v_rfbox : fills a rectangle with rounded corners.

v_rbox

x1 y1 x2 y2 v_rbox : draws the outline of a rectangle with rounded corners.

8 The Mouse

vsc_form

addr vsc_form : defines the appearance of the mouse, for this purpose addr is the address of a block of 37 words defined as follows:

addr	: mouse hot spot x	addr+8	: mask color
addr+2	: hot spot y of the mouse	addr+10	: 16 words for the mask
addr+4	: number of planes (1)	addr+42	: 16 words for the front
addr+6	: color		

v_show_c

f v_show_c : show the mouse cursor according to the values of f

f=0 display

f=1 consider the number of calls to v_hide_c

v_hide_c

hides the mouse cursor.

vq_mouse

returns the mouse state:

intout	: mouse button (bit0=left, bit1=right)
intout+2	: mouse x
intout+4	: mouse y

vex_butv

addr vex_butv : installs a routine called each time a mouse button is pressed. This routine receives in D0 the state of the buttons, it must restore all the registers that it uses and only call on the BIOS or XBIOS. We receive in control+18 the address of the old routine.

vex_curv

addr vex_curv : installs a routine called each time the mouse cursor needs to be drawn. The routine receives in D0 and D1 the x,y coordinates of the mouse. For the rest, see vex_butv.

vex_motv

addr vex_motv : installs a routine called each time the mouse has moved. The routine receives in D0 and D1 the coordinates x,y of the mouse, it can modify them (to slow down, accelerate or limit displacements). For the rest, see vex_butv.

9 Cursor in text mode

In "relative" mode all these operations are diverted (no real VDI call) and manage the FORTH text cursor in the current page. In "absolute" mode, the VDI call is made:

vq_chcells

indicates the number of rows and columns available in text mode

intout : lines
intout+2 : columns

v_exit_cur

exits text mode and redisplays the mouse.

v_enter_cur

clears the mouse and returns to text mode.

v_curup

one line up. If we are already at the top, nothing happens.

v_curdown

one line down. If you are already at the bottom, nothing happens.

v_curright

a position to the right, if you are already on the far right, nothing happens.

v_curleft

a position to the left. If you are already on the far left, nothing happens.

v_curhome

brings the cursor back to the top left of the window without clearing the screen.

v_eeos

erases from the cursor position to the bottom of the window.

v_eeol

erases from the cursor position to the end of the line.

vs_curaddress

l c vs_curaddress : sets the row and column of the text cursor. The coordinates start at 1 and are reduced to their maximum if exceeded.

v_curtext

" string" v_curtext : displays a string at the current cursor position, possibly displays on the next line and scrolls if necessary.

v_rvon

switch to reverse video.

v_rvoff

returns to normal video.

vq_curaddress

returns the coordinates of the text cursor.

intout : line
intout+2 : column.

vq_tabstatus

indicates the presence of a graphics tablet or mouse:

intout : 0=absent, 1=present.

v_hardcopy

starts a screenshot.

v_fontinit

addr v_fontinit : allows you to replace the system font, for this purpose addr is the address of the header of the new font (in LineA format).

10 Block functions

For these functions, the management of MFDBs (definition of blocks) has been separated. Therefore, we first define the two blocks (source and destination) then we call the desired function:

mfdbs mfdbd

returns the address of the mfdb source or mfdb destination. They are defined as this:

addr	data address	
addr+4	w	width in pixels
addr+6	h	height in lines
addr+8	w in words = int((w+15)/16)	
addr+10	f	format 0 = shifter, 1 = VDI
addr+12	p	number of planes
addr+14,+16,+18 reserved.		

fillmfdb

fills one mfdb with three possible syntaxes:

* addr w h p f m fillmfdb : fills the mfdb m with
  addr data address
  w h width and height in pixels
  p planes number
  f format
* -1 m fillmfdb : this is the screen, everything is filled for you, the data address is physbase.
* m' m fillmfdb : copies in m the data found in the other mfdb m'..

Note: for VDI block functions using the screen, you won't use -1 mfdb fillmfdb. This facility is reserved for the modules M&E management.

Here, to specify the screen, you just have to zero the data address this way:

```
0 mfdbs !   \ the source mfdb is the screen
```

Even if the width of the areas to be treated is not a multiple of 16, l must be a multiple of 16. It is therefore necessary to take the multiple of 16 immediately above the desired width for the operationsl. For example a 30x30 sprite will be allocated a 32x30 area. In 16 colors, each point requires half a byte, so 32x30/2=480 bytes necessary, on the other hand in the following copy functions we can specify l=30.

imagesize

mfdb imagesize : returns the size in bytes of the data block defined by the mfdb.

vr_trnfm

transforms the source block (with its format) into a destination block (with a different format). Normally the two blocks have the same characteristics, apart from the format.

vrt_cpyfm

copies a monochrome block to a monochrome or color block, you must fill first the two mfdb, and then:

x y x' y' l h m c c' vrt_cpyfm
  x,y top left corner in source
  x',y' top left corner in destination
  w,h size of the block to be moved (can be smaller than the size total of each block)
  m mode (1=replace, 2=trans, 3=XOR, 4=reverse trans)
  c,c' color indexes for foreground and background.

For an icon or mouse mask m=2, c=0 and c'=1.

vro_cpyfm

copies a color block to a color block, you must fill first the two mfdb, and then:
x y x' y' l h m vro_cpyfm
x,y top left corner in source
x',y' top left corner in destination
w,h the size of the block to be moved
m mode (0=delete dest, 3=replace , 6=XOR, 7=OR)
For icon data (after drawing the mask) m=7.

For example, to copy an area from screen to screen:

```
0 mfdbs !
0 mfdbd !
0 0 100 100 16 16 3 vro_cpyfm \ copies a 16x16 square from the
                          position 0.0 to position 100,100.
```

savebitmap

" file" mfdb pal savebitmap : save to disk the bitmap block described in the mfdb.
" file" : a full pathname of a file to be created.
mfdb : address of a 20-bytes mfdb (it can be mfdbs or mfdbd).
pal : address of a palette

The palette is ignored if the number of planes is greater than 8. In this case, the pixel description includes the color.
On the other hand, from 1 to 8 planes, the palette must contain the colors in VDI format (that you can obtain with savevdipal).

loadbitmap

" file" mfdb pal loadbitmap : loads a block from disk and fills the mfdb and eventually the palette.
" file" : the full path name of the file.
mfdb : address of a 20-bytes mfdb (it can be mfdbs or mfdbd).
pal : address of a palette

The palette is ignored if the number of planes is greater than 8. In this case, the pixel description includes the color.
On the other hand, from 1 to 8 planes, the palette must contain the colors in VDI format (that you can use with setvdipal).
The block itself is allocated into the dictionnary with allot and its address saved into the mfdb.

11 Bézier curves

These functions can only be used with the presence of SpeedoGDOS.

v_bez

f1 f2 ... fn x1 y1 x2 y2 .... xn yn n v_bez : outputs a Bézier curve based on n points whose coordinates are passed on the stack. For each point we must specify a flag (f1 to fn):

fi : bit0=0 : polyline (no bends)
: bit0=1 : curve
: bit1=0 : point processed normally
: bit1=1 : jump to this point without drawing. (Cut in the curve)

v_bez_fill

f1 f2 ... fn x1 y1 x2 y2 ... xn yn n v_bez_fill : same behavior as the previous function, except that the interior of the curve is filled with the current fill attributes .

v_bez_on

allows access to Bézier curves.

v_bez_off

prohibits access to Bézier curves.

v_bez_qual

x v_bez_qual : sets the quality of the curves, x goes from 0 (low quality, fast) to 100 (maximum quality, slow).

intout : the selected value.

v_set_app_buff

addr n v_set_app_buff : forces the VDI to use a specific buffer for generating curves. The buffer is defined by its address addr and the number of paragraphs n (16 bytes each). Before quitting, an application must deallocate this buffer using:

```
0 0 v_set_app_buff
```

By default, the internally allocated buffer is 8KB.

12 Metafiles

To open a metafile, handle 31 is used with v_opnwk. A file is automatically created and opened with the name GEMFILE.GEM in the current directory.

vm_filename

" file" vm_filename : allows you to assign a name different from GEMFILE.GEM to the metafile. It will nevertheless be necessary to erase (fdelete) the default file GEMFILE.GEM which will have been created by v_opnwk.

vm_pagesize

w h vm_pagesize : indicates the dimensions of the metafile page in tenths of millimeters.

vm_coords

xmin ymin xmax ymax vm_coords : sets the limit coordinates to be used on the metafile page. xmin,ymin are the upper left coordinates and xmax,ymax are the lower right ones. By default, we have:

```
0 32767 32767 0 vm_coords
```

v_meta_extents

xmin ymin xmax ymax v_meta_extents : sets, in the coordinate system chosen with vm_coords, the clipping rectangle in the metafile.

v_write_meta

elements_intin elements_ptsin n_int n_pts v_write_meta : writes in a metafile a function having a subcode created by the user. The first of the intin elements must be this subcode (those from 0 to 100 are reserved).

13 Other Instructions

v_alpha_text

" string" v_alpha_text : sends the string to the currently open station. This station must be the printer or a metafile, so it must be opened explicitly (v_opnwk) so that the handle variable is correctly populated.

v_bit_image

r x y h v " file" X Y X' Y' v_bit_image : loads an *.IMG image file from disk and displays it in the current station.

X Y X' Y' : the coordinates of the rectangle surrounding the image.

" file" : the name of the file (with path).

r : ignore (0) or consider (1) image proportions

x,y : fractional (0) or integer (1) scale on each axis

h,v : with full scale, the image will be framed as best as possible with the following settings (h:horizontal, v:vertical):

h : right(0), center(1), left(2)

v : top (0), center (1), bottom (2).

vq_scan

returns information about the open printer station (handle must be correct):

intout : grh (grh/div=number of graphic lines per pass)

intout+2 : number of passes per printed page

intout+4 : alh. (alh/div=nbr of lines per line of text)

intout+6 : number of lines of text per page

intout+8 : div.

xv_opnwk

variant of v_opnwk in which one can specify the maximum coordinates of the station:

id l lc m mc f tc fs fp fc c xmax ymax xv_opnwk

Returned values:

handle : station handle

intout : xmax taken into account (.w)

intout+2 : ymax taken into account (.w)

control : address of the Memory station (if id=61)

xv_updwk

variant of v_updwk in which the station buffer address is imposed:

addr xv_updwk.

These last two combined instructions allow for example to open a printer station with the dimensions of an image loaded in memory, then to call xv_updwk with the address of this image, it will be automatically printed.

The following instructions are implemented in the new color printer drivers under SPEEDO GDOS. After going through v_opnwk (21), here are the available functions:

vq_driver_info

returns **pad** on the stack with the name of the driver and in intout:

intout	: 0, new functions not available
intout+2	: version number of the library
intout+4	: driver version number
intout+6	: 1=mono, 3=CYM, 4=CYMK
intout+8	: mask of supported attributes:

0: quaddri
1: negative
2: mirror
3: HARD multiple copies
4: SOFT multiple copies
5: landscape

vq_bit_image

returns the following information:

intout	: 0, call does not exist
intout+2	: version number of v_bit_image
intout+4	: maximum number of images on a page
intout+6	:mask of available formats:

bits0 and 1 : 00=monochrome IMG
bits2 and 3 : 00=TGA not supported
: 01=TGA 2 uncompressed
Other values reserved.

vq_image_type

" file" addr vq_image_type : returns information about the specified image file:

intout	: 0, call does not exist
intout+2	:0 unknown file, 1=IMG, 2=TGA
addr	: number of color planes of the image
addr+2	: width
addr+4	: height

vq_margin

returns the following information:

intout	: 0, call does not exist		
intout+2	: top margin	intout+4	: bottom margin
intout+6	: left margin	intout+8	: right margin
intout+10	: Horizontal DPI	intout+12	: Vertical DPI

vs_crop

X Y X' Y' L P vs_crop
positions marks for cutting when printing on several pages:
X Y X ' Y' : framing rectangle
L : length of cutting lines
P : position of the lines with respect to the coordinates

(set everything to 0 to remove these marks)
intout : 0, call does not exist

vs_page_info

n " string" vs_page_info : passes information about the document according to the value of n:

n=0 App name
n=1 Document title
n=2 Name of document creator
n=3 Remarks.

intout: 0, call does not exist

AES functions

1 Presentation

The principle of calling the AES functions is based on the filling of input tables and possible recovery of parameters in the output tables. Some functions presented here are unique to FORTH and only serve to facilitate the use of AES primitives.

Input tables;

control

leaves on the stack the address of the control field containing the following information:

control	:function code
control+2	: number of words in intin
control+4	: number of words in intout
control+6	: number of addresses in addrin
control+8	: number of addresses in addrout

intin

leaves on the stack the address of the intin field containing the input parameters on 2 bytes each.

addrin

leaves on the stack the address of the addrin field containing the input addresses on 4 bytes each.

Output tables:

intout

leaves on the stack the address of the intout field containing in return the information on 2 bytes returned by the function.

addrout

leaves on the stack the address of the addrout field containing in return the addresses on 4 bytes provided by the function.

global

leaves on the stack the address of the global field filled by appl_init and rsrc_load.

global : AES version
global+2 : number of simultaneous applications (-1 under MultiTOS)
global+4 : ID returned by appl_in it
global+6 : long user word (FORTH uses it for menus!)
global+10 : pointer to the RSC file loaded with rsrc_load.
global+14 : 12 bytes reserved
global+26 : _AESmaxchar (version 4.00 or >)
global+28 : _AESminchar (version 4.00 or >)

Manual functions:

In normal use, the input tables are filled in for you, all that remains is to read the information in the output fields if necessary. However, for some functions that have not yet been implemented, a "manual" call will be necessary:

actrl

fn ii io ad actrl : fills the control field with f=function code, ii=number of words intin, io=number of words intout, ad=number of words addrin. The addrout part is not taken into account and initialised to zero as there are few functions using it. At worst you would have to add "n control 8 + w!" after this call.

aintin!

fills the intin field according to the number of values indicated in control+2. Values are taken from the stack.

addrin!

fills the addrin field according to the number of values indicated in control+6. Values are taken from the stack.

aes

calls the aes function assuming all three input arrays are properly populated.

aes!

assumes the control array is populated, then it populates the addrin array, then the intin array, then calls the AES. This is equivalent to

```
addrin! aintin! aes
```

Here is now the list of AES functions, for the syntax we have respected this order:

intin-values addrin-values function name

This is also valid for the aes! function. Be careful, this is not always the order of the syntax in C, sometimes the addresses precede the other data (go figure why...).

2 General functions

appl_init

declares an application to the GEM. The function is useless, even under the compiler since FORTH takes care of this. Nevertheless, the value returned by app_init is important (in the management of certain messages or events), so it is a simulation of the function which is carried out and which returns the id parameter:

intout : id of the current application or -1 if error.

appl_read

id n addr appl_read : reads n bytes in the event register and deposits them from the address addr. The id parameter is your identifier. The function is useless for all standard GEM events.

intout : 0 if error

appl_write

id n addr appl_write : writes n bytes in the event register, these bytes are taken from the address addr. The id parameter is your identifier.

intout : 0 if error

appl_find

" name" appl_find : returns the appl_id of the application whose name is specified. The name must contain exactly 8 characters (padded with blanks and without the file extension).

intout : appl_i d or -1 if not found.

appl_search

mode name type ap_id appl_search

appl_trecord

n addr appl_trecord : the AES records n events that are described at the address addr. Each event uses 6 bytes:

word : event code
long word : parameters

intout : number of recorded events.

appl_tplay

n sc addr appl_tplay : the AES replays n events at the address addr. The sc parameter indicates the desired speed (100=normal, 1000=x10, 50=half, etc.)

appl_getinfo

type appl_getinfo : returns information about the AES.

scrp_read

addr scrp_read : returns from the address addr the name of the access path to the clipboard (GEM clipboard). If intout contains 0, just create a \clipbrd directory and specify it using scrp_write.

scrp_write

addr scrp_write : indicates to the system the access path to the clipboard. addr points to a null terminated string, for example: "C:\CLIPBRD\". Function to be used only if scrp_read returns 0 (so that my clipboard does not yet exist).

shel_get

n addr shel_get : copy n bytes from the AES buffer (the desktop.inf) to the addr address.

she_put

n addr shel_put : copy n bytes of the addr address to the AES buffer.

shel_envrn

v " var=" shel_envrn : finds the value of an environment variable.

v is a variable that contains as output the address of the first byte of the string found, or contains zero if the search failed.

shel_write

mode wisgr wiscr cmd tail shel_write

shel_read

name tail shel_read

shel_find

buffer shel_find

fsel_intput

file path fsel_input : opens the file selector. For this purpose path and file are two strings containing the access path and the name of the default file proposed to the user. It is dangerous to use string constants because any modification of the path and of the file modifies the starting string (and therefore possibly its length).

intout : 0 if error
int out+2 : 0=cancel, other=confirm
file and path changed according to user actions.

fsel_exinput

file path title fsel_exinput : same principle as fsel_input, except that a text is added which is displayed as a title in the selector, thus indicating the operation carried out (30 characters maximum).

path

path file path : to be used in conjunction with the previous function. The two strings are popped and merged into **pad** whose address is left on the stack. For example:

path="C:\FORTH*.FOR" and file="TEST.FOR"

We will then find in pad="C:\FORTH\TEST.FOR". Note that the possible mask in path is removed.

getpath

fullpath getpath : return in **pad** the path extracted from the string fullpath removing an eventual mask or filename, pad is stacked.

getfile

fullpath getfile : return in **pad** the filename or mask extracted from the string fullpath removing an eventual path, pad is stacked.

change.ext

" ext" " path+file" change.ext : return in **pad** the same pathfile but with the new extension specified. pad si stacked. Example:

" FOR" " C:\FORTH\MYPROG.PRG" change.ext type

will display: C:\FORTH\MYPROG.FOR

form_alert

b string form_alert : opens an alert box, b corresponds to the number (1, 2 or 3) of the button linked to the RETURN key (0 if none). The string has this structure:

" [n][line1|line2|....][button1|button2|...]", there can be up to 5 lines of 30 characters and three buttons of 10 characters each, n is the icon number (0 none, 1=!, 2= ?, 3=stop).

intout : selected button number.

form_error

n form_error : displays an alert box corresponding to TOS error n. Some values of n correspond to a specific message, others to a simple "TOS error #n" box . To turn the GEMDOS error (-32 to -49) into the corresponding code n, 31 is subtracted from its absolute value. Example, code -36 (access denied) corresponds to box 36-31=5.

n=2, 3 or 18 : folder or file not found.
n=4 : insufficient memory to open an additional document.
n=5 : document already existing or write-protected.
n=8, 10 or 11 : insufficient memory to launch the application.
n=15 : Specified drive does not exist.

3 RSC resource files

They are the ones that traditionally contain the dialog boxes, menus and alert messages of an application. In the paragraphs devoted to these different elements, we will see how to create such objects without going through an independent file.

rsrc_load

string rsr c_load : loads a resource file and adapts it to the current resolution, relocates the addresses.

intout : 0 if error

rsrc_free

frees the memory used by a resource file.

rsrc_gaddr

t i rsrc_gaddr : searches for the address of the ith object of type t in the resource file loaded in memory.

t=0	tree	t=8	tedinfo-text
t=1	object	t=9	tedinfo-mask
t=2	tedinfo	t=10	tedinfo-valid
t=3	ICONBLK	t=11	subpointer ICONBLK
t=4	BITBLK	t=12	idem
t=5	string	t=13	idem
t=6	pointer on BITIMAGE	t=14	subpointer BITBLK
t=7	ob_spec	t=15	idem
		t=16	idem

intout : 0 if error

addrout : the address sought.

In a file with a single tree, to obtain the pointer essential to the objc_draw and form_do functions, we will use: 0 0 rsrc_gaddr

rsrc_saddr

t i addr rsrc_saddr : modifies the address of the ith object of type t, addr is the address that is imposed.

intout : 0 if error.

rsrc_obfix

o addr rsrc_obfix : converts the coordinates of the tree object o pointed to by addr from values in characters into pixels according to the current system font.

rsrc_rcfix

h rsrc_rcfix : fixes all the coordinates and pointers of the resource whose header address is h . If another file has already been loaded by rsrc_load, you must use rsrc_free before this call. This function requires an AES 4.xx, but FORTH provides its own routine with and older AES.

Tip on an older AES:

h is an even address, if you give h+1 then all color icons will be turned into icons so you can display an extended RSC on an old system.

Another tip: if intin = 'FORT' then the FORTH routine is used regardless to the AES and the global array is not modified (useful for a temporary ressource without rsrc_load).

rsrc_mono

flag pat addr rsrc_mono : adapts an RSC/tree for a monochrome display.

flag : 0 if addr is the address of a RSC header (located in global+14 if it was loaded with rsrc_load)
: 1 if addr is the address of a single tree (from rsrc_gaddr for example)

pat : from 0 to 7 force all objects with a colored background to this pattern in black and white
: -1 computes the best pattern to match the luminosity of the original color. For an ICON, is selects black on white or white on black according to the luminosities of the two original colors.

form_center

addr form_center : calculates the coordinates of a tree so that it appears centered on the screen.

intout+2 to +8 : X,Y,W,H

form_dial

f x y w h X Y W H form_dial : prepares the screen to draw a tree of objects, without this call, the redraws messages will not be correctly handled by the AES.

f=0 : screen reservation (X,Y,W,H), (x,y,w,h are undefined, 0 for example)
f=1 : draw a rectangle that grows from (x,y,w,h) to (X,Y,W,H)
f=2 : draw a rectangle that shrinks from (X,Y,W,H) to (x,y,w,h)
f=3 : release screen (X,Y,W,H), (x,y,w,h also undefined)

Steps 1 and 2 are optional.

form_do

i addr form_do : passes control to the AES for managing the form pointed to by addr, i is the index of the editable field receiving the cursor (0 if there is none).

intout : index of the object that ended the dialog, bit 15 is set in the event of a double click. To have the index no matter what:

```
intout %h7fff and
```

form_button

o c addr form_button

form_keybd

o n kc addr form_keybd

gem_input

" V|Prompt : ____ " str gem_input

Build a temporary object tree with the prompt, the editing field and an Ok button to allow you to enter data that will be stored into the string str.

If, before the call, the string is not empty, characters will be displayed in the field.

The V character is optional and specifies the desired type of entry, as in an FTEXT object:

X : all characters

9 : only digits

As a default, X is used..

Example :

```
3 string AGE
" 9|Age : ___ "  AGE  gem_input
AGE type
```

4 Menus

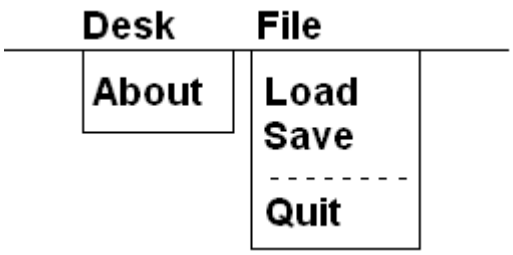
Either the menu is loaded in an RSC file, or it is created in FORTH. It is only in this second case that the first part of this paragraph is to be used.

menu

addr menu : creates a menu. For this purpose, addr must be the address of the first string of an array of strings (type array$) organized as follows:

- first all menu bar titles
- an empty string
- the option of title 1 (in general Info) + an empty string
- the options of title 2 + an empty string

So on for each title. For example, to create the following menu:



It will take 10 strings of 12 bytes :

```
12 10 array$ MENU
```

Then you have to fill in the table strings in order. The part concerning the accessories is managed by FORTH.

```
" Desk "    0 MENU $!
" File "    1 MENU $!
" "         2 MENU $!
" About "   3 MENU $!
" "         4 MENU $!
“ Load ”    5 MENU $!
" Save "    6 MENU $!
" ---------"7 MENU $!
" Quit "    8 MENU $!
" "         9 MENU $!
```

```
Always provide a blank on the right
and left for check_marks
and aesthetics.
```

Then to finish:

```
0 MENU menu
```

returns on the stack the address of the tree (like that obtained by rsrc_gaddr) which must be kept for all subsequent calls. For example:

```
0 MENU constant menu TREE
```

gemindex

n gemindex : provides the GEM index corresponding to menu string n. In the previous example, the string n°6 (Save) is not necessarily the 6th object of the tree in memory. However, it is this internal index that we need to know in order to use the AES functions correctly. For example, to disable option 7 (a separator line):

```
7 gemindex 0 TREE menu_ienable
```

strindex

n strindex : provides the string number of an option whose internal index is known. For example, evnt_mesag returns this index when a menu entry has been selected:

```
1 TREE menu_bar                 \ draw the menu
begin
      here event_mesag
      here w@ 10 =              \ wait for "menu selected" event
until
here 8 + w@                     \ internal index of chosen option
strindex                        \ its string number
case
      3 of ...                  \ it can only be 3, 5, 6 or 8
      5 of ...                  \ other strings not being
      6 of ...                  \ options.
      8 of ...
endcase
```

menu

addr setmenu : addr is the address of a menu tree (the one returned by menu) , the menu thus pointed becomes the current menu (the one used for gemindex, strindex and other menu operations). This feature is only useful when using multiple menus.

The instructions that follow are the actual AES calls , so they are applicable to any menu (loaded with rsrc_load or created in FORTH).

menu_bar

f addr menu_bar : draws (f=1) or cancels (f=0) the menu bar whose tree is pointed to by addr.

intout : 0 if error

menu_icheck

i f addr menu_icheck : puts a mark (f=1) or erases it (f=0) in front of the GEM index entry i of the menu pointed to by addr.

intout : 0 if error

menu_ienable

i f addr menu_ienable : activates (f=1) or deactivates (f=0) the GEM i index entry of the menu pointed by addr. Disabled options appear in grey text.

intout : 0 if error

menu_tnormal

i f addr menu_tnormal : switches to reverse video (f=0) or to normal video (f=1) the GEM index title i of the menu pointed to by addr. As soon as a menu entry is chosen, the title remains in reverse video until the user has resaturated it.

intout : 0 if error

menu_register

id string menu_register& : integrates an accessory or an application (under MultiTOS) into the menu bar. The id parameter is the one returned by appl_init, the string contains the text that will appear in the first drop-down menu.

intout : id of the accessory to keep (for event_mesag) or -1 if error.

See also ***>accname !***

menu_text

i tree string menu_text : modifies the name of the i GEM index option in the menu tree. The string contains the new name. In order not to overflow the frame, the length of the string should not exceed that of the old one.

intout : 0 if error

menu_attach

flag item addr mdata menu_attach

menu_istart

flag imenu item addr menu_istart

menu_popup

xpos ypos menu mdata menu_popup

menu_settings

flag set menu_settings

5 Events

event_keybd

waits for a key to be pressed and returns:

intout : keyboard code/ASCII code (high/low byte)

event_button

c m s evnt_button : waits for an event on the mouse buttons.

c : maximum number of clicks to expect
m : button to watch (1=right,2=left,3=both)
s : return if button pressed (s=1), released (s=2), whatever (s=3)

We get in return:

intout : number of clicks observed
intout+2 : mouse x during the event
intout+4 : mouse y
intout+6 : copy of s
intout+8 : status of special keys (bit0=sh_d,bit1=sh_g,bit2=ctrl,bit3=Alt).

event_mouse

f x y w h evnt_mouse : waits for the entry (f=0) or the exit (f=1) of the mouse cursor from a rectangle defined by x,y,w,h. We get in return:

intout+2 : mouse x intout+6 : mouse k
intout+4 : mouse y intout+8 : special keys

event_timer

t evnt_timer : waits for time t (in milliseconds on 4 bytes) to elapse.

event_dclick

v f evnt_dclick : reads (f=0) or sets (f=1) the double click speed. If f=1, v must be an integer from 0 to 4, if f=0, v is meaningless.

intout : speed taken into account.

event_mesag

addr evnt_mesag : waits for an event to occur and fills the 16-byte buffer pointed to by addr. We get in return:

addr : event number
addr+2 : id of the program that triggered it
addr+4 : 0 if sufficient buffer, or number of additional bytes

(see appl_read)

The event number can be one of the following:

10: MN_SELECTED	: addr+6=title index, addr+8=option index
20: WM_REDRAW	: addr+6=handle of window to redraw, addr+8 to addr+14=x,y,w,h
21: WM_TOPPED	: addr+6=handle of the window to activate.
22: WM_CLOSED	: addr+6=handle of window to close.
23: WM_FULLED	: addr+6=handle of the window that we want in full screen.
24: WM_ARROWED	: addr+6=handle of the window which was activated the arrows addr+8: 0=page_up, 1= page_down, 2=up, 3=down, 4=page_left, 5=page_right, 6=left, 7=right
25: WM_HSLID	: addr+6=handle, addr+8=cursor position horizontal (0 to 1000)
26: WM_VSLID	: addr+6=handle, addr+8=cursor position vertical (0 to 1000)
27: WM_SIZED	: addr+6=handle, addr+8 to addr+14=x,y,w,h new dimensions
28: WM_MOVED	: addr+6=handle, addr+8 to addr+14=x,y,w,h the new position
40: AC_OPEN	: addr+8=id of the accessory to activate (the one returned by menu_register)
41: AC_CLOSE	: addr+6=id of the accessory to close.

event_multi

choice c m s f1 x1 y1 l1 h1 f2 x2 y2 l2 h2 t addr event_multi
: waits for a combination of events according to the bitmask in choice:

choice bit 0 : evnt_keybd
bit 1 : event_button (cm s)
bit 2 : event_mouse (f1 x1 y1 l1 h1)
bit 3 : event_mouse (f2 x2 y2 l2 h2)
bit 4 : evnt_mesag (buffer in addr)
bit 5: evnt_timer (time t)

Unnecessary parameters must still appear (zero for example). returns the following values:

intout : observed event (coded as "choice")

intout+2	: mouse x	intout+8	: status of special keys
intout+4	: mouse y	intout+10	: code key pressed
intout+6	: mouse k	intout+12	: number of clicks observed

6 GEM windows and FORTH pages

Writing in a GEM window allows clean programming because the corresponding part of the screen is reserved, the different movements of windows, openings or closings are automatically accompanied by GEM messages making it possible to redraw any parts that may be rediscovered. However, inside a window, nothing prevents us from creating several FORTH pages there (to do multicolumning for example, or to separate drawing and text). However, if the pages are not housed inside a GEM window, parts of the screen may be dirty without any of the applications being notified.

When FORTH is launched, the main window is already open, with a compiled program it is fastopen which will take care of it. The corresponding FORTH page, page0, will automatically fit the window. It allows the output of text, graphics, keyboard entries. It will be more than enough for many applications.

It should be noted that FORTH opens a second VDI station for the standard text outputs, so that the user settings do not interfere with the proper functioning of the editor, for example. In this VDI station, the clipping is automatically performed on page0 so the output of even buggy texts (it can happen) will not bypass the window.

&page

&page XXX: creates a new word XXX corresponding to a FORTH page. It currently has no size or position definition. When executed, XXX leaves an address on the stack containing:

addr: WORDS x,y,w,h of the window
addr+8: WORDS x,y of the cursor and w,h of a character
addr+16: WORDS number of columns and lines (see setsubpage)
addr+20: FLOATS x0, x-len, y0, y-len, z0, z-len (used by gr_xxx functions)
addr+68:

defpage

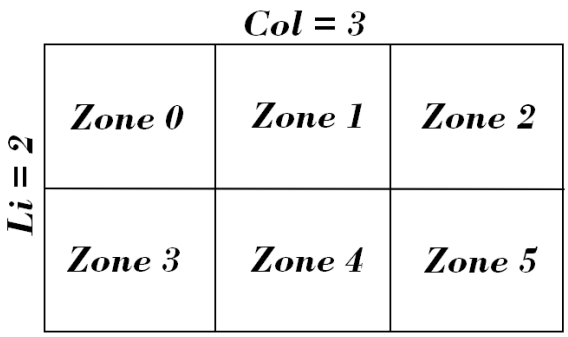
x y w h XXX defpage : sets dimensions of page XXX. Initialises the position of the text cursor in the page (0,0 at the top left). But, the page is not yet usable, it must be declared as an active page.

setpage

XXX setpage : sets XXX as the active page. All text output will be done in this page, if relative mode is enabled, all VDI output will be relative to the top left corner of this page.

setsubpage

col li XXX setsubpage : set the number of colums and lines to cut the page XXX.



getsubpage

i XXX getsubpage : returns the X, Y, W, H coordonates of zone i in the page XXX. Next is the image of the example:

```
&page ZONE4                          \ create a page
3 2 page0 setsubpage                 \ set grid to 3×2
4 page0 getsubpage ZONE4 defpage     \ get coordonates to define ZONE4
ZONE4 setpage                        \ and set this page
```

If parameter i exceeds the number of zones, then zone 0 will be used.

setmargin

m setmargin : set the margin thinkness inside every zone. Default value is zero, else, if X, Y, W, H is the size of the zone, then it returns:

X+m, Y+m, W-2m, H-2m when you call **getsubpage**.

pageclip

performs a clipping on the active page in order to limit the VDI outputs.

page0

predefined default FORTH page behaving as created by **&page**. It is the one opened under the interpreter or with **fastopen** in a compiled program. This word therefore leaves the address of a 68-byte zone described under **&page**.

page

current page, same behavior as **page0**.

fastopen

in a compiled program opens a full screen window with the same attributes as under FORTH and then returns its handle to the stack. You can keep it if you want to act on this window (change the title for example...). If the window is already open (as in the interpreter) fastopen simply returns the handle.

fasthdl

Returns the handle of the FORTH window, even if closed, it is not deleted.

fastclose

close the window opened with fastopen. Even in a program this operation is not necessary since it is automatically carried out before the exit of the program.

relative

mode in which the coordinates of the VDI functions are relative to the upper left corner of the active page and where the text functions of the VDI are simulated in this page.

absolute

mode in which the coordinates of the VDI functions are absolute (relative to the total screen) and in which the text functions of the VDI are executed in the normal way.

tovdi

x y w h tovdi : turns an AES rectangle x,y,w,h into a VDI rectangle x,y,x',y' taking into account the mode **relative** or **absolute** (always absolute for the AES).

toaes

x y x' y' toaes : turns a VDI rectangle x,y,x',y' into an AES rectangle x,y,w,h taking into account the mode **relative** or **absolute** (always absolute for the AES).

cls

deletes the current page and places the text cursor at the top left.

wind_create

- def x y w h wi nd_create : create a GEM window (it is not displayed yet), xywh are the maximum dimensions it can take, def is a bitmask indicating the elements of the window:

bit 0 : name	bit 6 : up arrow
bit 1 : close r (close)	bit 7 : down arrow
bit 2 : fuller (maxi)	bit 8 : vertical slider
bit 3 : mover	bit 9 : left arrow
bit 4 : info	bit 10 : right arrow
bit 5 : sizer (size)	bit 11 : horizontal slider

It is absolutely necessary to keep the returned value, it is the handle of the window, i.e. an identifier which will be used in all the following functions:

intout : handle or -1 if the window could not be created.

wind_open

hdl x y w h wind_open : opens the hdl handle window at the specified coordinates.

intout : 0 if error.

wind_close

hdl wind_close : closes the hdl window. (It is not destroyed, we can always reopen it later).

wind_delete

hdl wind_delete : destroys the hdl window. You must first go through wind_close.

intout : 0 if error

wind_set

modifies a window, there are several possible syntaxes:

hdl def 1 wind_set	: modifies the present elements of the hdl window (see wind_create)
hdl string 2 wind_set	: assign a title string to the hdl window
hdl string 3 wind_set	: assign an info string to the hdl window
hdl pos 8 wind_set	: horizontal cursor position (pos=1 to 1000)
hdl pos 9 wind_set	: vertical cursor position (pos=1 to 1000)
hdl 10 wind_set	: activate the hdl window
addr obj 14 wind_set	: installs a tree of objects (from the object obj) pointed to by addr instead of desktop background.
hdl t 15 wind_set	: sets the size of the horizontal cursor (t=1 to 1000)
hdl t 16 wind_set	: sets the size of the vertical cursor (t=1 to 1000)

For any other value of the flag (other than 1,2,3,8,9,10,14,15,16), you must use the general syntax of wind_set:

hdl param1 param2 param3 param4 flag wind_set

hdl x y w h 5 wind_set : new window position.

wind_update

f wind_update : handle screen and mouse control

- f=0 : the screen is free
- f=1 : you reserve access to the screen to undertake changes
- f=2 : the AES regains control of the mouse (menus, forms)
- f=3 : you take control of the mouse.

In screen updates, you must first call wind_update with f=1 before erasing the mouse.

wind_get

hdl f wind_get : returns information about the hdl window depending on the value of f:

f=4 : workspace dimensions
f=5 : dimensions of the total space
f=6 : dimensions of the previous total space (before fuller)
f=7 : maximum total space dimensions
f=8 : horizontal slider position (1 to 1000)
f=9 : vertical cursor position (1 to 1000)
f=10 : returns the handle of the active window (hdl has no meaning then)
f=11 : dimensions of the first REDRAW rectangle
f=12 : next REDRAW
f=15 : horizontal slider size (1 to 1000)
f=16 : vertical cursor size (1 to 1000)

Returned values:

intout : 0 if error
intout+2 : x or position (f=8,9) or handle (f=10) or size (f=15.16)
intout+4 : y
intout+6 : w
intout+8 : h

wind_new

clears all application windows and initialises v_hide_c and wind_update counters.

wind_find

x y wind_find : returns the window handle under x,y coordinates.
intout : handle (0 if desktop background)

wind_calc

f def x y w h wind_calc : provides the total space (f=0) from the workspace or the workspace (f=1) from the total space taking into account the elements of the window in the def bitmask (see wind_create).

intout : 0 if error
intout+2 to intout+8 : x, y, w, h.

By adding 2 to f, we get back x,y centered on the desk. This is a FORTH feature, not AES!

get_xywh

addr get_xywh : brings the 4 word values located at address addr back onto the stack.

get_xyxy

addr get_xyxy : brings the 4 word values found at address addr to the stack, then transforms w and h into x' and y' for dialogue with the VDI:

x'=x+w-1 y'=y+h-1

Indeed, the VDI rectangles are defined by two opposite points and not by the width and height from the upper left corner.

newin

p newin : redefine the size of the FORTH window to p% of the desktop size. Important : this is a brutal closure (window is closed, deleted and then recreated and reopened), the handle of the window may change. No test is done on the limits, don't try to use a too small value or a value greater than 100%. Page0 is then redimensionned to the new window and set as the default page.

set_redraw

hdl x y w h set_redraw : prepares the redraw loop for the hdl window. The dimensions of the rectangle are given by evnt_mesag or evnt_multi.

redraw

using the parameters specified in set_redraw, returns one by one the rectangles to redraw. During the first call, FORTH reserves the screen (1 wind_update) and erases the mouse (v_hide_c). When there are no more rectangles, the screen is freed (0 wind_update) and the mouse is displayed again (1 v_show_c).

returns 1 and x y x' y' if a rectangle is to be redrawn

returns 0 if done.

Small example:

```
here event_mesag
here w@ 20 =                    \ this is the REDRAW message
if
      here 6 + w@               \ the handle
      here 8 + get_xywh
      set_redraw                \ prepare the loop
      begin
            redraw
      while
            ....                \ here x y x' y' are available
            v_recfl             \ fills a rectangle for example!
      repeat
then
```

7 The Graf library

graf_handle

returns the following information:

intout	: handle to open a VDI station		
intout+2	: character width	intout+6	: cell width
intout+4	: character height	intout+8	: cell height

graf_mkstate

returns the following information:

intout+2	: mouse x	intout+6	: mouse k
intout+4	: mouse y	intout+8	: special keys

graf_mouse

f graf_mouse : sets the shape of the mouse:

f=0	arrow	f=4	flat hand
f=1	cursor	f=5	thin reticle
f=2	bee	f=6	thick reticle
f=3	pointing hand	f=7	crosshair outline

additionally: f=256 disable mouse
f=257 enable mouse
f>65535 considered as an address pointing to a mouse definition block, the definition is the same as for the vsc_form function of the VDI.

graf_dragbox

x y w h X Y W H graf_dragbox : inside a rectangle (XYWH), allows the displacement of a small rectangle of dimensions (w,h) from the coordinates (x,y). The mouse button must be pressed before the call, the function stops when released.

intout	: 0 if error
intout+2	: final x of the small rectangle
intout+4	: final y

graf_growbox

x y w h X Y W H graf_growbox : draws a growing rectangle.

graf_movebox

w h x y x' y ' graf_movebox : draws a rectangle (w,h) which moves

from x,y to x',y'.

graf_rubberbox

x y w h graf_rubberbox : draws a rectangle whose one corner (x,y) is fixed and whose opposite angle follows the movement of the mouse. (w,h) are the minimum dimensions of the rectangle. The mouse button must be pressed before the call, the function stops when it is released.

intout : 0 if error
intout+2 : final width
intout+4 : final height

graf_shrinkbox

X Y W H x y w h graf_shrinkbox : draws a shrinking rectangle.

graf_slidebox

p c dir addr graf_slidebox : in an object tree pointed to by addr, allows horizontal (dir=0) or vertical (dir=1) movement of a child object with index c inside a parent object of index p. The mouse button must be pressed before the call, the function stops when it is released.

intout : position of the child from 0 (left or top) to 1000 (right or bottom).

graf_watchbox

0 i s_in s_out addr graf_watchbox : watches the passage of the mouse in the object of index i of the tree pointed by addr. If the mouse is inside its ob_state=s_in, if it is outside then ob_state=s_out. The function stops as soon as the button is released.

The first parameter 0 must appear.

intout : mouse outside (0) at the end or inside (1).

8 Object trees

This paragraph deals with the creation of an object tree (form) directly in FORTH, ie without going through a separate RSC file. This manipulation requires a few calculations and probably several attempts for the settings. The first work is the reservation of the zones in which the definitions of the objects will be stored.

dialogue

ob_zone ted_zone alt_zone dialogue : starts the creation of a tree.
ob_zone : object block address, 24 bytes per object.
ted_zone : address of the TEDINFOS block, 28 bytes for each object of type TEXT, BOXTEXT, FTEXT or FBOXTEXT.
alt_zone : other structures, 14 bytes for each IMAGE, 34 by ICON and 36 by USERDEF.
It is obvious that at the beginning these areas are empty.

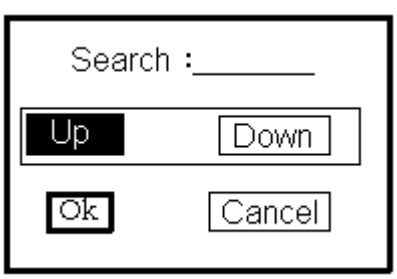
child<< >>

n child<< ... >> : declares that all objects created in this structure are children of object n. The structure cannot be nested (that does not mean that we cannot have children inside children).

lastobj

marks the last object as the end of the tree. Indispensable with FTEXT to avoid any error.

Let's illustrate this with an example, let's create the following form:



Object 0 is a BOX containing an FTEXT, an IBOX which contains two radioBUTTONs and two BUTTONs.

Here is the skeletton of the creation:

```
24 7 * allot constant OB_ZONE       \ 7 objects in all
28 allot constant TED_ZONE          \ a single TEDINFO
                                    \ (that of the FTEXT)
OB_ZONE TED_ZONE 0 dialog           \ the last
                                    \parameter is dummy
"box creation"                      \object 0

0 child<<                                       \ the children of 0
      "creating the FTEXT (Search)              \ object 1
      "creation of the IBOX"                    \ object 2
      "creation of the 1st button (Ok)"         \ object 3
      "creation of the 2nd button (Cancel)"     \ object 4
      >>

2 child<<                                   \ the children of IBOX
      "radiobutton creation (Up)"           \ object 5
      "create radiobutton (Down)"           \ object 6
      >>
lastobj                                         \ finished!
```

Each object created requires 24 bytes so used (example for object with index i):

ob_zone+24*i		: pointer to next object
	+2	: pointer to first child
	+4	: pointer to last child
	+6	: item type
	+8	: flag
	+10	: state
	+12	: spec (long word, 4 bytes)
	+16	: X
	+18	: Y
	+20	: W
	+22	: H

flag is defined as:
bit 0 : selectable
bit 1 : default
bit 2 : exit
bit 3 : editable (FTEXT)
bit 4 : radio button
bit 5 : last object
bit 6 : touch exit
bit 7 : hide tree
bit 8 : indirect

state is defined as:
bit 0 : selected
bit 1 : crossed (BOX type)
bit 2 : checked
bit 3 : disabled
bit 4 : outlined
bit 5 : shadowed

The coordinates are given in two-byte characters:

low byte : number of characters
high byte : signed number of points to add.

```
Example:    5           -> 5 characters
            %h0205      -> 5 characters plus 2 pixels.
            %hff05      -> 5 characters minus one pixel.
```

For the creation of objects, here are the useful words:

BOX

IBOX

BOXCHAR

flag state spec x y w h BOX : same syntax for all three. spec is a bitmask defined as follows:

bits 31-24 : ASCII character code (for BOXCHAR)
bits 23-16 : edge width (+inside, -outside)
bits 15-12 : edge color
bits 11-8 : character color
bit 7 : 0 transparent, 1 opaque
bits 6-4 : fill style
bits 3-0 : fill color.

BUTTON

STRING

TITLE

flag state string x y w h BUTTON : same syntax for all three.

TEXT

BOXTEXT

flag state x y w h string font just color TEXT : same syntax for both.

The font is small (5) or normal (3), justification centered (2), left (0) or right (1). color corresponds to spec of a BOX (bits 0 to 23).

FTEXT

FBOXTEXT

flag state x y w h string1 string2 string3 just color FTEXT

: same syntax for both. just and color are identical to TEXT. The three strings are the text, the input mask and the validation mask.

IMAGE

flag state x y w h addr color IMAGE : for this object, w is the width in bytes (even) and h is the height in points. The color is 1 to 16. Addr points to the data, each word constitutes 16 consecutive points.

ICON

flag state x y x' y' col_char string addr ICON : creates a 32x32 icon.

x' y	': coordinates of the character in the 32x32 image.
col_char	: character color: bits15-12=1st plane color, bits11-8=background color, bits7-0=character ASCII code.
string	: displayed under the icon in 6x6 font on 13 characters maximum.
addr	: data address, 128 bytes for 32*32 points of the mask, and 128 bytes for 32x32 icon dots.

USERDEF

flag state x y w h code param USERDEF : creates an object whose appearance and behavior is managed by a user routine. The routine must be a compiled or assembled word, it receives on a temporary stack (don't forget that we are in the AES) the address of a block thus defined:

addr	: address of the tree
addr+4	: object index
addr+6	: previous state
addr+8	: new state
addr+10	: absolute X, Y, L, H of the object
addr+18	: x, y, x', y' of the clipping rectangle
addr+26	: param (on 4 bytes)

If the two state values are equal, the object must be drawn in this state (a call to ob_draw). If they are different, you have to take care of passing the object from one state to another (for example, the object has been selected). The value of param is left free to the user, one can for example design the same routine for several objects, the param making it possible to distinguish them.

The routine must return a state whose remaining bits are those whose processing will be left to the AES, we return 0 if we take care of everything. The coordinates are in absolute mode with respect to the physical screen . If a VDI routine must be used, "absolute" must be specified first.

objc_change

index 0 x y w h state f addr objc_change : modifies the state of the object with index i in the tree pointed to by addr, xywh are the dimensions of the clipping rectangle if the object must be redrawn with its new state (f= 1), f=0 otherwise and xywh are undefined.

intout	: 0 if error

objc_draw

i d X Y W H addr objc_draw : draws a tree of objects pointed to by addr, the coordinates are those returned by form_center, i is the idex of the first object to draw (most often 0) and d is the number of generations to draw in depth (with 2, the AES will go to the children of the children of the starting object), we will choose a value exceeding 7 to have the whole tree.

intout : 0 if error

objc_find

i d x y addr objc_find : returns the index of the object located under the coordinates x,y. i and d define the search fork (see objc_draw) and addr is the address of the tree.

intout : index of the sought object or -1 if none.

objc_edit

obj kc idx mode tree objc_edit

objc_offset

i addr objc_offset : calculates the coordinates of the object i of the tree pointed to by addr with respect to the origin of the screen.

intout : 0 if error
intout+2 : x
intout+4 : y

objc_add

p c addr objc_add : links the child of index c to the parent of index p in the tree pointed to by addr.

intout : 0 if error

objc_delete

o addr objc_delete : unlinks of the object at index o in the tree pointed to by addr. Objects are not deleted.

intout : 0 if error

objc_order

c p addr objc_order : gives the child of index c the position p in the order of the children of the same parent, the tree is pointed by addr.

p=-1 : c becomes the last child
p=0 : c becomes the first child
p=1 : c becomes the second child
p=2 : etc.

objc_sysvar

m attr x1 x2 objc_sysvar : returns (m=0) or sets (m=1) the attributes of 3D objects, attr determines which attributes and x1,x2 are the values (if m=1 and meaningless if m=0).

intout : 0 if error
intout+2 : x1 in reading
intout+4 : x2 in reading

getradio

i addr getradio : i is the index of a box into a tree pointed at by addr and containing radio buttons. The function returns the index of the first radio button that is selected. It returns zero if none.

The search is performed only on one level, the radio buttons must all be direct children of the box with index i.

9 Forms in windows

The call to form_do is blocking, ie the user must absolutely answer it and exit before being able to use the menu or another form again. To overcome this, the forms are placed in windows. Thus we can move from one form to another by selecting the appropriate window, access to the menu will also be possible.

&wdial_create

tree &wdial_create XXXX : creates a word XXXX corresponding to a structure which will allow the inclusion of the tree of tree objects in a window. At this stage nothing is displayed yet, only a call to form_center has been made so that the first opening is done in the center of the screen.

On execution XXXX leaves the address of the following structure on the stack:

addr : tree (long)
addr+4 : x,y,w,h (words) of workspace (tree size)
addr+12 : x,y,w,h (words) of total space.
addr+20 : window handle or 0 (not yet opened or closed)
addr+22 : index of the editable object containing the cursor or 0.
addr+24 : position (in characters) of the cursor

The main box of the tree is automatically modified: no OUTLINE or SHADOW effect and its border is set to zero.

wdial_open

i " title" XXXX wdial_open : opens the window for structure XXXX (it contains the coordinates, the address of the tree). "title" is the title of the window, i the index of the editable object which will receive the cursor first or 0 if there is none.

If the window is already open, it is topped.

The function returns the window handle or an error code.

(the tree is not drawn, it is by evnt_multi and wdial_formdo that the drawing will be done through REDRAWS)

wdial_close

XXXX wdial_close : closes the window and the form disappears.

wdial_evnt

buf intout wdial_evnt : This is a standardized call to **evnt_multi** including the following events:

evnt_mesag(whose buffer address is buf)

event_keybd

evnt_button (only the left button is monitored)

It is therefore used to monitor the actions on the windows, the menu, the form buttons and the editable objects (FTEXT or FBOXTEXT). It fills buf (min 16 bytes) and intout (min 14 bytes) like evnt_multi.

Imposing the intout parameter makes it possible to choose any other zone (instead of the standard intout) and thus to make other AES or VDI calls before processing the event (it would otherwise be erased by another call).

wdial_formdo

buf intout wdial_formdo : manage forms in windows as soon as an event has occurred. buf and intout are those of evnt_multi or wdial_evnt. (If it's the intout of evnt_multi, it's the standard intout and it may be cleared on every AES or VDI call!).

- for a window event, checks that it is in the list of those created with wdial_open and supports: REDRAW, MOVED, TOPPED and CLOSED.

- for a button event checks that the click is above an object in the foreground window opened with wdial_open and supports SELECTABLEs buttons, cursor passing from one FTEXT to another.

- for a keyboard event checks that the window in the foreground contains an EDITABLE object (or a DEFAULT button) and supports the complete editing of the field, the passage from one FTEXT to another (tab or arrows) as well as the ENTER key (DEFAULT button).

Returned values:

0 : the event does not concern a window opened with wdial_open or the event has been entirely handled by the function.

1 : the event requires additional processing by the user, we then find XXXX on the stack (the structure concerned) as well as the index i of the EXIT button that caused the event:

i=-1 : CLOSER button of the window

i>=0 : another EXIT button of the form.

Once this action complement has been taken care of, wdial_formdo must be called again in order to ensure that no other event was pending. Only when 0 is received can wdial_evnt be called back for another series of events. This is why it is important to define a second 'intout' zone independent of the standard zone in order to preserve its values over several calls to wdial_formdo.

At each call, the event mask (in intout) is updated to remove those already processed.

wdial_change

i XXXX wdial_change : used to deselect the EXIT button i that was selected by

the user. If the window of the object tree is in the foreground the button will be redrawn, otherwise only its STATE is modified and it will be correctly redrawn as soon as the window comes back to front.

Here is a skeleton program using these words:

creation of trees with 'dialogue' or loading with 'rsrc_load'

```
tree1 &wdial_create fen1            \ create structures
tree2 &wdial_create fen2
....
16 constant allot BUF               \ buffer MESAG
14 constant allot OUT               \ new intout

possible display of a menu and/or a form ( wdial_open )


begin
      BUF OUT wdial_evnt            \ watch for events
      ( reaction to events not concerning
      forms (menu->open forms with
       wdial_open , other windows...)
      begin
            BUF OUT wdial_formdo    \ takes care of the rest
      while                         \ if other than 0
            case
            fen1 of                 \ tree-separated processing
                  case
                  -1 of fen1 wdial_close endof  \ close
                  i1 of react to i1 button      \ one EXIT button
                        i1 fen1 wdial_change    \ normal state
                        endof
                  i2 of ....                    \ next EXIT button...
                  endcase
            endof
            fen2 of                 \ ditto for the others
                  ....
            endof

            endcase
      repeat                  \ continue formdo to end events
again                         \ resume event, other events
```

10 Pseudo text windows

Managing a text window requires some work, espacially to record everything that has been desplayed to restore the content when the window has been covered by another rectangle. You can ease this task with those gem-lists that are simple GEM object trees with each line assigned to an element from a string array.

This is very useful, especially if you use the previous set of instructions that manages trees in windows: everything will be managed including the redraws, the moves and topping the window.

gem_list

i T$ l n " options" gem_list : create in memory a pseudo-window tree with n lines of lenght l, each one associated to an element of the T$ string array starting from element i.

Options are specified by a single letter each :

A : two arrows, up and down, are added and will be selectable.
S : lines are also selectable.
E : lines are editable (FTEXT)
B : a border is drawn

The function returns the address of the tree, it is created in the dictionnary.

If arrows are present, then they are objects #1 and #2 of the tree.

If the lines are selectable, then the first one is object #1 (without arrows) or #4 (with arrows present).

assign_array

i T$ adr assign_array : allows you to reassign to the gem_list pointed at by adr the elements from the string array T$ starting at element i.

resize_list

n adr resize_list : resize the gem_list to n lines, n being limited to the value used when creating the list. If arrows are used, then n can't be less than 2.

Returns 1 of Ok, or 0 if error (n outside the acceted range).

11 Auto source code generation

Managing a menu, the dialog boxes requires a huge GEM work that can be done automatically. FORTH offers a word that generates a whole source code from the analysis of a RSC file with its menu, dialogs and alerts.

rsc2for

opens the file selector in which you'll select the DEF file corresponding to your RSC. Both files must share the same location and name and differ only by their extension (DEF or RSC).

This instruction is sensible to the **select+/select-** setting.

Upon return, you'll find under the editor the source code to manage your RSC.

Limitations :

- the ressource file can contain only one menu.
- only named objects (those listed in the DEF) are managed.

The first lines are the loading of the DEF (to generate the variables), the loading of the RSC, the creation of one window for every dialog and some vairables.

Each tree TTTT has its routine **do_TTTT** that manages the window closure and that calls every subprogram **do_XXXX** for each EXIT button.

You just have to fill those subprograms.

The menu MMMM has its routine **do_MMMM** that calls the **do_XXXX** corresponding to each menu entry.

One **main** routine runs the program, it displays the menu bar (if available) and contains as comments the lines to open the windows enclosing the trees:

```
\ index " title" win_TTTT wdial_open
```

You'll have to modify the window title and set the index of the first FTEXT object that will receive the cursor, zero if none. Those lines are comments because you may desire that the opening of the windows will occur at another moment and not at start.

The main loop manages the calls to the menu, the dialog boxes, the window moves and ends up when the **exit_flag** variable is non zero. It's up to you to decide when to change this value.

You can ask for the creation of a **pseudo-window (gem_list)** directly from the ressource file. To do so, you must create a free string whose name starts with the underscore character (_) and give this value to the string:

G=l,n,options

G for gem_list, l characters per line, n lines and the options of gem_list. The underscore will be removed from the name to define the name of the window. As the names are limited to 8 characters, then a window name can't exceed 7.

The ligne can be completed with the initial values of the strings. For example, to create a selectable list of three languages:

G=8,3,S,Français,English,Italiano

Yous strings will be initialized and the window ready to be displayed.

Ainsi, la free string de nom _LISTE et de valeur G=10,5,S génèrera :

- la création d'un tableau de 5 chaîne de 10 caractères nommé **LISTE$**
- la création d'un arbre de 5 lignes associées à ce tableau d'adresse **tree_LISTE**, dont les éléments sont sélectionnables.
- la création des routines de gestion de la fenêtre **win_LISTE** avec une routine **do_LISTE** appelée à chaque sélection de ligne.

You can manage **drop_down** lists : they appear when you clic on a field and allow you to select a specific value. In order to do so, the field that will call the drop_down must be marked with an extended flag and you have to create a separate dialog with a string list. What you have to do is :

➔ the source object must be « selectable » and « exit » with a name XXXX for example, and you set the extended flag 8 of the object_status.

➔ the dialog must be named exactly XXXX0 (with a ending zero), built with only one box (BOX) and a string list (STRING) « selectable » and « exit ». The first string is used for cancel and doesn't modify the selection. You have to name it in order that the user can understand that (Cancel, or some signs suggesting to go back).

The associated routine, **do_XXXX**, will open the drop_down each time you click on the source object and will return the selected string's index (from 1 to n, 1 being the cancel option) . Furthermore, if the source object's type is STRING or TEXT (or any of its derivatives), then its content is automatically updated with the selected string value. Ensure that the source string and the list element s**hare exactly the same length**!

Last, the word **set_drop_down** allows you to init your field with a specific default value taken from the list of the drop_down.

Instead of creating your drop_down dialog, you can use the facility of a **gem_list**. In this case, this gem_list must be named _XXXX0 (be sure that this fits in 8 characters!) and have exactly those options : S (selectables lines) and B (for a border) :

G=l,n,SB

You can use the facility of string initialization here or, it'll be up to you to fill the string array XXXX0$ with the list you want to appear if you use a dynamic list.

Similarly, you can generate the **routines to manage the FORTH window** or a **user window** with more widgets in the same way. To do so, you must create a free string whose name starts with the underscore character (_) and give this value to the string:

F=

to manage the FORTH window. This one is directly mapped to page0, it contains only the name, the mover and the sizer widget.

Or the value :

W=options

to manage a user window with the widgets you desire among which :

N (name), I (info), S (sizer), F (fuller), C (closer)

U (up), D (down), V (vertical slider), L (left), R (right), H (horizontal slider).

At the end of the source code, you'll find the complete **structure** as it has been detected in your resource file. Here is an example :

```
\ menu  MYMENU -> do_MYMENU tree_MYMENU
\     MItem MINFO -> do_MINFO
\     MItem MCPU -> do_MCPU
\     MItem MGRAPHIC -> do_MGRAPHIC
\     MItem MQUIT -> do_MQUIT
\ dial  CPU -> do_CPU tree_CPU win_CPU
\     FText CPUTEXT
\     Buttn CPUTEST (Exit ,Selct) -> do_CPUTEST
\     Buttn FPUFLAG (Selct)
\ dial  GRAPHIC -> do_GRAPHIC tree_GRAPHIC win_GRAPHIC
\     FText GWIDTH
\     FText GHEIGHT (Edit )
\     FText GBITS
\     Buttn GTEST (Exit ,Selct) -> do_GTEST
\ alrt  ALQUIT
\ alrt  ALINFO
\ glst  LIST -> do_LIST tree_LIST win_LIST
\ fwin  WINDOW -> do_WINDOW
\ uwin  USER -> do_USER
```

At start **a menu** named MYMENU (it's a constant telling the order number of this tree in the RSC, if the menu is the very first one, then MYMENU=0) managed by the procedure do_MYMENU (you shouldn't have to modify it, it's complete) and whose address in memory is tree_MYMENU. This last can be used as a parameter for functions such as menu_icheck to check or uncheck a menu entry.

Then come **the four menu items** associated with their procedures (they all are empty for now). The item MINFO will activate the procedure do_MINFO. In this one, you can simply open an information alert box :

```
: do_MINFO
     1 ALINFO do_alert
;
```

Following are the **two dialogs** that will appear in a window. Each one comes with its name, procedure, its tree address and its window structure. For example, the CPU dialog is managed by the procedure do_CPU (you shouldn't have to modify it, it's complete), its memory address is tree_CPU and its window structure for every wdial_* call is win_CPU.

Let's say that the action of the menu entry MCPU is to open the CPU window :

```
: do _MCPU
     0 " Cpu" win_CPU wdial_open drop
;
```

0 as first parameter, because there is no editable FTEXT. And let's say that the action of the menu entry MGRAPHIC is to open the GRAPHIC window :

```
: do _MGRAPHIC
     GHEIGHT " Graphic" win_GRAPHIC wdial_open drop
;
```

GHEIGHT as first parameter, because it's the index of the editable FTEXT.

Similarly, you'll note thet the EXIT buttons such as GTEST or CPUTEST have their own procedure, it's empty and you'll have to decide what to do with them.

If you must **modify the text string** of an object, the easiest way to do is is to create a FORTH string and assign it to the object.

For example, the CPUTEXT object can be managed as this :

```
10 string CPUTEXT$
>comp
CPUTEXT$ CPUTEXT tree_CPU assign_string
```

At start its content will be the one found in the RSC. But if you change the content, a simple redraw message will update the display:

```
" MC68060" CPUTEXT$ $!
win_CPU update_win
```

The button CPUFLAG has the Selct flag, meaning that it can be selected or not. You can test it as this :

```
CPUFLAG tree_CPU get_obstate 1 and
```

This returns 1 if selected or 0 otherwise.

You can **select it** with :

```
CPUFLAG tree_CPU get_obstate 1 or
CPUFLAG tree_CPU set_obstate
win_CPU update_win
```

Or **unselect it** with :

```
CPUFLAG tree_CPU get_obstate %hFE and
CPUFLAG tree_CPU set_obstate
win_CPU update_win
```

Then, the **alert list** reminds you of their names. The procedure to use them, do_alert, is discussed below.

Then comes a **pseudo-window (gem_list)** named LIST. Its routine do_LIST will eventually call :

up_LIST and **down_LIST** if arrows are required in the options

lines_LIST if the lines are selectable.

This last procedure receives on the stack the number of the selected line starting from zero.

Finally are the definitions of two windows through the Free Strings. The first one is the **Forth window (fwin)** with the facilities fastopen/fastclose and linked to page0 for text and graphic output. You'll just have to complete the routine:

redraw_WINDOW

that, as a default, calls v_recfl to fill an empty rectangle when the content is to be reconstructed.

The second one is a **user window (uwin)** named USER with every widget you wanted. According to its elements, you'll have to complete the following routines:

arrows_USER to manage the arrows

hslid_USER and vslid_USER when the sliders are moved

redraw_USER to redraw the content.

Some routines integrated to the source code:

If the RSC contains alerts, the procedure **do_alert** is added and you invoke it with:

```
n XXXX do_alert
```

n being the default button number and XXXX the name given in the DEF file.

assign_string

Allows you to assign a FORTH string to an object of type STRING or BUTTON easing the modification of those objects.

```
str XXXX tree_TTTT assign_string
```

In the dialogue named TTTT, assign the string str to the object XXXX. The string is initialized with the content found in the RSC.

assign_fstring

```
str XXXX tree_TTTT assign_fstring
```

Similar to the previous word but for objects of type TEXT or FTEXT.

get_obstate
Returns the state byte of an object.
`XXXX tree_TTTT get_obstate`
Returns the state byte of object XXXX in the tree TTTT.

set_obstate
Set the state byte of an object.
`s XXX tree_TTTT set_obstate`
Sets to s the state byte of object XXXX in the tree TTTT.

update_win
`win_TTTT update_win`
Redraws the window containing the tree TTTT if some elements have been modified (strings, state of an object, ...).

If a window if present, then you can use this routine:

max_xywh
returns the max dimensions of a window on screen. You can use it in conjunction with wind_create or wind_open.

If a drop_down is used, then you have access to :

set_drop_down
`source tree_source n tree_dropdown set_drop_down`
give to the source object of the tree_source, the value of the n^{th} string from the tree_dropdown.

The unclassified

1 DMA sound

set play

two possible syntaxes:
1) -1 setplay : returns the detected DMA system
0 : none
1 : DMA (STE/TT/Falcon)
2 : SAGA (Apollo Vampire)

2) r ms f setplay : sets DMA sound parameters.
r : play once (0), repeat sound (1).
ms : bit #0 : mono (0) or stereo (1) sound
bit #1 : 8 bits (0) or 16 bits (1) Only on SAGA!
f : frequency in kHz (0=6.25, 1=12.5, 2=25, 3=50).

play

three possible syntaxes:
1) a t play : plays the sound located at address a and of size t in bytes. The sound is on signed 8 bits, for stereo sound one out of two bytes corresponds to the right channel, the other to the left channel. The settings are those passed to setplay.
2) 0 play : stop the current sound.
3) -1 play : returns when the current song is finished.

st_allot

tt_allot

these are variants of allot that force the allocation of ST RAM or TT RAM on a TT.
n st_allot : allocates a block of n bytes in the dictionary if it is in ST RAM, otherwise asks GEMDOS for a block in ST RAM. The address is returned on the stack.
n tt_allot : allocates a block of n bytes in the dictionary if it is in TT RAM, otherwise asks GEMDOS for a block in TT RAM. The address is returned on the stack.
st_allot is necessary for sound files, because the DMA cannot access the TT RAM. On SAGA, it doesn't matter.

loadbin

d t " file" addr loadbin :

if addr<>0 : loads a file at the indicated address, d is the offset of the first byte in relation to the start of the file, t is the size in bytes. returns the size actually read or a GEMDOS error code.

if addr = 0 : same as before but FORTH allocates a bloc with malloc and returns the size actually read and one value under the block address or one GEMDOS error code. If additionnaly d<0, this means to allocate a bigger block and to load the file after abs(d) bytes. For example, if addr=0 and d=-100, FORTH allocates t+100 bytes and loads the file a position 100 in the bloc, so you have 100 bytes free before your file.

Note: using t=-1, the whole file is loaded whaterver its size.

savebin

t " file" addr savebin : creates a file and writes to it t bytes found at address addr. Returns the number of bytes actually written or a GEMDOS error code.

2 ST REPLAY cartridge 16 (discontinued)

setreplay

f init inter setreplay : sets the parameters for cartridge management.
f : frequency from 48 Hz to 614400 Hz.
init : address of an initialization routine.
inter : address of the interrupt routine.

replay

executes the initialization routine, launches TIMER A with the interrupt routine. The right click of the mouse allows the stop. The interrupt routine can also put an end to it, the replay function then returns control.

The two routines must be written in assembler, the three words which follow make it possible to generate the instructions giving access in reading and in writing of the sound as well as the instruction ending the treatment:

replay_in *(deleted)*

n replay_in: generates the move $FB0000,Dn instruction in the dictionary. It is therefore the register Dn which contains the word read.

replay_out *(deleted)*

n replay_out: generates the tst instruction ($FA8000,Dn.w). So writes on the cartridge port the word value contained in Dn.

replay_end *(unused)*

generates the instructions which redirect the interrupt on a simple RTE and which set an internal flag to 1 ending replay.

3 Programming the timers

Besides the sound routines, it can be useful to program an interrupt routine at any frequency:

timerA

Two possible syntaxes:
timer_stop timerA : stops timer A, the constant **timer_stop** is actually 2.

rout mfp mode freq end timerA : launches Timer A with the following parameters:

rout:	: address of the routine to be executed periodically. Ends with an RTE.
mfp	: address of MFP registers to use. We'll use one of the two constants: **MFP_ST** = $FFFA80 **MFP_TT** = $FFFA00. On an ST, only the MFP ST exists!
mode	: counting mode, we will use the following two constants: **count** =8 for a synchro with an external source **delay** =0 for time-based counting.
freq	: desired frequency in Hertz
end	: routine end mode, we will use the following two constants: **auto_end** =0 for automatic return to normal during RTE **soft_end** =1 for a scheduled return.

Example:

```
routine MFP_ST delay 9600 auto_end timerA
```

launches the ST timer A on the specified routine at the frequency of 9600 hertz with automatic end of routine.

```
routine %hFFFA00 0 9600 0 timerA
```

has exactly the same effect as the previous line except that the names of the constants have not been used.

Next:

`timer_stop timerA` (or 2 timerA): stops execution of the routine.

timerB, timerC, timerD

are programmed in the same way as timer A.

timer_calc

freq timer_calc returns the internal values Control ans Data of the closest frequency desired. Control at the top of the stack.

4 Mouse, Joystick

joyst

n joyst : sets the monitoring mode for joysticks.
n=1 : only joystick 1 is monitored and the mouse remains active. Note: the fire button of joystick 1 is equivalent to the right mouse button.
n=2 : both joysticks are monitored and the mouse has no more effects.
In bith cases, some vectors from Kbdvbase are modified. You must use **mouse** to restore the default context as soon as the joystick access is no longer needed.

mouse

signals to the machine that a joystick and the mouse are connected (normal mode). This mode is automatically reactivated when returning to desktop and Kbdvbase vectors are restored.

mousex mousey mousek

respectively return the x, y coordinates of the mouse and the state of the two buttons (bit0:left, bit1:right). No vector modified here, only VDI calls.

jx0 jy0 fire0

return the horizontal position (-1:left, 0:center, 1:right), the vertical position (-1:up, 0:center, 1:down) and the state of the button (1:pressed, 0:released) of joystick 0 (the one that takes the place of the mouse). This one is only active in **2 joyst** mode.

jx1 jy1 fire1

return the horizontal position (-1:left, 0:center, 1:right), the vertical position (-1:up, 0:center, 1:down) and the state of the button (1:pressed, 0:released) of joystick 1. Instructions available in modes **1 joyst** or **2 joyst.**
Note: on the Apollo Vampire, both DB9 joysticks are always available and fire0 is equivalent to mousek=1. Furthermore, the USB Joypad is mapped on Joystick 1. Two cases:

mode **1 joyst**: **fire1** returns a bit mask with every Joypad button, key A being the fire:

0000 00sb 0000 YXBA (s=start et b=back)

mode **2 joyst**: **fire1** only returns the A as the fire button.

5 The others

desk

Displays a menu bar, allows adjustment of the position and size of the FORTH window, gives access to accessories or to other applications (under MultiTOS). You return to FORTH using the Return option or by reactivating the window (under MultiTOS). The FORTH base page is resized and reactivated (because those of the user are no longer necessarily well adjusted).

ltype

" string" ltype : sends a string of characters to the printer. We can by this means send a series of command codes. Only character 0 marking the end of the string, cannot be sent.

&cookie

&cookie XXXX : creates a word XXXX whose 4-letter name corresponds exactly to the cookie you are looking for (think of capital letters if necessary). On execution XXXX leaves on the stack:

either 0 if the cookie does not exist,

or 1 if it exists. Below the 1 is the value of the cookie on the stack (sometimes an address pointing at a block of parameters, sometimes a version number , etc...).

```
&cookie MiNT                    \ creates the word
MiNT if                         \ if there is 1
      ."Version" h.             \ if Mint present, its value is the version
else
      ."MiNT not installed"     \ if absent, it is reported
then
```

cache

x cache: sets the value of the CACR register managing the 68030 cache. Possible values of x:

%ha0a : empty en disables both caches
%h101 : enable both caches
%ha00 : empty and disable the data cache
%h100 : enable data cache
%h00a : empties and disables instruction cache
%h001 : enable instruction cache

call

addr call : jumps to a routine in assembler at the address addr. It must end with RTS (in normal use). It must only modify A4, A5, A6 knowingly:

a6 : data stack pointer
a5 : interpretation pointer
a4 : stack pointer for returns.

trace

f trace : enter trace mode immediately.

If f=0 only user words are paused.
If f<>0 most words are paused (some are automatically executed as stacking a constant, real or integer, internal codes for variables, etc)

When in pause, the word is displayed with the current stack content just before it is executed. Options are:

[Enter] execute the next words until a new stop (according to "f")
[S] (Single Step) execute only one word and stop again regardless of flag "f" passed to trace. This is convenient to inspect the stack just before and after a word.
[L] (leave) run program until a loop exit or a word exit occurs.
[R] (until Return) if the word is a user word, then execute it completely and pause after return.
[B] (breakpoint) run program until return to this current position.
[Esc] quit execution and go back to the command line.

untrace

stops tracing programs at the next word.

list watch

list 'word1 'word2... 'wordn watch Set a list of words to watch in trace mode.

In this case, only those words lead to a stop (whatever they are, user words or FORTH words) regardless of flag "f" passed to **trace**. The list remains valid even after an **untrace**. To disable it, just enter an empty list:

list watch

trace_win

y h trace_win : sets the TRACE window to line y and to a height of h lines (this in lines of text).

The program imposes a value of 3 for h at the minimum.

If y is negative, the position is calculated from the bottom of the screen.

To trace a program comfortably, you can, modify the FORTH window so that it does not overflow on the last 3 lines and thus have two separate areas : execution and tracing.

The default settings are three lines at the bottom of the screen.

illegal

insert the "illegal" instruction of the processor that leads to an exception. This is useful to set a breakpoint when using a degugger.

In normal mode, the instruction is followed by a RTS and then return to the interpretation loop.

In compiled mode, the instruction is just inserted into the program.

You must know that:

- A5 is the interpretation pointer to the next instruction (WORD)
- A4 is the pointer to the return stack (also used by loops)
- A6 is the pointer to the current FORTH stack.

Math extensions

1 Functions

A function is nothing but a FORTH word leaving its result on the stack with however two limitations:
- the function must be a compiled word (with :: ... ;)
- the function takes its value(s) in &float type variables

For example:

```
&float x
:: function x f@ x^2 %f5 f- ;
```

This programs f(x)=x² - 5

solve

'FFF X %ferr solve : this word is used to solve an equation of type f(x)=0.
FFF : the name of a function (note the quote before!)
X : the variable used by the function
%ferr : a real value corresponding to the desired precision.

In return, solve returns on the stack the precision reached and in X the solution found.

With the example above:

```
%f2 x f!                          \ first x value
'function x %f0.0001 solve f. \ displays the precision reached
x f@ f.                           \ displays the solution 2.236....,
                                  \ the square root of 5.
```

gauss

'FFF X %fa %fb n gauss : this word is used to calculate the integral of the FFF function for the variable X going from a to b. The number of intervals n makes it possible to fix the precision of the calculation.
In return, gauss returns an approximation of the integral.

In practice, any polynomial function, up to degree 9, is integrated exactly with a single interval (n=1). For the others, changing n to 10, 100 or more can make it possible to obtain a more precise value.

This method does not take the values at the terminals, therefore, the generalized integrals (1/sqr(x) in 0) or extended by continuity (x*log(x) in 0) are accepted.

With the previous function:

```
'function x %f0 %f1 1 gauss f.      \ displays -4.666...,
                                    \ the integral of x²-5 from 0 to 1.
```

maximum

'FFF X %fa %fb %ferr maximum

Search for x from a to b where the function reaches its maximum, err is the maximal error accepted..

There must be one and only local maximum between a and b.

Return the precision reached and the value in X.

minimum

'FFF X %fa %fb %ferr minimum

Works like maximum but for a local minimum between a and b.

lambert

%fa lambert

Lambert function that returns w such as $w \times e^{w} = a$, it is defined for $a \in [\frac{-1}{e} ; +\infty]$ and returns a value in the main branch $[-1 ; +\infty]$. This function is an algorithm based on newton's method, so it is relatively slow, especially on a machine without FPU.

lambert1

%fa lambert1

Same behavior as **lambert** except when $a \in [\frac{-1}{e} ; 0]$, the returned value is from the secondary branch in $[-\infty ; -1]$.

Example: to solve $x^{x} = 0{,}75$, we use $x = e^{lambert(\log(0{,}75))}$

```
0.75 log lambert exp f.  -> returns x = 0,6362....
0.75 log lambert1 exp f. -> returns x = 0.1535....
```

There are two solutions !

2 Plotting 2D curves

To draw a 2D curve, you must define the FORTH page where the points will be drawn using **setpage**. As a default, **page0** corresponding to the FORTH window will be used.

Then you must set the limits for x and y axis using **gr_window**. You can add a grid and the axes with **gr_axes** and **gr_grid**.

Finally, you can draw your functions, parametric or polar.

gr_window

%fxmin %fxmax %fymin %fymax gr_window

Defines the mathematical limits for the x and y axis. Those four parameters are real numbers and you must check that xmin<xmax and ymin<ymax.

gr_axes

Without parameters. This word just draw the axis is they appear in the window.

Side effect: the function uses v_pline and, in output, applies a "0 0 vsl_ends" that removes the arrow ends.

gr_grid

%fxstep %fystep gr_grid

Draws a grid with a vertical line every "xstep" and an horizontal line every "ystep". Check that those two values are real and positives.

Side effect: the function uses v_pline and, in output, applies a "0 vsl_style" that reset the default line style.

gr_y(x)

'FFF x %fx0 %fx1 n gr_y(x)

Plots the function FFF of type y=f(x), with variable x from x0 to x1 with n points. The function must be a compiled one and must return one value y(x).

Example: drawing of a hyperbola

```
&float x

\ y=1/x

:: yx
   x f@ 1/x
;

cls
( %f-5 %f10 %f-4 %f4 ) gr_window
                       gr_axes
( 'yx x %f-5 %f10 99 ) gr_y(x)
```

gr_xy(t)

'FFF t %ft0 %ft1 n gr_xy(t)

Plots the parametric function FFF of type x(t) and y(t) using variable t from t0 to t1 using n points. The function must be compiled and must return two values on the stack : x(t) then y(t).

Example: drawing alpha symbol:

```
&float t
\ x=-cos(2t) et y=sin(3t)

:: xyt
   t f@ fdup f+ cos fneg
   t f@ %f3 f* sin
;

cls
( %f-3.14 %f3.14 %f-1.1 %f1.1 ) gr_window
                                 gr_axes
( 'xyt t %f-3.14 %f3.14 100 )    gr_xy(t)
```

gr_r(t)

'FFF t %ft0 %ft1 n gr_r(t)

Plots a polar function FFF of type r(t) with variable t (the angle) from t0 to t1 using n points. The function must be compiled and must return r(t) (the algebraic distance to the origin) on the stack.

Example: drawing of a 4 leaf clover:

```
&float t

\ r=cos(t)*sin(t)

:: rt
   t f@ sincos f*
;

cls
( %f-1.1 %f1.1 %f-0.7 %f0.7 ) gr_window
                               gr_axes
( 'rt t %f-3.14 %f3.14 100 )   gr_r(t)
```

gr_getpoint

n gr_getpoint

enters an interactive mode with the graphic in the current page and allows you to read coordinates and, eventually, to stack them for future use. The mouse pointer turns into a cross when entering the clickable zone, and a simple arrow outside.

As soon as you click into the graphic zone, an alert box opens with the real coordinates of the point. At this moment, you can:

- **Stack it !:** x,y coordinates are left on the stack and if you've reached the maximum points, the function exits.
- **Quit :** end of the function.
- **Ok ! :** the point is ignored and you can click on another one.

The n parameter of the function determines its behavior::

- n >0 the max number of points you can stack (pairs of reals x and y)
- n = 0 only read is allowed, you can't stack.
- n = -1 no limit for the number of points even if over 99 the display will be hazardous.

Returns the actual number of stacked points with their coordinates under.

Side effect: the function calls evnt_multi, form_alert and graf_mouse. Upon return, the mouse shape is the arrow.

g_plot

```
%fx  %fy  gr_plot
```

Plots a point in the current page converting real values to screen coordinates. The function calls v_pmarker and accepts the settings of size, style and color.

gr_area

```
'test  x  y  n  gr_area
```

Fills a zone according to the criteria from the test function.

This function must return an integer value 0 (false) or else (true) corresponding to a condition on x and y.

The whole FORTH page is parsed and the test function called for every pixel (so youd 'ont have to initialize x or y), upon return, the pixel is plotted or not.

The n value is the desired density.

If n=1, every pixel is tested.

If n=2, one pixel out of 2 is tested in width and height (so 1/4 pixels)

If n=3, one pixel out of 3 is tested in width and height (so 1/9 pixels)

Etc.

Upon return, x contains a statistical approximation of the colored area (the closer to 1 for n, the more precise for the area).

3 Plotting 3D curves

To plot a 3D curve, you have to select the output page with **setpage** and define the limits with **gr3_window**. You can eventually add a frame with **gr3_grid** as well as the axis using **gr3_axes**.

Finally, you define your function that returns x, y and z from a single parameter t and a call to **gr3_xyz(t)** makes the curve appear.

gr3_window

%fxmin %fxmax %fymin %fymax %fzmin %fzmax gr3_window

Defines the limits of the three axis, for each the min value must be lower than the max value.

This call resets the parameter of **gr3_view** to zero: projection x,y,z and the rotation **gr3_rot** to zero.

gr3_grid

Draws a 3D-background frame of the plotting zone.

gr3_axes

Draws the three axes x,y and z (taking into account the parameter gr3_view). If the given limits don't include the origin, then the axes may not appear.

gr3_xyz(t)

'FFF t %t0 %t1 n gr3_xyz(t)

Plots the parametric function FFF of type x(t), y(t) and z(t) using variable t from t0 to t1 using n points. The function must be compiled and must return two values on the stack : x(t), y(t) and then z(t). The drawing takes gr3_view into account.

gr3_plot

%fx %fy %fz gr3_plot

Plots a point in the current page converting real values to screen coordinates. The function calls v_pmarker and accepts the settings of size, style and color.

The drawing takes gr3_view into account.

gr3_view

n gr3_view

Defines the axis order on the screen. The default value is **n=0** for a **x,y,z** projection.

This means x forward, y to the right et z up.

If **n=1**, it's a **z,x,y** projection.

If **n=2**, it's a **y,z,x** projection.

To read the current mode, use **-1 gr3_view** that returns the value 0, 1 or 2. So you can easely modify the view angle without changing anything else in your program. Of course, after this change, you'll have to call again gr3_grid, gr3_axes and gr3_xyz(t)/plot to get the new view.

4 Plotting 3D surfaces

Two words are available to plot surfaces of type Z=f(X,Y), they differ by the quality of the rendering, and thus, by their speed. The previous 3D functions are still useful (**gr3_window, gr3_grid, gr3_axes**) and the orientation can also be modified using **gr3_view**.

The function you want to plot must be assembled and use two variables (for X and Y) and return the corresponding Z.

gr3_z(xy)l

'fn X Y n gr3_z(xy)l

Plot the fn function using the two variables X and Y with a fast line rendering without hidden lines.

The parameter n is the number of intervals (if n=20, then 21 lines are drawn).

The color is set with **vsl_color.**

gr3_z(xy)f

'fn X Y n gr3_z(xy)f

Plot the fn function using the two variables X and Y with polygons taking account of hidden lines.

The parameter n is the number of intervals (if n=20, then 21 lines are drawn in both directions).

The color is now set with **vsf_color** as a filling VDI function is used (**v_fillarea**). Side effects are the modification of **vsf_perimeter** and **vsf_interior**.

gr3_rot

%fangle gr3_rot

Set the rotation angle (in radions) of the surface around the Z-axis. If angle=0, there is no rotation and the plotting is faster.

5 Diagrams

Here are some words to generate easely pie charts or bar charts. One important parameter is "t", defining the type of filling for the surfaces.

Parametre t

t filling type, used in loop from the lowest value to the highest and again. For a pie chart or a simple bar chart, the patterns are used in sequence. For a stacked bar chart, every new bar starts a new sequence of patterns up to the top os the stack.

- t=0 keep current fill settings
- t=1 colors (0-1 in monochrome, 0-4 in ST Medium and 2-15 for the other modes)
- t=2 patterns 9 to 24
- t=3 12 different hatchings
- t=4 density 1 to 8
- 'XXX the type can be the code of an assembled FORTH word that will be called for every new circular sector or every new rectangle, so you can set your own filling styles. Your routine will receive two values on the stack:
  - At the top, the n value. For the first call, n≥100, this means that a new serie is starting, you have to set your initial parameters, prepare for the first style and return a "n0" value (selected by yourself). For the next call, you will receive "n0+1", and set your next filling style, and return "n0+1". Then you will receive "n0+2" and so on. When a new serie is starting, you'll receive n≥100 again.
  - The second parameter is the c value. For the bar charts, it is the number of the current column (starting from 1). For pie charts, it is the starting angle of the current sector in 1/10 of degrees (For example, for a half-pie with 4 sectors of 45°, you receive sequentially 0, 450, 900, 1350). In all cases, this value must be dropped, only n is returned.

The data

Ai is the address of the first value in an array, the current **size** setting defines the datat type:

- .b or .w size data are bytes or words, they are unsigned (0-255 ou 0-65535).
- .l size data are long signed values
- .f size data are floating point values, 8 bytes each.

Combined labels

To add a legend on the x-axis of a bar chart or on every sector of a pie chart, the differents words are combined onti a single string. For example:

```
" MoTuWeThFrSaSu" 2 7
```

means that it contains seven words of 2 characters each. Should a word be shorter than c, you must pad it with spaces. When displaying the string, the final spaces are removed.

Font size for the labels

The parameter f sets the font size, based on the four standard fonts included in the TOS ROM. For a pie chart, you have to specify the desired size. For a bar chart, FORTH makes for you the best choice to fit the text around the diagram.

- f=0 small 6×6 (the one used for the desktop icons)
- f=1 fonte 8×8 (the one used in ST Low 320x200)
- f=2 fonte 8×16 (the one used in ST High, TT Medium)
- f=3 fonte 16×32 (TT only)

The (half-)pie charts

For those charts, you specify the coordinates, it's up to you to manage the space on screen..

gr_pie

x y r A1 n t gr_pie

Draw a pie chart:

- x y r center coordinates and radius of the circle.
- A1 n address of the n data to be represented

gr_hpie

x y r A1 n t gr_hpie

Draw a half-pie chart:

- x y r center coordinates and radius of the semi-circle.
- A1 n address of the n data to be represented

gr_pie_label

x y r A1 n t " xxx" c f gr_pie

Draw a pie chart with labels:

- x y r center coordinates and radius of the circle.
- A1 n address of the n data to be represented
- " xxx" c n combined labels, c characters each
- f font size

gr_hpie_label

x y r A1 n t " xxx" c f gr_hpie

Draw a half-pie chart with labels:

- x y r center coordinates and radius of the semi-circle.
- A1 n address of the n data to be represented
- " xxx" c n combined labels, c characters each
- f font size

Bar charts

Those diagrams use the current FORTH page. If you want to limit the size of the output, you have to create a page with **&page**, set its dimensions using **defpage** and select if with **setpage**. The default choice is using **page0**.

The following functions use the same base as the the drawing of 2D mathematical functions. They call **gr_window**, then **gr_axes** and eventually **gr_grid** to ease the reading with a grid.

gr_bar

A1 n t gr_bar

Draw a bar chart:

- A1 n address and number of values to be represented (they can be signed).
- t filling type

gr_barg

dy A1 n t gr_bar

Draw a bar chart with a grid. This grid is aligned vertically on the bars and uses the floating point value dy as a scale on the vertical y axis.

- A1 n address and number of values to be represented (they can be signed).
- t filling type
- dy real number (8 octets) for vertical scaling
  - ► If dy is negative, then the grid is drawn under the bars.
  - ► if dy is positive, then the grid is drown over the bars.

gr_sbar

A1 A2 ... Ak k n t gr_bar

Draw a bar chart where each bar is a stack of k values taken from the k addresses:

- A1 A2... Ak k n k arrays of values that will be stacked in n columns.
- t filling type

Values can be either positive or negative, they will be stacked over or under the x-axis.

gr_sbarg

A1 A2 ... Ak k n t gr_bar

Draw a bar chart where each bar is a stack of k values taken from the k addresses:

- A1 A2... Ak k n k arrays of values that will be stacked in n columns.
- t filling type
- dy real number (8 octets) for vertical scaling
  - ► If dy is negative, then the grid is drawn under the bars.
  - ► if dy is positive, then the grid is drown over the bars.

Values can be either positive or negative, they will be stacked over or under the x-axis.

gr_bar_hlabel

" xxx" c n gr_bar_hlabel

display the n labels combined into the string with c characters each under the diagram. For this to run correctly, a bar chart must have been drawn before and its page still selected. Else, the relsult may be erratic.

The number of labels is not necessary the number of columns of the chart. For example, imagine that you have drawn a 12 columns diagram for the 12 months of the year. You can perform one of these calls:

To label each month:

```
" JFMAMJJASOND" 1 12 gr_bar_hlabel
```

To group the month into four seasons:

```
" WinterSpringSummerAutumn" 6 4 gr_bar_hlabel
```

For a single comment:

```
" One year evolution" dup len 1 gr_bar_hlabel
```

gr_bar_vlabel

" xxx" gr_bar_vlabel

Add a legend on the vertical axis of a bar chart. The text is rotated by 90°. For this to run correctly, a bar chart must have been drawn before and its page still selected. Else, the relsult may be erratic.

gr_bar_title

" xxx" gr_bar_title

Add a title over a bar chart. For this to run correctly, a bar chart must have been drawn before and its page still selected. Else, the relsult may be erratic.

Multitasking

This is a cooperative multitasking system. Each task decides when to hand over to the next one. This feature is available in interprtated mode only. If you use assembled routines, you must exit from them before using the multitasking set.

1 About threads

A thread is a task. It is defined by:

- a FORTH word, the one to be executed
- a personal stack created with **&stack**
- a personal page for text and graphic outputs created by **&page**
- a unique identifier, an integer value defined by the user.

The remaining internal pointers to separate the tasks are managed by FORTH.

Different threads can share the same page, this is up to you to manage how. Else the two instructions **setsubpage** and **getsubpage** are useful to share the window space.

A thread can have three different states:

- ✔ run : this is the default state when it is created.When it's its turn, the program is executed until **thnext** or **thsync** is called to pass control to another thread.
- ✔ pause : when it's its turn, the program is ignored until another thread wakes it with **thwake**. A thread can pause itself or pause another one.
- ✔ sync : similar as pause , but will automatically be woken up when all synchronized tasks have reach their "sync" point.

When a thread reaches the end of its program, it is automatically deleted from the tasks lists. This deletion can be done using **thkill** or **thkillall**.

2 General instructions

thread

page stack id 'word thread

Defines a thread with its page, stack and the word that will be executed. The identifier is a value of your own. The task is not run yet, only added to the list.

vthread

page stack id 'word vthread

Similar to thread but a VDI station is open. This allows your thread to have specific graphic settings independant from the other threads.

thid

returns the identifier of the current running thread. In the direct mode, the value zero is returned. So a null value is not recommended for your threads.

thrun

exits the direct mode and enters the multitasking mode. Tasks are execute using thnext to pass from one to the next.

There are two ways to go back to direct mode:

- ✔ all threads are finished, then they are all deleted.
- ✔ the user has pressed Control+Alt+Shift (one only Shift)

If the user is responsible, then it's just a pause and you can use **thrun** again to resume execution where it stopped.

thnext

inside a thread, hands over to the next task.

thpause

id thpause : pauses the thread whose identifier is specified. A thread can pause itself using:

```
thid thpause
```

thwake

id thwake : wakes up a paused thread. If it was not paused, then nothing changes.

thkill

id thkill : removes the specified thread from memory and from the tasks list. Its stack and page still exist.

thkillall

Removes and deletes all threads from memory. If the instruction was called by a thread, then this goes back to direct mode.

thstat

displays the list of the currunt tasks with their identifier and status.

3 Synchronization

The synchronization system is inspired by the Transputers. If several tasks must be synchronized to a certain point to exchange data for example:

1. you select a synchro channel (there are four of them)
2. you specify how many task must reach this point with **thnsync**
3. each thread that ends its own part must call **thsync** to wait for the others
4. one particular thread must call **thsyncmain**

As soon as all tasks have reach their sync point, the system wakes them all up and passes control to the one that used thsyncmain. This last has the responsability to collect data and prepare the next work for the group. When it will call thnext, everything must be ready to work again.

thnsync

n c thnsync : tells the system that n threads must wait for each other on the synchronization channel #c (from 0 to 3).

thsync

c thsync : pauses the current thread until all threads have called thsync on the channel #c.

thsyncmain

c thsyncmain : pauses the current thread until all threads have called thsync on the channel #c. At this moment, all threads are woken up and control is given to this particular one.

4 Local variables (or kind of)

In FORTH, variables are global. This means that one name always points to the same memory location, and so, to the same value.

Let's imagine that several threads use this word that returns the grey level of the color of a pixel located in X and Y:

```
: grey
      X @ Y @ v_get_pixel           \ returns the color index at X,Y
      intout w@ 1 vq_color          \ returns the components of that color
      intout 2+  w@ 3 *             \ R*3
      intout 4 + w@ 6 * +           \ G*6
      intout 6 + w@ +               \ B
      10 /                          \ grey level =(3R+6G+B)/10
;
```

If each thread works on differents places on the screen, then the coordinates X,Y will be messed.

The solution is to create new variables that are internally arrays indexed by the ID of the thread. You can imagine that consecutive thread identifiers are recommended to limit the amount of required memory.

This way, the thread ID=1 will use the value of index 1 from the array, this automatically. The direct mode will use the values of index zero of the array.

If our program runs two threads and the direct mode, we will need three values of 4 bytes each and define X and Y with :

```
3 4 thvariable X
3 4 thvariable Y
```

The word "grey" will run with different values for X and Y according to the current thread.

thvariable

n s thvariable XXX : create a variable for n threads (from 0 to n-1) using s bytes each.

For bytes values (access with c@ and c!):

```
n 1 thvariable BYTE
```

For words (access with w@ and w!):

```
n 2 thvariable WORD
```

For real numbers (access with f@ and f!):

```
n 8 thvariable REAL
```

Strings are possible! Don't forget to count the ending zero and two bytes for the size, you'll have to round to an even number.

For example, to use 20 characters strings:

20 + ending zero + word size = 23, this is 24 bytes when rounded.

```
n 24 thvariable STRING20
```

M_PLAYER / MP_STE extension

This extension allows you to communicate with the video players M_PLAYER and MP_STE either to replay or to create videos. The player must be installed as an accessory.

1 Reading videos

vd_set

" Name" vd_set defines the player's name (exactly 8 characters, eventually padded with spaces). Returns 0 if error or the player's AES identifier if Ok. Examples: :

```
" M_PLAYER" vd_set .    \ look for M_PLAYER.ACC
" MP_STE  " vd_set .    \ same with MP_STE.ACC.
```

vd_info

" path\file" vd_info

returns the address of an information block about the video file :

```
adr         WORD status       1=Ok
                              0=File not found (nothing else filled)
                              -1:unknown type (only file_name filled)
                              -2:Error, M_PLAYER didn't run...
adr+2       CHAR file_name    (14 bytes: max 12 + nul + even address)
adr+16      CHAR file_type    (28 bytes: max 26 + nul + even address)
                               such as 'Video for Windows (AVI)'
adr+44      WORD graph_status
                              1: supported
                              0: no graphics (infos filled with garbage)
                              -1:unsupp (infos filled)
adr+46      WORD width
adr+48      WORD height
adr+50      LONG number of frames
adr+54      LONG compression  (4 bytes such as 'cram', 'cvid'...)
adr+58      WORD sound_status
                              1: supported
                              0: no sound (infos filled with garbage)
                              -1: unsupp (infos filled)
adr+60      WORD sound bits   (more often 8 or 16)
adr+62      WORD channels     (1:mono, 2:stereo)
adr+64      LONG frequency
adr+68      LONG version      4 ASCII bytes representing the version of
                              the player, for example '4.28'
```

vd_play

" path\file" " options" vd_play

Launches the player to visualize the video. Options can be:

+d/-d enables/disables the dialogs boxes, default +d.

All other options are only valid in "-d" mode, else the settings from the dialogs overwrite the command line options.

+p/-p enables/disables the sound, default +p.

+s/-s enables/disables synchronization between images and sound or time informations, default +s.

+a/-a for a BAT file, enables/disables the creation of the video, if disabled, you get a slideshow of the images. For any other type of video file, enables the disassmbling mode. Défaut -a.

+e/-e enables/disables the error messages, default -e

+i/-i enables/disables the repetition mode, default -i.

+xn/+yn enforce the coordinates for display. If a resolution change is necessary, this is ignored. On a TT in TT-Low, ligned on a 16 boundary. As a default, the animation is centered.

If you use **ANIPLAY**, the options are slightly different:

+d/-d enables/disables the GEM interface, default +d.

+i/-i enables/disables the repetition mode, default +i.

+p/-p enables/disables the sound, default +p.

+s/-s enables/disables skipping frames for synchronization, default +s.

+xn/+yn as for M_PLAYER: position (default is centered)

Unlike M_PLAYER, those settings remain valid from one call to another.

2 Creating videos

There are two ways to create a video with M_PLAYER / MP_STE.

• Either you already have a set of images and eventually a sound file, you just have to write your BAT file (refer to the player's manual).
Then, just call **vd_play** with option +a:

```
" BAT file"   " -d+a"  vd_play
```

And you'll get your video.

• Either, you want to generate your images one by one and assemble them on the fly using M_PLAYER in parallel. In this case, the payer calls a routine of yours each time a new image is required. That's what vd+create does ! *(With MP_STE there are some limitations, see **(*)**).*

vd_create

'gen z " out" b w h f k q t (or " sound") vd_create
Runs the creation of an animation:

gen name of an assembled FORTH word generating 16 bits images.
On entry, it gets the desired frame number (from 0 to f-1)
Upun return, two parameters are sent via **vd_frame**.

z **(*)** zoom factor, three possible values:
%h0101 no zoom
%h0201 double the image
%h0102 reduce the image to its half
If you add %h8000 then the player's progression box is displayed. For example, using %h8102 you get the box and half of your images.

" out" out file name to be created.

b **(*)** number of bits and format of the output file:
8 8 bits, dithering to 256 colors for a MOV RLE8.
16 16 bits, for a MOV RLE16.
-8 8 bits, dithering to 256 colors for a AVI RLE8.
-16 16 bits, for a AVI CRAM16.
4 4 bits, you send a DEGAS image, 320x200 16 colors for a FLM 16 colors.
-4 4 bits, you send a NEOchrome image, 320x200 16 colors for a FLM 16 colors.
1 monochrom, you send a DEGAS image 640x400 for a monochrom FLM.

w,h width and heigth of the source image. Ensure that the width is aligned on a 4 boundary (even after the eventual zoom).

f total number of frames.

k key-frames frequency, those particular frames rebuild the whole screen, this allows to maintain synchronization even on a slow machine (ifi k=5, images 0,5,10,... will be key_frames)

q compression quality (from 1, bad to 5, good).

t either a time information in 1/200 second for each image (it must be < 65536, so limited to 5 min 27 sec per image), either a string containing the path and file name of a sound and synchronization will be adapted to the sound length.

The sound must be:
- 8 or 16 bits uncompressed
- mono or stereo
- a frequency close to standard AVIs: 11,025 or 22,05 or 44,1 kHz.

(*) With **MP_STE**, the zoom is ignored and only the $8000 flag is read. Likewise, modes b=+/-8 and b=-16 are not available.

With **M_PLAYER**, the zoom can be used with AVI/MOV but is ignored with FLM as those animations have a fixed image size. Then use %h0101.

vd_frame

buffer format vd_frame : within gen routine, returns to M_PLAYER the address of the buffer containing the image and precises the pixel format :

```
0     : pixels with NOVA 16 bits format   vvvbbbbb xrrrrrvv
-1    : pixels with Falcon 16 bits format rrrrrvvv vvvbbbbb
4     : DEGAS image (2 null bytes, 32 bytes palette XBIOS,
        32000 bytes screen image 320×200×16).
-4    : NEO image (4 null bytes, 32 bytes palette XBIOS, 92 null bytes,
        32000 bytes screen image 320×200×16).
1     : DEGAS image (2 bytes 0 & 2, 32 bytes palette XBIOS,
        32000 bytes screen image 640×400×2). In this case, the palette is
        ignored and fixed to white/black by M_PLAYER.
```

Note : To send the current palette into the image, you can use **savevdipal** and **pal2xbios**. Example in 16 colors:

```
32034 allot constant DEGAS    \ the whole image
  192 allot constant PALv     \ the palette VDI 16×6 bytes

PALv savevdipal               \ save the current palette to PALv
PALv DEGAS 2+ 16 pal2xbios    \ turn the palette to XBIOS into the image
```

General usage of the gen routine:

For a 120×96 image in 16 bits, we need a zone of 120×96×2 bytes:

```
23040 allot constant ZONE

:: gen                  \ word must be assembled !
      drop              \ on entry, the frame number is given, drop if unused
      ....              \ here we must fille the ZONE with 16 bits pixels
      ZONE -1 vd_frame  \ returns the address and the format
;

'gen %h8101 " C:\OUT.AVI" -8 120 96 50 10 5 20 vd_create
```

This will create and animation AVI with RLE8 compression, 50 images 120×96 with a good quality of 5 and a speed of 20/200 sec per image, this means 10 images per second.

The key frame rate is k=10, so every second the player can resynchronize if the replay is too slow.

The SuperCharger Extension

This extension allows to communicate with the Supercharger. The latter is a PC card connected to the ACSI port equipped with an 8MHz NEC V30 and a RAM of up to 1 MB.

1 The Tool Box

In order to be able to communicate, special routines must be loaded into memory and accessible by Trap #4.

Either you use the resident program SCTB.PRG supplied with the Supercharger, or we will use TOOL_BOX.BIN which we can loaded when executing the FORTH program.

Any device on the ACSI port is assigned a number from 0 to 7. By default, the Supercharger is assigned DMA 3. If you have a special configuration, you must modify this value before any calls to the other functions:

sc_dma

n sc_dma : this word sets the device number to n

sc_check

sc_check : tests the presence of the ToolBox.
Returns:
- a negative error code if the routines are not present
- zero and the version number if the routines are present
the version is a long representative 4 characters in ASCII.

Error codes:

0	OK
-1	missing ToolBox routines
-2	(reserved)
-3	Supercharger not connected
-4	Time Out
-5	Protocol error
-6	Hotkey ABIO (PC emulation in progress!)
-7	(reserved)
-8	illegal function call
-12	file already loaded (with sc_load)

If sc_check fails, you need to load the routines using the following function:

sc_load

" path\TOOL_BOX.BIN" sc_load : load and install routines on Trap #4
Returns an error code. This one can be less than -32:
-33 file not found
-39 insufficient memory (requires 18000 bytes of ST-RAM)

sc_addr

sc_addr : returns on the stack the address where the ToolBox is loaded by sc_load. If the call has not been executed, or after sc_unload, zero is returned

sc_status

sc_status : returns Supercharger status
0 if the Tool_Box is already in communication.
-6 ABIO Hotkey (MS Dos running)
Any other value means an undefined state.

sc_unload

sc_unload : uninstalls the communication routines and frees the buffer in ST-RAM.
Returns an error code or zero.
If an error has occurred, the memory block is not freed.
Note: it is absolutely necessary to use this call before quitting a FORTH program, otherwise Trap #4 will point to an area no longer containing the routines.

II Slave mode

In this mode, it is the Atari which controls and pilots the Supercharger. It is the first mode you have to set with **sc_boot** before being able to communicate.

PC side addresses:
The NEC V30 uses two 16-bit registers to represent an address.
A segment register (block of 16 bytes) from %h0000 to %hFFFF
An offset register to add to the segment from %h0000 to %hFFFF.

Thus, %h0200:045A represents the address :

%h0200 x 16 + %h045A = %h0245A.

On the Atari side, we will use a long word formed from the segment and the offset.

The memory card is organized as follows:

```
Supercharger 512k :
      0000:0000 to 0000:2FFF  12 kB for the ToolBox
      0000:3000 to 6000:7FFF  404 kb free
      B000:8000 to B000:FFFF  32 kB free
      F000:0000 to F000:FFFF  64 kB free

Supercharger 1 MB :
      0000:0000 to 0000:2FFF  12 kB for the ToolBox
      0000:3000 to F000:FFFF  1012 KB free
```

sc_boot

sc_boot : switches to slave mode.
Returns an error code or zero if Ok.

In this mode, three commands are available:

sc_sendm

atarisrc pcdest size sc_sendm : sends size bytes from the Atari atarisrc address to the pcdest PC address.

Warning:
- the Atari address must be in ST RAM!
- size is limited to 65535 bytes
- the transfer is afster if the addresses are even and size is a multiple of 512 bytes.

sc_getm

pcsrc ataridest size sc_getm : receives size bytes from the pcsrc PC address to the Atari ataridest address.

Same limitations as for sc_sendm.

sc_exec

pcaddr sc_exec : launches the execution of a program on the PC side at the pcaddr address and switches to **cooperative mode**.

Only CS and IP registers will be initialised for the NEC V30:
CS : 16 most significant bits of pcaddr
IP : 16 low bits of pcaddr

From there, Atari and PC must collaborate. It is the PC that decides to return to **slave mode**.

III The collaborative mode

In this mode, a program is running on the PC side and each call from the Atari via Trap #4 must correspond to a call via Int 60h on the PC side.

On the return of these calls, you can regularly receive the error -4 which signals a Timeout. This is simply because the NecV30 is running part of its program and not responding immediately to a transfer request.

We can use for example:

```
begin
      call one of
      four transfer instructions
      dup -4 =
while
      drop
repeat
```

We only exit the loop if there is no Timeout. Returns on stack 0 (everything is ok) or another error code.

sc_xsendb

n sc_xsendb : sends the byte n to the PC.

PC side:

```
mov ah,9          get byte function
int 60h           on return al = byte received
```

sc_xgetb

sc_xgetb : receives a byte from the PC.

PC side:

```
mov al,bytevalue to send in al
mov ah,10         send byte function
int 60h
```

sc_xsendm

atarisrc size sc_xsendm : send size bytes from atarisrc address to PC.

PC side:

```
                  es:di destination address
mov cx,size       number of bytes to receive
mov ah,4          Receive Data function
int 60h
```

sc_xgetm

ataridst size sc_xsendm : receives size bytes from the PC at the address ataridst.

PC side:

```
                    ds:if source address
mov cx,sizenumber of bytes to send
mov ah,5            Send Data function
int 60h
```

There are two last functions on the PC side:

The first to **return to slave mode** while keeping the ToolBox:

```
mov ah,0            function terminate
int 60h             return to slave mode
```

The second to completely **erase the ToolBox** from PC memory. Following the latter, the Atari must redo a call to **sc_boot** if it wishes to communicate again:

```
mov ah,1            kill tool box function
int 60h             indefinite mode
```

IV Data exchange

The NEC V30 stores its data in Little Endian. The following instructions will be used to communicate during data exchanges using several bytes:

iw! i! if!

These three instructions are the pendants of w!, ! and f! and store the data (word, long, float) in Intel format at the address indicated.

Example:

```
%f2.36 addr if!         \ stores an Intel float
addr 8 sc_xsendm .      \ and send it to the program
```

iw@ i@ if@

These three instructions are the counterparts of w@, @ and f@ and retrieve data (word, long, float) in Intel format to Atari format.

Example:

```
addr 8 sc_xgetm.        \ get 8 bytes intel
addr if@ f.             \ and displays it as a float
```

Parx M&E Modules

This instruction set facilitates the use of M&E modules defined by Parx Systems. These are image import/export modules and effect modules.

We must first indicate the path to the modules folder, it follows the following organization:

PARX.SYS folder

\RIM	RIM modules subfolder (read image)
\WIM	WIM modules subfolder (write image)
\IFX	IFX modules subfolder (effects)
\FORTH	FOM modules subfolder (Forth Modules)
PARX.TRM	dithering module. (TRM for tramage in french)
PARX.MEM	memory management module.
PARX.Mnn	module MOT (for french motif = pattern on nn planes)
PARX.Bnn	BRO module (for french brosse = brush on nn planes)
PARX.Pnn	PAL module (palette on nn planes)

The FORTH does not use the memory module but will take advantage of its own organization of the dictionary as a stack of blocks that can be reserved, extended and freed (see **here** , **remember** , **restore**) during the use of RIM, WIM and IFX.

As for the module itself, it can be loaded into the dictionary or into an external block by malloc.

Module management

modset

" path" flag modset : sets general options.

flag : if bit0 is at 1, then the functions dorim/dowim/doifx and dotrm display the bee as a mouse cursor during their work.

The path of the PARX folder, it must end with "\". Example:

```
" D:\PARX.SYS\" 1 modset
```

Does not return a value.

modload

Loads a module and assigns it to a variable.
Two possible syntaxes:

- " name" flag var modload : loads a module.

var : a variable that will remain associated with the module, as input:
var = 0 allocate a block with malloc and load the module into it
var = -1 load module into dictionary
name : module path in the PARX folder, example: " WIM\XIMG.WIM".
To avoid adding the path of the PARX folder, ie if the name is already complete, simply add 128 to the first character of the string.

- flag var modload

var = addr the module is already in memory at the address contained in var, be careful though: this address must be preceded by 32 free bytes for temporary variables.

flag : if the module is configurable, flag indicates the attitude to follow:
flag=-1 skip config
flag=other start the configuration with code=flag, by default the code must be equal to 1, but some modules accept other values (example JPEG_68K.RIM with 1 it displays a copyright, with 0 it just relocates the program to memory.)

In the 32 bytes before the module, this last receives some informations that it can save or not for future use. See the section **FORTH modules management** for details on this block.

In the particular case of the TRM, it is absolutely necessary to launch the configuration, flag=-1 is forbidden, it is a reseved value (see **Graphic Card and TRM 2.52**)! The standard use is:
flag : 0 for standard Atari video mode
: 1 for a graphics card (the one set by EDIT_TRM)
This value can be obtained by the **graphic_card** function .

Return: for the cases var=0 or -1, fills the variable with the address of the module. This variable will be needed for each subsequent call! Each module has its own variable, you can manage several modules at the same time, possibly in an array.
Returns:
a GEMDOS error code if the loading was not successful
0 : if everything went well.
-1 : error, configuration failed
-2 : the module does not support the call to configuration
-3 : if the module is of unknown type

modunload

var modunload : If it is the TRM, the TRM_END function is called and only in the case of a module loaded with var=0, the function frees the memory block and resets var to zero. No value returned.

modmload

" mask" flag array-element modmload

Loads all the modules corresponding to the mask and lodges the addresses in each element of the array starting from the one indicated.

Returns the number of loaded modules.

If a module does not load or is of unknown type, memory is not allocated and the module does not appear in the list.

Initially, the first array element must contain 0 or -1:

0 : all modules will be loaded in an area allocated by malloc

-1 : all modules will be loaded into the dictionary.

(In practice, if the value is not zero, it will be considered -1).

Example:

```
100 array MODULES                    \ array of 100 elements
0 0 MODULE !                         \ MODULES(0)=0, load with malloc
" rim\*.rim" 1 0 MODULES modmload     \ load all RIM
constant NUMBER                      \ number of loaded modules
```

modmunload

array-element n modmunload

Delete n modules pointed to by the array starting from the given element. This is the reciprocal function of modmload. To clear all modules from the previous example:

```
0 MODULES NUMBER modmunload
```

modfmt

var modfmt

- for a RIM/WIM/IFX : returns the image format of the module.
  - 0 shifter, coding of an Atari screen (interlaced planes up to 8 bits, rrrrrvvvvvvbbbbbb for 16-bit, RGB for 24-bit and xRGB for 32-bit)
  - 1 VDI, independent format (separate planes up to 8 bits)
  - 2 can handle both depending on the case.
- for TRM : returns the number of dithering formats available.
- for BRO, PAL, MOT : returns the number of planes.

modsname

var modsname : returns in **pad** (on the stack) the short name of the module.

modlname

Four possibilities:

- for a RIM/WIM/IFX
  var modlname : returns in pad (on the stack) the long name of the module.

- for the TRM
  i var modlname : returns in pad on the stack the name of dithering n°i

- for BRO, PAL, MOT and MEM
  var modlname : returns the short name, there is no long name.

- for a FOM
  i var modlname : i=0 : returns in pad on the stack the long name
  i=1,2,3 returns the redefinition name of modinfo, modpal or modexe when available.
  With a FOM extended (modtype=n), then from i=3 up to i=3+(n-1), returns the redefinition name of every subroutine of modexe.

modver

var modver : returns the version of the module (in decimal).

Note: from *version 210* of the **TRM** , in addition to the available dithering (see **modfmt**), there is a quality **zoom** (*dithering=-1*) and a simple zoom (*dithering=0*).

modtype

var modtype : returns the module type.

For a RIM:

0	requires to have the whole file in memory.
1	can work with segments of the file (not really supported here).
2	RIM generating, creates the image itself (fractals, scanner...), the dimensions of the image must be specified.
3	RIM generating, but the dimensions are fixed.
4	RIM animation.

For a WIM:

0	saves an image to disk.
2	WIM generating, sends an image to a device.
4	Animation WIMs.

For an IFX:

0	effect on a single source image
1	effect on two source images

Pour un FOM:

0	module with PARX standard format (three routines modinfo, mod pal and modexe, eventually redefined, see **modlname**).
n	module FORTH extended, the modexe routine is split into n subroutines that you'll have to call by their redefined namesl (see **>fom** et **modlname**).

For the others:

-1	is the TRM
-2	the MEM module
-3	a BRO brush
-4	a PAL pallet
-5	a WORD pattern
-6	an unlisted PARX module.

Managing a RIM, reading an image

Once the RIM is in memory, you will be able to use it to load and decode an image.

You have two methods, the first automatic calls **dorim** and takes care of the whole procedure, the second is manual and calls the **modinfo** , **modpal** and **modexe** triplet .

Automatic method:

dorim

addr size " name" handle pal flag var dorim

You must first load the file into a new block of the dictionary obtained with allot and leave the file open.

addr	: file block address
size	: size of loaded data
name	: string name of the file (or simply its extension)
handle	: that of the file still open
pal	: area of 1536 bytes to receive a possible VDI palette , zero if you are certain that it will be unused.
var	: corresponding to the module loaded with modload

Dorim presents the file to the RIM module and tries to decode it, we get:

2 : Ok, decoded file and data in mfdbs , PAL contains palette
3 : Ok, decoded file, data in mfdbs, no palette
-1 : error in modinfo, file not supported by this module
-2 : error in modpal
-3 : error in modexe, damaged file.

If successful, the dictionary contains in order:

1 the block with the file
2 followed by a possible additional temporary block
3 followed by the block with the decoded data

Typically, the file and the temporary block are no longer needed but take up memory. This is the role of the flag:

flag : 0 leave the three blocks as they are
: 1 erase the file and the temporary block and move the data to this new free zone. The dictionary end pointer is adjusted.

It is up to you to close the file.

Manual method:

For this you will need:

1 fill the **mfdbs** with data from the screen (see **fillmfdb**)
2 open the image file and load it into memory, leaving it open
3 call **modinfo** to verify that it is recognized by RIM
4 call **modpal** to manage the palette and any buffer
5 call **modexe** to perform the decoding.
6 close the image file.

modinfo

addr size " name" handle var modinfo:
check the image with:

addr : address where the file is loaded
size : file size
" name" : name of the file, here you can only put its extension, it is the extension that is checked by the module.
handle : file handle returned by fopen.

returns:

0 : image not recognized
1 : insufficient data (impossible if the whole file has been loaded)
2 : Ok, contains a palette
3 : Ok, no palette.

If the file is recognized then **mfdbs** is populated as follows:

mfdbs : LONG offset to palette data (or 0)
mfdbs+4/+6 : image width and height
mfdbs+8 : Width in 16 pixel packets
mfdbs+10 : format
mfdbs+12 : number of planes
mfdbs+14 : LONG palette size

At this point you can decide whether or not to continue loading with **modpal** depending on the given information.

modpal

pal var modpal
optionally loads the palette and gives information about the necessary working buffer:
pal : address of a 1536 byte zone receiving the palette in VDI format ,
or zero if there is no palette.

returns:
0 : error
1 : insufficient data (impossible if the whole file has been loaded)
2 : Okay !
3 : Ok, with block overflow. In this case, below is returned the additional size needed following the image as a working block. It might be a good idea to load the image file into the dictionary and use **allot** to extend the area if needed.

modexe

dest var modexe
starts decoding the image.
dest : address of a buffer large enough to accommodate the pixels.
For this purpose, we can use

```
mfdbs imagesize
```

to get the size you need.
returns:
0 : error
1 : insufficient data (impossible if the whole file has been loaded)
3 : Okay !

After this call:
mfdbs : correctly filled.
pal : contains palette data (if present) in VDI format
dest : contains the decoded image

Managing a WIM, writing an image

Once the WIM is in memory, you will be able to use it to save an image.

You have two methods, the first automatic calls **dowim** and takes care of the whole procedure, the second is manual and calls **modinfo** , **modpal** and **modexe** triplet .

Automatic method:

You must correctly fill the **mfdbs** with the data of the block to be saved then call dowim:

dowim

" filename" PAL flag var **dowim**

attempts a backup of the mfdbs-defined block with its palette to disk.
flag : bit0 = 1 to automatically handle format transformation with
vr_trnfm, this only happens if the only difference between your
block and what the WIM is able to handle is the format.
bit1 = 1 to add or modify the file extension with the recommended one
by the module. Otherwise, keep the given one.
var : the pointer to the module
PAL : address of a palette if used

returns:
0 : ok, everything went well, the file is saved.
-1 : the dimensions of the block are not suitable, use the TRM
-2 : the block format is not suitable, use vr_trnfm
-4 : the number of planes is not suitable, use the TRM
... -7 : any combination of -1, -2 and -4.
-8 : error in MODPAL, the format is incompatible
-9 : error in MODEX
other : other negative code, this is a GEMDOS error code during creation
or writing to the file.

Side effects:

- mfdbd is used
- pad contains the recommended extension or the final name of the modified file
- the dictionary was used for temporary blocks

Manual method:

1 correctly fill the **mfdbs** with the data of the block to be saved
2 call **modinfo** to know the possibilities of the module
3 call **modpal** to provide the palette and see compatibility
4 create the file and leave it open
5 call **modexe** to perform the backup
6 close file

modinfo

var modinfo
provides information about its possibilities in conjunction with your previously completed **mfdbs** .

returns:
0 : no palette export
1 : pallet export
2 : pallet export possible
also returns the desired extension in **pad** .

Additionally, it can modify the **mfdbs** for you:

- w and h can be modified if the export is limited to certain sizes.
- format can be changed if the module only supports shifter or VDI.
- number of planes can be changed depending on the image format.

It's up to you to decide if the changes are right for you to proceed with.

modpal

pal var modpal
rereads the mfdb and an eventual palette.
pal : address of a palette in VDI format or zero if unitized.

returns:
-1 : incompatible format
other : this is the size of the work buffer requested, everything is OK.

modexe

work handle var modexe
starts saving the image in a file you just created.
work : address of the work buffer whose size was provided by **modpal** .
handle : file handle provided by **fcreate** .

returns:

-1 : error
other : this is the size remaining to be saved from the work buffer. Do not use fseek, the file pointer is correct.
Most often it's zero because the WIM has already written everything.

At the end of this call, you still have to close the file with **fclose** and free any buffer.

Management of an IFX, image effect

For this type of module, only the call to modex is used. The effect applied to the image may take one or two consecutive calls.

If the module is type 0, the effect only requires an image described in mfdbs with possibly an output to a destination image stored in mfdbd.

If the module is type 1, the effect requires two images described in mfdbs and mfdbd as well as an additional mfdb as an output image or temporary image.

Type 0 IFX

This is the most common module, only one source image is needed.

Automatic method:

doifx

PAL1 PAL2 var doifx
handles the two consecutive calls and returns:

0 : ok, effect applied at the first call
1 : ok, effect applied after the 2nd call without destination palette
2 : ok, effect applied after the 2nd call, PAL2 contains the destination palette
-1 : error on first call
-2 : error on second call

Manual method:

modexe

PAL1 0 var modex
attempts to apply the effect to the image described in mfdbs with the PAL1 palette (this may be null for a monochrome or TC image).

Return:
0 : ok, effect applied! No second call necessary.
-1 : error, the effect cannot be applied to the described image
1 : module needs a second call without a palette, mfdbd describes the second necessary image of which we will have to reserve the block and provide the address in mfdbd.
2 : the module needs a second call with palette, mfdbd describes the second necessary image of which we will have to reserve the block and provide the address in mfdb d and an address of a sufficient area for the second palette.

For the second call: PAL1 PAL2 var modexe
PAL2 may be null in case of return 2 on the first call.

Return:
0 : ok, effect applied!
-1 : error.

Type 1 IFX

This is a rarer type of module, two source images are required.

Automatic method:

doifx

PAL1 PAL2 mfdb3 PAL3 var doifx
handles the two consecutive calls and returns:
0 : ok, effect applied to the first call
1 : ok, effect applied to the 2nd call without destination palette
2 : ok, effect applied to the 2nd call, PAL 3 contains the destination palette
-1 : error on first call
-2 : error on second call

Manual method:

modexe

PAL1 PAL2 addr 0 var modexe
Attempts to apply the effect to mfdbs/PAL1 and mfdbd/PAL2 images, addr is the address of a third mfdb (20 bytes) possibly used.

Return:
-1 : error, the effect does not apply to these images
0 : ok, effect applied, no other call to make.
1 : the module needs a second call without a palette, addr describes the third necessary image of which we will have to reserve the block and provide the address in mfdbs (?).
2 : the module needs a second call with palette, addr describes the third necessary image of which we will have to reserve the block and provide the address in mfdbs (?) as well as an address of a sufficient area for the second palette.

For the second call: PAL1 PAL2 adr PAL3 var modexe
PAL 3 may be null in case of return 2 on the first call.

Return:
0 : ok, effect applied!
-1 : error.

TRM management, screening module

This module allows transformations of image data from and to all possible formats. You can change the number of planes, the VDI or SHIfter format, dither in monochrome, in gray, zoom.

If it's just a matter of adapting the number of planes to the current resolution, here it is:

```
mdfbs mfdbd fillmfdb    \ copy source to destination
work_out 2- w@          \ get current planes number
mfdbd 12 + w!           \ and modify it in the destination mfdb
```

There are two modes, the automatic mode with the **dotrm function** and the manual mode with the **modinfo** , **modpal** and **modexe** triplet .

Automatic method:

dotrm

%bPRCCCID t pal-source pal-dest flag var dotrm

You must first fill the mfdbs with the data from the source image then the mfdbd with the expected modifications (without worrying about the address of the destination data, it will be allocated for you).

%bPRCCCID	: options, see details in modinfo below
t	: dithering method number (see modfmt)
pal-source	
pal-dest	: see details in modpal below
var	: the variable associated with the TRM module

Dotrm presents the mfdbs to the TRM module and tries the transformation, we get:

0	: Ok, data transformed in mfdbd
-1	: error in modinfo, incompatible options
-2	: error in modpal , transformation not supported
-3	: error in modex

If the transformation is successful, you have in the dictionary:

1 the source data block pointed to by mfdbs

2 followed by the transformed data block pointed to by mfdbd

It is possible that the source data is no longer useful, this is the role of the flag:

flag	: bit0=0	leave memory blocks as they are
	: bit0=1	if the source block is the last allocated in the dictionary, the function overlays the (lost) source through the destination and readjusts the dictionary end pointer.
	: bit1=1	force output in SHIFTER mode, if the TRM returns a block to the VDI format, then a call to vr_trnfm is made. This transformation covers the destination block.

Manual method:

modinfo

%bPRCCCID t var modfinfo
Set dithering parameters.

Options are a bitmask:

D, direction	: 0 in import, the source comes from a RIM and we want to display
	: 1 in export, the source will be exported with a WIM.
I , interact	: 0 automatic mode
	: 1 interactive
CCC, col.	: 000 destination palette imposed by the program
	: 001 palette proposed but not necessarily retained by the TRM
	: 010 gray palette (it will be copied for you)
	: 100 current palette imposed (it will be copied for you)
	: 101 current palette proposed (it will be copied for you)
R , overlay	: 0 the TRM can perform dithering over the original image
	: 1 forces the use of a second buffer (recommended)
P, palette	: 1 use a default palette from the TRM (if ccc=0/1)
	: 0 take the proposed palette
t, dither	: dither number from 1 to n (see **modfmt**) or -1 for **zoom quality** and 0 for **simple zoom** *(zoom available from TRM 210).*

Returns 0 if error or 1 if the parameters are accepted.

Attention: if you **impose** the palette, this can take a **long time** on a 68000.

modpal

pal-source pal-dest var modpal
manages the palettes and returns the size of the blocks needed.

mfdbs	: must be filled with information from the image
mfdbd	: also filled with the number of desired planes but the address data equal to zero.
pal-source	: image palette address, zero otherwise (for TC images)
pal-dest	: address of a block of 1536 bytes max with the imposed or proposed palette. According to the CCC modes (see modinfo), it is a palette already filled (modes 000 and 001) or just a zone where will be copied automatically by the modpal function i.e. the current palette (modes 100 and 101) or a palette of grays (mode 010).

In return:

<0 : error
0 : Ok, no block to reserve
>0 : Ok, the value is the size of the block to reserve , if this value is equal to %h7FFFFFFF, you must reserve the largest possible block and indicate the size obtained in the first 4 bytes of the block.

It is also necessary to look again at the address put in the mfdbd:

0 : then we must allocate an area to receive the destination image whose the size is calculated with:

```
mfdbd imagesize
```

and fill the address of the mfdbd with the beginning of this block.

=mfdbs : equal to the address in mfdbs, it means the TRM will overwrite the source image for its transformation .

modexe

buffer var modexe
performs dithering.

mfdbd : must be up to date with modpal requests.
buffer : address of the buffer requested by modpal.

Returns:

<0 : error
0 : success.

In return, the raster data is filled in and pal-dest contains the palette retained for this new image. This is not necessarily the proposed palette, the TRM may have adapted the colors for their needs.

Graphic card and TRM 2.52

This TRM can be configured to suit every special setting for 15, 16, 24 and 32 bits modes. But, you can let FORTH do it for you! To acheive that, the TRM must be set on the FORTHGFX card and:

```
" name"  -1   var  modload
```

Setting -1 for the flag tells FORTH to use the value **screen_info** to set correctly the TRM for the current display. An easy way to do this:

```
" name"  graphic_card negate   var  modload
```

If **graphic_card** returns zero, no change is made and if it returns 1, then you get the expected -1 for the flag. Using -1 on another TRM/GFX is equivalent to 1.

Managing a RSN, playing a sound

Once loaded in memory, you'll be able to use the RSN to load and replay a sound. The dialog with the module is partly through the stack and partly through a buffer named sndb (sound buffer).

sndb

Returns on the stack the address of the sndb:

0	LONG	sound identifier if supported ("SNDH", "WAVE",...) or zero if unsupported file type.
4	LONG	size of the source file
8	LONG	uncomrpessed size (=source if the file is uncompressed)
12	LONG	file handle
16	LONG	address of the loaded file
20	LONG	number of audio tracks in the file
24	LONG	number of the current track or zero if sound stopped bit 15 set if pause.
28	LONG	MIDI mask for ON/MUTE each track. Modpal sets all ON.
32	LONG	MIDI param1 with code -2 in modexe
36	LONG	MIDI param2 with code -2 in modexe
40	WORD	MIDI replay speed (default 100)
42	WORD	MIDI mask for ON/MUTE each channel. Modpal sets all ON.
44...		undefined
60	STRING	track title (31 characters max+zero)
92	STRING	track author (31 characters max+zero)
124	BYTE	free
125	BYTE	free
126	BYTE	internal flag used by FORTH
127	BYTE	internal flag used by FORTH

This block is filled by the module, the user can read it.

modinfo

handle var **modinfo**
check if the audio file is supported by the module.
handle: the audio file handle, opened by the user
var: pointer to the module

This function fills four entries in the **sndb** : the handle, the file size and the uncompressed size, and finally the file type.

Returns:
0 unknow audio, TYPE = 0
1 supported audio file, it is compressed.
2 supported audio file, it is uncompressed.

modpal

Note : *Although it is not a matter of palette, I kept the same name for this function as the one used by the images modules.*

adrs adrd var **modpal**
load the sound and manages different buffers.
adrs source address where to load the file.
if adrs=-1 ignore this bloc
if adrs=0 let FORTH manage the block creation in the dictionnary
adrd destination address where the audio will be uncompressed or modified.
if adrs=-1 ignore this bloc
if adrs=0 let FORTH manage the block creation in the dictionnary
var module pointer

After modinfo, the user can reserve itself the buffers, but the easiest way is to let FORTH do it all using 0 0 var modpal.

This function fills into the **sndb** the title, the author, the number of tracks and the file address.

Returned value:
0 error (can't read for example)
1 loaded and uncompressed. If adrs=0, FORTH frees the source block.
2 loaded.
-2 MIDI laoded but too many tracks (>32), in this case writing word 32 at file+10 allows you to use the first 32 tracks.

modexe

n var **modexe**
replay track #n from the audio file defined in the sndb.
n track number (from 1 to total tracks number)
var module pointer

Reserved values:
n = 0 stop sound
n = -1 get current track (or zero is sound stopped)
n = %h8000 toggles between play and pause. Value returned has the bit 15 set when in pause. Example 1 for track 1 playing and %h8001 for track 1 in pause.

Returned value:
-2 error, unknow file (can happen after uncompression)
-1 missing the required hardware to replay this sound
0 sound stopped
n>0 current track number

Specific options are available for the **MIDI module** :

```
n      flag -2 p modexe
```
with 0 ≤ n < 32 set track n to ON (flag=1) or MUTE (flag = 0)

```
n      flag -2 p modexe
```
with 32 ≤ n < 48 set channel n-32 to ON (flag=1) or MUTE (flag = 0)

```
%h100 s     -2 p modexe
```
set s as replay speed (default value is 100, set by modpal)

```
%h101 flag -2 p modexe
```
enable (flag=1) messages sys_ex or disable them (flag=0).

```
%h102 flag -2 p modexe
```
enable (flag=1) messages Pressure or disable them (flag=0).

```
%h103 flag -2 p modexe
```
enable (flag=1) messages Program Change or disable them (flag=0).

An option is available for the **XLR8 module**:

`%h102 flag -2 p modexe`

force 50Hz (flag=1) or not (flag=0), in this case the replay depends on the VBL frequency (50, 60 or 70Hz).

FORTH Modules management

Those modules, in spite of their use of the PARX M&E protocol, can't be called outside FORTH as they rely on its environment. From the FORTH point of view, the three calls to **modinfo**, **modpal** and **modexe** transmit internally the address of the **sndb** to the routines. This zone can be used for data exchange with the module, but it can ignore it. Another way to exchage data is directly the use of the FORTH stack.

During **modload**, the module receives in the 32 bytes preceding its own address adr:

- adr-4 : array of the FORTH words addresses, you can call them pushing and poping values from the stack pointed by the CPU register A6.
- adr-6 : user VDI handle
- adr-8 : FORTH internal VDI handle
- adr-12 : aespb address (giving access to all AES arrays)
- adr-16 : vdipb address (giving access to all VDI arrays)
- adr-20 : address of the variable containing the language (word)
- adr-24 : address of an extra block :
  - 0 : address of the word list (byte size+word, end if size=0)
  - 4 : number of words in FORTH
  - 8 : number of words at the beginning of the list that create other words
  - 12: VDI byte flag for coordinates (FF relative mode, 00 absolute mode)
  - 16: Long version (%h00xxyyzz for version x.y.z)
  - 20: address of work area block, the Forth dictionnary
  - 24: size of the dictionnary
  - 28: address of the 16 Forth flags (FF=true, 00=false)
  - 32: address of word variable with total number of words (base+user)

So, it's up to every module to define the usage of the calls **modinfo**, **modpal** and **modexe**, specifying if parameters are to be passed through **sndb** or through the stack.

Last feature available, a module can propose names to redefine the standard calls to modinfo, modpal and modexe. This can be done using the directive **>fom** before calling modload. An **extended Forth module** car offer several sub-functions for modexe, in this case using **>fom** is an obligation.

The modules must be located in the PARX.SYS folder, subfolder FORTH with the extension FOM (Forth Module).

>fom

>fom word1,word2,...,wordn

ask for the import of the specified words from the modules via modload. You don't have to import every available word, and if a word doesn't exist in any module, it will just be ignored. The word list must use the comma as separator and no space. The words must appear exactly as they are in the modules. Use **modlname** to list them.

Easy inclusion

The FORTH modules can be managed by hand, but a set of *.INC files makes it easy to use them. These are simple FOR files containing the required instructions to load the module and give access to its new words.

Example with the module DEBUG.FOM, you can simply use in your source code:

```
9 >include "%DEBUG.INC"
```

This will execute the necessary instructions contained in the file, hoping that you put it into the default *.FOR folder. Those files use the **flag 10**.

➔ If the **flag 10 is true** (default), loading is performed in **verbose mode**, then are displayed the module name, the result of modload and various other specific informations for that module.

```
10 >true  \ verbose mode
9 >include "%DEBUG.INC"
```

➔ If the **flag 10 is false**, loading is performed in **mute mode**.

```
10 >false \ mute mode
9 >include "%DEBUG.INC"
```

You'll certainly have to adapt the source code of the *.INC to fit your hard disk organization, where are located the FOM modules of PARX.SYS. The loading is an inclusion, this is meaningless under the interpretator, but if you compile your program, the module will be included into the binary, so the access to the PARX.SYS folder won't be required anymore if you spread out your software.

►The CALENDAR.FOM module V1.00

This module opens a dialog with a calendar from year 1583. You can modify the year, the month, the day, get the day of week, and, using the OK button, get in return a string representing the date in a format that can be used by the FORTH words dealing with the dates.

It offers two new words, **set_calendar** and **calendar** to replace modinfo and modexe.

First, let's use the standard calls:

Initialization :

```
variable p                              \ pointer to the module
" D:\PARX.SYS\" 1 modset                \ define the path
0 p !                                   \ let FORTH manage the memory
" FORTH\CALENDAR.FOM" 1 p modload . \ load and init
```

Return 0 (Ok)

Set language and format :

```
code format p modinfo .                 \ code = language
                                        \ 0=US, 1=GE, 2=FR, 4=SP, 5=IT
                                        \ format = 0 (MM/DD/YYYY),
                                        \        = other (DD/MM/YYYY)
```

Return 2 (Ok)

Open the calendar :

```
0 p modexe              \ if no date, opens with current date
if                      \ if return <>0, it's the OK button
      type              \ display the date string returned in pad
else
      ." Abandonned"    \ if 0, then Abandon button, no string
then

" 25/7/2024" p modexe   \ instead of 0, you can also specify a starting
                        \ date.
```

Using the interface :

Set the year and month and click on **Update**. The length of the month and the days of the week are updated.
The **arrow buttons** (next or previous year) perform the Update function too.

The **Ok** button returns the date in pad (on the stack) and a non null value at the top.
The **Abandon** button simply returns zero and no string.

To make it easier and more readable, the module proposes those two words that must be declared with **>fom** :

set_calendar

code format set_calendar *(replace p modinfo)*

calendar

date calendar *(replace p modexe)*

Here is the new program :

```
>fom set_calendar,calendar          \ ask for import
variable p                          \ pointer to the module
" D:\PARX.SYS\" 1 modset             \ define the path
0 p !                               \ let FORTH manage the memory
" FORTH\CALENDAR.FOM" 1 p modload . \ load and init

0 0 set_calendar drop               \ replace modinfo, set US and MDY

0 calendar if type then             \ open calendar and display the
                                    \ selected date.
```

►Extended Module BINARY.FOM V1.00

This module allows you to access every **binary function** of the 68000 that was not yet included in FORTH.
As it is an **extended FOM**, you must use **>fom** to declare the subroutines of modexe.
Those operations always keep track of the last bit (tested or moved) into the carry bit of the status register. You can get this value with **Bcarry** that will return 0 or 1. Some of the following operations can apply to a byte, a word or a long word, this is fixed by **Bsize**.

The default values set by **modload** are size = Long and carry = 0.

Bsize

size Bsize : size can be .b, .w or .l. Set the operand size of the rotation and shift operations.

Bcarry

Bcarry : returns the last tested or moved bit, 0 or 1.

Bcs

Bcs : Carry set : then carry = 1.

Bcc

Bcc : Carry clear : then carry = 0.

Blsl, Blsr, Basl, Basr

val #n Blsl : Logical shift Left
val #n Blsr : Logical shift Right
val #n Basl : Arithmetic shift Left
val #n Basr : Arithmetic shift Right

Those functions shift val by #n bits. They only apply to the part specified by Bsize and then carry equals to the last shifted bit.

With “Arithmetic” the sign of val is preserved, with “logical”, no.

Brol, Bror

val #n Brol : Rotate left
val #n Bror : Rotate right

Those functions rotate val by #n bits. They only apply to the part specified by Bsize and then carry equals to the last shifted bit.

Broxl, Broxr

val #n Broxl : Rotate left with extended
val #n Broxr : Rotate right with extended

Those functions rotate the group (val+carry) by n bits. They only apply to the part specified by Bsize and then carry equals to the last shifted bit.

Example, on a byte with a rotation to the left, imagine C next to the left of the byte:

C xxxx xxxx and the rotation operates on 9 bits.

This means that every bit of val goes through C, and the previous value of C returns to val on the right.

Bclr, Bset, Bchg, Btst

val #n Bclr : bit test and Clear (bit #n to zero)
val #n Bset : bit test and Set (bit #n to 1)
val #n Bchg : bit test and Change (bit #n changed)
val #n Btst : bit test (fills carry, val unchanged)

Those functions apply to bit #n of the long word val, n goes from 0 to 31. First, this bit #n is tested and copied into carry. Then, the operation on the bit is performed. Val remains on the stack, eventually changed.

Example, let's reverse the bit order of a long :

```
>fom Bsize,Broxl,Blsr   \ import only what's needed
variable p                          \ module pointer
" D:\PARX.SYS\" 1 modset             \ access path
0 p !                               \ let FORTH mamange memory
" FORTH\BINARY.FOM" 1 p modload .   \ load and init module

.l Bsize                            \ size LONG
%h12345678 0                        \ value to reverse and 0 to start
32 ndo
      swap 1 Blsr       \ shift right one bit and fills carry
      swap 1 Broxl      \ rotate left one bit and use carry!
nloop
uh.                     \ display the reverse value
drop                    \ drop the original value
                        \ that is equal to zero after 32 shifts...
```

►Extended Module CONFIG.FOM V1.01

This modules allows you to manage a text configuration file containing various lines based on this pattern:

key = value

You can read the config file with **CFGload**, save it using **CFGsave** and manage the keys: read them (**CFGgetkey**), modify/add (**CFGsetkey**) , or delete them (**CFGdelkey**). If the file doesn't exist, you can create an empty one in RAM with **CFGcreate**.

As with every extended module, you must use **>fom** to access to the words.

- In the file, there must be **only one key per line**, there can be spaces around the = or not, this makes no difference. The value runs to the end of the line.
- The line separators can be CR+LF or only one of them.
- Every line stating with **;** or with **** is considered as a **comment**.
- A key is a continuous set of characters with no spaces, the value can have spaces.

The file size can't exceed 64kB as the program uses word instructions for its loops.

CFGset

extra flag CFGset
Set some general parameters :
extra : size that will be added to the file size for the memory block, this is to prevent errors in case of an increase of size when you add keys.
flag : 1 for case sensitive when searching for a key
0 don't distinguish upper/lower case.

At start, extra=1024 and flag=1. There is no returned value.
If your file is only to be read, you can set extra = 0.

CFGload

" path\file" CFGload
load the configuration file in a block with "extra" more bytes (see CFGset).
Returns : 1 Ok, file laoded
0 Error, file not found or not enough memory.

You can use shel_find to build the whole path for a configuration file located in the application's folder.

```
" TEST.CFG" pad $!
pad shelfind
intout w@
if
      pad CFGload
      if
            ." File loaded!"
      else
            ." Loading error"
else
      ." File not found"
then
```

CFGsave

" path\file" CFGsave
save the current memory image of the config file.
Returns : 1 Ok, file saved
0 Error, can't write or unknown path.

CFGcreate

CFGcreate
create an empty configuration file in a block of "extra" bytes (See CFGset). This is to be used when your file is not found, or when you run the application for the first time.
Returns : 0 error (extra=0 or not enough memory)
1 Ok, block created.

CFGfree

CFGfree
Free the block, its content is lost! You can use CFGload again.
Returns : 0 error
1 Ok

CFGgetkey

" key" CFGgetkey
Look for the specified key and return:
0 : not found
1 : found, and under on the stack, pad with the string value.
A key can be simply :
KEY=
With an empty value, CFGgetkey returns 1 (key found) and an empty string in pad. This is equivalent to DEFINE in C, the word exists but has no value. A flag.

For the returned string, the routine removes the trailing spaces.

CFGsetkey

" key" " value" CFGsetkey
Give a value to a key.
If the key doesn't exist, it is appended to the end of the file.
If it exists, its value is updated and the old content is lost.
Returns 0 not enough memory for the new key
1 Ok, key was added/modified.

In case of error, it's up to you to save, free and reopen the configuration file with again "extra" more bytes:

```
" key"      " vaue"    CFGsetkey
0= if
      " path\file" CFGsave drop
      CFGfree drop
      " path\file" CFGload drop
then
```

CFGdelkey

" key" CFGdelkey
Removes the key and its value from RAM, not on the disk unless you save it!
Renvoie 0 the key doesn't exist.
1 Ok, key removed.

Note : The functions CFGgetkey, CFGsetkey, CFGdelkey **use pad.**

►Extended Module RTF.FOM V1.03

This module allows to export the whole content of the editor (or the marked block) to the **RTF (Rich Text Format)** format for both the Atari system (Atari Works for example) and the PC system (Wordpad for example).

The style will be this :

✔ the **FORTH words** appear in bold
✔ the created words are underligned when created
✔ the *comments* appear in italic

For use in the **Atari** system:

◆ either the default mode (System Font) and the text attributes modify the aspect of the words.
◆ either a list of 3 fonts to represent the normal, bold and italic.

For use in the **PC** system:

◆ either the default mode (Arial family font)
◆ either you specify another family (Courier for example)
◆ in both cases, the text attributes allows the PC to select the correct font within the family.

RTFmode

n RTFmode : select the Atari mode (n=0, default) or PC (n<>0). This call modifys the behavior of RTFfont, so it must be used prior to RTFfont. Returns 2 for Ok.

Note: if the PC is selected than most characters have their ASCII code correctly translated (for example those with accents).

RTFfont

According to the mode selected with RTFmode, for the Atari :
0 RTFfont : select default mode (Système Font)
" name" Id n RTFfont : set the font for normal (n=1), bold (n=2) or italic (n=3).
*(name and Id are those returned in pad and intout by the function **vqt_name**)*

For the PC :
0 RTFfont : select default mode (Arial family)
" name" RTFfont : set another family name for fonts.

Returns 2 if Ok, and 0 if error (bad value of n, must be 0,1,2,3).

RTFsave

" path\file" RTFsave: save the source (or the marqued block) using the format specified by RTFmode and RTFfont.

If you want to save a FOR file different from the one loaded into the editor, you must use the **sndb** for additional informations:

```
" .RTF" @ sndb !   \ 4 first bytes of sndb with a magic value
addr sndb 4 + !    \ the next for bytes are the address of a complete FOR
                   \ file followed by, at least, two nul bytes if
                   \ version<1.03, else those bytes are useless.
```

Returns 2 if Ok and 0 if error (file error). If the **sndb** was used then the magic value is cleared.

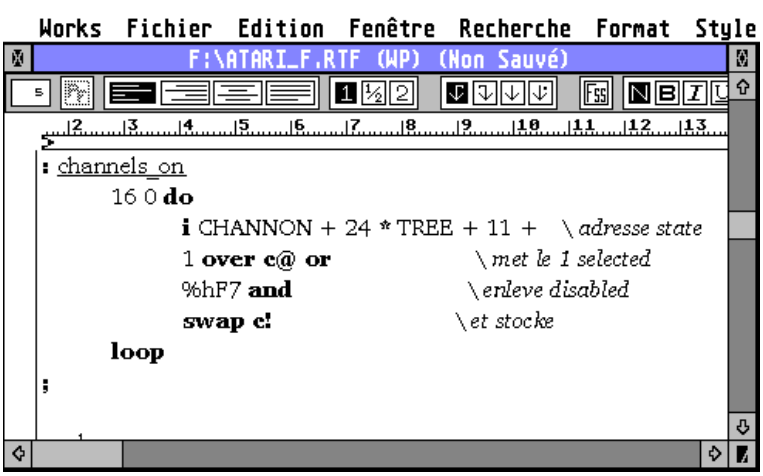
RTFtab

val RTFtab : defines the size of one tabulation. Default is 720 but 400 is closer to the aspect under the editor.

An example : The first part loads the module and create in memory the word “go”. Then, the editor is cleard with buf! and loadb loads a new source, the one to be saved as RTF ! Running “go”, saves the file with four different settings:

```
variable p
>fom RTFsave,RTFmode,RTFfont
" D:\PARX.SYS\" 1 modset
" FORTH\RTF.FOM" 1 p modload .

: go
   ." Saving for Atari : "
   0 RTFmode . space
   0 RTFfont . cr
   ."    Default   : "
   " F:\ATARI_D.RTF" RTFsave . cr
   ."    With fonts: "
   " Bitstream Charter"        5648 1 RTFfont . space
   " Bitstream Charter Black"  5709 2 RTFfont . space
   " Bitstream Charter Italic" 5649 3 RTFfont . space
   " F:\ATARI_F.RTF" RTFsave . cr
   cr
   ." Saving for PC    : "
   1 RTFmode . space
   0 RTFfont . cr
   ."    Default (Arial) : "
   " F:\IBM_D.RTF" RTFsave . cr
   ."    Another family  : "
   " Courier" RTFfont . space
   " F:\IBM_F.RTF" RTFsave . cr
;
>comp
buf! loadb go
```

Here is what you can get under **Atari Works** with the Atari mode and the three fonts (Bitstream Charter).

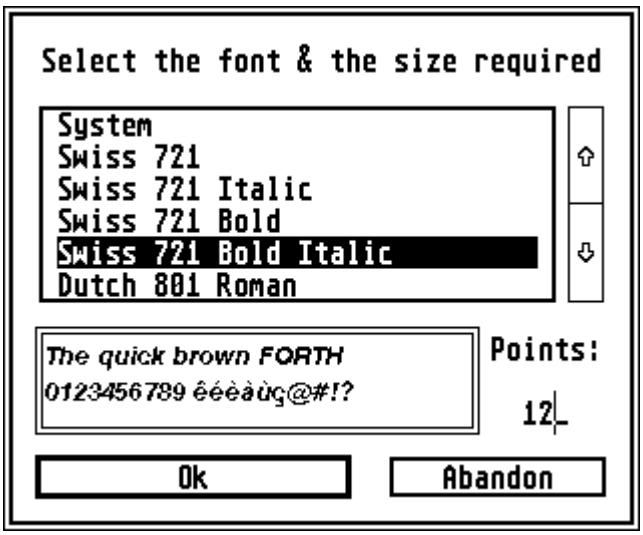


►Module FONTSEL.FOM V1.02

This modules allows you to select a GDOS font through a dialog.

It only works if a GDOS font manager is installed on your system (GDOS, SpeedoGDOS, NVDI,...).

It requires a first call to **FSinit** to set its behavior, then an optional call to **FStitle** to personalize the dialog's title, and last the main call to **FSopen** that actually opens the font selector.



FSinit

flag var FSinit : set the parameters.
flag 0 the module must perform a call to vst_load_fonts
n vst_load_fonts has been called yet, n is its returned value.
var 0 no exchange variable
var 4-bytes variable used in input and output with the current value of the size that appears in the dialog.(max 999).

Returns : 0 error (GDOS is not installed)
n number of available fonts (including the system font)

FStitle

" string" FStitle : personalize the dialog's title. 35 characters maxi.

Returns 2 Ok.

FSopen

flag FSopen : opens the selector.
flag 0 no preselected font
n open with font n preselected.

Returns 0 Abandon, or no selected font
n number of the selected font (that you can use with **vqt_name**).

►Extended module DIGIDRUM.FOM V1.00

This module allows you to play drums mixing two channels. The Yamaha is used as the standard output, but you can also use several sound cards. You can define simple patterns that loop or a complete song based on patterns.

Apart from laoding the module, you also have to declare a drum kit composed of a maximum of 16 samples, 16 kB each at 20kHz, 8 bits, mono, this is done using **DDkit**.

Then you have to create a string array containing the patterns. They are very simple descriptions, as a text, of the two channels, this array is given to **DDpatterns**. This can be satisfying to replay simple looping scores. A more elaborated solution is to program a song based on several patterns, in order to do this you must create an integer array that will receive the programmation of the song, this array is given to **DDsongs**.

DDkit

addr DDkit set the address of the drum kit. Upon return, pad contains a string describing each sample with its associated letter code (from A to P). Pad is not left on the stack.

DDtempo

n DDtempo set the tempo in 1/50th of second, n runs from 1 to 49. This gives the time for each beat, default value is 6 (8,33 beats per second).

DDfreq

n DDfreq set the quality of the replay.
- n=0 6,6 kHz, low quality, but more CPU time.
- n=1 10 kHz, average quality, less CPU time left.
- n=2 20 kHz, high quality but few CPU time left.

Default value is n=2.

DDoutput

n DDoutput set the output device.
- n=0 Yamaha (low quality, intense CPU use)
- n=1 Psound card (Average quality)
- n=2 ST Replay 16 card (High quality)
- n=3 ST Replay 8 or MV16 card (High quality)

Default value is n=0.

DDpatterns

addr DDpatterns set the address of the first element of the string array describing the patterns. The strings are used by pairs as there are two channels. The lenght of the strings sets the maximum niumber of beats per pattern.

Example :

```
32 100 array$ PATT        \ 100 strings of 32 beats maxi
0 PATT DDpatterns         \ set the first element address
```

DDplaypatt

n DDplaypatt replay and loop the pattern defined by strings n and n+1. They must share the same length!

Example (with the default kit) :

```
" --E--E" 2 PATT $!             \ channel 1 the closed hit hat
" A--B--" 3 PATT $!             \ channel 2, bass and snare
2 DDplaypatt                    \ replay a boogie woogie
```

DDstop

n DDstop stop the current sound according to n :
n=0 immediately
n=1 at the end of current pattern

DDsongs

addr DDsongs set the address of the first element of the song array. You can estimate the required length according to the song complexity. The first 10 elements are reserved for labels and then each instruction can use a maximum of 2 elements. Let's say that (2×instructions number + 10) is a good value.

Example :

```
100 array SONG        \ an array with 100 values,
                      \ enough for 45 instructions.
0 SONG DDsongs        \ and set the first element address
```

DDplaysong

n DDplaysong replay a song starting at the element n of the array.

Programming a song :

To program a song, you have to define how the existing patterns are repeated or chained together. You can also use pedals plugged into the joystick port and that can use the four directions bits.

#new

flags #new : clears the song and init the programming pointers. The flag is a bit mask describing the initial position of the pedals. This is used to adapt the module to every situation. For example, my own pedal says “1” when released and “0” when pressed.

```
flag    right  =    %b1000
        left   =    %b0100
        up     =    %b0010
        down   =    %b0001
```

#here

l #here : defines the current position as label l (value from 0 to 18).

#where

l #where : returns the array element pointed to by label l.

#patt

r n #patt : repeat n times the pattern n.

#gosub

r l #gosub : execute n times the instructions at label l.

#ret

#ret : returns from a gosub or ends the song if no gosub is in use.

#tempo

n #tempo : modifies the current tempo, see DDtempo.

#tempo+ #tempo-

n #tempo+ : add n to the current tempo (or subtract if #tempo-).

#mute

n #mute : insert a silence of n beats before the following instruction.

#pedal

" mask" l #pedal : go back to label l until the condition specified by mask is satisfied. The mask is a string of exactly four characters for the four movements of the joystick. For each, you can use:

- x bit ignored
- 0 the bit must be zero
- 1 the bit must be 1
- o (letter O lowercase) a set of bits where one of them at least must be one.

Exemples :

```
" xx0x" l #pedal   :back to l until UP is zero
" xxx1" l #pedal   :back to l until DOWN is 1
" ooxx" l #pedal   :back to l until RIGHT or LEFT is 1
" oooo" l #pedal   :back to l until one bit is 1
```

#goto

l #goto : jump to label l.

Example :

```
" --F--F" 0 PATT $!      \ boogie woogie open hit hat
" A--B--" 1 PATT $!
" --E--E" 2 PATT $!      \ boogie woogie closed hit hat
" A--B--" 3 PATT $!

%b0010 #new              \ new song, pedal with UP=1 when released
                         \ and that send “0” when pressed
0 #here
   1 2 #patt             \ verse with closed Hithat
   " xx0x" 0 #pedal      \ until pedal pressed
2 #here
   1 0 #patt             \ chorus with open Hithat
   " xx0x" 2 #pedal      \ until pedal pressed
   0 #goto
1 #here                  \ song starts here
   8 #tempo              \ set tempo
   0 #goto               \ and start with verse, no ending!
1 #where DDplaysong      \ start playing at label 1
```

►Extended module DEBUG.FOM V1.03

This module allows you to use a second screen to send debugging information without using the current display. What you need is:

- A TT with a graphic card, the second display is in ST High, monochrome on the standard VGA output.
- A MegaSTE with a graphic card, the second display is in ST High monochrome or ST Medium according to the attached monitor.

If the machine has no graphic card, then a 32000 bytes buffer is allocated and you can swap from the user screen to the debug screen using the Alt+HELP combination:

- A TT, the second screen is always in ST High.
- A ST(e), the second screen is ST High on a monochrome monitor and ST Medium on a color monitor.

Calling **modload** initializes the second screen and returns a bit mask:
bit#0 : 1=system font 8×8 available
bit#1 : 1=system font 8×16 available
bit#2 : 1=Second screen ST High, 0=ST Medium
bit#3 : 1=GEM on a graphic card

When quitting the program, or calling **modunload**, a message tells you that the module has been uninstalled (**do_exit** routine is present). You have to validate with CapsLock, if a buffer was allocated, then it is freed.

Dfont

n Dfont select font according to n :
n=8 ask for system font 8×8
n=16 ask for system font 8×16
n=addr is the address of a font header with size 8×h and defining the whole 256 characters set

returns 2 if Ok, 0 if the font is unavailable or with bad characteristics.

Dauto

mask Dauto set the automatic behavior for display :
bit#0 : 1=auto-spacing after a number (default 1)
bit#1 : 1=auto-newline after a number (default 0)
bit#2 : 1=auto dup for numbers (default 1), else removed from stack
bit#3 : 1=auto dup for strings (default 0), else removed from stack
bit#4 : 1=auto Dback for dump/+/- (default 0)
bit#5 : 1=catch bomb errors (default 0)

-1 Dauto just return the current mask value

Dconfig

display an intercative menu to easely adjust the settings of Dauto. Use up and down arrow to select the line and Space, Enter or right and left arrow to swap between ON and OFF. Use Escape to quit.

Dcls

clears second display

Dcr

new line and eventual scrolling on second screen

Dspace

display a space

D. Df.

display top of the stack in decimal (for D.) or floating point (for Df.) (see Dauto for more)

Dh.

display top of the stack in hexadecimal with 8 digits (see Dauto for more)

Dtype

display string on top of the stack (see Dauto for more)

Dnewl

back to the start of current line and clears this line.

D.s

display the stack content, in decimal.

Dkey

pause and wait for CapsLock (its state is then restored), a right arrow is displayed to indicate that a key press is required, the arrow is then removed.

If you have no graphic card, then Dkey swaps to the debug screen if it was not displayed. In this case, upon exiting this function, the user screen is restored unless you have used Alt+Help during the function.

Dump

addr n Dump display the content of memory starting at address addr with n lines of 16 bytes each. On a line, the bytes appear both in hexadecimal and in ASCII.

The supervisor mode is used for reading, so protected parts of memory are accessible.

Dump+ Dump-

reuse the last settings of Dump and display the next (Dump+) or previous (Dump-) n lines in memory.

Dsave Dback

save current cursor position (Dsave) or restore it (Dback). See Dauto for the auto Dback for Dump/+/-.

This system has only one level, so if you call Dsave twice, then the first saved position is lost.

Catching bomb errors:

When such an error occurs, you get on the debug screen the type of error, the PC value (address where the error happened), the value of the 16 CPU registers and information about the localization of the PC and the A5 register that is the interpretation pointer in FORTH.

The CapsLock key must be pressed to go back to the command line.

When this is done, stacks are cleared and the program is stopped.

Note:

- most of the time, the PC address is the one of the instruction following the error (except for "illegal instruction").
- on a 68000, the PC value for "bus error" or "address error" is not precise and can be several bytes away form the actual error.

►Module QRCODE.FOM V1.00

This module allows you to generate a QR-code from any text. All versions (from 1 to 40) are supported with any error recovery level (from 0 to 3) and any mask to enhance readability (from 0 to 8).

It can encode in **NUM** (only digits, versy compact), in **ALPHA** (upper case, digits and everything to write an URL, compact) and in **BIN** (binary mode, all characters allowed, not so compact). For now, Kenji characters are not supported.

You must start with **QRinfo** that returns the size of a required buffer. Then **QRcode** creates the image (with an optional icon centered) returned in a format directly usable by a VDI call for display. Last, **QRsave** allows you to save to disk a file with two available formats: IMG or BMP.

QRinfo

mess err QRinfo : prepares the encoding of the message mess (a null-terminated string) with the error recovery level err, this makes a QR-Code readable even if it is partly damaged.

Note: the address of the message must not change from QRinfo to QRcode, the string can't be a constant.

err = 0 for 7% error recovery
1 for 15%
2 for 25%
3 for 30%

You can easely understand that, the higher err, the larger the QR-Code.

Returned value:

-2	bad err value (not in 0-3)
-1	empty message
2	Ok

Moreover, the **mfdbs** contains the following informations:

+0	LONG	buffer size for **QRcode**
+4	WORD	selected version (1-40)
+6	LONG	4-chars encoding NUME, ALPH, BYTE, KENJ
+10	WORD	encoding (1 num, 2 alp, 3 byt, 4 ken)
+12	WORD	err

QRcode

buf QRcode : generate the QRcode using the given buffer whose required size was returned via the **mfdbs**. To be simple, if you want a fixed buffer for every possible case, a size of 80KB is enough.

Upon entry if:
buf+0 = 'ICON'
buf+4 = addr
then this is the address of an ICONBLK structure describing an icon that will be added in the center of the QR-Code.

Returned value:
-1 bad encoding mode (if you modified mfdbs+10)
2 Ok

Furthermore, **mfdbs** if filled and points to a monochrome image bloc.

+0	LONG		address of the pixels (somewhere in the buffer)
+4	WORD	w	width in pixels
+6	WORD	h	height in lignes
+8	WORD	W	width in words = (l+15)/16
+10	WORD	0	format shifter
+12	WORD	1	number of planes

If you want to display it on screen at position x,y :

```
0 mfdbd !                          \ destination screen
0 0                                \ position 0,0 in the source
x y                                \ position on screen
mfdbs 4 + w@ mfdbs 6 + w@          \ width and height
1 1 0                              \ mode replace, colors black/white
vrt_cpyfm                          \ copy block
```

QRsave

" path\file" i QRsave : save the QR-Code image on disk under the specified name and according to i :
i = 0 format Atari IMG monochrome
1 format PC BMP monochrome

Returned value:
-2 file error
-1 bad i value (not 0,1)
2 Ok, file saved

The Pre-processor

1 Including a file when compiling

One can assemble a routine beforehand and include it in a FORTH program at compile time, in conjunction with COMP.PRG, this will give a one-piece program (the file is included once and is part of the executable). It is of course possible to include something other than executable code.

For the executable code , rather than choosing an object format (of certain compilers) since they are not all equivalent, I chose to include Atari executable files directly.

>include

m >include " path\file.ext"

includes a file inside the program at the current dictionary location. The command word m is a bitmask:

bit 0 : 0=a PRG file
1=other

bit 1 : 0=ensure pointer parity after loading
1=ignore parity

bit 2 : 0=relocate absolute addresses
1=do not relocate

bit 3&4 : valid only if bit0=1 (Other format)
00 = general file
01 = a FORTH source file to include
10 = a PARX module
11 = a DEF file (constant definitions associated to a RSC file)

In case of a **FORTH source file**, the file is not included as is, but interpretated. If it contains executable parts, they will be executed, if it contains definitions they will be added.

Several levels of inclusion are possible.

In case of a **DEF file**, all names with 8 characters maxi from a RSC file (created with INTRFACE or ORCS) are added as constants with their values.

See below the different possibilities offered by this command.

Including a routine in an assembled word

Example, in a word I want to shift a value 3 bits to the right (fast division by 8). I am preparing the following program in assembler:

```
      text

      move.l (a6),d0    ; the value taken from the stack
      asr.l #3,d0       ; shifts
      move.l d0,(a6)    ; put back on the stack

      end
```

Which I assemble to C:\DIV8.PRG

All that remains is to write in FORTH:

```
:: test
      p @
      0 >include "C:\DIV8.PRG"
      p !
;
```

This word will divide the variable p by 8!

Remember:

** no spaces in the filename string*

** the TEXT and DATA sections are included in the definition, it's up to you to 'jump' over the datas if the word must continue.*

Including assembly word definitions

If the entire word must be defined in assembler, a particular structure of the PRG file must be used:

* the TEXT area contains definitions and data

* the DATA zone contains the addresses and names of the words created, it is therefore reserved.

Example, the following words can be prepared in assembler:

```
      text
plus8:
      dc.w 437          ; all the words assembled in FORTH begin
                        ; by this code.
      addq.l #8,(a6)    ; adds 8 to the value on the stack
      rts
```

```
minus8:
      dc.w 437
      subq.l #8,(a6)    ; subtracts 8 from the value on the stack
      rts

hello:
      dc.w 437
      lea string(pc),a0
      move.l a0,-(a6)   ; stack the address of a string
      rts

      dc.w 26           ; in FORTH, a string is preceded by its
                        ; maximum length.
string: dc.b "Hi, I'm assembled!",0
      even

      data

      dc.b "ADD_DEFS"         ; magic word for a series of definitions
      dc.l plus8,minus8,hello,-1
                              ; sequence of addresses and -1 to end
      dc.b "8+",0,"8-",0,"hello",0
                              ; sequence of names separated by a '0'

      end
```

Which I assemble to "C:\DEFS.PRG"

Just add in the FORTH program:

```
0 >include "C:\DEFS.PRG"
```

to have the words '8+', '8-' and 'hello'.

Remember:

** no spaces in filename string*
** only the TEXT area is included, the DATA area is simply read but not kept.*
** always leave bit 2 of the command word at 0 if your program uses absolute addressing.*

'Raw' inclusion of a file other than PRG

The problem lies in retrieving the address at which the file was uploaded. The easiest way is to use one of the FORTH internal codes, it is **[var]**, used in a variable.

A variable is constructed like this:

[var]
+value on 4 bytes

The action of [var] is to provide on the stack the address of the area immediately following it!

Thereby:

```
: block
      [var]
      1 >include "C:\IMAGE.IMG"
;
```

will include an image whose address is obtained by running block. Despite the use of: ... ; (interpreted mode), the word block is perfectly assemblable because it is recognized as a variable (it starts with [var]!).

No word will be executed after [var] because it contains in itself the equivalent of ';' (end of word).

Thereby:

```
: block
      [var]
      1 > include "C:\IMAGE.IMG"
      16 +
;
```

will not return the address of the image increased by 16, because the end of the word is reached with [var].

Including a PARX module

A PARX module requires 32 free bytes just before itself in memory for the local variables. The inclusion makes your application independant of the PARX.SYS folder. The procedure is easy, you create a word containing the module (with **>include**). This word returns the module's address that will be used to perform the initialization with **modload**.

```
variable p        \ pointer to the module
: my-module
      17 >include "path\module"     \ 17 to load a PARX module
;
my-module p !     \ get the address into p
1 p modload .     \ and init the module
```

Summary

** to **include a program** the command word is 0*
** to include and compile another **FOR file**, the word is 9*
** to include a **DEF file**, the word is 25*
** to include a **PARX module**, the word is 17*
** to include another file, the command word is 1*
** if it is a program:*
*the DATA section starts with "**ADD_DEFS**":*

include the definitions of the TEXT zone whose
addresses and names appear in the DATA zone.
the DATA section does not start with "ADD_DEFS":
include the TEXT and DATA fields as
part of a definition already started.

** use bit 1 of the command word (ignore parity of the dictionary) only knowingly: on a 68000, word accesses at an odd address cause a bus error, on all 680X0s, an assembler instruction must begin at an even address.*

For further

You can define variables in a PRG file:

```
      text

var:
      dc.w 140          ; this is the code of [var]
      dc.l 1234         ; the variable is initialised to 1234

      data
      dc.b "ADD_DEFS"
      dc.l var,-1
      dc.b "my_variable",0

      end
```

We can do FORTH in assembler...

```
      text

forth:
      dc.w 96           ; 'pad' code
      dc.w 92           ; code of 'len'
      dc.w 124          ; code of '.'
      dc.w 79           ; code of ';'

      data
      dc.b "ADD_DEFS"
      dc .l forth,-1
      dc.b "pad_len",0
      end
```

Is equivalent to:

```
: pad_len
      pad len .
;
```

which displays the length of the string contained in pad.

One can also mix in the same word an interpreted part and an assembled part:

```
      text

      move.l (a6)+,d0         ; the value on the stack
      btst #0,d0              ; the units digit
      beq.s .lb0
```

```
      moveq #0,d0             ; returns 0 if the number is odd
      move.l d0,-(a6)
      rts
.lb0:
      moveq #1,d0             ; returns 1 if it is even
      move.l d0,-(a6)
      rts

      end
```

Assembled to C:\EVEN.PRG.

Next:

```
: parity
      p @ [ass]
      0 >include "C:\EVEN.PRG"
;
```

will return 0 or 1 depending on whether the value of p is odd or even. As for [var], the code [ass] (the one which is worth 437) contains in itself the equivalent of ';', thus, it is the last word executed in parity. We cannot call parity from an assembled word (since the beginning of its definition is interpreted).

Use variables defined in FORTH

If the variable is defined in FORTH as part of the program, there is the problem of providing its address to the assembler part if it must access it. The easiest way is to define an 'init' routine to which we will give this precious information which will be saved and accessible to other routines:

```
      text

init:
      dc.w 437
      lea addresses(pc),a0
      move.l (a6)+,(a0)+      ; retrieve two addresses and store them
      move.l (a6)+,(a0)+
      rts

addresses: dc.l 0.0           ; room for two addresses: a variable and pad

caps:
      dc.w 437
      lea addresses(pc),a0
      move.l (a0)+,a1         ; variable address
      move.l (a0)+,a2         ; pad address
      move.l (a6)+,a0         ; channel address
      move.l a2,-(a6)         ; returns pad on the stack
      moveq #0,d1             ; 0 characters to start
.lb0:
      move.b (a0)+,d0         ; a character
      beq.s.lb1               ; if null, end of string
      addq.l #1,d1            ; count characters
      and.b #$DF,d0           ; to uppercase
```

```
      move.b d0,(a2)+          ; in pad
      bra.s.lb0                ; Carry on
.lb1:
      move.l d1,(a1)           ; string length returned in variable
      rts

      data
      dc.b "ADD_DEFS"
      dc.l init, uppercase, -1
      dc.b "init",0,"uppercase",0

      end
```

Assembled to "C:\DEFS.PRG".

Thereby:

```
variable length

0 >include "C:\DEFS.PRG"

>comp

pad length init          \ allows the word 'uppercase' to run correctly

" Hello" uppercase       \ returns to the pad stack containing "HELLO" and the
                         \ length 6 in the length variable.
```

2 Prepare the assembler source in the Editor

To facilitate the debugging of a program, the assembly text can be written in the FORTH editor, exported and assembled during compilation:

>export

```
m >export "path\file.ext"
....
>comp
```

The word >export makes it possible to send in a text file all the lines following it until the next >comp (which must be the first word of its line!) The command word m contains only the value of the flag (see > true and >false).

3 Using an external assembler

Once the code has been typed in the editor and exported, it can be assembled directly from FORTH during compilation, it is the word >exec which is responsible for calling the assembler.

>exec

m >exec "path\prog.ext"
"command line"

executes the specified program by passing it the command line (which must be on the following line).

the word m:

bit 0 : 0=key pressed after program execution
1=don't wait
bit 1 : 0=ERROR if the program returns a positive value
1=don't report an error
bit 2 : 0=ERROR if the program returns a negative value
1=don't report an error
bit 3 : 0=close FORTH window during execution
1=ignore window

In any case, a null value returned by the assembler does not generate an error. The rule would be that only negative values correspond to errors, but some programs return a positive code. Thus, bits 1 and 2 make it possible to adapt to all situations.

In the event of a key wait (bit 0 = 0), pressing 'ESC' interrupts the processing as an error.

4 Conditional Compilation, Flags

If a FORTH program is compiled several times in succession without having modified the exported parts, it is useless to re-export them and re-assemble them with the commands >export and >exec.

Thus, we have a rudimentary but sufficient conditional compilation. The command word 'm' of the three instructions >export, >exec, >include, can contain the number of a flag to be tested to validate or ignore the action. This flag is signaled in the upper word (bits 16 to 31) of 'm':

>true

n >true : validates flag n (from 1 to 15). At startup, all flags from 1 to 12 are

'true' by default.

Flag 13 : true under the interpreter and false under the compiler.
Flag 14 : true on a 68030 version, false on a 68000 version
Flag 15 : true on a French version, false on the English.
Flags 13 to 15 can be redefined.

>false

n > false : invalidate flag n (from 1 to 15).

examples:

```
%h40000 >export "f:\test.s"
```

will only export if flag 4 is true. By taking the opposite of 'm', we only validate the operation if the flag is false:

```
%h-20008 >exec "F:\assemble\asm.ttp"
"f:\test.s"
```

only assembles if flag 2 is false, the 8 corresponds to bit 3, do not close the FORTH window during execution.

>ask

n >ask "prompt" : at compilation time, display the prompt followed by (Y/N) and wait for a key. If you press Y, the flag n will be true, else it will be false.

>ifflag

>endf

n >ifflag : continue compiling if flag n is true,
otherwise jump to next >endf.
-n >ifflag : continue compiling if flag n is false,
otherwise jump to next >endf.
In both cases, >endf must be the very first word on its line!

5 Other External Links

We can as well export something other than assembler, execute something other than ASM.TTP. One can imagine that a FORTH program needs an image computed by POV:

```
0 >export "F:\POV\image.pov"              \ send script to disk
...
here is the script of the image
>comp

0 >exec "F:\POV\pov30_82.ttp"             \ calculate the image
"-iIMAGE.POV -oIMAGE.TGA +w100 +h100"     \ in 100x100

: picture
      [var]
      0 >include "F:\POV\IMAGE.TGA"       \ include it in the program!
;
```

The word image will then deposit the address of the file in RAM.

6 Default paths

When using the directives >include, >export et >exec, you don't have to specify the whole access path of a file if it's the same as the default path for the sources files FOR (take care, this one can be modified when using the fileselector with saveb!).

To append the default path, just use the % character. For example, if my default path is "F:\FORTH\SOURCES*.FOR", then :

```
0 >include "%HELLO.FOR"
```

will load: F:\FORTH\SOURCES\HELLO.FOR

```
0 >export "%ASM.S"
```

will export F:\FORTH\SOURCES\ASM.S

```
0 >exec "F:\ASSEMBLE\ASM.TTP"
"%ASM.S"
```

will give F:\FORTH\SOURCES\ASM.S as a command line to ASM.TTP.

For >exec, you can only use % in the command line.

Should the character % appear in a line, just double it: %% !

The compiler

This separate program, the FORTH.LIB library, to generate an independent executable program from a source file created under the FORTH interpreter.

1 Install

For this you need to have (68030 version / 68000) :

COMP.PRG	/	COMPSTE.PRG	the compiler
FORTH.LIB	/	FORTHSTE.LIB	the library
COMP.INF	/	COMPSTE.INF	some general settings

As for FORTH.PRG, you can use **FORTHINS.PRG** to create and modify your COMP.INF file. It's easier!

comp.inf

is a 4 or 5 lines text file containing the following information:

Optional line
This line must start with a **#** followed by some flags. If absent, then the default parameters will be applied.

"**L**" will generate a LOG file (COMP.LOG or COMPSTE.LOG) that duplicates every line displayed on screen. Default is no LOG file.

First line
Size to reserve for the compilation (the source+FORTH.LIB+the relocation table+the compiled part itself) in the format:
%xxx+ (or -)yyyyy, ie xxx% of free memory plus (or less) yyyyy KB. For example %100-400 will reserve everything minus 400KB. If the code contains the >exec command (calling an external program), it will absolutely be necessary to leave space.

Second line
path to *.FOR source files

Third line
full path to forth.lib

Fourth line
access path to the directory of generated executables (*.PRG).

A possible file is:

```
%50+100
F:\FORTH\SOURCES\*.FOR
F:\FORTH\FORTH.LIB
F:\FORTH\PROGRAMS\*.PRG
```

2 Use

To compile a program, all you need to do is create a source file under the interpreter (*.FOR) and run COMP:

1) in the selector, choose the source file to compile (if the file is not found, a second attempt is done forcing the extension to .FOR)

2) the window displays the operations carried out:
- if an error occurs it is signaled and the processing interrupted
- if all goes well, a selector opens again

3) indicate the name and the path for the program to be written

The program is now available on disc. Do not worry about its impressive size for a source of a few lines, the size increases little thereafter with a larger source.

Warning: Under the compiler, you cannot use the *.TXT files obtained with export, it is imperative to use the *.FOR files obtained with saveb or append.

If your source uses assembled definitions (with :: xxx ;), then the window title line displays (if there have been optimizations):

Optim #Imm:00000 1>2:00000

giving the optimization statistics. The first number are optimizations related to the use of constants as arguments and the second are chaining optimizations where the output value of a function is retrieved by the next one without going through the stack.

Tip: *if one **SHIFT key** is pressed when the compiler is launched, then this forces the **creation of a LOG file** without having to modify the INF file.*

3 Pitfalls to avoid

A program running perfectly under the interpreter can sometimes behave surprisingly when compiled. Here are some common errors:

1°) I write under the editor this:

```
: plus
      fastopen drop
      1 2 + .
      key drop
;
```

I compile it with compilb then I run it, it shows me 3 under the interpreter. I pass it to COMP and the executable does nothing (immediate return to desktop).

What my source is missing is the 'execute the word plus' operation that I was running by hand under the interpreter. It is therefore necessary to add to my source a line containing the word to be executed:

```
plus
```

In fact a compiled program behaves like the interpreter, and this source did not contain any executable part, only a definition. As a result, the program just saved the definition and couldn't find anything to run!

2°) The compiler (as well as the interpreter) only deals with lines validated by Enter, so there must be an empty line following the last line of text. In the absence of this phantom line, the last commands of the program could be forgotten.

3°) Under the interpreter there is always an open window, the use of Fastopen is therefore not necessary, on the other hand for an independent program this is essential before any graphic output (texts or others), moreover fastopen returns the handle of the window, which allows you to customize it (by changing the title for example with wind_set).

4 "Normal" programs

A program will be said to be "normal" when it is run from the desktop and not from the AUTO folder or as an accessory. In this case an internal VDI station is opened (for simple text outputs) as well as an external station for the user with the identifier in the word handle variable.

In this case , fastopen opens a window similar to that of the FORTH interpreter and fastclose closes it (useless at the end of the program, because the exit routine ensures the closing before returning to the desktop).

In general, the size of the area reserved for the dictionary is sufficient for all applications, but in some cases the use of "reserve" may be necessary (see the next paragraph). This **default size** is equal to **three quarters** of the free memory.

5 TOS/TTP programs

Unlike in normal programs, neither the VDI nor the AES are initialized. The program runs in text mode and full screen.

All the calls to display a number or a string a redirected to GEMDOS calls, as the calls to accept a value such as **input** or **input$**.

The **expect** instruction is a simple **input$** and doesn't allow the edition of an existing string.

To generate such a program, you have to specify a value of zero for the screen percent of the GEM window in the **>prgflags** directive.

After compilation, an extension **.TOS** is proposed and, if your program uses ARGC/ARGV then an extention of **.TTP** is proposed.

6 Accessories

As for a normal program, two VDI stations are open and fastopen/fastclose only handles the single window.

An accessory, by nature, remains present in memory waiting to be woken up. Therefore, if 3/4 of the memory is allocated to it, there will not be much left for other applications. A word managing the size of the dictionary is therefore essential:

reserve

n p reserve : sets dictionary size to p% of free memory plus n kbytes. For example:

```
10 0 reserve            \ 10 kB reserve
0 100 reserve           \ reserve all memory
-50 100 reserve         \ all memory minus 50KB
0 75 reserve            \ 3/4 of the memory
```

returns on the stack a GEMDOS error code (negative) or the allocated byte size (positive).

Side effect: reserve re-allocates a 1028 byte dynamic variable management table. See structures and the ***settable*** *statement. Therefore it is necessary to leave at least 1028 free bytes with reserve!*

In the absence of a reserve instruction, 3/4 of the memory is allocated as for a normal program. If we use reserve, this instruction must precede all those using the dictionary (declaration of variables, creation of words, assembly instructions etc ...), the easiest way is to put reserve on the first line of the source!

To determine the necessary size, all you have to do is launch the accessory without

reserve and, when you judge that the use it makes of the dictionary is at its maximum, make it display the value of full (memory occupied in the dictionary).

>accname

>accname string : force an early menu_register call.

Forth initilalizations take time and sometimes, the AES opens its desktop before the program has called menu_register. In this case, your accessory is not vivible to the user. It only appears in the menubar after an application has been run and the menu updated.

To correct this, use **>accname** that performs a menu_register call just after appl_init. But at this time, the memory blocks are not yet allocated, the variables are not created, so you still have to use menu_register to get your accessory identifier and store it for future use, this will be a simulated call that will fill intout as expected.

The string parameter is the same as with menu_register, be sure to use the same!

7 AUTO programs

These are the programs run from the AUTO folder before AES is installed. Therefore AES instructions are not available! During launch, no VDI station is open. Fastopen will do this (an internal physical station for text output plus a virtual station for the user). This allows you to change the resolution of the front screen and then open a VDI station properly with the new dimensions. Fastopen opens page0 on the whole screen and erases it. For the moment, the value returned by fastopen is always 1 in an AUTO program, and therefore useless.

Fastclose closes the two stations (beware, no more usable VDI instructions ! no more text outputs either (except with the GEMDOS words)). This makes it possible, for example, to start a program in low resolution then to switch to medium resolution, always by accessing the VDI properly with several successive fastopen and fastclose pairs. Fastclose is not necessary at the end of the program because the exit routine takes care of it.

Remember that the mouse (mousex, mousey, mousek, v_show_c, v_hide_c) is managed by the VDI, therefore accessible in an AUTO program as soon as fastopen is called.

8 STE or TT/Falcon programs

By using COMPSTE.PRG and FORTHSTE.LIB one will generate programs compatible with the STE (and perhaps with the STF). That is to say which avoid the use of codes reserved for the MC68030 and the MC6888 2 coprocessor. These programs also run on TT and Falcon. With COMP.PRG and FORTH.LIB the programs will only run on

TT/Falcon.

You will also need to create FORTHSTE.INF and COMPSTE.INF files to set the general parameters.

9 Additional Instructions

basepage

returns the address of the Basepage of the program, under the editor it is the basepage of FORTH.

vq_aes

returns 0 if AES is available and -1 if the program is launched from the AUTO folder. Under the interpreter we always get 0.

_app

returns 0 if the program is launched as an accessory (ACC) and a non-zero value if it is launched as an application (PRG, TOS, TTP...). Under the interpreter we always get a non-zero value.

10 GEMDOS header flags

In the header of a program is a long word at offset 22 that determines the behavior of certain program functions.

bit 0 :1 to erase only the BSS part of the RAM (fast load)
bit 1 :1 to load the program in TT Ram if available
bit 2 :1 to direct Malloc to TT Ram if available
bit 3 : not used

bits 4 & 5 : manage access to program memory space in a multitasking environment:

- 00 private memory (reserved for the application)
- 01 fully shared memory
- 10 shared memory only in supervisor mode
- 11 shared memory read but private write

>prgflags

l p n >prgflags : sets three values for the generated program.
n : the program flags
p : the percent of the screen for the window when you use fastopen
if you specify **zero**, then you'll get a **TOS/TTP program**.
l : the language code (0=US, 2=FR,...)

n and p are simply ignored under the interpreter.

By default:
p = 100
l = language code found in the system ROM
n = the flags set in FORTH(STE).LIB, typically, we have:
FORTH.LIB flags at 01x111
FORTHSTE.LIB flags at 00x001

11 English or French version

Depending on whether you are using the French or English compiler, certain strings (alert messages) will be adapted in the generated program.

Your FORTH.LIB or FORTHSTE.LIB library remains the same regardless of the version of the compiler but the generated program will have different texts.

Table of contents

2 General functions:

3 Text outputs

10 Block functions

11 Bézier curves

12 Metafiles

13 Other Instructions

AES functions

1 Presentation

2 General functions