Group: GENERAL - ΓΕΝΙΚΑ

- ALL [ΟΛΑ]
- ARITHMETIC FUNCTIONS [ΑΡΙΘΜΗΤΙΚΑ]
- BITMAP COMMANDS [ΕΝΤΟΛΕΣ ΕΙΚΟΝΩΝ]
- BROWSER COMMANDS [ΕΝΤΟΛΕΣ ΙΣΤΟΥ]
- COMMON DIALOGUES [ΚΟΙΝΕΣ ΦΟΡΜΕΣ]
- CONSOLE COMMANDS [ΕΝΤΟΛΕΣ ΚΟΝΣΟΛΑΣ]
- CONSTANTS [ΣΤΑΘΕΡΕΣ]
- DATABASES [ΒΑΣΕΙΣ ΔΕΔΟΜΕΝΩΝ]
- DEFINITIONS [ΟΡΙΣΜΟΙ]
- DOCUMENTS [ΕΓΓΡΑΦΑ]
- DRAWING 2D [ΓΡΑΦΙΚΑ 2Δ]
- FILE OPERATIONS [ΧΕΙΡΙΣΜΟΣ ΑΡΧΕΙΩΝ]
- FLOW CONTROL [ΡΟΗ ΠΡΟΓΡΑΜΜΑΤΟΣ]
- INTERPRETER [ΔΙΕΡΜΗΝΕΥΤΗΣ]
- MODULE COMMANDS [ΧΕΙΡΙΣΜΟΣ ΤΜΗΜΑΤΩΝ]
- MOUSE COMMANDS [ΕΝΤΟΛΕΣ ΔΕΙΚΤΗ]
- OPERATORS IN PRINT [ΧΕΙΡΙΣΤΕΣ ΤΗΣ ΤΥΠΩΣΕ]
- PRINTINGS [ΕΚΤΥΠΩΣΕΙΣ]
- SCREEN AND FILES [ΟΘΟΝΗ ΚΑΙ ΑΡΧΕΙΑ]
- SOUNDS AND MOVIES [ΗΧΟΙ ΚΑΙ ΤΑΙΝΙΕΣ]
- STACK COMMANDS [ΕΝΤΟΛΕΣ ΣΩΡΟΥ]
- STRING FUNCTIONS [ΑΛΦΑΡΙΘΜΗΤΙΚΑ]
- TARGET AND MENU [ΣΤΟΧΟΙ ΚΑΙ ΕΠΙΛΟΓΗ]
- VARS READ ONLY [ΜΕΤΑΒΛΗΤΕΣ ΣΥΣΤΗΜΑΤΟΣ]

Group: INTERPRETER - ΔΙΕΡΜΗΝΕΥΤΗΣ

identifier: ABOUT [ΠΕΡΙ]

1) Help form for application
About your_title$, text_for_help$
2)
About your_title$, width_in_twips, text_for_help$
3)
About your_title$, width_in_twips, height_in_twips, text_for_help$
4)
About
Close help form by code
5)
About ! your_title$, width_in_twips, height_in_twips, text_for_help$
wait ctrl+f1 (from console)
or use About Show from an event About from controls in a form (and form has this event too)

6)

```
About ! ""
clear help
```

7) Using About Call {} (run as a global function), we can click in words, or some words in [ ], and automatic call the About call function and with some about commands we can change page and title too. From forms and About event we can call this function with About Show "topic one" and about$ now return "topic one". Only one About call function may exist anywhere. A second one replace the old one. This function can't called like any other function. We can't call anything but global modules/functions

For console applications we have to prepare About Call {} with a About !..... so in an Input command we have early the about form prepared

For a form application we don't have to prepare about, we have events. So one way is to use About in each event, or if we want a "book" of help, we can make an About Call { } function and from each event we use About Show "topic". Also in Unload event we use About to close help form. Help form also closed when a messagebox apear.

About ! tile$, widthinTwips,HeightinTwios, Text$

use in brackets [ ] text for hypertext by using the about call { }

advance example

```
about ! "Title", 8000,6000,"that [is] easy"
about call {
select case about$
case "is"
about  "Title: IS", 8000,6000,"that [is] easy [too]"
else
about  "Title: too", 8000,6000,"that [is] easy"
end select
}
\* use ctrl+f1 to open About...
input "name:",a$
```

identifier: ASSERT   [ΑΞΙΩΣΗ]

```
ASSERT X=10, Z>40
```

EVALUATE ALL CONDITIONS, RAISE ERROR FOR ALL RESULT OF TYPE FALSE (OR ZERO), DISPLAY THE FAULT EXPRESSIONS.

ESCAPE OFF DISABLE ASSERTS

identifier: CLEAR   [ΚΑΘΑΡΟ]

CLEAR
clear all variables and arrays.

We can clear statics (so x in example next time get 10)
Write this in module a (Edit a and then Copy). Every time we execute A we get values from variables z, N and x.

```
module alfa {
    static z=10
    N=20
    module beta {
        static x=10
        x++
        Print x
        if x>=15 then clear  \\ clear local static/variables
    }
    beta
    Print z, N
    z++
}
alfa
```

Example for deconctruction in objects

```
class alfa {
        remove {
                print "ok"
        }
}
k->alfa()
\\ clear k engage remove if k is the last pointer to object
clear k
k->alfa()
\\ now module end execution
\\ so remove engage automatic
```

identifier: CLIPBOARD   [ΠΡΟΧΕΙΡΟ]

1. Clipboard A$
We copy A$ to clipboard. If a$ contain image then that image is copied. We can load from clipboard using Clipboard$ for text and Clipboard.image$ for image.
2. Clipboard A
If A is a number then converted to string and copy to clipboard. If A is an image then that image

copied to clipboard. We can get that image using Clipboard.Image for bitmap and Clipboard.Drawing for metafile (emf)
3. Clipboard A as "bmp"   ' export to a DIB bitmap in any case (bitmap or emf)
4. Clipboard A as "emf"   ' export as metafile if A is a metafile

```
move 0,0
clear A$
copy scale.x, scale.y to A$  \\ copy to A$
clipboard A$     \\ screen copy as DIB in clipboard.image
```

identifier: DOS   [ΚΟΝΣΟΛΑ]

When we use USER mode this command return an error.

1) dos [, sleep time after call cmd.exe]
   just open the cmd window

2) dos "command" [, sleep time after call cmd.exe]
    Execute the command and cmd window stay opened

3) dos "command" [, sleep time after call cmd.exe];
    no window open.

identifier: EDIT   [ΣΥΓΓΡΑΦΗ]

Open internal editor and create/modify/rename modules and functions

```
EDIT modulename
EDIT functionname(
EDIT functionname%(
EDIT functionname$(
EDIT something, position in text
EDIT this as that
```
(you can remove last module/function with REMOVE command)

We can edit UTF-8, UTF-16 (LE & BE) and ANSI. Internal all changed to unicode UTF-16 LE.
EDIT "app.gsb"
We can open a disk file to edit with "gsb" type (george small basic...was the first name of M2000)
EDIT "file.txt"

We can edit text . We discard the changes by using popup menu or Shift-12.

EDIT ! 4
set 4 char for indentation and tab character for edit
EDIT
Open the list of commands that recorded in command line screen. So line editor has no arrow movements bat with up and down we can find something we have wrote or we can open the list and we can write any new line, them we go back with esc and use up or down arrow (same job for both), and find the command you want.

For putting text in a variable at run time we can use INPUT, uses this text editor.
SEE INPUT

identifier: END   [ΤΕΛΟΣ]

End used to end program – and close environment. This can be used in level 0 (CLI). We can use SET END in a module or function. Using END without SET in a module or function is same as BREAK.
There are two variants for specific use: [END SELECT] and [END SUB]

identifier: FAST   [ΓΡΗΓΟΡΑ]

fast   this is the default handling of refresh rate (one that defined by REFRESH)
fast !   this is the extreme condition. No screen refresh until an input or a return to command line happen
we can use Refresh command when we want refresh of screen. Using "set fast !" we can draw to screen and we can refresh it at the end.

We can use REFRESH to do a screen refresh any time
write in a module:

```
Set Fast !
Refresh 100
Profiler   ' set the special timer
For i=1 To 1000 {
    Print i     ' the new line from print cause a refresh request. But extreme condition drop this.
    If i Mod 333=0  Then Refresh
}
Print Timecount
```

' set slow to do automatic refresh more often

identifier: FKEY   [ΚΛΕΙΔΙ]

function keys F1 =1 to F12=12 and shift+F1 =13 to shift+ F12=24

```
1) FKEY 1,"Help"
 FKEY 1 { commands }
   set a function key
2) FKEY 1
      return key definition
3) FKEY
      return list of Functions Keys (only those with a setting)
4) FKEY CLEAR
     erase all functions keys
5) FKEY 1 ,""
 FKEY 1 {}
 Clear FKEY
```

identifier: HELP   [ΒΟΗΘΕΙΑ]

1) Show Help
CTRL +F1

2) Help string expression
searching Help database using a string

3) Help identifier
as 2 but using identifier

4) Reading all topics from Help2000.mdb (old)

```
\\ Get all English topics from Help Database
Form 80,50
Flush
Document All$
nl$={
}
Dir appdir$
\\ for Close Base (is optional here)
base$=dir$+lcase$("HELP2000")+".mdb"
```

```
Retrieve "HELP2000", "SELECT * FROM COMMANDS ORDER BY ENGLISH"
Read n
For i=1 to N {
    Retrieve "HELP2000", "SELECT * FROM COMMANDS ORDER BY ENGLISH", i
    Drop
    read Gr$, Memo$, En$, group_id
    If group_id<>22 then {
        Retrieve "HELP2000","GROUP", 1,"GROUPNUM ", group_id
        Drop 2 : Read group_gr_en$
        En$=filter$(En$,"_")
        If Right$(En$,1)="(" then En$=En$+")"
        Report En$ +" ("+Rightpart$(group_gr_en$, ","+chr$(160))+")"
        All$=En$ +" ("+Rightpart$(group_gr_en$, ","+chr$(160))+")"+nl$
    }
}
Print
ClipBoard All$  ' to clipboard
Dir user
Save.Doc All$, "English.txt"
Win "WordPad", Quote$(Dir$+"English.txt")
Close Base base$  \\ optional
```

5) Read all topics from dat file

```
Document aLL$, doc$
Load.doc aLL$, appdir$+"help2000utf8.dat"
A=Val(paragraph$(aLL$, 1))
B=Val(paragraph$(aLL$, 2+A))
for i=1 to b
        m=Val(rightpart$(paragraph$(aLL$,2+A+i+3*B),"!"))+1
        b$=mid$(leftpart$(paragraph$(aLL$,2+A+i+3*B),"!"),2)
        iF right$(b$,1)="(" then b$+=")"
        c$=b$+" ("+ Rightpart$(paragraph$(aLL$, m), ","+chr$(160))+")"
        Print #-2, c$
        doc$=c$+{
        }
next i
Save.Doc doc$,"English.txt"
Win "WordPad", Quote$(Dir$+"English.txt")
```

identifier: KEYBOARD   [ΠΛΗΚΤΡΟΛΟΓΙΟ]

```
keyboard load  ' load/select US keyboard
keyboard load "00000409"  ' we can load any keyboard. This is the US keyboard too
keyboard load "00000410"  ' Italian
keyboard remove "00000410"
```

```
Keyboard !   'Open OSK (On Screen Keyboard)
Keyboard StrExpression$
```

Place characters to keyboard.

```
Keyboard 65,")"
Keyboard {? "This is like we press these keys"}, 13
Input A$
Print A$
```

Using ? Locale$(94) you get for current locale the Keyboards Numbers

Set the locale using:

Locale 1033 (also the same can be done with LATIN statement)

You can chech current locale using ? Locale

identifier: LIST   [ΛΙΣΤΑ]

1) LIST USERS

show list of users (look USER)

2) LIST

show all variables with values (big string are cropped)

We can send to a file

LIST filenumber

```
    open dir$+"thisname.txt" for output as f
    list f
    close f
```

Now we can read

```
    open dir$+"thisname.txt" for input as f
    while not eof(f) {
        line input #f, a$
        print a$
    }
```

    close f

3) List Com
   List Com to NameofModule
\\List declare type statements for com objects
you can use choose.object to open the list only (this statement maybe change in later versions)

identifier: LOAD   [ΦΟΡΤΩΣΕ]

from version 9 there is a cache for every loading from code, so next time we use the stored in memory, not the file. If we want the file, then we use Load New command. From console we load from disk any time.

Load Modules ... (for loading modules/functions without executing calls to modules, using cache)
Load New Modules ...(for loading modules/functions without executing calls to modules, refreshing cache)

Load name
------Load name.gsb from Dir$ current directory (m2000 user directory) and execute
Load name, "password"
------As previous but you save it before with user password

Load "name with spaces" && "this name with spaces"
------Load one and merge with second and then execute
Load "name with spaces", pass$ && "this name with spaces", pass$

Anything you load after termination of current level of run, erased. But this can be change if we load code in a module when we call it with special Call! command. Because anything local can be stay alive we have to ensure that the loading happen once. So in next call! we have all variables, modules, classes, functions as static. We can use For This {} to make a block where all erased at exit.

identifier: MONITOR   [ΕΛΕΓΧΟΣ]

Monitor   in console
or Set Monitor from a module

Display information about M2000 environment

identifier: NEW   [NEO]

```
New
```

Clear stored modules and functions

(Stack and Variables defined in CLI aren't clear. Use Flush for stack and Clear for variables)

identifier: PROTOTYPE   [ΠΡΩΤΟΤΥΠΟ]

```
prototype {
      Print "ok"
} as alfa$  ' or alfa (without $)
Print len(alfa$)
Inline alfa$
```

identifier: RECURSION.LIMIT   [ΟΡΙΟ.ΑΝΑΔΡΟΜΗΣ]

By default limit for subroutines is 10000 calls

```
Recursion.limit 1000
Recursion.limit 0   \\ this means no limit
```

see examples below

Subs have own stack, not system. Methods and Functions use the system stack.

For system stack:

check stack limit and known paths with Monitor command.

```
switches "-REC"
```

change limit to safe stack size 300

```
switches "+REC"
```

change limit to default stack size 3375

only limit change not actual stack size.

example 1:

```
\\ this is end to 9999
a(1)
```

```
sub a(x)
     print x
     x++
     if x<10000 then a(x)
end sub
```

example 2:

```
\\ this is not end to 9999 because  we use { } and this use stack space.
a(1)

sub a(x)
     print x
     x++
     if x<10000 then { a(x)}
end sub
```

identifier: REM   [ΣΗΜ]

```
REM : PRINT "THIS LINE NOT EXECUTED BUT HAS SYNTAX COLOR"

REM {
          PRINT "THIS BLOCK NOT EXECUTED AND HAS NO SYNTAX COLOR"
          \\ IT IS A MULTILINE REMARK
}
```

identifier: REMOVE   [ΔΙΑΓΡΑΦΗ]

REMOVE

We remove the last created module in memory
we can see the modules names only in memory with ? switch
MODULES ?
Free library (see Declare)
Remove "user32"

identifier: SAVE   [ΣΩΣΕ]

SAVE LOCAL means that all top modules/functions can be loaded as local
SAVE without LOCAL means that all top modules/functions loaded only as global

```
1) SAVE "name with spaces"
   SAVE name
```

```
2) SAVE name, modulename2run
3) SAVE "name" @, modulename2run    \\ this hide code
   SAVE "name" @
4) SAVE "name" @ @ "special code"
   need the special code with load command.
```

identifier: SCRIPT   [ΣΕΝΑΡΙΟ]

```
Script modulename
Script commands_in_string$
```

This command can run in CLI, or by using Set command in program interpreter

```
\\ global modules for temporary use
Module Global [001] {
     const n=5, z=50
     Function Hello$ {
          ="Hello There 1"
     }
}
Module Global [002] {
     \\ any definition here is global
     const n=55, z=550
     Function Hello$ {
          ="Hello There 2"
     }
}
\\ script can call only global modules
\\ and is a special call, which permanent use of modules definitions
Set Script [001]
Start()
Clear  \\ clears only variables, we need this for erasing constants too
Push "[002]"
\\ Script can execute Script again
Set Script "Script "+letter$
Start()


Sub Start()
     Print n, z, Hello$()
End Sub
```

identifier: SLOW   [ΑΡΓΑ]

CLI command only.
SLOW

Inside a module use SET SLOW
Engage the slow operation of interpreter. For any command the interpreter send a refresh to the environment form.

We can engage slow as choice in Control Form: use TEST to display variables and or execution code as run, and we see the stack also. You can stop, restart or engage like slow function (not the same)

identifier: SORT [ΤΑΞΙΝΟΜΗΣΗ]

SORT [descending] document$ [, start, end[, from_position]]
For Documents see 1 and 2, for Inventories see 3, 4, for arrays see 5 and 6

1)

```
\\ use key to see next lines. Report going to wait mode, when display lots of lines.
Document alfa$ ={one
    two
    three
    four
    five
    six
    seven
    eight
    nine
    ten
    }
Sort alfa$, 1, 10
Report alfa$
Print "***************"
Sort descending alfa$, 1, 10
Report alfa$
Print "***************"
Sort alfa$, 1, 5
Report alfa$
```

We can use Methods of Document to change the compare function
Methods are SetTextCompare (default), SetBinaryCompare, SetDatabaseCompare (I can't figure if it works)

and SetLocaleCompare with or without parameter. (without use stored locale id in document)
(we can store explicitly locale id using With a$, "Lcid", 1033 or using a variable for input/output:
With a$, "Lcid" as aName : aName=1033)

```
Document a$={péché
          sin
          peach
          pêche
          }

Method a$, "SetLocaleCompare", 1036
Sort ascending a$, 1,4
Report a$
Method a$, "SetLocaleCompare", 1033
Sort ascending a$, 1,4
Report a$
```

2) We can set the position from where we run compare
so we can put data for specific width and keys for any length

```
document a$={1234beta
     4567alfa
     1212zeta
     9921epsilon
     }
Sort descending a$, 1,4, 5
Report A$
```

3)

```
Sort ascending  inventory1 as number
Sort  inventory1 as number, 1
Sort descending  inventory1 as number
Sort  inventory1 as number, 0
Sort ascending  inventory1 as text
Sort  inventory1 as text 1
Sort descending  inventory1 as text
Sort  inventory1 as text, 0

Sort ascending  inventory1 as text, way
Sort descending  inventory1 as text, way
```

way can be
0 Binary Compare
1 Text Compare (default)

2 Database Compare

1033 or 0x800 or any other Locale Id

Inventories need uniquel keys (binary compared), except for Inventory queue.

Inventories can accept only keys (and later we can attach something to any key)

"Print a" normally show items of Inventory, but if a key has no item, then key presented

```
Inventory  a="sin","pêche","péché","peach"
sort a as text, 1
Print a, "standard ascending"
sort a as text, 0
Print a, "standard descending"
sort ascending a as text, 1036
Print a, 1036
sort ascending a as text, 1033
Print a, 1033
```

4) Using Columns in Keys in Inventories (using chr$(1) between columns of key)

```
inventory queue alfa
with alfa, "Stable", false ' so we use quicksort  and not insertion sort (default in inventory queue)
\\ 0 means ascending text
\\ 1 means descending text
\\ 2 means ascending numeric
\\ 3 means descending numeric
\\ keys have to place a chr$(1) to handle it with columns.
def d$(a$, n)=a$+chr$(1)+str$(n,"")
Append alfa, d$("a1",1):=1,"c1",d$("a1",2):=2
Append alfa ,"b1",d$("a1",3):=3,"z1",
d$("a1",4):=4,"c2",d$("a1",5):=5,"b5",d$("a1",100):=100,"z1", "sin","pêche","péché","peach"
Print "without sort"
Print alfa
Print "sort first part of key ascending as text (chr$(1) used to break parts), second part
ascending as number"
Sort alfa, 0,2
Print alfa
Print "same as previous but use 1036 for clid for text"
Sort ascending alfa, 1036, 0,2
Print alfa
Print "sort key ascending with automatic number recognition"
Sort ascending alfa as number
Print alfa
Print "sort key ascending as text, 1036"
Sort ascending alfa as text , 1036
Print alfa
print  "done"
```

identifier: START   [ΑΡΧΗ]

START   (Cold restart)
START "","" (WARM RESTART)
START "VERDANA" (before 6th version this was the only way to alter font in screen, now we can use FONT)
START "","BEEP:BEEP"

For altering screen size and font, use MODE, WINDOW, FORM, LAYER

identifier: SWITCHES   [ΔΙΑΚΟΠΤΕΣ]

at M2000 console
\\ changes stored to windows registry, and by using break, or start command, or when we start a new m2000 environment, switches are loaded with these new values.
switches "+DEC -DIV -PRI"
at a module or function
\\by using break, or start command, or when we start a new m2000 environment, switches are loaded with OLD values.
set switches "+DEC -DIV -PRI"
\\ we can apply the switches in the command line when we start a gsb file or m2000 editor
These switches used temporary for current run of interpreter

TXT
    -TXT compare strings as binary (> < >= >= <> =)   **by default**
    +TXT compare strings as text (> < >= >= <> =)
DEC
    -DEC use decimal point as dot
    +DEC use decimal point as local (for code is always dot)   ***by default*** if locale char is dot then a bypass change to -DEC (internal)
    If we use Locale command (Locale 1032) we change decimal point to that locale until next Switch Command, ot Locale, or a Break by key, or by command using Start at console (or Set Start from code)
DIV

-DIV like VB6 with integers. Not the same for Double (mod return fractions too) ***by default***

Print 10 mod 2.3*2 is Print 10 mod (2.3*2)

+DIV like Python

Print 10 mod 2.3*2 is Print (10 mod 2.3)*2

PRI

-PRI ***by default***

Print True Or False And False

Print (True Or False) And False

+PRI

And has priority over Or

Print True Or False And False

Print True Or (False And False)

REC

-REC

300 limit for recursion for functions (if we strart m2000.exe from any other but the m2000.exe, we have to use Set Switches "-REC", because execution stack maybe has small size)

+REC

3260 limit for recursion for functions ***by default** (if we start m2000.dll from m2000.exe)

Change placed after break (using Start as software break or using button break), and for all next loadings of M2000.dll

FOR

\\ only for current running interpreter

-FOR

using For loop as normal in M2000, always execution of block, using from step the absolute value, if starting and ending number are different, or using normal value if starting and ending number are equal (to compute the final value for control variable)

+FOR

using normal sign step, and if sign of (ending point- starting point) isn't equal to step sign then we get no execution of for loop.

DIM

\\ only for current running interpreter

-DIM

normal a Dim A(4) has 4 items (0 to 3 or 1 to 4 depends of Base, by default is 0 to 3)

+DIM

if base 0 then we get for dimension items>0 one more, so a Dim A(4) has items from 0 to 4, so Len(A())=5

also these arrays expands from last dimension (right one, but normal is from left one, or using Dim Ole make it from right one)

SEC

\\ for all interpreters

- SEC

Old Version 8. This make unsecure conditions like in the example below. If b.alfa exist then b.alfa get a new value, but we want a new local variable. We can use Local alfa=100. We get more speed for small names.

+SEC

By default this is the best mode for SECure intrepreting.

```
set switches "-SEC"
Module b {
     alfa=50 : Module c { Module b {alfa=100} : b } : c : Print alfa
}
b   \\ print 100
set switches "+SEC"
Module b {
     alfa=50 : Module c { Module b {alfa=100} : b } : c : Print alfa
}
b   \\ print 50
```

TAB

+T wAB by default, using tab as character 9.

-TAB using TAB as spaces.

when we use character 9 after letters we get various spaces depending the column width (as the tab width)

when we use -TAB we can't insert character 9. If we change the switch when we have lines with character 9, the characters preserved, but they don't preserved when we copy to clipboard.

We can set the width of tab using Edit ! number

SBL

+SBL show boolean type True/False (depend for the choosen language Greek or Latin)

-SBL show boolean type as -1/0

Type of literals True and False is double with values -1 and 0, but a comparison return always a boolean

we can produce the boolean using a comparison:

```
A=True: Print A=True
```

we get True if we use +SBL or -1 if we use -SBL

INP

+INP in FIELD variable which return the cause of exit in FIELD when Enter pressed return 13

-INP in FIELD variable which return the cause of exit in FIELD when Enter pressed return 1 (default)

MDB

+MDB help from Help2000.mdb

-MDB help from help2000utf8.dat (default)

NBS

-NBS change TAB characters to NBS when we put code from editor to clipboard

+NBS default value

Special switches (for command line)

Test

+TEST και -TEST

start an auto run program in test mode (debug mode)

Norun

-NORUN load program without execution of any command except those which create modules and functions.

Switches which set the registry values, and used when we run a new interpreter or we press Break or execute the Start command.

Linespace

+LINESPACE 150 set linespace to 10 pixels when we use 96 dpi screen (1440/96=15 twips per pixel). Values from 0 to 600

-LINESPACE set to 0

Size

+SIZE 12 set fontsize to 12pt. Values from 8 to 48 integer only

-SIZE set fontsize to 15pt

Font

-FONT choose Monospac821Greek BT font if exist

Bold

+BOLD set BOLD for font

-BOLD set no BOLD for font

Pen

+PEN 5 set pen (text color) 5 from basic colors 0 to 15. If paper is 5 then set 15-5 as pen

-PEN set pen to 0 and paper to 7

Paper

+PAPER 0 set 0 for screen color. If pen is 0 then set 15-0 for pen

-PAPER set pen to 0 and paper to 7

Dark

+DARK choose color set for dark background when we copy or cut code, on the html part of export (editor export text and html to clipboard)

+DARK choose color set for bright background

Casesensitive

+CASESENSITIVE for filenames on search/open procedure (always preserve case at save procedure)

-CASESENSITIVE for filenames, case is not sensitive
Ths switch make the change immediate.

Greek

+Greek set dialog messages to Greek language
-Greek set dialog messages to English language

we can use Greek or Latin to change this local. We have Latin for second language, because maybe we can change in future the english language with another language, so the Latin is for letters only.

A second example of faulty old 8.0 interpreter
If we use a Module Z {} and we Call k, then we get only 10, and Call Z inside Z not work.
If we use +SEC then this example run ok.

```
set switches "-SEC"
Module Z {Print "Error" }
Module k {
     global b =10
     Module Z {
          Print b
          b--
          if b>0 then call Z
     }
     call Z
}
Z
call k
set switches "+SEC"
```

Use command Monitor to check switches
When we use M2000 interpreter as an object, in another application, then maybe we get a lower stack, so we can use Set Switches "-REC" to set limit to 300
Setting of stack can be done to M2000.exe, not to M2000.dll

```
\\example
Form 80, 50
Dim Base 0, b$(2)
b$(0)="-DEC","+DEC"
' need global because Set run at global space
global aa=each(b$())
while aa {
     for skip=false to true {
          Print "Switches "+quote$(array$(aa))
```

```
        set switches array$(aa)
        def fom1$()="{0:1:"+str$(-tab)+"}{1:2:"+str$(-tab)+"}"
        if skip else locale 1032 : Print "locale ";locale
        print format$(fom1$(), -1.66, 2.45)
        print -1.66, 2.45
        if skip else locale 1033 : Print "locale ";locale
        print format$(fom1$(), -1.66, 2.45)
        print -1.66, 2.45
        print $(3), str$(.66), str$(-.45), $(0)," never changed"  ' no 0 and always . for decimal point, space infront for positive number
        print $(3), str$(.66,0), str$(-.45, 0), $(0)," never changed"   ' $(3) means right justification on this column and after
        print $(3), str$(.66,1033), str$(-.45, 1033), $(0)," never changed"   ' using locale direct
        print $(3), str$( .66,"") , str$(-.45,"")
        ' $(0) is the default justification, strings may use other columns and have left justification
        ' numbers have right justification
        print $(0),.66, -.45
    }
}
```

identifier: TEST   [ΔΟΚΙΜΗ]

1) Test
    Enter test mode (opens the Control Form)
2) Test ModuleName
    Call module in test mode (opens the Control Form
3) Test !
    Close control form by code
4) Test "name of breakpoint"
5) As in 4 but with an added list of expressions for printing in form for each step of execution

```
\\ A new style for breakpoints
\\ just press enter to stop to next breakpoint
Test "start", test("ok"), A
For A=1 to 100 {
    Wait 1
    Print A
    if A=41 then test "ok" : test test("ok2")
    if A=81 then test "ok2": test
}
Test !
```

identifier: VERSION   [ΕΚΔΟΣΗ]

Version
as command print version number (there is a read only variable to check version in code)

identifier: WIN   [ΣΥΣΤΗΜΑ]

```
WIN "CALC"
win "netflix:\"   ' open app netflix (which previous downladed from Microsoft Store)
WIN DIR$
WIN "WORDPAD"
WIN "WORDPAD","COMMAND LINE"
```

To run another gsb file (M2000 script) use the USE command (look also to PIPE command)

identifier: WRITER   [ΣΥΓΓΡΑΦΕΑΣ]

Display writer name

Group: MODULE COMMANDS - ΧΕΙΡΙΣΜΟΣ ΤΜΗΜΑΤΩΝ

identifier: ERROR   [ΛΑΘΟΣ]

```
Error 100
Error "Problem"
Error "Problem 1000"
Error 0  \\ shutdown environment
```

FLUSH ERROR ' use it before Error statement to prevent to merge current message with next one.

```
\\ Example 1
Try ok {
     a=1/0
}
Print ok   \\ false we have error in the block level
Print Error

\\ Exanple 2
Module bb {
     a=1/0
}
Try ok {
    bb
}
\\use flush error
Print ok    \\ true we don't have error in the block level - is inside bb
Print Error    \\ true
Print Error$  \\ using error$ a "flush error" command executed
Print Error$  \\ so now is an empty string
```

identifier: ESCAPE   [ΔΙΑΦΥΓΗ]

```
Escape on
Escape off
```

Esc key is used to stop execution (perform a clean up also). We have Break key to stop execution and erase anything in environment. Esc key by default in manual mode is on. Using FAST ! we gain speed but we loose escape functionality (we can get it by using refresh command, and for that time only)

EXAMPLE

```
Set fast !
For i=1 to 1000 {
    print i
    \\ remove \\ from refresh to stop it
    \\ refresh
}
```

identifier: FUNCTION   [ΣΥΝΑΡΤΗΣΗ]

I) Normal Functions

Functions are like modules, they have a name with a hidden prefix, from module's or function's name which is the parent of it.

There are functions in groups, functions without name, and functions in a variable as lambda (and these functions can be included in array items, and inventory lists).

A function after execution erase anything that defined inside (like modules). We can call the same function again, as an argument to function (see DEF description later). Functions have recursion (about 3300 calls only).

We can make A(), A%() and A$() as three functions which return double, integer (is an Int(double) as a return), String

Also functions can return arrays, groups, lambdas, inventories, events, or an array of all of these.
See Lambda for more help

To define a function we can use one line definition with DEF command, or Function name { } like modules.

1) One Line definition using DEF

```
Def Alfa(X)=X**2
Print Alfa(10), Alfa(Alfa(Alfa(2)))
```

2) Using anonymous call (a function in a string use "{   }" to place the code (and multiline too), and for return a = as command is used.

```
Print Function("{=number**2}", 10)
```

3) Using anonumous call but give the same name as the module which lazy$() function prepare the function

```
M=5
Print Function(lazy$(number**2+M), 10)
```

4) As 3 but for multiline functions. We can pass back an array of values.

```
Let K=100, N=1000
M=2   \\ M is local too
Function fake {
     Read New M
     \\ in normal functions we can't read K because is out of scope
     Local N=500  \\ without Local we change modules N
     =M**2+K, 2*M, 3*K+N
     K++
}
Dim A()
A()=Function(lazy$(&fake()),10)
Print A(0), A(1), A(2)
Print M  \\ 2 but if we delete New in Read statement we get 10
Print K
```

5) Lambda is a special construction (is an object inside). We can define variables to hold a state. Using lambda in groups we can move data and functionality from/to groups.

```
A=lambda X=3 -> {=X : X++}
Print A(), A()
B=A       \\ not only we have a copy of function but a copy of internal state variables
Print A(), A(), B()
C=lambda->number**2
Print C(2)
Dim A(10)
A(0)=A, B, C  \\ assign values from 0
Print A(0)(),A(1)(),A(2)(3)   \\ 7 6 9
```

```
Inventory Alfa=1000:=C, "square":=C, "counter":=B
Print Alfa(1000)(5), Alfa("square")(3), Alfa("counter")()  \\ 25 9 6
```

6) Using Function name {}

No parameter signature needed. We can Read parameters like Module. But while Modules call Modules passing own stack, Functions always have own stack (except if we call them by Call statement, as modules)

```
Function Name1 {
\\ multi number return as array
    Print Module$  \\ print the real name of Function/module
    =1111,2222,3333
}
Dim z(), k()
z()=name1()
Print z(2)
Function Name2 {
\\ multi object return to array
    group A {X=1,Y=3}
    B=A
    A.X++
    A.Y--
    =A,B
}
\\ array defined with Dim before: Dim K()
K()=Name2()
Print K(0).X, K(1).X  \\ 2  1

\\ Example 1:
\\ optional parameters
Function A {
    Y=100
    Read ? X, Y
    =X*Y
}
Print  A(), A(3), A(2,4)  \\ 0, 300, 8

\\ use default value
Function BB {
    Y=10
    Read X, Y
    =X*Y
}
```

```
Print BB(3,?) \\ 30

Push &A()   \\ push as anonymous  Function
For This {
      \\ use it for temporary definitıons
      Read &AB() \\ Read Function and give a name
      Print AB(2,4)  \\ 8
}

\\Example 2:

Group A {
Private:
      k=10
Public:
      counter
      Module SetK {Read .k}
      Module IncK {.k++}
      Function  GetK {=.k : .counter++}
}
Print A.GetK()  \\ 10   (counter=1)
Push &A.GetK()  \\ this is a reference to A.GetK() function
Read &AK()  \\ this is a new function which is the same as A.GetK() and has a special reference
to group A
A.SetK 55
Print AK()  \\55 (counter=2)
Push &AK()
A.IncK
Module DeepA {
      \\ this is the defition only
      Read &AK()   \\ here the special reference to group A is tied with function.
      Print AK()  \\56 (counter=3)
}
DeepA &A.GetK()  \\ this is the call to DeepA
Print A.counter  \\3


\\Example 3:
\\ use Function like subroutine (but without a subroutine call)
\\ if you wish to read parameters then use Read NEW (not Read only)
KK=1234
Function Name3 {
      Print KK
```

```
    Print module$  \\ B (module$ isn'r Name3(), change to module, so KK is known)
}
Call Local Name3()

\\ Example 4:
Function Name4 {
    Read A()
    Print A(0).X   \\ 2
    Print A(1).X  \\ 1
}
\\ K() prepared in example 1
Call Name4(K())
Call Function Name4, K()

Function FuncAsModule {
    Read A,B
    Push A, B
}
\\ STACK NOT ERASED AFTER CALL
Call FuncAsModule(10,1)
Print Number, Number


\\QuickSort Example:
Function partition  {
      Read New &A(), p, r
      x = A(r)
      i = p-1
      for j=p to r-1 {
         if A(j) <= x then {
               i = i+1
             Swap A(i),A(j)
           }
       }
       Swap A(i+1),A(r)
      = i+1
    }
Function quicksort {
   Read New &A(), p, r
   if p < r Then {
     q = partition(&A(), p, r)
     call local quicksort(&A(), p, q - 1)
     call local quicksort(&A(), q + 1, r)
   }
}

Dim  arr(10), old()
arr(0)=23,21,1,4,2,3,102,54,26,9
old()=arr()
Call local quicksort(&arr(), 0, 9)
For i=0 to 9
     Print arr(i), old(i)
next i

\\ Example check current base
Function Base {
     Dim a(1)
     =valid(a(0))
}
Base 1
Print Base()
```

II) Simple Functions

Functions like subs, staticaly written to a module or a function.

A simple Function has these differences

1) Use of the caller stack of values
2) Use of the caller name space
3) We have to use Local to define local variables (same for Subs)
4) We can't place them by reference
5) We can't use them as members of Groups
6) We can't use CALL to call simple functions like subs

About them:

1) They have less resources, so we can call 10% more simple functions from normal functions in expression
2) Don't use the same simple function name as a sub name, in same scope.
3) Code of a simple function maybe not in the code of a caller, but in the code where the caller's code taken the first time.

```
\\ This is a module name, execution of Module statement create the module in the
module's/function's list.
Module ModuleName {
          \\ This is a normal function (a definition in one line)
          \\ the same as Function Hello$(x) {=String$("H", x)}
          Def Hello$(x)=String$("H", x)
```

```
    \\ Hello$() exist in modules/functions list
    \\ @Hello$() are static in the code names.
    \\ first time the posiiton searched from the end,
    \\ second time the position is known.
    \\ if an array exist with same name as the Hello$() then
    \\ the @Hello$() call the simple function
    \\ and we need to call Hello$(*10)  with asterisk to use the norma function above.
    Print @Hello$(10), Hello$(10)

    InternalUse()    ' Gosub is optional unlees an Array exist with same name
    Gosub InternalUse()

    \\ this is a simple function
    Function Hello$(x)
        =String$("Hello", x)
    End Function
    \\ this is a subroutine
    Sub InternalUse()
        Print @Hello$(5)
    End Sub
}
ModuleName
\\ Function Hello$() exist in the same original code
Print @Hello$(3)
```

About Names:
(Only for names in Modules/Functions List and Variables/Arrays list)
When two things have the same name
Array preffered from Functions from interpreter
Any local of same kind preffered from a global one.

For subs and simple functions: we have to use different names for subs and simple functions.

So the problem may happen if we have a Global array, and then we make a Global function with same name
To indicate the function from the array we have to use Alfa(*) where asterisk indicate that it is a function.
To create libraries, forget Global functions, or you have to use asterisk to be sure about any conflict.

```
\\ A global array exist before module defined
\\ Problems may exist if we define a global function after global array with same name
\\  If we use a Function Global Alfa(x) {=x/100} in place of Global Alfa(1 to 10)=4
\\ In all Inner definitions the internal Alfa(x) always executed.
Global Alfa(1 to 10)=4
Print Alfa(1)=4
Module Inner {
        Function Global Alfa(x) {
                =x**2
        }
        Print Alfa(10)=4
        Print @Alfa(10)=100
}
Inner
Module Inner {
        Function Global Alfa(x) {
                =x**2
        }
        Print Alfa(10)=4, Alfa(*10)=100, @Alfa(10)=30
        Function Alfa(x)
                =x*3
        End Function
}
Inner
Module Inner {
        Function Alfa(x) {
                =x**2
        }
        Print Alfa(10)=100, @Alfa(10)=100
}
Inner
Module Inner {
        Function Alfa(x) {
                =x**2
        }
        Print Alfa(10)=100, @Alfa(10)=30
        Function Alfa(x)
                =x*3
        End Function
}
Inner
```

identifier: HALT   [ΑΛΤ]

Halt send execution to console. We can use Continue to continue execution.

identifier: INLINE   [ΕΝΘΕΣΗ]

INLINE string_expression$
We place some commands in line in code.
With this and module$ and module command I make an example of using pseudo objects...(like objects).

Example 1

```
a$=quote$(1,"aaaaaa",2,3)
inline "data "+a$       'put the line data 1, "aaaaa", 2, 3
```

Example 2

```
X=1
Module Alfa {
      \\ this is not for calling as a module
      \\ because X is not visible
      X++
}
\ Inline Code just paste code from Alfa and if found  "," then past after Inlice code again.
Inline code Alfa, Alfa
Print X
```

We can know if a user module exist using Module() function (also tell as if a user function exist), without call them

```
Print Module(Alfa)
```

Example 3

```
\\ This was the fist idea for objects before Group objects exist  in M2000.

form 60,40
Module ONELEVELDOWN {
' my class has a global var and construction prototype
' I use @@ for replacing with current object at construction stage
' This was my first idea before making the Group object, and the Class command
' I use @Print to direct the interpreter to use it as original Print because
' optimizations are setting to on by default and if we make a new Print
' then always for this context interpreter treat this as user function
```

```
'
' See how we use module$ (the name of module) and how I change module name when module
run

global mine.global = 1000, last.class$=""
global general.proto$ ={
    ' this is a big multiline string using for inline code
    ' BUT BEFORE WE DO THE INLINE ACTION WE HAVE TO REPLACE @@
    ' WITH THE OBJECT NAME  - which is the module name
    push module$
    module @@
    read parent$  ' now we have our parent module (or object??)
    nextclass$=last.class$
    last.class$<="@@"     \\ because last.class$ is global
    a=1  ' this are properties
    b=2
    n$="no man"
    ' this is an array (I found problem with WINE here)
    dim a(10)
    function personal {
        ' this is a member
        ' every object has own personal function
        ' c is local, mine.global is global, @@b is object properties
        c=12
        mine.global++
        =(@@.a+@@.b)*mine.global+c
    }
    module parent$
}
Module me.set {
    Read modulename$ : Module modulename$ ' prepare to insert in object namespace
    Read A, b, n$     ' we can read because we are in right namespace
    Module parent$ ' restore old namespace (using a not global var, but we can make it as
global if we like to do....)
}
Module me.print {
    Read modulename$ : Module modulename$
    @Print A, b, n$
    Module parent$ 'restore old namespace
}
' this is first level constructor using classname$ and general.proto$
classOne$={inline replace$("@@",classname$, general.proto$ )}
Push "one", "two","three","four"
```

```
For i=1 To 4 {
    Read classname$
    Inline classOne$   'call construstor
}
' now we have all objects
' lets play with one
' this is a function for any object of our type classOne$
me.set "one",1,2,"a string field for one"
me.set "two", 4,5,"a string field for two"
me.set "three",7,8,"a string field for three"
me.set "four", 100, 120, "a string field for four"
' and this is a function for any object again
Flush
me.print "one"
@Print "now all together"
Data "one", "two","three","four"
For i=1 To 4 {me.print }

' all properties are read write....
' this is an array inside object
' we can place . in left hand part of assignment if it is a var only, not array or function
' else we have to place @ where . is
one.a(4)=1000
@Print "one.a(4)=";one.a(4)
' you can use functions inside objects
For i=1 To 3 {
@Print "one.personal()="; one.personal()   ' as you see this function see properties, local var and a global var for all objects
}
me.set "two", 1000,2000,"George"
@Print "three.n$=";three.n$
@Print "three.personal()=";three.personal()
four.a(6)=4
' and now we access objects as a linked list using nextclass$
showlist$ = {
    ' pushing first
    repeat {
        module letter$  ' object namespace get it from stack
        ' we can make a variable here..new for the object..
        ' later we can see if is there with VALID
        if not valid(check) then check=true
        if check then {
            @Print "Object:";module$ ,, ' this is the name
            @Print ">>>>>>>>>>>>>>>>";n$,,
        }
        push nextclass$ ' this is the next class
    } until nextclass$=""
    read empty$ ' we throw it from stack
    @Print "module name:";module$, " or ";Module.name$
}
Push "one"
Inline showlist$  ' we use inline because we dont want to erase new value...check...
@Print "For showlist we prepare a new property named check"
@Print "For three we change proprty check to false"
\\ this was an old way to pass a global or a static variable - interpreter make three.check not
modulename.three.check
three@check=False
Push "one"
Inline showlist$

' repeat until restore name of module automatic
@Print "module name:";Module$, " or ";Module.name$

''' list  ' list the variables
' we can't delete an object, but all objects they are destroyed after this module end
' right now!
}
ONELEVELDOWN
```

identifier: LINK   [ΕΝΩΣΕ]

```
\\ We can link variables to other variables
\\ We can't link twice, for same variable in list2
\\ We can link more than once a variable in list1, but each time to new name
Link variable_list1 to variable_list2
```

```
\\ we can link to same name shadowing then old one.
Link variable_list1 local to variable_list2
\\ we can link arrays too
Link array1() to array2()
Link array1$() to array2()
Link array1() to array2$()
\\ we can link a pointer to array with an array
\\ in a new module write:
A=(1,2,3,4)
Link A to Array1()
A+=10
Print A
Print Array1()
Array1(3)+=10
Print Array1(3), Array(A, 3)
Return A, 0:=5,1:=3,2:=100,3:="500"
Print A
Print Array1()
A++
Print A   \\ 4th element stay 500 because is string
Link A to Array1$()
Print Array1$()
Array1$(3)="hello"
Array1$(1)="200"
A+=500
Print A
Print Array1$()
```

```
a$="one"
Link a$ to b$
Print b$
```

this is the same as

```
a$="one"
Push &a$  \\ push a weak reference as a string in stack
Read &b$  \\ read weak reference and make the link
Print b$
```

```
\\ Example using Lazy$(), weak$() and Link
a$="{=1000*number+number}"
```

```
\\ a$ is a weak reference to anonymous function in a string (has a block {} inside)
\\ actually a weak reference to function is a copy of code and sometimes there is
\\ a weak reference to object (if it is group member)
Link weak a$ to aa()
Print aa(10,-100)
m=300
b$=weak$(m)
\\ b$ is a weak reference to m
Link weak b$ to d
Print d
d+=5000
Print m, m=d, Eval(b$)
\\ using a dot in string variable, interpreter expect a weak reference
b$.+=500
\\ we can use dot in Eval() but is optional
\\ in Eval$() is not optional because without dot return the string not the reference string
Print m, m=d, Eval(b$.)

Group Beta {
    counter=0
    Function Check (x, y) {
        .counter++
        =x**2+y
    }
}
Module CheckThis (&f()) {
    Print f(11,3)
}
Link Beta.Check() to Z()
Print Z(10,2), Beta.counter
CheckThis &Beta.Check()
CheckThis Lazy$(Beta.Check(number, 30))
CheckThis Lazy$(Beta.Check(20, 30))
Print Z(12,4), Beta.counter
```

Link parent

\\ is a variant of Link, used for linking a parent group variable with an local name. Used in Groups in Groups. Only one level up allowed, and only for variables (we can' link a function, except lambda functions, and only if lambda has no references to group variables)

```
Group alfa {
    K=10
    m=lambda N ->{
        =100+N*.K
```

```
          N++
     }
     Group beta {
            Value {
               link parent m to m
               For this   {
                    =m()
               }
          }
     }
}
For i=1 to 10
     Print alfa.m()
     Try {
          m=alfa.beta
          Print m
     }
Next i
alfa.m=lambda N=1000 -> {
          =100+N  \\ with no reference to .k
          N++
}
For i=1 to 10
     Print alfa.m()
     Try {
          m=alfa.beta
          Print m
     }
Next i
```

identifier: MODULE   [TMHMA]

A module is a basic programming block of commands that can be called by name. Like module we have Functions. Functions are modules that return a value to an expression. So function we can say (but not restricted to that) called from expressions and modules from line of code or form command line (from console).

Modules have a name with on letter at least. We can put dots.

To create a module in command line we have to write EDIT nameofmodule
We can exit from editor with ESC, or Shift+F12 if editor get mad (sometimes using ctrl+z/c/v need to get back the original code, by canceling changes)
We can create modules in modules.
Modules are saved with SAVE command and loaded with LOAD command. Each time we have to give a name. That name is a file name, not a module name. We can use ctrl+A to save again in same name as the last save. This keystroke has to be deployed in M2000 command line.
We run a module just using the name of it

IMPORTANT: A module rule say that everything that created in a module, erased at the end of execution of module.

So if we load modules from disk in a module, then those modules erased at the exit from that module.
Every module can create modules, functions, threads and can store values to variables, arrays and in the special stack.

Module Decoration

We can call a module decorated, which means to pass a new definition for an inner module inside module.

```
\\ Decorated modules
Group a {
    x=100
    module finish (a) {
        .x+=a
        print "My good finish", .x
    }
}
module beta {
    print "beta"
}
module alpha (b){
    module finish (x){
        print "standard finish", x
    }
    module delta {
        print "internal delta"
    }
    print "ok", b
    delta
    finish 50
}
print a.x
```

```
alpha 100 ; delta as beta, finish as a.finish
print a.x
alpha 200
```

Modules ?

A module use the parent stack always. A function start with own stack. A thread is a part of a module that read same values but has own stack. Stack can hold numbers and strings and for bit of time references to arrays and variables. Variables and Arrays erased as first law says, but not the stack. The top parent is the command line interpreter CLI, as it run in the main thread. Other parents are all other threads and the functions. A recursive function create a new stack in every level of recursion.

We can use recursive call in a module using the CALL statement

A module name is as a namespace for named creations (threads use numbers in variables, so they have no names but handlers). We can change this name for the time of execution only. But when we do that all creations from the old name cannot founded except we use the old name as a prefix.

```
Module a {
    b=10    ' we create the a.b numeric variable
    module c    ' here we change the name space
    b=20
    print b, a.b  ' we use prefix a as qulifier for the old b value
    b++    ' add one to b
    a.b++  ' add one to a.b
    print b, a.b
    list  ' we see the list of variables
    module a
    list ' we can see the list of variables is the same
}
```

Now list print A.B=11, A.C=21

Another example using weak reference in a string. Inside module C we call A$.B so we know that a B module is calling, but we can change the prefix (but with that prefix a module B must be loaded in the list of modules before). Functions can be called as modules to and can be passed by reference without needing to be loaded, they load in the Read statement.

```
Module B {
    Read A, B
    Print A*2, B**2
}
Module C {
    Read A$
    A$.B 20
}
C Module$, 20
```

Here is a more advanced example
We use two threads to call the same module so we want to use per thread namespace.
We pass values to modules writing to Stack. We can write it before we call a module or we can set a frame of values after the name. Those values are written on top of stack but the first become the top of stack.
In b "AA", k we write on top the string "AA" and next the value of k. Stack holds values only (we can place references to variables as values too, see the next example)
We can get the values from stack by using READ but that mean that the variables have a prefix at that time. So We can read the stack by using Letter$ for string and number for number (there is function to ask for what we have, because if we use letter$ and no string is on top of stack then an error happen)
In b module we only make a variable C and then we print the list of variables (from all modules)

```
module a {
    module b {
        module letter$
        read c
        list
    }
    k=1
    thread {
        k++
        b "AA", k
    } as aa  interval 10
    o=100
    thread {
        o++
        b "BB",  o
    } as bb interval 10
    main.task 20 {
        if inkey$<>"" then exit
        print o, k
        refresh  ' need that to refresh the screen....
    }
}
```

Example of By Reference Passing to module
We can place it in a module or in a module in a module

```
a=10
module b {
    read &a
    a++
}
b &a
print a

' module in a module
module k {
    a=10
    module b {
        read &a
        a++
    }
    b &a
    print a
}
k
```

identifier: MODULES   [TMHMATA]

```
MODULES
    print all the modules in memory and in current directory
MODULES !
    print all the modules in memory and then sort by name and print the modules in current directory
MODULES ?
    print all the modules in memory
MODULES ? "alfa"
    print all the modules in memory with name start from alfa
MODULES ? "*alfa"
    print all the modules in memory with name having alfa
MODULES ? "","DIM "
    print all the modules in memory who have DIM in the code
MODULES ? "*alfa","DIM "
    print all the modules in memory with name having alfa and have DIM in the code
```

```
\\ For searching files we use FILES
FILES "GSB", "PRINT|DIM"  ' return all files *.gsb with the two words PRINT and DIM
```

You can use any folder to save and load a module. One folder is the USER and by default this is the first folder when we open m2000 without using an execution after we provide a gsb file. When we have call M2000 loading a module from disk, by clicking a gsb, file then the current folder is the folder of that file. So if we have some modules as libraries (we can load when a program run) we have to change to USER directory (it is the directory that open for a user, so in the same computer but another user we have other USER folder)

Use this to change to the user directory
DIR USER
Use this to open the user directory with explorer
WIN DIR$
So you can move or delete modules by using explorer.

identifier: PIPE   [ΑΥΛΟΣ]

Advance topic.
We use a base.gsb to run a trg.gsb
we can call FEED in trg.gsb to feed values to base.gsb
we can call TERM in trg.gsb to end the loop in base.gsb

```
\* WRITE THIS IN TRG.GSB
MODULE C {
    WINDOW 12, 7000,4000
    FORM 32, 25
    FORM
    SHOW
    MOTION.W 8000,2000
    PRINT "THIS IS A TARGET"
    STACK
    READ A$
    PUSH A$, A$
    MODULES
}
MODULE FEED {
    READ PIPE$
    PUSH PIPE$
    O=0
    REPEAT {
        O=O+1
        TRY KK {
            PIPE PIPE$, 1,2,3
            WAIT 10
        }
    PRINT KK, O
    } UNTIL KK OR (O=100)
}
MODULE TERM {
    READ PIPE$
    PUSH PIPE$
    O=0
    REPEAT {
        O=O+1
        TRY KK {
            PIPE PIPE$, 0
            WAIT 10
        }
        PRINT KK, O
    } UNTIL KK OR (O=100)
}
C
\**************************** WRITE THIS IN BASE.GSB

MODULE B {WINDOW 12, 6000,4000
FORM 32, 25
FORM
SHOW
MOTION.W 500,2000
PRINT "THIS IS THE BASE"
K$="ANAMEFORPIPE"
USE TRG.GSB 1,"MESSAGETXT",3 TO K$ AS L
THREAD L INTERVAL 200
C=0
PAR$=""
NM$=""
GETOUT=FALSE
EVERY 200 {
                C=C+1
                PRINT C
                IF INKEY$=" " THEN EXIT
                IF C=1000 THEN EXIT
                IF K$<>"" THEN {
                            STACK K$
                            STACK  ' SHOW ONLY
                            READ NM$, PAR$
                            FLUSH
```

```
              IF PAR$="N" THEN GETOUT =TRUE
              C=0
        }
        IF GETOUT THEN EXIT
      }
PRINT "FINISH"
THREAD L ERASE
}
B
```

now do this
Load BASE
so Base is loaded and call B automatic, and then you see two consoles running, and in second console you can send messages to first one. Use FEED and TERM to end listening in first console

identifier: STOP   [ΔΙΑΚΟΠΗ]

Use stop to temporary pause the program flow (threads can be run in the background). The program open a prompt with user name at console and we can use the same in Test form, when we change the prompt of bottom textbox to > we have to use Delete

STOP
We see that in console
user_name>_
We can give Exit to return to execution
user_name>Exit
We can use any statement, including loops, alter variables, make new variables, all of them at the point of stop.
There is another command, HALT similar to STOP. Halt always return to Console input, and direct commands to command line interpreter, at global name space.

identifier: SUB   [ΡΟΥΤΙΝΑ]

```
SUB ALFA(X)
\\ commands
END SUB
```

Subs are written at the end of modules and functions. We can pass by reference (use &) or by value anything like modules, but subs are not modules, they stay at module's (or function's) namespace and have the same ability to erase any new item that we make on their space (except threads). So a Sub see anything in a module, except we declare it as Local, also parameters in parameter list are local too. We can exit with Exit Sub. We can use a module

variable as static but we must define before we gosub to sub. We call subs with Gosub (also Gosub call simple routine by name or by number without parameters). We have recursion (we use Recursion.Limit 1000 times to limit it, but we can change it). Interpreter search from bottom to find a sub, and record it for second call. When in execution interpreter find Sub then this is like an exit
So in a module we can write:

```
y=100
k=30    \\ use it as static
Gosub Alfa(3)
Alfa(13)  \\ Gosub is optional for subs except there is an array with same name
Alfa(5)
Print y

Sub Alfa(x)
     Local y=10
     If X>10 Then Exit Sub
     Print x*y, k
End Sub
```

Calling a sub without Gosub, may cause problem if an array exist with same name. Interpreter first check for arrays. A global array with same name may broke the code. So it is better for safety to use Gosub in call.

identifier: THREAD   [NHMA]

A thread is a part of a module that can be run in a time sharing plan.
Each thread can have static variables, and use private stack, for calling modules and subs.

Refresh command make a refresh to current form output (some versions before Refresh allowed threads to run, but this changed)
Wait 1 can be used inside thread in a loop if we want to block that thread and want to run other threads. If we use input command or Key$ to wait for keypress, these blocks the thread but Interpreter can run other threads (using wait inside).

1) creation

```
THREAD {
     Static A=5   \\ this executed once
.............
} AS K \\ here we can put a command like Interval or Execute
```

K has the thread number. That number is the handler of the thread. Threads have no names. They are part of the module where they created. Every thread has own stack for values but uses the same variables as the module use. So variables and arrays never erased from threads..

2) Handle

```
Thread K Interval 200
Thread K Hold
Thread K Restart
Thread K Erase
Thread K Execute A=10
```

in a thread we can use THIS as the handler (we can't read but command Thread knows what is). When a thread run, are deleted from a waiting list, a request with the handler miss it, so only THIS can tell to system that we request the current thread. A module or a function has no thread handler, so a request to THIS produce an error.

3) Report the TASK manager list of tasks. Internal task as MAIN.TASK and AFTER not showing the hidden handlers.

```
Threads
```

```
Module A {
    ' Simple Counter with keyboard interface
    If Version<6 Then Exit    ' no threads for older versions
    Hide
    FORM 50
    Cls 1
    Pen  14
    Print "THREADS IN M2000"
    Print "Internal Thread Example - press space for menu"
    Pen 15
    Cls 4,2
    Show
    c=1
    THREAD {
        c=c+1
        Print c
              } As B
    THREADS     ' for information
    THREAD b interval  80
    List
    ex=False
    Main.Task 100 {
        k$=Ucase$(Inkey$)
        If k$<>"" Then Print k$
```

```
                Select Case k$
                Case "A"
                      THREAD b Hold
                Case "B"
                      THREAD b restart
                Case "C"
                      THREAD b erase
                Case "Q"
                      ex=True
                Case "I"
                      THREADS
                Case >=" "
                {
                      Print "A - HOLD"
                      Print "B - RESTART"
                      Print "C - ERASE"
                      Print "Q - QUIT"
                      Print "I - INFORMATION"
                      Print "ANY OTHER RETURN HERE"
                }
                End Select
                'An exit from  module  erase all threads created from it
                If ex Then Exit
          }
}
```

Second Example Using EVERY and WAIT and AFTER
See the thread handlers b and c
AFTER create a thread but hide the handler.
The b thread has heavy duty
We can change EVERY with TASK.MAIN
We can see that line   print "MOUSE:",mouse prints more often in MAIN.TASK
This is happen because MAIN.TASK is a thread and all threads running in controlable time slices, one by one.
Here we see a thread C that run as the thread B run for a long time. This can be done because WAIT give time to run another task.
When we use MAIN.TASK after the end all threads for that module are erased. EVERY didn't do that. So we see how we can erase threads. When a module end...all threads ends.
Because a thread is "Time Loop" we can avoid to use loops in threads. Here is the bad example. To do better we have only to put in c the I++ and an IF statement to reset the I when became greater from a limit. We have better control and not threads inside threads (it is better to use the thread by thread story, and because the have each one a time slice, the order of

execution maybe changed, and that is the multitasking. Changing intervals we give more time to run as we step down one or step up the time intervals of others)

```
' write in a module
form 60,32
cls 1
pen 15
after 500 {
     print "One shot thread from AFTER"
}
i=0
thread {
     print i
} as c
thread c interval 30
thread {
     ' very big thread
     Print "Big Thread Start"
     for i=1 to 10 {
          wait 100      ' wait give time to threads to run
     }
     Print "Big Thread End"
} as b interval 200
every 200 {
     if mouse>0 then exit
     print "MOUSE:",mouse
}
threads
thread c erase
threads
thread b erase
print "end"
threads
```

Same example using Task.Main (or Main.Task is the same command)

```
form 60,32
cls 1
pen 15
after 500 {
     print "One shot thread from AFTER"
}
i=0
thread {
     print i
} as c
thread c interval 30
thread {
     ' very big thread
     Print "Big Thread Start"
     for i=1 to 10 {
         wait 100      ' wait give time to threads to run
     }
     Print "Big Thread End"
} as b interval 200
Task.Main 200 {
     if mouse>0 then exit
     print "MOUSE:",mouse
}
threads
thread c erase
threads
thread b erase
print "end"
threads
```

Examples using Thread.Plan

```
\\ Concurrent
Thread.Plan Concurrent
\\ A=100  we can't use local variable with same name as a Static in thread variable
\\ if we do that we get an error.
M=500
N=0
StartThreads(20)
\\ using a thread for this task.
\\ we can exit by using mouse click or counting threads give zero
Task.Main 50 {
    Threads
    Refresh
    if mouse or N=0 then exit
}
Threads

End
Sub Display()
```

```
    Print ">>", N
End Sub
Sub StartThreads(Many)
    For K=1 to Many {
        N++
        Thread {
            Print A*M, B  \\ static A is used before any local A
            A--
            Display()
            \\ block needed after then if concurrent plan used
            If A=0 then { N-- : thread this erase }
        } as T execute static A=10, B=T
        \\ Now we start thread
        Thread T interval random(10,100)
    }
End Sub


\\ Sequential
Thread.Plan Sequential
\\ A=100  we can't use local variable with same name as a Static in thread variable
\\ if we do that we get an error.
M=500
N=0
Gosub StartThreads
\\ using a thread for this task.
\\ we can exit by using mouse click or counting threads give zero
Task.Main 50 {
    Threads
    Refresh
    if mouse or N=0 then exit
}
Threads
End
StartThreads:
For K=1 to 20 {
    N++
    Thread {
        Print A*M, B  \\ static A is used before any local A
        A--
        \\ changing thread plan we have a problem because
        \\ thread this erase is executed without control from If
        \\ so we need to use block { N-- : thread this erase }
        \\ blocks are executed before task manager change thread
        If A=0 then N-- : thread this erase
    } as T execute static A=10, B=T
    \\ Now we start thread
    Thread T interval 100
}
Return
```

identifier: THREADS   [NHMATA]

1) Threads
    Show all threads

2) Threads Erase
    Erase all threads (by disposing from thread pool)

identifier: USE   [ΧΡΗΣΗ]

1) Use "Hello.Gsb" 1, "alfa", 2

2) Use "Hello" 1, "alfa", 2

3) a$="HELLO" : Use a$ 1,"alfa", 3

4) USE TRG.GSB 1,"hello again",3 TO K$ AS L
  THREAD L INTERVAL 200
  ' execute trg.gsb send 3 items and prepare L thread to listen a pipe and put the result to string K$ as an stack variable ...see Envelope$()

5) USE PIPE TO K$ AS L : THREAD L INTEVAL 200
  ' make a pipe for input to K$ using a thread with handler L

example 1

```
MODULE HELLO {
    Form 40
    Show
    Print "Example 1, Hello"
    If Instr(parameters$, "-OK")>0 Then {
        Print "You call me from Shell",,"    with parameter -OK"
    } else {
        L$=Envelope$()
        If Len(L$)>0 Then {
            Print "You call me with USE and ";len(L$);" parameters"
            stack
        } else {
            Print "You just call me"
        }
    }
    Print "So Hello Again"
    Fkey 1, "end"
    Fkey 2, "h1"
    Fkey 3, "h2"
    Print "Choose a Function Key"
    Fkey
}
MODULE H1 {WIN DIR$+"HELLO.GSB -OK"
    }
MODULE H2 {use hello.gsb 1, "aaaa", 3
    wait 2000
}
HELLO
```

example 2

```
' this is the BASE.GSB and below is the TRG.GSB
' Base open Trg and Trg can send data to Base and can stop it.
' Base open TRG and give a pipe to it. Anyone can write to that pipe if knows the name.
' So we can make one base and we can execute many TRG. So the BASE can handle input from every TRG
' We can define in each TRG a new pipe for input and we can send the pipe name to Base
' So the Base can sen back to a specific TRG.
' So with that use we can have one SERVER and many CLIENTS. We can use pipes with computer names in front. So we can comunicate. But we can start a Base in one computer and run a Trg  in any other.
' What we can do is to make a standard "listening" pipe, in a known computer, and in all Trg programs we can setup that name for access to base. Then Base can known from a message from a Trg about how to transfer to the Trg data, just with the name of the pipe from computer who runs the Trg.

MODULE B {WINDOW 12, 8000,6000
FORM 32, 25
FORM
SHOW
MOTION.W 500,2000
PRINT "THIS IS THE BASE"
USE TRG.GSB 1,"hello again",3 TO K$ AS L
THREAD L INTERVAL 200
C=0
PAR$=""
NM$=""
GETOUT=FALSE
EVERY 200 {
                C=C+1
                PRINT C
                IF INKEY$=" " THEN EXIT
                IF C=1000 THEN EXIT
                IF K$<>"" THEN {
                            STACK K$
                            STACK  ' SHOW ONLY
                            READ NM$, PAR$
                            FLUSH
                            IF PAR$="N" THEN GETOUT =TRUE
                            C=0
                }
                IF GETOUT THEN EXIT
            }
PRINT "FINISH"
}
```

look to PIPE to find the TRG.GSB

```
MODULE SERVER {\\ Server - Client in one module
\\ USER DEFINED FUNCTION IN ONE LINE
FUNCTION CONSUME$ { READ &A$ : =A$ : A$="" }
\\ A new special thread prepare here with id 30123
\\ If pipe can be used then ALFA$ has the full pipe name.
USE PIPE TO ALFA$ AS 30123
 IF ALFA$="" THEN PRINT "Ooops" : EXIT
\\ Here is the client part...
\\ we have to clear ALFA$ (ALFA$ is in server side)
\\  Pipe name can be calculated by this function PIPENAME$("S30123")
PIPE CONSUME$(&ALFA$),"George",2,3
WHILE ALFA$="" {
    IF MOUSE THEN BREAK   \\EXIT FROM MODULE
    WAIT 10
    PRINT "Wait..."
}
\\ WE GET pipename$ (one that we clear), a match string "SNN" of what items follow.
\\ and finally the items
\\ So we can dump these to stack
STACK ALFA$    : DROP 2  \\ drop pipe name and match string
READ A$, A, B
```

```
PRINT A$, A, B
THREAD 30123 ERASE
THREADS



}
MODULE B {FUNCTION CONSUME$ { READ &A$ : =A$ : A$="" }
K$="13232"
M$=CONSUME$(&K$)
PRINT LEN(K$), M$
}
MODULE SERVER1 {\\ Server - Client in one module
\\ Setup of a pipe thread to link to a buffer$
\\ Syntax USE PIPE TO BUFFER$
\\ We can erase pipe thread by erasing all threads THREAD ERASE
\\ Or by using ID(A$) user function to find thread id from pipe name.
\\ First Buffer$ has the full pipe name. We have to erase to start fill it
\\ So we use a user function with more than one command Comsume$()

\\ USER DEFINED FUNCTION IN ONE LINE
FUNCTION CONSUME$ { READ &A$ : =A$ : A$="" }

\\ As old basic (FN is an optional prefix in function name )
DEF ID(A$)=VAL(REPLACE$(PIPENAME$("M"),"", A$))

USE PIPE TO ALFA$
IF ALFA$="" THEN PRINT "Ooops" : EXIT
L=ID(ALFA$)
PIPE CONSUME$(&ALFA$),"George",2,3
WHILE ALFA$="" {
    IF MOUSE THEN BREAK   \\EXIT FROM MODULE
    WAIT 10
    PRINT "Wait..."
}
\\ WE GET pipename$ (one that we clear), a match string "SNN" of what items follow.
\\ and finally the items
STACK ALFA$    : DROP 2  \\ drop pipe name and match string
READ A$, A, B
PRINT A$, A, B
THREAD L ERASE
THREADS
```

```
}
MODULE SERVER2 {\\ Server - Client in one module
\\ The easy way. ALFA$ is cleared
\\ L=12345   \\ here we use an auto thread number.
USE PIPE TO ALFA$ AS L
PIPE PIPENAME$(L),"George",2,3
WHILE ALFA$="" {
     IF MOUSE THEN BREAK   \\EXIT FROM MODULE
     WAIT 10
     PRINT "Wait..."
}
STACK ALFA$    : DROP 2  \\ drop pipe name and match string
READ A$, A, B
PRINT A$, A, B
THREAD L ERASE
THREADS


}
```

Group: FLOW CONTROL - ΡΟΗ ΠΡΟΓΡΑΜΜΑΤΟΣ

identifier: AFTER   [ΜΕΤΑ]

```
kk=10
After 2000 {kk=20 : keyboard chr$(13)}
Input "A$=", A$
If  kk>10 Then Print "Big time to response"
Print  A$
```

If you don't response in 2 seconds then the thread send a return and close the input

This is a thread that start after 2000 ms for one time only.

Write this in a module:

```
MyVar=100
After 1000 {
     Print "ΟΚ", MyVar
}
Module InsideThis {
      MyVar=500  ' this is local
      After 500 {Print "I am in InsideThis"}
      Wait 2000
      Print "I am in InsideThis...too"
}
InsideThis
```

Explanation:

Create Myvar, Establish thread to run once after 1 second, Create Module, Call Module

Inside Module: Create Myvar, Establish thread to run once after half second, Wait 2 seconds

First thread executed and Print "I am insideThis"

Second thread, from parent module, executed and Print ok 100

End of waiting

Print "I am in insideThis..too"

All threads started at WAIT (or in a Every or in a Main.Task) see below a typical example

```
\\ look this too
MyVar=100
After 1000 {
      Print "ок", MyVar
      After 2000 {
            Print "ок2", MyVar
      }
}
Module InsideThis {
      MyVar=500  ' this is local
      After 500 {Print "I am in InsideThis", MyVar}
      Wait 1000
      Print "I am in InsideThis...too"
}
InsideThis
Wait 2000
```

\\ Example

Write in a Module

```
KK=10
after 100 {KK=20}
for i=1 to 100 {
      print i
      wait 0
}
print KK
```

If you shadow wait 0 (make it a remark) then After never run, so KK is 10. Because inside "for" there is nothing to give time to thread system. Wait 0 is like Doevents in VB6

This is an example that we use After and Main.Task, these are threads.
You can have as many After as you like, and you can set a thread inside a thread.
Main.Task can run in different modules.

You can change Main.Task with Every to see what happen. Every isn't a thread, but give time to threads to run.

example 1

```
after 1000 {
    print "ok"
}
wait  2000
print "end"
```

example 2

```
--------------------------
Cls 0,0  ' Black like the night sky, set split screen to 0 row, so not used here
Pen 14 ' Yellow like a star
Flag=False
' Internal module
Global kk
Module Stars {
    Cls
    For i=1 To 200 {
        x=Random(0,Width-1)
        y=Random(0,Height-2)
        Print Part @(x,y);"*"
    }
    kk++
}
' Now we start profiler timer
Profiler
Stars
If False Then {}   ' do nothing, but profiler add the If duration
d=timecount+10 ' plus 10 to be polite to the system
After 10*D {Flag=True}
Main.Task D {
    Stars
    If Flag Then Exit
}
```

```
Cls  7
Pen 1
Print kk
```

identifier: BREAK   [ΔΙΕΚΟΨΕ]

1) Break for SELECT CASE

```
a=5
b=10
select case  A
case 1, 5 to 20
    { print "ok"
     if b=10 then  break
    }
case 2,4
    {print "and ok"
        continue
    }
case 3
    { print "that"
        print "and that"
    }
else
    print "??"
end select
```

2) Break for exit from a module (see 1)

```
module InsideThis {
    print  $(,8)  ' 8 chars Tab
    for i=1 to 20 {
        for k=1 to 10 {
            Print "this",i,k
            if i=2 And k=2 Then Break
        }
    }
    Print "This isn't printed"
}
call InsideThis   ' "call" is optional here
```

```
Print " This is printed"
```

If you want to print "this isn't printed" then you can work with error creating and resuming Using Try {   break }, but if using variable in Try then break are not stopped. You can use Error as a break:

```
module InsideThis {
    print  $(,8)  ' 8 chars Tab
    try er {
         for i=1 to 20 {
            for k=1 to 10 {
                Print "this",i,k
                if i=2 And k=2 Then error 10
            }
        }
    }
    Print "This isn't printed"
    a=error    \\ error =0 now - normaly we read error after a If not er then { print error}
    flush error  \\another way to flush error without reading
    ? error, a   \\ now error=0 and a=10
}
call InsideThis   ' "call" is optional here
Print " This is printed"
```

So the Error 10 generate a software error with number 10
Variable Er created and have 0 if an error occur inside TRY er   {   }

If you place Exit where you have Break then the exit are break only one block.
Error X break until a try catch it or terminate the program if no try exist.
Break terminate until module level.

```
m=0  \\ m never get value 100
try {
for i=1 to 100 {
    for j=1 to 10000 {
        if i=1 and j=10 then break
    }
    m=100
}
}
? i , j, m
```

identifier: CALL   [ΚΑΛΕΣΕ]

1) CALL [VOID] [FUNCTION] NAME [parameter list]

```
Clear \\ erase all variables
Flush \\ Flush Stack for values
Function Our {
    Read a,b,c,d
    Print b,c,d
    If a=1 Then {
        =0
    } Else {
        =1
    }
}
\\ using Void to drop return value (if any)
Call Void Function Our(0,2,3,4)
Call Void Function Our 0,2,3,4
Call Void Function Our, 0,2,3,4
Try Ok {
    Call Our(0,2,3,4)
}
If Not Ok Then Print Error$
a$="Our("
Call a$, 1,2,3,4
Call Our(, 1,2,3,4
Push 4,3,2,1  ' 1 goes to top of stack
Call Our(
Global x=10
Module Alfa {
    Print "Ok", x
    \\ here we use recursion, by using Call modulename
    If x>1 Then x-- : Call Alfa
}
Alfa
```

2) CALL LOCAL NAME (or String)

```
x=10
Function alfa {
    x++
    Print x
}
Call Local alfa()  \\ 11
Call Local "alfa()" \\ 12
Function beta {
    local x=1000
    Call Local alfa()
}
Call Local beta()  \\ 1001
Call Local alfa() \\13
Print x  \\ 13
Function delta {
    Read New &x
    Call Local alfa()
}
m=1233
Call Local delta(&m) \\ 1234
Print m \\ 1234
```

3) When module name is same as a built in command

3.1) When Module is Local

```
Module Print {
    while not empty {
        read a
        Print $("\t\r\u\e;\t\r\u\e;\f\a\l\s\e"), a,$("")
        if width-pos<tab then @print
    }
}
Call Print -1,0,3
.Print -1, 0, 3
Print
Print -1, 0, 3
```

3.2) When Module is Global

```
Module Global Print {
    while not empty {
        read a
        @Print $("\t\r\u\e;\t\r\u\e;\f\a\l\s\e"), a,$("")
        if width-pos<tab then @print
    }
}
Print -1,0,3
@Print -1, 0, 3
```

4) Call Event   raise an event, sending parameters to all functions inside Event Object. Look about Event in Definitions.

5) Call Operator "+"   (we can pass a parameter, a group, or leave it before in stack).
 This Call can be used only in modules, functions, operators in a Group.

identifier: CASE   [ME]

look SELECT

identifier: CONTINUE   [ΣΥΝΕΧΙΣΕ]

1) In a SELECT CASE continue the Break state.

```
a=5
b=10
select case  A
case 1, 5 to 20
     { print "ok"
           if b=10 then  break
     }
case 2,4
     {print "an ok"
          continue
     }
case 3
     { print "that"
          print "and that"
     }
else
     print "??"
end select
```

2) In a loop continue exit block but not the loop.

```
for i = 1 to 1000 {
     if i> 10 then exit
     print i 'print until 10
}
print i  ' 11

for i = 1 to 1000 {
     if i> 10 then continue
     print i  ' print until 10
}
```

```
print i  ' 1001

for i=1 to 1000
     if i>10 then next i : goto 1000
     print i
next i
1000 print  i
```

identifier: DO   [ΕΠΑΝΕΛΑΒΕ]

SEE REPEAT

identifier: ELSE   [ΑΛΛΙΩΣ]

Look IF and SELECT

identifier: ELSE.IF   [ΑΛΛΙΩΣ.ΑΝ]

see If

identifier: EVERY   [ΚΑΘΕ]

```
Every 100 {

}
```

If block of code need more time then it takes. The block isn't run on a thread, and using a "wait" mechanism that threads allowed to run. For better results (about time synchronization) use Main.Task because is a thread, and as all threads is running in background. If threads are running in background then maybe Every can't run commands in the own block of commands.

You can nest Every but the inside Every must have If ... Then Exit if we want to get out.

copy this example in a module

```
Refresh 5000
desktop 100
d=100
window 12,0
cls, height-10
gradient 2,7
i=0
```

```
legend$={this is Verdana Font 20pt

}  ' this is a multiline assigment
myangle = pi
bb=0
thread {
    Refresh 5000
    d+=5
    if d<256 then desktop d
    if i<2000 then i+=25
    gradient 2,7
    print @(0,0)
    double : bold : report 2,"M2000 example using Thread" : bold : normal
    move i,i*2
    draw to 6000,6000+i
    draw to 8000+i,4000+i
    move mouse.x, mouse.y
    draw angle myangle, 2000
    circle 300
    draw -2000, 2000
    step -400,-400
    legend legend$+ str$( now,"hh:mm:ss"),"VERDANA", 20, pi/4+myangle
    cls
    print mouse.x, mouse.y, @(tab(3)),time$(now), str$( now,"hh:mm:ss"), myangle
    list
    refresh
      if keypress(38) then {
        myangle += pi/8
      } else.if keypress(40) then {
        myangle -= pi/8
      }
} as main interval 200
aa=0
every 100 {
aa++
if mouse then exit
}
desktop 255
refresh 50
```

identifier: EXIT   [ΕΞΟΔΟΣ]

Exit
used for exit from a block

Exit Sub used for an exit from a Sub
Exit For used for an exit in a For Next loop
Exit For label for an exit and jump to label

```
\\ example
For i=1 to 10
     if i=5 then exit for 10
     print i
Next i
5 print "not here"
10 print "Hello"
```

identifier: FOR   [ΓΙΑ]

We can use integer (using %) or real or long (declare long in previus statement) Variable.
At the beginning variable searched in the local list and if not found then a new variable defined

We can use nested for

```
FOR I=1 TO 10 STEP 2 {
     PRINT I
}

FOR I%=1 TO 10 STEP 2 {
     PRINT I% ' integer
}
```

If start is bigger than end then step used as negative. But if start and end is equal then sign matters, so if we have a negative step then after For value be end plus step (so if end is 5 and step is -5 then variable get 0)

```
For i=100 to 1 {
    print i  ' from 100 to 1
}
For I=100 to 10 step 10 {
    print i  ' print 100, 90,80...10
}
```

Use CONTINUE and EXIT in a FOR

```
For I=1 ro 100 step 2 {
    IF I >50 Then Exit
    If I>30 and I< 45 Then Continue
    Print I
}
Print I

\\ without using For
i=10
a=Random(1,3)
{
    If i<30 Then Loop  \\ change an internal flag only
    If a>1 And i>25 Then Exit
    Print i
    i++
}
\\ Without using for or loop in a block
i=10
a=Random(1,3)
090 Flag=False
100 If i<30 Then Flag=True
110 If a>1 And i>25 Then 150
120 Print i
130 i++
140 If Flag Then 90
150 Print "Ok"
```

here k is global (but erased after module or function ends running)

```
set k=50
for k=k to 10 {   'here we have a new k that read the only known k, the global k
    print k           'here interpreter choose local first
}
```

```
set print k     ' set command send all chars until end of line to CLI the level 0 or global for
executions manual commands
                    so at that level only the global variables are listed (without special name
reference)
```

If we pass a name from a global variable (without all path) then a new variable defined so when we read the global variable that was unchanged but the internal *for* control value changed as we wish.

Warning
If we set start and end value the same then step go to 0 so value of counter not changed, except we use Step, so change happen one Step.

If you change i inside For then nothing happen for For because actual variable are private to For.
Don' t use For inside threads, because a thread is running in a loop at intervals, so we can adjust the time to make it as fast or slow we want, changing a value with ++ for i (i++)
so

```
i=0
Thread {
     i++
     print i
} as  aForThread
thread aForThread interval 100
main.task 80 {
     if mouse<>0  then exit
}
```

Check the refresh rate!

```
Refresh 100  ' By default refresh is 25
Profiler
For I=1 To 1000 {
     Print I
}
Print Timecount
```

Some additions:
1. We can use global variable (and class variable in a class module), using <= instead of =

```
global i
module beta {
     for i<=1 to 1000 { }    \\ operator <= say Use global
}
beta
print i
```

2. Negative/positive step for equal start and end point leave proper value

```
for i=1 to 1 step -1 {
}
print i  \\ 0

for i=1 to 1 step 1 {
}
print i    \\  2

for i=1 to 1 {
}
print i   \\ nornal print 1
```

```
\\ USING NORMAL BASIC FOR
\\ WE CAN APPLY SWITCH IN COMMAND LINE, OR IN CODE
SET SWITCHES "+FOR"
FOR I=10 TO 1
     PRINT I  \\ CAN'T EXECUTE LINES
NEXT I
FOR I=1 TO 10 STEP -1
     PRINT I   \\ CAN'T EXECUTE LINES
NEXT I
\\ FAST FOR BLOCK
FOR I=10 TO 1 {
     PRINT I
}
FOR I=1 TO 10 STEP -1 {
     PRINT I
}

\\ NORMAL FOR M2000
\\ USE AUTO DIRECTION -1 HERE
\\ ALWAYS EXECUTE FOR
SET SWITCHES "-FOR"
FOR I=10 TO 1
     PRINT I
NEXT I
FOR I=10 TO 1
     PRINT I
NEXT   'without use of variable i

\\ FAST FOR BLOCK
FOR I=10 TO 1 {
     PRINT I
}
```

identifier: GOSUB   [ΔΙΑΜΕΣΟΥ]

1) Gosub label or number

Simple call of routines

We run a batch of commands until a Return founded, and we continue to next line or command

2) Gosub SubRoutine1()   \\just call subroutine

2.1) SubRoutine1() \\ without Gosub if no array with same name exist

A subroutine is more advanced from simple routine for two reasons: Subroutine at the end of execution destroy any new variable, array, module or function that defined when run, and all variables, arrays, modules and functions from calling module or function can be used as static (exist in any call, subroutines also can be call itself - recursion)

3) Gosub SubRoutine1(X,Y,&A(),B)   \\passing parameters byvalue and by reference

3.1) SubRoutine1(X,Y,&A(),B) \\ without Gosub if no array with same name exist

We can use Local to make variables local to subroutine, but not static (local variables also destroyed on return - exit sub)

```
\\\ Indirect call
Gosub alpha(100,200)  ' this is the nornal call
b$="(100, 200)"
Gosub "alpha"+b$
b$="alpha(100,200)"
Gosub b$
Dim b$(10)
b$(3)="alpha(100,200)"
Gosub b$(3)

Sub alpha(x, y)
          Print x, y
```

```
End Sub

\\ Simple Gosub
X=10.5333
Gosub alfa : Gosub alfa
Gosub alfa
Exit

alfa:
    Print Format$("{0}={1:2}    Int({0})={1:0}","Alfa", X)
    X++
    Return

\\ Write this in another module
Let A$="000"
Def TrimNum$(A)=Trim$(Str$(A))
Def TrimNum2$(A)=format$("{0}",A)
Gosub A_name1(10.05, A$)
Print A$, Len(A$)
Gosub A_name2(10.05, &A$)
Print A$
Sub A_name1(X, B$)
    Print TrimNum2$(X)+B$
Exit Sub
Sub A_name2(X, &B$)
    B$=TrimNum2$(X)+B$
    Print B$
Exit Sub

\\ Example using Local and Exit Sub
k=100
m=50
Alfa(30) \\  capital or not is the same for M2000.
alfa(60) \\ nothing x>50
alfa(10)
Print m, k
End
Sub Alfa(x)
    Local k=Random(1,10)
    If x>50 Then Exit Sub
    Print x*k
    m++
End Sub
```

identifier: GOTO   [ΠΡΟΣ]

```
Goto label
Goto Number  (5 digits most, no Zero in leftmost position)
```

Labels must end with symbol ":"
After labels you can insert commands
you can't jump out of block code

```
I = 0
Repeat {
again:
    i++
        A$=Inkey$
        If A$="" Then Goto again
        If A$=" " Then exit
        Print i
}  Until i=1000
```

You can use here Continue instead of Goto label. But if you want a different entry then Goto is here to do right this.

Look On var GOTO label1, label2...

Jump allowed backward and forward in a block of code in curly brackets { }

```
Module a {
i=1
again:
i++
if i=10 then exit
break
goto again
}
call a
```

Using a block inside you can forget the goto, but goto is used to have an entry other than the start of a block or to skip some code. If you put code in curly brackets then an Exit is same as a jump to end of block.

```
Module a {
i=1
{ i++
      if i=10 then exit
      print i
      loop
      }
}
call a
```

You can write numbers like that:

```
100 print "ok"
200 if mouse>0 then exit
       ' this line has no number...numbers in right most position is only labels, not needed for commands
300 goto 100
```

if a goto is for the first line of block then you can swap with the Restart command (no need to search for label, just restart the block).

in a module or a block of code:

```
print "ok"
if mouse=0 then restart
print "End"
```

identifier: IF   [AN]

in one line:

```
If condition Then  command(s)  | Else.If condition2 |Then|Else| |  command(s)Else command(s)
If condition Else command(s)

If condition Then NumericLabel   \\ same as Then Goto NumericLable
If condition Else NumericLabel   \\ same as Then Goto NumericLable
If condition Then command1 : command2 ...
If condition Else Command1 : Command2 ...
If condition Then { commands in a line separated with : }
If condition Then  command(s)  Else command

If condition Then { command(s) } Else command
If condition Then {
    a block of commands
  }
If condition Then {
     a block of commands
} Else command

If condition Then {
     a block of commands
} Else {
     a block of commands
}

If condition Then {
     a block of commands
} Else.If condition then {
     a block of commands
} Else.If condition then {
     a block of commands ....
...................many Else.If....
} Else {
     a block of commands
}
```

New from version 8.9 (rev 12), we can split the structure with blocks or line of commands (in the same line of command)

```
If condition Then  a line of commands
Else.If condition then   a line of commands
Else.If condition then {
    a block of commands ....
....................many Else.If....
}
Else  a line of commands
```

Also we can use Else.If Else for reverse logic

Using End if (no for jump out of structure, because we get error for not computed End If)

```
If condition Then

Else.If  condition1 Then

Else.If  condition1 Else  ' reverse logic

Else.If  condition1 Then

Else

End  if
```

or

```
If condition Else  ' no Else.If after Else
End if
```

any of three conditions that not fits skip the line

```
if false else.if false else.if false else print "ok"
if true then if true then if true then print "ok"
```

About condition:
You can use AND OR XOR NOT, >,<,>=,<=,<>, == (this is for rounding before the operation)

For strings ~ is the VB6 Like operator

```
a$="a"
print 2<4 and a$="a" xor false   ' no need brackets
```

first compute the 2<4 then the a$="a" then the first AND so that condition shrink to this: result xor false
Not is like a macro and put "-1-" plus "look for expression", so this is right Print not A$="a" because a$="a" is logical expression, and give a number 0 or -1, so -1- -1 = 0 and -1- 0 = -1.

{a} is a string, but we can't use it in an expression for condition.
This is valid we have a string only expression:

```
Print  {1st line
2nd line
3rd} + {1st line too
....
.............
............}
```

But this isn't because there is > inside, so this is a condition expression. We have to use variables or non multiline strings

```
Print {1st line
2nd line } > {1st line
..................
}
```

This is valid

```
Print "Aaa"<"B"
```

```
\\ Examples
a=random(0,2)
Print a
if a=0 then { goto 100 } else.if a>1 then 300
050 Print "This is ";
100 Print "not ";
300 Print "ok"

a=random(0,2)
Print a
if a=0 then {
     goto 100
} else.if a=1 then {
     goto 50
} else 300
050 Print "This is ";
100 Print "not ";
300 Print "ok"
```

´

identifier: LOOP   [ΚΥΚΛΙΚΑ]

LOOP is a command that change the state of a block of code and has effect when a block of code end. So one loop command is enough to change the state of a block of code (in curly brackets). The next time that block run has a non loop state, so if loop command is shadowed then the block at the end exit.

```
i=0
{ if i<100 then loop
     print i
     i++
}
```

You can use If condition Then Exit to leave a looping block. A Continue in a loop stated block act as a restart. A Restart is a jump to first line of block.

```
i=0
{i++
     print i
     if i>100 then exit
     loop
}
```

Avoid to use LOOP in a thread, because threads are run in intervals so they are already in a loop time based.

The strength of this command can delivered when we need a multicondition block to run again, so different If choose the Loop command, some other the exit command, some other the continue command

```
I=0
{
I++
IF I<100 THEN LOOP    ' WE SET A LOOP CONDITION HERE
PRINT "LOOP CONDITION CHECKED IN THE END OF THE BLOCK"
}
PRINT I

I=0
```

```
{
I++
IF I<100 THEN RESTART    '  THIS IS A COMMAND
PRINT "THIS PRINT ONLY ONE TIME AT THE END"
}
PRINT I
```

identifier: MAIN.TASK   [ΚΥΡΙΟ.ΕΡΓΟ]

```
MAIN.TASK 100 {

}
```

Like EVERY but this is a thread so it has time to run as other threads. We can EXIT or perform asn ESC to exit from this. We can't have nested MAIN.TASK in a module.

identifier: ON   [ΑΠΟ]

On Variable GOTO label1, label2, ...

Labels can be numbers

Jumps can do in the same block of code, forward or backword.

```
module B {
{
      5 read a
   30 on a goto 100,200,300,400
   50 Print a
   100 Print "10000"
   200 Print "20000"
   300 Print "30000"
   400 Print "40000"
   600 Print "ok"
}
print "OK..............."
}
B 0
B 1
B 4
B 100
```

a<1 exit block
a =1 goto 100
...
a=4 goto 400
a>4 continue to next line (here 50)
So for a>total number of labels we can have an error condition or a chance to perform a better On GOTO.

identifier: PART   [ΜΕΡΟΣ]

if M= true then Part skip block
When Part block execute M created if not exist and turn to true.
At the end of execution of block M turn to False

```
Part {
     Print  "ok", M
} As M
```

used where we want code to run without other code run concurrent.
we can call subroutines or modules from block.

identifier: PROFILER   [ΑΝΑΛΥΤΗΣ]

You can measure the time consuming in a piece of code. Just place PROFILER in the beginning and then copy the TIMECOUNT read only variable.

```
Profiler
k=0
For i=1 to 1000 {
      k++
}
print timecount
```

identifier: REPEAT   [ΕΠΑΝΑΛΑΒΕ]

Repeat or Do (is the same statement)

```
Repeat {
      If b=3 Then Exit
      if k=5 Then Continue
      ................
} When A<>1


Repeat {
      If b=3 Then Exit
      if k=5 Then Continue
      ................
} Until A=1

Repeat {
      IF B=3 Then Exit
      if k=5 Then Continue
      ................

} Always
```

use REPEAT or DO, is the same

```
Do {


} When b<=3

Do {

} Always
```

```
Do {

} Until b>3
```

Without Block (scan to find Until, so with block is faster):

```
Do

When b<=3

Do

Until b>3

Do

Always b>3
```

identifier: RESTART   [ΞΕΚΙΝΑ]

Restart as a command to jump to start of a block

```
i=0
{i++
if random(10)>5 then restart   ' this can overdrive the loop so the print command can print
number>10
print "normal", i
if i<10 then restart   ' and this condition says that no loop occur if i>=10...after the print
command,
print "last command"
}
```

Look THREAD and MOVIE for Restart id (for other purposes) identifier: SELECT   [ΕΠΙΛΕΞΕ]

You can use BREAK and CONTINUE in curly brackets for special purpose in a select case (see that exclusive)
You can use Goto то jump out of block in a Case or Else clause.
Without brackets, and in one line of commands that allowed after a Case statement Break and Continue act as normal.
After a Case can be one line with commands (separated with colon) or a block of commands, but no a blank line.
You can use Arithmetic or String variable for cases
You can have conditions and variables or user functions in every case statement. All the conditions are investigated if no {Break} command break that, or a condition founded true, and for that case the next line command or block of command executed.
There are no optimization. Executed as it is.

```
SELECT CASE A
CASE 1 TO 100, 120, >500
....
CASE 105
{
....
}
ELSE
....
END SELECT
```

```
{
a=9
here:
a-=4  ' case 5 and then case 1
Print "Last Test"
\*  break in level of first block after CASE ..break the cases, so all cases run including the conditions in the case
\* break out of block like case 1.5 or in deeper block like in case 1...are a big break
Select case A
Case 1
     {if true then { break } else continue} '  check with if false - here break is like in case 1.5
Case 1.5
     break  ' that is a break for module , or outer block...like the one here..
Case 2
     {if true then break}    ' break the select structure so case 4 and 5 command are run
Case 4
     Print "case 4"  : Print "now we can execute a second statement in same line"
Case 5
     goto here   ' jump backward
Else
     {
     Print "this is a block of code"
     Print  "No jump can do from here to go out of block"
     Print "A break and a continue have specific duty in a block inside case"
     }
End Select
print "no error"
}
print "end"
```

identifier: THEN   [TOTE]

Look If

identifier: THREAD.PLAN   [ΣΧΕΔΙΟ.ΝΗΜΑΤΩΝ]

1) Thread.plan Concurrent
Make each thread to execute one command and pass the execution to other thread. Any block executed as one command. Use If inside block because there is a problem if we have ELSE after THEN (the split is perfect but not for IF)
2) Thread.plan Sequential   \\ by default

No two threads execute commands at the same time.
We can't change plan if threading pool have threads
In this example using concurrent plan we can stop "for next" changing m variable because thread L can be run when task.main thread run also. If we use sequential plan, m is changed by any thread in turn of execution. Using For k=1 to 100 { } we not allow other thread to run concurrent.

```
thread.plan concurrent
\\ thread.plan  sequential
m=0
thread {print "ok" : m=50 } as L interval 10
main.task 100 {
m=0
for k=1 to 100
print k
if m>0 then keyboard " " : exit for
next k
if inkey$=" " then exit
}
```

identifier: TRY   [ΔΕΣ]

Error trapping
1) using Variable

```
Try Var {
     block of code
}
```

if var=0 then we have an error Error$ and Error number can be readable
2) non stopped Try

```
Try Var {
     block of code
}
```

We can produce a software error.

```
Error "this is my error"
```

or

```
Error 1000
```

```
try a {
     error 100
}
if not a then print error : flush error  \\ because ERROR 100 is on error stack
try a {
     x=12/0
}
if not a then print error$
try {
     Print "A try Var block can't stop Break command"
     try a {
          break
     }
     print "can't print here"
}
Print "A try block can stop Break command"
```

We can check also an expression with Valid()

```
print Valid(thisisnotavariable)
               0
a$="1"
Print valid(a$)
               -1
```

Also you can use TEST to run step by step and read variables to see any faulty condition..

identifier: UNTIL   [MEXPI]

Look REPEAT

identifier: WAIT   [ANAMONH]

1) Wait milliseconds
2) Wait
this is like Wait 0

Wait command allow to run threads, so it is a waiting state for a module, to give time for threads (after the end of the module all threads from that module are erased).

WAIT 0
allow task manager to run some threads in a thread, except when we Set Fast !

Here is an example to demonstrate the use of refresh, wait and a high value for refresh to perform only a final refresh

```
\\Thread.Plan Concurrent
Thread.Plan Sequential
Clear  \\ clear static variables
Set Fast !
\ Set Fast
\ Set Slow
Form 80
Cls #ffaa22
Pen 1
Refresh 50
Thread {
    Print @(Width-8,0,width,1,15);str$(now,"hh:mm:ss")
} as M interval 20
wait 100
Thread M interval 1000
i=0
Batch=100
Profiler
For n=1 to 3 {
    Thread {
        For k=1 to Batch : move 6000,6000 : draw to random(12000), random(10000) , L: Next k
: i+=Batch
        refresh 1000
         } As G Execute Static L=random(100000)
     Thread G interval 20
}
Main.Task 100 {
    if i>8000 then exit
    if mouse then exit
}
Refresh
X=Timecount
a$= Format$("Time to complete {0:2}, lines {1}",X, I)
Clipboard a$
```

```
Pen 15 {print  @(0, height-2),a$}
Refresh 40
```

identifier: WHEN   [ΟΣΟ]

```
Look Repeat
i=0
Repeat
          i++
          Print i
When i<10
```

identifier: WHILE   [ΕΝΩ]

```
While condition {
block of code
}

While Iterator_Object {
    block of code
}
```

use exit for exit inside block
continue or restart to restart the block without finishing
break to exit for module (but break cannot break a block inside a Case)

You can nest those Whiles with Do Until or any other block of code.

```
\\ first variant
a=10
While a>0 {
     Print a
     a--  ' here a change so condition a>0 maybe change
}


\\ second variant
k=(1,2,3,4,5)
k1=Each(k,1, 3)
```

```
Print "k1"
While k1 {
    ' 3 times
    Print Array(k1)
}
k1=Each(k,1, 3)    ' from start to 3rd
k2=Each(k,-1, 3)  ' from end to 3rd (we can use here -3 as 3rd from last)
Print "k1, k2"
While k1, k2 {
    ' 3 times
    Print Array(k1)*Array(k2)
}
k1=Each(k,1, 3)
k2=Each(k,-1, 3)
Print "k1*k2"
While k1 {
    While k2 {
        ' 3x3 times
        Print Array(k1)*Array(k2)
    }
}
```

Exist with no block

```
While [comparison or iterator, ] comparison or iterator

End While
```

Group: STACK COMMANDS - ΕΝΤΟΛΕΣ ΣΩΡΟΥ

identifier: COMMIT   [ΑΝΑΘΕΣΕ]

like Read, the commit statement work only for arrays, member of groups
We pass the pointer of array to the member name.

```
group alfa {
        dim a()
        module k {
                commit .a()
        }
}
dim z(20)=1, z1(30)=100
alfa.k  &z()
Print alfa.a(19)=1
alfa.k  &z1()
Print alfa.a(19)=100
```

identifier: DATA   [ΣΕΙΡΑ]

We can import DATA lines of numbers and strings
That data can be loaded from a module by loading and after the execution we send the data to stack
So data put to the stack items. We can push items to the top of stack or we can use DATA to append items to the bottom of stack
So DATA 1,2 leave 2 to the bottom and a second DATA 3, 4 leave 4 to the bottom
Stack is using to hold values. We can use the stack to pass a reference, a reference is creates as a string.

```
FLUSH ' clear the stack
PUSH 2, 1
DATA 3, 4
```

now we have top item 1 and after that 2 and after that 3 and the last one...4

Why I use flush to clear the stack? Because if some items are in the stack then the DATA write values behind that so we take from read the wrong values.
When we call a function then new stack created. All modules have common stack if are exist in the same thread. We can call a module from a module and we can pass values to stack. That values are in priority, at the top and place by internal use of DATA and then merged all other items. When a module end, maybe can leave some items in the stack. Threads use common namespace with modules but have own stack. So we can deliver data using stack but we can use variables and arrays.

Example using stack as FIFO (First in First Out)

```
Flush
DATA 1,2,3,4
DATA 5,6,7,4+4

For i = 1 to 8 {
    read a
    print a
}
```

DATA 1,3,4 ' The last item in that list become the last item in the stack (we can say that this is the bottom item)
See PUSH

identifier: DROP   [ΠΕΤΑ]

DROP 4
drop 4 items from stack,
if STACK.SIZE<4 then an error happen

DROP 0

drop nothing, no error

If we give a negative number then we get an error

we can use Drop, Stack.Size and Stackitem() and Stackitem$() to read a frame of parameters. Only a variable needed to mark the frame: S=Stack.Size, then we can do anything to the stack and we can return to parameter frame using DROP stack.size-s
If we get a negative number then we miss some parameters from frame.
We can check the type of stack items using Envelope$(). If we try to read a string using STACKITEM() then an error occur. So a frame must be a set of known items and types. If we have 4 modules that need the same parameter we can place it in stack and we can call each module without passing each time a variable (so no need to read variable, copy to stack, read to a local variable)

identifier: FLUSH   [ΑΔΕΙΑΣΕ]

FLUSH
empty the stack
Two more variants
FLUSH ERROR ' empty last error message
\\ Not used form 9.4 version FLUSH GARBAGE ' run manual garbage collector

Stack Lesson
A module hold parent stack. A function start a new stack every time run.
So calling a module we can search stack for returned values.
We can pass values and references to values.

Use PUSH to push on top (Last in First Out)
Use DATA to merge values at the bottom (First In First Out)

```
PUSH 1,2,3
STACK
DATA 1,2,3
DATA 4,"string also", 5,6
STACK
3  2 1 1 2 3 4 "string also" 5 6
```

To pop values use Number for numbers and Letter$ for strings

Use READ to pass values to variables

```
flush : push 10 : read x : print x
B=10 : push &B : read &K: print K=B
```

&B will be a string as a name for used as reference to K. Only Read (and special Commit) can found references..

```
B=10 : push "B" : read &K: print K=B
```

If FF isn't a variable a Push &ff produce error. But if we push "FF" and before the reading we make a FF variable the link can happen..errorless.

Look SHIFT, OVER, DROP, STACKITEM(), STACKITEM$(), NUMBER, LETTER$, ISNUM, ISLET, EMPTY, FLUSH, STACK, ENVELOPE$() STACK.SIZE() STACK$(), READ, COMMIT

identifier: OVER   [ΠΑΝΩ]

1) OVER 5
2) OVER
3) OVER 5,5

Copy over the top the item in 5th position in the stack (if we have less items then an error occur)
The second case is equal with OVER 1 and this duplicate the top item (same item in position 1 and 2)
The third make double the 5 top elements of stack.

Look SHIFT, OVER, DROP, STACKITEM(), STACKITEM$(), NUMBER, LETTER$, ISNUM, ISLET, EMPTY, FLUSH, STACK

identifier: PUSH   [ΒΑΛΕ]

Any module, function, thread has a mate stack for values. Modules that called from modules have the parent module stack. So A module can call a module passing values to stack.

Functions are parents for the stack . Stacks from parents are lost if parent die..(end or exit). Only running modules, functions and thread have modules. A thread is a special module part that can be run in intervals and share variables from "brother" module, but has own stack. When in a module we call a function some data written in a temporary stack and merded to functions stack. The same happen but without making a new stack when a module call another module. So from a module we can get many values as items on the stack.

A function return only one value, but in expressions. Modules cannot used in expressions. Threads have no name, but a handler for each. When a thread call a module then that module use the threads stack. When a thread call a function then new stack created for that function. Every thread run all in a time slice, an interval. So any module or function in the thread are cleared when that slice ends. So anything in those stacks cannot used to return values. The only way for a thread to act to values is by using the same namespace as those of the creator module, and read and alter variables and arrays.

Some commands from DATABASES and from DIALOGUES uses the stack to return data

We can use Push to send a list of items to current stack. If we use SET PUSH then any variable will be from level 0 (global level) but the stack will be the current.

PUSH and DATA are the commands that we can use to handle not only by value items but we can include by reference IF we know when they putting and know when we can get.

```
A=1
Push &A
read &B
B=5  ' now A=5
```

```
PUSH 1,3,4   ' the last is the first or top in the stack (first that we read)
DATA 1,3,4  ' The last is the last in the stack or in the bottom.
```

So if we place some PUSH commands...we send data to the bottom
But if we use DATA we spread data as we read, the last line...last, the last item in line...last
So DATA and READ are like BASIC's

When we call a MODULE the system make a temporary stack and we pass values like a DATA and then that stack is merged on top of the current stack. So when we read values from the module we know..that there must be some values for us, and so we can get references. The references are strings containing the name of the variable or array that the READ instruction has to link with a new variable or array accordingly. So the linking process can be many times (if we get the value and push it back three times we can create by a three variables reading three clones of the original variable).

In some languages to passing values to a function conform a way as exclusive written in the script. For some languages there is an overloading method, where many definitions have same function name. But in M2000 any module may have some commands to interpret the stack. Any module may alter context (by the parent) or use INLINE code, that can import in a string, to run it. So by using the STACK not only we send values, but also we change the behavior of the module. This called interaction. Modules interact with other modules to make things happen. Feedback can be sent through the stack. So a PUSH isn't only for sending to the module we call but for sending back to the module that calls.
M2000 uses functions to handle items and read before pop about the type of it (references are string type...only READ understand what that is). See STACK, ENVELOPE$, LETTER$, NUM, ISNUM, ISLET, EMPTY and command DROP
We can use strings STACKS, a normal string arranged to work as stack (but stacks aren't strings from 6th version)

We can push expressions (as results)

```
PUSH "a">a$, 12+4>5
```

Example for use by reference (open a module and copy those lines)

```
    dim pp(10)
    pp(4)=1234
    push &pp()
    read &kl() ' we make a new name but the data are the same
    print kl(4)
    dim kl(50) ' we redim the pp() array
    g$="George Karras"
    module aa {
        read &kk(), &k$
        k$="The best of "+k$   'we change some values
        kk(4)=kk(4)*2
    }
    aa &pp(), &g$  ' we pass by reference to module aa
    print g$
    print pp(4)
```

From that addition from a coincidence happen something that was not in plan. The variables and array items are holded in variants. So we can get the reference from a number and link to a string variable.

```
A$="100"
push &A$
read &A
print A
```

```
' now A=100 and A$="100" both share the same container.
```

From 7th edition, of M2000 language, we have document strings, a modified version of a string that holds paragraphs. We can reference document type strings, and we can link to a number, but if we place a value to that number then the string looses the linked document object and became a number value that can be read as a string.

A fast info here
1. Documents in stack are pushed as strings
2. References are pushed as strings
3. Function references are strings as "{definition}". So function reference is a copy. Functions reference from classes (groups) use an expanded form including the reference of group also just after last closed bracket.
4. Declared objects (Word.Application or other) can't exist in Stack. We can use references to variable to pass in a call to module.

identifier: READ [ΔΙΑΒΑΣΕ]

```
READ A$, B$
```

defines variables (not documents), and pop values from stack
cannot define array items, but can read for arrays items

```
READ A(1),B%(10,2), C$(10)
```

B%() is a two dimension integer array (all arithmetic are double, so this integer is a double with no decimals

```
READ  &A(), &B
```

We can read references to arrays and variables (and documents, and groups also). References are stored as strings but READ when see "&" before a name, looking for reference and make local variables (or global if we set SET before).

We can pass an array and we can redimension it using the reference. The actual variable always live before start and end a module or function thread that get the reference.

Any change of one of the reference change the actual data.

For documents, each read is an append to document.

```
A$=STACK$(1,2,3,4)  ' we can create stacks as strings
READ FROM A$, K , ,M    'we can read  specific part of the A$.
? K, M
   1    3
```

We can read from a Goup in the order that variables are written in Group List, we read always using by reference pass

```
Group thisgroup {n, m}
READ FROM thisgroup, A, B
A=2
? thisgroup.n
```

We send the reference from a module's parameter list, or a function's or by using PUSH or DATA. In all these situations we write to stack. So only a READ can create a reference. We can create reference for an array item, but from an array to an array. We can't change reference, if b1 reference a1 then that's all. We can't read a reference if variable exist, but no fault return and reference dropped from stack. We can make many reference for one variable, one by one or one from one...all references are the same. Look the following example.

```
A=100
Push &A
Read &b   ' b is a reference to A
Push &b   ' b place the A reference  not the b
Read  &C  'c is a reference to A
c++
? a, b, c
101,101,101
```

This is a way to misuse the reference system
This cannot happen if we use Functions and call them as modules by using CALL command

```
' this is a  module called "a"
flush
alfa=100
push &alfa        ' pushing string "A.alfa"
module k {       ' we can't call a module with same name of the calling module (normally modules
have no recursion)
    module a {     ' so we create k to hold a
        print "this is a"
        alfa=150     ' now alfa has a full name A.alfa,  so this variable not created here...is
actually the first A.alfa
        stack
        module c {
            print "this is c"
            stack
            read &b
            print b    ' now b is reference then one and only one A.alfa
        }
        c  ' now b is erased
    }
    a    '  in a no variable created (Except in module b in module a)
}
k
? alfa
150
```

```
' example
' we put multiline strings to command Push.
push {aaa
}, {bbb
}
a$=""
document  a$
read a$
read a$
report a$
```

we see a two line text:

```
aaaa
bbbb
```

identifier: REFER   [ΑΠΕΔΩΣΕ]

Refer is a variant of Read, we can use it to make references without using & before names also we can use it to make references to all members of a group (variables and arrays).
In the example bellow, x and z$ are private, hidden from outside, but refer can get references (by the order in the list of group members)
We can't make a second reference to one that exist. We can make references in a block for temporary definitions, like For object { }.

```
group alfa {
private:
        x=1, z$="ok"
public:
        module print {
                Print .x, .z$
        }
}
alfa.print
\\ now we can refer to private variables
\\ from outside the group
refer from alfa, x1, z1$
Print x1, z1$
x1++
z1$+="...."
alfa.print
```

identifier: SHIFT   [ΦΕΡΕ]

1) SHIFT 5
2) SHIFT
3) SHIFT N, M

shift 5th stack item (1st is in position 1, this is the top item)
2nd variation is same as SHIFT STACK.SIZE
3rd from N, M items
4

```
FLUSH
\*  we use in this example the FIFO (all items make a stack that join to the bottom of the stack, but now stack is empty)
DATA 1,2,3,4,5
SHIFT 3
\* we use stack to display the stack
STACK
    3 1 2 4 5

\* use SHIFTBACK 3 to place the top item to the third place
SHIFTBACK 3
STACK

\* In this example we push some items (we can push numbers or letters -strings-)
\* and we change the value of third item
flush
push 1, "alfa", 3,"beta"
stack
shift 3 : push number+2: shiftback 3
print stackitem(3)

\\Simple example
data 1,2,3,4,5
```

```
stack
shift 3,3
stack
shiftback 3,3
stack
flush
```

A module in a module has same stack so we can make "stack commands" as modules
use DROP OVER SHIFT SHIFTBACK

```
module swaptop {
     shift 2
}
module Dup {
     over
}
module Dup2 {
     over 2 : over 2
}
Module DupN {
     read N
     for i=1 to N { over N}
}
```

identifier: SHIFTBACK   [ΦΕΡΕΠΙΣΩ]

1) SHIFTBACK 5
2) SHIFTBACK
3) SHIFTBACK N, M
Shift top to 5th position in the stack (if we have less items then an error occur)
The second case is equal with SHIFTBACK STACK.SIZE and make top item as bottom item
Third case: Shift from top M items to N
Look SHIFT, OVER, DROP, STACKITEM(), STACKITEM$(),
     NUMBER, LETTER$, ISNUM, ISLET, EMPTY, FLUSH, STACK

identifier: STACK   [ΣΩΡΟΣ]

About stack
M2000 has a stack system to place by value items without names but in an order. We can place

numbers or strings. There is a system to place a reference to a variable but this reference is a string item and only a READ command can understand that this was for reference and not for value.
Any function and thread has own stack, but modules in same thread have a common stack.
Every line of code run in a module (or by SET command as from command line interpreter), use that module as namespace, and can push or pop values to a stack without name or can define variables and arrays with name.
Return form database commands placed on the stack. We can use stack to send information using pipes.
Here we can see how to use these major stacks with temporary stacks in strings

1) STACK
print to output (background, screen, printer or layer) the stack items
We can see the stack in the Control Form (used to trace a running program) using TEST command or ctrl & I (also any non using combination open the Control Form)
2) STACK A$
We can prepare stacks in string variables, with STACK$() function, and here we can dump the string to the stack
We can use array item as temporary stack. We can execute one by one command in command line interpreter or we can write all of this example to a module using edit a and copy from this help screen (with some clicks cursor become alive..)

```
            dim a$(10)
            a$(2)=stack$(1,2,"ok")
            stack a$(2)
            stack
            ? len(a$(2))
```

3) STACK A$, "NN"
We send only the top two numbers of string A$ who act as a stack in a variable. We can use S for String Item
We can read the kind of items in a temporary string stack by using ENVELOPE$()   with parameter
So ENVELOPE$(STACK$(2, 20.5)) is equal to "NN"
4) STACK string expression
in 2 and 3 the variables looses content
We can use string expression but if interpreter find that result is a stack. So we can add variables ready with stack content.
5) STACK NEW {}

```
    FLUSH
    DATA 1,2,3,4
    STACK NEW {   \\ REPLACE STACK WITH A NEW ONE
        PUSH 10,20
        DATA 1,2
        STACK
```

```
  }  \\ NOW OLD STACK RETURN
  STACK
```

6) STACK VARIABLES (AS OBJECTS)

```
 a=Stack
 a=Stack:=1,2,3  ' we can put  (stack:=1,2,3) as a value too
 b=Stack Up a, 2  ' 1,2 to b and leave 3 to a
 a=Stack:=1,2,3
 b=Stack Down a, 2  ' 2,3 to b and leave 1 to a
 Print b ' 2,3
 a=Stack:=!a,!b
 Print a   ' 1,2,3

  \\ use ! before a stack object as argument to pass stack items as arguments
  Function b (x,y) {
       =X**Y
   }
   s=stack:=3,2
   s1=stack:=100,2
   Print b(2,4), b(!s) , b(!stack(s1))
   Print len(s), len(s1)



   \\Create one
   A=Stack  ' Empty
   A=Stack:=1,2,3  ' three items, top is 1
   Print A  \\ show items
   Stack A  \\  place items to modules stack, at the bottom, A is empty
   A=[]  \\ get modules stack. Module stack is empty now use [ ] without space between
   Print Len(A), StackItem(A), StackItem(A,1)
   Stack A {
       Over 2  ' get a copy of 2nd as a new top
       Drop 2 ' drop 2 from top
       Read A, B  ' read means, pop item
       Push A, B ' now top is value of B - before was value of A
       Shift 2 ' now top is value of A
       ShiftBack 2 ' now top is value of B
   }
   Stack Stack(A, 3)  ' pass a copy of 3 first items of A to bottom of module stack
   B=Stack:=100,"string", 200
   Stack B {
       \\ place B as module stack
       Stack Stack(A, -3)  ' pass a copy of last 3 items of A to bottom of module stack
```

```
        \\ now at the close, old module stack restored.
    }
```

Example:

```
    Flush
    Data 1,2,3,4
    a=Stack:=20,10,1,2
    Stack a {
        Stack
        Drop 2
        Stack
    }
    Stack
    Print StackItem(a), Len(a)  \\ 1   2
    Stack a {
        Read x, y
        For This {  \\ block for temporary use - all new names erased at the end of blockl
            Dim m(10)=5
            Push m()
        }
    }
    n=Stack(a)  \\ Get a copy of stack a
    Stack n {
        For This {
            Read k  \\ we get array in a variable (arrays have two interfaces for M2000)
            k+=100  \\ this interface has some operators
            Push k  \\ we can push it back.
            Print k  \\ all array items 105
            Stack a {
                Read m()
                Print m()   \\ all array items 5
            }
            Print Len(a)  \\ 0
        }
    }
    a=Stack:="Alpha","Beta", 100  \\ := is a Data command for stack, it is optional
    Print Len(a), Len(n)  \\ 3 & 1
    n=Stack(a,n)  \\ Here we get a new stack with items from both, but from 'a' first
    Stack n {
        Stack    \\  just show stack item
    }
    Stack a {
        Stack
    }
    a=Stack(a,-3)  \\ reverse 3 last (a has 3 all, so reverse the order of items in a)
    Stack a {
        Stack   \\ show stack items
    }
    Stack Stack(a,-2) {  \\ Stack() return always a new object)
    \\ last two items in reverse
    \\ a isn't change
        Stack
    }
    Stack a {Drop} \\ drop top item
    a=Stack(a,-Len(a))
    nn=Each(a)
    While nn {
        Print StackItem$(nn)
    }
```

Example (old examples here)

```
---------------------
MODULE A {FLUSH
A$=QUOTE$("AAAAAAA",1,"BBBBBBB", 2)
B$=STACK$("AAAAAAA",1,"BBBBBBB", 2)
PRINT A$
PRINT B$
TRY K {
INLINE "DATA "+A$
READ P$,Q,R$,T
PRINT P$, Q,  R$, T
}
IF K THEN PRINT "OK"
STACK
Q$={PRINT "OK"
PRINT 1*2
TONE 1000, 100
}
INLINE Q$
}
```

SEE FLUSH, ENVELOPE$, LETTER$, NUMBER, ISNUM, ISLET, EMPTY, DROP

Group: DEFINITIONS - ΟΡΙΣΜΟΙ

identifier: AUTO_ARRAYS   [ΑΥΤΟΜΑΤΟΙ_ΠΙΝΑΚΕΣ]

Tuple as auto arrays using ( ) and at least one comma

```
a=(,)    len(a)=0
a=(1,)  len(a)=1
a=(1,2,3,4)
Print a
m=each(a End to Start)   ' same as each(a,-1,1)
While m {
     Print array(m), m^
}
Print a#sum(), a#min(), a#max()
where=-1
Print a#max(where), where
where=-1
Print a#min(where), where
\\ for string
Print a#min$(where), where
Print a#max$(where), where
\\ shearch for value 3,  from 3rd position (0 is for first position)
Print a#pos(3), a#pos(2->3)
\\ search items in an array
Print a#pos(1,2), a#pos(2->1,2)
Print a#pos((1,2)), a#pos(2->(1,2))
Print a#pos("a","b"), a#pos(2->"a","b")
Print a#pos(("a","b")), a#pos(2->("a","b"))

a=(1,2,3,4,5)
Print a#rev()
Print a#sum()=15
Print a#max()=5, a#min()=1
k=-1
L=-1
Print a#max(K)=5, a#min(L)=1
Print K=4 ' 5th position
Print L=0 ' 1st position
Print a#pos(3)=2 ' 3rd position
Print a#val(4)=5
\\ tuples in tuple
a=((1,2),(3,4))
Print a#val(0)#val(1)=2
Print a#val(1)#val(1)=4
a=(1,2,3,4,5,6,7,8,9)
fold1=lambda ->{
     push number+number
}
Print a#fold(fold1)=a#sum()
Print a#fold(fold1,1)=a#sum()+1
even=lambda  (x)->x mod 2=0
b=a#filter(even, (,))
Print b  ' 2 4 6 8
Print a#filter(even)#fold(fold1)=20
map1=lambda (a)->{
     push a+100
}
c=b#map(map1)
Print c  ' 102,103, 104, 105
numbers=lambda p=1 (x) ->{
     push x+p
     p++
}
oldnumbers=numbers  ' we get a copy of numbers with p=1
c=c#map(numbers)
Print c  ' 103, 106, 109, 112
zfilter=lambda -> number>106
tostring=lambda -> {
     push chrcode$(number)
}
oneline=lambda -> {
           shift 2   ' get second as first
           push letter$+letter$
}
Line$=c#filter(zfilter)#map(tostring)#fold$(oneline,"")
print Line$="mp", chrcode$(109)+chrcode$(112)
zfilter=lambda -> number>200
Line$=""
Line$=c#filter(zfilter)#map(tostring)#fold$(oneline,"")
\\ lines$ can't change value becuse filter has no items to give
Print Line$=""
\\ if we leave a second parameter without value the we get No Value error
Try {
     Line$=c#filter(zfilter, )#map(tostring)#fold$(oneline,"")
}
Print error$=" No value"
```

```
\\ second parameter is the alternative source
Line$=c#filter(zfilter,(109,112))#map(tostring)#fold$(oneline,"")
Print Line$="mp"
c=(1,1,0,1,1,1,1,0,1,1,0)
\\ hard insert
Print c#pos(1,0,1)  ' 1  means 2nd position
Print c#pos(3->1,0,1)  ' 6  means 7th position
\\ using another tuple
Print c#pos((1,0,1))  ' 1  means 2nd position
Print c#pos(3->(1,0,1))  ' 6  means 7th position
t=(1,0,1)
Print c#pos(t)  ' 1  means 2nd position
Print c#pos(3->t)  ' 6  means 7th position
```

identifier: BINARY   [ΔΥΑΔΙΚΟ]

```
BINARY {
    kwOVA5kDkQMgAKcDkQOhA5ED
}  AS A$
PRINT A$
```

In block there is binary data in BASE64. We get in A$ the data as binary (decoded)

We can use numeric variable to get a Buffer with data. We can use array items, numeric to get buffer, or string to get data in string.

We can use menu Insert Resource to insert this structure with data from file

identifier: BUFFER   [ΔΙΑΡΘΡΩΣΗ]

```
Buffer clear alfa as integer * 100
a$=Field$("Panther", 100)
Print Len(a$), alfa(0)   \\ 200 byte, real address at offset 0
\\ We use always Return to place data to buffer (can take list of offsets/values)
Return alfa, 0:=a$
\\ read each character as 2 bytes, and is equal as we read with Mid$()
For i=0 to 6
    Print Eval(alfa, i), Chrcode(Mid$(a$, i+1,1))
Next i
Print Len(alfa)  \\ 200 (2*100)
m$=Eval$(alfa,0,7*2)
Print Len(m$), m$
Return alfa, 30:=65000  \\ 0 .. 65535
Return alfa, 1!:=255 \\ at offset +1 byte
\\ read from position 30 (x2bytes)
\\ we can read per byte
\\ 253*256+232=65000
\\ position 60 value 232 (Low Byte) and position 61 value 253 (High Byte)
\\ in Eval() if we use as byte or abything as ... then first number is an offset in bytes
\\ without "as ..." is an offset of spaces, here integers (2 bytes)
\\ so 30 is +60 offset, but 60 and 61 are equal +60 and +61 offsets
Print Eval(alfa,30), Eval(alfa, 60 as byte), Eval(alfa, 61 as byte)
```

2) use of a structure

```
Structure MyStruct {
    a as byte*8
    b as integer*20 ' for string is ok
    c as long*4
}

Print MyStruct("a") \\ offset 0
Print MyStruct("b") \\ offset 8
Print MyStruct("c") \\ offset 48
Print Len(MyStruct) \\ 64 bytes
Buffer MyBuffer as MyStruct*20
Print Len(MyBuffer) \\1280 ή 64*20 bytes
\\ 6th MyStruct (5*length of MyStruct), at b offset (+8)
Return MyBuffer, 5 ! b :="George"
Print Eval$(MyBuffer, 5 ! b, 6*2) \\ 2 byte each letter
Return MyBuffer, 3 ! b :=str$("George"+chr$(0)) \\ 7 bytes
Print chr$(Eval$(MyBuffer, 3 ! b, 7)) \\ 1 byte per letter
```

3)Example with nested structures

```
\\ structures inside are inventrories (objects)
\\ values are offsets. Print beta("kappa") return offset

structure stringA {
    str as integer*20
}
structure beta {
    Alfa as byte
    align1 as byte*3
    Kappa as long*20
    name as stringA*10
    kappa2 as double
}
Buffer clear alfa as beta*100
Print Len(alfa), len.disp(alfa)
Print 5*len(beta)+ beta("kappa"), beta("kappa")
Return alfa, 50!kappa!10:=50
sLong=4
\Return alfa, 5*len(beta)+ beta("kappa")+10*sLong! :=50 as long
\Print eval(alfa, 5*len(beta)+ beta("kappa")+40  as long)
Print eval(alfa, 50!kappa!10)
Return alfa, 50!name!5:="George", 50!name!6:="Hello"
Print eval$(alfa, 50!name!5, 20)
Print eval$(alfa, 50!name!6, 20)
Print Len(alfa), type$(beta)
```

4) Sign/Unsign values

```
buffer alfa as long * 50
return alfa, 5:=240,6:=uint(-5),30:=3405435354
Print eval(alfa,5), sint(alfa,6), sint(alfa,30)

\\ a sign long to unsign
A=uint(-500)
\\ reverse, an usign to sign long
Print sint(A)
\\ we can use Long B to keep an unsign
Long B=sint(3405435354)
Print uint(B)
\\ we can use normal variables to keep an unsign
\\ but these are 8 byte double
AA=3405435354   \\
A%=3405435354  \\ internal is a double
Print AA, A%
Return alfa, 10 :=AA+1 as double  \\ 8 bytes - offset 10*4=+40
Print eval(alfa, 40 as double)  \\ offset  40 byte
buffer alfa1 as integer * 100
Return alfa1, 10 :=AA as double  \\ 8 bytes - offset 10*2=+20
Print eval(alfa1, 20 as double)  \\ offset  20 byte
```

5) Execute machine code

Look Execute for example of Buffer Code

6) Use of Math library

```
Structure VecType {
        x As Double
        y As Double
```

```
          z As Double
}
Structure Vectors {
      Vec1 as VecType
      Vec2 as VecType
}
Print Len(VecType), Len(Vectors)
K=Each(VecType)
While K {
      Print Eval$(K, K^)+" offset at "+Eval$(K)
}
Buffer Clear Alfa as Vectors*10
Print Alfa(0), Alfa(0,"Vec1"), Alfa(0,"Vec2")

Print Alfa(0, "Vec1", VecType("X")) , Alfa(0,"Vec1", VecType("Y")), Alfa(0,"Vec1", VecType("Z"))
Print Alfa(0, "Vec1", VecType("X")!) , Alfa(0,"Vec1", VecType("Y")!), Alfa(0,"Vec1", VecType("Z")!)
Print Alfa(0, "Vec1")
Print Alfa(0, "Vec1"!)  ' REMOVE ALFA(0) OFFSET
Print Alfa(0!) ' ADD BYTES TO OFFSET ALFA(0)

VecDbl=Lambda Alfa, VecType (n, a$, b$) -> {
      =Eval(Alfa, Alfa(0, a$, VecType(b$)!) as double)
}
Declare Math Math
Print Alfa(1)-Alfa(0)
x_offset=VecType("x")
For N=0 to 9 {
      Method Math, "Vector",Alfa(N,"Vec1"), 1200,2500,1500
      Method Math, "Vector",Alfa(N,"Vec2"), 3000,2000,1000
      Print VecDbl(N,"Vec1", "x")
      Print VecDbl(N,"Vec1", "y")
      Print VecDbl(N,"Vec1", "z")
      Print Eval(Alfa, Alfa(N, "Vec1", x_offset!) as double )
      Print VecDbl(N,"Vec2", "y")
     Print VecDbl(N,"Vec2", "z")
     Method Math, "VecString",  Alfa(N, "Vec1") as Result$
     Print Result$
     Method Math, "VecString",  Alfa(N, "Vec2") as Result$
     Print Result$
}

Declare Math Nothing
```

Another example using String (BSTR type)

```
structure alfa {
    a as long
    c as String*5  ' pointers to string
    z as integer*20
    zero as long
}

Print len(alfa) , Type$(alfa("c"))
Buffer clear Mem as alfa*10
Return Mem, 2!c!4:="hello there"
Hex Eval(Mem, 2!c!4)    ' string address in Hex
Return Mem, 2!z:=Field$("Fix String",20) ' no zero
Print Eval$(Mem, 2!z, 10*2)  ' need to pass byte number for fix string
```

identifier: CLASS   [ΚΛΑΣΗ]

A class definition has a body of a Group object and produce a function which return an object. If the returned object assign to a new variable then a new object Group created. If it returns to previous defined group then we get a merge, where members with same name get value from new one, and members which are new added. If it returns to a array item, or inventory item, or stack item (at position through Return command) then only pointer change, the old one destroyed and the new one take place just using internal a pointer to new one.

One Module with same name as the class name used as constructor function (we don't return any value because class function always return Group)

Name for class can be with or without $ as last character. We use $ when we have Value function to indicate to expressions evaluator that the return value is a string.

We can include a Class: part (except of Private: and Public: parts), and any member as class members used only from constructor, and are not included in return group. So we can make constructor, or and some class definitions inside to exist once before the final returned group.
Parts Class, Private, Public can be exist more than one time (interpreter use flags to track the state of those parts)
If we define Properties, then any Private state change to Public automatic. (Properties are groups in group with values, and a link to a private value in parent group.)

So we can see the definition of Group.
Class functions in Module or Function body are global (until the exit of Module/Function)

Class can be defined inside group and that is a Group function. Also we can make group members using class name (after definition of class) before name of member.

```
Class Alfa {
private:
    z=10
    dim alfa(10)=5
public:
    X,Y
    module Alfa {
    read .X, .Y, .z
    .X/=.Z
    }
    Function FillArray {
        =.alfa()
    }
}
MyGroup=Alfa(10,20, 5)
for MyGroup {
    Print .X, .Y
    Try { Print .z}
    Try {Print .alfa(5)}
}
Dim K()
K()=MyGroup.FillArray()
Print K(3)

\\ Remove (deconstructor) function
Global RefAlfa=0
Class Alfa {
    id
    Remove {
        Print "Remove"
        RefAlfa--
    }
  Class:
    module Alfa (.id){
        RefAlfa++
    }

}
```

```
A->Alfa(3)
z->A
Print refAlfa=1
Clear A
Print refAlfa=1
Clear Z
Print refAlfa=0
```

identifier: CONST   [ΣΤΑΘΕΡΗ]

Variables for reading only

1)
```
    const a=10, b%=12, c$="ok"
```
2)
```
    global const a=10
```
3)
```
    global const a
```
4)
```
    const a
```
Any global variable can be shadow from a local one
so we use 3 and 4 in a module to get a global const inside module
we can use global without this way, but if we assign a value then a new local variable shadow the global one, so we set a copy of constant in a new variable using const a (for global const a).

When we list the variables with List command, we see constants with values in [ ]

identifier: DECIMAL   [ΑΡΙΘΜΟΣ]

```
DECIMAL A=1201290190192090291@
' Type Decimal 27 digits (we can use decimal point )
```

identifier: DECLARE   [ΟΡΙΣΕ]

```
Declare MessageBox Lib "user32.MessageBoxW" {long alfa, lptext$, lpcaption$, long type}
Print MessageBox(Hwnd, "HELLO THERE", "GEORGE", 2)
Remove "user32"
```
(if we call MessageBox() again then new reload happen)
All libraries removed when we finish the run of m2000.exe
```
Declare timeGetTime Lib "winmm.timeGetTime" {}
```

```
while MessageBox(Hwnd, "HELLO THERE "+str$( timeGetTime()),"GEORGE",2)=4 {
Print "one time again"
Refresh
}
```

Example for passing string by reference

```
Declare PathAddBackslash Lib "Shlwapi.PathAddBackslashW" { &Path$ }
P$ = "C:"+String$(Chr$(0), 250)
A= PathAddBackslash( &P$ )
Print LeftPart$(P$,0)
\* LeftPart$ is a new function and is like this Mid$(P$,1,Instr(P$,Chr$(0))-1)
```

Example of using an object inside a function using declare to create/destroy

```
Function MsgBox {
     Read Prompt$, Buttons, mTitle$
     declare Alfa "WScript.Shell"
     method Alfa, "Popup", Prompt$, 0, mTitle$, Buttons as msgbox_result
     declare Alfa nothing
     =msgbox_result
}
a=MsgBox({aaaaaa
безопасность
cccccccccccc
dddddddddddddd
},1024+64+3,"безопасность")

? a
```

Example 2:

```
Clear
Declare vs "MSScriptControl.ScriptControl"
Declare Alfa Module
Print Type$(Alfa)
With vs, "Language","Jscript", "AllowUI", true
Method vs, "Reset"
Print Type$(Alfa)
Method vs, "AddObject", "M2000",  Alfa
Method vs, "ExecuteStatement", {

          M2000.AddExecCode("Function BB {=1234 **number} : k=2");
```

```
        M=M2000.ExecuteStatement("Print 1234, BB(k)");
        M2000.AddExecCode("aa$=key$");
}
Method vs, "eval", {"hello there"} as X$
Print X$
Method vs, "eval", {"hello there too"} as X$
Print X$
List
Declare vs Nothing



mybuf$=String$(Chr$(0), 1000)
\\ This is for global use. You can declare local without using Global
Declare Global MyPrint Lib C "msvcrt.swprintf" { &sBuf$,  sFmt$, ... }
\\  ... means any number  and kind
\\  & means by reference
\\  ! before a numeric expression to pass it as Long
A=MyPrint(&myBuf$, "Γειά P1=%s, P2=%d, P3=%.4f, P4=%s", "ABC", !123456, 1.23456, "xyz")
Print Left$(myBuf$,A)

\\\\\\\\\\\\\\\\\big example\\\\\\\\
Declare form1 form
With form1, "visible" as visible
with form1, "Title", "Download Example"
layer form1 {
        window 12, 12000,6000
        cls #333333
        cursor 0,2
        report 2,"close form to exit download"
}
Method form1, "show"
cls,0
report 2, "Download 3 Files using Download object"
cls, 10
declare withevents d(2) download
w=0
m=row
Function d_Start(new con, host$,port) {
        cursor 0,m
        Print Part con, host$, port
        Print under
        m=row
```

```
		w++
	}
	Function d_DownloadProgress(new con, many, total) {
		cursor 0,3+con
		Print over $(4,10), con,  many;"/";total
		\\ do not use Refresh here
		cursor 0, m
		layer form1 {
			cursor 0,3+con
			Print over $(4,10), con,  many;"/";total
		}
	}
	Function d_DownloadError(new con) {
		cursor 0,m
		Print "DownLoad Error"
		Print  con, number, letter$
		m=row
		layer form1 {
			cursor 0,3+con
			Print over $(4,10), con,  "failed"
		}
		w--
	}
	Function d_DownloadComplete(new who, a$) {
		cursor 0,3+who
		print over a$+" - complete"
		layer form1 {
			cursor 0,3+who
			print over a$+" - complete"
		}
		w--
	}
	declare d() over   ' one more withevents
	\\ GetUrl$() function used if we wish the property url to connect on the fly, and disposed
	\\ because object is naked (not in a mHandler carrier), can't used as value
	\\ so we have to pass it by reference
	\\ M2000 pass array items by reference using a second variable, which pass by reference
	\\ and at the exit get the value from the variable and try to store it back to array to proper item
	\\ no error happen if array has no item with specific item number
	\\ this method is like copy in copy out
	\\ VB6 lock arrays if we pass an array item by reference. M2000 never lock arrays.
	Function GetUrl$(&where) {
		With where, "url" as url$
	=url$
}
Dim url$(3)
for i=0 to len(d())-1
	With d(i), "url" as url$(i)  \\ connect property url of d(i) to url$(i)
next i
Method d(0), "DownloadFile",
"http://www.vbforums.com/showthread.php?818583-Binary-Code-Thunk-Experiment",
dir$+"vbforums.html"
Method d(2), "DownloadFile", "http://www.cs.virginia.edu/~evans/cs216/guides/x86.html",
dir$+"x86.html"
Method d(1), "DownloadFile","http://rosettacode.org/wiki/Rosetta_Code",
dir$+"rossetacode.html"
cursor 0, m

for i=0 to len(d())-1
	Print Part "Read Property Url for"+str$(i)+":";~(11); url$(i)  ' GetUrl$(&d(i))
	Print Under
Next
Print "press left mouse to exit"
refresh
m=row
cls,m
kk=0
thread {
	cursor 0, m
	print kk
	kk++
	m=row
} as mm interval 10
While w
	cursor 0,2
	print over $(4),"live connections:";w, visible
	if keypress(2)  or not visible then exit
	bb$=inkey$
	\\ wait 0
	\\ threads need a wait or a Key$, or Inkey$ or a Main.Task or a Every, or a Modal form to
be opened.
	\\ com events no need this.
End While
threads erase
wait 100
cursor 0,2
```

```
print over $(4),"live connections:";w
cursor 0,m
print "done"
declare d() nothing
declare form1 nothing
```

identifier: DEF   [KANE]

1) Def Currency A, B, C

```
 for numbers
  Decimal, Double, Single, Currency, Long, Integer, Boolean
  all variables are local. If a local exist before def command then error happen.
  these variables have auto conversion when assign new value
  for strings
  String

  Def a=0&, b=10#, c=20~, d=2%, e=1000@
\\  long currency single integer decimal
  Print Type$(a), Type$(b), Type$(c), Type$(d), Type$(e)

 \\ we can use as type to give number, or special char
 \\ d get Long as the type of anything else without specific type
 \\ a Def without type is a Def Double
def long a=10, b=20, c as double=30, d=40, e=1000@
Print Type$(a), Type$(b), Type$(c), Type$(d), Type$(e)
```

2) DEF A(X)=X**2 : DEF A$(X$)=X$+X$

```
PRINT A(100), A$("00")
```

We can make functions like old basic, but internally they are normal functions:

```
Function A {
     Read X
     =X**2
}
Function A$ {
     Read X$
     =X$+X$
}
```

identifier: DIM   [ΠΙΝΑΚΑΣ]

DIM is a statement to dimension arrays

additions in ver 8
We can push arrays in stack and we can get a copy. We can return a copy of array from a funcrion. New Stock command. New class attach to arrays. New free style, we can make any item any class.

```
Function GetArray {
     read items
     dim a(items) : for i=0 to items-1 {a(i)=i+1}
     =a()
}
dim K()
K()=GetArray(10)
for i=0 to 9 {print k(i)}
```

Arrays in M2000 can have up to 10 dimensions. By default start from 0 in each dimension. We can change this using BASE 1 command. Each array hold inside the state of base for private use, so we can mix arrays with base 0 and base 1.

From 6.1 version an array can redim without loosing items.

```
1) Dim A(10)=0, a$(30)="",A%(10,10)=1
   Dim Base 1, A(), B(10,20)  \\ change only for these arrays
   Dim A(30)  \\ A(1) to A(30)  this is a redim. Interpreter knows A() base.
   Dim Base 0, N(10)  \\ N(0) to N(9)   \\ Base [0 or 1 but not expression]
   Dim Base 1 to N()  \\ change base to 1
   Print dimension(N(),0) \\ 1 return base
   Dim New N()  ' make a new one and shadow any old one
   (use New in a  For This { } or in Subs, where all new are for temporary use)
   A()=N()  \\ now A() change base because A() is a copy of N()
   Group Alfa {
        Dim Base 0, M(10)=1
        Module GetItem {
             Read .M(number)
        }
        Module PrintItem {
             Read subscription
             Print .M(subscription)
        }
   }
   Print Alfa.M(9)  \\ 1
   Alfa.GetItem 0, 30
   Alfa.PrintItem 0  \\ 30
```

2) Global A(10) ' this is a global array

3) Local A(10) ' this is a local array (in ver.8) It is optional because all new variables and arrays are local by default in modules ans functions. Using For This { } we can make local variables. But any module level arrays and variables are accessible unless we hide them making local with local instruction.

We can initialize any array in the DIM statement (optional). Names as A, A$ and A% allowed together.

We can pass arrays by reference through stack. Arrays live in the module or function where we create them. At the end of creation module or function those arrays are disposed. We can use arrays from other modules if we use global setting or by reference passing. We can redimension an array in a module other than the creation module. We have way to ask for dimensions and upper index limit. Arrays can change number of dimensions without loosing values (but we can have moving values, values that change indexes). We can use an array of strings as an array of numbers, and the reverse, when we assign a reference to the array.

```
4) DIM A(2,2), A$(20,5)
     A(0,0)=1,2,3,4
      \\Only for Sting Arrays
      \\ we can't read from stack in the following statement  (so A(3,0):=number can't work, because temporary the stack change with an empty one)
      A$(3,0) :="Name",1,2,3*100,"Name too"  'we put a record in an array (numbers converted to strings)
```

```
' We say that this is a module a

     ' array a() erased when module a end.
     set dim a(20)
     a(10)=34
     module that {
          print a(10)
     }
     that     ' we call module that and inside that a() has a global reference so it exist

     dim c%(10)    ' name of c% is a.c%()  but a. can omit in this module
     c%(2)=5.5  ' only integer part written to C%(2)

     module thattoo {
        ' a is name of parent module
        print a.c%(2)       ' we can use a.c% to find a "friend" array
        a.that  ' or a friend module
    }
    thattoo   ' we call thattoo
    a(10)=600
    module byref {
        read &two%(), &th()   ' Only Read statement can assign a name to a pointer
        print two%(2), th(10)
        list
    }
    ' we can reference  global a() and local c%() using &
    ' we send pointers to stack
    byref &c%(), &a()
```

```
' namespace
```

in a module a write

```
module b {
module c {
a=10
set b=20
list
}
c
}
b
```

we see from command list

```
C.A = 10,  B = 20
```

because any variable written as modulename.varname except global variables that not have modulename

```
' Let say that we have another module b
a=300
set b=1000
a
```

We call b and from b we call a that make a new module b that make a new module c and then show us the list of variables

```
B.A = 300, B = 20, C.A=10
```

Now we can insert a A=500 in module b in module a before we call c

we will see
B.A = 500, B = 20, C.A=10

because B.A. exist for the first B so exist for the second they have common namespace

In a gsb file we can place some global definitions for vars and arrays. (before the instruction to call first module, the starting module)

About names see ABOUT MODULES

```
dim a(10)
function a {
      read X
      =10*X
}
a(2)=5, 1000
print a(3), array("a",3), a(4), array("a",2)
print function("a",3), a(* 3)
```

in CLI

```
DIM A$(10)
```

in a module

```
 ' erase global array (without removing the name) from module
 ' no comments in a SET line
SET DIM A$(0)
 ' redim array (without removing the name)
SET DIM A$(20)
 'push a reference
SET PUSH &A$()
 'pop and create a new array name who reference the global array
READ &A$()  '  we can choose the same name...is a local name.
 ' so now we use global array as local
DOCUMENT A$(10)={1st line
second line.}  ' Only item 10 is a document var.
A$(10)={Hello Again
}
DIM A$(30 )   'redim without using SET
REPORT A$(10)
LIST   ' give a list of variables and arrays
```

```
\\ Version 9
Group N {
    dim base 1, x()
    value {
        =.X()
    }
    set {
            Print dimension(.x(),0)
            read .x()
            dim base 1 to .x()
            Print dimension(.x(),0)
        }
}
N=(1,2,3,4,5)
Print N
Print N.x(5)
```

identifier: DOCUMENT   [ΕΓΓΡΑΦΟ]

DOCUMENT list of string vars

Create or modify string vars to a list of paragraphs (a document). We can modify array items
For documents the = used to append text

CLEAR A$ used to clear paragraphs   ' without variable CLEAR clear all variables
We can use <= operator (this not create a new variable, as = do), only in strings not in arrays.

```
a$={HELLO
DUDE
}
DOCUMENT A$, B$
a$="TEST1"
b$="TEST2"
REPORT A$
REPORT B$
PRINT DOC.LEN(A$),DOC.LEN(B$)

' example
dim A$(10)
for i=0 to 2 {
    document A$(i)
}
for i=1 to 3 {
    A$(0)={0) 1111111111111111
    22222222222222222222222222222
    }
    A$(1)={1) 11111111111111111
    22222222222222222222222222222
    }
    A$(2)={2) 11111111111111111
    22222222222222222222222222222
    }
}
clear a$(2)   ' fast clear
a$(2)="ok"
for i =0 to 2 {
    report A$(i)
}
```

We can use <= to modify global vars (instead of =)
For global array items
write in CLI
DIM A$(20)
now write in a module
' we can't append comment in a SET line. We use SET to access a Level 0 or global array A$()

```
SET DOCUMENT A$(10)
A$(10)={1st line
second line.}  ' Only item 10 is a document var.
A$(10)={ Hello Again
}
REPORT A$(10)
```

we see now 2 lines (one paragraph) because first insert haven't a new line indicator (a line break).

```
1st line
second line. Hello Again
```

Another way to access a global array is to use reference

```
SET PUSH &A$()
READ &A$()   ' now A$() created as local and as a reference to global A$()
```

See Load.Doc and Edit.Doc

Each paragraph has own number. so when we delete or insert paragraphs, those paragraphs that are in document have same number as before. Check this program. Paragraph(a$,

order_no)=paragraph_number. Paragraph.index(a$, paragraph_number)=order_no.
There is no paragraph_number 0. This is a fake paragraph_number one before the start and after the end. So when we use Forward from paragraph number 0, we get the next one, any number non zero (means we have a paragraph) or zero (means end of paragraphs)

```
Form 60, 40
Document a$={aaaaaaa
bbbbbbbbb
ccccccc}
Insert to 2 a$={

    Hello

}
m=Paragraph(a$, 0)
flush
If Forward(a$,m) then {
    While m {
        data m
            Print Paragraph$(a$,(m))
    }
}
Print Part "done"
Print Under
k=1
While not empty {
    Read x
    Print x=paragraph(a$, k), x, paragraph.index(a$,x)=k, k
    k++
    Print Paragraph$(a$, x)
}
Print k=doc.par(a$)+1
\\ this is the right order of paragraphs
For i=1 to doc.par(a$) {
    overwrite  paragraph(a$,i),3 a$="    "
    insert to paragraph(a$,i)  a$=trim$(str$(i))
    Print paragraph$(a$, paragraph(a$,i))
}
```

identifier: ENUM   [ΑΠΑΡ]

Enumeration is local by default

We can use Set Enum to make a global one, but we can use it as local, and we can return enum values from functions.

```
\\ we can use any numeric type including double
\\ by default Dog=1, Cat=2
\\ we can change it: Dog=0, Cat   ' so now Cat=1
\\ we can change it: Dog=100, Cat=200
\\ we can put a new line in place of comma
\\ or after comma,
\\ we can't leave a comma as last character except new lines, in the enum block.
Enum Pets {Dog, Cat}
a=Dog
Print a=1  ' true
a++
Print Eval$(a)="Cat", a=2
k=Each(Pets)
While k {
      Print Eval$(k), Eval(k)
      a=Eval(k)
      Print a<Cat
      Alfa(a)
}
a=Dog
AlfaByRef(&a)
Print a=Cat

Sub Alfa(b as Pets)
      Print b
End Sub
Sub AlfaByRef(&b as Pets)
      Print b
      b++
End Sub
```

Look Enumeration

identifier: ENUMERATION   [ΑΠΑΡΙΘΜΗΣΗ]

```
\\ we can use any numeric type including double
\\ by default Dog=1, Cat=2
\\ we can change it: Dog=0, Cat   ' so now Cat=1
\\ we can change it: Dog=100, Cat=200
\\ we can put a new line in place of comma
\\ or after comma,
\\ we can't leave a comma as last character except new lines, in the enum block.
Enumeration Pets {
    Dog
    Cat
}
a=Dog
Print a=1  ' true
a++
Print Eval$(a)="Cat", a=2
k=Each(Pets)
While k {
    Print Eval$(k), Eval(k)
    a=Eval(k)
    Print a<Cat
    Alfa(a)
}
a=Dog
AlfaByRef(&a)
Print a=Cat

Sub Alfa(b as Pets)
    Print b
End Sub
Sub AlfaByRef(&b as Pets)
    Print b
    b++
End Sub
```

Look Enum

identifier: EVENT   [ΓΕΓΟΝΟΣ]

An Event is an object in M2000. We can push it to stack like a lambda function, and with Match("E") we can check if it is in the top of stack.
Events can be part of groups also.

We can make a global one:

```
Global Event A { }
```

or a local:

```
Event A {
    Read message$
    Read &message_by_reference$
    Function {
        Print message$
    }
}
```

We can use some keywords for handling events:

```
Event A Hold
Event A Clear
Event A Release
Event A New aaa() [, bbb()]   ' we can bind some functions for multicast purposes
Event A Drop aaa() [, bbb()] ' we can drop any function
```

Example 1:

```
\\ We use a Lambda which return another Lambda, we feed event c twice (three add and one drop) with the same lambda b
\\ We raise event by using Call Event, and then we pass as Event as a parameter to predefined Iterator,
\\ so we can use inside a lambda function (or any other) calls to functions (but functions are not global)
\\ We see later how we change functions, and how we hold it and release it

Iterator = Lambda -> {
    Read x,z
    =Lambda x, z->{
        Read e
        If x>=z Then Exit
        Call Event e x
        x++
        =True
    }
}

a=Iterator(10,15)
b=Lambda->{Print Number}
Event c {Read x}
\\ multicast
Event c New b(), b(), b()
Event c Drop b()
\\ FIRST USE
Call Event c 5
\\ SECOND USE
```

```
While a(c) {}
Event c Hold
Call Event c 5+2
\\ nothing happen but addition 5+2 processed
Call Event c 5+b(2)
\\ nothing happen but we see 2 because b() run
Event c Clear
\\ now we erase any function from Event and set to Hold
Function k {Print Number**2}
Event c New k(), b()
\\ reset Iterator
a=Iterator(10,15)
While a(c) {}
```

Example 2:

```
\\ Change Iterator. Now produce a lambda which need a reference to an Event.
Iterator = Lambda -> {
      Read x,z
      =Lambda x, z->{
            Read &e   \\ now we want by reference call
            If x>=z Then Exit
            Call Event e x
            x++
            =True
      }
}

a=Iterator(10,15)
b=Lambda->{Print Number}
Event c {Read x}
\\ multicast
Event c New b(), b()
Call Event c 5
Event c Hold
\\ passing by reference
While a(&c) {}
Event c Release
Call Event c 124
```

2) Events in groups with WithEvents

These are light events, without object or specific commands

```
Group WithEvents Alfa {
      Events "a", "b"
```

```
    X=10
    Module Check {
        Def M as long
        call event "a", &M, 20
        Print stack.size, M
     }
     Module CheckB(what$) {
        call event "b", what$
    }
}
M=100
a=500
b=4000
Function Alfa_a(New &a,b) {
    M++
    a=M
    Print "ok", a, b, M
    push 500  ' can't return using stack, stack is private
}
Function Alfa_b(a$) {
    Print "From Event:";a$
}
Alfa.Check
Stack ' print empty line, stack is empty
dim k(10)=alfa
Module Z (M()){
    For i=0 to 9{
            For M(i) {
                .check
            }
    }
    M(5).checkb "This is M(5)"
}
Z K()
Group WithEvents Beta=k(3)
Beta.checkb "Hello There"
```

The same example with event object
here we have to attach functions (and maybe more than one, so we have passing parameters to all functions in the event list), and also we have to specify the signature of parameters

```
Group Alfa {
    Event A {
        Read &a, b
    }
    Event B {
        Read a$
    }
    X=10
    Module Check {
        Def M as long
        call event .A, &M, 20
        Print stack.size, M
     }
     Module CheckB (what$) {
            call event .B, what$
    }
}
M=100
a=500
b=4000
Function Alfa_a(New &a, b) {
    M++
    a=M
    Print "ok", a, b, M
    push 500  ' can't return using stack, stack is private
}
Function Alfa_b(New zz$) {
    Print "From Event:";zz$
}

Event Alfa.A New Lazy$(&Alfa_a())
Event Alfa.B New Lazy$(&Alfa_b())
Alfa.Check
Stack ' print empty line, stack is empty
dim k(10)=alfa
Module Z (M()){
    For i=0 to 9{
            For M(i) {
                .check
            }
    }
    M(5).checkb "This is M(5)"
}
Z K()
Group WithEvents Beta=k(3)
```

```
Beta.checkb "Hello There"
```

identifier: GLOBAL   [ΓΕΝΙΚΗ]

```
GLOBAL  A=10, A(10,20)
```

We can make global variables and arrays. They stay alive until creator module or function or sub or inside a For This {} . We can assign new value bu using <= or using Let. If a global exist then a new command Global with same identifier make a new presence and hide old one (but not delete it). so never use Global in a loop. A global act as static if we never make a local in calling modules or and functions.

```
Global A=10, b()
A<=20
A+=10 \\ make  global 30
Let A=50
A=100  \\ now we have a local A and we hide global, for this module
for this { dim k(100)=1 : b()=k()}   \\ array k erased in for this but not global array b()
Print b(3)
```

We can also use a function

```
Function Feed { dim a(100)=10 : =a() }
b()=Feed()
Print b(50)
```

We can make global arrays without hide same named array by using SET DIM

```
SET DIM alfa(100)=2
```

See Local, Let, Dim

identifier: GROUP   [ΟΜΑΔΑ]

1) just a simple group, we can place a value for a variable (is optional) in group definition

```
     group alfa {a,b,c=4,d$,Long i}
     input "a:", alfa.a
```

2) A group with a function and a static variable

```
group beta {
     b=1, k
     name$
     function c {
          read a
          .b++   \\ or this.b++
         .k<=a   \\ use <= to change group's variable
          =a*.b  \\  or =a*this.b
     }
}
print beta.b
print beta.c(5), beta.b
print beta.c(5), beta.b
print beta.c(5), beta.b
     1
    10       2
    15       3
    20       4
```

3) Pass a group to a module...add this code to group beta from (2). So now we can pass not only variables, but functions too to a module.

```
module any {
     read &mygroup
     ? mygroup.c(5), mygroup.b
  }
any &beta
     25       5
```

4) A more complicated example. Introducing to local variables and functions. These functions are local to the module that we create the group. We can use this group in other modules but not the local variables, arrays or and functions and modules. Main scope for local definition is to do manipulations on data only for creator (module or function). We use this to address variables, functions and modules. But we can't use "this" for any local in any other call to a reference to group from any other module or function.
Here we use <= to asign a value without define a new variable (if we use this.many = a then we create a new this.many. So we use this.many<=a so interpreter search for it...in the group). Original name for many is delta.many

```
  group delta {
     dim ar(1)
     local many=1
     local module redim {
          read a
          if this.many>=a then exit
          this.many<=a
          dim this.ar(a)
     }
     Local function Integrity {
     try ok {
         if this.many=dimension(this.ar(),1) then {
             = true
          }  else {
             = false
          }
     }
     If not ok then =false
     }
 }
 delta.redim 10
 delta.ar(5)=10
 print delta.integrity()
 module testme {
     read &mDelta
     try ok {
     mDelta.redim 50
     }
     ' we can redim ar() because is not local using straight redim
     dim mDelta.ar(100)
 }
 testme &delta
 print delta.integrity()
 delta.many=100
 print delta.integrity()
```

5) Using prototype

```
T$={A,B, C$}
group a type t$
group b type t$
group c type t$
```

6) adding members and using group.count, member$() and member.type$()

```
t$={a, b, c$}
group a type t$
group a {
    a=100     'this in not added because exist
    d=20
    function alfa {
    =this.a*this.d
    }
    dim a(20)=20
    long k
}
for i=1 to group.count(a) {
```

```
print member$(a, i), @(20), member.type$(a, i)
}
print a.alfa()
```

7) There is two labels Private: and Public: for making some items private.

GENERAL INFO

We can make group of variables, arrays, events, lambdas, inventories and buffers and groups. Scope ot the group is to make a set of variables and relative functions and modules to be passed in a module by reference. Groups are not objects.
Groups are like objects but is a collection (we say a group) of variables that exist, in the execution state, in a module or function and when module or function ends the execution, those variables and the group are erased . Passing a group by reference to a module or function we pass all variables in group by reference. We can pass one variable of group by reference only too.
In any group we can add members, but we can delete them, they are deleted with all members, with group as we saw that before. A reference to a group can add members too but that members are not members of the original group. So the reference group to a group is an other object, not just a reference, although the variables, members, in the group are references. Members are variables in the list of the module or function. We can use LIST to display variables. We can look if a member exist using VALID() function.

SIMPLE EXAMPLE

```
group a_name {
    Long a, b, c
    Dim a%(20)=6
    a%(3)=10
    d$="Something"
    b$  ' no value
    function inc {
        this.a++
        =this.a
    }
}

print a_name.inc(), a_name.inc(), a_name.inc()
print a_name.d$
a_name.A%(4)=1,2,3,4,5
for i=1 to 10 {
```

```
print a_name.a%(i)
}
```

USING PROTOTYPE

The same example but we use a string as a definition of the group

```
prototype$ = {
    Long a, b, c
    Dim a%(20)=6
    a%(3)=10
    d$="Something"
    b$  ' no value yet
    function inc {
        this.a++
        =this.a
    }
}
group a_name type prototype$
group b_name type prototype$
group c_name type prototype$

print a_name.inc(), a_name.inc(), a_name.inc()
print a_name.d$
a_name.A%(4)=1,2,3,4,5
for i=1 to 10 {
    print a_name.a%(i), b_name.a%(i)
}


\* Here we can see how we can display the members of the group (except locals)
\* Also the use of THIS to read variables from group.
form 80,48
flush
group a {
    a=4
    group bb {
        c=34
        c1$="George"
        group dd {
            d=45
            e=13
            group chaos {
                a=1
                b=2
                c=3
                }
        }
        module other {
        print "Hello"
        print this.c1$, "YES"
        }
        function mine$ {
            read a$
            =a$ + " "+.c1$
        }
    }
b=5
c=6
dim a(20)=10
document aa$="ok"
}

function count_member {
read &a
read from a;
=stack.size
}
read from a,  a1,Group2 ,,, Group3, , ,Group4
list  \\ give a list of variables/arrays/objects
group Group3 {
    K=67
    L=54
}
push &Group3
read &all
print group.count(A),  count_member(&a)
print group.count(all), count_member(&all)
for i=1 to group.count(A) {
print member$(a,i), @(20),member.type$(a,i)
}
' These 4 variables are one (a.bb.dd.chaos.c)
group4.c=1000
print group3.chaos.c
print group2.dd.chaos.c
print a.bb.dd.chaos.c
print group3.k  , valid(a.bb.dd.k)
\* in Group3 we make 2 more members local
```

```
module TestMe {
      read &mygroup
      print mygroup.mine$("Hello")
      mygroup.other

}
a.bb.other
TestMe &Group2
```

****

Big Example (copy all lines until the end of text here to a module)

****

```
rem 1) use of private/public/for/read ?/valid()
group alfa {
private:
      x, y
public:
      \\ ? in read means optional
      module setXY { read ? .x, .y}
      function getX {=.x}
      function getY {=.y}
}
print valid(alfa.x)   \\ 0 false, we can't read alfa.x
alfa.setXY 10, 40
print alfa.getX()  \\ 10
for alfa {
      print .getX(), .getY()
}
alfa.setXY ,20
for alfa {
      print .getX(), .getY()
}
alfa.setXY 5
for alfa {
      print .getX(), .getY()
}

rem 2) use by reference
module CheckMe {
      \\ a new group prepared with all items reference to actual items
      \\ modules and functions are just copies to new group
      \\ special local items in actual item never referenced (we don't have here)
      read &byref
      for byref {
        print "old value X:";.getX()
        .setXY 100
        print "new value X:";.getX()
    }
}
CheckMe &alfa
Print "alfa value:";alfa.getX()
rem 3) use byref a function only from group
module CheckFunc {
    read what$, &func()
    Print what$;func()
}
CheckFunc "value for x is ", &alfa.getX()
CheckFunc "value for y is ", &alfa.getY()

rem 4) using weak reference
weakref$=weak$(alfa)
\\ we have to use function() to resolve weak reference for functions
print function(weakref$.getX())
module CheckMe2 {
    \\ always here read actual group
    read weakref$
    print "inside CheckMe2 ";function(weakref$.getX())
    weakref$.setXY 1,1
}
CheckMe2 &alfa    \\ &alfa is a weakref  and *read* inside module decide what to do
for alfa {
    print .getX(), .getY()
}

rem 5) we can add some item too.
group alfa {
    module printXY {
        print .x,.y
    }
}
alfa.printXY

rem 6) Overide a module, on a reference of a group
Module CheckReplace {
    read &rep
    group rep {
        module printXY {
            print ">>>", .x,.y
        }
    }
    rep.printXY
}
CheckReplace &alfa
alfa.printXY

rem 7) Find if a group is by reference or not
\\ using a group local variable
group alfa {
    local stamp
}
\\ and a global function using weak reference (is not a reference)
function global IsRef { read a$ : =not valid(eval(a$.stamp)) }
print IsRef(&alfa)   \\ 0 false is not reference
module CheckDeep {
    read &any
    print IsRef(&any)  \\ -1 is a reference
}
CheckDeep &alfa
rem 8) Make an array of same group
alfa.setXY 100, 200

dim MyAr(10)=alfa
\\ using =alfa, all modules and functions are common for all items
MyAr(0).printXY   \\ 100, 200
MyAr(2).setXY 2,4
MyAr(2).printXY
Print valid(MyAr(2).stamp)  \\ it is valid
For MyAr(2) {
    print IsRef(&this)   \\ no
    CheckDeep &this    \\ yes
}
\\ this is a copy
MyAr(0)=MyAr(2)
MyAr(0).printXY   \\ 2,4
rem 9) redim array preserving values, and instatiate new items
dim MyAr(20)
\\ if we use =alfa then all items restore to aa values
\\ so we use stock to sweep 10 items from item 10 (11th, 0 is the first item)
stock MyAr(10) sweep 10, alfa
MyAr(dimension(MyAr(), 1)-1).printXY   \\ 100, 200
MyAr(0).printXY  \\ 2, 4 preserved
For MyAr(1) {
    group this {
        module printXY {
            print ">>>", .x,.y
        }
    }
    this.printXY
}
\\ no change out of For obj { }
MyAr(1).printXY   \\ 0 to 9 use common modules/functions
For MyAr(19) {
    group this {
        module printXY {
            print ">>>", .x,.y
        }
    }
    this.printXY
}
\\ change happen for good
MyAr(19).printXY  \\ 11 to 19 use own modules/functions
\\ so now we can make a temporary array of alfa
\\ we do a copy one by one item
\\ and a copy back for all to MyAr()
\\ Ma() erased automatic after For This { }
for this {
    Dim Ma(20)=alfa
\\      for i=0 to 19 { Ma(i)=MyAr(i)}
\\      same as above
    stock ma(0) keep 20, MyAr(0)
    MyAr()=Ma()
}
\\ now modules are equal
MyAr(1).printXY
MyAr(19).printXY
For MyAr(19) {
    \\ temporary change
    group this {
        module printXY {
            print ">>>", .x,.y
        }
    }
    this.printXY
}
\\ no change happen, all items have common modules/functions.
MyAr(19).printXY
```

Example with operators for groups

```
Group alfa {
\\Private:
    x, y
\\Public:
    Module GetXY { Read  ? .X, .Y }
    Operator "++" {
        .X++
        .Y++
    }
    Operator "=" {
       Read N
       push .X=N.X and .Y=N.Y
    }
    Operator ">" {
       Read N
       push .X>N.X and .Y>N.Y
    }
    Operator "+=" {
        Read N
        .X+=N
        .Y+=N
    }
    Operator "+" {
         Read M
        .X+=M.X
        .Y+=M.Y
    }
    Operator "*" {
         Read M
        .X*=M.X
        .Y*=M.Y
    }
}
alfa.GetXY 100,300
Dim A(10)=alfa
a(1).GetXY 1,30
```

```
for a(1) {This+=5}
Print a(1).x
a(2)=a(1)
Print a(1).x,a(2).x
Print a(1).x*(a(1).x+a(2).x),$(3),"6*(6+6)"  \\ $(3) for right justfication
a(4)=a(1)*(a(1)+a(2))
a(5)=a(1)*(a(2)+a(1))
Print $(0), A(4)>A(5)  \\\ $(0) return to normal
Print A(4)=A(5)
Print A(4).x, A(5).x
print type$(a(2))
Print a(2).X, a(2).Y
a(2)+=100
Print a(2).X, a(2).Y
```

see example for operatrors in Class

identifier: INVENTORY   [ΚΑΤΑΣΤΑΣΗ]

inventory is an object with keys or keys and values.

```
\\ keys can be numeric or and strings
inventory alfa = 1,2,3,4:="value",5
print exist(alfa, 1)  \\ true
print exist(alfa, 100) \\ false
append alfa, 100
print exist(alfa, 100) \\ true
delete alfa, 3
print exist(alfa, 3) \\ false
print alfa(1) \\ 1 return key
return alfa, 1:=1000
print alfa(1) \\ 1000 return value
append alfa, "a string":="ok"
append alfa, "func":=lambda->{=number**2}
for i=0 to len(alfa)-1 {
    print format$("pos. {0} key {1}",i,eval$(alfa,i))
}
print alfa("func")(2)  \\ 4
for this {
    \\ temp deleted after "for this"
    group temp {X=10,Y=30}
```

```
     append alfa,3:=temp
}
print alfa(3).x   \\10
\\ alfa(3) means item with key 3 in inventory alfa
for alfa(3) {
     print .x, .y  \\10  30
}
alfa(3).x++
print alfa(3).x   \\11
dim a(30)=100
append alfa, "array":=a()
print alfa("array")(29)  \\ 100
print type$(alfa(3))  \\Group
print type$(alfa("array"))  \\ mArray  (object for arrays)
print type$(alfa("func"))  \\ lambda  (object for lambda function)
\\ we can copy an item from inventory
m=alfa("func")
print m(2) \\ 4
g=alfa(3)
print g.x  \\11
dim k()
k()=alfa("array")
print k(29)
\\ note the $ after name alfa
s$=alfa$("a string")
print s$   \\ ok
\\ this a two stage command
\\ first we call exist
\\ second we read absolute position
\\ positions changed if we delete something

if exist(alfa,"func") then k=eval(alfa!): print k  \\ 6
if valid(k) then print eval$(alfa,k)  \\ func

group beta {
     inventory m
     x=10
}
beta.m=alfa
dim a(100)=beta
a(0).x+=10
\\ property (variable) x is different for each group in array
print a(0).x, a(1).x  \\ 20, 10
print a(0).m("array")(0)  \\ 100
a(0).m("array")(0)+=1000
\\ inventory m is common for all groups in array and for beta
print a(0).m("array")(0)  \\ 1100
print a(1).m("array")(0)  \\ 1100
print beta.m("array")(0)  \\ 1100
dim copyA()
copyA()=a()
a(0).m("array")(0)+=1000
a(0).x=12345
print copyA(0).m("array")(0) \\2100
print copyA(0).x \\ 20  - no common
inventory AnEmptyOne
beta.m=AnEmptyOne
print len(beta.m) \\ 0 items
\\ only reference for beta.m changed
print len(copyA(0).m) \\ 9 items
```

```
\\ example 2
Flush
Class Alfa {
    Private:
        Inventory M
    Public:
        Module Alfa {
            Dim  A(20)
            Append .M, "North":=A(), "East":=A(), "South":=A(), "West":=A()
            Append .M, "North.Cursor":=0, "East.Cursor":=0, "South.Cursor":=0,
"West.Cursor":=0
        }
        Module PutValue {
            Read Where$, A%, V
            If Exist(.M, Where$) then {
                .M(Where$)(A%)=V
                Return .M, Where$+".Cursor":=A%
            }
        }
        Function ReadCurValue {
            Read Where$
            If Exist(.M, Where$) then {
                =.M(Where$)(.M(Where$+".Cursor"))
            }
        }
        Module TestMe {
            Print Dimension(.M("East"),1)
        }
}
B=Alfa()
B.PutValue "East", 2, 1233
Print B.ReadCurValue("East")
B.Testme

\\example3
Function B$ {
    Dim A$(10)="ok"
    =A$()
}
Inventory Alfa=1:=B$()
Print Alfa$(1)(3)
If Exist(Alfa,1) then {
    Print Dimension("Alfa$",1)
}
```

identifier: LAMBDA   [ΛΑΜΔΑ]

a=lambda variable list -> { block of a function body, using non local variable list }
a$=lambda$ variable list -> { block of a function body, using non local variable list }

```
a=lambda->100
B=100
a=lambda B ->B*Number
B-=50
Print  a(10)   \\ 1000
c=lambda B ->B*Number
Print  c(10)   \\ 500
d=lambda -> {
    read B
    =lambda B ->B**Number
}
m=d(3)
```

```
Print m(3)  \\ 27
```

we can pass array to variable list as a copy.
we can pass inventory and buffer and those always passed by reference

For recursion we can use Lambda() or Lambda$() inside defintion, because if we use the name then in a copy that name be invalid for a call.

```
Print lambda (a) -> {
    =a**2
} (3)  ' =9


makebold$ = lambda$ -> {
    Read fn$
    =lambda$ fn$ -> {
        ="<b>"+fn$()+"</b>"
     }
}
hello$=lambda$->"Hello World"

Print hello$()
hello$=makebold$(hello$)
Print hello$()
```

identifier: LET   [ΣΤΗ]

```
LET x=20, y=30, z$="ok"
```

Without LET, in an assignment, a left hand expression is always in priority, so for A(b+10)=a+10 array is searching, if found then evaluate b+10 and we get array item and then interpreter evaluate a+10. In a Let command first evaluate a+10 and then interpreter find array and then evaluate b+10. Also a Let command never hide a global if we give same name with "=". So we don't need here to use "<=" to make the assignment for global variable.

identifier: LOCAL   [ΤΟΠΙΚΗ]

1) Local make new variables always. In a block of code For This { } any new definitions lost after the end of block execution, so local variables with same name using Local used to shadow old values.

Example:

```
Let X=4, Y=5
```

```
For This {
     Local X=10, Y=30
     a()
}
a()
End
Sub a()
     Print X, Y
End Sub

\Example 2 - We can define arrays too
Local a(10)=1  \\ in group defintions Local is added to command, so there we use Local Dim a(10)=1
Print a(3)
```

2) In a group local do something else. Look example

```
\\ we don't need to use a Class function to make a Group. We can do this with Group command
\\ We can make Local arrays, modules and functions. This command is interpreted in Group definitions only
\\ A class definition is a group defintion, in a function which return a group.
Group Alfa {
     Local X=10
     Local Dim M(3)=1
     Z=3
}
Alfa.M(2)=1000
Alfa.X++
Print Alfa.X, Alfa.M(2)  \\ 11  1000
Push &Alfa   \\ send weak reference to stack
Read &Beta  \\ make Beta a reference to Alfa but excluding Local variables and arrays (or functions and modules)
Print Valid(Beta.X), Valid(Beta.M(2))  \\ local variables are not included in a reference
\\ but using a weak reference we can
A$=weak$(Alfa.X)
Print Eval(A$.)  \\ this is an evaluation of a weak reference - see dot after $
\\ using weak reference we ca use group like it is a global
Module Inside {
     Read A$
     Print format$("Inside {0}",Eval(A$.X))
}
Inside weak$(Alfa)
Theta=Alfa  \\ local variables included in a copy, but with start values
Print valid(Theta.X), Theta.X, Theta.M(2)
```

```
Dim A(10)
A(0)=Alfa \\ local variables are immutable in array items
Alfa.X+=20
Print Alfa.X   \\ 31
A(0).X+=20   \\ can change only in the opening of float group in A(0)
Print A(0).X   \\ 10
For A(0) {
    .X+=20
    Print .X  \\ 30 because A(0) is in opening state
    Inside &This  \\ we call Inside wih weak reference of this, which is the open group from A(0)
}  \\ now open group is closed to A(0), so X next time gets the default value 10
Print A(0).X  \\ every time we open A(0) X get the original value
```

identifier: LONG   [ΜΑΚΡΥΣ]

```
LONG a, b=12, c=34.5
```

c has value 34

These are Long type variables, with 4 bytes

normally a A is a double, but LONG change it to LONG type
Long range A value from -2,147,483,648 to 2,147,483,647

identifier: METHOD   [ΜΕΘΟΔΟΣ]

We can call methods for com objects (including user forms in M2000). which we define using Declare, and including document and inventory objects
Example using VbScript, and passing array

```
Global a()
mm=10
Module CallFromVb {
    \\ Number get first parameter is numeric else error
    Print Number
}
Module Global CallFromVbGlobal {
    Read X()
    X(0)++
    a()=X()
    ? "ok"
}
```

```
declare global vs "MSScriptControl.ScriptControl"
declare Alfa Module
Print Type$(Alfa)  \\ name is CallBack2
With vs, "Language","Vbscript", "AllowUI", true, "SitehWnd",  hwnd
Method vs, "Reset"
Method vs, "AddObject", "__global__",  Alfa, true
Method vs, "AddCode",{
    dim M(9), k   ' 0 to 9, so 10 items
    sub main()
        CallModule "CallFromVb", 1000
        M(0)=1000
        CallGlobal "CallFromVbGlobal", M
        ExecuteStatement "Print a(0)"
        k=me.Eval("a(0)")
        CallModule "CallFromVb", k
        ' use Let to assign a number to variable
        ExecuteStatement "let mm=12345"
        k=me.Eval("mm")
        CallModule "CallFromVb", k
        CallModule "CallFromVb", M(0)
    end sub
}
method vs, "run", "main"
Declare vs nothing
If error then print error$
Print Len(a())
Print a()
```

****************Example use Document (internal object)*************************

```
Document a$={péché
        sin
        peach
        pêche
        }

Method a$, "SetLocaleCompare", 1036
Sort ascending a$, 1,4
Report a$
Method a$, "SetLocaleCompare", 1033
Sort ascending a$, 1,4
Report a$
```

identifier: PROPERTY   [ΙΔΙΟΤΗΤΑ]

Properties are groups inside group which return value

1) Make a property in a group for read only from outside

name$ is read only, but has a private variable [name]$, and can change value

property item has no ++ or -- operator like variable X

```
Group Alfa {
    X=10
    Property name$  {value} ="John"
    Property item=100
    Module SetNewName {
        Read a$
        .[name]$=a$
    }
}
```

2) Make a property with a set member

Alpha has two members, a Value and a Set

We can't access easy parent's variables, we need to Link a parent variable/array to a new name

In the example an event from parent group linked to a new name in property member value

```
Group Something {
    Event C {
        Read What$
        Function {
            Print What$
        }
    }
    Property Alpha {
        Value {
            Link Parent C to C
            If Alpha>300 Then Call Event C, "High Value"
           \\ value is the output variable. So if we change value we change output without changing original value.
        }
        Set {
            If Value>400 Then Value=400
        }
    } = 10
 }
Something.Alpha=500
M=Something.Alpha  \\ Late Raise Event
Print M
```

3) A property is a group. we can make anything for this group

```
Group Something {
    Property Alpha {
        Value {
            .[count]++
        }
        Set
    } = 10
    Group Alpha {
       Property count {value}=0
    }

 }
Something.Alpha=500
For i=1 to 100 : N=Something.Alpha : Next i
Print Something.Alpha.count
```

4) Special care for properties with a name of a read only variable like Double (Double=7 by default)

```
Group AnyName {
    Property aNumber {
        Value
        Set {
          .[Double]<=value*2
        }
    }
    Group aNumber {
    \\ Because Double=7 as a constant for M2000
    \\ macro Propery use =Double so return constant 7
    \\ We can make property Double by hand
    Private:
        [Double]
    Public:
        Group Double {
            value {
                Link parent [double] to dbl
                =dbl
            }
        }
        Operator "+=" {
            Read N
            Link parent aNumber to a
            a=a+N
        }
        Function Root {
            Link parent aNumber to a
            =sqrt(a)
        }
    }
}

AnyName.aNumber=100
Print AnyName.aNumber
Print AnyName.aNumber.Double
Print AnyName.aNumber.Root()
AnyName.aNumber+=1000
Print AnyName.aNumber.Double
```

5) Properties with parameters

We can make value or set member of a property to take parameters. Here in the example we make an index as a parameter

Parameter list has to put in a list ( ) aftet value or and after set.

```
Class CheckProperty {
Private:
    Dim Base 1, A(3)
Public:
    \\ we make a property as readonly if we make only the value part
    \\ (parts are value and set, as functions)
    \\ using value  and a list of names (id1, id2,...)  we can make parameters for properties
    Property Name$ {value}="CheckProperty"
    \\ a property is a group and we can make properties too
    Group Alfa {
        Property Name$ {value}=".Alfa(index) - ReadOnly"
    }
    \\ Property make [Alfa] as private variable - no use for here
    Property Alfa {
        Value (index) {
            Link parent A() to A()
            Try Ok {
                value=A(index)
            }
            Flush Error
            If not Ok then Error "Index out of limits "
        }
    }
Class:
    Module CheckProperty {
        .A():=100,450,1890
    }
}

M=CheckProperty()
\\ Now M has a read only property Alfa() with one parameter
Print M.Alfa(1), M.Alfa(2),M.Alfa(3)
Print M.Alfa.Name$
Try Ok {
    Print M.Alfa(10)
}
Print "My name is:";M.Name$
If not Ok then Print "Error"+Error$
```

identifier: SET [ΘΕΣΕ]

```
SET A=10
SET A=23 : DIM K(20)=10 : PRINT "OK"
```

SET command send all text until the end of line to Command Line Interpreter (CLI). This interpreter is part of M2000 Interpreter that run from command line (from console).

See that (as example to understand the end of line):

```
INLINE "SET A=10 " : B=20
LIST
```

The B variable also became a global variable...because inline leave that:

```
SET A=10 : B=20
```

so SET get line "A=10 : B=20"

so we can write that

```
INLINE {SET A=10
} : B=20
LIST
```

So now we have a global and a local variable.

Because in CLI we can't make remarks, we can't place remarks in the same line that has a SET command

You can write a number label and maybe a command and after ":" we can place a Set command but without remarks

```
1000 print "ok" : set a=100
```

See example for FOR var=startValue to endValue step stepValue { code here }

here we define k as global (but erased after module or function ends running)

```
set k=50
for k=k to 10 {   'here we have a new k that read the only known k, the global k
     print k          'here interpreter choose local first
}
set print k     ' set command send all chars until end of line to CLI the level 0 or global for
executions manual commands
                    so at that level only the global variables are listed (without special name
reference). Look by using LIST var references
```

Any array and variable declaration can have Global scope. When the module that call the SET command ends then those global variables and arrays also erased from memory.

In a module A we make a variable A so the real name is A.A (module name + point + variable name)
A global variable has no module name and dot. In the right side in a assignment we read the local variable unless we create a global variable with same name. So global variables are like static but for all loaded modules, and have limited time to exist until the creator terminate action.

```
' Let say that before we run any module we create a
     a=10
'Then we write edit a  (we can have a as module name and as variable)
' we put this
     set push &a
     read &b
     b=100
```

We press Esc and we run this module

```
     a
```

Now we print a

```
     print a
     100
```

Second we read and declare b the same time as a reference to a (because only a READ command can join variables and arrays)
Why? Because the SET command send PUSH &a to CLI so in the stack (same as for Module a) a reference to variable "a" stay for a read command. Now we put 100 to b but the memory for b is the memory of global a. So when we leave from module a, only the reference to the global variable "a" lost, and the name b for that reference, but not the global a (that was declare before

the module run).

```
a++  can be used for global variables
a--
a+=100
a-=100
a*=2
a/=5
a-!
```

identifier: STATIC   [ΣΤΑΤΙΚΗ]

static variables in modules and functions, and threads
static variables are stored in a collection in execution object. So if we have a global module, and we call from different execution objects, then for each object we have different collection.
Subs can't have own static variables because run on execution object of module or function, but they can see static variables.
We can't have static in a thread/module with same name as local variable in module.
We can't have objects as static variables, except objects using pointers assign to variables.
We can't pass by reference static variables (except by value pointer to object)
static variables clear with clear command (clear every variable, including static variables)

Write this example in a file using Edit "st.gsb" press Esc and load it using Load st

```
Module A {
    if match("N") then Drop : Clear
    Module Global M {
        STATIC A=0, B=10, Z=(1,2,3,4,5)
        Z+=20
        Print Z
    }
    Module test1 {
        M
        M
        M
    }
    Print "three times call M from test1"
    test1
```

```
     Print "two times call M from this Module ";Module$
     M
     M
     Print "three times call M from test1"
     test1
}
Module B {
     K=100
     a 1  ' so now we clear but static variables are for b Module only
     Print K
}
A
B
```

More advance example:

```
Clear
Z=100
Module Localmodule {
     Print "This module can called from parent only"
}
Localmodule
Module Global test1 {
     Static M=10
     M++
     Print M
}
\\ this is not a ordinary function
\\ but a part of this module
Function fake (&feedback) {
     Z++
     feedback=Z
     test1
     Localmodule
}

Module Delta {
     Read &func()
     \\ we run a part of parent module
     N=0
     Call func(&N)
     Print N
```

```
    Call func(&N)
    Print N
    \ \  call test1 from Delta
    test1
    test1
}
test1
test1
\\ we pass a part of this module
Delta Lazy$(&fake())
test1
```

identifier: STOCK   [ΣΤΟΚ]

Any array can be used as stock for values

```
   1) STOCK arrayname(index)  in  value1, value2, value$...
         copy values from index
   2) STOCK arrayname(index)  out  Var1, Var2, Var$...
         copy to variables from index
3.1) STOCK arrayname(index)  keep 20,  arrayname(index+100)
        Relocate 20 items from inde
3.2) STOCK arrayname(index)  keep 20,  arrayname2(index+100)
         Store in other array from index to 100+index
4.1) STOCK arrayname(index)  sweep 100
         Make Empty all items
4.2) STOCK arrayname(index)  sweep 100,  class1()
         Put class1() from index to index+100-1
```

```
dim a(200), a$(3)
dim ov$(5)="OK"

stock a(0) in 1,2,"George", ov$()
stock a(0) out a,b, a$(1) , kl$()

kl$(1)=kl$(0)+"George"
print a,b,a$(1), kl$(1)
print
stock a(3) out kl2$()
print kl2$(1)
stock a(3) in kl$()
stock a(3) out kl2$()
print kl2$(1)
```

```
stock a(0) sweep 200
? type$(a(13))
stock a(0) sweep 200, "1234"
? type$(a(13)), a(13)   'but is string..."
stock a(0) in 1,2,3
? a(2)
stock a(0) keep 3, a(100)
? a(101)
```

identifier: SUPERCLASS   [ΥΠΕΡΚΛΑΣΗ]

A Superclass can bind to any object, but any object (group)can hold one reference for Superclass.
Because a group may have any number of groups inside, we can give superclass for each. See the second example

```
Superclass Alpha {
    Unique:
        x, y
        Dim A(20)
    Public:
    \\ k created for superclass and for object (two different things, two differnet values)
        k=10
        Module DisplayInfo {
            For Superclass, This {
                .k++
                \\ in one prin we can handle k from superclass and k from object.
                Print .k, ..k
                \\  .x and .y are unique here
                .x++
                Print .x, .y
                 \\ Here we print arrays (only from Superclass, because we deFine it in Unique
part)
                Print .A()
            }
        }
}
\\ there is no Alpha.DisplayInfo, or other value. Alpha is closed, only can create/merge other
objects (groups)
k=Alpha
k.DisplayInfo
Class Delta {
    Property ReadOnly {Value} = 1234321
    Module Delta (This) {}
}
m=Delta(Alpha)
m.DisplayInfo
Print m.ReadOnly
\\ here Array A() has a special holder for all items (are groups), for code only.
\\ because all Delta() get Alpha as SuperClass (merged to This in constructor)
\\ all have SuperClass common variables/arrays through a dedicated pointer
\\ so we can export this array, and return from this block (maybe a module)
\\ and Alpha erased, but not data if there are objects who hold pointer to them
Dim A(100)=Delta(Alpha)
A(4).DisplayInfo
```

This is the second example.

```
Superclass Kappa1 {
    x=10
    Module Display {
        For Superclass {
            .x++
            Print .x, This.x
        }
    }
}
Superclass Kappa2 {
    x=50
        Module Display {
        For Superclass {
            .x++
            Print .x, This.x
        }
    }
}
\\ we can create groups using Group command
Group Alfa {
    Group Beta {
        \\ no definitions yet we get from superclass
    }
}
\\ first we bind superclass for each group.
Alfa.Beta=Kappa2
Alfa=Kappa1
\\ now we have two superclass, one for outer group, and one for inner group
```

```
Alfa.Display           \\ 11 10
Alfa.Beta.Display   \\ 51 50

Z=Alfa

Z.Display                \\ 12 10
Z.Beta.Display       \\ 52 50
Alfa.Display           \\ 13 10
Alfa.Beta.Display   \\ 53 50

Dim A(10)
A(3)=Z
A(3).Display          \\ 14 10
A(3).Beta.Display  \\ 54 50

Inventory Inv1 = 100:=Z
\\ keys for inventories are strings, but we can handle like numbers if they are like numbers
Inv1("100").Display      \\ 15 10
Inv1(100).Beta.Display  \\ 55 50
```

identifier: SWAP   [ΑΛΛΑΞΕ]

```
SWAP numeric var, numeric var
SWAP numeric array item, numeric array item
SWAP string var, string var
SWAP string array item, string array item
```

Swap is 2 times and more faster for swapping string variables and 6 times more faster for numeric variables
I f a string is a Document (which is an object) then there is no problem to swap with a string that it isn't document.

Internal all arrays and variables are variant, so swap copy only the 16 byte per variant.
Document is an object with the reference in a variant. So a copy of that variant is a copy of the reference.

```
a=10
b=3
c=0
\* we set FAST !  with no refresh  - we can't put comments to a SET line command
set fast !
```

```
profiler   ' resetting the counter
for i=1 to 10000 {
      a=b
      b=c
      c=a
}
print timecount
profiler
for i=1 to 10000 {
      swap a, b
}
print timecount
\* restore to normal speed which we can control the Esc and Break buttons.
set fast
```

Another way to control the speed can be the use of REFRESH.

```
REFRESH 400
```

and at the end

```
REFRESH 25  ' normal
```

give a time close to SET FAST !

identifier: VAR   [ΜΕΤΑΒΛΗΤΗ_]

Look VARIABLE

identifier: VARIABLE   [ΜΕΤΑΒΛΗΤΗ]

```
VAR A, B, C
VARIABLE A, B, C
VARIABLES A, B=200, C
```

CREATE LOCAL VARIABLES LIKE LOCAL BUT IF EXIST ANY OF THEM AS LOCAL, SKIP THE CREATION PART AND PASS VALUE IF WE HAVE ONE OF SAME TYPE

identifier: VARIABLES   [ΜΕΤΑΒΛΗΤΕΣ]

LOOK VARIABLE

identifier: WITH   [ΜΕ.ΑΝΤΙΚΕΙΜΕΝΟ]

With statement is complex to describe full here
Can be used for input/output/ attach to a name/ attach to a name with parenthesis to place index
Here is a small example
We use two properties, ReadyState and RensponseText in two variables.
htmltext, aa and aa$ have no direct pointer to object, so if we pass to stack for return those, then after module exit, object get Nothing automatic, so anything in stack show nothing too.
We can pass them by reference to other modules or functions.

```
Declare htmltext "Microsoft.XMLHTTP"
testUrl$ = "http://httpbin.org/"
With  htmltext, "readyState" as aa
Method htmltext "open","GET", testUrl$, true
Method htmltext "send"
While aa<>4 {
     Print Over aa
     if mouse then exit
     wait 1
     refresh
}
Print
document bb$
if aa=4 then {
     With  htmltext, "responseText" as aa$
     bb$=aa$
}
Report bb$
Rem : Clipboard bb$
Declare htmltext Nothing
```

Using WithEvents to get events from a com object

```
\\ using WScript.Shell
Module UseShell {
     Declare  shell"WScript.Shell"
     With Shell, "Environment" set env ("PROCESS")
     With env, "item" as env.item$()
     Print "SYSTEMROOT="+env.item$("SYSTEMROOT")
}
UseShell
```

Using evants, we have to use functions called as subroutines, so we have to place NEW to read to new variables, and form version 9 we have to use & for by reference only to those parameters which passed by reference (the old model made them all by reference in a middle stage)

```
Module UsingEvents {
    Form 60, 32
    Cls 5, 0
    Pen 14
    Declare WithEvents sp "SAPI.SpVoice"
    That$="Rosetta Code is a programming chrestomathy site"
    margin=(width-Len(That$))/2
    EndStream=False
    \\ this function called as sub routine - same scope as Module
    \\ we can call it from event function too
    Function Localtxt {
        \\ move the cursor to middle line
        Cursor 0, height/2
        \\ using OVER the line erased with background color and then print text over
        \\ ordinary Print using transparent printing of text
        \\ $(0) set mode to non proportional text, @() move the cursor to sepecific position
        Print Over $(0),@(margin), That$
    }
    Call Local LocalTxt()
    Function sp_Word {
        Read New StreamNumber, StreamPosition, CharacterPosition, Length
        Call Local LocalTxt()
        Cursor 0, height/2
        Pen 15 {Print Part $(0), @(CharacterPosition+margin); Mid$(That$,
CharacterPosition+1, Length)}
        Refresh
    }
    Function sp_EndStream {
        Refresh
        EndStream=True
    }
    Const  SVEEndInputStream = 4
    Const  SVEWordBoundary = 32
    Const SVSFlagsAsync = 1&

    With sp, "EventInterests", SVEWordBoundary+SVEEndInputStream
    Method sp, "Speak", That$, SVSFlagsAsync
    While Not EndStream {Wait 10}
    Call Local LocalTxt()
}
UsingEvents
```

Another example using speech synthesis, and events to print words as the systme speak.

```
Module UsingEvents {
    Declare WithEvents sp "SAPI.SpVoice"
    That$="This is an example of speech synthesis."
    EndStream=False
    Function sp_Word {
        Read New StreamNumber, StreamPosition, CharacterPosition, Length
        Rem: Print StreamNumber, StreamPosition, CharacterPosition, Length
        Print Mid$(That$, CharacterPosition+1, Length);" ";
        Refresh
    }
    Function sp_EndStream {
        Print
        Refresh
        EndStream=True
    }
    Const  SVEStartInputStream = 2
    Const  SVEEndInputStream = 4
    Const  SVEVoiceChange = 8
    Const  SVEBookmark = 16
    Const  SVEWordBoundary = 32
    Const  SVEPhoneme = 64
    Const  SVESentenceBoundary = 128
    Const  SVEViseme = 256
    Const  SVEAudioLevel = 512
    Const  SVEPrivate = 32768
    Const  SVEAllEvents = 33790

    Const SVSFDefault = 0&
    Const SVSFlagsAsync = 1&
    Const SVSFPurgeBeforeSpeak=2&

    With sp, "EventInterests", SVEWordBoundary+SVEEndInputStream
    Method sp, "Speak", That$, SVSFlagsAsync
    While Not EndStream {Wait 10}
}
UsingEvents
```

Group: DOCUMENTS - ΕΓΓΡΑΦΑ

identifier: APPEND.DOC   [ΠΡΟΣΘΕΣΕ.ΕΓΓΡΑΦΟ]

APPEND.DOC DOCNAME$ [, DOCTYPE]
append a document to an exist file using default (Utf-8) or specific type.

In a module write this:

```
form 80,32
dim a$(4) : a$(0)="UTF-16LE","UTF-16BE","UTF-8","ANSI"
Print over $(6),"Example of saving and append text in four types"
for i=0 to 3
    clear testthis$
    document testthis$ = {1st line
                    2nd line
                    3rd line
                    }
    testthis$={4th line
                    }
    print under "Save one time and append one time more:"  : print
    \\ 10 chars left indent - we have to reset it after
    print @(10), : report testthis$ : print @(0),   \\ return cusror to far left
    save.doc  testthis$, "alfa1.txt", i     \\ we use user dir
    append.doc testthis$, "alfa1.txt", i
    print part format$({Length for file "{0}" in bytes {1} type {2}, total characters {3}}, "alfa1.txt", filelen("alfa1.txt"), a$(i), doc.len(testthis$)*2)
    \\     edit "alfa1.txt"
next i
print under
```

identifier: EDIT.DOC   [ΔΙΟΡΘΩΣΕ]

Edit.doc [code] a$, [position],[title$],[background_color]
open text editor in lower part of screen (using the split screen height). A **cls , 0** make split screen equal to screen so we edit in full height. Edit code, with code tag, make available the help for M2000 commands.

a$ must be a Document. We can upgrade any string and any array string item to Document by using Document command
using transparent background over background

```
hide
back 16 {
    cls
    pen #77eeff
    gradient 1,5
    document a$
    edit.doc a$,,,pen
}
show
```

identifier: FIND   [ΕΥΡΕΣΗ]

FIND
FIND document_varOrarrayItem$, string_to_find$ [, starting_position]
We have to search a srting in a paragraph, so in string_to_find$ no chars below 32 allowed

Return to stack one or three values. One if nothing find with value 0. Three if find, and top we have the position, next the paragraph order (or index, here is the same), and paragraph position. No case sensitive.

Example (text from BBC NEWS 17th August 2015)

```
document alfa$={Dozens of migrants on the Greek island of Kos have begun registering on a passenger ship which will be their temporary shelter as they seek asylum.
The first to board was a group of Syrian refugees, who have been living rough on Kos since they arrived.
}
report alfa$
find$="on "
find alfa$, find$
read find_pos
while find_pos>0 {
    read par_order, par_pos
    print under left$(paragraph$(alfa$, par_order, par_pos), 15)+"..."
    print under format$("Position {2}, Paragraph {0}, Position {1}", par_order, par_pos, find_pos)
    find alfa$, find$, find_pos+1
    read find_pos
}
```

identifier: INSERT   [ΠΑΡΕΜΒΟΛΗ]

INSERT

```
// for strings (or string array elements)
a$="123456"
Print Len(a$)
```

```
insert 3 a$="**"
Print a$, len(a$)  \\12**3456
a$="123456"
Print Len(a$)
insert 3,2 a$="**"
Print a$, len(a$)   \\12**56
a$="123456"
Print Len(a$)
insert 3,2 a$=""
Print a$, len(a$)    \\ 1256

// for strings using simple variables without $
a="123456"
Print Len(a)
insert 3 a="**"
Print a, len(a)  \\12**3456
a="123456"
Print Len(a)
insert 3,2 a="**"
Print a, len(a)   \\12**56
a="123456"
Print Len(a)
insert 3,2 a=""
Print a, len(a)    \\ 1256

// for arrays without $
Dim a(10) // work too as: Dim a(10) as string
a(2)="123456"
Print Len(a(2))
insert 3 a(2)="**"
Print a(2), len(a(2))  \\12**3456
a(2)="123456"
Print Len(a(2))
insert 3,2 a(2)="**"
Print a(2), len(a(2))   \\12**56
a(2)="123456"
Print Len(a(2))
insert 3,2 a(2)=""
Print a(2), len(a(2))    \\ 1256


//for Documents, just variables with $
Document a$, b$
```

```
a$={aaaaaaaaaaa
      12345678901234567890
      }
b$=a$
' 2 is the 2nd character
insert to 2, 10 a$={Hello
      two paragraphs
      }
Report b$
Report a$
Clear a$
Document a$=b$
' 2 is the order number of paragraph
' 10 is the 10th character in the paragraph
insert to 2, 10 a$="Hello"  ' we can use a third parameter to delete characters before the insertion.
Report b$
Report a$
Clear a$
a$=b$
Overwrite 2,10 a$="Hello"
Report b$
Report a$
Clear a$
a$=b$
Overwrite 2 a$="Hello"  \\ replace line
Report b$
Report a$

// Using Arrays (also need the $ character)
Dim a$(10), b$(10)
Document a$(3), b$(2)
a$(3)={aaaaaaaaaaa
      12345678901234567890
      }
b$(2)=a$(3)
' 2 is the 2nd character
insert to 2, 10 a$(3)={Hello
      two paragraphs
      }
Report b$(2)
Report a$(3)
Clear a$(3)
```

```
Document a$(3)=b$(2)
' 2 is the order number of paragraph
' 10 is the 10th character in the paragraph
insert to 2, 10 a$(3)="Hello"  ' we can use a third parameter to delete characters before the insertion.
Report b$(2)
Report a$(3)
Clear a$(3)
a$(3)=b$(2)
Overwrite 2,10 a$(3)="Hello"
Report b$(2)
Report a$(3)
Clear a$(3)
a$(3)=b$(2)
Overwrite 2 a$(3)="Hello"  \\ replace line
Report b$(2)
Report a$(3)
```

identifier: LOAD.DOC   [ΦΟΡΤΩΣΕ.ΕΓΓΡΑΦΟ]

LOAD.DOC document_name$, filename$ [, LocaleID]
Can open UTF-16LE, UTF-16BE, ANSI and UTF-8
Internal the text converted to UTF-16
Inside document a property used to hold the type of file
So when we do Save.Doc we use that type, or explicit we can use a type for that save only.
Text command save in temporary folder for logging purposes.

```
g$="George"
Text utf-8 alfa.txt { lines of text
                This is a variable "##g$##"
                second line
                }

Document a$
Load.doc a$, Temporary$+"alfa.txt"
Report a$
```

identifier: MERGE.DOC   [ΣΥΓΧΩΝΕΥΣΕ.ΕΓΓΡΑΦΟ]

MERGE.DOC document$, filename$
merge document from a file to document$

```
\\write something press esc
Edit "alfa1.txt"
```

```
\\write something press esc
Edit "alfa2.txt"
Document alfa$
Load.doc alfa$, "alfa1.txt"
Merge.doc alfa$,"alfa2.txt"
Report alfa$
```

identifier: OVERWRITE   [ΚΑΤΑΧΩΡΗΣΗ]

OVERWRITE

1) Use it with Piece$(), it is the reverse function, for strings and Documents

```
A$="AAA.BBB"
Overwrite A$, ".", 1:="Hello",5:="one more"
Print A$
Print type$(A$)
Dim K$(5)
K$(3)="AAA.BBB"
Overwrite K$(3), ".", 1:="Hello",5:="one more"
Print K$(3)
Print Type$(K$(3))
clear A$
Document A$="AAA.BBB"
Overwrite A$, ".", 1:="Hello",5:="one more"
Print A$
Print type$(A$)
Dim K$(5)=""
Document K$(3)="AAA.BBB"
Overwrite K$(3), ".", 1:="Hello",5:="one more"
Print K$(3)
Print Type$(K$(3))
```

1.1) Using string variables without $: We have to use & before the name. Because actually a reference is a String, so the numeric expression evaluator finds that no numeric expression start (but without & this evaluator "thinks" that has a numeric expression to consume). The overwrite using numbers before string used for Documents, see later (2)

```
string A="AAA.BBB"
Overwrite &A, ".", 1:="Hello",5:="one more"
Print A
Print type$(A)
Dim K(5)
K(3)="AAA.BBB"
```

```
Overwrite &K(3), ".", 1:="Hello",5:="one more"
Print K(3)
Print Type$(K(3))
```

```
2) Document a$, b$
a$={aaaaaaaaaaa
    12345678901234567890
    }
b$=a$
insert to 2, 10 a$={Hello
    two paragraphs
    }
Report b$
Report a$
Clear a$
Document a$=b$
insert to 2, 10 a$="Hello"
Report b$
Report a$
Clear a$
a$=b$
Overwrite 2,10 a$="Hello"
Report b$
Report a$
Clear a$
a$=b$
Overwrite 2 a$="Hello"  \\ replace line
Report b$
Report a$
```

identifier: SAVE.DOC   [ΣΩΣΕ.ΕΓΓΡΑΦΟ]

```
SAVE.DOC document$, filename$, type
//  convert to string, base of type loaded (using load.doc)
SAVE.DOC document$ as varname$
//  we can overrife the type by using our type
SAVE.DOC document$, type as varname$
```

type for save document (if document loaded form file then type is kept as number, the document internal format is UTF-16LE)

0 UTF-16LE   -4 UTF-16LE No BOM   10 lf   -10 lf NoBom, 20 cr -20 cr NoBom
1 UTF-16BE -1 NoBom   11 lf -11 lf NoBom 21 cr -21 cr NoBom
2 UTF-8 -2 NoBom   12 lf -11 lf NoBom 22 cr -22 cr NoBom
3 ANSI
1032 - ANSI Greek Locale

```
a=-12
Print " Encode "+if$(abs(a mod 10) mod 4+1->"UTF-16LE", "UTF-16BE", "UTF-8", "ANSI");
Print " | newline:"+if$(abs(a div 10) +1->"crlf", "lf", "cr");
Print " | BOM: "+if$(a<0->"No ", "")
```

```
Document α$={One paragraph
}

Save.Doc a$,"name1.txt"

Document b$={Another paragraph
}

If Exist("name1.txt") Then {
     Append.Doc b$,"name1.txt"
} Else {
     Save.Doc b$,"name1.txt"
}

Document k$
Load.Doc k$, "name1.txt"
Report k$
```

Because k$ is a document, an object (a com internal type) we can handle properties like ListLoadedType
This variable written when we read a file. So we can change it or simple read it.

```
With k$, "ListLoadedType", 10   ' lf   - write a new value
```

or

```
With k$, "ListLoadedType" as LLT
Print LLT  ' read
LLT=10  ' write
```

identifier: SORT(DOCUMENT)   [ΤΑΞΙΝΟΜΗΣΗ(ΕΓΓΡΑΦΟΥ)]

See SORT

identifier: WORDS   [ΛΕΞΕΙΣ]

WORDS document$

```
Document alfa$ ={ one, two, three
     and more, like this and that, or one and two
     }
Print Doc.words(alfa$)
M=Stack.size+1
Words alfa$
\\ default output all words ascending sort
\\ and duplicates
if Stack.Size >= M then {
Print Stack.Size - M
     For i=Stack.Size to M {
          Print letter$
     }
}
Print Doc.Unique.words(alfa$)
\\ now word change output
\\ all words ascending sort
\\ and a number at the right place if we have more than one occurance
M=Stack.size+1
Words alfa$
if Stack.Size >= M then {
     For i=Stack.Size to M {
          Print letter$
     }
}
```

Group: FILE OPERATIONS - ΧΕΙΡΙΣΜΟΣ ΑΡΧΕΙΩΝ

identifier: BITMAPS   [ΕΙΚΟΝΕΣ]

```
DIR "C:"
BITMAPS               ' SORTED BY DATE & TIME
same as  FILES "BMP|JPG|GIF|DIB|ICO|CUR|PNG|TIF"

BITMAPS !             ' SORTED BY NAME
FILES "bmp"          ' AS BITMAP
FILES  ! "bmp"       ' AS BITMAPS !
FILES  + "bmp"     ' SORTED BY DATE AND TIME  AND STORED AS MENU ITEMS
FILES  ! +  "bmp"  ' SORTED BY NAME AND STORED  AS MENU ITEMS
```
use MENU ! to open menu

identifier: CLOSE   [ΚΛΕΙΣΕ]

```
CLOSE variable_as_file_handle
CLOSE variable_as_file_handle1, ..., variable_as_file_handleN
```

CLOSE no parameter close all file and database connections
CLOSE BASE base_name$ [, base_name2$]
we can close connections by base name.

handles can be from 1 to 511
handles provided from OPEN statement, when we give a variable as file handler.
So we don't choose number

We can use # in front of a variable.
Close #f
or
Close f
is equal

identifier: DRAWINGS   [ΣΧΕΔΙΑ]

```
DIR "C:"
DRAWINGS
FILES "wmf|emf"
```

print on screen names of wmf files in directory DIR$

```
DRAWINGS               ' SORTED BY DATE & TIME
DRAWINGS !             ' SORTED BY NAME
FILES "wmf"          ' AS DRAWINGS
FILES "wmf"       ' AS DRAWINGS !
FILES "wmf"     ' SORTED BY DATE AND TIME  AND STORED AS MENU ITEMS
FILES +  "wmf"  ' SORTED BY NAME AND STORED  AS MENU ITEMS
```
use MENU ! to open menu

identifier: FILES   [ΑΡΧΕΙΑ]

FILES   print to screen all txt files in DIR$
FILES "mdb"   print to screen all mdb (databases) files in DIR$
FILES + "jpg"   fill list with file names with jpg extension without extension and path
MENU !   Open the list - we can add more selections by MENU + "name"

FILES "gsb", "While" print to screen all gsb files with "While" included.
FILES "gsb", "While|Black" print to screen all gsb files with "While" and "Black" included.
FILES !   ' sorted by name
FILES !! "*" ' sorted by type
FILES !! "T*.*" ' sorted py type

identifier: GET   [ΠΑΡΕ]

File handling look OPEN, CLOSE, PUT
GET variable_as_file_handle, variable_field$, record_position
GET variable_as_file_handle, array_field$(index), record_position

Example
Records are specific length strings of bytes, so you define record length when you OPEN the file.
' Read last record, records have no fields separators, and can be anything, not only text.

here=RECORDS(fhandler)
Get fhandler, buffer$, here

if we use WIDE for open the file then that file must be created as unicode by the same statement.

identifier: LINE INPUT   [ΓΡΑΜΜΗ ΕΙΣΑΓΩΓΗΣ]

LINE INPUT #variable_as_file_handle, variable_string$
LINE INPUT #variable_as_file_handle, variable_string$(index)

retrieve characters from file until find cr or cr+lf or cr+cr or lf+lf founded. That last chars aren't included. So if we want to place the string in a file we must add chr$(13) or chr$(13)+chr$(10) (or just PRINT # them)

Use function Eof(#variable_as_file_handle) to find if we get at the end of file.
Use SEEK #variable_as_file_handle, char_position if you have store a specific position...at the OUTPUT or at the APPEND.

Line break found as one CR or one LF or CRLF or LFCR

SEE INPUT AND INPUT$()

identifier: MOVIES [TAINIEΣ]

```
DIR "C:"
MOVIES
FILES "avi"
```

showing movies files in directory DIR$

use win dir$ to open an explorer and to put some avi files if you wish to use with your programs.

```
MOVIES              ' SORTED BY DATE & TIME
MOVIES !            ' SORTED BY NAME
FILES "avi"         ' AS MOVIES
FILES  ! "avi"      ' AS MOVIES !
FILES  + "avi"     ' SORTED BY DATE AND TIME  AND STORED AS MENU ITEMS
FILES  ! +  "avi"  ' SORTED BY NAME AND STORED  AS MENU ITEMS
```

use MENU ! to open menu

identifier: NAME [ONOMA]

```
NAME filename1 AS filename2
```

rename files in current directory DIR$

identifier: OPEN   [ANOIΞE]

This is the ANSI way (1 byte per char)
command can define variable_as_file_handle if not exist
EXCLUSIVE IS OPTIONAL
# IS OPTIONAL
OPEN file_name$ FOR INPUT EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR OUTPUT EXCLUSIVE AS #variable_as_file_handle
If we use file_name$ as an empty string For Output, then Print # send output to screen, for test purposes
OPEN file_name$ FOR APPEND EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR RANDOM EXCLUSIVE AS #variable_as_file_handle
LEN=total_length_for_a_row_in_chars

This is the UNICODE way (2 bytes per char)
command can define variable_as_file_handle if not exist
EXCLUSIVE IS OPTIONAL
# IS OPTIONAL
OPEN file_name$ FOR WIDE INPUT EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR WIDE OUTPUT EXCLUSIVE AS #variable_as_file_handle
If we use file_name$ as an empty string For Output, then Print # send output to screen, for test purposes

OPEN file_name$ FOR WIDE APPEND EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR WIDE RANDOM EXCLUSIVE AS #variable_as_file_handle
LEN=total_length_for_a_row_in_chars

Maximum LEN is 32766 (but for unicode use 32766/2 form max length of chars)
Always use an even number because system read as unicode the ansi file and then convert it to ANSI, so in a position for a unicode char we get two ansi chars.

We don't have BINARY files in M2000. You can use WIDE RANDOM as a form of binary file.
We can provide a new variable as file handle. Number as handle is given automatic. So if variable had a value before Open statement then that value after should be lost.

Functions
EOF(i)   ' true when we have end of file situation
RECORDS(i) ' return length of file as bytes for RANDOM the number of records
each record is a string of fix length. We can use MID$() to exrtract fields. Use FIELD and FIELD() for input and to create Fields.

records(i)+1 is the first free position to put a new record. You can't delete a record, but you can define a simple field of one char to mark to delete a record when you replicate the file.

Statements for RANDOM only
GET #i, rec$, pos   ' this pos is record number, not position in bytes.
PUT #i, rec$, pos   ' Seek command and Seek() function NOT work with RANDOM

For non RANDOM files:
These are text files or data files as text with data in each line.
Text File
position=SEEK(#filenumber) ' we can record the position before we write with PRINT or and WRITE
PRINT #filenumber, "aaaa"; ' Using to print text lines that we can read with LINE INPUT, use ; for adding without line break
PRINT #filenumber, "123",4,"567"; overwrite or at the end append 1234567
PRINT #filenumber, "A text line",chrcode$(10); \\ now we place linefeed as end of line (using chr$(10) is the same for ANSI)
WRITE #filenumber,"aaaa"   ' write string in double quotes and place a comma between fields. All " inside change to ""
(if we open for append we can change the position...and we can overwrite the file)

We can set the position from where we INPUT, LINE INPUT or INPUT$ (positions are in bytes always)
SEEK #filenumber, position
\\ Line Input find line break as CR (13) or LF (10) or CRLF or LFLF (LF is Line Feed, CR is Carriage Return)
LINE INPUT #i, line$ ' When we use WIDE then line breaks can be one of this CRLF LFLF CRCR CR LF
INPUT #filenumber, A$, B   ' read what Write write. Line breaks are throwing. So in A$ we can't put chr$(13) or chr$(10)
a$=INPUT$(#filenumber, 20) ' we can read number of chars. If we use WIDE in the OPEN command then chars are WCHAR (2 bytes)

Never forget to state WIDE for if you use WIDE before because no BOM used here.

If you wish more functionality you can use database with many tables, with sorting and memo fields (variable size fields).
look BASE for information.

We can use LINE INPUT for reading lines unicode or not (specified in the OPEN command by WIDE mark). LINE INPUT use line break as a CR or LF or both or CRCR or LFLF
We can use WRITE #filenumber to make comma delimited lines of text (don't format numbers

with comma as decimal)
A Write command bound a string with "". Works for unicode type of files too. We can place multiline strings. Also we can place chr(34) and automatic saved as "", and when we read from INPUT then this is change to ". INPUT$() and LINE INPUT never process that. Also PRINT # never change output if we use WIDE chars. (for ANSI the actually bytes from string converted to ansi).
An INPUT #filename command expects strings from a WRITE #filename command in "" and also works for unicode too
A PRINT #f command also work for unicode text. With no CR LF printed

```
MODULE A {open dir$+"uni4.txt" for wide output as k
a$={alfa "ok"
      beta
      }
write #k, 1,2,3,a$, 5
write #k, 100,2,3,"George", 5
close k
}
```

```
MODULE B {open dir$+"uni4.txt" for wide input as k
while not  eof(k)  {
input #k, a,b,c,d$,e
print "--------------------start---------------------"
print a,b,c
report d$
print e
print "----------------end-----------------------"
}
close k
}
```

Example for random access

```
\\ for repeat the example - close all files (perhaps we need it) and delete the file too
\\ we place a Sleep 300 to give os some time to do the delete
CLOSE
dos "del "+quote$( dir$+"rec.that"), 300;
\\ run once without wide, unhide the remark after next line, and hide next line.
open dir$+"rec.that" for wide random as f len = 20
\\open dir$+"rec.that" for random as f len = 20
Print Records(F)
a$=field$("ok",10)+field$("ΑΒΓΔ",10)
print a$
put #f,  a$, 1
```

```
Print Records(F)
a$=field$("ok2",10)+field$("ΑΒΓΔ2",10)
put #f,  a$, 2
b$=field$("",20)
get #f, b$, 2
Print format$("Read {0} chars",  LEN(b$))
print b$
Print Records(F)
close #f
```

identifier: PUT   [ΔΩΣΕ]

PUT #i, rec$, pos

rec$ must have length of Record as defined in OPEN type FOR RANDOM. Pos 2 is in 1*Lenght position in file, so pos is the record number starting from 1. Total records are given with RECORDS(i). You can append to a random file when you write to pos records(i)+1.

You may work with Put and instead save strings you can save buffers, but you have to use Append and seek to move the file cursor and the Input and seek to read (you can open two or more time a file, making a new cursor each time). If file not exist just open for output once with no data then close so you have the file to append next.

identifier: SEEK   [ΜΕΤΑΘΕΣΗ]

SEEK #L, 10
we position file cursor to 10th byte.
minimum 1 and maximum the maximum long number.

In any case for ansi or for wide (unicode) this is a position in bytes.

```
Report 2, "you make a temporary UTF-16 file"
text utf-16 this.txt {alfa
    beta
    gama
```

```
      }
L=1
open temporary$+"this.txt" for wide input as L
seek #L, 3  ' for UTF id number
while not eof(l) {
      print  seek(#L), @(tab+2),
      line input  #L, a$
      print a$
      }
close #L
```

identifier: SOUNDS   [HXOI]

```
DIR "C:"
SOUNDS
FILES "wav"
```

print on screen sound files as filenames

```
SOUNDS                ' SORTED BY DATE & TIME
SOUNDS !              ' SORTED BY NAME
FILES "wav"          ' AS SOUNDS
FILES  ! "wav"       ' AS SOUNDS !
FILES  + "wav"     ' SORTED BY DATE AND TIME  AND STORED AS MENU ITEMS
FILES  ! +  "wav"  ' SORTED BY NAME AND STORED  AS MENU ITEMS
```

use MENU ! to open menu

identifier: WRITE   [ΓΡΑΨΕ]

Using filename "" in Open for Output we send output to screen
we can change separator for fields and the character for decimal point
if we open file with "" as file name then we get Write output to screen

```
Write With  stringsep$, decimalse$ [, [JsonFlag], Nouseofchr34]
```

stringsep$ only one char used, if "" then we get the default ","
decimalse$ only one char used, if "" then we get the default "."
jsonflag if non zero means enconding string using \n and other characters

like json strings

so we can put mulrinlie text withoutbreaking the line

Nouseofchr34 a no zero value means no "" araound a string.

if Nouseofchr34 then null string from revision 37 version 12 works

so Write With "", "" reset the writing using Write

The same for Input #
We use Input With as Write With
We can define different set so we can read with one way and write with another.

We can use ANSI or UTF16LE files
For ANSI files we have to use LOCALE to set the locale for conversion
A Local 1032 used for greek conversion.

Write with chr$(9), ","
Write with ",", "."
There is a third parameter and if it is true then all strings have escape chars exactly as string$(a$) does
Write #i, A#, B, 12*3242*K, C$, D
Write formatted data to file with file number i using comma between values. If we open the file ..FOR WIDE OUTPUT then we send Unicode formatted data we have to read by opening FOR WIDE INPUT...see OPEN

You can write data in a data file with comma between fields and with quotes to bound strings (and we can use multiline strings also). This format can be readed from Microsoft Excel or similar. Use filetype CSV for your file. Don't use ; but only comma and not as last char. Each line c an have variable number of values. We have to make a same way as we make the file..to read after.
We can save SEEK(#i) positions and later in a FOR INPUT (or FOR WIDE INPUT) we refer to that. We make a big error if we made a WIDE file (with WCHAR) and we read without WIDE statement (like an Ansi file).

Use INPUT #i, Var List to read it. Writing and reading can't do for the same file in one time. So one time you have to write data adding one time or more and many other times you can read it. You read from the start unless you use SEEK to change position for reading.
To change data one way is by duplicating the file with a second name and at the end you can rename it with NAME statement. It is better to use database (see BASE) for altering a lot of records, or for sorting and for computing by using SQL). You can use simple random access files (for getting and putting back records) if no sorting or heavy computing needed.

Using an open file for APPEND we can move to position of writing with SEEK command anywhere so we can overwrite the file, but it is not recommended

From version 6.7 you can open for WIDE output and for WIDE input, means that you can use WCHAR , 16bit Unicode (as the way vb6 saves strings). So for that file type the WRITE send two bytes per char.
You can send Multilne Strings as strings variables, for the ANSI and for the UNICODE version. All chr$(34) inside string are changed to double chr$(34), and from the INPUT # we get the right string with one chr$(34).
So strings must contain text data. Binary data you can export by using OPEN... FOR WIDE OUTPUT and PRINT # with ; (but always we have even length of bytes).

look OPEN

Group: CONSOLE COMMANDS - ΕΝΤΟΛΕΣ ΚΟΝΣΟΛΑΣ

identifier: BACK [ΠΕΡΙΘΩΡΙΟ _]

Background layer is the layer that all other layer are top of it and moved together with it.
Background layer can be filled with DESKTOP command and can cutting by FORM command
We can move it by using MOTION.W and we can define it using size of font and width and height in twips with WINDOW command
We can make all window transparent using DESKTOP command, or a color of it.
We can redefine the size of font used in background as we send commands
We can use BACKGROUND or BACK as equal commands

1)

```
BACK {
    ' any command to screen going to background
    Gradient 1,5
}
```

2)

```
    BACK 20 {
    Print "20pt Chars"
}
```

identifier: BACKGROUND [ΠΕΡΙΘΩΡΙΟ]

see BACK

identifier: BOLD   [ΦΑΡΔΙΑ]

We can use BOLD for output to Screen / Layers, Background and Printer
1)
```
BOLD : PRINT "BOLD TYPE" : BOLD
```

2)
```
BOLD 1
' any print type as bold
BOLD 0
```

identifier: CHARSET   [ΧΑΡΑΚΤΗΡΕΣ]

We can define charset for LAYERS, BACKGROUND and PRINTER.
For greek
CHARSET 161
This is equal to GREEK command (from SETTINGS we can define the startup charset)
CHARSET 0
for default charset
PRINT CHARSET

We have another point that we can declare code page (like charset, but not the same values). That is for a Database, as we using the BASE command to make a new one. We can use UNICODE only in long binary fields (we declare that as OLE in the TABLE command). Using codepage we can sort the tables accordingly to that.

identifier: CLS   [ΟΘΟΝΗ]

Screen coordinate 0,0 is top left corner

1) CLS 1
clear screen, put blue color (0 to 15 are basic colors, negative numbers represent rgb values, use Color() function, or HSL())

2)CLS 1, 2
clear screen and set 3rd row to split screen top line
We can print anywhere but only the lower part of split screen scroll down (or up)..
SCROLL SPLIT number expression ' We can define the split screen without clear it
SCROLL UP and SCROLL DOWN to perform scrolling by text row.

3) CLS ,3

clear screen with last background color and put the 4th row as split screeen top line.
We can define scrolling part in Background, Screen and Layers (not on the printer)

identifier: CURSOR   [ΔΡΟΜΕΑΣ]

CURSOR 0, 0
PRINT "HELLO"
we set cursor to 0,0 (left top corner)
PRINT @(0,0);
PRINT @(0,0), "HELLO"   (1)

PRINT "AA", POS, TAB
Here POS is equal to TAB (each row can have equal space tabs)
Each comma (,) advance to next TAB position. except a coma after a print operator. In (1) the @() operator use for own the comma.
Two comma without parameter is a line break for Print.
Print "aa",,"bb",,"cc"
aa
bb
cc

identifier: DESKTOP   [ΕΠΙΦΑΝΕΙΑ]

DESKTOP
copy desktop to a deep background

DESKTOP 100
From 0 (100% transparent) to 255 (0% transparent) we can control transparency to major form of M2000 (with background and layers).

DESKTOP 255, 1
the second parameter is the 100% transparency when the color found. Here color is one from 16 basic.

DESKTOP IMAGE A$
A$ is a bitmap in a string or a filename to a bitmap (jpg is ok)

DESKTOP HIDE
We can hide the desktop so we can catch something behind it if we set transparency to color (2nf parameter).

DESKTOP CLEAR
We can clear the deep background, and set to non transparency condition.

Deep Background always have screen size and fit on screen. It is here to block clicks to go down. We can see what happen if we hide this background.
One good think to do we these commands is the transparent form. We can have 1 as backcolor in form so we can see behind all area that has color 1 (from desktop 255, 1) and we display an image that is like a water mark, always behind.

identifier: DOUBLE   [ΔΙΠΛΑ]

We can use DOUBLE for output to Screen / Layers, Background and Printer
We change text height to DOUBLE so we can print for some lines in double height and then with NORMAL we change to normal height
We can use switches in PRINT command or we can use REPORT command (multiline text formatting)

```
double
report 2, "center, double and proportional"
normal
print
```

identifier: FIELD   [ΠΕΔΙΟ]

Screen in M2000 Environment is used for many reason. One of them is to read character input. The basic console has rows and columns for characters. We can make input forms with fields that read non proportional text and one by one (to validate each stage of input).
We can prepare string variables with FIELD$() function
Each FIELD command exit by using up down arrows or ESC.

```
A$=FIELD$("123", 10)
FIELD x, y, lengthOfInput as A$
```

We get the reason for exit in a variable FIELD, as -1 up arrow, 1 down arrow or enter, 99 for Escape. We can validate the result and provide feedback with FIELD NEW (push 0 to FIELD read only variable)

Special Field is the Password field
FIELD PASSWORD 1,1, 10 AS PW$

we can't copy or cut from a FIELD , but we can paste.
In Field we can use INSERT keyboard key to change from insert text to overwrite.
If we ask for width of a string and we don't have in the screen then automatic open a dialog to get the input.

New numbers for exit:
-20 Home
20 End
-10 pageup
10 pagedown
1 Tab
-1 Shift Tab

identifier: FONT   [ΓΡΑΜΜΑΤΟΣΕΙΡΑ]

1) FONT "VERDANA"

We can change font in backgorund, screen, layers.
? FONTNAME$

We get the latest fontname$.
2) FONT LOAD "path\name.ttf" [, "path\name.ttf"]
We can use current dir, so we can givee only the filename.
this variant of FONT only load fonts, we have to choose using Font "fontname" where fontname is different from name.ttf
Using Choose.Font we get a dialog and pressing End we see the last item which is the new font from Font Load

3) FONT REMOVE "path\name.ttf" [, "path\name.ttf"]
We can remove one or more fonts if they not used. Using the same current directory if we didn't provide the path. All loaded fonts removed at the end of ececution of M2000 interpreter, so Font Remove is optional.

identifier: FORM [ΦΟΡΜΑ]

About screen
Except from WINDOW and MODE we have a way to found the size of font to use by declare the width in characters or width and height. Also we can show or hide the non character display area around the screen. This non using area is cutting from background. We say that area as Border (but is part of background layer)All the layers exist on background. All layers and background are movable and can alter size, font size, colors, can display images, text, and drawings. When background move...all other layers move together.

Also we can use Printer as a screen to (without refreshing). But for printer we can't use Form command (nothing happen) only the MODE command or by using PRINTER command we can define the size for font as the basic size to print.
M2000 Environment has a background and a screen on it, and 32 layers that can be using as screens or sprites (with transparent parts).

From 6.7 rev 21 we have line spacing (as twips using it as top and bottom border for each line). We can change using LINESPACE 0 or any positive value until 1000 twips. We can read that number using LINESPACE as read only variable.

We have 5 variants for FORM command

1) FORM 80
using the LINESPACE define a display mode with 80 chars width, and as many lines as it can. we can alter the linespace but see the 2 variant
This command can be used to printer (we can set only the char width automatic), and for layers
2) FORM 80, 25
If the background has height as screen then this command alter LINESPACE to use as many space form screen can be used
If not then maybe some part above and below screen are showing bacgkround
This can be used for Layers but without auto line spacing adjustment.
3) FORM
The border is cutting so now we know what variant 2 doing
4) FORM ;
The border is back again
5) FORM 80, 25;
Now we have a force creation of the text resolution we want. This is the best
Using ! after Form we make the internal algorithm to give more space around text frame (used as safe area for TV monitors)
Try Form ! 60,40 and Form 60,40 to see what happen

We can use the cutting process more than one time, so we can make three lines of text and we can move to top side of screen, so we can make applications for a part of the screen, using exactly the resolution we want (without we define a font size..that will be defined automatic).

Example
Make a column of 20 chars and 40 lines in the top left corner

```
form ;          '  pop  the border if is cutting
form 20,60  ' centered using maximum linespace (Automatic) to fill as possible the screen height
form            '  cutting the border
motion.w 0, 0  ' move background  to the left (so all the layers moved together)
form 20,40  ' we cut some lines - linespace not alter - 10 lines up and 10 lines down are cutting and showing the black background
motion 0,0  ' move form to top  - so now 20 lines are black lower the screen
form  ' cutting border , now 20 lines are gone...
```

So with one command and the variants we can make any screen type.

identifier: FORMLABEL   [ΕΤΙΚΕΤΑ.ΦΟΡΜΑΣ]

JUSTIFY: 0: LEFT, 1: RIGHT, 2: CENTER
FORMLABEL "LABEL", [FONTNAME$], [FONTSIZE], [JUSTIFY], [LETTERSPACE]

Example Using FORMLABEL to print exactly the same label as the PRINT
We use

```
WINDOW 8, WINDOW
FONT "LUCIDA CONSOLE"
MODE 22.5
PRINT FONTNAME$, MODE
CURSOR 10, 9
PRINT "ABCDE--------------"
CURSOR 10, 10
MOVE !
FORMLABEL "ABCDE--------------",,,,TWIPSX
CURSOR 10, 11
PRINT "ABCDE--------------"
REM:EXIT
CURSOR 10, 10
PEN 0 {
        PRINT "ABCDE-------------- PROOF"
}
```

identifier: FRAME [ΠΛΑΙΣΙΟ]

We can draw frames from the graphic cursor to any other point, using a background color and or a border color.

1) Frame widthInChars, heightInChars, color, border color
2) Frame @ cornerabsoluteposX, cornerabsoluteposY, type, 1stParameter, 2ndOptionalParameter
3) Frame @ (widthInChars), (heightInChars), type, 1stParameter, 2ndOptionalParameter

Look FILL to read type for read made constructions (using OS specific routines)

Example

```
--------------------
form 30,20
frame @ (30),(20),4,3
width 10 {
     cursor 1,1
     frame 13,8,2,5
     Print "Top Left"
     cursor 16,1
     frame 13,8,2,5
     Print "Top Right"
     cursor 1, 11
     frame 13,8,2,5
     Print "Bottom Left"
     }
cursor 16,11
'frame @ 13+pos,8+row-1,4,1   is the same as frame @ (13),(8),4,1
'frame @ 29,18,4,1   we can make the additions easy because we have in Cursor the right pos and row
frame @ (13),(8),4,1     ' so this form isn't compatible with old versions but is easy to follow as is the same as the first form of FRAME
pen 0
Print "3D frame"
pen 14
```

identifier: GRADIENT [ΦΟΝΤΟ]

GRADIENT 1 ' fill the output with one color (CLS clear the split screen...but GRADIENT paint entire screen)

GRADIENT 1,2 ' fill from color 1 to color 2 (we can use 0 to 16 standard colors or Color() function (RGB model) or HSL() function (convert to rgb), or system colors using 0x prefix)

GRADIENT 1,3,0 ' from left to right

Print and Report command type letters without clear first the place to type. Only in a Field input and the Query of command line (using the same routine as FIELD) clear always the background (the FIELD command clear from start, the "Query" clear as we type).
For this reason if we want to print in a cursor location many times a value then we have to use FRAME command or the print operator @() which can alter the background of a frame with color or image.

Example

```
-------------------
Module A {
     Mode 12, 8000
     Background {
          Desktop
          path ! {Gradient 1,2}
     }
}
```

identifier: GREEK   [ΕΛΛΗΝΙΚΑ]

Change charset to Greek. Also change the language of error messages to Greek language

See LATIN and CHARSET

identifier: HIDE   [ΣΒΗΣΕ]

HIDE
We hide the screen (not the background). We can hide all (including the background) if we minimized from the task bar - we can control the title in task bar and if we want to show or hide, and so if we can minimized or close the application from there.

Layers also can be hide using the PLAYER command (layers are also players, forms that act like hardware sprites). Screen can have 32 layers. Screen and Layers also can move and change dimensions.

See SHOW, LAYER, PLAYER

identifier: HOLD   [ΚΡΑΤΗΣΕ]

We can hold the screen output (only the main screen, not layers, no the background), and we can alter the screen and then we can release again. Graphic cursors, colors (pen and background) can't hold. We can do nice things with moving things on screen. We can release (with command RELEASE) put some sprites with SPRITE command and then use the REFRESH 0 to refresh display. That is a small sprite engine with software sprites. You can use "hardware" sprites, using PLAYER command. You can use both without problem (sprites

HOLD

(There is a HOLD mark for Threads as part of other commands)

identifier: ICON   [ΕΙΚΟΝΙΔΙΟ]

We can change the icon for the title in title bar (see TITLE)
ICON "file.ico"

identifier: ITALIC   [ΠΛΑΓΙΑ]

ITALIC : PRINT "THIS IS LETTERS" : ITALIC

or we can use
ITALIC 1   ' on
ITALIC 0   ' off

identifier: LATIN   [ΛΑΤΙΝΙΚΑ]

Change charset to Latin. Also change the language of error messages to English language

See Greek and CHARSET

identifier: LAYER   [ΕΠΙΠΕΔΟ]

Screen can have 32 layers. The 32n is at the top..
We can create by using an image or we can create with the first LAYER command and the size would be the size of the layer from where the LAYER command start to execute. Size of font must selected by using MODE size or FORM to make an automatic size match to a specific char resolution. We can use the WINDOW command to give size and dimensions
If we run this when the output is to screen the layer take that size

```
LAYER 1 {
    Mode 16
    print "this is 16pt letters  ' this command output to layer 1 but we can't see yet
}
PLAYER 1, 3000,4000  ' now we use player command to set the position
PLAYER 1 SHOW
' OR
LAYER 1 {
    window 6, 10000,8000  ' we set a size or without it the size from the calling layer
    form 32, 20  ' this do a cut bottom and left to display 32 chars X20 lines
    print "this has resolution 32x20 chars  ' this command output to layer 1 but we can't see yet
}
PLAYER 1, 3000,4000  ' now we use player command to set the position
PLAYER 1 SHOW
' OR
MOVE 0,0
A$=""
COPY 10000, 8000 TO A$
PLAYER 1, 3000, 4000 USE A$   ' NOW LAYER HAS A PICTURE BACKGROUND
LAYER 1 {
    form 32, 20  ' this do a cut bottom and left to display 32 chars X20 lines
    CLS ' WE REPAINT THE PICTURE
    print "this has resolution 32x20 chars  ' this command output to layer 1 but we can't see yet
}
PLAYER 1 SHOW   ' NEEDED ONE  TIME TO SHOW THE LAYER
```

We can make TARGETS in the layers see TARGET command

Anytime we can move or hide and then show again the layer. We can draw and we can open FIELDS for input. We can used with transparent pixels also (but the transparent mask changed only if we change the image we place with SPRITE command/
We can make threads to move the layers, to show as info in some time after. Use AFTER command in combination to EVERY or WAIT to see the effect.

identifier: LEGEND   [ΕΠΙΓΡΑΦΗ]

We can print in screen, layer, background and printer with many ways using PRINT and REPORT but also we can use LEGEND to print ANGLED text and with specific font for each call. We can include it in a WIDTH { } structure to alter the "outline" width of the text. We can also print multine text.
Always the print starts from graphic cursor (not the text cursor)

we can prepare multiline text using { }

```
a$={1st line
     second line
     third line
     }   ' this bracket mark the space to exclude before writing to a$
```

1) Print a legend with a specific font and size
LEGEND a$, "TAHOMA", 22

2) Like (1) with angle in rads (use pi/2 for 90 deg)
LEGEND a$, "TAHOMA", 22, pi*2/3
3) Like (2) but we can format the text, center , right or left
LEGEND a$, "TAHOMA", 22, pi*2/3, 2
1 for right, 2 for center, 3 for left
' no text wrapping like REPORT does
4) like 3 but we can change quality
LEGEND a$, "TAHOMA", 22, pi*2/3, 2, 0
0 or non 0. With 0 quality no anti-alliasing perform so we can take it as sprite and make transparent without small glitches
5) LEGEND a$, "TAHOMA", 22, pi*2/3, 2, 0, 150
like 4 but we add an extra space in each char. Negative number used for exclude space (from version 8, rev 42)
Look example at the end

This is a special variant used for display multiline text. This isn't use graphic cursor but in chars cursor
We give a frame as a width and a height and the command show text in a$ without scrolling
6) LEGEND ! text$, width_in_chars, height_in_char_rows
We need this after we perform a text input with INPUT ! command

Legend use the Bold and Italic flag for current layer. In this example we alter these and set again in the previous state, using two variables to store them. Also Legend use pen color, line width, and color { } to fill color inside text area.

Example

```
cls #ffbb77,0
pen #7700ff
anti_aliasing = true
center = 2
move 6000,6000
oldBold = Bold
oldItalic = Italic
Bold 1 : Italic 1
pen #77aaff {
    width 3 {
        color #775511 {
                LEGEND  {Hello
                My name is George
                }, "Arial Black", 40, pi/8, center, anti_aliasing,-60  ' 60 twips are 4 pixels in 15 dpi
screens or 5 pixels in 12 dpi
        }
    }
}
Italic oldItalic : Bold oldBold
move 12000, 6000
LEGEND  {Hello
My name is George}, "Arial Black", 40, pi/8, center
```

identifier: LINESPACE   [ΔΙΑΣΤΙΧΟ]

```
Print Linespace
Linespace 120  ' twips, alter in 2 pixels so 30 twips for twipsX=15
Form 80
```

identifier: LOCALE   [ΤΟΠΙΚΟ]

```
Locale localeID
Local ";\Y\e\s;\N\o"   ' set boolean values
Local "" ' reset boolean values to original values.
```

this is a setting for some functions, not for the screen or file or the system locale.
chr$() can give us the unicode code from any ANSI code based on locale.

We can set a global Locale ID or we can use in place: Print chr$(249,1037)

This global setting affect
All of this functions return strings
These return the unicode representation
    CHR$(ansi number) , CHR$(ansistring, 0), Ucase$(ansistring,0), Lcase$(ansistring, 0)
This make an ansistring (one ANSI char per 2 bytes)
    STR$(string to convert, 0)
We can convert to 1 byte char (so in any 2 bytes we have 2 ansi chars) by using STR$(string) and we do the reverse by using CHR$(string).
But these commands work with system locale. So we need to make first an ansistring and then we can compact it to a real ansi string with str$(). So this A$=Str$(Str$("Ελληνικά", 1032)) make an Ansi string that we can do the reverse in any system locale by using this:
? chr$(chr$(A$), 1032)
the inner CHR$() make the 4 chars 8 chars with chars with values below 256. The outer take the ansi code and replace with the UTF16LE code (as used by Windows).
If we want to feed an open file using an other locale, say 1049, and we wish to use the ansi version of OPEN command, we can prepare each string with STR$(string2put, 1049) as export$, and the system do a hidden Str$(export$). When we read from file a hidden Chr$(import$) give as a value and we have to do a CHR$(export$,1049) to get the right string.
If we wish we can use WIDE in the OPEN command and then we can write and read mix chars from any set, as Unicode UTF-16LE.
This is an example of how to print a charset from a locale

```
locale 1055
for i=33 to 255 {
print chr$(i);
}
print

' or we can use directly the locale id
for i=33 to 255 {
print chr$(i,1037);
}
print
```

identifier: MARK   [ΣΗΜΑΔΙ]

We can display a mark like a circle or ellipse.

MARK widthInChars, HeightInChars, Color, circleOrellipse

With last parameter 1 we print circle always, but with 0 maybe we print circle if width and height are equal.
Cursor isn't changed so we can use it also as a background style,

identifier: MODE   [ΤΥΠΟΣ]

```
 1) MODE sizeofFont, screenwidth, screenheight
 2) MODE sizeofFont, screenwidth
 3) MODE sizeofFont
```

Use scale.x and scale.y to read the screen size (also you can read printer, background and layers dimensions using proper structure, see BACKGROUND, PRINTER and LAYER)
You can use FORM to automatic adjust char resolution. Only type 3 can be used in Layers (the other must be used with WINDOW command.
As we see there are 3 commands for alter the screen.
MODE ' change font size, and in screen set the screen inside background
WINDOW ' change background font size and window size. For layers also change the size, and the font size
FORM ' this is a command that alter automatic the font size to the resolution we choose.

Mode suspend the TARGETS system (see TARGET), but not erase them. Use TARGETS NEW or START to clear TARGETS.

identifier: MOTION   [ΚΙΝΗΣΗ]

POSX and POSY are given by us

```
MOTION CENTER  ' we center the screen (based on the background)
MOTION POSX,POSY ' we move the screen using top left corner
MOTION POSX ' moving horizontal
MOTION ,POSY ' moving vertical
```

We can read the top and left properties as motion.x and motion.y

identifier: MOTION.W   [ΚΙΝΗΣΗ.Π]

We can move the Window (not the layers, we can move them from PLAYER command)

```
MOTION.W;    ' center window
MOTION.W POSX,POSY  ' we place in specific left and top values
MOTION.W POSX  ' Alter only X coordinate
MOTION.W ,POSY ' Alter only the Y coordinate.
```

Maybe we must call DESKTOP (if we call it before)

identifier: NORMAL   [KANONIKA]

revert the command DOUBLE

Now each row in console occupy one row.

identifier: PEN   [ΠΕΝΑ]

Pen colorindex
colorindex can be:
1) QBasic Colors: 0 to 15
2) Windows Colors: &H8000000F& in vb6 ...must convert to 0x8000000F for M2000
3) 0 to -16777215 (negative numbers) for RGB values

We can use function Color(redvalue, greenvalue, bluevalue), or we can open color selector with Choose.Color with parameter or not, and get the color from stack.
Also we can read the color value (as negative number) from a point in screen, with POINT, so MOVE 3000,3000 : A=POINT
The color selector get hex value or we can scroll up/down to find a color with the help of the rotating plane (b g r, g r b, r b g) who can change by clicking on headline "B | G | R".

We can Print Pen so we get the negative value or if matched with one of QBasic color, then that positive number return.

identifier: REFRESH   [ΑΝΑΝΕΩΣΗ]

Screen Refresh and Do some Events helper function
1) Refresh
Refresh the output (not for printers)

2) Refresh 0
Reset internal refresh counter

3) Refresh 100
(also reset the counter(
Sets the time period to perform auto refresh. By default has 40 as value. This no a delay. When we have a request for refresh then if the counter is above our limit a refresh happen and clear the refresh counter.

Auto refresh can be handled from SET SLOW or SET FAST ans SET FAST !

For delay use WAIT not Refresh. So to perform an AFTER thread we have to perform some WAIT or EVERY or a MAIN.TASK structure and not Refresh. But in a thread for synchronization we have to perform screen refresh after all of our drawings to screen. A thread is like a WAIT if nothing doing inside.

Here is an example of how we can use software sprites (not PLAYER), which they have transparency for all the image and for a bit only. We use HOLD and RELEASE so we don't have to draw non changing objects. We use a EVERY flow control and we hold the refresh by resetting the internal counter and perform the refresh when we like to do (after the drawing of sprites). A sprite command

```
REFRESH  500
CLS 1,0   ' COLOR BLUE AND  RESET SPLIT SCREEN SETTING TO TOP LINE
PRINT "This is first line"    ' WE CANT SEE THE FOUR PRINTS
PRINT "This is second line"   'BECAUSE WE SET A HIGH REFRESH RATE
PRINT "This is third line"
PRINT "This is forth line"
MOVE 0,0
A$=""
COPY 3000,2000 TO A$
GRADIENT 3, 5   ' CLEAR SCREEN WITH GRADIENT
PRINT @(0,0),"SPACE BAR TO EXIT"
COPY 6000,6000 TOP A$, 10, 200
COPY 6000,6000 TOP A$, 45
COPY 2000,2000 USE A$, 45
HOLD  ' save screen
MOVE 6000,6000
I=1
X=MOUSE.X
Y=MOUSE.Y
EVERY 20 {
    I++
    IF INKEY$<>"" THEN EXIT
    ' YOU CAN MAKE THE SECOND ROTATED OBJECT TO HAVE FLICKERING
    ' IF YOU DO THAT:
    ' COMMENT THE LINE BELOW AND PUT REFRESH TO 10 IN THE FIRST LINE
    REFRESH 0   ' RESET THE REFRESH COUNTER
    RELEASE
    PRINT @(0,1),STR$(NOW,"HH:MM:SS")
    MOVE 6000,6000
    SPRITE A$, -1, I
    IF MOUSE=1 THEN {
    X=MOUSE.X
    Y=MOUSE.Y
    }
    MOVE X, Y
    SPRITE A$, 1, I, 200, 80
    REFRESH   ' DO NOW A SCREEN REFRESH ONLY
}
```

identifier: RELEASE   [ΑΦΗΣΕ]

RELEASE
release the screen which we hold with HOLD command.

identifier: REPORT   [ΑΝΑΦΟΡΑ]

REPORT is the command to format multiline text and make columns of text like in a page maker. Not only use word wrapping but also use spaces with variable length. It is good for reports, to the screen and to the printer.

```
Report!4  ' make tab width 4 characters
REPORT "we can use simple lines or multilines"
REPORT {This is a multilne text
    this is the second
    }  ' spaces are controlled from the position of this bracket so this print
This is a multilne text
this is the second
```

Report start from POS (cursor X position) and in every line keep that POS for left space indent.

```
Print "                    ";   ' we put some space to the left for each line - each line maybe is a paragraph with many lines when we print
REPORT {This is a multilne text
      this is the second
      }
```

So now is time to change the way we report, and some variants

```
REPORT typeOfreport, A$
REPORT typeOfreport, A$, WidthSeeTheWay
REPORT typeOfreport, A$, WidthSeeTheWay, Lines2Display
REPORT typeOfreport, A$, WidthSeeTheWay, Lines2Display LINE startLine
```

new variants

```
      REPORT typeOfreport, A$, WidthSeeTheWay, Lines2Display AS VAR$  [VAR$()]
      REPORT typeOfreport, A$, WidthSeeTheWay, Lines2Display LINE startLine AS VAR$ [VAR$()]
```

(for output each line to string, we can use also a starting line)

typeOfreport:
0 Format left and right
1 Format Right
2 Format Center
3 Format left

There is a way before we print to find how many lines needed to print
See the example here:

```
c$={s
d
s.
.
..
. .... ....
}
for i=1 to 20 {
cls
cursor 10,10
print "*1234567890";
report 2, c$,i,-1000  ' NO PRINT ONLY TO GET REPORTLINES
cursor 10,11
print "*";
k= REPORTLINES
print @(pos,row, pos+i,row+k, 5,8);
report 2, c$,i,k
print i , k
k$=key$
}
```

Using negative number for lines2Display (the absolute number indicate the maximum limit) we get in REPORTLINES the lines of text actually we can print. So after that we can set the start line and the lines to print (on the screen or on the printer).

When we report to the screen there is a system to hold printing after 2/3 of screen height that are printed. We press space or click it with mouse to continue. This system not engaged if we report buy using part of text.
As we can see Report use cursor to specify left margin and use width in twips to specify the printing width IF the number is larger than the width in chars. So we can put the width in chars or in twips and the command can understand what we want...

Formatting Proportional and Not Proportional text we can print with PRINT command, but not with wrapping text. In a print command we can use the tab system (equal spaced). See the example (we use operator $() to alter the Formatting and the 8 is the tab distance, or width.

```
    FORM 48,32
    FOR I=0 TO 8 {
    PRINT $(I, 8),145355,34.23432,34.222
    }
```

identifier: SCROLL [ΚΥΛΙΣΗ]

```
SCROLL UP
SCROLL DOWN
SCROLL SPLIT 5
```

CLS ,5 'same except the CLS clear the bottom scrolling part of screen.

We can define a layer and scroll it by moving, but here the scrolling is text only, and for rows.

When we REPORT a multiline text, or in MODULES or FILES or same type commands that print multiple lines of text , there is a stop scrolling auto function that wait us to press a key or click the screen to scroll more lines.

Multiline text can assign by using {}
so

```
A$={first line
    second line
    third lines
    }
    ' last bracket defines the indent of spaces that not included in the variable
    REPORT A$
Output:
first line
second line
third lines
```

identifier: SHOW   [ΑΝΑΨΕ]

SHOW

We can SHOW the screen but we can run an application without showing the screen. Show command show the main screen (layers are showing by PLAYER command). First time the show command show both the window and the screen inside.

If a FIELD command is executed then automatic the screen is showing. The same for the CLI (command line interpreter). If CLI open and wait for command then automatic the screen showing,

If we didn't do anything for waiting, then the program close. We can use a$=key$ to wait for a key press.

There is a Show read only variable too. It is -1 when form is activated.

See HIDE, TITLE

identifier: WINDOW   [ΠΑΡΑΘΥΡΟ]

\\

1) Define size of font (the mode type) , width and height and by using ";" we center it

```
Window modetype, widthTwips, heightTwips;
```

2) Define size of font (the mode type) , width and height - Window is at the left top corner

```
Window modetype, widthTwips, heightTwips
```

3) Define size of font (the mode type) and width (height produced automatic)

```
Window modetype, widthTwips
```

4) Restore the maximum width and height

```
Window modetype, 0  ' we choose monitor 0
Window modetype, 1  ' we choose monitor 1 if 2nd monitor exist
Print Window  ' We get the monitor number (from 0) for current layer
```

Window change the screen inside and the background behind. Background have the wdth and height in pixels. but foreground, the "screen", has equal or less size depended from the modetype (the size of the fonts) and the linespacing we use. That extra space can be cut or restored with the FORM command.

WINDOW also defines the size and font size for Layers, without affect screen or background. Maybe our text not fit exactly to the size we define. We can use FORM after the WINDOW to redefine the modetype accordingly to the character resolution we provide in Form.

Look the example . Describe many aspect of M2000 programming model.

With these simple commands we make a window with a title bar that we can drag to other position and at the right top corner we can close (exit from the application). As we drag the window in a new position two threads are running and produce some data on the screen. Threats here can run without using EVERY or MAIN.TASK or WAIT. We use a FIELD command to input some text. So during that input threads running in the background.

Also we see how to create TARGET. We give a command (as a SET command) , so MM is a SET MM so we have to place it before TR module, in the first level (In M2000 we can make modules inside modules). We scan targets with scan command. We can have loops in targets but because here we use SCAN in a thread, and a thread is a time based loop, a loop with long duration will stop all threads.

```
Module MM {
    set mx=mouse.x
    set my=mouse.y
    set  startdrag=1
}
Module TR {
    ' we use SET to make some global variables
    set  startdrag=0
    set mx=mouse.x
    set my=mouse.y
    window 6, 0
    Set c = True
    Window Random(7,12) , Scale.X*.7, Scale.Y*.7;
    show
    Cls 15,0
    Gradient 0,15
    Move 0,0
    Fill Scale.X,Scale.Y/Height-25,1,6,0
    Pen 7
    ' We can move the window using the mouse on the title
    Target A,"MM",Width-1,1,,,5,"M2000 EXAMPLES USING THREAD"
    Print @(Width-1,0);
    Pen 0
    ' We close clicking top and left
    Target B, "set c = false" ,1,1,6,,5,"X"
    Pen 15
    Modules  ' just print list of modules in memory and in current directory
    AA$=Field$("",10)
    kkk=0
    Pen 0
    CLS ,20
    www=0
    Thread {
        if startdrag=1 then {
            Motion.W Motion.Xw-MX+Mouse.X,Motion.Yw-MY+Mouse.Y
        }  else {
            scan 0.001      ' only this command process the targets
        }
        if mouse=0 then  set  startdrag=0
            kkk++
    } as AA
    ' thread variables are common to the module that we create them.
    Thread {
        www++
        cls :  pen 0
        print www, mx, my, mouse, kkk
        ' we can end the program from this thread
        If not C Then Set End
    }  As BB
    Thread AA Interval 50  ' we give some time here
    Thread BB Interval 200 ' some more time here that isn't 50+200 but 200 all or maybe more..
    ' Threads need something to delay so then can run
    ' So we use here a FIELD command
    ' We can type, and we can move the window before the end of the typing...
     Field 10,10,10 As AA$
}
TR : End
```

Group: SCREEN AND FILES - ΟΘΟΝΗ ΚΑΙ ΑΡΧΕΙΑ

identifier: ?   [ΕΛ?]

Look Print

identifier: HEX   [ΔΕΚΑΕΞ]

HEX 10, 0X10+0XAF00

print hex values (unsigned), 0 or positive. If number is out of range return -??? or +???.

identifier: INPUT   [ΕΙΣΑΓΩΓΗ]

Input create variables if not exist.

1) INPUT a$ [,b,b$,b%...] input list of variables
One prompt, we can advance with a comma or enter. Numbers can have exponent part
New line after
2) INPUT string expression, a$ [,b,b$,b%...] input list of variables
One prompt, we can advance with a comma or enter.
New line after
3) INPUT a;
No new line
4) INPUT "x,y=",X,Y
One prompt only
x,y=1312,445
' use ; as last char for not using a newline
INPUT "x,y=(",X,Y; : print ")"
x,y=(121,454)
you can use the try structure

```
TRY er {
     INPUT "x,y=(",X,Y; : print ")"
}
```

variable er has 0 if an error occur, so we can take again the values
(you can use the FIELD command if you want more control.

1-2-3-4 We can paste code but not copy. You can't use arrows, only backspace for delete. Automatic scrolling up and down. You can input values in any layer, and in the background. This command give us an easy way to input values, using a floating style, seting values one by one. Use FIELD to construct forms (field use arrows and we can insert and or overwrite text). Both two commands use the non proportional text system. We can change style before these commands, italic, bold, double, pen color. Back color use the one we define with CLS.

5) INPUT #filenumber, A$,B$, A
Input from file that we open with OPEN command. Read numbers and strings, that have stored

with WRITE command.
From OPEN we can can declare if we take UNICODE (WCHAR) or ANSI, using ...FOR WIDE INPUT or FOR WIDE OUTPUT
Example for export/import csv (unicode, no BOM). delete wide to perform same program for ANSI file (1 byte per char)

```
Write with chr$(9), ",", true  ' true means that strings replaced as json strings
Open "my.dat" for wide output as #k
Write  #k, "Row 1", -213231313132312.23123@, -52322323.32#, -1.2121e+10
Write  #k, {Line1
                Line2
                Line3
                }
Close #k

Input With chr$(9), ",", true  ' true means that strings readed as json strings
Open "my.dat" for wide input as #k
Input #k, R$, M1=0@, Z1=0#, check
Input #k, Txt$
Close #k
Print R$, M1, Z1, check
Print Type$(M1), Type$(Z1)
Print M1
Print Z1
Report txt$
```

5.1) INPUT WITH stringsep$, decimalse$ [, [JsonFlag], Nouseofchr34]

stringsep$ 　　　only one char used, if "" then we get the default ","
decimalse$ 　　　only one char used, if "" then we get the default "."
jsonflag 　　　　if non zero means enconding string using \n and other characters like json strings
so we can put mulrinlie text withoutbreaking the line
Nouseofchr34 a no zero value means no "" araound a string.

if Nouseofchr34 then null string from revision 37 version 12 works

so Input With "", "" reset the writing using INPUT

We can use ANSI or UTF16LE files
For ANSI files we have to use LOCALE to set the locale for conversion
A Local 1032 used for greek conversion.

6) INPUT ! A$, Width in chars [Len =20] by default Len=50
  INPUT ! A, Width in chars

INPUT ! A%, Width in chars   (work for array items too)

Field variable give the reason to exit. 99 Esc, -1 or 1 for arrows Up and Down (and for down also the return key)

we can use Set Switches "+INP" to return Field=13 if we press enter (normaly return 1)

7) INPUT ! A$, Width in chars, Height in chars [, Title$]

7.1) INPUT ! A$, Width in chars, Height in chars[, Title$, cursor_pos

cursor_pos len(a$)+1 set cursor at rightmost place.

We use text editor to input text. We can copy & past unicode text, we can drag and drop (drag is by default not enabled, use right menu to change it). Use Ctrl & Z and Ctrl & Y for undo and redo. Use Ctrl & A for selecting all or Shift Ctrl & A to deselect. Use ctrl &X to cut, ctrl & C to copy and ctrl & V. Text editor has proportional text, at the size and style of the non proportional text. Use TAB and shift TAB to place spaces at the left (indentation) , you can mark a lot of lines and you can move left to right or in reverse. Use f2 and f3 to find up or down direction the selected text.

Text editor has own layer so after the closing (we can use ESC or we can descard changes with F12). F1 changes from word wrapping to non word wrapping view. You can use unicode text.

Use LEGEND to display the top part of the text that can be viewed in the same area, as the input area.

For one line text you can use the 9th variant in $() in Print, which print all letters without word wrap but with cut in the excess

Also you can use REPORT to justify (and use word wrapping), to calculate the max lines, to print specific lines.

Printing to the screen happen without clearing the background (only INPUT and FIELD clear under the text. In the command line interpreter we have a special version of INPUT, that remember what we input and with up and down arrows (both act as the same button) we view all the list. We can edit that list with a command EDIT without ID for making modules and functions.

If we want to clear the screen we use CLS. (cls clear the lower part, the scrolling screen or split screen, but CLS,0 set all screen as scrolling screen and erase it using current background color. We can use @() operator in PRINT or the FRAME command to prepare part of background.

We can write text as part of the module

```
a$= {1st line
        2n dline
        2dl ine
        } last bracket position used for intendetion (so this space are not insert to the string
```

identifier: PRINT   [ΤΥΠΩΣΕ]

1) use ? for PRINT

```
PRINT  1,2 3
```

we can print numbers, strings or results form expressions

```
print $(0,6)    ' 6 CHAR COLUMN
a=5
c=100
b$="A"
print a=5 and c>2, 100, "a">b$
     -1      100      -1
```

2)A special PRINT is HEX to convert decimal to hex (negative values show ???, only unsigned)

```
HEX 1,2,3
```

we can write 0xFFFF as a hex number

```
HEX 0xAA + 0xFF
          0x01A9
```

3) Print to open file

Using filename "" in Open for Output we send output to screen

included no separation spaces so we can place ," ", or any other char

```
PRINT #F, A, B$
```

Insert Newline : (2 chars)

```
Print #F
```

use WRITE #F to write strings in "" and insert "," after each print item except last

we can use INPUT #F to get from WRITE #F

and LINE INPUT #F to get from PRINT #F

4) Print to a Layer

```
LAYER 1 {
     PRINT "alfa"
}
```

5) Print to Background

```
BACKGROUND {
          print "alfa"
     }
```

6) Print to Printer

```
     PRINTER {
          print "alfa"
     }
```

7) Use @() internal operator

```
Print @(10,10), "Print at 10,10"
print @(10,10,18,11), "Hello there"
```

@(has a lot of functions. Used to prepare the background for specific item. Print prints transparent. So if we need background or because we need to overwrite, then we use @(. Also we can put bitmaps with that function in a place defined by character positions.

@(X,Y)

@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor)
@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor, Outlinecolor) 'we can make boxes with printing data
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$)   ' fill width an image...stretching it to fill the area
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$, 1) ' last parameter says "keep aspect ratio of image"

8) Use $() internal operator to format item and to the output way (with or withoit proportional text, and the justification, left, right, center or spread to other column). For proportional text we can use REPORT which have left and right justification and can calculate lines before actually print, and print starting form any line, the lines we want. So Print can be used for multi column printing of items, but for items that didn't need multi lines.
$(ColumnSJustifyID)
$(ColumnSJustifyID, ColumnSWidth)
$(StringWay2format$)
$(StringWay2format$, ColumnSWidth)

modes from 0 to 8
For numbers and logic expressions
0 = right, 1 = left, 2 = center, 3 = right - All of these using one character width for all letters
4 = right, 5 = left, 6 = center, 7,8 = left - All of these using each character own width and kerning

For strings
0 = right, 1 = right, 2 = center, 3 = left - All of these using one character width for all letters
4 = right, 5 = right, 6 = center, 7 = left , 8 = left and right both - All of these using each character own width and kerning
For 0 and 4 strings can overwrite next column(s) and advance the cursor to the next one

```
? $("00000",5),1,2,34,5  ' we can print code numbers
0000100002000340000 5
```

Special chars for StringWay2format$, f   or strings
1)@
char from item to print or space - justify right
2)&
char form item or nothing
3)<
lcase (using system locale)
4)>
ucase (using system locale)

5)!

justify left

print $("@@@@@"),"aa";$("!@@@@@@@@@@ok"),"George";   ' need ; to use that mode

? $(0) return to normal

For numbers

1)0

0 or number

2)#

number or nothing

3).

dot position

4)%

multiply by 100 and writing with a % at the end

5),

thousands delimiter

6) E- E+ e- e+

using in scientific notation (we can write numbers with scientific notation)

For date - time

1):

for time

2)/

for day

3)c

give the natural writing date and time

? $("c"), today

? $("c"), now

4)d

day only without space in the left (1 - 31)

5)dd

as 4 but with 0 in the left (01 - 31)

6)ddd

3 letters of day

7)dddd

name of day (using system locale)

8)ddddd

full day description

9)w

print number of day, 1 for Sunday

10)ww

week number (1-54)

11)m

month number (1 -12)
12)mm
as 11 but with 0 if needed (01-12)
13)mmm
name of the month, three letters
14)mmmm
name of the month
15)q
quorter of the year (1-4)
16)y
day of the year as number (1-366)
17)yy
2 digits for year (00-99)
18)yyyy
number of year (100 -99990
19)h
hour without left space (0 - 23)
20)hh
like 19 but with 0 left (00 - 23)
21)n
minutes with space left (0 - 59)
22)nn
like 21 but with 0 as left char, if needed(00 - 59)
23)s
Seconds with no space left (0 - 59)
24)ss
As 23 but with zero at left if needed (00 - 59)
25)ttttt
Time full description
26)AM/PM
 12 hour time (ucase)
27)am/pm
12 hour time (lcase)
28)A/P
like 26 but one letter only
29)AMPM
using defaul values for time format from Windows

9) Scrolling
If we print in the bottom line then scrolling happen, except we use Print Part and Print Under (one row before).
We can use SCROLL UP, SCROLL DOWN and SCROLL SPLIT number. The first two used to scroll up or down without printing, and the last one to define an upper part of screen that can not

scroll. This is the same as CLS 1,number without clearing the screen as CLS does.

10) Moving the cursor.
Cursor is hidden. But we can use @() to move it as we print or we can use CURSOR command

11) Break line

```
     11.1)  without parameter is a New Line
          Print
     11.2) two comma is a new line, three is two new lines, four is three....
          Print ,,,   we give 2 new lines
          Print 1,2,,3,4  we print
               1     2
               3     4
     11.3) Print without using tab stop
Print "alfa";"beta"
alfabeta
12) Using ; as last thing in the print to hold the new line.
        print "X=";
```

New additions in version 8

```
13) Print Over  $(6),"Center proportional text"   \\ clear background in all width, set one column
only, leave a thin line under,
                                                                         \\ no new line, no column wrap
14) Print Part "ok", 1          \\ no new line, no column wrap
15) Print Under     "Under a line"     \\ draw a horizontal line in current row, change row and do
as Print Part.
16) Print ~(#FFAA33), "Color"           \\New internal operator ~()
17) Print Back 1,2,3  \\ Clear background for this line, and maybe more if needed
18) Print #-2, format$("Hello\n\there\t100\t500\6000")
Open "" For Output as #F    ' F turn to -2
Print  #F, format$("Hello\n\there\t100\t500\6000")
Close #F
```

Example #1

```
dim bb(20)
l$={multi line string
     this is other "paragraph"
     this is { inside multiline string}
}
print l$ ' wrong way
report l$ ' this is nice
k$={print dimension("bb"), dimension(bb()      )}
```

```
    print k$
    print dimension("bb"), dimension(bb())
    dim bb(10,1)
    i= dimension("bb")
    oldtab= tab
    print "bb() dimensions=";i
for j=1 to i {
    print $(3,7),j,$(1,1), ".",$(3,8), dimension(bb(),j)  ' $(3,7) justify right non proportional printing
7 chars column, next 1 char column,  next 8 char column...
}
print $(0,oldtab)
```

Example #2

```
Flush
j=3  \\ from version 8 we have 4 chars minimum column
Form 80,32
Repeat {
    j=j+1
    Cls
    Print "Example for types of Print, Column:";j;" charcters";
    Cursor 0,Row+1
    Data 0,1,2,3,4,5,6,7, 8
    Repeat  {
        Read i
        Print $(I,j),145355,"George Karras", "The Best",
        Cursor 0,Row+1
    } Until Empty
    Cursor 0,Row+1
    Print $(0);"press any key"
    a$=Key$
} Until j=25
Print
```

Group: OPERATORS IN PRINT - ΧΕΙΡΙΣΤΕΣ ΤΗΣ ΤΥΠΩΣΕ

identifier: $([ΕΛ$]

You can use ? for PRINT

```
? "Hello"
PRINT "HELLO"
```

$(ColumnSJustifyID)

```
$(ColumnSJustifyID, ColumnSWidth)
$(StringWay2format$)
$(StringWay2format$ , ColumnSWidth)
```

```
print $("#.###;(#.###);\z\e\r\o", 10),123.21312,3123.45, -3, 0
print 1,3,4
print $(";\t\r\u\e;\f\a\l\s\e"), -1,0,-1
print $("")
print -1,0,-1

FORM 60,25
? $(3,20),  format$("{0:3}",-12.5678e45)
? $("##.##E+###"),   \\ use a coma to prevent a new line
?  -14.56789e33, -14.56789e33, -14.56789e33,   \\ last column need a comma  60 width 20 char a column
?  -14.56789e33, -14.56789e33, -14.56789e33,
? $(""),"OK"
```

After those operators we put comma (not needed a semi colon). That comma is not a new tab position

```
Print 1,2,3 ' we use three columns
    1     2     3
Print 1,,2,,3 ' we use three lines and put number in the left most column
    1
    2
    3
? $("###.##",10),1.266
    1.27
? str$(1.23,"0.0")
1.2
```

The StringWay2format$ is the same as the format string of VB6. We use it in STR$() function for formatting before use it or we can use operator $() to do that for all the next prints (until we change that)

For numbers and logic expressions
0 = right, 1 = left, 2 = center, 3 = right - All of these using one character width for all letters
4 = right, 5 = left, 6 = center, 7,8 = left - All of these using each character width and kerning

For strings
0 = right, 1 = right, 2 = center, 3 = left - All of these using one character width for all letters

4 = right, 5 = right, 6 = center, 7 = left , 8 = left and right both - All of these using each character width and kerning
For 0 and 4 strings can overwrite next column(s) and advance the cursor to the next one

example - copy this to a module (open it with Edit A and you can run it by using A)

```
------------
Flush
j=1
Form 80,32
```

Copy this to a module and watch to learn

```
Flush
j=1
Form 80,32
Repeat {
     j=j+1
     Cls
     Print "Example for types of Print, Column:";j;" charcters";
     Cursor 0,Row+1
     Data 0,1,2,3,4,5,6,7, 8
     Repeat  {
          Read i
          Print $(I,j),145355,"George Karras", "The Best",
          Cursor 0,Row+1
     } Until Empty
     Cursor 0,Row+1
     Print $(0);"press any key"
     a$=Key$
} Until j=25
Print
```

identifier: @([ΕΛ@]

@() operator move the cursor, fill a block defined by text coordinates with backcolor and or outline, fill a block with an image
not needed to place a semi colon to hold the cursor, a nice comma is what we have to put if we have something to put. If not then without a semicolon a CR is automatic added as the last part of a PRINT statement.

```
     print @(10,10);
     print 1
     is equal to this
     print @(10,10), 1
```

@(X,Y)
@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor)
@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor, Outlinecolor)
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$)   ' fill width an image...stretching it to fill the area
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$, 1) ' last parameter says "keep aspect ratio of image"

When we PRINT to screen M2000 prints without altering the background, just overwrite the chars. So if we know we have a clear screen then we don't care to fill the background. If we want to overwrite a value in a place that we have write before then we need to clear that part of screen only. Defining outline color can can make lines for columns. M2000 never type above that lines (we have to cross the columns that the line cut those to cut the line with text). Print can be output to the printer, or layers or background layer using @. So we can print to printer page, with lines, background colored forms, and images at text coordinates.

If we give a wrong filename (not exist in current dir or the path of file is unknown, or and name not exist), then a bitmap with background color printed on the screen.

We can use Bitmap preloaded in a string
If we open jpg the IMAGE command look the orientation and a rotation performed if needed.
use:

```
    IMAGE filename$ to bitmapstring$
```

identifier: ~([ΕΛ~]

Print ~(#12ff00), "Alfa"

Group: TARGET AND MENU - ΣΤΟΧΟΙ ΚΑΙ ΕΠΙΛΟΓΗ

identifier: CHANGE   [ΑΛΛΑΓΗ]

change Target_Handler, pen 15
after handler we can put name of attribute and a value, in any order
attributes:
TEXT, PEN, BACK, BORDER, COMMAND

We use SCAN to give time to work. We can put action in threads (look THREAD) so we can process clicks on targets while we do something else (mostly checking flags).The idea for the targets is to make indicators that take acknowledge from user by clicking and then for response we can change text or color or both and any other attribute.

Targets are visual objects in M2000 environment
We can create targets on any of 32 Layers, and in basic forn and in the background.
Targets can act to send keypress to environment (inside only), or can run a specific module (global)
Targets works with handlers that we hold in variables.
We can make easy a target to move a layer or the environment's own window.

There is a new update for Targets for user forms

Look TARGET

identifier: MENU [ΕΠΙΛΟΓΗ]

Menu is a special listbox that serves in two major ways:
As a menu that wait to choose something (like popup), and as a toolbox, floating in the space of environment.
We can construct menu and then we can choose how to display, or we can do all in one line
Some commands change the style of menu, for all next callings until we change those.
We don't have sub menus. We can open a new menu with a new title as sub menu.
Menus can give nothing if we press escape or when we click outside. For toolbox a click outside do nothing.
Menu have font size equal to font size from the calling layer, and use line spacing as the calling layer does. Menu open in text coordinates, but if we use title we can float it anywhere (except if we state HOLD).
As simple menu popup with one line.
Menu items can choose by Enter, by double click, by slide right
Menu list can scroll by using a hidden scroll bar, or by hold down left mouse button and push to the desired direction.

```
MENU "choose me","or better me"                  ' we can choose nothing with escape or clicking outside
```

Now the interpreter wait us to take action (threads can be run in background, and they can check if we have a MENU.VISIBLE)
a simple command MENU clear the internal list. That list we can read using MENU$() and the choice is reading by MENU read only variable. Menu equal to zero means nothing chosen. In a thread we can read MENU as selected but not chosen.

```
MENU    ' clear the list
MENU +"1st line","2nd line"   ' append two items
MENU +"3rd line","4th line"   ' append two more
MENU !   ' just show the menu

' Advance example using threads
' copy this in a module
i=0
thread {
     i++
     if menu.visible then print i, menu$(menu)
} as a
menu title "choose"
after 100 {
     thread a interval 100
}  ' so we start thread "a" when the menu is showing
menu "1st line","2nd line"
print menu
' thread is erased at the end of the running module...or we can erase anytime by using: thread a erase  or inside thread by using: thread this erase

' example of using multiline title, and new colors

a$={This
     is
     a 3 line menu title
     }
' menu title a$ hold   ' no float menu
menu title a$   ' use menu title "" to delete title lines
menu frame     ' use  off to disable   frame is always as for buttons..black.
menu fill color(100,130,200),COLOR(255,255,50),7   ' title color, list color and text color
' MENU FILL 1  ' we reset  list color and text color  to default (reverse pen and background color)
menu "I want some space              ", "next item", "next item too"
```

identifier: SCAN   [ΣΑΡΩΣΕ]

```
scan
      scan targets waiting for a press on target  see TARGET
scan  .5
     scan targets waiting 0.5 second for a mouse press on target  see TARGET
```

identifier: TARGET   [ΣΤΟΧΟΣ]

We can make targets, visible or not. Targets are frames on layers to be used by clicking on it, when Scan command used)

```
Cursor left1, top1
TARGET  handlervariable, "Beep",  tWidth, tHeight, FrameColor, OutlineColor, LabelPositionId, "label"
```

We can use Change to set some properties
Position Number (LabelPositionid)
1 2 3
4 5 6
7 8 9
We have to use one number, so number 5 means in the center of frame
If we add 10 to this number if we want the command to interpret as keys.
We can give a module name for command (in place of Beep command). This module has to be global. So when we click on a target we can call that module (threads can be run in the background). That module can set up a thread after some time to start. So in the exit of a target click we have code that start once with no specific module, but the global one. We can erase threads with THREADS ERASE
Target handlervariable, 0 we make individual target to not respond
Target handlervariable, 1 we enter target to list for scanning and we redarw it
Mode command erase all targets from specific layer
A cls erase the visual part of targets, not the scanning process.
Targets New erase all targets, from all layers.
We can use targets in background, basic layer and 32 layers above.
A very simple system for scanning targets with irregular shape is to make a hidden layer and insert there a picture with painting areas so you can search similar coordinates for the color (and so color became a new handler).

We can use Targets in user Forms, see the example (we didn't use SCAN for user forms). Command now call specific module (where we make the form). We can use an event for targets. Targets on forms used with one click. All other controls need two click to execute, one click to choose. Targets didn't get focus.

```
Declare form1 Form
m=false
Module zz {
        \\ target call zz to own object
        \\ so we have to use Layer {} to send to console
        Layer {
                print "ok", timecount
                refresh
                }
        change A, text "ok"
}
Function form1.MouseDown {
        Stack
        refresh
}
Function form1.Target {
        \\ normal events executed to console
        \\ so we have to use Layer to redirect to Form1 layer
        Print "target:", number, A
        Refresh
        Layer Form1 {
                Target A, m
                if m then
                        change A, back 2
                else
                        change A, back 0
                end if
        }
        m~
}
Layer form1 {
        window 12, 12000, 8000
        form 32, 20
        motion.w;  ' center window
        cls #333333, 0
        cursor 4, 4
        \\ we use immediate execution, in this module
        \\ so we call zz as this module (call local pass current scope to zz)
        target A, "call local zz", 10, 3, 2,15, 105, "press me"
        cursor 4, 8
        \\ 115 means 5 center, 10 use target event, 100 proportional font rendering
        target B,"z", 10, 3, 5,15, 115, "press me too"
}
Method form1,"show",1
Declare form1 nothing
```

identifier: TARGETS   [ΣΤΟΧΟΙ]

TARGETS NEW
clear memory for targets. Same can be done with Mode and Window commands, but we clear

layer also.

Group: DRAWING 2D - ΓΡΑΦΙΚΑ 2Δ

identifier: CIRCLE   [ΚΥΚΛΟΣ]

Make a circle at current graphic position

CIRCLE radiusInTwips, ratio, borderColor, fromStartAngle, tofinishAngle

CIRCLE FILL color2fill, radiusInTwips, ratio, borderColor, fromStartAngle, tofinishAngle

Ratio 1 is circle, but <1 or >1 is eclipse.
graphic cursor unchanged

identifier: COLOR   [ΧΡΩΜΑ]

Define color/path in a block of code. There are four variations for color or path structure

```
1) COLOR fillcolorNu [, fill style] {
     block of code
     }
     All graphics inside structure has this color as fillcolor
2) COLOR ! fillcolorNum [, fill style] {
     block of code
     }
     All graphics inside structure has this color as fillcolor  AND DRAWING with Xor type of blend
     So second time the graphics erased and the screen restored
     There are also other ways to restore the screen...such as HOLD  (the bitmap) and then
RELEASE (the bitmap) of screen.
3) COLOR {
     block of code
     }
     No fill color. Only drawing lines with pen color. See an example in POLYGON
4) COLOR {
     block of code
```

```
};
```

Look the semi colon after the ending of block

All the area defining in block code, from a polygon, a circle, anything, creates path for exclusive printing inside.

If you like you can call it PATH (because it is a path, as in windows terminology, but for M2000 is a color manipulator, so is a part of color structure)

You can reset the path with these commands

```
        Move 0,0
        Path {POLYGON 0, x.twips, 0,0,y.twips,-x.twips,0,0,-y.twips};
        ' first 0 is the color, so first point is x.twips,0 the top line or you reset it with CLS (clear screen)
        or better Path {};  clear path
        or CLS
```

identifier: CURVE   [ΚΑΜΠΥΛΗ]

Bezier Curves. Two variations, one with polar coordinates

1) Curve X1,Y1......XN, YN

```
    CURVE 1000,1000,0,2000,500,1000
```

pos.x advance to 1000+0+500 = 1500 twips right

pos.y advance to 1000+2000+1000=4000 twips left

2) CURVE ANGLE angleRad1, DistanceTwips1,angleRad2, DistanceTwips2,angleRad3, DistanceTwips3, angleRad4, DistanceTwips4

Example1:

```
        CLS,0   \\ current background color and reset split screen to start from top
        MOVE 5000,5000
        PEN 1
        FOR Q=PI/8 TO 2*PI STEP PI/8 {
            MOVE 5000,5000
            PATH 8 { CURVE ANGLE PI/3+Q,500,PI/4+Q,3000,PI+Q,2000 }
        }
        PEN 15
```

Example2:

```
        CLS,0
        MOVE 5000,5000
        PEN 1
        FOR Q=PI/8 TO 2*PI STEP PI/8 {
            MOVE 5000,5000
```

```
                    PATH ! 2 {
                         CURVE ANGLE
PI/3+Q,500,PI/4+Q,1000,PI+Q,2000,PI/3+Q,500,PI/4+Q,100,PI+Q,1000
                        }
               }
               PEN 15
```

Example3:

```
               CLS,0
               MOVE 5000,5000
               PEN 1
               FOR Q=PI/8 TO 2*PI STEP PI/8 {
                    MOVE 5000,5000
                    PATH RANDOM(7,15) {
                        CURVE ANGLE
PI/4+Q,500,PI/Q,1000,PI*1.3+Q*Q,1000,PI/Q,500,PI/2+Q/2,1000,PI*1.3+Q,4000
                    }
               }
               PEN 15
```

identifier: DRAW   [ΧΑΡΑΞΕ]

1) DRAW RelativePosXTwips, RelativePosYTwips [, color]

color can be 0 to 15 basic colors, or an rgb value (negative numbers for M2000), or system colors like 0x80000001

You can use COLOR(rValue, gValue, bValue) values from 0 to 255

You can use hexadecimal form with minus sign -0xAA00FF

(M2000 reads hexadecimal as unsign long, but sign is -1* so -0xAA00FF is an arithmetic expression of -1 * 0xAA00FF). Better you can use Html color hex #FF00AA (always 6 hex digits else we get an error)

2) DRAW ANGLE rad, distanceTwips [,color]

3) DRAW TO absolutePosXTwips, absolutePosYTwips [,color]

If you want to change line width use WIDTH { } structure

Drawing can perform inside closed paths, see PATH {} structure, so anything drawing outside the path surface stay unchanged.

see FILL and FLOODFILL
also see POLYGON, CIRCLE, CURVE

identifier: DRAWING   [ΣΧΕΔΙΟ]

1) Without dimensions, system find the final dimensions, from bounding rectangle
If we use Gradient then we get it inside a fake frame of the same dimensions from the layer where we make the drawing.
We can draw the drawing using Image, Sprite and Copy. In sprite transparency not working for emf, but if emf has transpatent areas these used. The statements may use emf plus, with trasparent brushes, but the Drawing statemt use GDI32 to produce emf.
We can load a file to a buffer using a simple statement A=Buffer("this.emf").
We can paste from clipboard: A=Clipboard.Drawing
We can export to file using Open "another.emf" for output as #F : Put #F, A: Close #f
We can export to clipboard: Clipoard A

```
Drawing {
statements...
} as A
```

2) Using dimensions

```
Drawing 3000,3000 {
statements...
} as A
```

identifier: FILL   [ΒΑΨΕ]

Fill a rectangle with 2 variations, one use windows system fill rectangle call
1) FILL widthTwips, HeightTwips [,color1 [, color2 [, direction[, part]]]]
    Filling a block from graphic cursor, with width in twips and height in twips,
    Pen color or color1 (1..15 basic colors, -0 to -16777216 , 0x80000001 system windows colors)
    2nd Color is for gradient, direction 0 for left right direction and 1 for top down,
    finally part 0 all gradient as a fill and part 1 as a window of screen filling (mimic a path)
    after execution graphic cursor has the down right corner of fill block

2) FILL @ widthTwips, heightTwips, way, number, borderTwips
    and a variant of this
    FILL @ widthTwips, heightTwips, StringExpression
    After execution graphic cursor not changed
        Way:
            0 - Frame
            1 - Background for letters
            2 - Draw

3 - Draw (Xor) so the second time the drawing erased without affect the screen
4 - Draw some icons from windows
5 - Center and print letters or numbers
6 - Like 4 but we can define which frame side we wish to display (1+2+4+8 for all)
3D frames: Fill @ 2000,200,4,1
We can use the 3rd way to make rectangle non flashing cursors

identifier: FLOODFILL   [ΓΕΜΙΣΕ]

Fill color for any closed line shape
1) FLOODFILL absolute_pos.twips, absolute_pos.twips, color.ID
Change color to color.ID on the color founding at graphic position
Color.id: Use the color representation of your choice like, see DRAW
2. FLOODFILL COLOR absolute_pos.twips, absolute_pos.twips, color.ID
Change color until found color.iD starting at graphic position with color.ID color.

```
FLOODFILL 2000,2000,5
FLOODFILL ,,5  ' where graphic posisiton is
FLOODFILL 1000,1000  'using pen for filling
```

identifier: MOVE   [ΘΕΣΗ]

Absolute moving without drawing, in twips.

```
MOVE 1000,1000
MOVE 3000
MOVE ,5000
```

current cursor at POS.X, POS.Y (read only variables)
Left top point at 0,0

Right Bottom point at X.TWIPS, Y.TWIPS (or SCALE.X, SCALE.Y) identifier: POLYGON   [ΠΟΛΥΓΩΝΟ]

1. Using a fill color and relative coordinates (you can add points if you like):

a) POLYGON 4, 1000,0,0,1000,-1000,0,0,-1000

Last point is moving the cursor

b) POLYGON 4, 1000,0,0,1000,-1000,0,1000,-1000

a and b is the same drawing but leave cursor in other point.

2. Polar coordinates

```
MOVE 5000,2000
POLYGON 4, ANGLE PI/4,3000,-PI/4,3000,PI/4,-3000,-PI/4,-3000
```

if we don't want to fill it, we can with path {} ori color {} (is the same structure

```
PATH { POLYGON 4, ANGLE PI/4,3000,-PI/4,3000,PI/4,-3000,-PI/4,-3000 }
```

We can strip the color (and other things width path or color (is the same structure)

curve get the line color, but polygon get the fill color and always close the line to perform the filling (or width color|path has a transparent filling)

identifier: PSET   [ΧΡΩΜΑΤΙΣΕ]

```
PSET
PSET 15
PSET COLOR(100,100,500)
PSET 15, 6000,6000 'graphic cursor not affected
PSET COLOR(100,100,500), 6000,6000
```

identifier: SMOOTH   [ΟΜΑΛΑ]

```
SMOOTH ON
SMOOTH OFF
```

Enable GDI+ for lines, bezier courves, polygons and circles.

identifier: STEP   [ΒΗΜΑ]

Moving graphic cursor but not drawing relative distances

```
STEP 2000, 3000
```

Moving with polar coordinates. Angle always act as compass. Relative distance

```
STEP ANGLE pi/4, 1000
```

identifier: WIDTH   [ΠΑΧΟΣ]

Define a block of code where commands for graphics use a specific line width

```
Width  lineWidth [, line style] {
     block of code
}
```

We can print text with outline setting with Width {}.

For GDI line style from 2 to 5 works only for width 1 pixel

For GDI+ works for every width.

Group: BITMAP COMMANDS - ΕΝΤΟΛΕΣ ΕΙΚΟΝΩΝ

identifier: COPY   [ΑΝΤΙΓΡΑΨΕ]

We can copy from current layer any part to that layer or a string variable

Or we can copy from a string variable to current layer using rotation angle and zoom factor

1) from current X and Y
```
COPY  x.destination, y.destination, width, height
move 2000,2000
copy 5000,5000,3000,3000
```

2) Copy screen to file
```
COPY filename$
```

3) Copy from current X and Y to a string
```
COPY width, height TO variable$
```

4) Copy to X and Y from a string - without altering the graphic cursor
```
COPY X,Y  USE variable$
```

5) As 4 but with angle and zoom factor
```
COPY X, Y  USE variable$, angle
COPY X, Y  USE variable$, angle, zoom_factor
```

6) As 4 but with angle and zoom factor and showing the image under at the corners
```
COPY X, Y  TOP variable$, angle
COPY width, height  TOP variable$, angle, zoom_factor
```

variable$ can be an item from an array
current X and Y stay as is.

Hot spot (the rotation point and where the X, Y link to image) is in right top corner of the output bitmap.
We can use SPRITE with hot spot at the center (also the background, area behind sprite, saved in SPRITE$ for easy remove). Keeping backgrounds we can handle sprites,without actually create objects. So we paint sprites as we copy image, but with some additions: mask color (for total transparency) and transparency for all other
We can use PLAYER an advance layer system with rotation but without painting to screen but on top of it. PLAYERS are act as real sprites having transparency with mask color, using a z-order system. We can make up to 32 of them. A PLAYER is an object that can be write on them as LAYER, and we can create targets on it, make drawings and copy images, and print using dedicated mode (font size) and fontname. So we can make toolboxes in a LAYER and move it as a PLAYER.

```
FORM 60, 32
CLS 7
```

```
PEN 15
PRINT "This is first line"
PRINT "This is second line"
PRINT "This is third line"
PRINT "This is forth line"
MOVE 0,0
CLEAR A$
COPY 3000,2000 TO A$
GRADIENT 3, 5   ' CLEAR SCREEN WITH GRADIENT
COPY 6000,4000 TOP A$, 30, 200
COPY 6000,4000 TOP A$, 45
COPY 2000,2000 USE A$, 45
```

identifier: IMAGE   [EIKONA]

1) Display an image to screen (bmp, wmf, emf, jpg, gif, ico)

```
Image filename$ [, width [, Height]]
Image BufferWithImage [, width [, Height]]
```

we can make image from file to buffers (holding there as files) to use with GDI+
Look Buffer()
if we give Height then we can stretch the image.

2) Image from file to variable

```
Image  image$ to newimage$  [(background color to fill transparent pixels)]  ' this is for emf
Image  image$  [(background color to fill transparent pixels)] to newimage$     ' this is for png bitmaps

Image  imageA to newimage$  [(background color to fill transparent pixels)]  ' this is for emf
Image  imageA  [(background color to fill transparent pixels)] to newimage$     ' this is for png bitmaps
```

3) As 2 but with resize

```
Image image$ to newimage$ [(background color to fill transparent pixels)]  , percentSize   ' 100 for same size
Image image$ [(background color to fill transparent pixels)]   to newimage$ , percentSize   ' 100 for same size
Image imageA to newimage$ [(background color to fill transparent pixels)]  , percentSize   ' 100 for same size
Image imageA [(background color to fill transparent pixels)]   to newimage$ , percentSize   ' 100 for same size
```

4) As 2 but resample Width and Height

Image Image$ to newimage$ [(background color to fill transparent pixels)], percentWidth, percentHeight
Image Image$ [(background color to fill transparent pixels)] to newimage$, percentWidth, percentHeight
Image ImageA to newimage$ [(background color to fill transparent pixels)], percentWidth, percentHeight
Image ImageA [(background color to fill transparent pixels)] to newimage$, percentWidth, percentHeight

65) Export as Bmp
\\ not for images in buffer (these are files in memory use file operation:
\\ Open... for output /close and a line inside Put #F, A if A is the buffer)
Image image$ Export filename$

6) Export as jpg
Image image$ Export filename$, percent_compression_quality
 100 percent_compression_quality give bigger file, more accurate to original image

image$ can be a filename and a path or an image in a string

identifier: PLAYER   [ΠΑΙΚΤΗΣ]

Above main screen there are 32 Layer acting as Players. Those players are "hardware" sprites, because the system draw them with GPU.

1) Player 0
Flush all players
2) player 1, 2000,3000 use AA$, 1, 0,angle [, ShiftX, ShiftY]   size 1.4
  ShiftX and ShiftY shift the place of player (so -1000, -1000 added to 2000, 3000, each time we place only a new position). Also these numbers used from Collide() function to shift the frame of interest from the origina position to the new position (as a new center)
2.1) player 1, 2000,3000 use AA$, MASK$, 0,angle [, ShiftX, ShiftY] size 1.4
First number is priority number. 1 for lowest and 32 for higher (32 player is top of all)
Position 2000,3000 relative to window (see Window)
AA$ is a string containing a DIB. Color for transparency, 0 to 15 and negatives for RGB (use Color(0..255, 0..255,0..255) to define color. We can give an intensity so we can make

transparent colors in a bigger region. With -1 in intensity we cancel transparency. And finally we have the size, 1 for 100%

3) Player 1 show
   A player can change invisible. So we must show to be displayed on screen
4) Player 1 hide
   The opposite. Player going to standby.
5) Player 1 swap 4
   We can change priorities. We use variables as handlers so we can change what the handlers show to us. Look Collision
   Two more commands added in version 10: Over and Under. We can place a player over another player. So the players from one to other change priority numbers.
6) Player 1, 5000,6000
   Moving Player to new position.

We can handle Players with command Layer as screen. So we can put targets, graphics, scrolling numbers and formatted text, images.

Example

In a module A copy this:

```
    Background {
        Gradient 1,5
    }
    Form 20,40
    Move 0,0
    clear AA$
    Cls 0,0  :  Pen 15
    Print "aaaaaaaaaa"  :  Print "{ {  } }"    \\ a simple image
    Copy 3000,1000 To AA$
    SZ=1  \\first size...1 in player means 100%
    Player 1, 2000,2000 Use AA$,  15, 0 size  SZ
    Player 1 Show
    Wait 10
    Cls   4, 3
    Move 0,0
    TRY {  \\ we use try { } yo catch Break or Escape
    EVERY 30 {
        XX=MOUSEA.X
        YY=MOUSEA.Y
        Draw To Mouse.X, Mouse.Y
        Background {
            Draw To Mouse.X, Mouse.Y
        }
        Player 1,XX-3000*sz/2, YY-1000*sz/2
        a$=Inkey$
        If A$<>"" Then {
            Select Case A$
            Case "1"
                If SZ>.6 Then SZ=SZ/1.2
            Case "2"
                If SZ<5 Then SZ=SZ*1.2
            Else
                BREAK
            End Select
            \\ change hot spot from middle to bottom right corner
            Player 1, mouseA.x-3000*sz/2, mouseA.y-1000*sz/2 Use AA$,15, 0 size SZ
            A$=""
        }
    }
    }
    Player 1 Hide
    Player 0 ' clear all
```

identifier: SPRITE   [ΔΙΑΦΑΝΟ]

There are software sprites (from earlier versions).
1) Put a sprite in graphic position - which is the center of sprite (the hot spot is always in the center of image of sprite)
Sprite a$
A$ is a bitmap DIB in a string see Copy and Image
2)
sprite a$, colormask
3)
sprite a$, colormask, angleindegree
4)
sprite a$, colormask, angleindegree, percentSize ' 100 for 100%
5)
sprite a$, colormask, angleindegree, percentSize, transparency [, 1]' 100 for no transparency
6)
sprite a$, colormask, angleindegree, percentSize, transparency_mask$ [, 1] ' 1 or non zero for no copy screen part to feed back area from sprite.

sprite$ is a read only variable that hold the surface which a sprite command replace. So sprite sprite$ on the same coordinates restore this part of screen.

```
refresh 100
clear a$
move 0,0
copy 8000,1000 to a$
move 1000,1000
sprite a$,,0
every 50 {
     refresh 0
     sprite sprite$
     move mouse.x, mouse.y
     sprite a$,,random(-30,30), random(30, 90), 50
     refresh
     if mouse>0 then exit
}
```

Using Hold (to hold screen) and Release (to release screen, before writing)

```
refresh 100
clear a$
move 0,0
copy 3000,1000 to a$
hold
move 1000,1000
sprite A$,,0
I=1
every 30 {
     refresh 0
     i++
     release
     move mouse.x, mouse.y
     ' sprite A$,,i,I*1.3+10, 50
     sprite A$,,i,I*1.3+10
     refresh
     if i>200 then exit
}
```

Group: DATABASES - ΒΑΣΕΙΣ ΔΕΔΟΜΕΝΩΝ

identifier: APPEND   [ΠΡΟΣΘΗΚΗ]

1) for tables in databases

```
APPEND nameofbase$, nameoftable$, fieldvalue1, fieldvalue2$, ....fieldvalueN
```

All fieldvalues are one of two general types, number or string. So a memo is a string.

Examples

```
Append Base$, "Record", Result, Str$(now, "SHORT DATE"), Code, Amount
Append Base$, "Person", CodePerson, Results, Val(Trim(Amount$))
```

2)Append work for Inventories and pointer to arrays

```
a=(1,2,3,4)
Append a,(5,6,7,8), (9,10,11,12)   \\ , ...
Print a
Inventory b=1,2,3,4:=300,5
Append b, 100:=500, "hello":=1000, "ok":="all right"
Print b$("ok"), b(4)
Print b
```

identifier: BASE   [ΒΑΣΗ]

From version 7, we handle UNICODE text and memo fields. No DAO used, but ADO for mdb only databases

I. Using numeric argument

Base 1   - for arrays from 1,

Base 0   - for arrays from 0

We can set any base using (TO): Dim a(-4 TO 4), b(-1 to 1, -1 to 1)

II. Using string argument

1. You can make a database easy

First you must define the name of DATABASE

```
    BASE "mine"   ' SECOND TIME DELETE THE DATABASE  AND CREATE IT AGAIN
```

with a single exist("mine.mdb") we can read if database exist as a filename in current directory

but we can check if "mine" database exist and can connect with it

```
    ? CheckBase ("mine")  ' not need the type... .mdb

            FUNCTION CHECKBASE {
                ok = false   ' needed because TRY can't create variable
                try ok {
```

```
                structure  letter$          ' EXPECT 1 OR 2 STRING VALUES FROM THE STACK
                                     ' this fill the stack with table names but return also an error
when
                                     ' file not exist
                                     ' can't connect to base (maybe is exclusively open buy other
app)
                                     '  we can check if a table exist by providing additional
parameter
                                     '  to the function stack  ? CheckBase ("mine","tablename")
                }
                = ok
            }
```

2.You need a table definition

A name for the table and a list of field triplet...name, kind and size

These are the kinds of fields

BOOLEAN, BYTE, INTEGER, LONG, CURRENCY, SINGLE, DOUBLE, DATEFIELD, BINARY, TEXT, MEMO

```
        ' 0 for length means..the database define only (we can do otherway)

    TABLE "mine" , "firstTable", "field1", long, 0, "title1", text, 40, "memo1", memo, 0
```

3. And you can set the order for simple reading DESCENDING or ASCENDING

```
    ORDER "mine", "firstTable", "title1", ascending
```

You can put as many tables as you wish.

You can put directory and the ".mdb" in the name of base

The base is password protected (no a srtong one) for a reason...To know if this is "our" base. If isn't "our" base then we can't delete it
You can delete the database using DELETE. This command finds the key and then if is right then delete it. Using this way we don't perform any reading with our password to find that is invalid. We read the password and then we know.

```
    DELETE "mine"
```

We can read the structure from any BASE, with STRUCTURE command
Use RETRIEVE тo get a row of fields (using SQL if you like)
Use EXECUTE to send a command without returning fields
Use SEARCH тo get a row of fields using a filter
Use RETURN to update row with new value

Use APPEND to push new row to any table
Use DELETE to throw a row with a specific value in a specific field
Use VIEW to fill a selection list (a drop down menu, basically) using a "SELECT DISTINCT field1 FROM table1" to get a table for one list
Use CLOSE BASE to release the conection (new command)

You can COMPRESS the base (to throw deleted rows)

About Language M2000 and how the interpreter can get values and send back to a database M2000 has a structure as a stack of values (all modules calling other modules in the some stack, but threads and functions has own stack), so when we want a list of data commands place there the data, and in the top the number of data that follows. So if we have a module and a command for a base retrieve field data, that data fill the stack and module can call an other module, as a routine, without passing "exclusive" parameters (like in other languages). This cannot be done with functions, but inside function we can call modules, and that modules can use the function stack as own stack.

identifier: COMPRESS   [ΣΥΜΠΙΕΣΗ]

```
Compress base$+".mdb"
```

erase previously "deleted" records

use CLOSE BASE dir$+base$+".mdb"
before make the compress call, to close any open connection to that base

identifier: DB.PROVIDER   [ΒΑΣΗ.ΠΑΡΟΧΟΣ]

```
DB.PROVIDER "Microsoft.Jet.OLEDB.4.0", "Jet OLEDB", "PASSWORD"
```

password for base encryption

identifier: DB.USER   [ΒΑΣΗ.ΧΡΗΣΤΗΣ]

```
DB.USER "dbusername", "dbpassword"
```

identifier: DELETE   [ΑΦΑΙΡΕΣΗ]

1) Delete database$, tablename$, fieldname$, fieldvalue$
Need exactly the field value (internal compare the value and if ok then erase the field)
So first you find the record, you take a copy of a specific field and perform delete. So it is better to not have duplicates (if is a unique key then we get error if we insert another copy of already used key)

2) Delete database$
Delete database only if created by M2000 (if has the "original" M2000 password).

(When we use BASE to make a database, then if base exist will be erased (checking for password for safety) and then a new empty file get the place of old one.)
We can use Execute to perform a multi record delete
Execute "HELP2000", "DELETE FROM COMMANDS WHERE GROUPNUM =22;"

identifier: EXECUTE   [ΕΚΤΕΛΕΣΗ]

1. For DataBases
EXECUTE basename, executestring
EXECUTE basename, executestring, timeout
by default timeout = 30 (minimum 10)

```
\* EXAMPLE
BASE "ALFA"  ' ERASED IF FOUND THE NAME
EXECUTE "ALFA", { CREATE TABLE Employees(ID autoincrement primary key,  LastName VARCHAR(40) ,  FirstName  VARCHAR(40)  NOT NULL, MyMemo TEXT )}
APPEND "ALFA","Employees",,"George","йваоаптаЭЭ","Γεια χαρά από Ελλάδα"
RETRIEVE "ALFA", "Employees", 1,"",""
READ MANY, ID, LASTNAME$, FIRSTNAMES$, MEMO$
PRINT $(4, 6),  "BASE:","ALFA"
PRINT  "NAME:",LASTNAME$ + " " + FIRSTNAMES$
PRINT "MEMO:"
PRINT "",
REPORT  MEMO$
CLOSE BASE "ALFA"
PRINT
\\ ))))))))))) New Example (((((((((((
\\ erase stack items
flush
\\ erase gEx.mdb if found, create a new one
base "gEx"
```

```
\\ we use semi colon to separate statements. Don't use semi colon inside strings in execute,
execute "gEx", {
        create table alfa
        (
                name varchar(50) not null,
                birth_date date default null
        );
        Insert into alfa values ("George","27-10-1966")         ;
        Insert into alfa values ("Alex",NULL)
}
print Stack.size=0
def CheckNull$(a, a$)=If$(type$(a)="Null"->"NULL", a$)
execute "gEx", {select * from `alfa`}
\\ now we have a rerurn value in stack of values
print Stack.Size=1, StackType$(1)="Recordset"

GetListA()

print "ok"



sub GetListA(RS)
if type$(RS)="Recordset" then
        with RS, "EOF" as new rs.eof, "fields" as new fields(), "fields" as new fields$(), "fields" as new rs.fields
        with rs.fields, "count" as new rs.fields.count
        print  "Test number of fields:", rs.fields.count
        while not rs.eof
                print $(6),fields$(0),
                print $(9)," ";CheckNull$(fields(1), fields$(1))
                method rs, "movenext"
        end while
        print $(0),,
end if
end sub
```

2. For executing machine code

```
\\ we make a buffer to use for DATA
Buffer BinaryData as Long*10
Return BinaryData, 1:=500
```

```
Buffer code alfa as byte*1024
\\ use https://defuse.ca/online-x86-assembler.htm
\\ to find opcodes
Pc=0
\\ x86 Machine Code
OpLong(0xb8, 5100)  ' mov eax,0xa     ' 10 to eax
OpByteByte(0x83, 0xC0 ,5)  ' add  eax,0x5   ' add 5 to eax
OpByteLong(0x3,0x5, BinaryData(1)) 'add eax, [BinaryData(1)]  ' add eax 500 from second long on BinaryData
OpLong(0xa3, BinaryData(0))  ' mov [BinaryData(0)], eax
rem : OpByte(0x31, 0xC0)  ' now eax=0  ' without this we get 5605 in M
Ret() ' return
\\ end of code
Try Ok {
     Execute Code alfa, 0
}
M=Uint(Error)
Hex M
Print M
Print Error, ok
Print Eval(BinaryData, 0)  ' 5605

Sub Ret()
     Return alfa, pc:=0xC3
     pc++
End Sub
Sub OpByteByte()
     Return alfa, pc:=number, pc+1:=number, pc+2:=number
     pc+=3
End Sub
Sub OpByte()
     Return alfa, pc:=number, pc+1:=number
     pc+=2
End Sub
Sub OpLong()
     Return alfa, pc:=number, pc+1:=number as long
     pc+=5
End Sub
Sub OpByteLong()
     Return alfa, pc:=number, pc+1:=number, pc+2:=number as Long
     pc+=6
End Sub
```

identifier: ORDER   [ΤΑΞΗ]

You can set the order for simple reading using DESCENDING or ASCENDING
ORDER "mine", "firstTable", "title1", ascending   [, "title2", way]

identifier: RETRIEVE   [ΑΝΑΚΤΗΣΗ]

1. RETRIEVE BaseName$, TableName$, 1, "",""
open base and place to stack on the top the number of records and under the fist row of item values, of theTableName$
2. RETRIEVE BaseName$, Sql$, 1, "",""
we can set a SQL string defining a recordset to read. We get always the number of records and the choosen record
3. RETRIEVE BaseName$, TableName$, 1, "FieldName","ValueString"
we get the 1st record where Fieldname = ValueString
4. RETRIEVE BaseName$, TableName$, 1, "FieldName", Value
we get the 1st record where Fieldname = Value

Use % not * in a Like operator (.. Name Like "A%")

identifier: RETURN   [ΕΠΙΣΤΡΟΦΗ]

1) For databases
RETURN "THISBASE","SELECT * FROM THISTABLE WHERE THISFIELD LIKE 'MAGELAN%'" , ,"EARTH THERE","DATA2","DATA3"
(use % not * in a string after LIKE, because ADO need %)
RETURN "THISBASE","THISTABLE" TO OFFSET, FIELD1, FIELD2....

SO we return to base THISBASE, in a table THISTABLE in a record which a field THISFIELD is like MAGELAN* only 2nd, 3rd and 4th fields (first field is the name so we can skip this with double coma)

If we use a table as look up table (using record from an offset) then is another format of RETURN
Lets say that we need to change specific field at Selection index

```
Retrieve "HELP2000","GROUP", Selection,"",""
```

we do what we want with that record and then we want to send it back..by OFFSET and no with SQL query

```
Return "HELP2000","GROUP" to Selection, ,groupname$
```

2) For flow control

```
Gosub alfa
End
alfa:
Print "ok"
Return
```

3) For Inventories, Arrays, Stacks

```
Inventory a=1:="Hello","other":="Yes"
\\ Inventory is a list with key and value
Return a, "other":="No", 1:="Hello again"
Print a$("other"), a$(1)
aa=each(a)
Print "key", "value"
while aa {Print eval$(aa!), eval$(aa)}
b=(1,2,3,4,5)   ' auto array
\\ Array(b) is the first item
Return b,3:=400, 0:=Array(b)+1,1:=Array(b,1)**2  ' for base 0 index 3 is the forth item
Print b
Print Array(b,3)
bb=each(b)
While bb { Print str$(Array(bb),"00000")}
\\ Stack is a linked list. Keys are positions from 1. We can move any item easy
m=Stack:=1,2,3,4,5
Return m, 4:=400,1:=StackItem(m)+1, 2:=StackItem(m,2)**2  ' Stacks have base 1
Print m
Print StackItem(m,4)
mm=each(m, -1, 1)
While mm { Print str$(Stackitem(mm),"00000")}
Stack m { Shift 4 : ShiftBack 5} \\ send 4th item to 5th position
Print  m
```

identifier: SEARCH   [ΑΝΑΖΗΤΗΣΗ]

```
SEARCH "basename", "tablename", Offset, "field", operator$, value
SEARCH "basename", "tablename", Offset, "field", operator$, value$
```

This is a more compact RETRIEVE
operator$ "LIKE" ">=" etc
For like operator string wildcard is % not *

identifier: STRUCTURE   [ΔΟΜΗ]

This is the command to get the structure of database

1) STRUCTURE "basename"
2) STRUCTURE "basename", "tablename"
3) Look Buffer - Structure command make types for data for Buffers in memory.

This is an example of how we use it

```
Form 80,50
DB.Provider ""   ' reset provider - maybe M2000 Interpreter change this to open an mdb file
a=Tab ' save TAB width
Print $(0,8) ' new tab  width 8 chars
Menu '  without parameter we clear menu list
Files+ "mdb"  ' With plus symbol we send Files output to Menu List

If Menuitems>0 Then {  ' if we have something
    Print "Choose a database:";
    Menu !                    ' A listbox open now
    If Menu>0 Then {
        base$=Menu$( Menu )
        Print lcase$(base$)+".mdb"
        Print "Structure"
        Print "Tables:";
         Structure base$   'leave values to stack
        Read TablesNum ' we make a new variable TablesNum
              ' and we get the value from stack

                Print  str$(TablesNum)  ' strings are justify left by default
        If  TablesNum>0 Then {
            For i=1 to TablesNum {
                Read Tablename$, TableIndexes
                Print i,") "+Tablename$
                Structure base$, Tablename$
                Read FieldsNum, TableIndexes
                        ' So we see if we have number at top on stack.
                If  FieldsNum>0 Then {
                    For F=1 To FieldsNum {
                        Read FieldName$, FieldType$, FieldLength  ' Structure command put
that in stack
                        Print "",F,") "+Field$(FieldName$,17),"",FieldType$, FieldLength
                        ' Field$() function is a function to place spaces or turncate the excess
length
                    }
                }
                If  TableIndexes>0 Then {
                    Print "","Order:"
                    Read FieldsNum
                    If  FieldsNum>0 Then {
                        For F=1 To FieldsNum {
                            Read FieldName$, FieldType$' Structure command put that in stack
                            Print "",F,") "+Field$(FieldName$,17),"",FieldType$
                        }
                    }
                }
            }
        }
    }
}
Print $(0,a)  ' restore TAB length


\\ using block {} inside structure we make unions
\\ so we have access to memory with one or more types
\\ byte, integer (2 bytes), and long (4 bytes) are unsigned numbers
\\ double (8 bytes)
\\ there is no string type. We have to use integer*number
Structure stringA {
    str as integer*20
}
Structure beta {
    Alfa as byte
    align1 as byte*3
```

```
      Kappa as long*20
      name as stringA*10
      kappa2 as double
}
Buffer clear alfa as beta*100
```

See Buffer, Return, Eval(), Eval$()

identifier: TABLE   [APXEIO]

Make a table definition
A name for the table and a list of field triplet...name, kind and size
These are the kinds of fields
BOOLEAN, BYTE, INTEGER, LONG, CURRENCY, SINGLE, DOUBLE, DATEFIELD, BINARY, TEXT, MEMO
' 0 for length means..the database hide the size.

```
TABLE "mine" , "firstTable", "field1", long, 0, "title1", text, 40, "memo1", memo, 0mo1", memo, 0
```

identifier: VIEW   [ΔΕΙΞΕ]

```
VIEW "basename", "basetable", OffsetFrom, OffsetTo
VIEW "basename", "select distinct thisfield from tableone", OffsetFrom, OffsetTo
```

Look MENU command
View insert items to the MENU listbox
USE % INSTEAD * FOR LIKE OPERATOR (M2000 USE ADO NOT DAO ANYMORE)

```
Menu ' clear the listbox
Menu + "<New>"  '   Add something to extend functionality eg. to make a new record here
View "HELP2000", "SELECT [ENGLISH] FROM COMMANDS WHERE GROUPNUM
="+Str$(No(selection))+" ORDER BY [ENGLISH]", 1, many
```

'variable many is filed before...
This is from DB_ENG.GSB the helper program to make this help.

Group: SOUNDS AND MOVIES - ΗΧΟΙ ΚΑΙ ΤΑΙΝΙΕΣ

identifier: BEEP   [ΜΠΙΠ]

Beep (sound as beep)

Beep 32 (window sound 32)

identifier: CHOOSE.ORGAN   [ΕΠΙΛΕΞΕ.ΟΡΓΑΝΟ]

```
CHOOSE.ORGAN  ' this is a list of SYNTHESIZER INSTRUMENTS
IF MENU>0  THEN ORGAN=MENU
```

identifier: MOVIE, MEDIA   [ΤΑΙΝΙΑ]

MOVIE or MEDIA can play avi files

3 way to play movies

1) Play one by one

```
Movie file1,file2  'play file1 then file2
Movie                    ' stop movie
```

2) Play file(s) and when finish resume the command execution

```
Movie  file1,file2;
```

3) Advance Use

```
MOVIE LOAD "SECOND"  ' load movie but not run
MOVIE TO 2                         ' move to 2 second (it is a single)
MOVIE 1000+motion.x,1000+motion.y             '  set the movie window position
MOVIE SHOW                        ' Show the paused movie (we show a static frame)
Wait 100  ' Windows 10 need some time to act so Wait
```

```
A$=GRABFRAME$            ' Grab the frame in A$ as image
COPY 1000,1000 USE A$ ' We can put that frame behind movie (movie has own window, we can't copy there)
MOVIE HIDE                        ' We can hide the movie window
                                          '(but we can't see that because we see the copying frame)
MOVIE TO 3                        ' change to 3 second of the movie
MOVIE 5000+motion.x,1000+motion.y            ' we move the movie window, we can change width and height also
MOVIE SHOW                    ' we show the frame
Wait 100
A$=GRABFRAME$           ' we take it
COPY 5000,1000 USE A$ ' and we print it
MOVIE     ' we close the movie
```

summary of advance commands

```
MOVIE LOAD "FILENAME OR PATH AND FILENAME"
MOVIE PLAY
MOVIE PAUSE
MOVIE HIDE
MOVIE SHOW
MOVIE RESTART
MOVIE TO 0.01
MOVIE left, top
MOVIE left, top, width
MOVIE left, top, width, height
```

We can use 1 and 2 as

```
MOVIE left, top, file1$  [,fileN$]
MOVIE left, top, width , file1$  [,fileN$]
MOVIE left, top, width, height, file1$ [,fileN$]
```

without position the movie is centered to the screen (the basic layer of the environment)
Movie 1.2 same as Movie to 1.2
Movie 0 same as Movie Restart
Movie -1 same as Movie Pause
Feedback:
We can read Duration (a read only variable)
Movie is also a read only variable. It is true if a movie play.
Movie.Counter or Media.Counter or Music.Counter is the same read only variable.
This counter is -1 if nothing is loaded.

identifier: MUSIC [ΜΟΥΣΙΚΗ]

MUSIC is same as MOVIE and MEDIA but without a Window to show something
So MUSIC -1 is a pause
MUSIC 1.2 goto 1.2 second
MUSIC 0 restart music

MUSIC as variable is TRUE if music play
MUSIC.COUNTER is -1 if no music loaded
DURATION is the total duration in seconds (with decimals)
We can play mp3 or MID or WAV

For WAV we have SOUND command also

identifier: PLAY [ΠΑΙΞΕ]

```
Score 1, 500, "c@2dc @2ef"
Play 1, 19
```

See Choose.organ and Score

Play score number and instrument (organ)
We have 16 voices/scores that can play synchronous
If we execute a play command to same score number that was played then we execute one score over the other. That happen because for each score a thread is open and the play command just say to start and use an organ (instrument). So any thread when the time is up send a change to next note or pause to same to a voice with same number as the score have. So two or more threads can send changes to same voice if they have the same score number.

Threads in M2000 are lived in a module and died with that. But those threads that send notes and pauses to synthesizer are not died. We can close the synthesizer with PLAY 0

```
\\ Keyboard - can play two or more notes at once
\\ some keybards can play max 6 notes, some other max 4
dim note1$(10,3), note2$(10,3)
FillArray()
Form 60,30
Pen 14
Cls 5
Double
Report 2, "Keyboard 005"
Normal
Report 2,{Menu
1-Exit  3-Xylophone 4-Piano 5-Saxophone  8-Show Keys/Notes 9-Rythm Yes 0-Rythm No

Space bar - set higher the volume for each note
-- George Karras --

}
Thread.Plan Sequential
Play 0 \\ clear music threads
Global kb$=" ", tempo=300, org=5, f=0, vol$="V90", voi(18), use(17) ' 0 ..17,
n=1
For i=1 to 16 { use(i)=True }
use(10)= false \\ For drum machine
\\ compute virtual clavie position
mm=2*(scale.x div 14)
mm2= mm div 2
kk=row/height*scale.y
kk1=scale.y/height*8
gram=scale.y/height*1.5
Cls , row+9
Module ClKey {
    Read a$, press, sel
    If press Then {
    If Instr(kb$,a$+"-")>0 Then Exit
    Next()
    voi(sel)=f
    Print a$, f
    Score f, tempo, a$+vol$ : kb$<=kb$+a$+"-": Play f, org
    } else {
        kb$<=Replace$(a$+"-","", kb$)
        If voi(sel)>0 Then {
         Play voi(sel), 0
        use(voi(sel))~
        voi(sel)=0
        }
    }
    Sub Next()
    Local i
    For i=1 to 16 {
        If use(i) Then Exit
    }
    If i<17 Then { f<=i : use(i)~ } else f<=1: use(1)=True : Print "!!!!!!!!"
    End Sub
}
\\ 10 for drum machine
Thread { Score 10,400,"CV90CC  ab Cd eCC" : Play 10,1 } as L Interval 60
Thread {
    ClKey "A#2", KeyPress(asc("A")), 1
    ClKey "B2", KeyPress(asc("Z")), 2
    ClKey "C3", KeyPress(asc("X")), 3
    ClKey "C#3", KeyPress(asc("D")), 4
    ClKey "D3", KeyPress(asc("C")), 5
    ClKey "D#3", KeyPress(asc("F")), 6
    ClKey "E3", KeyPress(asc("V")), 7
    ClKey "F3", KeyPress(asc("B")), 8
    ClKey "F#3", KeyPress(asc("H")), 9
    ClKey "G3", KeyPress(asc("N")), 10
    ClKey "G#3", KeyPress(asc("J")), 11
    ClKey "A3", KeyPress(asc("M")), 12
    ClKey "A#3", KeyPress(asc("K")), 13
    \\ https://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx
    ClKey "B3", KeyPress(0xBC), 14 \\ VK_OEM_COMMA
    ClKey "C4", KeyPress(0xBE), 15 \\ VK_OEM_PERIOD
    ClKey "C#4", KeyPress(0xBA), 16 \\ VK_OEM_1
    ClKey "D4", KeyPress(0xBF), 17 \\VK_OEM_2
} as al Interval 50
Print "oк"
Threads
Thread L Interval 6000
Main.Task 100 {
    Display(mm, kk, mm2, kk1)
    Refresh 1000
    If KeyPress(asc("1")) Then Exit
    If KeyPress(asc("3")) Then org<=14 : tempo<=100
    If KeyPress(asc("4")) Then org<=5 : tempo<=300
    If KeyPress(asc("5")) Then org<=65 : tempo<=5000
    If KeyPress(asc("8")) Then n=1-n
    If KeyPress(asc("9")) Then Thread L Restart
    If KeyPress(asc("0")) Then Thread L Hold
    If KeyPress(32) Then { vol$ <= "V127"  } else vol$ <= "V90"
    Print "--------"
}
Threads Erase
Print "END"
Sub Display(p0,y0, p1, y1)
    Clavie(p0, y0, p1, y1, 0, &note1$())
    Clavie(p0-p1/2,y0,p1,y1*2/3, -1, &note2$())
    Refresh 1000
End Sub
Sub Clavie(p0, y0,p1, y1, p3, &n$())
    Link n$() to n()
    p3-!
    Local k=-1, i
    For i=p0 to 9*p1+p0 step p1
    k++
    If n$(k,0)<>"" Then {
        Move i+p3*p1/6, y0
        If p3 Then {
            Fill p1-p3*p1/3-15,y1-15, 7* (1-(voi(n(k,2))>0))+1, 0,1
        } else {
            Fill p1-15,y1-15,15,7* (1-(voi(n(k,2))=0))+1,1
        }
        Move i+p3*p1/6, y0
        Fill @ p1-p3*p1/3,y1,2,1
        Move i+p3*p1/6, y0+y1-gram
        Pen p3*15 { Fill @ p1-p3*p1/3,gram,5,n$(k,n)}
    }
    Next i
End Sub
Sub FillArray()
Local n,p, n$, k$, no
Stack New {
    Data "A#2", "A", 1, 1, 2
    Data "B2", "Z", 2, 1, 1
    Data "C3", "X", 3, 2, 1
    Data "C#3", "D", 4, 3, 2
    Data "D3", "C", 5, 3, 1
    Data "D#3", "F", 6, 4, 2
    Data "E3", "V", 7, 4, 1
    Data "F3", "B", 8, 5, 1
    Data "F#3", "H", 9, 6, 2
    Data "G3", "N", 10, 6, 1
    Data "G#3", "J", 11, 7, 2
    Data "A3", "M", 12, 7, 1
    Data "A#3", "K", 13, 8, 2
    Data "B3", ",", 14, 8, 1
    Data "C4", ".", 15, 9, 1
    Data "C#4",";", 16 , 10, 2
    Data "D4", "/", 17, 10, 1
    While Not Empty {
    Read n$, k$, no, n, p
        If p=1 Then {
           note1$(n-1,0):= n$, k$, no
        } else {
             note2$(n-1,0):= n$, k$, no
        }
    }
}
End Sub
```

identifier: SCORE   [ΦΩΝΗ]

```
 SCORE 3, 1000, "C5F#@2B@2C5F#@2B"
    SCORE 1, 1000, "D@2E@2C#3 @2D5@2V90 @3F#4V127"
               '/ C C# D D# E F F# G G# A# B
               '/
    PLAY   1, 19, 3, 22  ' VOICE, INSTRUMENT
```

for score 3 we have 1000 miliseconds period for a note, so we can use @2 for 1000/2 until @6 for 1000/32 as periods

C5 is C in 5th octave

V is for volume V1 to V127

We can place a space for pause and we can give a time period for that as for notes.

Each score is copied to a bank so a new copy can change the bank

When a bank is in use each reading is droping from bank. When bank is empty that voice is stopped.

We can change the bank before the reading is over. We can't read the bank, only we can

write it.
    PLAY 0 for erasing synthesizer threads, and empty all the banks.
    Use VOLUME for total volume control, CHOOSE.ORGAN to find an instrument

identifier: SOUND   [ΗΧΟΣ]

play wav
SOUND filename$
SOUND ""   Stop sound

identifier: SOUNDREC   [ΗΧΟΓΡΑΦΗΣΗ]

1) SoundRec   ' begin to record using 8bit, 11025khz, 1 channel.
2) SoundRec New "filename.wav", 41000 stereo hifi   ' just set, use Soundrec insert to start recording
          stereo for 2 channels, hifi for 16bit
  without stereo we get 1 channel, without Hifi we get 8 bit
  we can use 11025, 22050, 41000 or other values for sampling
3)SoundRec Insert
  from the current position
4)SoundRec OverWrite
  record from the begin
5) SoundRec Delete 10 to 20
  cut from 10ms to 20ms
6) SoundRec Stop
  stop the recording
7) SoundRec Test
    play what we have at that moment (stop it then start the test)
8) SoundRec Pos 10
          in a recording session we move the "tape" to 10ms from the start and then continue recording
          in stopped session we move the "tape" to 10ms from the start
9) SoundRec Save
  we use the name we place in New....
10) SoundRec Save "filename.wav"

21) SoundRec End
free resources. Also close the soundrec.level (which make noise if we play sound among this)

```
Volume 100
Soundrec new dir$+"alfa.wav", 22050 STEREO HIFI
Profiler   ' start high resolution timer (used with Timecount)
Print "Start Recording"
Soundrec Insert
Do
        Print Over $(1), String$("|", Soundrec.level)
        Print Part  $(3, Width), Timecount Div 1000
        Refresh 100
        if keypress(32) Then Exit
        Wait 1
Always
m=Timecount
Soundrec stop
Print over
Print "Recording stop - now play Test"
Volume 100
Profiler
        Soundrec test
Do
        Wait 1
Until Timecount>m
Print "Test Play Stop"
Soundrec Save
Print "Recording saved"
Soundrec End
```

identifier: SPEECH   [ΛΟΓΟΣ]

```
SPEECH "I can speak english"
SPEECH "I am a boy"!
SPEECH "SPELLING"#

SPEECH "I can speak english", voice
SPEECH "I am a boy"!, voice
SPEECH "SPELLING"#, voice
```

```
print speech
```

prints number of voices

```
for i=1 to speech {
     print speech$(i)
}
```

print name of all voices

identifier: TONE   [ΤΟΝΟΣ]

Not for simulator, but in a real Windows Machine.
1) Tone   play 1khz for 1/10 of a second
2) Tone 200 play 1khz for 0,2 of a second
3) Tone 200, 5000 play 5khz for 0,2 of a second
Tone is generated from hardware and Windows Kernel stop any service for threads in M2000.
So this command aren't suitable for thread programming

```
Module A {Module AA {
For i=1 To 10 {
Tone 50, 150
Tone 25, 300
}
}
For i=1 To 5 {
AA
Wait 200
AA
Wait 800
}
}
Module B {For i = 1 To 10 {
TUNE "cac "
Wait 200
}
}
Module C {For i=100 To 8000 Step 20 {
```

```
Tone 2,i   'play
}
}
```

Check a thread (copy to a module)

```
I=0
THREAD {
PRINT I
I=I+1
} AS L INTERVAL 40
WAIT 1000
TONE 1000,3000
WAIT 1000
```

identifier: TUNE   [ΜΕΛΩΔΙΑ]

Using tone from KERNEL (if running in an original Windows Machine, no a virtual)

```
Tune melody$
Tune duration, melody$
```

```
Module B {
    For i = 1 To 10 {
        TUNE "cac "
        Wait 200
    }
```

```
Tune "C3BC#"
```

Number is the Octave, used to change current octave, Space is Pause.

Use VOICE and PLAY for sending MIDI messages to internal synthesizer (work on virtual machines) identifier: VOLUME   [ΕΝΤΑΣΗ]

Sound Volume

```
Volume 100 ' maximum
Volume 0 ' minimum
```

```
Volume 100,50   (same as Volume 100)
Volume 100, 0 ' Right Ch. 100%
Volume 100, 100 ' Left Ch. 100%
Volume 100, 25  ' Right Ch. 100%, Left Ch. 50%
Volume 100, 75 ' Right Ch. 50%, Left Ch. 100%
Volume 50, 25  ' Right Ch. 50%, Left Ch. 25%
Volume 50, 75  ' Right Ch. 25%, Left Ch. 50%
Volume 25, 50 '  Right Ch. 50%, Left Ch. 50%
```

Group: MOUSE COMMANDS - ΕΝΤΟΛΕΣ ΔΕΙΚΤΗ

identifier: JOYPAD   [ΛΑΒΗ]

```
JOYPAD 0,1  ' use these two joypads
```

```
JOYPAD  ' with no parameters no joystick is in use
```

Four functions can be used for joypads

```
Print Joypad(0)  ' read all buttons in onw binary number
Print Joypand.direction(0)
one of that
   DirectionNone = 0
   DirectionLeft = 1
   DirectionRight = 2
   DirectionUp = 3
   DirectionDown = 4
   DirectionLeftUp = 5
   DirectionLeftDown = 6
   DirectionRightUp = 7
   DirectionRightDown = 8
 Print Joypad.analog.X(0) , joypad.analog.Y(0)
 you have to calibrate before use analog values
```

identifier: MOUSE.ICON   [ΔΕΙΚΤΗ.ΜΟΡΦΗ]

```
MOUSE.ICON 2

MOUSE.ICON DIR$+"THAT.ICO"

MOUSE.ICON SHOW
MOUSE.ICON HIDE
```

Change the mouse icon. You can hide mouse if you give a total transparent icon

Group: BROWSER COMMANDS - ΕΝΤΟΛΕΣ ΙΣΤΟΥ

identifier: BROWSER   [ΑΝΑΛΟΓΙΟ]

We can open html pages
1) open and show in standard center position smaller than the screen
BROWSER "www.google.com"
2) open and show using twips
BROWSER "www.youtube.com",1000,1000,5000,3000
3) Cancel and close
BROWSER ""
4) Reading from temporary folder
BROWSER "alfa"
is equal to BROWSER temporary$+"alfa.html"

Example
We make a temporary html file, then we open it in with browser and then we get info from the form in the html
Copy these lines in a module and run it

```
show
username$ = "George"
text alfa.html {<!doctype html public "-//w3c//dtd html 3.2//en">
<html>
<head>
<title>(Type a title for your page here)</title>
<meta name="GENERATOR" content="M2000">
<meta name="FORMATTER" content="M2000">
<meta content="this, other, that" name=keywords>
<meta content=All name=robots>
<meta HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=windows-1253">
<meta NAME="Author" CONTENT="GEORGE KARRAS">
<style type="text/css">
body {
   overflow:hidden;
}
</style>
</head>
<body bgcolor="CYAN" text="#000000" link="#0000ff" vlink="#800080" alink="#ff0000">
<CENTER>This is an Html  form</CENTER></H1><HR>
<script> var m1;var p1;var total;function alfa() {
total="?onoma=2000";total=total+"&username="+escape(m1);total=total+"&password="+escape
(p1);document.title=total;setTimeout ("alfa()",100);return true}</script>
<FORM NAME="ValidForm" action="about:blank" >
<INPUT NAME="onoma" TYPE="HIDDEN" MAXLENGTH=4 SIZE="4" VALUE="1000">
<TABLE ALIGN=CENTER BORDER=0 RULES=NO COLS="2">
<TBODY>
<TR VALIGN=TOP  >
<TD  >User name:</TD>
<TD  ><INPUT NAME="username" TYPE="TEXT" id="uname" MAXLENGTH=16 SIZE="16"
VALUE="##username$##" ></TD>
</TR>
<TR VALIGN=TOP  >
<TD  >This is our password:</TD>
<TD  ><INPUT NAME="password" ID="pass" TYPE="PASSWORD" MAXLENGTH=8
SIZE="8"></TD>
</TR>
</TABLE>
<CENTER><BR><HR>
<input type=button value="Inform me" onclick="m1=uname.value;p1=pass.value;alfa()">
</CENTER>
</FORM>
</body>
</html>
}
onoma$=""
browser alfa.html
while onoma$=""  {refresh}
list   ' just show all variables
browser ""
text alfa.html  ' delete the temporary file
```

Print "OK"

identifier: TEXT, HTML   [KEIMENO]

Save or Append formatted text (we can pass variables) to a file in temporary folder as log or html for browser.

From 7 edition there are 2 modifiers,
utf-8 and utf-16 by default (without modifier we save in ANSI)
Text utf-8 name.txt {line 1
}

1)
1,1 Using a variable (also can be used that type in 2,3,4,5
a$="alfa.txt"
TEXT alfa.txt { lines of text
second line}
a file alfa.txt is created to temporary$ (the %temp% folder)
1.2 using a filename inline
TEXT alfa.txt { lines of text
second line}
a file alfa.txt is created to temporary$ (the %temp% folder)

2)TEXT alfa.txt + { lines of text
second line}
we can append lines

Example for 1 & 2

```
        alfa$="ok"
        b=1
        text t123.txt {First line, we write some variables here "##alfa$##", ##b##
            Second Line
            Third Line. Text Command insert a CRLF to the last line
            }   '  from last bracket position excluded the leading spaces from second line and after
        text t123.txt + {We append some lines here
            And that line too
            }  '  from last bracket position excluded the leading spaces from second line and after
        ' Now we can look if file exist
```

```
          ' temporary files are deleted at the program end
          print exist(temporary$+"t123.txt")
          '
          open temporary$+"t123.txt" for input as k
          while not eof(k) {
               line input #k, a$
               report a$     ' display line as paragraph here
          }
          close #k
          text t123.txt ' we erase the file
```

3) We can write html files and open with BROWSER (no example here)
so we can replace TEXT to HTML and not provide type (provided automatic)

4) We can set up html forms and we get parameters in variables/

We use a simple statement like this: while onoma$="" {refresh}
For waiting the "send" button to send information back to us
Below in the example we make a variable username$ with a value. We have in the text below this ##username$##. This will replaced by the value of username$ before saving to a temporary file.
M2000 take the sending string from page and make variables (if no name found) and assign to them values .
password$ is a variable who created when while loop (above) exit. And exit the loop because the variable onoma$ take a value.
This is a very old code (more than 10 years) but can run well. The meta tags are for joke. And the title also never displayed.

Put this in a module

```
show
username$ = "George"
text alfa.html {<!doctype html public "-//w3c//dtd html 3.2//en">
               <html>
               <head>
               <title>(Type a title for your page here)</title>
               <meta name="GENERATOR" content="M2000">
               <meta name="FORMATTER" content="M2000">
               <meta content="this, other, that" name=keywords>
               <meta content=All name=robots>
               <meta HTTP-EQUIV="Content-Type" CONTENT="text/html;
charset=windows-1253">
               <meta NAME="Author" CONTENT="GEORGE KARRAS">
```

```
        <style type="text/css">
        body {
           overflow:hidden;
        }
        </style>
        </head>
        <body bgcolor="CYAN" text="#000000" link="#0000ff" vlink="#800080"
alink="#ff0000">
        <CENTER>This is an Html  form</CENTER></H1><HR>
        <script> var m1;var p1;var total;function alfa() {
total="?onoma=2000";total=total+"&username="+escape(m1);total=total+"&password="+escape
(p1);document.title=total;setTimeout ("alfa()",100);return true}</script>
        <FORM NAME="ValidForm" action="about:blank" >
        <INPUT NAME="onoma" TYPE="HIDDEN" MAXLENGTH=4 SIZE="4"
VALUE="1000">
        <TABLE ALIGN=CENTER BORDER=0 RULES=NO COLS="2">
        <TBODY>
        <TR VALIGN=TOP  >
        <TD  >User name:</TD>
        <TD  ><INPUT NAME="username" TYPE="TEXT" id="uname" MAXLENGTH=16
SIZE="16" VALUE="##username$##" ></TD>
        </TR>
        <TR VALIGN=TOP  >
        <TD  >This is our password:</TD>
        <TD  ><INPUT NAME="password" ID="pass" TYPE="PASSWORD"
MAXLENGTH=8 SIZE="8"></TD>
        </TR>
        </TABLE>
        <CENTER><BR><HR>
        <input type=button value="Inform me"
onclick="m1=uname.value;p1=pass.value;alfa()">
        </CENTER>
        </FORM>
        </body>
        </html>
        } '  from last bracket position excluded the leading spaces from second line and after
onoma$=""
browser alfa.html
while onoma$=""  {refresh}
list
browser ""
text alfa.html  ' delete the temporary file
Print "OK"
```

5) text alfa.html
delete file in temporary$
We can delete any file in temporary$ (%temp%) if we write simple the name of it after text.
M2000 has no direct delete command except for the temporary folder.

Group: COMMON DIALOGUES - ΚΟΙΝΕΣ ΦΟΡΜΕΣ

identifier: CHOOSE.COLOR   [ΕΠΙΛΕΞΕ.ΧΡΩΜΑ]

1) CHOOSE.COLOR
    default dialog for color picking

2) CHOOSE.COLOR color
    like 1 but display our choice

use arrows left right to rotate columns, so the most left can change by one...as we move up down the color list.
slide the bottom box to send color (find in the stack)

identifier: CHOOSE.FONT   [ΕΠΙΛΕΞΕ.ΓΡΑΜΜΑΤΟΣΕΙΡΑ]

CHOOSE.FONT
Open a custom dialog to choose font. Dialog has 4 zones. Top zone has the title dialog, we can move, or we can close clicking the square icon. Next zone is a list box with font names. We can scroll by pushing up or down, holding down left mouse button. Third zone hold attributes editor. We can scroll 3d zone and we can check bold and or italics and edit numbers size and charset, or we can choose from presets. Last zone is for preview and testing. With tab we change from 2 to 3 to 4 zone and in 4th we can press any key to open the editor. We can shift right the 4th zone as OK or just press Enter. Use square at the top for Cancel. Also the dialog is sizable, using the down or right border, or down and right corner.
All the

```
MODULE A  {

    MODULE SAY {
        READ K
        IF K=0 THEN {
            PRINT "NO"
        } ELSE {
            PRINT "YES"
        }
    }

    CHOOSE.FONT

    IF NUMBER>0 THEN {
        READ A$, S, C, I, B
        PRINT "FONT:";A$
        PRINT "SIZE:";S
        PRINT "CHARSET:";C
        PRINT "ITALIC:";
        SAY I
        PRINT "BOLD:";
        SAY B
    } ELSE {
        PRINT "NOTHING..."
    }
}
```

identifier: DIR   [ΚΑΤΑΛΟΓΟΣ]

set the current directory to user dir

```
DIR USER
```

set any

```
DIR "C:\"
```

set relative , use ..\ or ..\..\ or more to go up one or more time then go down to this folder
If we omit the last slash, automatic putting by the command

```
DIR "..\this"
```

Open dialog (using top folder the previous topfolder or the user dir)

```
DIR ?
```

Open dialog and we can put a new folder

```
DIR ? "?"
```

Open dialog and we can set top folder

```
DIR ? "C:\"
```

Open dialog and set top folder and set a caption and we can put a new folder

```
Dir ? "C:\?" , "For Output"
```

or with two lines

```
DIR {For Output
------------ } ? "C:\?"
```

In any combination we can use TO S$ for store the path in a variable and not make that the current directory

```
' We can't open twice the file/folder selector but
' when we open dialog threads allowed to run in background
' Statement TRY is for error trapping. If error occur in the block after TRY then the variable (here OK) get 0 or FALSE.
' OK is a variable that TRY make (TRY make new variable as READ do, numeric only)

DIR USER
AFTER 200 {
    PRINT "CHECK"
    TRY OK {
            DIR ?
    }
    IF NOT OK THEN PRINT ERROR$
}
DIR ?
```

identifier: OPEN.FILE [ΑΝΟΙΓΜΑ.ΑΡΧΕΙΟΥ]

We can set the Top Folder. We can go up further than the Top Folder
All file dialogues are safe for use because we can't delete or move file or folder, we can't execute any file from it.

1. OPEN.FILE ' open a dialog for file(s) selection. Version 7 have a new custom dialog
2. OPEN.FILE aName$
3. OPEN.FILE aName$, folderOrIndirectPath$

4. OPEN.FILE aName$, folderOrIndirectPath$, titleOfDialog$
5. OPEN.FILE aName$, folderOrIndirectPath$, titleOfDialog$, fileTypes$
Result we found in the Stack
Escape or a click to a square at top left corner have the result an empty string

we can omit some parammeters..
folderOrIndirectPath$ is the Top Folder (we can use "*" as the Computer Level, where we can see anything inlcuding the common folders in the LAN)

```
OPEN.FILE ,,"Load Text File","txt"
```

The new selector always display the path without hidden parts, by using wrapping in slash and space can wrap the path. The new file selector change size and zoom, by dragging the lower right corner. From setup we can see all the files from all folders under the current, and we can open the multiselection option, and we can change the type of sorting.

If we press letters on keyboard, these accumylate the searching filename and we see the finding in progress. Move arrows up down or change chosen one and this reset the search string.

We can give "..\media\" as an indirect path for folderOrIndirectPath$. If folder doesn't exist an error occur and we see no file selector.
In fileTypes$ we can give file types separate using "|". This is the new way, but selector use the old too (by refine it, holding only strings to match the file types). The old is from standard selector "Bitmap (*.bmp)|*.bmp|Photo (*.jpg)|*.jpg".

In titleOfDialog$ we can pass a multine text but is not suppored by that file selector. If we give a three line string or document, we will see the middle line (all titles centered)

Using the current dir as topfolder.
We can use function FILE$() or FILE$(title$) or FILE$(title$,"TXT")   'or we can put | to make multy type selection

```
A$=FILE$("IMAGE A","JPG")
```

identifier: OPEN.IMAGE   [ΑΝΟΙΓΜΑ.ΕΙΚΟΝΑΣ]

We can set the Top Folder. We can go up further than the Top Folder
All file dialogs are safe for use because we can't delete or move file or folder, we can't execute any file.

1. OPEN.IMAGE ' open a dialog for file(s) selection. Version 7 have a new custom dialog
2. OPEN.IMAGE aName$
3. OPEN.IMAGE aName$, folderOrIndirectPath$
4. OPEN.IMAGE aName$, folderOrIndirectPath$, titleOfDialog$
5. OPEN.IMAGE aName$, folderOrIndirectPath$, titleOfDialog$, fileTypes$

we can omit some parammeters..

OPEN.IMAGE ,,"Load Photo","jpg"

Like OPEN.FILE but can preview images on the file selector

By default this is the fileTypes$ "BMP|JPG|GIF|WMF|EMF|DIB|ICO|CUR"
We can open at the Computer Level (showing the drives and the LAN common folders)
OPEN.IMAGE ,"*"

see OPEN.FILE for details.

identifier: SAVE.AS   [ΑΠΟΘΗΚΕΥΣΗ.ΩΣ]

1. SAVE.AS ' open a dialog for saving a file. Version 7 have a new custom dialog
2. SAVE.AS aName$
3. SAVE.AS aName$, folderOrIndirectPath$
4. SAVE.AS aName$, folderOrIndirectPath$, titleOfDialog$
5. SAVE.AS aName$, folderOrIndirectPath$, titleOfDialog$, fileTypes$
Result we found in the Stack

folderOrIndirectPath$ is the top folder.
In M2000 we can't create folder directly from code except by use of OS command (we can send command to OS...if we allowed to do something like this).
So if we need a folder to write then:
1. We can use temporary folder as Temporary$....DIR TEMPORARY$ or using it in OPEN command
2. We can use USER folder..by the command DIR USER   'note here we are not use a variable a USER$ but an ID that DIR expect.
3. We can make a folder by the user action...using the command DIR to open a folder selector with folder creation capability. So we can set a top folder and allow the user to insert a folder there.

identifier: SETTINGS   [ΡΥΘΜΙΣΕΙΣ]

Settings
Open a form to make settings like font, name and size and bold on/off, pen color and screen color, case sensitive or not for file names.

identifier: SUBDIR   [ΥΠΟΚΑΤΑΛΟΓΟΣ]

```
Try {
    SUBDIR ALFA
}
DIR  ALFA
```

Create ALFA if not exist, and then we set ALFA as current Directory.

identifier: TITLE   [ΤΙΤΛΟΣ]

```
TITLE "a label for taskbar",  1 or 0
```

by default without second parameter is 1 (showing the form of program)
With 0 Form is minimized in taskbar.

TITLE with no parameters erase the presence in taskbar.

The taskbar label is useful because you can utilize Alt Tab, to switch to M2000, and you can perform by the context menu minimize or restore to original size, the form of M2000, without stopping the running of any application.
You can run a M2000 program without showing UI, because all programs start with minimized option, as a hidden UI, and when they execute a SHOW or need the input of a user then can open the hidden UI (user interface). Dialogues can open without showing the main form. There is a msgbox for asking for a OK or Cancel, see ASK(and look for Dialogues to see what else you can open, without showing ui.

Additional commands

with ICON you change the icon of program that showing in taskbar.

Group: ARITHMETIC FUNCTIONS - ΑΡΙΘΜΗΤΙΚΑ

identifier: #END([#ΤΕΛ(]

```
a=(1,2,3,4)
b=(1,2,3)
? a#start(b)#end((100,))
// 1 2 3 1 2 3 4 100
```

identifier: #EVAL([#ΕΚΦΡ(]

Used for array pointers (or tuples). Can execute a lambda function, or a group which return a value. Also can used to access items by order for arrays, for stack and inventory objects. When return an array simple return the pointer (#val() return a copy).
For string values we use the #eval$(). But be careful when we apply multiple #statements we can use a #xx$() statement as the last one.

```
m=(1,2,(100,(200,201,202),300),4)
Print m#eval(2,1,2)=202
Print m#val(2)#val(1)#val(2)=202
class alfa {
	value (x){
		=500*x
	}
}
m=(1,2,(lambda (x)->500*x), alfa())
Print m#eval(2,3)=500*3, m#eval(3,3)=500*3
inventory queue k=1,2,3:=300,4
m=(1,2,k)
Print k(3)=300, k(2!)=300, m#eval(2,2)=300
\\ this type of inventory dosn't guarantee the item position
inventory k1=1,2,3:=1000,4,5
```

```
m=(1,2,k1)
delete k1, 2, 4
If exist(k1, 3) then k1_item_pos=eval(k1!)
Print m#Eval(2,k1_item_pos)=1000
\\ to access by key from m we need a link to the second array interface
link m to m()
Print m(2)(3)=1000
```

identifier: #EXPANSE([#ΑΝΟΙΓΜΑ(]

```
a=(,)
? a#expanse(100)#mat("=", 1)#sum()=100
```

identifier: #FILTER([#ΦΙΛΤΡΟ(]

```
a=(1,2,3,4,5,6,7,8,9)
fold1=lambda ->{
     push number+number
}
even=lambda (x)->x mod 2=0
b=a#filter(even, (,))
Print b ' 2 4 6 8
Print a#filter(even)#fold(fold1)=20
```

identifier: #FOLD([#ΠΑΚ(]

```
a=(1,2,3,4,5,6,7,8,9)
fold1=lambda ->{
     push number+number
}
Print a#fold(fold1)=a#sum()
Print a#fold(fold1,10)=a#sum()+10
```

look #filter(), #map(), #fold$(), #pos()

identifier: #HAVE([#ΕΧΕΙ(]

```
PRINT (1,2,3,4)#HAVE(1,2)=TRUE
PRINT (1,2,3,4,1,2)#HAVE(3->1,2)=TRUE ' from 4th
PRINT (1,2,3,4,1,2)#HAVE(0->(1,2))=TRUE ' from 1st
```

identifier: #MAP([#ΑΝΤ(]

```
a=(1,2,3,4,5,6,7,8,9)
fold2=lambda ->{
	Shift 2
	Push letter$+"-"+letter$
}
odd=lambda (x)->x mod 2=1
b=a#Filter(odd, (,))
Print b ' 1,3,5,7,9
map1=lambda ->{
	Push chrcode$(number+64)
}
z=a#Filter(odd)#Map(map1)
Print z  ' A C E G I
Print "["+a#Filter(odd)#Map(map1)#Fold$(fold2, "CODE")+"]"="[CODE-A-C-E-G-I]"
```

identifier: #MAT([#ΜΑΖΙ(]

```
 a=(1,2,3,4,5)
? a#mat("+=", 100)#str$()="101 102 103 104 105"
```

identifier: #MAX([#ΜΕΓ(]

```
a=(1,2,3)
Print a#max()=3

z=-1
print (1,2,3,-3)#max(z)=3, z=2
```

identifier: #MIN([#ΜΙΚ(]

```
Print (1,2,3)#min()  ' we can't use sign or put it in expression (unless in first place)
```

```
a=(1,2,3)
Print -a#min()  ' but this we can use it everywhere
Dim a(3)
a(0)=1,2,3
Print a()#min()

z=-1
Print a()#min(z)=1, z=0  ' position in a()
```

identifier: #NOTHAVE([#ΔΕΝΕΧΕΙ(]

```
PRINT (1,2,3,4)#NOTHAVE(1,2,5)=TRUE
PRINT (1,2,3,4,1,2)#NOTHAVE(3->2,3)=TRUE ' FROM 4th
PRINT (1,2,3,4,1,2)#NOTHAVE(0->(2,2))=TRUE ' FROM 1st
```

identifier: #POS([#ΘΕΣΗ(]

Uou can't mix strings and numbers in search list

```
c=(1,1,0,1,1,1,1,0,1,1,0)
Print c#pos(1,0,1) ' 1  means 2nd position
Print c#pos(5->1,0,1) ' 6  means 7th position
c=("hello","1", "hello","2")
Print c#pos(0->"hello", "2")=2 ' 3rd position
```

identifier: #REV([#ΑΝΑΠ(]

```
Print (1,2,3)#rev()
    3     2     1
```

identifier: #SLICE([#ΜΕΡΟΣ(]

```
a=(1,2,3,4,5)
Print a#slice(1,2)  '  2 3
Print a#slice(1,2)#rev()  ' 3 2
Print a#slice(2) ' 3 4 5
Print a#slice(,2) ' 1 2 3
```

identifier: #SORT([#ΤΑΞΙΝΟΜΗΣΗ(]

```
Print (10,3,4,1)#SORT(1)
10 3 4 1
Print (10,3,4,1)#SORT()
1 4 3 10
Print (10,3,4,1)#SORT(0,1,3)
10 1 3 4
```

identifier: #START([#ΑΡΧΗ(]

```
a=(1,2,3,4)
b=(1,2,3)
? a#start(b)#end((100,))
// 1 2 3 1 2 3 4 100
```

identifier: #STUFF([#ΥΛΙΚΟ(]

Return value as is (not as #VAL() which change the value of string type, see example)

```
CLASS ALFA {
          X=10
}
DIM A(10)
Z=POINTER(ALFA())
A(3):="OK", 100, (1,2,3,4), Z
M=A()
? M#STUFF(6).X
Z=>X++
? M#STUFF(6).X
ZZ=M#STUFF(6)
? Z IS ZZ
? A(3)
? M#STUFF(3)="OK", M#VAL(3)=0, M#VAL$(3)="OK"
? M#STUFF(4)=100, M#VAL(4)=100, M#VAL$(4)="100"
? M#STUFF(5)#MAX()
```

identifier: #SUM([#ΑΘΡ(]

```
Dim A(10)=1
Print A()#sum()
k=(1,2,3,4)
print k#sum()
\\this is ok
m=k#sum()+k#sum()
```

This is an error m=(1,2,3,4)#sum()+(1,2,3,4)#sum()   ' after operators we can't place an auto array.

```
\\these are ok
m=(1,2,3,4)#sum()
m=(1,2,3,4)#sum()+10
Print (1,2,3,4)#sum()
```

identifier: #VAL([#ΤΙΜΗ(]

```
a=(1,2,3)
Print a#val(0)=1
```

identifier: ABS([ΑΠΟΛ(]

```
PRINT ABS(-5)+5
    10
```

absolute value

For complex numbers

```
Print Abs((1,-2i))=Mod((1,-2i))
```

identifier: ADDRESSOF   [ΔΙΕΥΘΥΝΣΗΑΠΟ]

Look module()

The modules which called from machine code, are like code of the module where we call the machine code. The code of those modules executed using the same scope (also anything we make on these modules didn't deleted at the exit).

Read the example:

```
Cls
far=if(rnd>.5->false, true)
Print if$(far=true->"Using dec ecx && jnz rel32","Using loop rel8")
clip=false
```

```
Print "Use this disassembler:"
Print "https://defuse.ca/online-x86-assembler.htm#disassembly2"
// Print monitor.stack  (display size in use of return stack )
// Print monitor.stack.size  (the pre-allocate return stack size, >30MByte)
// Version 11 Revision 15
Buffer clear BinaryData as Long*10
Return BinaryData, 1:=500
Buffer code clear alfa as byte*1024

module beta {
        print :print "done"
}
module beta2 {
        print eval(BinaryData,0),
}
Pc=0
Code_Prolog()
Op(0x51, 0x31, 0xC0) ' push ecx : xor eax, eax
OpLong(0xb9, 100)  ' mov ecx, 100
mLoop=pc  // need that
op(0x01, 0xc8) ' add eax, ecx  (eax=eax+ecx)
OpLong(0xa3, BinaryData(0))  ' mov ds:BinaryData(0), eax  ' copy to data section
        Op(0x51, 0x50) ' push ecx: push eax
        CallModule(AddressOf beta2)
        Op(0x58, 0x59) ' pop eax : pop ecx
if far then
        op(0x49) ' dec ecx
        Loop2(mLoop)
else
        Loop(mLoop)  // 2 bytes (dec ecx)
end if
CallModule(AddressOf beta)
Op(0x59, 0x31, 0xC0) ' pop ecx : xor eax,eax

Ret()
document doc$
for i=0 to pc-1
doc$=hex$(eval(alfa, i),1)
next
Print "Code:";doc$
if clip then clipboard doc$: exit
Try Ok {
    profiler
    Execute Code alfa, 0
    print timecount
}
M=Uint(Error)
Print eval(BinaryData,0)
Print monitor.stack
End
Sub Op()
    while not empty
        Return alfa, pc:=number
        pc++
    end while
End Sub
Sub OpLong()
    Return alfa, pc:=number, pc+1:=number as long
    pc+=5
End Sub
Sub Code_Prolog()
    op(0x58, 0x59, 0x59, 0x59, 0x59, 0x50)
    End Sub
Sub Ret()
    Return alfa, pc:=0xC3
    pc++
End Sub
// module() return the real address of module calling function.
// The AddressOf number passed as parameter to this function
// so we Push the AddressOf modulename1
// and then we call code at address module() (using relative addressing)
Sub CallModule()
    Return alfa, pc:=0x68, pc+1:=number as long
    pc+=5
    Return alfa, pc:=0xE8, pc+1:=module()-pc-5-alfa(0) as long
    pc+=5
End Sub
// 0xE2 cb      LOOP rel8     D       Valid   Valid   Decrement ECX ; jump short if count ≠ 0.
// 0xE1 cb      LOOPE rel8    D       Valid   Valid   Decrement ECX ; jump short if count ≠ 0
and ZF = 1.
// 0xE0 cb      LOOPNE rel8   D       Valid   Valid   Decrement ECX ; jump short if count ≠ 0
and ZF = 0.
Sub Loop()
    Return alfa, pc:=0xe2, pc+1:=(number-pc-2+256)
    pc+=2
End Sub
```

```
// 0x49                    dec    ecx  // OF, SF, ZF, AF, and PF flags are set according to the result.
// 0x85 jnz rel32
// by default this (number-pc-5+0xFFFFFFFF) is type of modulo 2^32  (same as binary.add())
Sub Loop2()
     Return alfa, pc:=0x0F,pc+1:=0x85, pc+2:=(number-pc-5+0xFFFFFFFF) as long
     pc+=6
End Sub
```

identifier: AND   [ΚΑΙ]

Operator AND

```
Print True And False
```

identifier: ARG([ΟΡΙΣΜΑ(]

return the argument of a complex number
(same value of PHASE() function)

identifier: ARRAY([ΠΙΝΑΚΑΣ(]

```
Dim a(5,5)=1
Print Array("a",1,3)
Print a(1,3)

\\ look array$()

Dim a(10)
Function a {
     Read  a
     =a*2
}
a(5)=3
Print a(5), array("a",5)  \\ 3 3
Print a(* 5)   ' 10   (use * to indicate that a() is a function)


a=(1,2,3,"text",5)
Print  Array(a, 0)  ' 1
Return a, 0:=1000, 4:="text b"
Print a, Array(a,0), Array$(a,4)
Append a, (6,7,8,9)
```

```
Print Array(a, 8)
Print Len(a)
Link a to a$()
Print a$(4)
Print array(a$(), 1)
```

identifier: ASC([ΚΩΔ(]

```
Print ASC("A")
```

return ascii code
See CHRCODE()

identifier: ASK([ΡΩΤΑ(]

```
PRINT ASK("this is a messagebox")
IF ASK("Info","Title","okTitle", "CancelTitle", Bitmap$) =1 then print "ok"
```

1. There is ASK$() also with same parameters but return string (ok or cancel title)
2. We can use it for input string (maximum 100 chars)

```
IF ASK("Info","Title","okTitle", "CancelTitle", Bitmap$, "string to input") =1 then print "ok" : read A$
```

now A$ hold the response from ASK. If Ask return 2 (cancel) then no string pushing in stack, so we can't read from stack.
3. Bitmap$ is a string that we can load a bitmap, or a filename to a picture (internal use vb6 LoadPicture)

All parameters are optional except the first one.

6 parameters
Text to Ask
Title
Text for Ok, use "*" as first character to make it default control, use only "*" to make it default control with default caption
Text for Cancel use "*" as first character to make it default control, use "*" only as the one before, and "" to not show
String for path or bitmap data
String for make Message box an InputBox.

identifier: ATN([ΤΟΞ.ΕΦ(]

```
Print Atn(1)  ' degrees
Print rAtn(1)  ' radians
```

return complex if we pass a complex number

identifier: BACKWARD([ΠΙΣΩ(]

```
Document a$={aaaaaaa
bbbbbbbbb
ccccccc
}
m=Paragraph(a$, doc.par(a$))
If backward(a$,m) then {
     While m {
          Print Paragraph$(a$,(m))
     }
}
```

identifier: BANK([ΤΡΑΠ(]

Banker's Rounding Same as Round() in Vb6

identifier: BIGINTEGER([ΜΕΓΑΛΟΣΑΚΕΡΑΙΟΣ(]

```
a=BigInteger("686479766013060971498190079908139321726943530014330540939446345918
55431833976560521225596406614545549772963113914808580371219879997166438125740
4028291115057151")
b=BigInteger("162259276829213363391578010288127987979798")
c=a*b
d=b*c
c=c+500u
Print "a=";a
Print "b=";b
Print "c=";c
d=c/b
Print "d=";d
Print "c/b modulus"; Mod(d)
Print "compare d=500";Mod(d)=500u
Method a, "AnyBaseInput", "FFFFFFFF", 16
Print "a=";a
```

```
Method a, "AnyBaseInput", "FFFFFFFFFFFFFFFF", 16
Print "a=";a
Method a, "AnyBaseInput", "FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", 16
Print "a=";a
Method a, "AnyBaseInput", "16"
Print "a=";a
Mε a, "OutputBase", 2
Print "a=";a
Method a, "intpower", BigInteger("100", 2) as a
Print "a=";a
a=a-BigInteger("10000000000", 2)
Print "a=";a
```

identifier: BINARY.ADD([ΔΥΑΔΙΚΟ.ΠΡΟΣΘΕΣΗ(]

Add two unsigned 32 bit numbers and return modulo 2^32 of the result.

```
HEX BINARY.ADD(0xF0000000,0xF0000001)
                    0xE0000001
```

identifier: BINARY.AND([ΔΥΑΔΙΚΟ.ΚΑΙ(]

```
a=0xFAAA
b=0x5550
Hex  Binary.And(a,b)
```

identifier: BINARY.NEG([ΔΥΑΔΙΚΟ.ΑΝΤΙ(]

an unsigned long can be negate, and Binary.Neg() return the same bits binary long as an unsigned long. So when we add binary.neg(a) to uint(a) we get always 0xFFFFFFFF. To get the signed integer we use sint(), so sint(0xFFFFFFFF)=0xFFFFFFFF& (which is -1).

```
\\ a=uint(0xFFFFABCD&)
b=0xFFFFABCD&
Print a>=0=False
Print b=-21555
Print binary.neg(b)=21554
Print  hex$(uint(b))="FFFFABCD", uint(b)
Print hex$(binary.neg(b)+uint(b))="FFFFFFFF"
Print binary.not(uint(b))=21554
Print hex$(binary.not(uint(b))+uint(b))="FFFFFFFF"
```

identifier: BINARY.NOT([ΔΥΑΔΙΚΟ.ΟΧΙ(]

get an unsigned number from 0x0 to 0xFFFFFFFF

```
Print Hex$(Binary.Not(0xAAAAAAAA))  ' 55555555
```

Look Binary.Neg()

identifier: BINARY.OR([ΔΥΑΔΙΚΟ.Η(]

```
a=0xFAAA
b=0x5550
Hex  Binary.Or(a,b)
```

identifier: BINARY.ROTATE([ΔΥΑΔΙΚΗ.ΠΕΡΙΣΤΡΟΦΗ(]

```
\ -31 to 31
a=0xFAAA00A
Hex  Binary.Rotate(a,1)
Hex  Binary.Rotate(a,2)
Hex  Binary.Rotate(a,-1)
Hex  Binary.Rotate(a,-2)
```

identifier: BINARY.SHIFT([ΔΥΑΔΙΚΟ.ΟΛΙΣΘΗΣΗ(]

```
\ from -31 to 31
a=0xFAAA
Hex  Binary.Shift(a,1)
Hex  Binary.Shift(a,2)
Hex  Binary.Shift(a,-1)
Hex  Binary.Shift(a,-2)
```

identifier: BINARY.XOR([ΔΥΑΔΙΚΟ.ΑΠΟ(]

```
a=0xFAAA
b=0x5550
Hex  Binary.Xor(a,b)
```

identifier: BUFFER([ΔΙΑΡΘΡΩΣΗ(]

```
Buf1=Buffer("file name")
```

Make a new buffer with a copy of the file name we pass.

We can load Png files and we can use Image and Sprite to draw to screen

identifier: CAR([ΠΡΩΤΟ(]

```
a=((1,2), (3,4), (5, 6))
b=Car(a)
Print Car(b)  ' 1
Print Car(Car(a))   ' 1
Print Cdr(Car(Cdr(Cdr(a))))  ' 6
```

identifier: CDATE([ΥΠΜΕΡ(]

```
a_day_as_number=cdate(a_day_as_number, +-years, +-months, +-days)
```

```
A=today
Print date$(cdate(A, 0,4,0))  \\ 4 months later
Print date$(cdate(A, 0,0,100))  \\ 100 days later
Print date$(cdate(A, 0,0,-1))  \\ yesterday
```

identifier: CDR([ΕΠΟΜΕΝΑ(]

```
a=((1,2), (3,4), (5, 6))
b=Car(a)
Print Car(b)  ' 1
Print Car(Car(a))   ' 1
Print Cdr(Car(Cdr(Cdr(a))))  ' 6
```

identifier: CEIL([ΟΡΟΦ(]

round to next integer value

identifier: CHRCODE([ΧΑΡΚΩΔ(]

```
Print Chrcode(Chrcode$(0x0645))
```

return the 16bit from first char in a string

identifier: COLLIDE([ΣΥΓΚΡΟΥΣΗ(]

```
Print Collide(2,100)
```

return true player 2 touch player 1, 100 means 100% the size of player 2 used for collision detection. So we can touch but we need more to raise the collision. Player 1 is in the lowest priority and so never collides with any player. Player 32 is the front most.
We can calculate the collision of a player (including 1) if it touch to a percentage of the size it have to a parallelogram defined by two points (absolute coordinates)

```
Print Collide(2,100,10000,1800,12000,3800)
```

we can use the player as a layer. The second is a way to print on the player as a screen. see LAYER command.

identifier: COLOR([ΧΡΩΜΑ(]

```
PEN COLOR(100,50,255)
```

```
pen color(0,0,0)   ' black
pen color(255,255,255) 'White
```

This function gives a negative number or zero

Pen (and all graphic commands) use negative number for RGB values, positive for basic colors and >=0X80000001 for system colors.

Color Constant like html using #

```
print color(0x020c3d),color("3d0c02"), color(#3d0c02), #3d0c02,color(61, 12, 2)
      -134205     -134205      -134205      -134205      -134205
```

```
print #af, 12  \\ af is a file handler
print #afafaf, 112 \\ is a color number RGB
cls 0X80000001  \\ this is a system color DESKTOP
cls 1   \\ this is QBcolor 1
cls color(61,12,2)  \\ this is Black Bean color look here: http://encycolorpedia.com/3d0c02
cls #3d0c02         \\ this is Black Bean color
cls 0x020c3d        \\ this is Black Bean color (as BGR)
```

if number is 0 to 15 then we have QBcolor and converted to rgb (as negative number)
if number >=0X80000000 then this is a system color and converted to rgb (as negative number)
if number>0 and (number and OxFF000000) =0 then this is an RGB color and we change sign to minus (so we have a negative as rgb)
if number<0 and (number and OxFF000000) =0 then we have a ready Rgb color
if we have 3 numbers Color(rvalue positive, gvalue positive, bvalue positive) we get an Rgb value.
So if we wand #000005 we can use the #mapped or Color(0,0,5).
Ox05 and 5 give the QBasic color 5

identifier: COMPARE([ΣΥΓΚΡΙΝΕ(]

```
a=10
b=11
'' need variable or array item for each place
Dim Base 1, A(2)
A(1)=4, 5
Print Compare(a,b)
Print Compare(a(1),a(2))
```

Work with Strings Variables and Array Items (with binary compare only)

For expressions use spaceship operator

```
Print 10+2<=>10+2, "aaa"<=>"bb"
```

identifier: CONJUGATE([ΣΥΖΥΓΗΣ(]

return the conjugate of a complex number.

identifier: CONS([ΕΝΩΣΗ(]

```
a=(1,2,3,4,5)
b=Cons(a, (6,7,8), a)
Print b
```

identifier: COS([ΣΥΝ(]

Return cosine of an angle in degrees
Return comple if we pass a complex number (cosine of a complex number)
For radians use rCos()

identifier: CTIME([ΥΠΩΡΑ(]

```
a=now+today
b1=CTIME(a,0,100,0)  ' 100 minutes later
b2=CTIME(a,100,0,0)  ' 100 hours later
Print Str$(a,"yyyy-mm-dd hh:nn:ss")
Print Str$(b1,"yyyy-mm-dd hh:nn:ss")
Print Str$(b2,"yyyy-mm-dd hh:nn:ss")
```

identifier: DATE([ΗΜΕΡΑ(]

```
A=Date("2015-10-11")
Print STR$( A ,"LONG DATE")
A=Date("March 7 2009", 1033)
Print Date$(A, 1033, "LONG DATE")
```

identifier: DIMENSION([ΔΙΑΣΤΑΣΗ(]

A function to read number of dimensions and items in each dimension of an array.

```
DIM A(10,20)
Print  Dimension(A(), 0) \\ base of array 1 or 0
Print Dimension(A())
     2
Print Dimension(A(),1), Dimension(A(),2)
     10      20
Print Dimension("A",1), Dimension("A",2)
     10      20
```

Base of Array can't change if we redim (using Dim command).
We can assign another array with any base, and the new one get that base.

```
\\Example
Base 0
Dim A()
For This {
    Base 1
    Dim tmp()
    A()=tmp
}
\\ now A() has base 1, and tmp not exist
```

An array can change dimensions without loosing items (Except we make an array with less items)

```
\\ B act as generator from 1
B=Lambda x=1 -> {=x : x++}
Base 1
Dim A(10)<<B()
Dim A(2,5)
Print A(1,5), A(2,5)
A=(,)   ' pointer to an empty array
' change pointer to array of 5 items
' these arrays are treated as base 0
A=(1,2,3,4,5)
B=(1,)  ' pointer to array with one item (look "," at the end)
B=((1,2),(3,4))
Print Len(A()), Len(A) , Len(B)  ' return number of items
```

A pointer can changed to point another item (array, inventory, stack), but never point to Null

From version 9.4 revision 4 we can use different low bound for each dimension

```
Dim a(3 to 10, -4 to 8)
lbound1=dimension(a(),1, 0)
ubound1=dimension(a(),1, 1)
lbound2=dimension(a(),2, 0)
unound2=dimension(a(),2, 1)
Print "Dimensions:";Dimension(a())
Print "Total items:";Len(a())
Print "Array of ";Dimension(a(), 1);" rows X ";Dimension(a(),2);" columns"
Print "First dimension form ";lbound1;" to "; ubound1
Print "Second dimension from"; lbound2; " to "; unound2
```

identifier: DIV   [ΔΙΑ]

```
\\ integer division
Print 10 div 2
5
```

identifier: DIV#   [ΔΙΑ#]

Euclidean division

```
a=-20
b=6
\\ Euclidean  div as div#, mod as mod#
c=a div# b
d=a mod# b
Print c, d  ' -4   4
Print a=b*c+d
\\ normal
c=a div b
d=a mod b
Print c, d ' -3  -2
Print a=b*c+d
```

identifier: DOC.LEN([ΕΓΓΡΑΦΟΥ.ΜΗΚΟΣ(]

print doc.len(a$) ' is a fast way to read length. No copy needed (as with the Len())
Works for strings as normal strings and for documents (structure with paragraphs)

```
a$={asasafd
     fsdfdsfsfds
     sfsfsf
     }
     document a$
print doc.len(a$)
print len(a$)
```

See command DOCUMENT

```
dim a$(3)
document a$(2)={1stline
2ndline
3dline
}
print doc.len(a$(2))
```

identifier: DOC.PAR([ΕΓΓΡΑΦΟΥ.ΠΑΡ(]

```
Document A$, B$
```

```
A$={aaaaaaa
          bbbbbbbbb
          cccccccccc
          }
B$={
}

Print Doc.Par(A$),Doc.Par(B$)
```

Return Paragraph number

identifier: DOC.UNIQUE.WORDS([ΕΓΓΡΑΦΟΥ.ΜΟΝΑΔΙΚΕΣ.ΛΕΞΕΙΣ(]

```
Document a$="Hello there and hello World!"
N=Doc.Unique.Words(a$)
Words a$
For I=1 To N
     Print Ucase$(Letter$)
Next I
```

Example return a sorted list of all words and a number of occurrence for each
Command Words place words in the stack.

identifier: DOC.WORDS([ΕΓΓΡΑΦΟΥ.ΛΕΞΕΙΣ(]

```
Document a$="Hello there and hello World!"
N=Doc.Words(a$)
Words a$
For I=1 To N
     Print Ucase$(Letter$)
Next I
```

This example return all words, in ascending order.
If we want unique words then we have to use Doc.Unique.Words()

identifier: DRIVE.SERIAL([ΣΕΙΡΙΑΚΟΣ.ΔΙΣΚΟΥ(]

Get an integer as drive serial, maybe a negative number
Use uint() to make it as an unsign integer (with same bits)

```
Print  hex$(hiword(uint(drive.serial("c:"))),2) + "-" + hex$(loword(uint(drive.serial("c:"))),2)
```

```
Hex Uint(Drive.serial("c:"))
```

identifier: EACH([ΚΑΘΕ(]

Iterator for objects (not for group)

```
1)
  N=EACH(A)
2)
  N=EACH(A, 1,3)
3)
  N=EACH(A START TO END)

  \\ FOR 1,2,3
While N {
      Print  N^   \\ return number of N cursor
}
```

For step we can reprogram N inside While loop, using a new Each(N, N^+2, -1)
Each Iterator has own cursor, so we can fold two or more for same object

Objects for A: Array, Inventory, Stack

identifier: EOF([ΤΕΛΟΣ(]

```
While Not Eof(#N) {
    '  do something
}
```

Return true if file cursor pass end of file (E_nd O_f F_ile)

identifier: EVAL([ΕΚΦΡ(]

```
X=10
Print Eval("12*X")
      120
```

Use Valid(Eval(A$)) to see if expression in A$ is valid
or use Try
Try Ok {a=eval(a$) } : if not ok then print "We have a problem with expression"
There are some variations for Eval(object) where object is a group or inventory

identifier: EXIST([ΥΠΑΡΧΕΙ(]

1) Using a string expression as parameter

```
IF EXIST("C:\AUTOEXEC.BAT") THEN PRINT "OK"
```

2) Using an object as first parameter, search an inventory using hash table

```
Inventory alfa=1, 2:=200,"beta":=1000, "100":="ok"
Print alfa
If Exist(alfa, 2) Then Print Eval$(alfa), Eval(alfa)
If Exist(alfa, 1) Then Print Eval$(alfa)
If Exist(alfa, 100) Then Print Eval$(alfa)
If Exist(alfa, "100") Then Print Eval$(alfa)
```

Without search maybe we get error

```
Print alfa$("100") ' ok
Print alfa(500) ' error
```

3)using third parameter to find same keys (for queue type of inventory)

```
a=queue:=1,2:="2A",2:="2B",3,4:="4A",4:="4B",5, 2:="3B"
m=each(a)
\\ $(4) for using proportional printing
Print $(4),"value as string", @(tab(2)),"position - 0 based", @(tab(4)),"key as string in quote"
Print $(0),

While m {
     Print eval$(m), @(tab(2)), eval(m!),@(tab(4)),quote$(eval$(m, m^))
}
\\ new optional parameter for Exist() function
\\ with parameter 0 we get the number of items with same key
Print exist(a,2, 0)=3 ' true
\\ using positive we get the 1st for 1
If exist(a,2,1) Then Print eval$(a)="2A"  ' true
\\ using negative we get last for -1
If exist(a,2,-3) Then Print eval$(a)="2A"  ' true

Print exist(a,4,0)=2
\\ we can iterate same keys using a for loop, and the exist with the third parameter
for i=-exist(a,4,0) to -1
     if exist(a,4, i) then Print eval$(a),  ' we get 4A 4B
next i
Print
```

identifier: EXIST.DIR([ΥΠΑΡΧΕΙ.ΚΑΤΑΛΟΓΟΣ(]

```
Print Exist.Dir("d:")
```

0 if we don't have a CD on d:
-1 if we have.
Look Drive$()

identifier: EXP([ΕΚΘ(]

```
? EXP(LN(3))==3
? EXP((1,0i))=EXP(1)
```

identifier: FILE.STAMP([ΑΡΧΕΙΟΥ.ΣΤΑΜΠΑ(]

```
Open "one.txt" For Wide Output Exclusive as #k
    Print #k, "Hello"
    Print #k, "Hello There"
Close #k
Print Str$(File.Stamp("one.txt"),"hh:nn:ss dd/mm/yyyy") , "utc creation time - by default"
Print Str$(File.Stamp("one.txt" ,1),"hh:nn:ss dd/mm/yyyy") , "utc creation time, 1"
Print  Str$(File.Stamp("one.txt" ,-1),"hh:nn:ss dd/mm/yyyy"), "local creation time, -1"
Print Str$(File.Stamp("one.txt" ,2),"hh:nn:ss dd/mm/yyyy"), "utc write time, 2"
Print  Str$(File.Stamp("one.txt" ,-2),"hh:nn:ss dd/mm/yyyy"), "local write time, -2"
```

identifier: FILELEN([ΑΡΧΕΙΟΥ.ΜΗΚΟΣ(]

```
\\ Example
Document a$={πρώτη σειρά
    δεύτερη
    τρίτη
    τέταρτη
    }
UTF_16LE=0
Save.Doc a$, "checkme.doc", UTF_16LE
If Exist("checkme.doc") Then {
     Print Filelen("checkme.doc")
}
```

identifier: FLOOR([ΔΑΠΕΔ(]

It is the same as Int()
Print Floor(-3.4)=-4, Ceil(-3.4)=-3

See Round(), Bank()

identifier: FORWARD([ΜΠΡΟΣΤΑ(]

```
Document a$={aaaaaaa
bbbbbbbbb
ccccccc}
m=Paragraph(a$, 0)
If Forward(a$,m) then {
     While m {
          Print Paragraph$(a$,(m))
     }
}
```

identifier: FRAC([ΔΕΚ(]

```
Print Frac(1.234)
     .234
```

return fractional part of number
always positive number or zero

identifier: FREQUENCY([ΣΥΧΝΟΤΗΤΑ(]

Return frequency for octave and note number

identifier: FUNCTION([ΣΥΝΑΡΤΗΣΗ(]

```
Print Function("name_of_function",1,4,5)
Print name_of_function(1,4,5)
```

```
Function Alfa (x){
     If x>100 Then {
          =x*500
```

```
    } Else =x
}
a$=&Alfa()
Print Function(a$,200), Function(a$,50)
```

identifier: GROUP([ΟΜΑΔΑ(]

```
Group Alfa {
Private:
    x=0
Public:
    Value {
        =.x
        .x++
    }
}

Print Alfa, Alfa, Alfa
Beta=Alfa
Print Beta , Beta, Beta  ' 3   3   3  is a double
Print Type$(Beta)
Delta=Group(Alfa)
Print Delta, Delta, Delta ' 4, 5, 6
Print Alfa, Alfa, Alfa' 4, 5, 6
```

identifier: GROUP.COUNT([ΟΜΑΔΑ.ΣΥΝΟΛΟ(]

Return number of members in a group
See example

```
Group Alfa {
Private:
    x=100
Public:
    a=10
    b=20
    Dim K(20)=50
    Module beta {
        Print .a, .b, .a*.b, .x, .K()
    }
}
```

```
Alfa.Beta
Print Group.Count(Alfa)  \\ 3 only variables and arrays
Flush ' empty stack
For i=1 to Group.Count(Alfa)
     Print "Weak Reference: "; Member$(alfa, i), Member.type$(alfa, i)
     Push Filter$(Member$(alfa, i),"(")  ' remove ( from array
Next i
Stack  ' look the stack now
Read &K1(), &B1, &A1
Try ok {
   Read &x1  ' it is private so no reference exist!
}
If not ok then Print Error$
Print K1(3), B1, A1
B1+=100
Print Alfa.b

\\ this is a way to get references (not by using weak references), including private variables
Read From Alfa, x2, a2, b2, M()
Print x2, a2, b2, M()
a2+=1000
x2*=500
alfa.beta
```

identifier: HIGHWORD([ΠΑΝΩΜΙΣΟ(]

```
' a is a double
a=0xFFFF0000
Print HighWord(a)  ' 65535
```

LowWord(), HiLowWord(), Hex$()
Hex

identifier: HILOWWORD([ΔΥΟΜΙΣΑ(]

```
a=0xFAAA
b=0x5550
Hex  HiLowWord(a,b)
Print LowWord(HiLowWord(a,b))=b
Print LoWord(HiLowWord(a,b))=b
Print HiWord(HiLowWord(a,b))=a
```

identifier: HSL([ΧΚΦ(]

```
Pen HSL(60,100,50)   ' hue 60 deg, Saturation 100%, Lightness 50%
```

identifier: IF([ΑΝ(]

1) ternary operator

   1.1 Using -1 or True and 0 or false

```
     K=5 : N=3
     Print If(K>N->100, 200)

     m=true
     Function TwoParam (x,y) {
          =x**y
     }
     \\ !(1,2,3) as parameter places 3 parameters in stack for values in function stack
     \\ if() for m=true return array (10,2) so we call TwoParam(!(10, 2)) or TwoParam(10, 2)
     Print TwoParam(!If(m -> (10,2), (30,4)))
```

   1.2 Using 1 and up for execution th Nth expression, we can ommit some expressions (return 0)

```
     \\ only one expression evaluated, all other just skipped
     Print If(Random(1,5) -> 10, 30, 45, 55, 60)
```

2) elvis operator

2.1 for "null" groups (M2000 groups have no null. But in a container we can pass 0 - converted to long- to delete a group)

```
Group M {
     X=100
}
Dim A(10)
\\ we get a copy of M
A(3)=If(A(3)->,M)
M.X++
A(3)=If(A(3)->,M)
Print A(3).X, M.X  \\ 100 101
\\ We can place anything in an array (if array not initialized as Array for Group)
A(3)=0  \\ this place a Long 0 in A(3)
\\ Now because A(3) is 0 we get a copy of M
```

```
A(3)=If(A(3)->,M)
Print A(3).X, M.X  \\ 101 101
M.X+=100
```

2.2 For functions, lambda, groups which return values

```
def f()=false
def g()=100
Print if(f()->,g())  \\ 100
def f()=true
Print if(f()->,g())  \\ -1  true
def g()=false
def k()=5000
def f()=false
Print if(f()->,if(g()->,k()))  \\ 5000
Class Counter {
    x, max
    Value {
        If .x>=.max Then Exit
        .x++
        =True
    }
Class:
    Module Counter (.x, .max) {}
}
F=Counter(10,18)  ' 8
G=Counter(5,6)    ' 1
K=Counter(1,3)  ' 2
counter=1
\\ count 8+1+2= 11 times
\\ 10, 5, 1 are before counting
\\ print from 11, 5 ,1
\\ to 18, 6, 3
While if(F->,if(G->,K)) {
    Print F.x, G.x, K.x, counter
    counter++
}
\\ to make 12, including 10, 5, 1
\\ we cant change F,G,K because are "properties"
\\ we have to clear them (only clear group object, not group members)
\\ In M2000 named groups, all members are exist outside group
Clear F,G,K
\\ so we make one more, for including 10,5,1
```

```
C=Counter(0,1) ' 1
F=Counter(10,18)  ' 8
G=Counter(5,6)    ' 1
K=Counter(1,3)  ' 2
\\ count 1+8+1+2= 12 times
\\ we don't want to print C.x
counter=1
While if(C->,if(F->,if(G->,K))) {
      Print F.x, G.x, K.x, counter
      counter++
}
```

identifier: IMAGE([EIKONA(]

1) Produce a buffer object from an image in a string

```
Img1=Image(a$)
```

2) for PNG in a buffer

```
Img1=Image(a, C1 [][,W1, H1], WAY])
```

C1 color for transparent pixels (so img1 remove transparency)
W1, H1 width & Height im twips
WAY (0 to 15), 4 for flip horizontal

3) Qr Code

```
Img1=Image("http://www.vbforums.com")
// color  0 - 15 ή color(R, G, B) R, G, B: 0 - 255 or #FF0077 (html color)
Img1=Image("http://www.vbforums.com", color(255, 0, 128))
// Error tolerate level
 0 ' The QR Code can tolerate about  7% erroneous codewords
 1 ' The QR Code can tolerate about 15% erroneous codewords
 2 ' The QR Code can tolerate about 25% erroneous codewords
 3 ' The QR Code can tolerate about 30% erroneous codewords
Img1=Image("http://www.vbforums.com", color(255, 0, 128), 3)
Img1=Image("http://www.vbforums.com", , 3)
```

identifier: IMAGE.X([EIKONA.X(]

If A$ has a bmp we can get the width in twips

```
Print Image.X(A$)
```

If A is a buffer loaded with an image file we can get the width in twips (for current output)

```
Print Image.X(A)
```

identifier: IMAGE.X.PIXELS([EIKONA.X.ΣΗΜΕΙΑ(]

If A$ has a bmp then we can get the width in pixels

```
Print Image.X.pixels(A$)
```

If A has a buffer loaded with image file then we can get the width in pixels

```
Print Image.X.pixels(A)
```

identifier: IMAGE.Y([EIKONA.Y(]

If A$ has a bmp we can get the height in twips

```
Print Image.Y(A$)
```

If A is a buffer loaded with an image file we can get the Height in twips (for current output)

```
Print Image.Y(A)
```

identifier: IMAGE.Y.PIXELS([EIKONA.Y.ΣΗΜΕΙΑ(]

If A$ has a bmp then we can get the height in pixels

```
Print Image.Y.pixels(A$)
```

If A has a buffer loaded with image file then we can get the Height in pixels

```
Print Image.Y.pixels(A)
```

identifier: INKEY([ENKOM(]

check keyborad for a delay for a key press between that delay.

```
Print Inkey(1000)


A=1
Thread {
     A++
} as M interval 10
Print Inkey(2000)
Print A
```

identifier: INSTR([ΘΕΣΗ(]

```
Print Instr("12345", "2")
     2
Print Instr("ΑΒΓΔΕΑΒΓΔ","Α",2)
     6
```

\\ Using 8bit/char strings, make them with Str$(string expression), using Locale

```
Locale 1033
Search$ = Str$("aetabetAbet")
Keyword$ = Str$("bet")
Print instr(Search$, Keyword$ as byte)=5
Print instr(Search$, Keyword$, 6 as byte)=9
Print instr(Search$, Keyword$, 1 as byte)=5
\\ now we get UTF-16LE
Search$ = "aetabetAbet"
Keyword$ ="bet"
Print instr(Search$, Keyword$)=5
Print instr(Search$, Keyword$, 6)=9
Print instr(Search$, Keyword$, 1)=5
```

identifier: INT([AK(]

```
>PRINT INT(12.5)
         12
```

Always return lower or equal integer
for 1.9 gives 1
for 1.1 gives 1
for -1.1 gives -2
for -1.9 gives -2

Use Round(-1.9, 0) to get -2

```
' there is a rounding function when we place number to integer variable
a%=-1.9
Print a%
         -2
```

Also works in For Next loop and For {} loop

We can make integers or round values at final stage
We can define decimal point char, using Locale

```
Locale 1032 ' set Greek locale, and now we get coma for decimal point.
Print Format$("{0:0} {0:1} {0:2:-8}", -1.8)
```

identifier: IS   [EINAI]

Operator Is
Variant Is Type for checking types for groups

```
\\ only for objects a and b
```

```
Print a is b
```

For group only:

```
Class Alpha {X=10}
M=Alpha()
Print M is type Alpha
Group Kappa {
          Type: One, Two
          X=10
}
Print Kappa is type Two
```

identifier: JOYPAD([ΛΑΒΗ(]

```
PRINT JOYPAD(0)
```

status of buttons on joystick 0. Each button is a bit.

identifier: JOYPAD.ANALOG.X([ΛΑΒΗ.ΑΝΑΛΟΓΙΚΟ.Χ(]

```
Print Joypad.analog.x(0)
```

Print analog position of joystick.

We have to enable joypad(s) with command Joypad 0, 1 and disable by using Joypad with no parameters

identifier: JOYPAD.ANALOG.Y([ΛΑΒΗ.ΑΝΑΛΟΓΙΚΟ.Υ(]

```
Print Joypad.Analog.Y(0)
```

We have to enable joypad(s) with command Joypad 0, 1 and disable by using Joypad with no parameters

identifier: JOYPAD.DIRECTION([ΛΑΒΗ.ΚΑΤΕΥΘΥΝΣΗ(]

```
Const DirectionNone = 0
Const DirectionLeft = 1
Const DirectionRight = 2
Const DirectionUp = 3
Const DirectionDown = 4
Const DirectionLeftUp = 5
Const DirectionLeftDown = 6
Const DirectionRightUp = 7
Const DirectionRightDown = 8
If Joypad.Direction(0)=DirectionRight Then {
     \\ do something
}
```

We have to enable joypad(s) with command Joypad 0, 1 and disable by using Joypad with no parameters

identifier: KEYPRESS([ΠΑΤΗΜΕΝΟ(]

Function Keypress(number) get the virtual key and perform an GetAsyncKeyState, only when M2000 environment are the foreground window

All keys in keyboard and all keys in mouse can be tested.

```
A=1
Repeat {
     Print A
     A++
} Until Keypress(1)  \\ 1 is left mouse button
```

see https://msdn.microsoft.com/en-us/library/windows/desktop/dd375731(v=vs.85).aspx
for virtual codes

identifier: LEN([ΜΗΚΟΣ(]

Return for:
string, document : length as chars (2 bytes)
Use Len.disp() to give positions for print (maybe some chars are top of others)
Array : length in items (like dimensions =1), or =0 if dimensions=0
Objects: Inventory, Stack, Arrays =number of items

identifier: LEN.DISP([ΜΗΚΟΣ.ΕΜΦ(]

```
\\ Display Length
Print LEN.DISP("ãž")  ' 2
Print Len("ãž")     '4
```

identifier: LN([ΛΦ(]

Natural Logarithm (base e)

identifier: LOCALE([ΤΟΠΙΚΟ(]

1) Get language id from letters

```
Print Locale("ΓΑΒ")
      1032
```

2) Check against an id

```
Print Locale("ΓΑΒ",1032)
      -1
Print Locale("ΓΑΒ",1033)
      0
```

identifier: LOG([ΛΟΓ(]

Logarithm base 10

identifier: LOWWORD([ΚΑΤΩΜΙΣΟ(]

```
A=0xFFEE12C3
Print Hex$(LOWORD(A),2)
Print Hex$(HIWORD(A),2)

HEX LOWORD(A),  HIWORD(A)
```

Also use Lowword() as LoWord()

identifier: MATCH([ΤΑΥΤΙΣΗ(]

```
push 3, "alfa",1,2
```

we can check the types in stack

```
Print match("nn")

Print match("nns")

Print envelope$()    \\ envelope return one of NSAG for each element in stack
NNSN
```

Types:
[N]umber, [G]roup, [E]vent, [B]uffer, [I]nventory, [A]rray, [F]unction, [S]tring, Sta[C]k
F is for Lambda functions (they have object inside)

```
\\Example
Flush \\ flush stack
Dim A(10)
Inventory Alpha=1,2,3
Buffer Clear Beta as Long*10
Group Delta {X=10}
L=Lambda->100
Function Zeta {=100}
Event K {Read X}
A=Stack:=1,2,"οκ",3
Push "οκ", L ,A(), Alpha, Beta,K, Delta, Zeta(), Stack(A)
Print Stack.Size
Stack
Print Envelope$(), Match("CNGEBIAFS"), Match("ΣΑΟΕΔΚΠΛΓ")  \\ also Greek letters can be used
CNGEBIAFS           -1           -1
\\ Envelope$() always return English Letters.
```

identifier: MAX([ΜΕΓΑΛΟ(]

Work only with variables and array items. Always two parameters

```
A=10
B=3
Print MAX(A, B)

Dim A(10)=10
A(3)+=10
Print MAX(A(3),A(5))

c="OK"
d="HELLO"
? MAX(c, d)="OK"
c$="OK"
d$="HELLO"
? MAX(c$, d$)="OK"
```

work with variables and array items but no array with brackets, use Max.data()

Look MIN()

Use Only 2 parameters.
Use Max.Data() for series of expressions, numeric or srtring type.

identifier: MAX.DATA([ΜΕΓΑΛΟ.ΣΕΙΡΑΣ(]

First expression define the type of other expressions (numeric or string)

```
Print MAX.DATA(1, 50*3,2)=150
Print MAX.DATA("a", "c","b")="c"
```

Look MIN.DATA()

identifier: MDB([ΒΑΣΗ(]

```
Print Mdb("name")
```
Return -1 if exist a name.mdb and have the standard password.

identifier: MIN([ΜΙΚΡΟ(]

Use Only 2 parameters.
```
Print Min(a(2), a(1))
Print Min(a, a(1))
Print Min(b,a)
c="OK"
d="HELLO"
? Min(c, d)="OK"
c$="OK"
d$="HELLO"
? Min(c$, d$)="OK"
```

Use Min.Data() for series of expressions

identifier: MIN.DATA([ΜΙΚΡΟ.ΣΕΙΡΑΣ(]

First expression define the type of expressions (numeric or string)

```
Print MIN.DATA(1, 50*3,2)=1
Print MIN.DATA("a", "c","b")="a"
```

Look MAX.DATA()

identifier: MOD   [ΥΠΟΛ]

```
Print 13 mod 3
1
Print 13.5 mod 3
1.5
```

identifier: MOD#   [ΥΠΟΛ#]

Euclidean modulo

```
a=-20
b=6
\\ Euclidean
c=a div# b
d=a mod# b
Print c, d  ' -4   4
Print a=b*c+d
\\ normal
c=a div b
d=a mod b
Print c, d ' -3  -2
Print a=b*c+d
```

identifier: MOD([ΥΠΟΛ(]

Only for BigInteger amd complex numbers
For complex numbers return modulus of complex number

For BigInteger return the modulus from last division (store with result)

```
BigInteger a=211111111111111111111133u
b=a/848484884
Print b, mod(b)
Print b*848484884+mod(b)=a  ' True
```

identifier: MODPOW([ΥΠΟΔΥΝ(]

```
Print modpow(123456789324234324u, 32042304828u, 234234224442u)=130773979602u
```
return (123456789324234324u^32042304828u) MOD 234234224442u

identifier: MODULE([TMHMA(]

```
Print Module(A)
```
return -1 if module A exist

```
Print Module(delta!)
```
return the same value as AddressOf delta

```
module delta {
}
Print addressOf delta = module(delta!)
```

identifier: NOT   [OXI]

```
Print Not True   ' 0  means false
Print Not False ' -1 means true
```

identifier: NOT_2   [ΔEN]

See NOT

```
Print Not True   ' 0  means false
Print Not False ' -1 means true
```

identifier: OPERATORS   [ΤΕΛΕΣΤΕΣ]

For objects only
pointer Is pointer, pointer is nothing
pointer can be any object handled with pointer
group_pointer is group_pointer, group_pointer is group
(group is group not work)

Arithmetic Operators
-3^2 gives 9, 0-3^2 gives -9
+ - = /
^ ** for power
Div

Div# (euklidean)
Mod
Mod# (euklidean)
< > <> <= >= = ==
<=> (spaceship operator)
Or And Xor Not
(,) empty array
(1,) one item array
(1,2) two item array

Operators for arithmetic variables (eg A++ as a statement)
For group properties (not variables) we have to define operators
+=
-=
*=
/=
-! change sign
~ \\ false <-> true
++
--

Operators for Strings
+
< > <> <= >= =
~ (like Like in VB6)
<=> (spaceship operator)

identifier: OR   [H]

Operator Or

Print True Or False

identifier: ORDER([TAΞH(]

Print Order("ALFA1232","ALFA800")

find the order of two strings, by searching names as names and numbers as numbers. This function used internal for Inventories when we choose: sort |ascending|descending| inv_name as numbers.

identifier: PARAGRAPH([ΠΑΡΑΓΡΑΦΟΣ(]

```
Clear
Document a$
For i=1 to 9 {
     a$="aaaaaaaaaaaaaa"+str$(i)+{
     }
}
a$="aaaaaaaaaaaaaa"+str$(i)
For i=10  to 1 step 2 {
     \\ delete
     d$=Paragraph$(a$, i, -1)
}

m=Paragraph(a$, 1-1)   ' 1 but -1 to start
d=Forward(a$, m)
While m {
     Print Paragraph.Index(a$,m), m
     Report Paragraph$(a$, (m))
}
```

identifier: PARAGRAPH.INDEX([ΑΡΙΘΜΟΣ.ΠΑΡΑΓΡΑΦΟΥ(]

=PARAGRAPH.INDEX(document_object, variable)
varIable with id of paragraph, return index number of paragraph (1)
See Paragraph()

```
Document a$ {paragraph1
                    paragraph2
                    }
N=Paragraph(a$, 1)
M=PARAGRAPH.INDEX(a$, N)
Print M=1
```

identifier: PARAM([ΠΑΡΑΜ(]

1) Feed from a string a list of values by place them inline code.
\\ compute parameters

```
a=10
b=40
\\ and fill results in a string using Quote$()
s$=Quote$(a*b, a/b)
\\ funtion Param() with a string merge results inline
Print Param(s$)
Module LookHere (x,y) { Print x, y }
LookHere Param(s$)
LookHere a*b, a/b
```

2) param(object) return an inventory list with properties and methods for an object

```
\\ to get this GUID we need to perform a LIST COM TO A
\\ where A is a module (run this in M2000 command line)
\\ We can't create any object (some need special license)
Declare Worksheet "{00020820-0000-0000-C000-000000000046}"
\\ we make an inventory
\\ we can write: Inventory alfa
\\ but param() with an object return an inventory
\\ keys are all names in capitals, and items are the definitions (a simple form)
\\ for properties and functions
alfa=param(Worksheet)
Report Type$(Worksheet) ' Workbook
IF LEN(ALFA)>1 THEN {
For i=0 to len(alfa)-1
Report 3, alfa$(i!) ' use index, not key
Next i
}
\\ we can write here Declare Worksheet Nothing
\\ But at the end all objects (ActiveX) erased
```

identifier: PHASE([ΦΑΣΗ(]

return the phase of a complex number
(same value from ARG())

identifier: PLAYER([ΠΑΙΚΤΗΣ(]

```
Print Player(12)
```

Return True if Player with priority 12 is visible (but maybe not actual visible if it is out of the

M2000 background form)

Function Collide() check collision on any player (visible or not). So we can filter the ouput using Player().

identifier: POINT([ΣΗΜΕΙΟ(]

```
' number in pixels. A$ must have an image inside
Print Point(a$, 10,10)

oldpoint=Point(a$, 10,10, Color(255, 0, 128))
```

To manipulate images we can use a Hidden Layer, and print image there, so we can use twips and Point to get color, or we can cut any part of image, and place it in another layer or another layer.
Hidden layer is any layer without a SHOW command (also SHOW as variable return -1 if layer is visible)

identifier: POINTER([ΔΕΙΚΤΗΣ(]

Pointers for groups.

```
k=Pointer(object)   \\ same as k->object   ' point to object (named)
k=Pointer((object))  \\ same as k->(object)   ' point to a copy of object (a float object)
k=Pointer(object_array(1,2))   \\ same as k->object_array(1,2)   ' point to object in array (must be group, and is a float group)

\\ float or unnamed groups hold by pointers, or in containers like stack, inventories and arrays.

\\Example
class alfa {
     x=10
}
\\ p is a pointer to group
\\ now point to a unnamed group. The unnamed group lives untii no pointers point to it.
p=pointer(alfa())
\\ this is the same as p->alfa()
Print p=>x

z=alfa()
\\ now point to a named group (we can't use the pointer if z erased, because has a reference
only)
p=pointer(z)
\\ this is the same as p->z
Print p=>x
p=>x+=10
Print z.x=p=>x   ' because z.x is the same as p=>x
p=pointer((z))   ' look the ( ), we get a copy of z as unnamed group.
\\ this is the same as  p->(z)
Print p=>x
p=>x+=10
Print z.x<>p=>x   ' because z.x is not the same as p=>x
list  ' show all variables
```

identifier: POLAR([ΠΟΛΙΚΟΣ(]

create a complex number

```
a=(8, -3i)
? polar(mod(a), arg(a))=a
? polar(abs(a), phase(a))=a
```

identifier: PROPERTY([ΙΔΙΟΤΗΤΑ(]

See Property$()

identifier: RANDOM([ΤΥΧΑΙΟΣ(]

```
Print Random(1,10)  \\ a number >=1 and  <=10
Print Rnd   \\ a number <1 and >=0
X=Random(!12345) \\ set generator to specific point and push old seed in one level stack
```

Example

```
Form 60,40
\\ first we see 6 random numbers from specific seed
x=random(!1234)
Print rnd, rnd, rnd
Print rnd, rnd, rnd  \\ look second line
\\ now we reset the seed
x=random(!1234) \\ push seed  - only one level for pushing seed
Push rnd, rnd, rnd
Drop 3  \\ drop three values from stack
\\ now we break the seed using a new one
x=random(!2341) \\ push seed
```

```
Print rnd, rnd, rnd
x=random(!)  \\ pop old seed
Print rnd, rnd, rnd  \\ here are the same as second line
```

identifier: RATN([ΑΤΟΞ.ΕΦ(]

```
Print rAtn(1)==pi/4
```

Look Atn() for degrees

identifier: RCOS([ΑΣΥΝ(]

return cosine of an angle in radians
see COS()

identifier: READY([ΕΤΟΙΜΟ(]

```
Document Doc$
```

Because an event in a form can be fired twice (if we leave the form and re-activate it again)
Inside an event in a M2000 Form:

```
'  Using Layer there is a callback to Load.Doc to abandon loading
'
If not Ready(Doc$) then exit
Layer FormObject {
     Load.Doc Doc$, Filename$
}

' Either way there is a Busy property in Document object
' and we can read it
If not Ready(Doc$) then exit
Load.Doc Doc$, Filename$
```

identifier: RECORDS([ΕΓΓΡΑΦΕΣ(]

```
Print Records(F)
```

F is file handler used with OPEN command.
We get the number of records in a random type file, or the length in bytes in any other.

identifier: RINSTR([ΘΕΣΗΔΕΞΙΑ(]

```
a$="123456123456123455"
n=Rinstr(a$,"34") ' search from right
m=len(a$)+1
While n>0 {
      Print n  ' chars from left
      Print Mid$(a$, n)
       n=Rinstr(a$,"34", m-n)  ' second parameter as chars form right
}

n=Rinstr(a$,"34")  ' search form right
While n>0 {
      Print n  ' chars from left
      Print Mid$(a$, n)
       n=Rinstr(a$,"34", -n)  'chars from left (using negative number)
}

\\ works for Ansi
Locale 1033
Search$ = Str$("aetabetAbet")
Keyword$ = Str$("bet")
Print rinstr(Search$, Keyword$ as byte)=9
Print rinstr(Search$, Keyword$, 1 as byte)=9
Print rinstr(Search$, Keyword$, 2 as byte)=5

Search$ = "aetabetAbet"
Keyword$ ="bet"
Print rinstr(Search$, Keyword$,1 )=9
Print rinstr(Search$, Keyword$,2 )=5
```

identifier: ROUND([ΣΤΡΟΓΓ(]

1. Print Round(12.11211212, 3)
round to 3rd decimal
2. Print Round(12.121312313123123123)
 print using round to 13 decimal

identifier: RSIN([ΑΗΜ(]

Return sine of an angle in radians
see SIN()

identifier: RTAN([ΑΕΦΑΠ(]

Print rTAN(pi/4)
using radians
See TAN() for degrees input

identifier: SEEK([ΜΕΤΑΘΕΣΗ(]

```
\\ Seek() always return bytes from start of file, and 1 for the first

M=Stack
Flush
Data "allfa", "beta","gama","epsilon", "delta"
open "a.dat" for wide output as #k
While Not Empty {
     Print #k, Letter$;
     Stack M { Data Seek(#k)}  ' Data statement push to end
}
close #k
a=1
b=each(M)
open "a.dat" for wide input as #k
While b {
     Seek #k, a
     Print Input$(#k, (Stackitem(b)-a)/2)   ''use /2 for Wide (UTF16LE)
     a=Stackitem(b)
}
close #k
```

identifier: SGN([ΣΗΜ(]

```
Print Sgn(10),Sgn(0),Sgn(-1)
give  1      0      -1
A=-5
Print Sgn(A)
     -1
```

identifier: SIN([ΗΜ(]

```
Print SIN(45)
```

Return sine of an angle in degree
Return sine of a complex if we pass a complex number
See RSIN() for radians

identifier: SINT([ΑΚΕΡΑΙΟ.ΔΥΑΔΙΚΟ(]

```
a=0xFAAA
Print  Sint(a), Sint(a,2)  ' second as negative
a=0xFFFFFAAA
Print  Sint(a)    ' negative number
Print a   ' as usnigned integer
Hex Uint(Sint(a))

\\ Use Uint() for reverse function
```

identifier: SIZE.X([ΜΕΓΕΘΟΣ.Χ(]

return the size on X axis of the frame which contain a legend

```
Print Size.X(legend$, font_name$, size_pt, angle_rad, extra_space_twips)
```

Example:

```
\\ set italic for font
italic 1
font$="Arial"
\\ set 80 characters by 50 rows
Form 80,50
\\ extra_space 5 pixels (to twips)
extra_space=twipsX*5
aling_value=3
document g$={M2000 Interpreter
Second line
Third line
Last line
}
lines=doc.par(g$)-1
if lines=0 then lines=1
\\ start rotation angle
```

```
number_of_blocks=9
current_angle=pi/3
font_size=17
distance_from_center=300
For i=1 to number_of_blocks {
    aling_value=random(1, 3)
    current_angle+=pi/number_of_blocks*2
    ep=(extra_space, 0)#val(random(0,1))
    ww=Size.X(g$,font$,font_size,0 ,ep)
    hh=Size.Y(g$,font$,font_size, 0,ep)
    w1=Size.X(g$,font$,font_size, current_angle, ep)
    h1=Size.Y(g$,font$,font_size, current_angle, ep)
    move x.twips/2, y.twips/2
    step angle current_angle,ww+distance_from_center
    select case aling_value
    case 2
        step angle current_angle+pi/2, hh/2-hh/lines/2
    case 3
        {
            step angle current_angle+pi/2, hh/2
            step angle current_angle, -ww/2
        }
    else case
        {
            step angle current_angle+pi/2, hh/2-hh/lines
            step angle current_angle,ww/2
        }
    end select
    pen random(11, 15) {
        legend g$,font$,font_size,current_angle, aling_value,0,ep
        move x.twips/2, y.twips/2
        step angle current_angle,ww/2+distance_from_center
        step angle current_angle+pi/2, hh/2
        draw angle current_angle, ww
        draw angle current_angle-pi/2, hh
        draw angle current_angle, -ww
        draw angle current_angle-pi/2, -hh
        move x.twips/2, y.twips/2
        step angle current_angle,ww+distance_from_center
        step -w1/2, -h1/2
        color {polygon 1, w1, 0, 0,h1, -w1, 0, 0,- h1 }
    }
}
```

```
italic 0
kkk$=key$
```

identifier: SIZE.Y([ΜΕΓΕΘΟΣ.Υ(]

return the size on Y axis of the frame which contain a legend

```
Print Size.X(legend$, font_name$, size_pt, angle_rad, extra_space_twips)
```

Example: look size.x()

identifier: SQRT([ΡΙΖΑ(]

return square root of a positive number

identifier: STACK([ΣΩΡΟΣ(]

```
a=Stack:=1,2,"alfa",4,5
b=Stack:=6,7,8,9
b=stack(a,b,a)
Print a
Print b
```

identifier: STACKITEM([ΤΙΜΗΣΩΡΟΥ(]

```
Push 1,2,3
Print stackitem(2), stackitem()
2      3
```

stackitem(1) is the top item in stack - same as stackitem()
if no value exist at index or no index (index greater than size) then an error occur

use it with DROP and STACK.SIZE and STACKITEM$() - for strings- to make frames of parameters on stack.

identifier: TAB([ΣΤΗΛΗ(]

```
Print Tab(3)
Print @(tab(2)),100,@(tab(1)),400
                    400          100
Print tab*3=tab(3)

Print $(,10), tab=10
```

look Tab, $()

identifier: TAN([ΕΦΑΠ(]

```
Print Tan(45)
```

a tan for 45 degree
return a complex number if we pass a complex number

```
Print rTan(pi/4)
```

See ATN()

identifier: TEST([ΔΟΚΙΜΗ(]

We can use Test "Breakpoint Name" somewhere in our code to change caption to form Control. Test() can be used to stop at specific Caption, or at specific condition. We have to feed Control Form field named Print with this function, or by code as the example bellow

```
For i=1 to 100 {
     if i=5 then Test "Start 1", Test("Point 1"), i
     if i=55 then Test "Point 1"
     Print i
}
```

identifier: TIME([ΧΡΟΝΟΣ(]

return time as double from a string representing time

```
Print Time$(Time("10:20"))
```

identifier: UINT([ΔΥΑΔΙΚΟ.ΑΚΕΡΑΙΟ(]

```
' Return a unsigned integer with same bits
a=-345
```

```
Print Hex$(Uint(a))
Hex Uint(a), a   " second print ???- no negative fot Hex Print
```

identifier: USGN([ΔΥΑΔΙΚΟ(]

```
a=8000000000
Print usgn(a)=0xFFFFFFFF   ' cut a to maximum
a=-34324
Print usgn(a)=0  ' cut a to minimum
a=34324
Print usgn(a)=34324
a=8534324
```

identifier: VAL([ΤΙΜΗ(]

1) First Parameter is a String Expression

```
Print Val("123")
Print Val("1 2 3")=123
Print Val("123.233")
Print Val("123,233", ",")
Print Val("123-233", "-")
Print Val("123,233", 0)
Print Val("123,233", 1032)
Print Val("123.233", 1033)
```

2) First Parameter is a Number Expression

```
\\ use it for type conversions
Def TypeOfExpression$(N)=Type$(N)
Print TypeOfExpression$(10)    ' Double
Print TypeOfExpression$(Val(10->0@))    ' Decimal
Print TypeOfExpression$(Val(10->0#))    ' Currency
Print TypeOfExpression$(Val(10->0&))    ' Long
Print TypeOfExpression$(Val(10@))    ' Double
Print TypeOfExpression$(Val(10#))    ' Double
Print TypeOfExpression$(Val(10&))    ' Double
Print TypeOfExpression$(Val(10#->0@))    ' Decimal
Print TypeOfExpression$(Val(10&->0@))    ' Decimal
Print TypeOfExpression$(Val(10@->0#))    ' Currency
Print TypeOfExpression$(Val(10&->0#))    ' Currency
Print TypeOfExpression$(Val(10@->0&))    ' Long
Print TypeOfExpression$(Val(10#->0&))    ' Long
```

We can use names instead zero+symbol
Decimal, Currency, Double, Single, Long, Boolean

```
A=Val(B->Boolean)
```

3) Return in a third parameter the length of valid numeric characters, from beginning of string, or -1 if not number not exist

```
Function IsNumeric(a$) {
    def m
    =val(false->boolean)
    Try {
        if islet then {
            z=val(a$,letter$, m)
        } else.if isnum then {
            z=val(a$,number, m)
        } else z=val(a$,"", m)
        =m>len(a$)
    }
}
Function IsIntegerNumeric(a$) {
    def m
    =val(false->boolean)
    Try {
        z=val(a$,"Int", m)
        =m>len(a$)
    }
}
Print IsIntegerNumeric("1221213123213")=true
Print IsIntegerNumeric("1221213.123213")=false
Print isNumeric("123131232131231231.23123123")=true
Print isNumeric("-123131232131231231.23123123e112")=true
Print isNumeric("-123131232131231231.23123123e112", ",")=false
Print isNumeric("-123131232131231231.23123123e112", 1036)=false
Print isNumeric("-123131232131231231.23123123e112", 1033)=true
Print val("233.44sec", 1033)=233.44
a$="233.44sec"
m=0
Print val(a$, 1033, m)=233.44
if m>0 then Print Mid$(a$, m)="sec"
\\ any string for decimal point with len >1 cut decimals
Print val(a$, "??", m)=233
if m>0 then Print Mid$(a$, m)=".44sec"
```

identifier: VALID([ΕΓΚΥΡΟ(]

1) Valid(numeric expression)

Evaluate the expression and if no error produced return true

```
Print Valid(10/0)
     0
Print Valid(AA)
     0   if AA not exist
     -1 if AA exist
```

We can use it with threads to see if a thread has make something.

```
Clear a, b
print valid("a+b"=12)
          0  '  fault in expression
```

2) Valid(@id)

look if id is a visible constant, variable or an array

```
X=10
A$="Yes"
Dim A(10)
Print Valid(@X)
```

3) Valid(@id1 as id2)

check if id1 is like id2

if id1 is an object check if id2 is same type of id1

But if id1 is a group then check if all members in id2 exist in id1, checking only identifiers and not types.

```
Superclass Alfa {
     x=10
}
Superclass Alfa2 {
     x=10
}
Epsilon=Alfa2
Beta=Alfa
Delta=Beta
Print valid(@beta as Alfa)
Print valid(@beta as Alfa2)
Print valid(@Delta as Alfa)
```

identifier: WRITABLE([ΕΓΓΡΑΨΙΜΟ(]

```
Print writable("c:")
```

```
    -1
```

If we pass a letter for a non exist drive then we get an error
use TRY var { } to catch the error (in Error$)

Group: STRING FUNCTIONS - ΑΛΦΑΡΙΘΜΗΤΙΚΑ

identifier: #EVAL$([#ΕΚΦΡ$(]

See #Eval()

```
class alfa$ {
        value (X) {
                =str$(500*x)
        }
}
z$=alfa$()
print z$(5)
m=(alfa$(), alfa$())
k$=m#val$(0)
print k$(5)
print m#eval$(0, 5)
print "done"
```

identifier: #FOLD$([#ΠΑΚ$(]

```
a=("a","b","c")
fold1=lambda m=1 ->{
    Shift 2
    If m=1 Then
        m=0
        Drop
        Push """"+ucase$(letter$)+""""
    Else
        Push letter$+","+""""+ucase$(letter$)+""""
    End If
}
Print "["+a#fold$(fold1)+"]" ={["A","B","C"]}
```

identifier: #MAX$([#ΜΕΓ$(]

```
a=("a","b","c")
```

```
Print a#max$()="c"

z=-1
print ("a","b","c","a")#max$(z)="c", z=2
```

identifier: #MIN$([#ΜΙΚ$(]

```
a=("a","b","c")
Print a#min$()="a"
z=-1
Print  a#min$(z)="a", z=0 ' position
```

identifier: #STR$([#ΓΡΑΦΗ$(]

```
Print (1,2,3,4)#STR$()
1 2 3 4
Print (1,2,3,4)#STR$(,)
1,2,3,4
Print (1,2,3,4)#STR$("-")
1-2-3-4
Print (1,2,3,4)#STR$("-","0000")
0001-0002-0003-0004
```

identifier: #VAL$([#ΤΙΜΗ$(]

```
a=(1,"A",2,"B")
Print a#val(1)="A"
```

identifier: ADD.LICENSE$([ΒΑΛΕ.ΑΔΕΙΑ$(]

add license for dynamic load of activeX (using Declare)
Same as License.add in VB6
One or Two Strings as parameters: The ProgID and the License Key
Info: https://msdn.microsoft.com/en-us/library/aa277583(v=vs.60).aspx

identifier: ARRAY$([ΠΙΝΑΚΑΣ$(]

```
Dim A$(2,5,2)="ok"
Print ARRAY$("A$",1,4,1)
Print A$(1,4,1)
```

```
\\ array by pointer (or tuple)
A=(1,2,3,"word")
A+=20
Print A
Print Array$(A, 4)
```

identifier: ASK$([ΡΩΤΑ$(]

```
PRINT ASK$("this is a messagebox")
IF ASK$("Info","Title","okTitle", "CancelTitle", Bitmap$) ="okTitle" then print "ok"
```

1. There is ASK() also with same parameters but return number (1 for ok or 2 for cancel)
2. We can use it for input string (maximum 100 chars)

```
IF ASK$("Info","Title","okTitle", "CancelTitle", Bitmap$, "string to input") ="okTitle" then print "ok"
: read A$
```

now A$ hold the response from ASK. If Ask return 2 (cancel) then no string pushing in stack, so we can't read from stack.
3. Bitmap$ is a string that we can load a bitmap, or a filename to a picture (internal use vb6 LoadPicture)

All parameters are optional except the first one.

6 parameters
Text to Ask
Title
Text for Ok, use "*" as first character to make it default control, use only "*" to make it default control with default caption
Text for Cancel use "*" as first character to make it default control, use "*" only as the one before, and "" to not show
String for path or bitmap data
String for make Message box an InputBox.

identifier: BMP$([ΕΙΚ$(]

```
print bmp$("alfa")
```

if exist an alfa.bmp in current folder we get full name (path+name+type)

identifier: CHR$([ΧΑΡ$(]

```
?  chr$(240)
```

if LATIN selected then this print the ascii code of the system locale else print the GREEK one
we can use a second parameter

```
? chr$(240, 1036)   prints the french letter in the 240 position in a French codepage
```

look CHRCODE$() and CHRCODE()

```
locale 1032
o$=Str$("ΓΙΩΡΓΟΣ ΚΑΡΡΑΣ") ' o$ is in ANSI 1032 1 byte char

locale 1033
o1$=chr$(o$)
' o1$ is same bytes but expanded to UTF16LE with no convertion, 2 byte char
Print chr$(o1$, 1032) ' convert ANSI 1033 (in UTF16LE) to ANSI 1032 (in UTF16LE)
\\ we get ΓΙΩΡΓΟΣ ΚΑΡΡΑΣ

\\ we can use Locale to set  locale id fro Ansi conversion
a$=Str$("Hello")
\\ a$ has a string as ansi string
Print len(a$)=2.5
Print "Bytes:"=Len(a$)*2  ' 5 bytes
Print Len(chr$(a$))=5
Print a$
```

```
  \* Using locale ID
FORM 60,32
SET FAST !
showchars  = true
MODULE SHOWME {
     READ  showletters, any_locale
     FOR I = 33 TO 255 {
          if showletters then {
               PRINT CHR$(I,any_locale);
          } else {
               PRINT CHRCODE(CHR$(I,any_locale)),
          }
     }
     PRINT
     PRINT
     REFRESH
}
```

```
\* print the unicode number from ansi using system
Print "SYSTEM"
     SHOWME showchars, 0
\* print the unicode number from ansi using Greek
Print "GREEK"
     SHOWME showchars, 1032
\* print the unicode number from ansi using Hebrew
PRINT "HEBREW"
     SHOWME showchars, 1037
PRINT "FINLAND"
\* print the unicode number from ansi using Finland
     SHOWME showchars, 2077
\* print the unicode number from ansi using Turkish
     PRINT "TURKISH"
     SHOWME showchars, 1055
SET FAST
```

identifier: CHRCODE$([ΧΑΡΚΩΔ$(]

```
Print chrcode$(0x0645)
```

return 16bit UTF8LE chars or surrogates (2*16bit)

identifier: DATE$([ΗΜΕΡΑ$(]

```
Print date$(today)
Print date$(today,1033,"Long Date")
Print date$(today,1033,"MMMMM d yyyy")
Print date$(today) ' use  Greek short day
Print date$(today,)  ' use current locale
Print date$(today,1033) ' short date
```

identifier: DRIVE$([ΟΔΗΓΟΣ$(]

```
Print Drive$("A:\")
```

A string contains the type of drive

identifier: DRW$([ΣΧΔ$(]

return full path with name and type of a wmf file if exist else return empty string

identifier: ENVELOPE$([ΦΑΚΕΛΟΣ$(]

1) print envelope$()
print a string with stack items as N or S depends of type of each.

Letters can be [N]umber, [G]roup, [E]vent, [B]uffer, [I]nventory, [A]rray, [F]unction, [S]tring, Sta[C]k

- a variable reference is a string that hold the variable expanded name

    chek this:

```
a=10 : push &a, &a
read expanded_name$, &b
print expanded_name$, b
```

2) print envelope$(a$)

```
\\  flush the stack to see example
flush
a$=stack$(1,2,5-3,"alfa", str$(123,""),  str$(123), str$(today,"dddd YYYY MMMM")
print envelope$(a$)
\\ now we place 2 numbers from a$ to stack
stack a$, "NN"
print envelope$(a$)
print envelope$()
stack
```

To find types on a stack object we have to make it temporary as the current stack

```
a=stack:=1,2,3,4
Stack a {
     Print Envelope$()
}
```

identifier: EVAL$([ΕΚΦΡ$(]

1) Using eval$() for get value (string) from pointer

```
a$="ok"
```

```
b$=weak$(a$)
b$.="yes"   \\ weak reference point to a$, but at every execution
Print a$
Print eval$(b$.)  \\ "yes"  \\ look dot after $
Print eval$(b$)=b$ \\ no what point to, but pointer only
```

2) Using for Inventory to get key, or get string value

```
Inventory alfa=1,"200",3:=500,4
b=each(alfa)
While b {
      Print eval$(b!), eval$(b)  ' key, element as string
}
```

3)Using for Buffer to get string from memory

```
Structure alfa {
    A as Long   \\ A is an unsinged 32 bit
    B as Integer*40
}
Print Len(alfa)   \\ 4 byte + 2 byte * 40 = 84
Buffer B1 as alfa*20   \\ 84*20=1680 Byres
Print Len(B1)
Return B1, 3!B:=Field$("Hello",40)
\\ Allways use bytes as third parameter
Print Eval$(B1, 3!B, 40*2)
Return B1, 10!B:=Str$("Good")  ' 4 bytes, 1 per char
Print Chr$(Eval$(B1, 3!B, 2*2)) ' read 4 bytes and expand them to 8.
```

identifier: FIELD$([ΠΕΔΙΟ$(]

```
A$=FIELD$("Content text",40)
```

Place the Content text in A$ and fill spaces to make it 40 chars long.

If len is smaller than Content text then a cut perform in the Content text.

identifier: FILE$([ΑΡΧΕΙΟ$(]

```
print file$("text file","TXT")
Load File$()
```

Open file selector for one file (if we change to multiselection from setup, we can select a lot but those selections are dropped, only the one that double click or slide return)
We can put multi type string "BMP|JPG"

identifier: FILE.APP$([ΕΦΑΡΜΟΓΗ.ΑΡΧΕΙΟΥ$(]

```
PRINT FILE.APP$("txt")
```

We get the application that os know to open the txt files.
We can use WIN to send a shell command
We can use OS to send an os commad

identifier: FILE.NAME$([ΟΝΟΜΑ.ΑΡΧΕΙΟΥ$(]

Return name and type of a full path plus name and type (if there are parameters after name then these excluded also)

```
Print File.Name$("c:\name1\name2.exe -help")
name2.exe
```

identifier: FILE.NAME.ONLY$([ΟΝΟΜΑ.ΑΡΧΕΙΟΥ.ΜΟΝΟ$(]

```
Menu \\ clear Menu list
Files + "*"  \\ feed Menu list with filenames
Menu ! \\ open Menu
If Menu>0 Then Print Menu$(Menu), File.Name.Only$(Menu$(Menu))
Print File.Name.Only$("c:\alfa beta\delta.klm")  \\ no check if file exist
\\ automatic remove of any parameters after name
Print File.Name.Only$("c:\alfa beta.bin\delta.klm  -help fileother.txt")  \\ no check if exist file
\\ can work with spaces
Print File.Name.Only$("c:\alfa beta\delta one.klm")  \\ no check if exist file
\\ remove any parameters after name,
Print File.Name.Only$("c:\alfa beta.bin\delta one.klm  -help fileother.txt")  \\ no check if exist file
```

identifier: FILE.PATH$([ΤΟΠΟΣ.ΑΡΧΕΙΟΥ$(]

Return path only from full path plus name plus type extension and maybe some switches after.

identifier: FILE.TITLE$([ΤΙΤΛΟΣ.ΑΡΧΕΙΟΥ$(]

```
PRINT  FILE.TITLE$("TXT")
```

identifier: FILE.TYPE$([ΤΥΠΟΣ.ΑΡΧΕΙΟΥ$(]

return extension letters from a file (remove path and name and maybe some parameters after)

identifier: FILTER$([ΦΙΛΤΡΟ$(]

```
? FILTER$("ABCDEFGH", "BCF")
ADEGH
```

Exclude chars in second string from first string.

look REPLACE$(for replace text no chars only)
The example above as series of relaces$()

```
a$="ABCDEFGH"
a$=REPLACE$("ABCDEFGH", "B")
a$=REPLACE$("ABCDEFGH", "C")
a$=REPLACE$("ABCDEFGH", "F")
```

identifier: FORMAT$([ΜΟΡΦΗ$(]

1) print format$(pattern$, parameter list)
You can use {0} many times and you can use any number. Parameter list can't contain empty parameter position.
But we can omit a parameter in pattern$ like this example:

```
report format$("label1 {1}", 12323, 45)
label1 45
```

We can use {} for multiline strings (automatic remove of space for Indentation based of position of last curly bracket)

```
        a$= format$( {parameters:{0}, "{1}"
                    {2}
                    }, "alfa", 12, "ok")
```

```
Print format$("number {0:2} and {0:4}, 12.3567)
```

This is a round function for numbers only. Work for scientific notiation also.

if we provide a string to a number position then we see the numbers {0:2} and not the string because string always search for {0} for 1st parameter ({1} for second).

```
Print format$("{0:-20}{1:2:-6},{1:2:6}ok", "lead spaces",10.2)
```

2) Process escape codes (using only one parameter as string with escape codes like json strings)

```
Print format$("Paragraph1\r\nParagraph2\r\nParagraph3")
\\ to pass formatted values use two time format$(), one to to process {} and second for escape codes
Print format$(format$("Paragraph1\r\nParagraph2 {0}}\r\nParagraph3", 100))
```

use String$("anything here without escape codes" as json) to produce escape codes

identifier: FUNCTION$([ΣΥΝΑΡΤΗΣΗ$(]

```
Print Function$("name_of_function$",1,4,5)
Print name_of_function$(1,4,5)
Print Function$("{=Mid$({1234567890},number,number)}", 3, 4)
Function Alfa$ (x){
    If x>100 then {
        ="Big"+str$(x)
    } else ="Small"+str$(x)
}
a$=&Alfa$()
Print Function$(a$,200), Function$(a$,50)
```

identifier: GROUP$([ΟΜΑΔΑ$(]

```
Class alfa$ {
Private:
    Name$
Public:
    value {
        =.Name$
    }
    Property Length {Value}=0
Class:
    Module alfa {
        Read .Name$
        .[Length]<=Len(.Name$)
    }
}
```

```
a$=alfa$("Hello")
Print a.length, a$, type$(a$), type$(a)
b$=Group$(a$)
Print b.length, b$, type$(b$), type$(b)
c$=a$
Print type$(c$), c$
```

identifier: HEX$([ΔΕΚΑΕΞ$(]

```
Print hex$(BINARY.NEG(a))
Print hex$(BINARY.OR(a,0xaa0000))
Print hex$(BINARY.AND(a, 0xFFFF))
b=HILOWWORD(0xFFAA, 0x1122)
Print hex$(HIWORD(b)),hex$(LOWORD(b)
Print hex$(BINARY.SHIFT(b,3))
Print hex$(BINARY.ROTATE(b,-6))
Print SINT(0xFFFFFFFF)  ' an usign integer to sign integer
Print hex$(UINT(-1)) ' a sign integer converted to unsigned with same bits
```

unsigned 32 bit (M2000 use real numbers as unsigned integers, so in a variable we can store more bits)

0x1234ABCE   ' unsigned hex numbers

HEX$(unsigned)
HEX$(unsigned, bytes)

BINARY.NEG(unsigned , unsigned)
BINARY.OR(unsigned   ,unsigned)
BINARY.AND(unsigned ,unsigned)
BINARY.XOR(unsigned, unsigned)
USNG(real or integer) convert to unsigned 32bit , so <0 give 0 and >0xffffffff give 0xffffffff
UINT(real as integer or integer) bits are in exact position after conversion. UINT(-1) is 0xffffffff
SINT(unsigned) so SINT(0xffffffff)=-1

HIWORD(unsigned) --> 16 bit
LOWORD(unsigned)
HILOWWORD(unsigned , unsigned)

BINARY.SHIFT(, -31 TO 31)
BINARY.ROTATE(, -31 TO 31)

Using Hex$ to get Binary form

Also One's Complement, Two's Complement

```
read b
function bin$ {
      read a
      b$=hex$(a,4)
      dim a$(16)

a$(0)="0000","0001","0010","0011","0100","0101","0110","0111","1000","1001","1010","1011","11
00","1101","1110","1111"
      document aa$
      for i=1 to len(b$) {
            aa$=a$(eval("0x"+mid$(b$,i,1)))
      }
      = aa$
}

print "Binary form for ";b; " in 32bits"
print  bin$(b)
print  bin$(binary.neg(b)) ,  " One's Complement -";b
print  bin$(binary.neg(b)+1) ,  " Two's Complement -";b
```

identifier: HIDE$([ΚΡΥΦΟ$(]

A cryptography function

```
M$=Hide$("alfabeta", "mycode", 100)
Print M$
Print Show$(M$, "mycode", 100)
```

identifier: IF$([AN$(]

ternary operator
look If()

```
Print If$(true, "true","false")

Print If$(3,"one","two", "three", "four")
```

identifier: INPUT$([ΕΙΣΑΓΩΓΗ$(]

```
n=1 ' for ANSI  ' seek advance *1
n=2' for Wide Open   ' seek advance *2
a$=INPUT$(#filenum, chars*2/n)
a$=INPUT$(#filenum, chars*2/n, 1032)  ' treat as ANSI bytes so chars has to be chars/2 to read ANSI chars
a$=INPUT$(#filenum, chars*2/n, -1) ' treat as UTF8, same as ANSI chars, we get chars/2 from a Wide Open  file.
```

Read number of chars from open file. If we use WIDE in OPEN command then we read unicode (so 2 bytes for each char)

Use SEEK #filenum, position (in bytes) to alter the reading position. Reading position change after a reading, or by moving it.

use records() to find filelength in bytes (so for WIDE type chars we have Records(#filenum)/2 chars).

SEEK always return byte positions, starting from 1. In an empty file, EOF(#filenum) is true because SEEK(#filenum)>RECORDS(#filenum). Each record for an open file of type INPUT, APPEND, OUTPUT has 1 byte length. Do not using in RECORD type file.

With INPUT$() we read line breaks as chars too.

using LINE INPUT we get string as paragraph without the line breaks in the input string.

identifier: JPG$([ΦΩΤΟ$(]

return full path with name and type of a jpg file if existe else return empty string

identifier: LAZY$([OKN$(]

Lazy$() get an expression or a reference to a function, and return a function (as function pattern) with a Module command to change the name space to name of where we create the function, so when we execute this function modules variables and modules and functions are visible.

We can use Eval$("x+y") for expressions where variables x and y exist in current module. We can pass a Lazy$("x+y") as parameter when calling a module and we get lazy evaluation, because evaluation happen in each execution of created function

This is a simple function pattern

```
a$="{=100}"
Print function(a$)
```

```
\\ Example of using Module with lazy evaluation
x=10
y=2
Module Alfa (&f()) {
    Print f()
}
Alfa Lazy$(x**2+y)
\\ In M2000 Modules must keep stack clean, because they get parent stack
\\ Functions use own stack
Module Alfa1 {
    \\ we can check if we have a number
    \\ if its true we make a function from it
    \\ we can match the stack "envelope", or first part of it
    If match("N") then read x : Push lazy$(x)
    Read &f()
    Print f()
}
\\ we can get eager evaluation or lazy evaluation
Alfa1 x**2+y
Alfa1 Lazy$(x**2+y)


\\ example showing passing multi line function for later evaluation
\\ Multiline Function run with same name as this module
\\ so x and y are visible

\\ Modules names are not the part we know before execution
\\ M2000 define a bigger name, using some rules. So this name is a "weak reference" to module

\\ we can see module name (and in a function we have a "module name")
\\ Subroutines are not modules, they run at module space.

Print Module$
x=10
y=2
\\ just a function with module name as this module
a$=Lazy$(x**2+y)
Report a$

Print Function(a$)
a$=Lazy$(number**2+y)
Print Function(a$, 10)
M=12345
```

```
Function MultiLine {
\\ read optional, and assign value to new variable x
    Read ? New x
    Rem : Print zz \\ this is an error, zz not exist
    \\ interpreter can show error position using an auto remark at first line
    Local M=10
    DoSomething(10, 2)
    if x>10 then {
        =x**3+2*y+m
    } else {
        =x**2+y+m
    }
    Sub DoSomething(m1, m2)
        \\ Subs can see anything in this function
        m=m1+m2
    End Sub
}
a$=Lazy$(&Multiline())
\\ Here we see first line as special remark for errors
Report a$
x=50
Print Function(a$, 10), Function(a$)
Print Function(a$, 20), Function(a$)
Print M
\\ Search for subs extend to all code of this module
\\ without check modules/functions borders
\\ if not found in this text, then at a second search interpreter look to parent module, if exist a parent module.
\\ Sub DoSomething() run without any reference from where interpreter found it
\\ so we get only the code, as this code was on last lines of this module
DoSomething(3,4)
Print M

\\ example using function as module
\\ but this function is like a part of module
Clear
Z=100
Module Localmodule {
    Print "This module can called from parent only"
}
Localmodule
Module Global test1 {
    Static M=10
    M++
    Print M
}
\\ this is not a ordinary function
\\ but a part of this module
Function fake (&feedback) {
    Z++
    feedback=Z
    test1
    Localmodule
}

Module Delta {
    Read &func()
    \\ we run a part of parent module
    N=0
    Call func(&N)
    Print N
    Call func(&N)
    Print N
    \ \  call test1 from Delta
    test1
    test1
}
test1
test1
\\ we pass a part of this module
Delta Lazy$(&fake())
test1
```

identifier: LCASE$([ΠΕΖ$(]

return the upper case using the system locale
Lcase$(string expression)   ' return lower case

Lcase$(string_expression, locale_number)   ' local_number apply a filter on string_expression when string has numbers on ANSI range (otherwise we get ?). The encoding is Utf-16Le in Input and Output string.

identifier: LEFT$([ΑΡΙΣ$(]

```
PRINT LEFT$("ABCDEF",2)
AB

Locale 1032
a$=str$("Γιώργος")  ' to ansi using Locale 1032
Print chr$(Left$(a$,3 as byte))="Γιώ"  ' Chr$() convert ansi to  UTF-16LE (using Locale 1032)
```

identifier: LEFTPART$([ΑΡΙΣΤΕΡΟΜΕΡΟΣ$(]

```
Print RightPart$("123456789","5")  ' after 5
Print LeftPart$("123456789","5") ' before 5
```

identifier: LOCALE$([ΤΟΠΙΚΟ$(]

```
Form 60, 30
Double
Print "Example 1.0"
Normal
Scroll Split 2  ' or use Cls, 2  to clear screen too
Locale 1033 ' use 1032 for Greek
For i=0 to 255 {
    If Locale$(i)<>"" then {
        \\ report stop after 3/4 of screen lines printed (lines from low part of split screen)
        Report format$("{0::-3}. Locale$(0x{1})={2}", i, hex$(i,1),Quote$(Locale$(i)))
    }
}
```

identifier: LTRIM$([ΑΠΟΚ.ΑΡ$(]

```
PRINT LTRIM$("  123")="123"
k$=STR$("  123")
PRINT CHR$(LTRIM$(k$ AS BYTE))="123"
PRINT LTRIM$(k$ AS BYTE)=STR$("123")
```

identifier: MAX.DATA$([ΜΕΓΑΛΟ.ΣΕΙΡΑΣ$(]

```
? max.data$("aa","bb")
bb
? max.data$("aa1234","ab","aa","bb")
bb
```

look MAX.DATA() for numbers
se also MIN.DATA$() and MIN.DATA() for numbers

identifier: MEMBER$([ΜΕΛΟΣ$(]

```
? MEMBER$(alfa, 3)
```

print 3d member's name in group alfa

look GROUP

identifier: MEMBER.TYPE$([ΜΕΛΟΥΣ.ΤΥΠΟΣ$(]

```
Print MEMBER.TYPE$(alfa, 2)
```

print data type of second member in group alfa

see GROUP

identifier: MENU$([ΕΠΙΛΟΓΗ$(]

We can put strings in menu and then we can read that as an array.
This is not an array but an internal function.

```
stack new {
    menu   \\ clear internal store, we need that because we use menu + to add
    data "alpha","beta","gama","delta"
    for i =1 to 4 { read a$ : menu + a$ }
    menu !
    if menu>0 then print menu$(menu)
}
```

identifier: MID$([ΜΕΣ$(]

like in BASIC

```
Print Mid$("12345",2)   \ 2345
Print Mid$("12345",2,2)  \ 23
```

Locale 1032

```
a$=str$("Γιώργος")  ' to ansi using Locale 1032
Print chr$(Mid$(a$,2,2 as byte))="ιώ"  ' Chr$() convert ansi to  UTF-16LE (using Locale 1032)
```

identifier: MIN.DATA$([ΜΙΚΡΟ.ΣΕΙΡΑΣ$(]

```
? min.data$("aa","bb")
aa
? min.data$("aa1234","ab","aa","bb")
aa
```

look MIN.DATA() for numbers

se also MAX.DATA$() and MAX.DATA() for numbers

identifier: PARAGRAPH$([ΠΑΡΑΓΡΑΦΟΣ$(]

Paragraph$() copy paragraph from document, or copy paragraph and remove it from document

```
Document Alfa$={First Paragraph
                          Second Paragraph
                          Third Paragraph
                          }
For i=3 to 1 {
     Report Paragraph$(Alfa$, i)
}
oldparagraph$=Paragraph$(Alfa$, 2, -1)
Insert to 2 Alfa$=oldparagraph$+{
     some other paragraph
     and another one
     }
Center=2
Report Center, Alfa$
```

identifier: PARAM$([ΠΑΡΑΜ$(]

Param$() get a string of literals (numbers and stings) and place data inline. First item must be a String

Use Param() if first item is number

```
a$={10,30,"alfa",40}
a=(param(a$))
```

```
Print a
' first is a string
a$={"hello", 10,30,"alfa",40}
a=(param$(a$))
Print a
```

identifier: PATH$([ΤΟΠΟΣ$(]

```
\\ Return path from special folders in Windows
Global Const CSIDL_ADMINTOOLS  = 0x30
Global Const  CSIDL_ALTSTARTUP  = 0x1D
Global Const  CSIDL_APPDATA  = 0x1A
Global Const  CSIDL_BITBUCKET  = 0xA
Global Const  CSIDL_COMMON_ADMINTOOLS  = 0x2F
Global Const  CSIDL_COMMON_ALTSTARTUP  = 0x1E
Global Const  CSIDL_COMMON_APPDATA  = 0x23
Global Const  CSIDL_COMMON_DESKTOPDIRECTORY  = 0x19
Global Const  CSIDL_COMMON_DOCUMENTS  = 0x2E
Global Const  CSIDL_COMMON_FAVORITES  = 0x1F
Global Const  CSIDL_COMMON_PROGRAMS  = 0x17
Global Const  CSIDL_COMMON_STARTMENU  = 0x16
Global Const  CSIDL_COMMON_STARTUP  = 0x18
Global Const  CSIDL_COMMON_TEMPLATES  = 0x2D
Global Const  CSIDL_CONNECTIONS  = 0x31
Global Const  CSIDL_CONTROLS  = 0x3
Global Const  CSIDL_COOKIES  = 0x21
Global Const  CSIDL_DESKTOP  = 0x0
Global Const  CSIDL_DESKTOPDIRECTORY  = 0x10
Global Const  CSIDL_DRIVES  = 0x11
Global Const  CSIDL_FAVORITES  = 0x6
Global Const  CSIDL_FONTS  = 0x14
Global Const  CSIDL_HISTORY  = 0x22
Global Const  CSIDL_INTERNET  = 0x1
Global Const  CSIDL_INTERNET_CACHE  = 0x20
Global Const  CSIDL_LOCAL_APPDATA  = 0x1C
Global Const  CSIDL_MYPICTURES  = 0x27
Global Const  CSIDL_NETHOOD  = 0x13
Global Const  CSIDL_NETWORK  = 0x12
Global Const  CSIDL_PERSONAL  = 0x5   ' My Documents
Global Const  CSIDL_MY_DOCUMENTS  = 0x5
Global Const  CSIDL_PRINTERS  = 0x4
Global Const  CSIDL_PRINTHOOD  = 0x1B
Global Const  CSIDL_PROFILE  = 0x28
Global Const  CSIDL_PROGRAM_FILES  = 0x26
Global Const  CSIDL_PROGRAM_FILES_COMMON  = 0x2B
Global Const  CSIDL_PROGRAM_FILES_COMMONX86  = 0x2C
Global Const  CSIDL_PROGRAM_FILESX86  = 0x2A
Global Const  CSIDL_PROGRAMS  = 0x2
Global Const  CSIDL_RECENT  = 0x8
Global Const  CSIDL_SENDTO  = 0x9
Global Const  CSIDL_STARTMENU  = 0xB
Global Const  CSIDL_STARTUP  = 0x7
Global Const  CSIDL_SYSTEM  = 0x25
Global Const  CSIDL_SYSTEMX86  = 0x29
Global Const  CSIDL_TEMPLATES  = 0x15
Global Const  CSIDL_WINDOWS  = 0x24
List
Print Path$(CSIDL_PROGRAM_FILES)
```

identifier: PIECE$([ΜΕΡΟΣ$(]

```
Print Piece$("alfa.beta",".", 2)  ' return beta   - numbers from 1
Print Piece$("alfa.beta",".")(1)  ' return beta  -  numbers from 0
' variable a get a new object (mArray) from Piece$() without piece number
' Here we get a pointer to array in right expression
a=Piece$("alfa.beta",".")   ' is the same: Piece$("alfa.beta",".")()
Print Array$(a, 1) 'return beta
Print a
Dim B$()
\\ Here we get a copy of array in right expression
B$()=Piece$("alfa.beta",".")
Print B$()
```

identifier: PIPENAME$([ΑΥΛΟΣ$(]

```
Print pipename$(OwnPipeHandler)
```

Using Pipe$() and Pipe we can make communications between two environments. The idea is that a "server"...open a pipe and wait for requests. Each client can use PIPE to send data. If client wants data to get back can create a second pipe and send the pipename along with data, and then stay to listening for data. For client handle own pipe using a handler. We have to place the path of the pipe so we need pipename$(pipehandler) to take it.

identifier: PROPERTY$([ΙΔΙΟΤΗΤΑ$(]

We can make a link to property (with no parameter) in an array or an inventory.
Property$() used for properties that return or read string
Property() used for properties that return or read anything but string

In the example we make property "title" as a link to m$, or using property$()
If property has a parameter then we use an array name propname$(), so we get property using propname(), without parameter

```
title "",0
declare form1 form
With form1, "title" as title$
dim a$(10)
a$(1)=property$(title$)
link title$ to m$
function form1.about {
     about "help1","bla bla"
     m$="help open"
}
function form1.click {
     a$(1)="new title"
}
method form1, "show",1
declare form1 nothing
title "back again", 1
```

identifier: QUOTE$([ΠΑΡΑΘΕΣΗ$(]

here I use ? for PRINT

```
? QUOTE$("alfa")
print "alfa"
```

in a string we can pass one " as chr$(34).

```
A$={"alfa"}   is ok in a module (not in command line interpreter)
A$={
"}"
} is ok because "}" is a string...
A$={ """} isn't ok because one " is missing
```

one way we have to print the ascii code of "...by using this

```
? asc(quote$(""))
```

so quote$("") isn't empty string but two chars ""

```
print quote$(2*4,3+1,"aaa"+"bbb")
8,4,"aaabbb"
```

we can use

```
flush  ' flush the stack
INLINE "DATA "+quote$(2*4,3+1,"aaa"+"bbb")
read  A, B , C$
```

we can make a string before we found inline, so we can put the result(S) to a string and then we give that string to INLINE

```
INLINE "DATA "+a$
```

identifier: REPLACE$([ΑΛΛΑΓΗ$(]

```
Print Replace$("a","b","AABAAABBBBAAA")
BBBBBBBBBBBBB
```

Look some string functions

FILTER$(QUOTE$(FORMAT$(
STRING$(STR$(MID$(RIGHT$(
TRIM$(MIN.DATA$(MAX.DATA$(

For variables only and array items
Is very fast because internal copied references.

```
SWAP a$, b$
```

Here also the searching done without copied the string

```
? COMPARE(a$,b$)
```

-1 for a$>b$, 0 for A$ equal to b$, and 1 for a$<b$)

Look special object Document (for big text) identifier: RIGHT$([ΔΕΞΙ$(]

```
Print Right$("alfa",2)
fa
```

is the same as BASIC function RIGHT$()

```
Locale 1032
a$=str$("Γιώργος")  ' to ansi using Locale 1032
Print chr$(Right$(a$,3 as byte))="γος"  ' Chr$() convert ansi to  UTF-16LE (using Locale 1032)
```

identifier: RIGHTPART$([ΔΕΞΙΜΕΡΟΣ$(]

```
Print RightPart$("123456789","5")  ' after 5
Print LeftPart$("123456789","5") ' before 5
```

identifier: RTRIM$([ΑΠΟΚ.ΔΕ$(]

```
PRINT RTRIM$("123  ")="123"
k$=STR$("123  ")
PRINT CHR$(RTRIM$(k$ AS BYTE))="123"
PRINT RTRIM$(k$ AS BYTE)=STR$("123")
```

identifier: SHORTDIR$([ΜΙΚΡΟΣ.ΚΑΤΑΛΟΓΟΣ$(]

```
? shortdir$("C:\Documents and Settings\All Users\Documents\")
C:\DOCUME~1\ALLUSE~1\DOCUME~1\
? shortdir$("c:\notexistfolder\")
return nonthing
```

return the DOS name of a folder or and filename.
if file or path not exist then return empty string

identifier: SHOW$([ΦΑΝΕΡΟ$(]

Return a hidden message produced from Hide$()
Use a string, a key as string and a value

Look HIDE$()

identifier: SND$([ΗΧΟ$(]

```
? SND$("BELL")
```

return path, name, file type (wav) if BELL.wav exist in current directory or return nothing.

identifier: SPEECH$([ΛΟΓΟΣ$(]

```
? speech$(1)
```

return voice name 1 from text to speech system.

identifier: STACK$([ΣΩΡΟΣ$(]

```
a$={line 1
      line2
      }
\\ we can't pass objects to stack$(), we can use object type stack for this (like a)
b$=stack$(10, a$, 300)
Print envelope$(b$) ' NSN  (number string number)
a=Stack
Stack a {
      \\ feed stack from string
      Rem : Stack b$  ' for all
      Stack b$,"NS"
}

Report StackItem$(a, 2)
Stack b$
Read K
Print K
```

identifier: STACKITEM$([ΤΙΜΗΣΩΡΟΥ$(]

```
push "alfa", "beta"
print stackitem$(), stackitem$(2)
beta      alfa
```

stackitem$() is same as stackitem$(1) and is the TOP item on stack.
if we give a wrong position (that isn't exist) we get an error

If at postition we want to get a string aren't a string value we get an error.

Using Stackitem$() for stack objects

```
a=stack
Stack a {
     Data "alfa", "beta"
}
Print stackitem$(a), stackitem$(a,2)
\\ old stack deleted and variable a point to new stack
a=stack:="alfa1", "beta1"
Print stackitem$(a), stackitem$(a,2)
Flush  \\ flush current stack
Data "alfa2", "beta2"
a=[]   \\ a get current stack, and current stack now is empty (use [ ] without space between])
Print Empty  \\ -1 means true
Print stackitem$(a), stackitem$(a,2)
Data "one more"  \\ to current stack
Stack a     \\ now a is empty, all data of a pushed  to end of current stack
Print Len(a)  ' 0
Print Stack.size ' 3
Stack
Read a1$, a2$, a3$
Print Empty  \\ -1 means true
a=Stack:=a1$, a2$, a3$
Print len(a)  ' 3
```

identifier: STACKTYPE$([ΣΩΡΟΥΤΥΠΟΣ$(]

```
b=(1,2,3,4)    \\ pointer to  mArray object
Dim b1(10)=1   \\ mArray object
\\ in stack we place pointer to arrays (mArray object)
a=Stack:=1,"hello", b, b1()
Stack a {
     \\  Stack object_type_stack {}
     \\ make object_type_stack as current stack, and old stack as pointer saved
     \\ and place back the old stack at the exit
     Print Envelope$()   ' NSAA letters in English
     Print Match("NSAA")   \\ English Letters
     Print Match("ΑΓΠΠ")  \\ Greek Letters
}
```

```
d=Each(a)  \\ Iterator for a
While d {
     Print StackType$(d)   \\  Number, String, mArray, mArray
}
Print StackType$(a, 2)
Print StackItem(a,1), StackItem(a)
Print StackItem$(a, 2)
Print StackItem(a, 3) ' 1 2 3 4
Print Array(StackItem(a, 3), 3)   ' 4
Print StackItem(a, 4) ' 1 1 1 1 1 1 1 1 1 1
```

identifier: STR$([ΓΡΑΦΗ$(]

```
str$(12)
str$(12,"##.###")
str$("Abcde","<")
```

str$("String to convert", localeID) \\
using 0 for localeID we convert to ANSI
use chr$(converted$, 1049) to make it unicode in specific language.

Here we get ansi code from greek letters
a$=str$("Γιώργος",0)
So now we can convert back to greek letters
print chr$(a$,1032)

Second parameter is the same as the parameter in format$() in vb6.

```
\\ we can use Locale to set  locale id fro Ansi conversion
a$=Str$("Hello")
\\ a$ has a string as ansi string
Print len(a$)=2.5
Print "Bytes:"=Len(a$)*2  ' 5 bytes
Print Len(chr$(a$))=5
Print a$
```

identifier: STRING$([ΕΠΑΝ$(]

1) Repeat a string n times.

```
Print String$("A", 3)
AAA
```

2) Print String$(a$)

translate to escape codes using \ for some specific chars.

3) Print String$(a$ as json)

translate to escape codes using json specific way

```
a$={alfa
      "beta"
      gamma
}
Print string$(a$)
Print string$(a$ as json)
' Using fromat$() to change to normal.
Report format$(string$(a$))
Report format$(string$(a$ as json))
```

4) Print String$(a$ as Encode64)

```
  Print String$(a$ as Encode64, 1)  ' compact
  Print String$(a$ as Encode64, 0, 6)  ' crlf every 60 chars, 6 chars pad spaces at left
```

5) Print String$(a$ as Decode64)

decoding from BASE64

6) Print String$(a$ as UTF8enc)

encoding to UTF-8

7) Print String$(a$ as UTF8dec)

decoding from UTF-8

8) Print String$(a$ as URLdec)

```
   Print String$(a$ as URLdec +)  ' change + with spaces
   Print String$(String$(a$ as URLencHTML5) as URLdec)
```

9) Print String$(a$ as URLenc)

Threre are three types of encoding, simple, HTML5 and RFC3986

```
    Print String$(a$ as URLencHTML5)
    Print String$(a$ as URLencHTML5 +)   ' spaces to + not to %20
    Print String$(a$ as URLencRFC3986)
    Print String$(a$ as URLencRFC3986 +)   ' spaces to + not to %20
```

Example for URL parseSS

```
Stack New {
      Data "foo://example.com:8042/over/there?name=ferret#nose",
"urn:example:animal:ferret:nose"
      Data "jdbc:mysql://test_user:ouupppssss@localhost:3306/sakila?profileSQL=true",
"ftp://ftp.is.co.za/rfc/rfc1808.txt"
    Data "http://www.ietf.org/rfc/rfc2396.txt#header1",
"ldap://[2001:db8::7]/c=GB?objectClass=one&objectClass=two"
    Data "mailto:John.Doe@example.com",   "news:comp.infosystems.www.servers.unix",
"tel:+1-816-555-1212"
    Data "telnet://192.0.2.16:80/",   "urn:oasis:names:specification:docbook:dtd:xml:4.1.2",
"ssh://alice@example.com"
    Data "https://bob:pass@example.com/place",
"http://example.com/?a=1&b=2+2&c=3&c=4&d=%65%6e%63%6F%64%65%64"
    data "ftp://username:password@hostname/"
    a=Array([])
}
function prechar$(a$, b$) {
    if a$<>"" then {=quote$(b$+a$)} else ={""}
}
z=each(a)
document s$="["+{
}
While z {
    a$=array$(z)
    s1$={          "uri": }+quote$(a$)+{,
        "authority": }+ quote$(string$(a$ as URLAuthority))+{,
        "userInfo": }+ quote$(string$(a$ as URLUserInfo))+{,
        "username": }+ quote$(string$(a$ as URLpart 3))+{,
        "password": }+ quote$(string$(a$ as URLpart 4))+{,
        "scheme": }+quote$(string$(a$ as URLScheme))+{,
        "hostname": }+quote$(string$(a$ as UrlHost))+{,
        "Port": }+quote$(string$(a$ as UrlPort))+{,
        "pathname": }+quote$(string$(a$ as UrlPath))+{,
        "search": }+prechar$(string$(a$ as URLpart 6),"?")+{,
        "hash": }+prechar$(string$(a$ as UrlFragment),"#")+{
    }
    s$="    {"+{
    }+s1$+"    }"
    if  z^<len(a)-1 then s$=" ,"   ' append to document
    s$={
    }
}
s$="]"
Report s$
```

identifier: STRREV$([ΑΝΑΠ$(]

```
Print StrRev$("abcd")="dcba"
```

identifier: TIME$([ΧΡΟΝΟΣ$(]

```
A=Now
Print Time$(A), Str$(A, "hh:nn")
Print time$(now,)
Print time$(now,1033)
Print ucase$(time$(now,1033, "hh:mm:ss tt"))
Print ucase$(time$(now,1032, "hh:mm:ss tt"))
Print ucase$(time$(now,1033, "Long Time"))
Print ucase$(time$(now,1033, "Short Time"))
Print Time$()   ' return the time zone string used by Windows
Print Time$(Time("UTC"))  ' return UTC time for now
```

Look Time()

identifier: TITLE$([ΤΙΤΛΟΣ$(]

```
Print Title$("george karras")
George Karras
```

identifier: TRIM$([ΑΠΟΚ$(]

```
PRINT "<"+TRIM$("        AAA         ")+">"
<AAA>
Locale 1032
a$=str$(" Γιώργος ")  ' to ansi using Locale 1032 Greek
Print len(a$)=4.5  ' 4.5*2=9 bytes
b$=trim$(a$ as byte)
Print chr$(b$)="Γιώργος"  ' ansi to  UTF-16LE (using Locale 1032)
```

```
Print ">"+str$(123)
> 123  (insert a space before 1)
Print ">"+str$(123,"")
>123  (without a space before 1)
Print ">"+str$(123,"00000")
>00123  (formatted)
```

identifier: TYPE$([ΤΥΠΟΣ$(]

```
a=10
b=(1,2,3,4)
Print Type$(a), Type$(b)   ' double  mArray
```

identifier: UCASE$([ΚΕΦ$(]

return the upper case using the system locale
Ucase$(string expression)

Ucase$(string_expression, locale_number)   ' local_number apply a filter on string_expression when string has numbers on ANSI range (otherwise we get ?). The encoding is Utf-16Le in Input and Output string.

identifier: UNION.DATA$([ΕΝΩΣΗ.ΣΕΙΡΑΣ$(]

```
Print Union.data$(65, 66,"C", "alfa", 0x2102)="ABCalfaℂ"
```

identifier: WEAK$([ΙΣΧΝΗ$(]

weak$() return a string with a weak reference For functions return a copy of function code and if it is a member of group, at the end of string is the weak reference to group
Is the only way to make a reference to array item

```
Dim A(20)=10
Print A()
A(2)+=10
Print A(2)
\\ expression X-1 evaluated before weak reference produced
X=3
a$=Weak$(A(X-1))
Print Eval(a$)
a$.+=20
```

```
Print Eval(a$)
Print A(2), A(2)=Eval(a$)
Print a$
M=100
a$=Weak$(M)
a$.+=20
Print Eval(a$)
Print a$
```

Another example

```
a$="{=1000*number+number}"
\\ a$ is a weak reference to anonymous function in a string (has a block {} inside)
\\ actually a weak reference to function is a copy of code and sometimes there is
\\ a weak reference to object (if it is group member)
Link weak a$ to aa()
Print aa(10,-100)
m=300
b$=weak$(m)
\\ b$ is a weak reference to m
Link weak b$ to d
Print d
d+=5000
Print m, m=d, Eval(b$)
\\ using a dot in string variabe, interpreter expect a weak reference
b$.+=500
\\ we can use dot in Eval() but is optional
\\ in Eval$() is not optional because without dot return the string not the reference string
Print m, m=d, Eval(b$.)
```

Group: VARS READ ONLY - ΜΕΤΑΒΛΗΤΕΣ ΣΥΣΤΗΜΑΤΟΣ

identifier: ABOUT$ [ΠΕΡΙ$]

```
A$=ABOUT$
```

read the feedback from About command.
In user help the clickable strings are in brackets [ ]

```
about ! "Title", 8000,6000,"that [is] easy"
about call {
     select case about$
     case "is"
     about  "Title: IS", 8000,6000,"that [is] easy [too]"
     else
     about  "Title: too", 8000,6000,"that [is] easy"
     end select
}
\* use ctrl+f1 to open About...
\*You can check help form as you input a string.
input "name:",a$
\* we can use About with no parameters to clear help system
```

identifier: APPDIR$ [ΕΦΑΡΜΟΓΗ.ΚΑΤ$]

PRINT APPDIR$
print application directory if we have a proper installation else we get a null string

identifier: BROWSER$ [ΑΝΑΛΟΓΙΟ$]

PRINT BROWSER$

give the title of the page in browser

identifier: CLIPBOARD$ [ΠΡΟΧΕΙΡΟ$]

Print Clipboard$
Clipboard "This is a text for clipboard"

Look   CLIPBOARD.IMAGE$, CLIPBOARD.IMAGE, CLIPBOARD.DRAWING identifier: CLIPBOARD.DRAWING   [ΠΡΟΧΕΙΡΟ.ΣΧΕΔΙΟ]

Loading Emf (metafiles) from clipboard

```
A=CLIPBOARD.DRAWING
MOVE 1000,1000
IMAGE            A, 8000
```

See also CLIPBOARD.IMAGE, CLIPBOARD.IMAGE$, CLIPBOARD$

identifier: CLIPBOARD.IMAGE   [ΠΡΟΧΕΙΡΟ.ΕΙΚΟΝΑ]

Load a bitmap in a buffer

```
A=CLIPBOARD.IMAGE
IMAGE A, 10000
```

Look CLIPBOARD.IMAGE$, CLIPBOARD.DRAWING, CLIPBOARD$

identifier: CLIPBOARD.IMAGE$ [ΠΡΟΧΕΙΡΟ.ΕΙΚΟΝΑ$]

```
a$=CLIPBOARD.IMAGE$
```

we copy the clipboard image (if any) in a$

Look CLIPBOARD.IMAGE, CLIPBOARD.DRAWING, CLIPBOARD$

identifier: CODEPAGE   [ΚΩΔΙΚΟΣΕΛΙΔΑ]

```
Print CodePage
```

return current codepage

identifier: COLORS   [ΧΡΩΜΑΤΑ]

```
PRINT COLORS
```

prints the number of colors that screen can show

identifier: COMMAND$ [ΕΝΤΟΛΗ$]

PRINT COMMAND$

prints the filename with path of the last loaded GSB (General Script Basic the prototype name of M2000)

when we load or save we didn't need path and type, just name.
load "alfa"
But we can use path and type, so we can use command$ to save
save command$
a message box open to ask if we want to overwrite the file or not.

command save command$ can be activated with ctrl+A in CLI (or prompt > stage)

identifier: COMPUTER$ [ΥΠΟΛΟΓΙΣΤΗΣ$]

Print Computer$

prints Computer Name

identifier: CONTROL$ [ΦΟΡΜΑ$]

PRINT CONTROL$
Return the active form name if is one of the AVI, MAIN, HELP, or a User Form. If a user form is part of an array of forms then we get name with parenthesis and index number inside.
We can use this feedback if we need to know what part of environment is in focus. If environment is minimized or has lost focus then this is an empty string

identifier: DIR$ [KAT$]

Print Dir$

Prints current directory

We can change with DIR statement

In a user directory we get a fake root as "."
and that dot is usable in that directory.
we ca return to user folder using dir command
dir user

identifier: DOS_var   [ΚΟΝΣΟΛΑ_μετ]

a=DOS
return value from last call to DOS (shell commend)

identifier: DURATION   [ΔΙΑΡΚΕΙΑ]

Print DURATION

print duration in seconds (as a float number) on a loaded media.

look MUSIC, MOVIE, MEDIA

identifier: EMPTY   [ΚΕΝΟ]

Print EMPTY
Give true if stack is empty

STACK.SIZE=0 is equal to EMPTY

identifier: ERROR$ [ΛΑΘΟΣ$]

Print Error$
print description of an error produced from a try structure

identifier: FIELD_as variable   [ΠΕΔΙΟ_μεταβλητή]

See FIELD
Read only variable FIELD return the status of last FIELD input. So we know if user press ESC or

Arrows to leave the input.
-1 previus
1 next (EXIT DOWN OR ENTER OR TAB)
99 Esc
-20 Home
20 End
-10 pageup
10 pagedown
1 Tab
-1 Shift Tab

identifier: FONTNAME$ [ΓΡΑΜΜΑΤΟΣΕΙΡΑ$]

? FONTNAME$
print the current fontname

we can change font with font command
font "Times"
It is better after change font to perform a MODE or a FORM command.

There are fonts that are not the same size even we set same size as number to specific property size.
Every layer and the background cant have any font we choose.

identifier: GRABFRAME$ [ΠΑΡΕΚΑΡΕ$]

a$=GRABFRAME$
This variable is internal a function that grab a frame in an open video in m2000. We have to put in a string.

identifier: GREEK_variable   [ΕΛΛΗΝΙΚΑ_μεταβλητή]

Print Greek
return True if current dialog's messages set to Greek language
Set Switches "+GREEK" set GREEK to true
Set Switches "-GREEK" set GREEK to false
also GREEK statement set GREEK to true but also change Locale to 1032
LATIN is statement opposite to GREEK, set variable GREEK to false and locale to 1033

identifier: HEIGHT   [ΥΨΟΣ]

Return number of rows in a layer, or in console's form, or in background form

identifier: HWND   [ΠΑΡΑΘΥΡΟ HWND]

```
Print hWND
```

return window handler (used for OS specific external functions)

```
Function BrowseForFile$ {
   Declare shell "Shell.Application"
   try {
   Method shell "BrowseForFolder", hwnd, "Choose a file:", 0x1000 as file
   Declare file.self use file, "self"
   try {
        with file.self, "Path" as file.self.path$
        if right$(" "+file.self.path$,1)<>"\" then {
             = file.self.path$+"\"
        } else {
             = file.self.path$
        }
   }
   declare file.self nothing
   declare file nothing
   }
   declare shell Nothing
}
Print BrowseForFile$()
```

identifier: INKEY$ [ENKOM$]

```
a$=inkey$
```

look if a key is recorded and remove it passing it as a char in a$
if no key is recorded then "" is returned.

identifier: INTERNET   [ΔΙΑΔΙΚΤΥΟ]

```
Print Internet
```

print True if pc has connection with Internet

identifier: INTERNET$ [ΔΙΑΔΙΚΤΥΟ$]

Print Internet$
127.0.0.1
or the public ip

identifier: ISLET   [ΕΙΝΓΡ]

If stack top is string (letters) then
Print IsLet
      -1

identifier: ISNUM   [ΕΙΝΑΡ]

If stack top is number then
Print IsNum
      -1

identifier: KEY$ [ΚΟΜ$]

a$=key$
wait for a key to pressed.
if environment is hide then it showing again.

identifier: LAN$ [ΔΙΚΤΥΟ$]

? LAN$
return IP from our LAN.

we can read computer name too from COMPUTER$

identifier: LETTER$ [ΓΡΑΜΜΑ$]

? letter$
remove a string value from stack and print it. An error produced if no string value is on top of stack

```
a$=letter$  ' we remove the top and place to string, if a string value is in top of stack.
```

We can read if we have any item with EMPTY flag, of with STACK.SIZE
We can find if we have a letter in the top of stack by using ISLET (is letter)
or by using ENVELOPE$(). if left$(Envelope$,1)="S" then ? "we have a string value on top of stack"

Letter$ read the string value dropping it from the stack.

```
a$=letter$+letter$  ' add the two top item if they are string or error produce
\* we can push  to the top (last in first out)
push "a","b"
a$=""
try ok {      ' this is an error trap
     a$=letter$+letter$
}
if a$<>"" then {
? "we get two string values from the stack", a$
}
\* so we print "ba"
```

identifier: MEMORY   [MNHMH]

physical memory (no virtual) free for use in MB

identifier: MENU.VISIBLE   [ΕΠΙΛΟΓΕΣ.ΦΑΝΕΡΕΣ]

menu.visible is a read only variable that we use in a thread when a menu is visible...
because when a menu is hide maybe a thread run and without reading this variable maybe reopen the menu by changing items

```
i=0
thread {
     i++
     if menu.visible then print i, menu$(menu)
} as a
menu title "choose"
after 100 {
     thread a interval 100
}  ' so we start thread "a" when the menu is showing
menu "1st line","2nd line"
print menu
```

' So we want to open.file before open menu and we want a title on the menu
' because title reconstruct the menu after the menu drawing one time
' we have to do some tricks with the threads (because a print in a thread refresh the screen and that isn't good for the menu when has a title to display - titles may have more than one lines using word wrapping - and for that running calculation done in the first display of the menu. A refresh by the print command cancel the next refresh)

so the above example if we wish to use title (and need the thread to run) and open a file and for both the thread must run then
we have to hold the thread when we finish with OPEN.FILE and start MENU and AFTER we can RESTART the thread

```
i=0
thread {
    i++
    if menu.visible then {
        print i, menu$(menu)
    } else {
        print i
    }
} as a
menu title "choose"
    thread a interval 100
open.file
thread a hold
after 100 {
    thread a restart
}  ' so we restart thread "a" when the menu is showing
menu "1st line","2nd line"
print menu
```

identifier: MENU_as variable   [ΕΠΙΛΟΓΗ_μεταβλητή]

```
Menu "Alfa","Beta"
? Menu
```

identifier: MENUITEMS   [ΕΠΙΛΟΓΕΣ]

```
Menu "Alfa","Beta"
```

```
Print Menu
```

Maybe Menu is 1 or 2 or 0 if no selection done.

identifier: MODE_variable   [ΤΥΠΟΣ_μεταβλητή]

Return Mode (height of frame of a line of text) in pt (12 pt is 1/72 of logical inch, a logical inch is 1440 twips, so 1440/72 twips are 12pt)
so 12 pt is equal to 1440/72*12=240 twips

identifier: MODULE$ [ΤΜΗΜΑ$]

```
? module$
```

we print the module's name (or function) that run in that moment (perhaps we are in a thread...so we now that because we write this code in a thread...)

identifier: MODULE.NAME$ [ΟΝΟΜΑ.ΤΜΗΜΑΤΟΣ$]

```
module beta {
	Print module$
	\\ processing the module$
	\\ to retrieve the displayed name
	Print module.name$
}
call beta
```

identifier: MONITOR.STACK   [ΕΛΕΓΧΟΣ.ΣΩΡΟΥ]

```
Print MONITOR.STACK
```

return the size of used interpreter return stack
M2000 Variables and Arrays are not saved in stack
Subroutines have private return stack, check Recursion.Limit
Calling Functions and modules increase the used stack size
Expressions with operators also increase the used stack size

identifier: MONITOR.STACK.SIZE   [ΕΛΕΓΧΟΣ.ΜΕΓΕΘΟΣ.ΣΩΡΟΥ]

```
Print MONITOR.STACK.SIZE
```

print the size of process stack for interpreter

Print MONITOR.STACK.SIZE-MONITOR.STACK
print size of free space in stack

identifier: MONITORS   [ΟΘΟΝΕΣ]

PRINT MONITORS
Return the number of monitors.
If we have 2 monitors we can place the console:
Window 12, 0   ' to montior 0
Window 12, 1 ' to monitor 1̈̋
we can use Window as readonly variable to get a number from 0 to monitors-1 for current monitor.
we can use monitor statememt to identify the current monitor (and which monitor is the primary one)

identifier: MOTION.WX   [ΚΙΝΗΣΗ.ΠΧ]

return left position of background form relative to Screen

identifier: MOTION.WY   [ΚΙΝΗΣΗ.ΠΥ]

Motion.wy is the TOP of current form (always form is the backround in M2000)
Motion.yw is the same

identifier: MOTION.X   [ΚΙΝΗΣΗ.Χ]

Return the left position of top left point of console form (MAIN), in relative with Background
Also used in Layers (1 to 32) (in a block like Layer 2 {}) but return relative to main

identifier: MOTION.XW   [ΚΙΝΗΣΗ.ΧΠ]

return left position of background form relative to Screen

identifier: MOTION.Y   [ΚΙΝΗΣΗ.Υ]

form 32,20;
form

```
y=motion.y
form 32,20
form
motion.w ,y
```

identifier: MOTION.YW   [ΚΙΝΗΣΗ.ΥΠ]

Motion.yw is the TOP of current form (always form is the backround in M2000)

identifier: MOUSE   [ΔΕΙΚΤΗΣ]

```
Print mouse, mouse.X, mouse.Y
```

1 or 2 for the two buttons or 3 for both.
Use keypress(1) or Keypress(2) to read mouse buttons (return true if now are pressed)

identifier: MOUSE.KEY   [ΔΕΙΚΤΗΣ.ΚΟΜ]

Mouse.key return the mouse key which pressed when pressed but not the next time if we keep hold the same mouse key pressed

```
every 150 {
     print mouse.key, mouse,  keypress(1)
}
```

Look Mouse, Keypress()

identifier: MOUSE.X   [ΔΕΙΚΤΗΣ.Χ]

```
PRINT MOUSE.X
```

Return the relative position, on X coordinate of mouse pointer, on current layer (background, foreground or one of 32 layers)

identifier: MOUSE.Y   [ΔΕΙΚΤΗΣ.Υ]

```
PRINT MOUSE.Y
```

Return the relative position, on Y coordinate of mouse pointer, on current layer (background, foreground or one of 32 layers)

identifier: MOUSEA.X   [ΔΕΙΚΤΗΣΑ.Χ]

PRINT MOUSE.X
Return the absolute position, on X coordinate of mouse pointer, from background top left corner

identifier: MOUSEA.Y   [ΔΕΙΚΤΗΣΑ.Υ]

PRINT MOUSE.Y
Return the absolute position, on Y coordinate of mouse pointer, from background top left corner

identifier: MOVIE.COUNTER   [ΤΑΙΝΙΑ.ΜΕΤΡΗΤΗΣ]

same as Music.Counter
return -1 if nothing played
return for current tune the time of play in msec from start.
Look Duration (to get duration of tune)

identifier: MOVIE.DEVICE$ [ΣΥΣΚΕΥΗ.ΠΡΟΒΟΛΗΣ$]

? MOVIE.DEVICE$
we get device name

identifier: MOVIE.ERROR$ [ΛΑΘΟΣ.ΤΑΙΝΙΑΣ$]

? MOVIE.ERROR$
we get movie error description

identifier: MOVIE.HEIGHT   [ΥΨΟΣ.ΤΑΙΝΙΑΣ]

MOVIE LOAD "SECOND"
PRINT MOVIE.HEIGHT' IN TWIPS

identifier: MOVIE.STATUS$ [ΚΑΤΑΣΤΑΣΗ.ΤΑΙΝΙΑΣ$]

? MOVIE.STATUS$
we get the movie status

identifier: MOVIE.WIDTH   [ΠΛΑΤΟΣ.ΤΑΙΝΙΑΣ]

```
MOVIE LOAD "SECOND"
PRINT MOVIE.WIDTH' IN TWIPS
```

identifier: MOVIE_as variable   [TAINIA_μεταβλητή]

Return True if something (video or music from internal media player still executed)
SAME AS MEDIA, MUSIC

identifier: MUSIC.COUNTER   [ΜΟΥΣΙΚΗ.ΜΕΤΡΗΤΗΣ]

same as Movie.Counter
return -1 if nothing played
return for current tune the time of play in msec from start.
Look Duration (to get duration of tune)

identifier: NOW   [ΤΩΡΑ]

```
Print Time$(NOW)
Print Str$(Now, "hh:mm AMPM")
a=Now+Today ' (time is the fraction, and date is the integer part)
```

identifier: NUMBER   [TIMH]

Pick a number from stack, from top of stack, if stack has a number at top or we get an error
Using Number we use info for one time only, no need for a variable to store before

```
Def A()=Number**Number
Print A(2,4)
```

Use Array to get array

```
Push (1,2,3,4)
Print Array
```

use Group to get a group

```
Group k {
     Value {
          =100
     }
}
Push group(k)
Dim a(10)
a(1)=Group
Print a(1)  ' 100
```

For Inventories and stacks we have to use variable or we can get using StackItem() and use Drop to drop from top

```
a=Stack:=1,2,3,4
Print StackItem()
Drop
```

We can read top item on stack without droping. But this can read objects too

```
Push (1,2,3,4,5)
Print StackItem()
or
Print StackItem(1)
```

Use Match(), StackType$(), Envelope$() to check for types before reading

identifier: OS$ [ΛΣ$]

```
? OS$
```

print the OS name

identifier: OSBIT   [ΕΛ OSBIT]

```
Print OsBit
```

return 32 or 64

Look IsWine, Os$

identifier: PARAMETERS$ [ΠΑΡΑΜΕΤΡΟΙ$]

Return parameters (in a string) with - or + as first char. These parametets was in command line before we start a name.gsb program in M2000.
We can call another program from a M2000 program, using USE and passing values, or using Win command and passing parameters
We can use first parameters known for "Switches" command

Open a file "Hello.gsb" using Edit "Hello.Gsb" and copy this:

```
MODULE HELLO {
    form 40
    Show
    Print "Example 1, Hello"
    If instr(parameters$, "-OK")>0 Then {
        Print "You call me from Shell",,"    with parameter -OK"  \\ double ,, means next line
    } else {
        L$=Envelope$()
        If LEN(L$)>0 Then {
            Print "You call me with USE and ";len(L$);" parameters"
            stack
        } else {
            Print "You just call me"
        }
    }
    Print "So Hello Again"
    fkey 1, "end"
    fkey 2, "h1"
    fkey 3, "h2"
    Print "Choose a Function Key"
    fkey
}
MODULE H1 {
    WIN Appdir$+"m2000.exe "+quote$(Dir$+"HELLO.GSB -OK")
}
MODULE H2 {
    use hello.gsb 1, "aaaa", 3
    wait 3000
}
Hello
```

identifier: PEN_variable   [ΠΕΝΑ_μεταβλητή]

Print Pen

Print a negative number if is an RGB value (0...-(256*256*256-1)) or one of 15 colors from 1 to 15 from default list of windows colors.

We can change PEN with PEN command

```
Pen 5
? Pen
            5
```

```
Pen color(255,255,255)
? Pen
              15
\* Because color 15 is color(255,255,255)
Pen color(255,250,255)
? Pen
 -16775935
\* this isn't from default list of windows colors
```

identifier: PLATFORM$ [ΠΛΑΤΦΟΡΜΑ$]

```
? PLATFORM$
```

return the platform description (like WINDOWS NT)

identifier: PLAYSCORE   [ΠΑΙΖΕΙΦΩΝΗ]

```
Print PLAYSCORE
```

if any score thread exist then playscore return true

identifier: POINT   [ΣΗΜΕΙΟ]

Return color of point in current graphic cursor

(using Layer {} we can change current layer so we can get point from anywhere - Forms also use Layer, but we can't get point from controls on form, only from form's Layer)

identifier: POS   [ΘΕΣΗ_μεταβλητή]

```
Print "ΑΒΓΔΕ", pos
Print pos
```

identifier: POS.X   [ΘΕΣΗ.Χ]

```
Move 3000,5000
Print pos.x
        3000
```

Print the current X position in graphics coordinates in current layer identifier: POS.Y   [ΘΕΣΗ.Υ]

```
Move 3000,5000
Print pos.y
        5000
```

Print the current Y position in graphics coordinates in current layer

identifier: PRINTERNAME$ [ΕΚΤΥΠΩΤΗΣ$]

```
? PRINTERNAME$
```

printer name and port

identifier: PROPERTIES$ [ΙΔΙΟΤΗΤΕΣ$]

```
b$=PROPERTIES$
```

we can save printer properties in a string.
we can use PROPERTIES command to setup again the saved properties.

this is an example...we save old properties values in b$
then we open the printer properties dialog
we get new setup in A$
we set back the old properties from B$
then we open the printer properties dialog to see only the change.

```
B$=PROPERTIES$
PRINTER !
A$=PROPERTIES$
PROPERTIES B$
PRINTER !
```

identifier: REPORTLINES   [ΓΡΑΜΜΕΣΑΝΑΦΟΡΑΣ]

```
Print reportlines
```

Return lines from last Report. Report command can be used to compute he needed lines before perform any printing.
Here is a small program. Copy in a module and run it

```
\\ this is a test text. Put another please.
```

```
form 60,25
c$={s
          d
          s.
          .
          ..
          . .... ....
          }
for i=1 to 10 {
          cursor 10,10
          print "*1234567890";
          report 2, c$,i,-1000  ' NO PRINT, using here only to get REPORTLINES
          cursor 10,11
          print "*";
          k= REPORTLINES
          print @(pos,row, pos+i,row+k, 5,8);
          report 2, c$,i,k
          cursor 0,20
          print over   $(6), "Results"    \\ 6 for center and proportional printing
          print under    \\ draw a line (underline text)  and then change text line
          print over     \\ clear one line
          print part i , k
          k$=key$
}
```

identifier: RND   [ΤΥΧΑΙΟΣ]

```
Print Rnd  \\ return a number >=0 and <1
```

identifier: ROW   [ΓΡΑΜΜΗ]

```
Cursor 2,2 : print row
       2
Print @(2,5), row
       5
```

pos, row, tab

identifier: SCALE.X   [ΚΛΙΜΑΞ.Χ]

Return Layer/Main/Background width size in twips
same as x.twips

identifier: SCALE.Y [ΚΛΙΜΑΞ.Υ]

Return Layer/Main/Background height size in twips
same as y.twips

identifier: SOUNDREC.LEVEL [ΗΧΟΓΡΑΦΗΣΗΣ.ΕΠΙΠΕΔΟ]

We can monitor sounds without record (just use the SoundRec New)

```
Soundrec New
Profiler
do
        print over $(1), string$("|", soundrec.level)
        Print Part  $(3, width), timecount div 1000
        Refresh 100
        if keypress(32) then exit
        wait 1
always
SoundRec End
```

identifier: SPEECH as variable [ΛΟΓΟΣ ως μεταβλητή]

```
print speech
```

prints number of voices for speech

```
for i=1 to speech {
    print speech$(i)
}
```

print name of all voices

identifier: SPRITE$ [ΔΙΑΦΑΝΕΙΑ$]

```
a$=sprite$
```

we can use this once after we do a SPRITE command. This hold the part of the screen that SPRITE command alter.

In M2000 we have software sprites and hardware (means handled by the system). The software sprites can be unlimited and has opacity control (and pixel with 100 transparency). can be rotated. If we want to movwe them we have to redraw the screen, so we can HOLD any static backround and then we place sprites. We have REFRESH counter set to a big number like 5000. Before drawing we set REFRESH to zero, then we draw all sprites and we do a REFRESH to move the buffer to screen. After a delay we set refresh to 0 and we RELEASE the screen that we hold before, anf then we place sprites in other postitions, and at the end we do a refresh (with no parameters). So we see moving sprites and never see the drawing procedure. So for these situations we don't need to read Sprite$.

Using Players we don't have to redraw anything, this happen by the system. Sprites can be used as brushes too.

identifier: STACK.SIZE [ΜΕΓΕΘΟΣ.ΣΩΡΟΥ]

```
PUSH 100, "123123"
PRINT STACK.SIZE
```

identifier: TAB [ΣΤΗΛΗ]

Return width of column (for print)

```
Print @(tab*3), "ok"
Print @(Tab(3)), "ok"
```

look Tab(), $()

identifier: TEMPNAME$ [ΕΝΑΟΝΟΜΑ$]

```
a$=TEMPNAME$
```

we create a new file name for temporary use with .tmp file type

We have to create the file, but it is sure that this is unique

identifier: TEMPORARY$ [ΠΡΟΣΩΡΙΝΟ$]

```
Print Temporary$
```

return directory for temporary files

See tempname$

identifier: THIS   [ΑΥΤΟ]

1. This is the object Group

```
Group Alfa {
      X=10
      Module Beta {
            Print This.X, .X
      }
}
Alfa.Beta
```

2. This in a For This {}
 For This {} used for temporary definitons.

identifier: THREADS$ [ΝΗΜΑΤΑ$]

```
? THREADS$
```

return a list of threads in current module or function, including all the threads in the lists of parents modules.

identifier: TICK   [ΤΙΚ]

Return tick for threads. If tick=0 then no thread run
tick advanced from M2000 Task Manager

use this for rimer

```
DECLARE timeGetTime LIB "winmm.timeGetTime" {}
for i=1 to 10 {
Print  timeGetTime()
}
```

identifier: TIMECOUNT   [ΦΟΡΤΟΣ]

```
Profiler ' set profiler on
For i=1 to 10000 {x=10}
Print Timecount  ' measure time in msec (as a double)
```

identifier: TODAY   [ΣΗΜΕΡΑ]

return date for today

```
Print Date$(today)
Print Str$(today, "LONG DATE")
```

identifier: TWIPSX   [ΠΛΑΤΟΣ.ΣΗΜΕΙΟΥ]

TWIPSX
twips per pixel for screen only
so we can use multiplies of twipsx for drawing pixels.
we can draw 10 pixel left

```
draw twipsx*10, 0
```

we can display the pixels width of our display form

```
Print scale.x/twipsx
```

identifier: TWIPSY   [ΥΨΟΣ.ΣΗΜΕΙΟΥ]

TWIPSY
twips per pixel for screen only
so we can use multiplies of twipsY for drawing pixels.
we can draw 10 pixel UP

```
draw 0, -twipsY*10
```

we can display the pixels height of our display form

```
Print scale.y/twipsy
```

identifier: USER.NAME$ [ΟΝΟΜΑ.ΧΡΗΣΤΗ$]

```
? USER.NAME$
```

we can have many users in m2000. Each of them have a user folder and can write only there.
Except the supervisor that can write anywhere.
if we change dir then we can return to user folder using this:

```
DIR USER
```

we can make new users or select if exist with

```
USER name
```

Supervisor has a phantom name. We can create a user with supervisor name but this will be another user, not the supervisor. So we can't change, by using USER, to the supervisor (it is another command that you have to read help carefully to found).

in a user mode commands WIN and DOS not run.

identifier: VOLUME_as variable   [ΕΝΤΑΣΗ_μεταβλητή]

Print Volume
print the audio volume

identifier: WIDTH_as variable   [ΠΛΑΤΟΣ]

return width as chars for current layer

identifier: WINDOW_variable   [ΣΥΣΚΕΥΗ]

Print Window
return number of MONITOR (0 for one monitor setup, or a value from 0 to MONITORS-1)

identifier: X.TWIPS   [Χ.ΣΗΜΕΙΑ]

Return Layer/Main/Background width size in twips
same as scale.x

identifier: Y.TWIPS   [Υ.ΣΗΜΕΙΑ]

Return Layer/Main/Background width size in twips
same as Scale.Y

Group: CONSTANTS - ΣΤΑΘΕΡΕΣ

identifier: ASCENDING   [ΑΥΞΟΥΣΑ]

Used in ORDER for Databases

identifier: BINARY_const   [ΔΥΑΔΙΚΟ_σταθ]

BINARY
constant used in tables in databases

identifier: BOOLEAN   [ΛΟΓΙΚΟΣ]

Used in command TABLE

identifier: BYTE   [ΨΗΦΙΟ]

used in command TABLE and in command Structure and in object BUFFER, also used for buffers, in Return command and in Eval()

identifier: CURRENCY   [ΛΟΓΙΣΤΙΚΟ]

used in TABLE command

identifier: DATEFIELD   [ΗΜΕΡΟΜΗΝΙΑ]

DATEFIELD used in TABLE

identifier: DESCENDING   [ΦΘΙΝΟΥΣΑ]

used in ORDER and SORT commands

identifier: DOUBLE_as constant   [ΔΙΠΛΟΣ]

used as constant in TABLE definitions

identifier: FALSE   [ΨΕΥΔΗΣ]

equal to 0

identifier: FALSE _2   [ΨΕΥΔΕΣ]

Equal to 0

identifier: FORMATTING_ANY TYPE   [ΦΟΡΜΑΡΙΣΜΑ_ΓΙΑ ΚΑΘΕ ΤΥΠΟ]

Used in Str$() and in $() for formatting purposes
excluded a, c, d, h, m, n, p , q, s, t ,w, y, / and :

identifier: FORMATTING_DATE AND TIME   [ΦΟΡΜΑΡΙΣΜΑ_ΗΜΕΡΟΜΗΝΙΩΝ ΚΑΙ ΩΡΑΣ]

1):
time separator
2)/
day separator
3)c
full date or full time
4)d
day as number with no lead zero (1 - 31)
5)dd
like d but can use leading zero
6)ddd
day with 3 letters
7)dddd
day by name
8)ddddd
full day
9)w
number for day (1 for Sunday and 7 for Saturday)
10)ww
Print number of week of year (1-54)
11)m
month number without leading zero(1 -12)
12)mm
month number including leading zero if needed (01-12)
13)mmm
month name with 3 letters
14)mmmm
name of month
15)q
number of quarter of year (1-4)
16)y
Number of day of year (1-366)
17)yy
number of year as last two digits
18)yyyy
number of year (100 -9999)
19)h
hour without leading zero (0 - 23)
20)hh
hour including leading zero if needed (00 - 23)
21)n

minutes without leading zero (0 - 59)
22)nn
minutes including leading zero if needed (00 - 59)
23)s
seconds without leading zero (0 - 59)
24)ss
seconds including leading zero (00 - 59)
25)ttttt
time full form
26)AM/PM
time indicator day/night using upper case letters
27)am/pm
time indicator day/night using lower case letters
28)A/P
time indicator with one letter A or P
29)AMPM
Using system letters for time indicator

identifier: FORMATTING_NUMBERS   [ΦΟΡΜΑΡΙΣΜΑ_ΑΡΙΘΜΩΝ]

For Str$() and for $() in Print command
1)0
digit or zero
2)#
digit or nothing
3).
decimal point
4)%
percent (number times 100 plus % as char)
5),
thousands separator
6) E- E+ e- e+
scientific notation for numbers

identifier: FORMATTING_STRINGS   [ΦΟΡΜΑΡΙΣΜΑ_ΑΛΦΑΡΙΘΜΗΤΙΚΩΝ]

For Str$() an d internal function in Print $(
1)@
space or a char from string
2)&
a char from string or nothing
3)<

lcase the following letters
4)>
Ucase the following letters
5)!
change print direction

identifier: INFINITY   [ΑΠΕΙΡΟ]

Print INFINITY, -INFINITY
          1.#INF   -1.#INF

identifier: INTEGER   [ΑΚΕΡΑΙΟΣ]

Used in TABLE for Databases

identifier: ISWINE   [ΕΛ ISWINE]

Print IsWinw

-1 if interpreter run on a Linux os using Wine

identifier: LONG_TYPE   [ΜΑΚΡΥΣ_ΤΥΠΟΣ]

LONG
type for TABLE command

also is a type for variables see LONG

identifier: MEMO   [ΥΠΟΜΝΗΜΑ]

used in TABLE command

identifier: PI   [ΠΙ]

```
? pi
          3.14159265358979323846264338 32
          Global pi=3.14159~
```

identifier: SINGLE   [ΑΠΛΟΣ]

Used in TABLE for Databases

identifier: TEXT_as constant   [ΚΕΙΜΕΝΟ_σταθερά]

Text used in TABLE

identifier: TRUE   [ΑΛΗΘΗΣ]

true is -1
false is 0

identifier: TRUE _2   [ΑΛΗΘΕΣ]

True is -1

identifier: VERSION_as constant   [ΕΚΔΟΣΗ_σταθερά]

Print Version
prints language version

Group: PRINTINGS - ΕΚΤΥΠΩΣΕΙΣ

identifier: PAGE   [ΣΕΛΙΔΑ]

PAGE
Sending what we have print until now to the printer and open a new page

```
PAGE 1 ' we send the last and we make a new with portrait orientation
PAGE 0 ' as before but with landscape orientation
PAGE PORTRAIT
PAGE LANDSCAPE
```

identifier: PRINTER   [ΕΚΤΥΠΩΤΗΣ]

1. We can change the ouput to the printer (for chars and graphics)

```
PRINTER {
....
exit printing  \\ use this to exit without sending the last page
}
```

2. Or we can define the size of letters (like modetype)

```
PRINTER  12 {
....
}
```

3. We can change the properties by using a form (as the os give to us).

```
PRINTER !
```

see Properties (a way to extract in a string all properties and then reuse it later)

4. We can choose from a menu list a printer

```
PRINTER ?
```

Use printername$ to read printer name
we get Menu=0 if no item selected (using Esc) or Menu$(Menu) for the proper name for option 5

5.We can give a name and port in a string using that prototype: printername (portname)

```
PRINTER 12, that$
```

We can report the printer in use

6. PRINTER

We can't perform a SCROLL UP or DOWN or CLS
If you wish to change background color you can use GRADIENT for each page.
GRADIENT 1, 1   make blue the printer page.
We can use Page command to send Page to printer inside a Print { } or outside before to make paper orientation landscape or portrait .

```
PAGE 1
Printer {
Form 64,66   \\ using 64 chars (for non analog printing) with 66 lines'
\\ we can use Report to print text with justification.
}
```

Name of current printer is in Printername$

identifier: PRINTER.MARGINS   [ΟΡΙΑ.ΕΚΤΥΠΩΤΗ]

```
Printer.Margins    ' reset margins
Printer.Margins LeftInTwips
Printer.Margins LeftInTwips, TopInTwips
```

Because Lmargin shift page left, the Right margin computed from the original width of the page, so we have to add the left margin. This is the same for bottom margin. These two place a white strip to exclude printing area, they don't resize the page.

```
Printer.Margins LeftInTwips, TopInTwips, RightInTwips
Printer.Margins LeftInTwips, TopInTwips, RightInTwips, BottomInTwips
```

identifier: PRINTING   [ΕΚΤΥΠΩΣΗ]

We can use switch commands (and not block command) by using

```
PRINTING ON
commands here
PRINTING BREAK     \\  IS LIKE OFF BUT WITHOUT SENDING LAST PAGE
PRINTING OFF
```

this is an old command style from earlier versions of M2000 interpreter, but can be used in threads so we can make background printing using thread.plan concurrent

See PAGE and PRINTER

identifier: PROPERTIES   [ΙΔΙΟΤΗΤΕΣ]

Properties for printers have a lot of things to take one by one and store for later use
This is a way to get the printer properties in a string. and then to restore them back.

```
B$=PROPERTIES$
PRINTER !  ' we change the properties here
A$=PROPERTIES$  ' we store the new properties
PROPERTIES B$
PRINTER !  ' now we return to the old set of properties
```