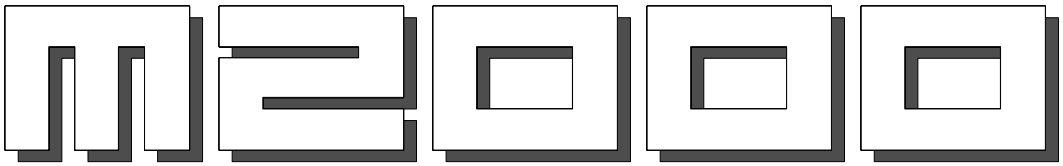

Programming Language

[The introduction paper]

George Philip Karras

June 2020

Preface

This paper describe a subset of M2000 Language, with the core structures and objects. An Interpreter for M2000 exist, as a programming environment with many capabilities like window managing and console managing together. Programming paradigm such as event driven and functional can be apply. Version of interpreter **9.9 Rev. 31**

The idea of M2000 is to be a loved language for pupils to experiment, to make some programs for fun and study. Also there are two vocabularies, one with identifiers in Greek letters, and one with identifiers in Latin letters. We can say: we have Greek and English words, but are identifiers with a meaning for the interpreter according a syntax.

The language first written in 1999 as a simple interpreter like a BASIC clone, but with curly brackets as block of code and a user dictionary like in FORTH of module names and functions, utilizing stacks of values to pass and get data during calls and everywhere in code. The upgrade of language was parallel to the upgrade of interpreter and environment.

M2000 Environment

The M2000 Environment came with M2000 Interpreter. The first idea was for a build of an Environment with a console in a full screen display with text scrolling, using a split screen, with a part of screen which non scrolling at the top, and mixing graphics like the way this did in a BBC model B 8bit computer. Additional was two more capabilities, for databases and for multimedia (images, sounds, movies). A help system driven by a database exist and is internal for all identifiers. Use Help All to see all Identifiers. The Help mdb for 17 revision of 9.9 version change to a dat file (we can use the mdb also see Switches).

Many versions prepared, with compatibility to previous, including layers of display, sprites, threads, tune programming with threads, speaking and finally support for COM objects, events, and private windows manager, windows and controls.

The environment written first in VB5 (Windows 98) and later in VB6 (Windows Xp and finally in Windows 7). First was the ANSI version, and later the Unicode. Also the first version exclude the tab character (converted to spaces), but the later use it combining with spaces. From doubles as the starting only numeric type added through tine many more based on Variant type (hidden to user). Right to left text can be used almost everywhere except some points where can't fit without more programming effort. The M2000 editor can handle RTL text. The RTL in the programming editor (so not for text) are prepared per colored chunk, so the LTR M2000 code stay as LTR.

How to study this paper

For better study use this paper as an entry point, and check the modules offered in the installation in the info.gsb.
If you plan to use M2000 interpreter for Old Style Programming, you have to read:

- The Language until but not included Lambda Object

- Old style programming with M2000 (all)

Or you have to see all the chapters and run each program to M2000 environment.

You find a lot of M2000 programs (300+) in Rossetacode.org

Install Environment

You have to install M2000 environment. Visual Basic 6 run time is part of all Windows Os, so the installation can be done from Xp to Windows 10. You found the setup file from M2000 blog, here:

georgekarras.blogspot.gr

After installation and the opening of M2000 console (running M2000.exe) we can give these statements:

```
Dir AppDir$ : Load Info  <press enter>
```

When you see the introduction message press the F1 to save info to user directory. So after the **End** statement in console which exit the M2000 environment, you can open it next time just using **Load info** and press enter.

The Help System

From revision you don't have to install Access 2007 runtime for help system.

About Author

George Philip Karras born in Athens in 1966. Educated in Technological Educational Institute at the city of Kavala, Greece, where he get a BSc on petroleum engineering in1989. Start the programming journey from 1981, on an 8bit micro, a Texas TI 99/4A from a friend. He loves his first own micro, Acorn Electron, which gave him the opportunity to explore structural programming and assembly language. Later he went to 16bit with Atari STE, and GFA BASIC. Finally he landing to a PC with Pentium 32bit CPU, and the Visual Basic. Then hi found that Visual Basic is a fantastic environment but not for pupils.

This pdf document can be opened in LibreOffice as editable document.

The Language

The language use internal Identifiers and user defined as modules names, and functions. Except identifiers used for flow control all internal identifiers can be replaced by user defined. Names are non case sensitive except for labels. To execute code we can use M2000 console as immediate mode for executing statements except loops, and we can make modules and functions at global level and use them as new identifiers. So there is no RUN statement.

Also we can execute a text file with the gsb as type, to load and execute statements automatic by the M2000 environment.

Literals

A literal is a notation for representing a fixed value in source code. We use literals when we initialize a variable, as arguments when we call named code (a module, a function, a subroutine), and as part of expressions.

Literals used in CSV files, a kind of text files which we can place literals and then we can read from it like we read from console. CSV means comma separated values. In M2000 we can choose two specific properties for CSV the character for decimal point and the character for comma.

For the source code the decimal point always is the dot. The Print statement print numbers using the current decimal point based on Locale number. The Input statement input numbers handling the dot key on keyboard as the current decimal point.

Numeric and Alphanumeric literals

Literals are two kinds: The numeric and the alphanumeric or strings. Numeric by default are double type, so 1 is double type 1 (no need to put a dot zero after one). There are some symbols for interpret a number to another type. So 100& is the 32bit value of 100 (signed integer), 100% is the 16bit value of 100, 100~ is the single floating point value of 100, and there are two more: The 100@ for decimal type and 100# for currency type. For strings we use a pair of " (character code 34), and the curly brackets { }. We can place new line characters - codes 13 and 10 inside curly brackets.

The following example use Print and Report statements. Report used for rendering text. On screen Report stop printing lines at ¾ of scrolling height and wait a spacebar keypress or left mouse button to click, to show the next page.

```
Print  "this is a string", {and this is a string too}

Print 12345678901234567890123456@  ' this is a decimal literal

Report {This is a multiline string
```

```
this is another line
}
```

Hex Literals

A Hexadecimal value need two characters: 0X or 0x, and at the end may use the & or %, depending of type of hexadecimal.

```
Print 0XFFFFFFFF= 4294967295   ' this is unsigned integer 32 bit
Print 0XFFFFFFFF&=-1 ' 32bit signed integer
Print 0XFFFF%=-1 ' 16bit signed integer
```

The unsigned integer is Currency type, but can be stored as 32 bit in a Buffer object. The 4294967295 value above is a double type. The = used for equality. There is == for comparison with rounding to 13 decimal. A comparison return always a boolean type.

Constant Values True and False

A True constant is a double type of -1 and a False constant is a double value of 0. All comparisons return Boolean type. A Boolean True is equal to -1 in any numeric type. A Boolean False is equal to 0 in any numeric type. Logical expressions using AND, OR, XOR, NOT return Boolean and may work with -1 and 0 instead of Boolean True and Boolean False. The non zero value is a True value also.

There are specific functions for Binary operations, like Binary,Not(), Binary.Neg(), Binary.And(), Binary.Or(), Binary.Xor(), Binary.Shift(), Binary.Rotate(), Binary.Add() which use Currency type for 32 bit unsigned numbers.

Html Color Literals

For colors we can use html format so #FF0000 is the Red color, which is the same as Color(255,0,0), which is the same as 12 (from palette 0 to 15, the Windows Default Colors). The Color(255,0,0) and the #FF0000 are negative values. We can use 0X800000XX where XX is from 00 to FF for the Windows System Colors. We read the pen color using the Pen read only variable.

```
Pen #FF0000
Pen Color(255,0,0)
Pen HSL(0,100,50)   ' Hue: 0 deg, Saturation: 100%, Lightness: 50%  (red)
Pen 12    ' red
```

Variables

Variable used for assigning values from literals or from expressions. A variable name must have a letter and any letter or number and the symbol dot and underscore. Some times we use dot before the name. We learn about it later. We use & before the name to indicate the pass by reference.

Variable Type

A variable hold the type of first assignment or declaration of type (there are some exceptions). An expression may change the type if we mix types and the top of all types is the double (not for accuracy but for how big number we can put). There is a read only value the INFINITY which we can get the -INFINITY and +INFINITY as double values. If a variable can't cast a numeric type then we get Overflow error. So the overflow happen to assignment not to the expression.

Numeric variables in M2000 can be defined simple by the first assignment.

```
a=0xFFFFFFFF   ' actual type is Currency

Print a, type$(a)="Currency"

b=0xFFFFFFFF& ' actual type is Long

Print b, type$(b)="Long"

c=0xFFFF% ' actual type is Integer

Print c, type$(c)="Integer"

d=100

Print d, type$(d)="Double"

dd=100~

Print dd, type$(dd)="Single"

a=123456789012345.6789#   ' was a currency type from first line

Print a, type$(a)="Currency"

f=12345678901234567890123456 7@

Print f, type$(f)="Decimal"
```

String Variables (max length more than 1 billion characters) need a $ as the last character:

```
a$="some characters"

$={First line

      Second line

      left margin adjust from closed bracket

      }
```

Boolean Type

There is no specific character to make a value as Boolean. Also the True and False are Double type. But we can define by type name Boolean or using the 1=1 for Boolean True, and 1=0 for Boolean False. All comparisons return boolean type, except the spaceship operator <=>.

```
a=1=1

print type$(a)="Boolean"
```

Statements to define Variables

Here we will see statements for defining variables with or without literal value. When no literal value given, with these statements a zero for numeric and an empty string for string given by default.

A variable get space for value dynamic (not before the source executed). When the code which define the variable exit the execution state then the variable erased both as name and as value.

Variables names have a scope which can be used, to read the value or assign a new value or alter the value using operators like ++ or – and others. M2000 have a shadowing mechanism. A global variable can be in any scope until either the code which define it exit, or a local variable with the same name defined, or a global variable with the same name defined in the chain of calling of named codes (a named code is a module, a function and a subroutine).

Global

The Global statement make new variable always. If x and z exist as global then we get shadowing of old x and z. Look that we assign a value in a global using <=. If we use = then we define a local variable with same name and at this scope we shadowing the global variable.

```
Global x, z=10
```

```
Print x=0, z=10

x<=200

Print x=200, z=10
```

The same statement can define global arrays too:

```
Global A(10)=10

Print A()
```

We will see about arrays later

Local

Statement Local used in Subroutines. Also we can use Local to make local arrays in subroutines.

```
Local x, z=10

Print x=0, z=10

x=200

Print x=200, z=10
```

Def

The DEF statement used for two things. To define functions in one line (always local) and to define once local variables for current scope. If we use Def to define a local variable which exist then we get an error.

```
Deg x, z=10

Print x=0, z=10

x=200

Print x=200, z=10
```

Defining variable using type name

We can define the Boolean type too.

```
Global x as Integer, z as Decimal=10, m as Boolean=True

Print x=0, z=10, m
```

```
x<=200  ' look the <= operator
Print Type$(x)="Integer", Type$(z)="Decimal", Type$(m)="Boolean"
```

For Local statement:

```
Local x as Integer, z as Decimal=10, m as Boolean=True
Print x=0, z=10, m
x=200
Print Type$(x)="Integer", Type$(z)="Decimal", Type$(m)="Boolean"
```

For Def statement multi types:

```
Def x as Integer, z as Decimal=10, m as Boolean=True
Print x=0, z=10, m
x=200
Print Type$(x)="Integer", Type$(z)="Decimal", Type$(m)="Boolean"
```

For Def statement multiple variables with the same type:

```
Def A,B,C   ' double by default
Def Integer A1, B1, C1
List
```

Using List we see the values of variables (the numbers have the type using the literal symbol). Here the last three give 0% (zero for integer 16bit)

Passing values to Variables

A variable can get value from user, using the INPUT statement. This statement also make a new local variable if the name is new.

```
Input "A,B=", A, B
Print A, type$(A)
Print B, type$(B)
```

A and B are Double type

```
Input "A,B=", A%, B%

Print A%, type$(A%)

Print B%, type$(B%)
```

A% and B% are double type but hold no fraction part, only the integer part. We can't pass 1.34 because dot key not used for this kind of variables in Input statement. There is a variation of Input which input values from text files CSV, so we can use "as long" or other types to define the variable type from file, as we see later.

We can use Input with predefined variables at any numeric kind except Boolean

```
Def A as Currency

Input "Money=", A

Print A

Print type$(A)
```

For Boolean use the Menu statement which open a drop down menu and pick True or False:

```
Print "Check:";     ' the semi colon hold cursor so menu open after Check: in
same line

Menu "True","False"

A=Menu=1

Print A

Print type$(A)
```

Stack of Values

Another way to pass a value to a variable by using the stack of values. You have to think two things: in scope there are identifiers like the local variables and the stack. The stack is a dynamic structure which we can put values, so we say that the stack is a stack of values.

We can use Push to push values to top of stack, and Data to place values to bottom of stack. We can use Read to read values to variables, always from top of stack. Check the Data statement (FIFO, first in first out)

```
Flush  ' empty the stack
```

```
Data 100, 200#, 300@  ' to the bottom a Double, a Currency and a Decimal.

Stack ' display the stack

Read A, B, C

List ' display the values of variables
```

Now we use the Push statement (LIFO, Last In first out:

```
Clear ' erase variables

Flush  ' empty the stack

Push 100, 200#, 300@  ' to the top

Stack ' display the stack

Read A, B, C

List ' display the values of variables
```

There are a lot of functions for using the Current stack. Also we can defined named stacks, a kind of object. We will see later about it.

Constants

Some times we want some values to be constants. So we want a name like a variable to get a final value. Statement List show the variables (all non hidden variables, and those that aren't in scope in context). Let say that this code is in a module B (for what is a module we will see in the following page).

```
const X=10
const X1=X*3
list
```

we get B,X=[10], B,X1=[30] where B.X is the absolute or global name for X. The value in square brackets indicate in the **list** report that the value is constant.

Enumerations

Some times we want a list of constants which are in an order, or at least in a set, which we can compare them. Say that the code is in module C (C a name for a module)

```
Enum Pets {Dog=100, Cat}

a=Dog
```

```
Print a=100  ' true

a++

Print a=Cat ' true

b=-a

Print Type$(a)="Pets", Eval$(b)="Cat"   ' true  true

List
```

We get from List: C.PETS*[Pets], C.DOG*[Pets], C.CAT*[Pets], C.A*[Pets]. C.B*[Pets] the *[ ] indicates that the name inside is an Object and the variable is a pointer to that object. M2000 Interpreter handle these objects as constants. The **a** variable can get a number which is part of the numbers in the Pets (100 or 101) or negative numbers -100 and -101. The object B has -101 value but is a Cat (like A which is 101). The type is case sensitive, as we use it in the text. So if we use Enum pEts then all the pointers above give *[pEts] and we have to use "pEts" to check the type of enum. We can use **Enumeration** instead **Enum**.

Expressions

An expression in a program source is a piece of code which return a value. So we can say that a literal is an expression without operator. Operators are the symbols which work on values to produce results, like the plus + operator where 10+10 has the result of 20.

For M2000 there are two types of expressions: The numeric expressions and the string expression. In numeric expressions there are also those known as logical expressions. Logical expressions also have two level, the compare level and the gate operator level. The compare level use operators like =, == (for number rounding comparisons), >=, <=, <>, ~ (a like operator for strings) plus IS for comparing pointer of objects and all return a Boolean type. The gate operator level works on Boolean types (or cast to Boolean for numeric values), use gates as AND, OR, XOR and the NOT. There are a special operator, the spaceship, <=>, which compare two numbers or two strings and return one of three results -1,0,1 to indicate the first a less from second, or equal to second, or more from second. So this is a compare which not return Boolean (a binary value) but a tri-state value.

Numeric expressions use operators like +, -, *, /, ** and ^ for power, and DIV and MOD for integer division (used also for non integer values) and modulo operation, plus DIV# and MOD# for euclidean division and modulo.

This expression -2^2 is equal 4 and 0-2^2 is equal -4. The difference in first case is the minus operator is the unary one, so first expression is (-2)^2 and the second 0-(2^2). So the next expression 0+-2^2 is equal 4 because we have 0+(-2)^2, the minus operator is unary. The next 0+---2^3 is equal to -8 because we have 0+(---2^3) which is 0+(-2)^3 which is -8.

In numeric expressions we can use Parenthesis. All expressions may have or not literals, and may have variables and other identifiers (constants, enumerations, arrays, functions, objects). String expressions use only operator +, so "A"+"B" return "AB". See an example where we have string expression in comparison and comparisons in numeric expressions.

```
A$="122"  : Print ("123">A$)*-3=6+3*(LEN(A$)=4-1) OR FALSE
```

We get True. Comparisons in numeric expressions need parenthesis. The boolean type cast to double, 0 or 1. LEN(A$) return 3, and 3=41 return True, so we get 6+3*-1, or 6-3, or 3. "123">A$ is true (as string comparison not numeric), So we get -1*-3, or 3, so we get 3=3 OR FALSE, or TRUE OR FALSE, so the result is TRUE.

There are many numeric and string built in functions. The name with a $ at the end is a string function.

We can make Groups with programmable operators to perform group expressions and return Groups.

All expressions may return objects. An object return a value of zero plus the object. If we add two objects which there isn't a programmable operator for the first one we get the last object as the return object plus the zero value. If a variable is new or has a numeric value then the object has priority and the variable get the object. If the variable has an object and we place another object which isn't compatible then we get error. There are more rules but not for now.

Some object constructed applying a list of variables without using parenthesis. If those objects are passed to a list of parameters we have to put them in parenthesis (except the last one, optionally).

```
A=((Stack:=1,2,4,8,16), List:="A":=50,"B":=200, "Name":="Goofy")

Print A#Eval(0,2)=4, A#Eval(1,1)=200, A#Eval$(1,2)="Goofy"

Def ByKey(a as list, k$)=a(k$)

Def ByKey$(a as list, k$)=a$(k$)

Def ExportKey$(a as list, where)=Eval$(a, where)

Print ByKey$(A#Val(1),"Name")="Goofy", ByKey(A#Val(1),"A")=50

Print ExportKey$(A#Val(1), 2)="Name"
```

Modules and Blocks

Blocks {}

Blocks of curly brackets { } used for mainly two reasons, plus too more: Firstly, a block has an internal logic to handle GOTO statements, for jumping out of block, and EXIT to exit from block. Secondly there is a set of identifiers for making the block as a loop block, and as alone a block can change to is a loop block using internal in the block statements like LOOP and RESTART.

This language use curly brackets as a block of code and as a string, depends the context. Some language's structures exist with block and without block like the For statement:

```
For I=1 to 100

Next I
```

and the faster, a bit, as for an interpreter which execute in one pass, except for foreword looking for some reasons, and one of those is it to get the block, separate it and execute it.

```
For i=1 to 100 {

}
```

The non block version first find the Next statement (without checking the variable after Next - which is optional) and make a block. So we didn't see a block but the block exist. We will find some other structures with no block inside. Also there some structures which use blocks only and we have to show the block using brackets.

```
Try ok {Print 1/0} : If Not ok Then ? "You get an error:";Error$
```

The ? As statement is the Print statement. A block here has no multi line arrange. We use semicolon to separate with the next statement in the line.

Modules

A **Module** is a named part of a block code, which we can call it. A module construct has a name and a block of code. Optionally we can add a parameter list. A module define a scope for user identifiers. We can't perform a Goto from this block to any other line outside a module's code. The Goto works internal with a return and a request to insert again to the specific label or number. But if nothing found then nothing executed in the block, and those Goto are the same as Exit.

From a module we can use any identifier that is already defined ion that module or as global in the chain of calling modules. An exception exist for names that are exist forward in the code, inside the module, the labels for jumps and the subroutines an simple functions (like subs they exist at the end of the main code of a mdoule). Calling a module

means that we pass the current stack of values. Stack of values is a collection of items which we can read from top and we can put items to top using PUSH, or to bottom using DATA. Using PUSH and READ we use stack of values as LIFO. Using in an empty stack of values DATA and READ statements we use it as FIFO. Calling a module with parameters we place the parameters in reverse order at top of stack so we read the left most parameter at the top of stack. Lets see an example (supposed this written In another module:

```
Module One {

      Read X, Y

      Print X, Y

}

One 10, 20

Call One 10, 20
```

First we make the definition of Module One, as local to the module which we place the code above. Next we call the module by name only with two parameters, 10 and 20. The value of 10 is in top of stack of values. So when the interpreter execute the code inside the One, read 10 and 20 to X and Y and then print them to console. The second call use a Call statement. There is a non observant difference from these two calls. Internal there is a difference. They are two different things. Inside Module One if we call One using the name only we get an error because One isn't local to One. Using the Call One we can use recursion for modules.

Using the call by name only (without CALL statement) we can use named parameters:

```
One %Y=500, %X=20
```

If we miss the X value above we get an error, because X can't initialized and X isn't optional. When we have to Read a variable, which isn't initialized we have to provide a value. The value also set the type of variable, if is numeric. The string variable has the $ character at the end of identifier name, so A$ is a string variable, and only string value can be hold. We can pass a new name without raise an error. We can check if the name exist in the target module using the Valid() function which return true if the expression inside valid() has no error (also the expression executed). So we can use variable names as signals to module.

```
Module YouSay {

      If valid(Yes) Then Print "Yes", Yes

      If valid(No) Then Print "No", No
```

```
}

YouSay %Yes=100

YouSay %No=100

YouSay
```

Because we pass the current stack, we can first push parameters to stack and next we call the ONE

```
Push 500, 20

One
```

Now we get 20 and 500. In M2000 the caller has to prepare the stack and the callee module has to clear the stack properly. The interpreter never check at the calling of a module for a signature of a parameter list. This means that we can send different set of parameters, and we can check by code the envelope of stack of values, as a string with a letter for each item on stack marking the type. Also we can push data back, using the stack of values.

Lets make a different example, with two modules which change a value.

```
Module T1 { Read X : Push X**2}

Module T2 {Read X: Push X+10}

Push 10

T1

T2

Print Number
```

We push 10 and then we call T1 which get 10 and push 10**2 or 100. Then we get 100 and push 100+10 or 110. Then we print the top of stack of values using Number which pop a number or return an error if no number found on top of stack of values.

As a syntactic sugar we can make T1 module without placing the Read statement (interpreter place it at the construction of the module)

```
Module T1 (X) {Push X**2}
```

or without space between name and parameter list:

```
Module T1(X) {Push X**2}
```

But the last one isn't a preferable one, because when we mark a module name and search using F2 (up) or F3 (down) the code, the search utility not stop in a T1(name. Another named block is the Function block and for that block the second form is the right one, because we use the function in expression always with parenthesis.

Changing Code For a Module at Runtime

All local modules (and normal functions) can change code by applying a newer definition. Exception exist for those modules which are fixed for a reason. The reason maybe:

- The module passed in a module call (without Call)
- The module is a member of group and has a tag as Final
- A Function is a member of group and has a tag as Final

There are subs and simple functions coded at the end of a module or function code, which are used by searching the definitions in the code (and not executing definitions as Modules and normal Functions. The names for those searched from the end one time and memorized the starting point for the next call. These simple structures have no block { }, they use Sub Name() End Sub and Function Name() End Functipn. We will see that later.

Modules In Modules

We can put modules in modules but we can't call an inner module from any other module except from the parent module and exactly from the point where the definition of inner module executed before the point of call. Exception exist for Global module. We can make a global module, and we can call it from anywhere, until the module where we make the global one exit the execution. So in a local module we can make a global module and this one **shadow** any global with the same name. Interpreter always do this for same names: Local Shadows Global. Newer Global Shadows Global. And we can change a definition if we redefine a local module with a new local module with same name.

Also we can use IF statement to choose a definition for a module:

```
Module Two {
      If Number=1 then
            Module T1 {
                  Push Number**2
            }
      Else
            Module T1 {
                  Push Number**3
```

```
                }
        End If
        T1
}
Two 1, 10 : Print Number
Two 2, 10 : Print Number
```

The first call return 100 and the second 1000.

Passing a module in a module

There is a **decoration** mechanism, which pass modules in a module at run time. If the target module (where we pass a module) has a definition for same name module then this definition can't save the code and drop it. The module pass to the target module only for the call. We have to use the normal call (without Call). Because Call used for recursion for modules, we can't use the decoration mechanism for recursive call.

As we see from the example a module not only get parameters but also can return data in the same stack of values where the import done. The (x) in Cnvt2Fahrenheit is a Read x statement, and read from the stack (mentioned before). The Push statement return a value to stack.

In the module temperature, in the example, we have two dummy modules, which do nothing. So if we use them we push a number and the number just stay in stack until we get it (or in another case we drop it using Drop, or Flush which clear all items in stack).

```
Module Cnvt2Fahrenheit (x){
        Push Round(x*1.8+32, 2)
}
Module Cnvt2Kelvin (x){
        Push Round(x+273.15, 2)
}
Module Temperatures (title$){
        Module cnvtFrom {
```

```
            }
            Module convert {
            }
            Print Title$
            for i=-20 to 35 step 5
                  convert i
                  cnvtFrom i
                  \\ numbers pushed to stack of values
                  Print Number, Number
            Next i
      }

      Temperatures "From Celsius to Celsius"
      Temperatures "From Celsius to Fahrenheit" ; convert as Cnvt2Fahrenheit
      Temperatures "From Celsius to Kelvin" ; convert as Cnvt2Kelvin
      Temperatures "From Fahrenheit to Kelvin" ; convert as Cnvt2Kelvin, cnvtFrom
      as Cnvt2Fahrenheit
      Temperatures "From Kelvin to Fahrenheit" ; convert as Cnvt2Fahrenheit,
      cnvtFrom as Cnvt2Kelvin
```

We can pass any module in a module, ny decoration, without the need to use a local module with a target name like the convert as above. But we have know the name using then Module() function which return true if a module name exist (can be called).

Group of Variables

We can make a **Group** of values, using the Group statement, and we pass this group to a new module One. Group A is a named Group, a static in some languages, but here in M2000 is just a local group, because the group disposed at the end of run of current module (which we make the Group).

```
Group A {
        X=10, Y=500
}
Print A.X, A.Y
Module One (Z) {
        Print Z.X, Z.Y
}
One A
```

The parameter list (Z) with the Z replaced with a READ Z inside the module One.

Groups ia not used only as a data structure, but can contain code, operators and they are the user object for object orientated programming,

We can use Modules inside a group. So lets make a module which add 10 to X value of group. The block of Group isn't the same kind as a block of code. See that we can use comma between identifiers. Also we can put some expressions using external variables which they used only at the construction.

```
Group A {
        X=10, Y=500
        Module AddTen {
                .X+=10
        }
        Module ResetX {
                .X<=10
        }
}
```

Identifier .X is the same as This.X and when we place a new value we have to use <= because a simple = make a local value. The values X and Y are group values, not local

values. So now we learn that except local and global variables there are the group variables. A group maybe is global so the group variables are globals too. So a group variable is local where the group exist and is in scope in group modules by using the dot or THIS plus dot.

Passing Groups by Value

A module is a second class citizen, because we can't change the name. A group is more complicated. A group is an object, wiith data in variables and code in modules. When we make a named group we have something bound to a module, like a local variable. Groups can be copied to stack of values and can be read to a new name. Lets see that:

```
Module Three (Z) {
      Print Z.X
      Z.AddTen
      Print Z.X
      Z.ResetX
      Print Z.X
}
Three A  ' this is s remark at end of line: A is a group -see definition above
```

Defining a Group from another Group

Also we can do a simple assignment B=A and B get everything A has. B has to be a group or a new value. If B is anything else we get an error. B isn't a pointer of A. B is another named group. We use the definition for A from previous example.

```
B=A
Print B,X
B.AddTen
```

A and B groups erased when the parent (a module, or a function) ends execution, at any exit statement, or when execution find the end of parent block (a normal exit). Inside the parent block we can call other modules (local to parent or global) but this only pause the execution (in a call the module transfer the execution to another module, and when that module exits continue to next statement).

We see before a way to pass a module code to decorate specific module in a module. A group may have the same use, but expanded with the use of variables, as group members, to hold state, or to curry information. We see before that we can pass a copy of a group passing it by value. So the modules inside group can be used from the module. Because all Group objects have one type, the Group we can pass any group, but we have to maintain the interface. In M2000 there are not specific way to declare an interface. The group declare an interface as is. We can add members in a group but we can't delete them. We will see how we can place temporary members later.

Passing Variables by Reference using &

We can pass a values by reference using & to both parts:

```
X=10

Module AddTen (&Z) {

        Z+=10

}

AddTen &X

Print X
```

Some languages expect expressions or variable for a parameter, and when we pass a variable we have a by reference pass. In M2000 we always pass by value. A reference is a by value pass of a weak reference. A weak reference is what interpreter need to make a new variable which reference another, like two names shared the same data.
So a reference for a variable has two stages always: First a stage to produce a weak reference and second the hard link, where the new name show to the data of the referenced one. Between first an second stage, in M2000 can exist a delay of use. Using weak$() function we can get the weak reference to a string variable or array, because for M2000 a weak reference is a string type. The second stage may raise an error if the time the interpreter do the resolving of weak to normal reference found that the variable not exist. This can happened when we return a weak reference at the exit of a module for a variable for that module. So at the exit the variable erased, but not the stack of values. So there, in stack of values, may we have a weak reference to a non exist variable.

Look the example above. We get 20, X passing by reference to module AddTen. We can pass a group also by reference.

Interaction using by reference pass

Let see first a group passing by reference. The mechanism is different from passing by a pointer. The &Z is a string containing the weak reference of Z. When we call module DoSomething we pass the weak reference and a Read statement in module expect to make a reference variable &Z, but not know for what until the real reference resolved (we

can add "as group" to get a validation before continue but we leave this for now). We didn't see the Read statement in the code, but the statement entered by the interpreter at the construction of module, when the definition of module executed. So when the interpreter found that the variable we send to is a group, do something else from just make a reference as an ordinary variable. Every group has a list of members for variables and a second one for functions. A named group has the second list as pairs of names and reference number to an actual source in a structure for sources. The first list has pairs of names of variables (and other inner groups) and reference number to a structure for variables. So the interpret make a three part story on the group and first make a new group under the name of Z. Then make new variables (members) with new numbers according to list but for each one variable the actual data is at the original number (from the reference one). So the Z.X now is a reference of A.X. Also new modules defined with a copy of the modules from the reference one. The third part has to do with the methods of the group (and inner groups). These methods are modules, functions, operators, and any of three specific methods, the value, the set and the remove method. A group may have no methods, or any type of them. So now each method source copied to the referenced group, at source level, with a hidden part. For each method, the hidden part is the weak reference which used for This variable. So we get fresh methods but linked by a weak reference to the new group (where we say they belong to). Why this happen? Because the new group has everything the reference one, and maybe we can add members, including methods, or change source for modules. So the This may have more members from the referenced This. So we can pass a reference group, then add some members and at the exit the original group has the original members, and may have changes only to variables, not to methods.

So for the example, we can change a module (it is not made with Module Final variant, the Final used only in Groups) in module ChangeIt by using another Group statement using the same name Z. This is a way to add members and change methods. So now the interpreter change not the original source but the copied source inside module ChangeIt for Z.

Another M2000 specific characteristic is that all variables and modules of Z are not inside Z . To call or use them outside from Z we haven't to communicate with Z. As we see each method knows the group which refer (by the hidden weak reference). So for the example in group A the A.X is a plain variable. We can pass it by reference if we want. Also a function as a member of group can be passed by reference and the pass include the weak reference to group, so we can change the state of group, from a function which we pass to another module or method of another group.

Because all group members (there are private and public members) are out of the group, we say that this is an open object, or a Named Group. Groups have another phase, the close one. When a group became a float group, without a name, all members are inside the group, as a unity. M2000 interpreter always use a group as open object, through a mechanism which give a system name, and temporary deploy the members to the list of variables and modules. A float group (another name for a closed group) may exist for a tiny space of time, when we merge a named group to another named group, as a copy value used to pass as parameter or return value, and in data containers like an array for longer periods. We can use pointers to groups, either to named groups (which hold only the weak

reference) or to float groups (which are real pointers). Pointers can change to any type any time, but references to groups always reference named groups, the weak reference they resolved once. We can change a general reference to something else, once resolved, with the exception for groups because we have always a new group reference to members of the referenced group.

So someone think that a referenced group may change to something else, like in other languages, by changing the pointer. No for M2000, because references and pointers are different things. References based in weak references (the actual name of what we reference), so we can change only the state of the group, but not the source of referenced methods (we can change them for the time we process the referenced group, but the original sources not changed returning from a call). Pointers to groups is another story. We use them like the group is moving to the place we execute methods of it. They can change group to point, and If the group is a float one, then the number of pointers to it counts for erasing or not. A named group always erasing when the definition of it erased, not when a pointer not point to it. Named groups are like variables, they erased at the end of execution of the place they defined, like a module or a function or a sub or a simple function (like sub), or block for temporary definitions the For obj block, a special variant of For statement .
Pointers passed by copy to new pointer. We can pass a pointer by reference using &. So by using reference we have one pointer (each show the original), and by value we have two (the original and the copy)

We have to remember that an open group always is a named group (has a name). First example has a Group with a method AddTen, all members are public by default (we will see how we make private later)

```
Group A {
      X=10
      Module AddTen {
            .X+=10
      }
}
Module DoSomething (&Z) {
      Z.AddTen
}
DoSomething &A
```

```
Print A.X
```

So now we see that passing A in ChangeIt by reference as Z, and execute the Z.AddTen we change A.X

The second example show how to change a method in Z in ChangeIt, and execute it. The new method read the Z.X member which is the A.X (as we see in previous example). At the return from the execution (or call) to ChangeIt we do a call to A.DoSoemthing, which use the original source on the same A.X.

Here we see that the two DoSomething modules, get a m as parameter. The Z.DoSomething can be anything, using any number of parameters. In M2000 the number of parameters and type never checked from call side. The method we call may use programming aids, to inspect the received parameters and do anything, or we can leave this to raise an error, just using a Read statement expecting to find the right type of values. The (m) changed to Read m. If not value found for m then an error produced. We have to place a value to m otherwise we get an error: ***variable m not initialized***.

In Groups we can write X, Y with no values and the initialization happen on the construction with the default values (a zero of type double for numeric variables, and an empty string for string variables), The major types numeric and string are always visible by the name. A string type has $ in the name, so a X$ is a string type. We can have X and X$ as two different variables, a number and a string.

Because Groups may have value (from a specific method called value), we may have Groups with a string name, say Group A$, but always interpreter make two names, the A$ and A for the same group. The first name A$ used for the value and the A for the members (as we learn before, the A is part of the weak reference for all methods of the group).

By default a Group return a copy of it as a float group. Using the Value member we can change it, but the Group() function bypass the Value members and perform the copy. Also the &A reference pass the group reference, not the value of A , of some kind of reference (which not exist as a value but as a function which return a value).

```
Group A {
    X=10
    Module DoSomething (m) {
        Print m+.X
    }
}
```

```
Module ChangeIt (&Z) {
    Group Z {
        Module DoSomething (m) {
            Print m+.X**2+100
        }
    }
        Z.DoSomething 2
}
ChangeIt &Alfa
A.DoSomething 2
```

So let see interaction of two groups. We have AA with AA.SendMessage to a Z group. The Z must have a method callback. If not we get an error, at the time we call it. We call Z.callBack (the case of letters isn't important for M2000, although exceptions exist for this rule), and we pass the &.X which is &This.X, so we pass the &AA.X but if we place &AA.X we get error, because AA.X expected to be a local to module variable. So the &.X is perfect for the call to refer to group member X.

We have a second group BB which have a method CallBack and we see that inside this method, we change the referred &.X as Z and save state altering the this.acc member just using .acc using operator ++. If we want to change the value assigning a number to .acc we have to use <= because in M2000 the = also make new variables and .acc=100 make a local variable acc with value 100, and the local value has priority for module. The .acc<=100 make the interpreter to find the .acc, because the operator <= not defined any variable, so one of two things happen, to find it or to raise an error. Using the = operator, first check if a local variable exist, and if not make one. So the = operator defines variables. The ++ operator expect to find the variable, so if no local variable exist check that this is a group member and check if the member exist. A abc<=100 check for a global one. So the <= operator change values to globals and members of groups (if that type of members may produce local variables using the = operator). We can use arrays as members to groups, using Dim A(10) we make A() with ten items (numeric type, with an empty value for start). In anarray because defined by DIM we not use <= for assign values to items of it, and this happen to global arrays too.

```
Group AA {
        X=10
```

```
            Module SendMessage (&Z) {
                  Z.callBack &.X
            }
      }
      Group BB {
            acc=0
            Module callBack (&Z) {
                  .acc++
                  Z+=10^.acc
            }
      }
      AA.SendMessage &BB
      Print AA.X, BB.acc
```

We get 20 and 1. We just perform this **AA.SendMessage &BB** and we get new state for AA and for BB.

A float Group

Another kind of group is the volatile one, or float group. The volatile one has no name, but has a position in one container. In the next example we use the two groups AA and BB and we make a copy of each in two positions in an array A(), so this array is the container for objects. Internal each item of an array is a Variant type of the OLE system of Windows, which can hold many data types, including objects, numbers and strings. If we didn't offer an initial value for each item of array we get an array of Empty value. We can check the type using **Type$()**.

In this example we will see the **For obj {} block**. First we make the named groups. We will see how we avoid to use named groups if we didn't need them to stay in the code, like static objects in other languages. We use Class to make functions which return float groups. So for the example we use Class function for AA and Group for BB.

Next we make an array of 10 items (Empty type), and we place to A(3) the return value of AA(), the group factory. The class statement produce a function named AA(). If a class is a

member of a group then the function is member of the group. If a class defined in a module or a function then it is global until the module or function end execution. There is a rule for globals/locals: Local identifier shadow any global, and a newer global hide any global unless a local hide it (so a newer global definition just hidden by a local with same name).

```
Class AA {
        X=10
        Module SendMessage (&Z) {
                Z.callBack &.X
        }
}
Group BB {
        acc=0
        Module callBack (&Z) {
                .acc++
                Z+=10^.acc
        }
}
Dim A(10)
A(3)=AA()
A(4)=BB
For A(4), A(3) {
        ..SendMessage &This
        Print ..X, .acc
}
For A(3) {
```

```
            For  A(4) {
                    Print .X, ..acc
            }
    }
    For A(3) {
            Temp= A(4)
            Print .X, Temp.acc
            A(4)=Temp
    }
```

So now we see that For statement with objects (from array items) get the float groups and opened giving them names (hidden) and bound member names combined with hidden names to global list of variables and to global list of modules and functions. We address each member by using dots. One dot for first and two dots for second. The This statement used for the first one. We can use folded For also, and the dots follows the number of objects counting the objects from these For statements

When a for statement exit, the opened groups closed again rewriting the values inside private lists. The third For {} shows us that we can use a Temp name to get the A(4) and then we can pass it back to A(4), making a float group from Temp. At exit of For object statement any new definition erased. So Temp erased (and the hidden opened groups).

We can use a For This {} block in a module which isn't a group member to make reference. A reference can't get new value, but we can make new references in a part of module using For This {} and erase them to make them again in next iteration. The same happen with the Temp group in the example. Because an array may have any type of group, some of them with same member names (like interface, here is a logical interface, if we use it for two or more different groups), the Temp=A(i) if Temp exist as Group merge the new float to it, so we get a blob of groups in one group, a total mess, so we need the Temp to be a new one, and For Object {} structure used for it.

If we make an array of same group, then the methods are being a part of array (is an object too), so we didn't get copies of methods for each item. In an array with any type, a group has all the members as float group, and we can change methods:

```
    Class Alfa {
            X=10
```

```
        Module Beta {
                Print .x
        }
}
\\ making A() of type Alfa()
Dim A(10)=Alfa()
Print A(3).x=10
A(3).x++
A(3).Beta      ' print 11
For A(3) {
        class z {
                M=100
                module Beta {
                        Print .x^2
                }
        }
        This<=z()
        .Beta  ' print  121
}
A(3).Beta             ' print 11  methods are locked.
Print A(3).M=100   ' but not variables,  M adding to A(3)
Print Valid(A(4).M)=False  ' only A(3) has a member M

\\ Now we make another without a group
```

```
Dim B(10)=0&            ' we can give a value, say a zero of type long.
B(3)=alfa()   '  arrays can get any value, including a float group
Print B(3).x=10
B(3).x++
B(3).Beta      ' print 11
For B(3) {
        class z {
                module Beta {
                        Print .x^2
                }
        }
        This<=z()
        .Beta  ' print  121
}
B(3).Beta            ' print 121
```

More about Variables

In M2000 language, can exist on same scopr **A, A%** and A**$** as three different names (identifiers). The two A and A% are numeric variables, and A$ is a string variable. The A% is a variable with automatic integer rounding at half unit, so if A% is 1 then A%/=2 make temporary ½ as 0.5 which turn to 1. The type of A% can be any numeric type.

Operator and Assignment together

Statements like A+=10 add 10 to A. Also A$+="ok" add "ok" to end of A$. For numeric variables +=, -=, *= and /= change the value, also the ++ and – change value by one.

So A++ is a statement which adds one to A. There are two more, the ~ and -!.

The A~ change to opposite Boolean numeric value or Boolean value if A is Boolean. So if A has value 5 (non zero) means true, so A~ turn A to zero. If A has value zero then A~ turn A to -1 (the true value as numeric).
The -! used to change sign. So if A=5 then A-! as statement turn A to -5.

These operators can't be used in expressions. So the ~ operator and assignment is different from ~ operator for string comparisons (the like operator).

We can use |div |mod |div# and |mod# (euclidean) so A|div 10 is the same as A = A div 10

Tuple of values

A variable may content a value or a pointer to an object. Let say that AA is a new name. This make the AA as a pointer to an array of three items

```
AA=(1,2,3)
```

This make AA as a pointer to an empty array. These type of arrays are like Tuple, but may change values and size.

```
AA=(,)
```

In the next example the AA has a tuple of two tuples. A BB=AA make BB to point to same object as AA. We can find the type of a variable using the Type$() function.
The $ in Type$() means that this function return a string. If we make a user function and we want to use it in a string expression we have to place a $ as the last character before parenthesis.

```
AA=((1,2),(3,4))
```

Tuples have special functions like the #sum(). We will see the tuple or Auto Arrays later.

```
AA=((1,2),(3,4))

Print AA#val(0)#val(1)=2

Print (1,2,3,4,5)#sum()=15, AA#val(0)#sum()=3, AA#val(1)#sum()=7
```

We can change a value on a tuple using Return statement. The Return statement do a lot of things, depending of the type of frist parameter. Without parameter is a return from a simple Gosub (like in Basic).

```
A=("George", 0)

Return A, 1:=100  ' change value

Print A#val$(0), A#val(1)   ' the second type is numeric         George 100
```

```
Return A, 1:=A#val(1)+1

Print A   ' this is the same as Print A#val$(0), A#val(1)          George 101
```

Another way to change values as the example before (A is a pointer to tuple)

```
A=("George", 0)

A+=100

Print A                 ' George  100

A++   ' add one to all numeric values in tuple.

Print A                 ' George 101
```

Using a reference to Array (so A and A() and A$() is the same tuple). Any array may behave like a tuple, and each tuple may behave like array, but aren't the same.

```
A=("George", 0)

Link A, A to A(), A$()

A(1)+=100

Print A                 ' George 100

A(1)++

Print A                 ' George 101

A$(0)+=" Karras"

Print A                 ' Karras 101
```

Assign Multiple Variables

We can assign multiple values like this

```
(A, B$) = (1000, "Hello")
```

The mechanism is simple the previous statement are equal to this one:

```
Stack New { Data ! (1000, "Hello") : Read A, B$ }
```

We can use New to shadow variables like this **(New A, B$)=(1000, "Hello)** the New clause, using in read. Also we can use types **(A as decimal, B$)=(1000, "Hello)**

User Functions

A function is like a module but may return a value or more than values in an array. We can call functions in expression or like modules using Call statement. A module get the stack of value from the caller, and the same is true for functions which call as module using Call statement. Functions in expressions have own stack of values. Because of this we can pass any number of arguments, and if we didn't use them in the function the stack dropped with any value left on it.

```
Function DoubleValue(x) {
        =x*2
}
Print DoubleValue(100)=200
Print DoubleValue(DoubleValue(50))=200
Def DV(x)=x*2    ' this is the same as DoubleValue()
```

Using Def statement can make one line function definition. In either form Interpreter take the (x) and add a first line in source of function as Read x.

Program: The Fibonacci sequence

This program shows as the first 10 items of the sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 Each number in a new line on console. The fb() function can call itself (a recursion call). Also we have the 1st item as 0 and the 2nd item as 1, by default. We use exit to exit soon from function. The = as statement is not an exit. Maybe we want to return a value but after that we want to change a by reference parameter. Here x is a by value parameter. As we see before a number by default is a double type. When a variable get a value get the type also. If later we pass another type to that variable a conversion happen (we get an oveflow error if type can't hold the value). The x here in fb() function created each time we call the function, and take the appropriate type also. In some cases the calculated value can change to a bigger numeric type so that new type can be the type of those x and the return value may also change.

In M2000 we didn't say anything for the return value of a function, except if the function used in a numeric expression or in a string expression. We know this looking the name, so here without $ at the name we know that fb() used in a numeric expression.

Because a fb() can be an array too, we didn't use a local array with same name as local function, or we have to use * so a fb(*) is always a function. We can use fb(*100) to pass 100 as parameter, the * character dropped by the interpreter.

```
Function fb(x) {
```

```
        if x<=1 then =0 : exit
        if x=2 then =1 : exit
        =fb(x-1)+fb(x-2)
}
For i=1 to 10
        Print fb(i)
Next i
```

We can change the fb() function to return a numeric type based on input type:

```
Function fb(x) {
        if x<=1 then =x-x: exit
        if x=2 then =1% : exit
        =fb(x-1%)+fb(x-2%)
}

k=fb(10%)
Print type$(k)="Integer"
k1=fb(10&)
Print type$(k1)="Long"
k2=fb(10~)
Print type$(k2)="Single"
k3=fb(10)
Print type$(k3)="Double"
k4=fb(10#)
Print type$(k4)="Currency"
```

```
k5=fb(10@)

Print type$(k5)="Decimal"
```

If in the above example change the literals 1% and 2% (Integer type 16bit) with 1 and 2 (Double) then only the first three Print return FALSE. This happen because Currency and Decimal are above Double. So the last two upgrade the return value to Currency and Decimal.

We use k, k2 to k5 and not k for all test because a variable get type from first assignment. So if in k=fb(10#) we have a k as type integer then the fb(10#) return Currency type but converted to integer type. Overflows may occur and raise error, when we assign a value to a variable. In an expression we get the arithmetic type which can hold the result (double).

Variadic Functions

A variadic function get any number of arguments, in current stack (functions have own stack). So we use here Empty which turn to True if current stack is empty, and then we use Number to pop a numeric value (or we get error), also we count the numbers

```
Function Variadic {
        acc=0
        many=0
        While not Empty
                acc+=Number
                many++
        End While
        if many>0 then =acc/many
}

Print Variadic(1,2,3,4,5,6,7,8,9,10)=5.5
```

Or more compact:

```
Function Variadic {
        acc=0
```

```
            many=stack.size
            While not Empty
                  acc+=Number
            End While
            if many>0 then =acc/many
      }
```

Or using tuple. Here first we check if stack is empty. If not then we check the Envelope$() which return by a letter the type of stack item. So here we want all items as N (numbers). So we make "NNNN...N" at the length of stack.size and we compare it with Envelope$(). If this is ok the we get all current stack to as stack object (this do the [] symbols) and converted to array using the array() function, and finally export the pointer of the array to acc. Next we get the sum of all items using acc#sum() and then we divide it using the length of acc (In items).

```
      Function Variadic {
            if Empty then exit
            if Envelope$()=string$("N", stack.size) then
                  acc=array([])
                  =acc#sum()/len(acc)
            end if
      }
      Print Variadic(100, 200, 300)=200
      m=(100,200,300)
      Print Variadic(!m)=200   ' the !m copy the items of tuple to stack of values.
```

Functions May Return Multiple Values

```
      Def Three(X)=x**2, x**3, x**4
      A=Three(2)
```

```
Print A#val(0)=4, A#val(1)=8, A#val(2)=16


\\ Def Three(X)=x**4, x**3, x**2   ' reverse order by hand
\\ using a tuple and a tuple function to reverse the order
Def Three(X)=(x**2, x**3, x**4)#rev()   ' reverse order
\\ See the ! operator for tuple and stacks
Push !Three(2)   ' place at top of stack each item in tuple
Read X, Y, Z
Print X=4, Y=8, Z=16


Def Three(X)=x**2, x**3, x**4  ' not need to reverse it in a call
Module Disp {
      Read X, Y, Z
      Print X, Y, Z    '   4    8   16
}
Disp !Three(2)
```

There is a check for the first value if is numeric (or object), If function is numeric major type, and for string ff it is string major type, and if not then we get an error.

A tuple is like a numeric value, because we place the pointer of tuple in a numeric variable (although the variable from that time return the tuple not the numeric pointer which is hidden to programmer). The following examples use Print to print a tuple to columns in console layer. If we place parenthesis for tuple in function alfa$() we get an error because tuple isn't a string.

```
Function Two(a$, x) {
      =(a$+a$, x+x)
}
Print Two("hello", 1)
```

```
Function alfa$ {
        ="hello","M2000","Programming","Language"
}
Print alfa$()
```

We can make something else for alfa$() to return a multi type array (tuple is like array but not exactly the same due to interface we use). We define an empty string array, and we feed it with a tuple, so now the tuple may have any type for items, and we return a string array which interpreter accept it without error, because an array with a name with a $ is a string major type. Arrays with parenthesis works like values. We can assign a new value, by getting a copy of items. So here we get a copy of tuple, not a pointer copy.

```
Function alfa$ {
        Dim ret$()
        ret$()=(100, "hello","M2000","Programming","Language")
        =ret$()
}
Print alfa$()
```

Functions Passing by Reference

Functions are different from Modules because they are used in expressions. Also there is another difference: Functions can be passed by reference.

```
Function Alfa$(a$) {
        ="ALFA "+a$
}
Function Beta$(a$) {
        ="BETA "+a$
}
```

```
Module Doit (&func$()) {
      Print func$("ok")
}
Doit &Alfa$()
Doit &Beta$()
```

How we can pass a function which use recursion calls? We can use Lambda() and Lambda$() for numeric or string functions as an alias for the current name.

```
Function fb(x) {
      if x<=1 then =0 : exit
      if x=2 then =1 : exit
      =Lambda(x-1)+Lambda(x-2)
}
Module ShowTen (&f()) {
      For i=1 to 10
            Print f(i)
      Next i
}
ShowTen &fb()
```

A function can be used as module using the Call statement, with a difference. If we return a non zero value then an error raised, with the return value (numeric or string) as the error information. Also the stack of values from caller passed to function the same way as in modules. See in this example we didn't put the = statement to return a value (So we return the default one).

```
Function CallFunc {
      Print "ok", Number
}
```

```
Call CallFunc(100)
```

As special call is the Call Local. Using this call we make the function as part of the calling code, so we have the same scope for identifiers. The next example return 100 and 11, because in function DummyName X is the same as the X from the Caller. This type of call used from objects with events.

```
X=10

Function DummyName {

        Print X*X

        X++

}

Call Local DummyName()

Print X
```

As we see functions are not first class citizen, except for passing by reference. There are two object types as functions. The first one is the Lambda object. The second one it the Event object. Also there is a simple form for functions, the simple function form.

Use of simple functions

Simple functions are lighter versions of normal functions, and always have one definition (we can't change these dynamically). Like subroutines, simple routines are in same scope as the caller's scope. They read the same variables, and we can use Local to create variables and arrays.

So a Local A(1 to 10, 1 to 10) make a local array of 100 items. Same as Local A(10,10).

Read about them in Old Style programming>Simple Functions

```
Module Alfa {

    Print @alfa(100)  ' 200

    End

    Function alfa(x)

        =x*2
```

```
        End Function
    }
    Module Beta {
            Static M=10
            Print @alfa(M), ' 20 18 16 14 12 10 8 6 4
            M--
            If M>1 then Call Beta Else Print
            Clear  ' clear static
    }
    Alfa
    Beta
    Beta
    Call Alfa
    Call Beta
    Call Beta
```

In the example above we call Alfa two times (one by using Call), and Beta two tines. Using Call Beta from Beta we can call recursive the Beta Module. The static variables from recursive call are accessing from all inner calls. Static variables are not part of name space. Beta call @alfa() which is not part of the block of code of Beta, but because exist in the original code where Beta defined, interpreter find it and link it to execute it. This happen without changing: the execution object (static variables stay there) and the name space (so local variables in Beta are visible to function @alfa(), but for now we don't have any). The name alfa() not exist in list of functions (the normal functions) and the call to @alfa() never use this list, so maybe we have the same name for a normal and a simple function, and maybe the normal is a global one, so a call to @alfa() is always a call to a simple function, not a defined in the list of functions/modules. When we write the code if the name we choose get a color showing a know internal function is an indication that we don't call them through @name() mechanism (which used to call the original internal functions). The following example demonstrate the use of replacing an inner function like Sgn() (return -1 for negative, 0 for zero, 1 for positive number). As we see the change happen for the CheckRegular not for the inner module not for the parent of CheckRegular. The @Sgn() always use the original internal function. So we can't use it for calling a simple function (we have to make it as normal only as in the example)

```
Module CheckRegular {
      Print Sgn(2)=1
      Function Sgn(X) {
            =X*2
      }
      \\ here we call @Sgn() the original function
      Print Sgn(2)=4, @Sgn(2)=1
      Module Inner {
            \\ here Sgn is the original
            Print Sgn(2)=1
      }
      Inner
}
CheckRegular
Print Sgn(2)=1
```

Simple functions can't change code like normal functions using a new definitions (exception exist for some normal functions in Groups which can't change definition if we define them as Final).

Also we can't choose a definition before the use of a function, using an If statement to apply a specific definition. A normal function exist when the definition executed, but a simple function exist always, like subs, as hard coded at the end of the current code, or the original code where the code first exist.

```
If rnd>.5 then
      Function alfa(x) {
            =x**2
      }
```

```
Else
    Function alfa(x) {
        =x**3
    }
End If
Print alfa(10)
```

If we place two times a simple function we use the last one in the code. We can place it only for tests, so we can change the last alfa() with another name say alfa1(), and call the code again, but the change can't be done from code. We have to change it by editing the code in an editor. See the End after the Print statement, it is optional.

```
Print @alfa(2)=8
End
Function alfa(x)  ' never used
    =x**2
End Function
Function alfa(x)
    =x**3
End Function
```

Another difference is the lack of a name space. Normal functions, like modules have own scope. Subs with parenthesis and simple functions haven't name space, they use the scope of module or function where they called. We have to use Local for making local variables.

Another difference from normal functions is the use of stack of values. When we call a normal function in an expression a new execution object get the code to execute, defining a new stack of values (and a new name space, so all variables are local to function). A simple value use the same execution object from the caller, and the same stack of values. So the simple function is the only way to call in an expression and feed with the current stack without replace him.

```
Module CheckIt {
```

```
        Flush  ' empty stack
        counter=0
        Print @alfa(10)=3628800, counter=10
        While not Empty
            Print Number  ' 1 2 3 ... 10
        End While
        End
        Function alfa(x)
            =If(x<2->1, x*@alfa(x-1))
            Data x   ' add to the end of Checkit stack of values
            counter++  ' use the same name space
        End Function
    }
    Checkit
```

Lambda Object

The Lambda object is a value type which include two things, a block of code (the function) and a list of closures. A lambda object has two identifiers, one for the object (the variable) and one for the function. Because there are two major kinds of return values, the numeric and strings we can make lambda function as numeric using Lambda, or string function using Lambda$. Closures are copies of actual variables and can change values. So lambda functions in M2000 can be used to hold state. A simple example where we change the fb() function with a fb lambda object. We use it like a function, passing function by reference. We can also use passing object by reference and passing object by value too.

```
    fb = Lambda (x) -> {
            if x<=1 then =0 : exit
```

```
        if x=2 then =1 : exit
        =Lambda(x-1)+lambda(x-2)
    }
    Module ShowTen (&f()) {
        For i=1 to 10
            Print f(i)
        Next i
    }
    ShowTen &fb()
```

Without the use of recursion. The fb lambda return a new lambda. In module ShowTen we call the f() function ten times and each time we get one item from the Fibonacci sequence. We use the function stack of values to push the sum a+b and then to read to b. Another way to make this without using stack of values: **Swap a, b : b+=a**

```
    fb = Lambda -> {
        = Lambda a=0, b=1 -> {
            =a
            Push a+b
            a=b
            Read b
        }
    }
    Module ShowTen (f) {
        For i=1 to 10
            Print f()
        Next i
    }
```

```
ShowTen fb()
```

Any time we place the f to a variable (Say d) we place not only the code of function but the list of closures as a copy. So a d=f store to d the state of f, and anytime later a f=d restore the state. Closures can be lambda objects too.

Example: Permutation Step

The next example use a construction of nested lambda functions by using closures and tuple for data structures. The goal of this example is to use a final function to pass a tuple of numbers or strings (or mix of them) and for each call we get the next permutation, until a passing by reference variable change value at the last permutation to indicate the end. The PermutationStep() get a tuple by value (is a pointer so it is like by reference) and return a lambda function which prepare inside. The result lambda function say StepA() get by reference a variable (the indicator for the last permutation), and return a tuple for current permutation.

Inside lambda function we use three functions, the CONS(), CAR() and the CDR(). The first add two or more tuple to one bigger, the second extract the first value from a tuple, the last one return a tuple from a tuple without the first value.

Let see how PermutationStep works. First we make c1 as a lambda which get two things, a f by reference and a as pointer to a tuple (we know that, and interpreter at run time find this). This c1 assign f the true value, so c1 used for the last permutation.

Next we get m as len of tuple a, and then we copy c1 to c. So while m>1 we make a new c1 with two closures, the c2 which hold th last c, and the m which is a pointer to an empty tuple. Also the parameter list is a &f and an a as pointer to a tuple. After making the new c1 we pass the c1 to c and we decrease the m value and loop or exit accordingly the condition in the While.

The last part return a new lambda with three closures, the c (which have a series of lambdas in closure space), the p for counting, the a as pointer to tuple. So only one parameter we have here, the &f (to indicate the last permutation).

When we call the function StepA we pass the &k as &f and let see what happen: The m is empty so m points to a. Next we have the = statement to return a construction of two tuple, the first item of m and the return from c2 (which is the same as StepA except the last one, which is different). So from that point f maybe is true, and this reset f to false and p increment by one. Because we use the **If then** in one line the part after the then skipped if we don't have an Else clause and we have false condition in if. So after the p++ we place a rotation of tuple members and a new if statement to check if p equal len(m) so the f get a true value. So if all internal c2 return f as true, we have the last permutation.

```
Module StepByStep {
```

```
Function PermutationStep (a) {
      c1=lambda (&f, a) ->{
            =car(a)
            f=true
      }
      m=len(a)
      c=c1
      while m>1 {
            c1=lambda c2=c,p, m=(,) (&f, a) ->{
                  if len(m)=0 then m=a
                  =cons(car(m),c2(&f, cdr(m)))
                  if f then f=false:p++:  m=cons(cdr(m), car(m)) : if p=len(m) then
p=0 : m=(,): f=true
            }
            c=c1
            m--
      }
      =lambda c, a (&f) -> {
            =c(&f, a)
      }
}
k=false
StepA=PermutationStep((1,2,3,4))
while not k {
          Print StepA(&k)
```

```
        }
        k=false
        StepA=PermutationStep((100,200,300))
        while not k {
                Print StepA(&k)
        }
        k=false
        StepA=PermutationStep(("A", "B", "C", "D"))
        while not k {
                Print StepA(&k)
        }
        k=false
        StepA=PermutationStep(("DOG", "CAT", "BAT"))
        while not k {
                Print StepA(&k)
        }
    }
    StepByStep
```

As we see above we don't have a recursion, but we have stored lambda functions as closures objects with state as closures too. This kind of programming cannot produced with ordinary programming paradigm. We can use recursion in lambda functions. The closures for each recursion call are the same, like they are global, but are local to all chain of calls.

```
    k=lambda m=10 (x) -> {
            m--
            if m<=0 or x<0 then =0: exit
```

```
            =x+lambda(x-1)
        }
        n=k
        Print k(100)=864
        Print n(3)=6, n(5)=14, n(10)=0
```

A lambda function may return a string like this:

```
        z$="0123456789ABCDEF"
        k$=lambda$ z$, n=0 (m) -> {
            if m=0 and n=0 then ="0" : exit
            if m=0 then  exit
            n=1
            = lambda$(m div 16)+mid$(z$, m mod 16 + 1, 1)
            n=0
        }
        Print k$(0)="0"
        Print k$(65535)="FFFF"
        Print k$(0xF00F)="F00F"
        Print k$(0xFFFFFFFF)="FFFFFFFF"
```

We can make the lambda like this **k$=lambda$ z$="0123456789ABCDEF", n=0 (m) ->** { }

Or a more sophisticated example:

```
        k$=lambda$ (m, b as integer=16) -> {
            if b<2 or b>16 then error "base out of range"
            if m=0 then ="0" : exit
```

```
            z$="0123456789ABCDEF"
            =lambda$ z$, b (m) ->{
                  =if$(m=0->"", lambda$(m div b)+mid$(z$, m mod b + 1, 1))
            }(m)
      }
      k=lambda (m$, b as integer=16) -> {
            if b<2 or b>16 then error "base out of range"
            m$=trim$(m$)
            if m$="0" then =0 : exit
            z$="0123456789ABCDEF"
            =lambda z$, b (m$) ->{
                  =if(Len(m$)=0->0, lambda(mid$(m$,2))+(instr(z$, left$(m$,1))-
1)*b**(len(m$)-1))
            }(m$)
      }
      Print k$(0)="0", k("0")=0
      Print k$(65535)="FFFF", k("FFFF", 16)=65535
      Print k$(0xF00F)="F00F", k("F00F", 16)=0xF00F
      Print k$(0xFFFFFFFF)="FFFFFFFF", k("FFFFFFFF", 16)=0xFFFFFFFF
      Print k$(100, 8)="144", k("144", 8)=100
      Print k$(100, 2)="1100100", k("1100100", 2)=100
```

Event Object

The Event object has two things: A signature and a list of functions. A call to an event is a call to all functions in the event object. Here is the only part of M2000 where the signature of parameter list matters before the call (all other parts need to matter in the callee). When

we call the event we pass specific parameters (by value or and by reference) to each function on the list. Functions may added anytime, or can be deleted anytime. Also an event object maybe enabled or not. All functions called like a Call to function (without returning a value), like a call to a module, but the stack of value is new for the call. We can add simple functions (state can change for global variables), or dummy functions (part of a module, as a call back, to store state to module) or functions members of groups, which can store state to group variables.

An event object is a first class object. We can pass it by value or by reference, and we can copy it to another event object (or new name to make it an event object). Event objects as a group member in a Group is a pointer to event object. Because we call multiple functions at once (in one command, but one after the other), we say we make a multicast call.
A variant of event object (without use of signature) can be used for monitoring the event messages from user forms (windows) from M2000 windows manager.

Except of these kind of events, groups may have light events, which are a mechanism to call an event service routine from a named group. So light events are not objects. They don't use signature and has no multicast functionality.

```
Flush ' empty stack
Print Stack.Size
Event A {
    Read message$, &z
    Function {
        ' this is the default function (optional)
        Print message$
    }
}
x=10
Call Event A, "ok 1", &x
Event A Hold
Call Event A, "ok 2", &x
Event A Release
Call Event A, "ok 3", &x
```

```
Function Addme {
        Read m$, &a
        a++
}
Event A new Addme()
Call Event A, "ok 4", &x
Print x=11
Event A drop Addme()
Call Event A, "ok 5", &x
Print x=11
Function callback {
        Print Stack.Size
        x+=10
}
Event A new lazy$(&callback()), Addme()

Module Inner (m) {
        z=50
        Call Event m, "Inner", &z
        Print z=51
}
\\ now A has a callback
Inner A
Print x=21, Stack.Size
```

So when we call the Inner module we pass the A event as a copy. When we call the event from inner, we call a part of outer module, named callback, which act as part of module and not as a function. So we can change the x variable. Because we can add or drop functions by logic we can change the response of a module. A callback is like the pass by reference, but now we pass code not only variables. We see before a call from a module as a group member passing by reference another group and change state to each group. Now we place the event object which hide the functions inside, and only the parameter signature used to perform the call. Also the call maybe hold or released, so a Call Event maybe not work, but no error raised.

The lazy$() function make functions callbacks. We can put a simple expression or a multiline code using a reference to a function (we do the last in the example above). Here is an example with a simple expression:

```
x=10

z=5

Push Lazy$(x**2+z)

Read &a()

Module TestMe (&b()) {

        Print b()

}

TestMe &a()

Print a()=105

x=100

TestMe &a()

Print a()=10005

\\ we can do the following but not in the module TestMe

Print eval("x**2+z")=10005
```

Light Events in Groups

Here is the use of light events for groups. See that the service routine has an underscore, not a dot between name of group and name of event. Events from COM objects also has the same type of service routines. Events using event object have name dot and event name.

In the following example we can use Events or Event and then we have to put a list of names as strings. That names are the light events. The B_alfa is the call back, or the event service routine. The Inner2 pass the group as a copy, the callback also copied. Just change B_alfa to B_alfa1. The code has no problem, the call event "alfa" may not have a service routine.

```
Group WithEvents B {
      Event "alfa"
      Module doit {
            call event "alfa", 100
      }
}
K=10
Function B_alfa {
      Print number, K
      K++
}
Module Inner (&C) {
      C.doit
}
B.doit
Inner &B
Module Inner2 (C) {
      C.doit
}
```

```
Inner2 B
```

The last example use an event to check the messages from a form. The statement Stack display the current stack (is the stack from event). The method Show with parameter 1 is the modal show for a form (has the loop inside waiting to close it). Because the focus is in the modal form, we have to refresh the console window, when we change it, using Stack.

```
Declare Form1 Form Event B

Function Dummy {

      Stack  ' just display stack

      Refresh

}

Event B new &Dummy()

Method Form1, "Show", 1

Declare Form1 Nothing
```

The Document Object

A Document object is an object which used with statements and functions for documents as object and in string expressions as string. For rendering text we have to use Report, or Print #-2, DocVar$ which use console like a file to output text, with interpretation of special characters, and render as non proportional font (for any font). The Print #-2 used for Text Art too, and not performing justification or column adjustment.

A document object has each paragraph as separate string in a double linked list. For speeding operations when displaying in an editor, each paragraph may split in different strings depends on wrapping. This object used in internal editor and in EditBox control.

The Len() functions get an expression but Doc.Len() get a Document object, so we get the length without any copy of the string.

The Report statement can render document and string variables for all lines, or part of lines, and also can do a dummy render only for counting the lines needed for rendering for the current layer using current font attributes and justification mode (4 modes, Left, Center, Right and Full, using 0,1,2 and 3 as codes). By default 0 (word wrap and left justifiy) used if no number provided as first parameter. The left margin obtained from the cursor position in row. The max width or a line by default is the width of the layer but we can put another number after the string to render and if is lower than 1000 then the number is the characters width or from 1000 then the number is in twips. Report has more optional

arguments, for the first render line from where we render the text, the number of render lines to display and a document to assign these lines (and int this case not to display).

Report statement in displayed layers (not in printer layes) hold the diplaying after ¾ of performing scrolling lines, and wait for space bar or mouse button to continue. In case of waiting threads and events can be executed.

We can sort lines (providing the character position on each paragraph from where we start comparisons). We can save to file, merge to file, load from file and append from file. We can load in any of UTF-16LE, UTF-16BE, UTF-8, and ANSI using Linux or Windows new line character. Also there is an auto encoding recognition, so when we save back a file the saving happen to the same encoding. BOM mark also used for recognition of encoding if exist, but there is an algorithm to find the encoding without using it.

We can count words, and unique words, very fast also we can get the words as result. The same counting happen when we press F9 in editor an in EditBox control.

```
a$={HELLO

DUDE

}  ' a$ is a string

Dim B$(10)

Document A$, B$(2)

\\ now a$ upgrade to Document, also B$(2) upgrade to Document

a$="TEST1"

b$="TEST2"

Report A$

Report B$

Print Doc.Len(A$),Doc.Len(B$)

\\ we can't erase using A$="" (it is append to document, not assignment)

Clear A$, B$(2)

Print Type$(A$), Type$(B$(2))
```

Document statements:
APPEND.DOC, EDIT.DOC, FIND, INSERT, LOAD.DOC, MERGE.DOC, OVERWRITE, SAVE.DOC, SORT, WORDS

Document functions
DOC.LEN(), DOC.PAR(), DOC.UNIQUE.WORDS(), DOC.WORDS(),

Document read only variables

Data Containers (objects)

There are three objects, as data containers. Each one may contain data or and other containers. We handle these by pointer, a variable name which hold a hidden pointer:

1. Auto Array (a tuple)
2. Inventory
3. Stack

Array Object

See about Arrays in the Using Arrays. Read this which have information about the array as object.

Arrays in M2000 are objects. There are two interfaces and one underline object, the mArray:

1. Name with Parenthesis like A() or A%() or A$(). (Array as a Value)
2. Name as pointer to Array like A. (Pointer of Array)

```
Dim A(4)=1

A=(1,2,3,4)

Print A  ' 1 2 3 4

Print A() ' 1 1 1 1

Print Len(A())=4

Print type$(A)="mArray", Len(A)=4

Push A(), A  ' so now top of stack has a copy of A pointer

Stack   ' show *[mArray], *[mArray]   we see two pointers.

Read A(), A

\\ get a copy of pointer and drop pointer

Print A   ' 1 1 1 1
```

```
\\ get a copy of array items in A() from pointer, and drop pointer

Print A()  ' 1 2 3 4
```

Passing by value an array A() is passing a pointer, and then if we read to A we get a copy of pointer, else if we read to B() we get a copy of items to B() at the time we read the B().

Lets see the following example. We make a new name A() as an empty array. We make a copy of auto array (or tuple) as (1,2,3,4,5), here with numbers but we can put anything, including pointers to containers. We push A() to stack, and then we copy new items (6,7,8,9) changing the Len(A()) form 5 to 4. The pointer of A() not changed. So we read B() (we can use an new name in READ) and we get a copy of A() as 6 7 8 9.

```
Dim A()

A()=(1,2,3,4,5)

Push A()

A()=(6,7,8,9)

Read B()

Print B()   ' 6 7 8 9

B(0)++

Print A() ' 6 7 8 9 because B() has different pointer, it is a different array
```

Lets see the following example. Variable A is a pointer to the literal (1,2,3,4,5) (a copy of the literal in the memory).Variable B is a copy of A/ We push A to stack, as a copy of pointer A. Now we place a new pointer to A with a copy of literal (6, 7, 8, 9). So now We read B() as a new name and because A is a pointer B() get the pointer only, so no items copied. So B() has the same pointer as B. We prove that by adding one to each item of B using B++.

```
A=(1,2,3,4,5)

B=A

Push A

A=(6,7,8,9)

Read B()

Print B()  ' 1 2 3 4 5
```

```
B++

Print B  ' 2 3 4 5 6

Print B() ' 2 3 4 5 6
```

How this is possible for these two example to Read B(), the first time using new pointer and the second time using the same pointer from mArray on stack?

Stack say for each interface that is mArray but they are different. The pointer is another object (named mHandler) which is a carrier of mArray. So interpreter knows what to do. All container pointers which are of the second interface (a simple variable name, not a name with parenthesis) are of the same mHandler object, but all reports from Stack and List for variables always return the underline object. So a container pointer is actually a mHandler. An array A() is actually mArray object, so when we put it to stack we put the mArray not the handler. So when interpreter read this from stack know what is the target (A or A()) and what iis the reading object (mHandler which have a mArray, or the mArray).

Names mArray and mHandler are actual names of objects in the source code of M2000 interpreter.

Arrays of first interface allow shallow copy, so a A()=B() means that A() get a copy of B(). Because B may have pointers to objects, B get copy of that pointers. But Group objects that are items in arrays without pointer we get a copy of Group.

```
Group A {X=10, Y=30}   ' this group is a named

Dim A(10)=A  ' a named group copied to each position using different pointer.

Print A(0).X, A(0).Y

Push A()

A(0).X+=10

Read B()

Print B(0).X=20 ' as expected - we see that in previous example

B(0).X=10

Print B(0).X=10

Print A(0).X=20  ' because A() has ten different Groups from B()
```

All objects above in A() and B() have unique pointers. We can use groups with unique pointers as Values.

Lets see the same example with pointers to Groups. We put the Group A as a copy in lambda function GetPointer, as a closure, and now we didn't use = to return a value, but arrow to return a pointer of A. Pointers are two types, a pointer to a named group, or a pointer to volatile group. The first pointer is a weak reference so here we want the second one. We have to place A in parenthesis, so ->(A) return the volatile A (a copy of A as a volatile group). A volatile group erased when no pointer points to it. A named group erased when the creator exit from call. Here we have a lambda function, and the closure kept as volatile group, and when we call the lambda function then interpreter create A using the volatile group and at the exit copy back the named group as the volatile group. So now each time the GetPointer() called in an expression return a pointer to a new volatile group. So for all items in A() we get ten pointers to different groups.

```
Group A {X=10, Y=30}

\\ we use (A) to get a pointer to a volatile group

\\ else we get a pointer as a reference to something which loose the reference

\\ when GetPointer return from call

GetPointer=Lambda A -> {

        ->(A)   ' or     =Pointer((A))

}

\\ << execute a function for each item in A()

Dim A(10)<<GetPointer()

Print A(0).X, A(0).Y

Push A()

A(0).X+=10

Read B()

Print B(0).X=20 ' as expected - we see that in previous example

B(0).X=10

M->B(0)

Print B(0).X=10, M=>X=10

Print A(0).X=10  ' because A(0) is B(0)
```

```
Print A(0) is B(0) = True, A(0) is M = True

Print A(1) is B(1) = True

Print A(1) is A(2) = False
```

When we read B() we get a shallow copy, and this means that we get copy of pointers to groups, not copy of groups. We prove this by altering the B(0).X and see that the same done to A(0).X also the A(0) is B(0) which compare pointers say that is equal, returning true. We see that we can get pointer from B(0) to M which is the same as A(0). Also see that M=>X used to get value from X of M.

Arrays of second interface works with pointers, and if the last pointer erased, say A=B the A points to the same array which point B, and A was a unique pointer to another array, then this array erased.

We can define arrays with parenthesis using:

- DIM
- LOCAL
- GLOBAL
- READ (reading from stack to first interface from either interface on stack)
- Auto Array is a literal definition of an array (1,2,3,4), or jagged arrays ((1,2),(3,4)

We can make references for arrays (each identifier can take one reference when defined)

- READ when we pass a weak reference
- LINK for any mix of interfaces.

We can get a pointer from an array (second interface):

- Assignment A=A() or A=(1,2,3,4) or A=B when B is a pointer of an array
- READ to second interface from stack of values when top of stack is a by value pass of either interface
- Read by reference second interface from a weak reference at top in stack from either interface.

Items in Arrays can be a mixed type. Lets see what happen if we make A() and A$() as reference of A. Especially when we pass a K() array to A() and next to A. The first make a copy o K() to what A() points, so A and A() and A$() point to a copy o K(). The second make A to point K() and A() and A$() as reference of A they point to K(). Reference A() and A$() can't take new reference.

```
A=("A String", 100)

Link A,A to A$(), A()  ' so now A, A$() and A() are the same

Print A(1)=100 ' true

Print A$(0)="A String" ' true
```

```
Print A  ' print all using columns

Print A#val$(0)="A String" ' true

Print A#val(1)=100 ' true

Return A, 0:="New name", 1:=500

Print A$(0), A(1)

Dim K()

K()=("old name", 100)

A()=K()

K(1)=1000

Print A

Print K()   ' A is not K()

A=K()  ' Now A is K()

K(1)=200

Print A

Print A(1)=200
```

Arrays have special functions, plus an iterator object. The iterator object use the Each() to return an iterator from a container as a pointer to a mHandler which hold a hard reference to A(). So we can't return as value an iterator, we have to use in place we make it. To use the iterator we have a special While which find the specific object and handle it accordingly. We can get the counter from m using ^ as m^ (because m is an object, the ^ operator isn't the power operator).

```
counter=lambda x=0 -> {x++ : = x}

Dim A(10)<<counter()

Print A()   ' 1 2 3 4 5 6 7 8 9 10

m=each(A())

While m
```

```
        Print Array(m), Array(A(), m^)  ' print 1 to 10 two values in a row

    End While
```

Because an array may have from zero to ten dimensions, the iterator iterate the array as one dimension. Read the following example. We make A(2,2) which have type Empty as initial value (this treated as zero if we read A() or empty string if we read a reference of A(0 to a A$()). Next we place to row 0, two values using **:=** assign symbol for arrays, so we can place a string and a number. The same for row 1. We print A() like one dimension in columns using Print (which have a private iterator). So now we make m as iterator of A() and because we didn't place start and end value, the iteration has to be for each item, like the A() has one dimension. The good thing is that we can program inside the While the iterator object, setting the next start value as m^ (the current) plus 2. To print the first item in a row we have to use Array$(m), see the $ in this function which return string, and for the second item we have to use the second optional parameter of Array(), so we have to place the number of item as m^+1.

When an array used from a pointer as tuple (also from an iterator) the first item is always at 0 and the last at number of items minus one.

```
    Dim a(2,2)

    A(0,0):="Alfa",100

    A(1,0):="Beta", 300

    Print A()

    m=each(a())

    while m

        Print Array$(m), Array(m, m^+1)

        m=each(m, m^+2)  ' add 2 to iterator

    end while
```

Special functions for tuples, and arrays (which treated as tuples) are:
#EVAL(), #FILTER(), #FOLD(), #MAP(), #MAX(), #MIN(), #POS(), #REV(), #SLICE(), #SORT(), #SUM(), #VAL(), #EVAL$(), #FOLD$(), #MAX$(), #MIN$(), #VAL$()
These functions return a value or an array (a copy) as a pointer, and can be used one after the other.

```
    A=("Z","A","B")

    Print A#sort() ' A B Z
```

```
A=(3,2,1)
Print A#sort() ' 1 2 3
A=(3,2,"Z", 1, "A","B")
Print A#sort()  ' 1 2 3 A B Z
Print A#sort()#rev()  ' Z B A 3 2 1
```

Also there are three more functions: CAR(), CDR(), CONS().

```
A=(1,2,3)
B=CAR(A)
Print Len(B)=1
Print B  ' 1
C=CDR(A)
Print Len(C)=2
Print C  ' 2 3
A=CONS(C,B)
Print Len(A)=3
Print A ' 2 3 1
A=CONS(A,A)
Print Len(A)=6
Print A ' 2 3 1 2 3 1
```

We can change dimension of an array without erasing items (if the new total items are same or greater than the old, and we didn't provide a value or a function per item). Also we can copy items from one array to another and from same array with care for overlapping items.

```
A=(1,2,"this is a String",4,5)
Module Redim (&A(), x) {
      Dim A(x)
```

```
}
Redim &A, 100
Print len(A)=100
Link A to A()
Print type$(A(0))="Double", type$(A(5))="Empty"
For n=0 to 40 step 5
      Stock A(n) keep 5, A(n+5)
Next n
Stock A(0) keep 50, A(50)
Dim B(5)
For n=0 to 95 step 5
      Stock A(n) keep 5, B(0)
      Print B()
Next n
Stock B(0) out a1, a2, a3$, a4, a5  ' we can place values from an index
Print  a1, a2, a3$, a4, a5
Stock B(0) in a5, a4, StrRev$(a3$), a2, a1
\\ same as  B(0):= a5, a4, StrRev$(a3$), a2, a1
Print B()
```

Another way to add items to a tuple:

```
A=(1,2,3)
B=A
A=CONS(A, (4,5,6))
Print B
```

```
Print A   ' A has a new pointer

\\ Now we use Append to keep the pointer

C=B

Append C, (4,5,6)

Print C

Print B  ' C is B, they have the same pointe
```

Inventory Object

An inventory object is a container with a key or a key and a value (if not value passed to a key then the key returned as value). A key maybe a number or a string or a mix of these. Internal the inventory works with a hash function. Internal is an array for items which expands as we place items. So expect for expansion or reducing, to add an item or to delete an item is a O(1) time complexity. For searching also we have O(1) time complexity.

The simple type List, has unique keys. We get error if we append a key and the key already exist

```
Inventory Alfa=1,2,3,5,6,17

Print Alfa    ' 1 2 3 5 6 17

Append Alfa, 4

Print Alfa ' 1 2 3 5 6 17

Sort Alfa As number

Print Alfa ' 1 2 3 4 5 6 17

Return Alfa, 1:="One", 2:="Two"

Print Alfa ' One Two 3 4 5 6 17


If exist(Alfa, 2) Then Print "key 2 exist and has value:";Eval$(Alfa)

If exist(Alfa, 5) Then Print "key 5 exist and has value:";Eval(Alfa)

Print Alfa(5)=5, Alfa$(2)="Two"
```

```
Print Alfa("5")=5, Alfa$("2")="Two"
```

we can't use Alfa$(1)="..." we have to use **Return Alfa, 1:="…"** but we can append

```
Alfa$(1)+="..."

Print Alfa$(1)="One..."
```

Another way to make a **list** (inventory), using parenthesis for handling then ":=". Also the inventory identifier exist as read only variable which pop an inventory from the stack (or raise error).

```
Push (list:=1,2,3:="ok",4,5)

k=inventory

Print k  ' 1  2 ok 4 5
```

From an inventory we can **delete an item**, but the order isn't guarantee.

```
Inventory Beta=1,2,3,4,5

Delete Beta, 3

Print Beta  ' 1 2 5 4
```

Inventories also have iterator objects:

```
Inventory a=1:="one",2:="two",3:="three",4:="four",5:="five"

m=each(a)

While m

      \\ index from counter, value as string, key as string, index from item

      Print m^, eval$(m), eval$(m, m^), eval(m!)

End While
```

A special version of Inventory is the **Queue inventory**. The Queue take same keys and never break the order, nor when sort.

```
Inventory queue delta=1,2:="two-1",3,1,2:="two-2"

Print delta  ' 1 two-1 3 1 two-2
```

```
sort delta
Print delta  ' 1 1 two-1 two-2 3
Print Len(delta)
Drop delta, 1    '  variant of Drop with same effect: Drop delta to len(delta)-1
Print delta ' 1 1 two-1 two-2
Print exist(delta,2)=true
Print exist(delta,2,0)=2
i=1
While exist(delta, 2, i)
      \\ value, item no
      print eval$(delta), eval(delta!)
      i++
End While
Append delta, 3
\\ we can change the value at specific index accessing delta properties
\\ index is base 0, so first item is 0,
With delta,  "index", 2, "value","New Value"
With delta,  "index", 0, "value",100
Print delta  ' 100 1 New Value two-2 3
\\ By default we change the last one of the same keys
Return delta, 2:="last one"
Print delta  ' 100 1 New Value Last one 3
Print Len(delta)=5
Inventory delta   ' new pointer, if the old one is the last then old data destroyed
```

```
Print Len(delta)=0
```

Combining Inventories and arrays:

```
Inventory a=1:=(1,2,3),2:=(3,4,5)
Print a(1)(0), a(1)#sum()=6, a(2)#sum()=12
z=a(1)  ' we get a pointer form a(1)
return z, 2:=100
Print a(1)#sum()=103
\\ we can place the pointer of a to an item in B()
Dim B(10)
B(2)=a
Print B(2)(1)#sum()=103
Print B(2)(2)#val(2)=5
z=a(2)
return z, 2:=1000
Print B(2)(2)#val(2)=1000
Append z, (6,7,8)
Print Len(z)=6
Print B(2)(2)#sum()=1028
```

Making a queue in expression:

```
Print (queue:=1,2,3:="300",400)
q=queue:=1,2,3:="300",400
```

Inventory sorting by columns in keys:

```
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"
```

Change an item value by an operator

```
alfa=list:=1,2,3,4,5
\\ without valuse, keys are values also
m=each(alfa)
\\ we can't use =, we have to use Return alfa, m^:=expression
While m {
        alfa(m^!)*=100  ' change valies
}
Print alfa  '100 200 200 400 500
\\ 2 is the key, 1! is the order number (start from 0).
Print alfa(2)=200, alfa(1!)=200
Print alfa(2) div 12 = 16  ' true
alfa(2)|div 12
Print alfa(2)=16 ' true
```

Stack Object

So we see until now, data indexing by number for one up to ten dimensions, and list of data in inventories using keys. The third container is the stack object a collection of values, with easy moving. The stack object is identical to stack of values object. The difference is that the current stack never expose a pointer. We can get the current stack only if we place a new one. The identifier [] do that, expose the pointer of current stack and leave a new stack as current stack.

```
Flush ' empry current stack
Data 1,2, "three", 4
m=[]
Print Empty=true
Print len(m)=4
```

```
\\ stack is one based

Push 1, 2, 3

Print Len(m)=4, StackItem(m, 1)=1, Stackitem$(m, 3)="three", StackType$(m,
3)="String"

Stack m {

      \\ hidden current stack temporary and make m as current stack

      Stack

      Print envelope$()="NNSN"

      Print StackType$(3)="String"

      While not empty

            If isnum then Print Number else.if islet then Print Letter$ else drop

      End while

}

Print "Now old stack restored"

Stack   ' show stack

Print Stack.Size=3

Read A, B, C

Print A=3, B=2, C=1
```

We can make stack objects in expression, also we can make iterator for stack object:

```
alfa(stack:=1,2,"three",3)

Sub alfa(m)

      Print m

      local mm=each(m)

      while mm

            if stacktype$(mm)="String" then
```

```
                Print stackitem$(mm)
            else
                Print stackitem(mm)
            end if
        end while
    end sub
```

Binary Data - Buffer Object

The three previous containers can get each other as item. The Buffer container can be an item for the previous containers but can't contain objects.

A Buffer is an object with a memory buffer, and in some cases we can make pointers to strings (BSTR). The memory buffer can be expand or reduced. Any buffer has a number of items, from one and above.

Items for buffer can be simple items or structures of items. An array of numbers in M2000 need at least (without other properties) number of items times 16, because each item is type of Variant (so a Variant can hold any type including objects). A simple item in buffer has the exactly bytes we need to spend for that. So a byte has one byte, an integer has 2 bytes, a long has 4 bytes, a double has 8 bytes. So an array in buffer of 100 double numbers have 100*8 bytes and not 100*16 bytes.

Simple items

So lets make 100 bytes of a clear memory buffer. The clear clause erase then memory buffer. We say byte*100 so we have item as byte and 100 of them. A byte can hold values from 0 to 255 (unsigned value)

```
    Buffer clear alfa as byte*100

    Return alfa, 0:=65, 1:=66, 2:=67      ' we place three values from 0 offset

    Print Eval(alfa, 0)=65

    \\ eval$(alfa, 0,3) get 3 bytes from offset 0 and return them as string

    Print chr$(eval$(alfa, 0, 3))="ABC"  ' Chr$(string) get a string as ANSI and
    return Utf-16Le

    A$=eval$(alfa, 0,3)
```

```
Print Len(A$)=1.5  \\ len return as words (2 bytes) so 1.5*2= 3 bytes.
For i=1 to 3
      \\ a mid$() for ansi strings
      K$=mid$(A$, i, 1 as byte)
      Print Len(K$)=0.5,  Asc(chr$(K$))   ' True 65 True 66 True 67
Next i
\\ using Str$(string) convert a Utf-16LE to ANSI using current Locale number
Return alfa, 0:=Str$("IKL")
Print chr$(eval$(alfa, 0, 3))="IKL"
Print alfa(0)  ' return the address of first byte in alfa
\\ We can place signed bytes, integers, longs using uint()
Return alfa, 20:=uint(-5)
\\ but we read them as unsigned, or we can convert them using sint()
Print eval(alfa, 20)=251
\\ sint() get 1 for byte, 2 for integer and 4 for long
Print sint(eval(alfa, 20),1)=-5
\\ copy pointer to new variable
a=alfa
\\ casting, when we use casting at Eval the offset always has byte unit.
Return a, 30:=0xFFFFFFFF as long
Print Eval(a, 30 as long)=0xFFFFFFFF  ' true
Print Eval(a, 29 as Integer)=0xFF00   ' so we have little endian
Buffer beta as long*25   ' Item is Long 4 bytes
Return beta, 0:=Eval$(alfa,0, 100)  ' copy alfa to beta
```

```
Print Eval(beta, 0)=5000009

Print Eval(beta, 0)=0x4C4B49

Print Eval(beta, 1 as byte)=0x4B

Return beta, 0:=0x4C4B48
```

Using Structures

So how we can but the second byte on first long on beta (at offset 0). We know that reading is easy, because we place the offset as byte at casting. So for returning values we need to use structures with unions.

```
structure aLong {
        {
                a as Long
        }
        {
                a0 as integer
                a1 as integer
        }
        a00 as byte
        {
                a22 as integer
        }
        a01 as byte
        a10 as byte
        a11 as byte
}
Print Len(aLong)=4
```

```
Buffer clear Delta as aLong*25

Print len(Delta)=100

offset=1

Return delta, offset!a01:=0XFF, offset+1:=0xFFFFFFFF

Print Eval(delta, offset)=0XFF00

Return delta, offset!a1:=0XFFAA

Print Eval(delta, offset)=0XFFAAFF00, Eval(delta, offset!a1)=0xFFAA

Print Eval(delta, offset*4+1 as integer)=0XAAFF  ' this is casting

Print Eval(delta, offset!a22)=0XAAFF

Hex Eval(delta,offset), Eval(delta, offset!a1)
```

BSTR Strings in Buffers

A BSTR is a windows system string, which a pointer point to an offset in memory and offset-4 is the length of string and from offset we have the string data. String data can be ANSI or Unicode (Utf16-LE) or any data, including zero bytes. For some cases a BSTR string always finish to double zero (two zero bytes), and these bytes aren't report to length. A second property for BSTR is that we can't write to it. So if we change a BSTR string we get a new one in a new pointer. So a pointer points to a "constant" string. All strings in M2000 are BSTR type. The variant type also work with BSTR type of string, so a Variant has 16 bytes, but only 4 bytes needed to point to a BSTR. An empty string has null pointer in variant. So except for null string all other have the BSTR structure (length+data+"00") with offset at first byte of data.

The pointers for BSTR in Buffers are pointers to pointers of BSTR. When a buffer object destroyed, the real pointers of BSTR are in a inventory and from there all strings erased. So at the end all memory return to system.

```
structure alfa {
        {
                a as long*4
        }
        c as String*4   ' 4*4=16
        b as long*4 ' 4*4=16
```

```
}


Print len(alfa)=32, Type$(alfa("c"))="String"
Buffer Clear Check as alfa*10
Print Len(Check)=320
offest=0
Return Check, offest!c!0:="Hello There"
Return Check, offest!c!1:=""
Return Check, offest!c!2:=""
Print Eval$(Check, offest!c!0)
Try {
        Print Eval$(Check, offest!c!3)=""
}
Print Error$  ' Invalid pointer for BSTR at Buffer


Print Eval(Check, offest!a!1)  ' point to a bstr pointer
Print Eval(Check, offest!a!2)  ' point to another bstr pointer
```

Offsets with ! Notation may have up to three level, or we have to use functions to calculate the final offsets. A structure is a an read only Inventory. The values are the offsets, and the labels are the keys. These inventories uses a second value for each item to store the type of item. As types we can use other structures.

Binary Files

Expanding the previous example (add these lines of code), to see how we can use Put and Get for binary files. If we want to read and write a binary file, we open for append, we use Seek to position the file cursor, and the same time we open the file for input and now we get a second file cursor. For this example we use a file for one buffer only. So no need for use of Seek statements. The Put and Get statements read the length of buffer and then use it for the operations. So if we expect to get from file cursor 100 byes we get error if there are not so many bytes in file. The FileLen() function return the file length in bytes. If a

buffer has pointers to BSTR then we make a mistake. We have to use strings inside buffer, ANSI or UTF-16LE, or UTF-8 (look Help String$() to see the variant which make conversions to some string encodings)

```
Open "Buffer01.bin" for output as #k
      Put #k, delta
Close #k
Print Filelen("Buffer01.bin")=100
Buffer clear Delta as aLong*25
Hex Eval(delta,offset), Eval(delta, offset!a1)
Open "Buffer01.bin" for input as #k
      Get #k, delta
Close #k
Hex Eval(delta,offset), Eval(delta, offset!a1)
```

Handle of Png File

We can use Buffer(filename_string) to get a file as a buffer in memory. If a buffer has an image file then M2000 can handle it using GDI+. Functions like Image.X() and Image.Y() get width and height in twips of the image in buffer. In buffer we have a compressed image, and the PC quickly expand it and drawing. Also if PNG file has information a percentage of opacity on a color or by a mask for each pixel then GDI+ handle it pretty fast. Look Appendix I for the test.png.

```
Png4=Buffer("test.png")
x=Image.X(Png4) div 2
y=Image.Y(Png4) div 2
hold
Mouse.icon Hide
every 25 {
      release
```

```
            move  mouse.x-x, mouse.y-y
            image Png4
            refresh
            if keypress(1) then exit
      }
      Mouse.icon Show
```

There is another variant, where we use a Thread to change N

```
      Png4=Buffer("test.png")
      hold
      N=1
      M=1
      Thread {
            if keypress(2) then N=if(N=359->0, N+M) : M++ Else M=1
      } as K interval 10
      Mouse.icon Hide
      Main.Task 25 {
            release
            move  mouse.x, mouse.y
            sprite Png4,,N
            refresh
            if keypress(1) then exit
      }
      Mouse.icon Show
```

Machine code execution

The following example uses two Buffers, one for data and one for code. A buffer for code at execution tine is locked for write to it. The first buffer is BinaryData, the second named alfa. We use a PC as program counter and some subs to place code. The PC is the offset for alfa buffer. We use casting for placing 32bit numbers.

Subs run at same scope as module or function where they called. See about subroutines..

The result can be retrieved from BinnaryData buffer or from accumulator (a non zero return an error) depending of the one before last opcode OpByte(0x31, 0xC0). Last opcode is Ret()

Machine code used in M2000 for experiments. It's very easy to hang the environment with a mistake in the 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     ' 5100 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
```

Old style programming with M2000

The old style programming is like programming with BASIC language, except the use of block of statements in curly brackets which is optional. Parts of old style are the flow control structures.

In M2000 we use the IF structure, the For Loop, the Do or Repeat loop, the While loop, plus a simple block loop. Also we may use Labels named or numeric. We can use GOTO to label, and GOSUB to label using the RETURN like in BASIC. Also we can use subroutines with local variables.

The Print Statement

We use in the following examples the Print statement. It is like the BASIC statement Print. We print numbers and strings in columns. If a number need more columns or string need more columns the print use more columns. Numbers are justify right, and string justify left to column. If we leave two comma without expression we get a new line. We get new line after the last expression. We can stop the text cursor (not visible), at column using coma.

```
Print 1,,2,,3

      1

      2

      3
```

We can use semicolon to break columns

```
Print 1;2;3

123

Print 1,2,3

            1           2           3
```

We will see more about Print ilater

Numeric Labels

We can use numbers before statement in a line. We can use colon to split the line in statements. We can use labels with a colon as the last character, alone in a line (or with a remark). A numeric label may have leading zero, from 1 to 5 digits.The next code is like BASIC code:

```
10 FOR I=1 TO 10
```

```
20 PRINT I

30 NEXT I
```

We can place the numeric labels in any order. But must be the first item in the line.

```
60 FOR I=1 TO 10

50 PRINT I

40 NEXT I
```

There is a different approach for FOR NEXT from basic (a special switch, a software switch change the default behavior to that of the BASIC). The For loop always execute once. The direction of counting always describing from the two values. So a FOR i=100 to 1 means counting from 100 to 1. A FOR i=1 to 1 means counting from 1 to 1 (one time). If we use a minus in STEP clause then if we have same start and end point the variable after the For loop get a new value less than the absolute value of STEP. Using the special switch a FOR loop may not happen because the step sign describes the up or down counting. The next example print only one time the value of 1. The Next clause has optional the variable name.

```
for i=1 to 10 step 0
      print i
next
```

Also we can use this **for i=1 to 10 step 0 {print i}**

We can use GOTO

```
10 I=1

20 PRINT I

30 I++

40 IF i<10 THEN 20



090 X=RANDOM(1, 3)

100 ON X GOTO 130, 140,150

110 PRINT "OK"
```

```
120 GOTO 390

130 PRINT "ONE": GOTO 110

140 PRINT "TWO": GOTO 110

150 PRINT "THREE": GOTO 110


390 X=RANDOM(1, 3)

400 ON X GOSUB 430, 440,450

410 PRINT "OK"

420 END

430 PRINT "ONE": RETURN

440 PRINT "TWO": RETURN

450 PRINT "THREE": RETURN
```

We can't jump out of a module. A jump from a block of code means that the block terminate at the point of jump.

We see that the GOSUB to a label using a RETURN statement used like the that on the BASIC language. There is one type of subroutines more, we say SUB /END SUB routines. These routines may have local variables and a parameter list. A subroutine like this have the same scope as the module, so the local variables just shadow the same name variables in the module.

The subroutines added to the end of module's or function's code. A subroutine can call itself, and the number of recursion is separate from modules and functions, by using a special stack per module or function. A module or function use a special object called a **basetask** underline (this object belong to M2000 interpreter). A subroutine has no basetask and rely on the module or function where called.

The If statement

Like in Basic there is the IF THEN ELSE statement. In M2000 there are three variants of IF statement construction:

1. One line IF construction without Block code
2. IF using blocks (in one or multi lines)
3. Multiline IF using END IF as the last statement.

The one line IF handle all statements after IF.

```
If X=20 Then Print "ok": X++
```

When X has a value of 20 we get the ok on the console (or current layer), and then add one to X. But look the next code. We get the "don't care" because the block terminate the IF structure (except we have an ELSE or ELSE.IF clause), So the second example is of 2nd type

```
X=2 : If X=20 Then {Print "ok": X++} : Print "don't care"
```

We can write the above as

```
X=2
If X=20 Then {
      Print "ok"
      X++
}
Print "don't care"
```

Or using the third type:

```
X=2
If X=20 Then
      Print "ok"
      X++
End If
Print "don't care"
```

The third type use identifiers IF condition THEN, IF condition ELSE, ELSE.IF condition THEN, ELSE.IF condition ELSE, ELSE, END IF each without other statements in the line except am optional remark sign ' or \. The logic is simple. If we have a statement after Then or Else or we have an open bracket then this isn't third type. If interpreter find that is the if of third type then expect to find END IF. If not find END IF then an error raised.

```
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 in IF statement

End if
```

So we may have the second type (which is good for jumps from blocks), without using an End IF:

```
If condition Then {

} Else.If  condition1 Then {

} Else.If  condition1 Else { ' reverse logic

} Else.If  condition1 Then {

} Else {

}
```

We can use IF THEN ELSE in a line without END IF and with or without blocks

```
X=Random(19,21)

Print X

If X<20 Then Print "Lower than 20" Else.if X=20 Then Print "Is 20" : Print "ok" Else Print "Above 20"

Print "Done"

X=Random(19,21)

Print X

If X<20 Then {Print "Lower than 20"} Else.if X=20 Then {Print "Is 20" : Print "ok" } Else {Print "Above 20"}

Print "Done"
```

Or this using multi lines (see the first using END IF)

```
X=Random(19,21)
Print X
If X<20 Then
      Print "Lower than 20"
Else.if X=20 Then
      Print "Is 20" : Print "ok"
Else
      Print "Above 20"
End If
Print "Done"
X=Random(19,21)
Print X
If X<20 Then {
      Print "Lower than 20"
} Else.if X=20 Then {
      Print "Is 20" : Print "ok"
} Else {
      Print "Above 20"
}
Print "Done"
```

After the THEN or the ELSE clause a number means a GOTO number. If we want a jump to a non numeric label (a text label) we have to place the GOTO statement. The later is true always if we use the third type of If.

Example of an Old Sort Algorithm

We have a string with words without termination character. We have offsets for each word (base of 1, so the first word has offset 1). We want to sort the words sorting only the offsets. This is an example of **spaghetti code**, using GOTO in main algorithm.

```
Module OldSort {
        function mxstrcmp(&Cmp$,  start1,end1, start2,end2) {
                end1+=1-start1
                end2+=1-start2

                hlp1$=mid$(Cmp$,start1,end1)
                hlp2$=mid$(Cmp$,start2,end2)
                =compare(hlp1$,hlp2$)
        }
        function Getone$(&buffer$, start1, end1) {
                end1+=1-start1
                =mid$(buffer$,start1,end1)
        }
        strSearch$ = "3ne2wo1hree6ive5ix4orth"
        \*strSearch$ = "OneTwoThreeFiveSixForth"
        Flush ' empty stack
        Data 1,4,7,12,16,19,0  ' these are offsets (base 1) in string strSearch$
        i=0   ' we place one more so we use arrays from index 1
        read startKeyWord
        do ' same as Repeat
                dim smarkerskt(i+1), emarkerskt(i+1)
                read nextStart
```

```
                smarkerskt(i)=startKeyWord
                If nextStart = 0 Then
                        emarkerskt(i)=len(strSearch$)
                        exit    ' no Break here (Break stopped only in a Try { } and in
module block )
                Else
                        emarkerskt(i)=NextStart-1
                        startKeyWord=NextStart
                end If
                i++
        always
        nKeywordskt=i+1
        dim sortptrs(nKeywordskt)
        module fillme {
                read &ar()
                k=dimension(ar(),1) : i=0 : do  ar(i)=i : i++ : k-- until k=0
        }
        fillme &sortptrs()
        for i=1 to nKeywordskt
                Print getone$( &strSearch$, smarkerskt(sortptrs(i-1)),
emarkerskt(sortptrs(i-1)))
        Next
        Print  "items:";nKeywordskt

\\ here the old code start
```

```
        s1 = nKeywordskt
        s2 = s1
        s3 = 0 :  s4 = 0 :  s5 = 0 :  s6 = 0
        a = 0

GP_SP_S1:
        s1 = int(s1 / 2)
        If s1 = 0 Then Goto GP_SP_S5
        s3 = s2 - s1
        s4 = 1

GP_SP_S2:
        s5 = s4

GP_SP_S3:
        s6 = s5 + s1
        a = mxstrcmp(&strSearch$, smarkerskt(sortptrs(s5-1)),
emarkerskt(sortptrs(s5-1)), smarkerskt(sortptrs(s6-1)), emarkerskt(sortptrs(s6-
1)))
        If a = 0 Then
                If sortptrs(s5-1) < sortptrs(s6-1) Then
                        a = -1
                Else
                        a = 1
                End If
```

```
            End If
            If a <= 0 Then  Goto GP_SP_S4
            Swap sortptrs(s5-1), sortptrs(s6-1)
            s5 -= s1
            If s5 >= 1 Then Goto GP_SP_S3

GP_SP_S4:
            s4++
            If s4 > s3 Then Goto GP_SP_S1
            Goto GP_SP_S2

GP_SP_S5:
            s1 = 0
            \* here the old code end.

            Print "-----------------------------------------------"
            for i=1 to nKeywordskt
                  Print getone$( &strSearch$, smarkerskt(sortptrs(i-1)),
emarkerskt(sortptrs(i-1)))
            Next i
            Print "done"
}
OldSort
```

The Select Case Structure

We can use Select case for numeric or string values. The select case has some limitations:

Between each case we can place a line of statements (using semi colon) or a block of code using { }.with any number of lines inside block. We can't place a case after a case without a line or a block of code.

The E**lse Case** or E**lse** can be used for the case which not included in the series of cases. Using the select case structure between a IF End If structure we have to use the Else Case, because the Else is the same as the else in that variant of If.

A case get a list of any number of Case expressions.

For numbers the case expression can be a number, a numeric expression including variables, a half comparison like >1000, and a range 10 to 300 where both limits included to the range. If we want the reverse range we place this <10, >399.

The same for strings, we can include a string value, an expression with variables, half comparisons and ranges.

Not all expressions evaluated, but only up to the one which return true.

When a case get a true expression, executed this only. If we place a block and inside the block we place the Break statement then all the cases until the Else Case (not included) executed without evaluation the cases for each one. We can stop this way of execution using a continue (inside a block of code).

```
For i=1 to 10
    x=random(1,10)*10**random(0,3)
    Print X
    Select Case X
    case 1, 3, 5
        Print "1 or 3 or 5" : Print "in same line statement"
    case 10 to 50
    {
        Print "10 to 50 use a block of commans"
        Print "ok..."
```

```
        }
        case <-1000, >1000
            Print "far away"
        else case
            Print "Dropped"
        end select
    Next i
```

The For Loop

The For loop has too forms, one with block and one without using the Next statement. Also There is another For statement using Objects, which isn't a loop structure, but here we see only the loop structure. As in BASIC there are two or three values, where the third is the Step, which control the loop, as a control value. We can exit from loop using EXIT or we can skip to the end of block and advance to next control value using CONTINUE. Also we can jump to some other point and not exactly the next statement after the end of loop. The variable which has the control value, can be change for current iteration, but in the next iteration get the next control value as normal, by copy from an internal true control value. So we can't handle the loop by altering the control value. We can use any numeric type for the loop.

There is a major difference from the For in BASIC. By default the step value used as absolute value, the direction of counting (up or down) always are from first value to second. If the first and second values are equal then we get one run of block and then if a sign exist in step the we get the algebraic result of Start_value+Step. If step is zero we get one run of block with start value. There is a Switch which change behavior to act as those in BASIC

To change mode of FOR: From console use SWITCH "+FOR" or from code in a block use SET SWITCH "+FOR" to engage the BASIC variation for FOR, use "-FOR" to return to default. Also we can start M2000.exe with swicth +FOR as command line argument). Use Monitor from console to see all the states of switches. Use Help Switches to find information for all switches.

Look at the following example, we can exit from a block which is part of a For structure:

```
    FOR i=1 TO 10 {
            IF i=5 THEN 500
```

```
        PRINT I
    }
    PRINT "NEVER PASS FROM HERE"
    500 PRINT "OK"
```

We can use the EXIT FOR with a number, In a FOR/NEXT structure (without a block):

```
    FOR i=1 TO 10
        IF i=5 THEN EXIT FOR 500
        PRINT I
    NEXT
    PRINT "NEVER PASS FROM HERE"
    500 PRINT "OK
```

We can use Continue in a For loop to continue the loop immediate (or exit if can't continue).

Ordinary loop structures Do Until (or Repeat Until), Do Always (or Repeat Always) and While {} and While/End While: All of these structures have a block of code (some have a hidden one), so Goto works fine

The Do Until loop

```
    X=random(1,3)
    Print X
    Do
        If X=2 then 300
        X++
    Until X>10
    Print "ok"
    300 Print "Last line"
```

```
X=random(1,3)
Print X
Do {
      If X=2 then 400
      X++
} Until X>10
Print "ok"
400 Print "Last line"
```

The While loop

```
X=random(1,3)
Print X
While X<=10
      If X=2 then 300
      X++
End While
Print "ok"
300 Print "Last line"

X=random(1,3)
Print X
While X<=10 {
      If X=2 then 400
```

```
        X++
}
Print "ok"
400 Print "Last line"
```

Jump from nested blocks

Check the following code (we use numeric and text labels), We can make a jump inside a block or we can make a jump in any other block (backward or forward)

```
x= random(1,3)
{
        {
                On x Goto 200, alfa, 400
                Print "---"
                200 Print "ok1"
        }
        alfa:  \\ only remark after a text label
        Print "ok2"
}
400 Print "ok3"
410 Print x
```

A block as loop

Any block can be used as a loop block. Each block has a flag, the loop flag. When a statement Loop executed the loop flag raised (as a Boolean value get True). When the block exit normal, check the loop flag and if it is true then restart the block. Each block start always with loop flag to false.

We have the Restart statement which restart the block (and as a new restart we have the loop flag reset to false).

Also we have two different exit statements. The Exit and the Continue. In any loop flag condition the Exit make an exit of block. The Continue statement in a block depends on loop flag, so when this flag is true we get a Restart, and when this flag is false we get an Exit.

Also there is a Break statement, which break all blocks until Module or a special block like those which are part to a statement like For or Do and other.

```
acc=0
\\ like do loop (See Else after IF)
\\ 1 .. 10
{
      acc++
      Print acc
      If acc>10 Else Loop
}

\\ like while (See Else after IF)
\\ 10 .. 1
{
      if acc>0 Else exit
      Print acc
      acc--
      Loop   \\ we can use Restart because is the last one
}
\\ print 0 .. 10
```

```
{
      if acc<10 then loop
      Print acc
      acc++
}
\\ print 10 .. 0
{
      if acc>1 then loop
      acc--
      Print acc
}
```

Simple Routines

A simple routine used for the use of same code for many times, from different parts of a program. Because the use of modules, we can call a routine if the routine exist in the module where we call it. The executed lines are like to belong to the calling position of the code. Named labels are case sensitive for interpreter.

```
X=0
While X<10
      Gosub 100
      Gosub 110  ' like a second entry
End While
Print X
500 End  ' need to stop the passing without normal call
100 Print "This is a routine"
110 If X>5 Then X=100 : Return
```

```
120 X++
130 Return
```

Or using named labels (each label in separate line):

```
X=0
While X<10
      Gosub first
      Gosub second  ' like a second entry
End While
Print X
End  ' need to stop the passing without normal call
first:  ' only remarks after this type of label
      Print "This is a routine"
second:
      If X>5 Then X=100: Return  ' we can exit from routine from anywhere
      X++
Return
```

Subroutines

A subroutine open a session for temporary definitions. All new definitions erased at the return from the subroutine. Names for subs aren't case sensitive (for greek names there isn't an automatic removal of any acute over letters, like in other identifiers).

Why subroutines and not modules ?

A module run on an execution object. A subroutine is lighter because use the same execution object as the caller's one. A module use execution stack to return from call, but subroutine use a special stack on the execution object. The stack for subroutines get chuncks from memory. The execution stack has standard size. So for recursion a subroutine is better. By design a subroutine is iin same scope in the module where we call it. Here we didin't insert END before e SUB definition, because a SUB identifier act as

END if founded at execution. All subroutines added to the end of code in a module or function and always are local to that module or function.

We can place a subroutine in module or in the code where the module defined. When a subroutine called then the code executed like it is in the module we call it. Just see the following code. In alfa() subroutine we use M. If M not exist we get error. Inside Inner module we read M as Inner M, from Outer we read M as Outer M. The X variable inside subroutine alfa() is Local. So this variable shadow the X in Outer when we call it from Outer. When we call alfa() from Inner, there is no X for module Inner.

```
Module Outer {
      X=5000
      M=2000
      Module Inner {
            M=1000
            alfa(100)
      }
      Inner
      alfa(300)
      Print X=5000
      Sub alfa(X)
            Print M, ". Value="; X
      End Sub
}
Outer
```

We can share subroutines from Outer module to Inner and other modules local to Outer.

Simple Functions

Like subroutines we can make simple functions. Unlike subroutines the calling process include the evaluator which get resources from stack memory quickly, so we are better a

ten percent from normal functions. We call them only in expressions using @ before the name. See some examples in User Functions>Use of simple functions

A simple function's name has to be different from a subroutine name is same context. Also the name of a simple function has to be different from a name of a standard function from vocabularies (Greek and English) of M2000 identifiers. We can check this easy in M2000 editor because a known function name has different color from user names. To replace temporary a M2000 function use standard function (Function name() {} or Def name()=…)

We can't use Call to call a simple function like a module. We can't pass a simple function by reference. Also a simple function use the stack of value of the caller. We have the same for subs and when we use Call for functions and Modules, but simple functions do that in expressions. So in a @a(1)+@b(2) the first one may leave values in stack of values, and b() take that values and replace them. So simple functions may be a mess if we don't get the things in a right manner. We have to use Local for variables and arrays.

The simple function has to exist in original source, but because at the call we pass the same name space as the current one, we read variables from that one, so M in the following example is from the specific name space (Modules and normal functions make name spaces), If we call @alfa() in Inner module we have M to be defined in the module, before the use of function.

```
Module Outer {
      X=5000
      M=2000
      Module Inner {
            M=1000
            Print "Value:";@alfa(100)=2000
      }
      Inner
      Print "Value:";@alfa(300)=5000
      Print X=5000
      Function alfa(X)
            =X*10+M
      End Sub
```

```
}

Outer
```

Using Arrays

A data structure, in memory, from the first step of programming, is the Array. To make an array we use the DIM identifier. We can copy a value to each item.

The DIM statement

The following example make the three possible types of arrays. The A%() works like the A % variable with rounding to 0.5. The base of an array by default is 0, so for 10 items we get indexes from 0 to 9. We can use up to ten dimensions. Here each array has one dimension. We can **resize** (and redimension) any array, preserving items if we didn't provide a value for all items. By default a Dim A(10) for the first time make each value as Empty type, which is a zero for calculations. So here we make Dim A(10)=0, all values as type Double. After the Dim A(20) we get ten more items, but they are Empty. So the third For Next loop put a 0 to each item from 10 to 19. We prove now that all items are Double from the last For Next loop

```
Dim A(10)=0, A%(10)=1, A$(10)="ok"

For i=0 to 9

      Print A(i), A%(i), A$(i)

Next i



Dim A(20)

For i=0 to 19 : Print type$(A(i)) : Next

For i=10 to 19 : A(i)=0 : Next I

For i=0 to 19 : Print type$(A(i)) : Next
```

We can make arrays with **specific range for lower and upper limit** for each dimension. We can use negative numbers for indexes too.

```
Dim A(-5 to 5, -5 to 5)=0

Print Len(A())
```

```
List
```

If the module is A (where we put these three lines) we get:

```
      121
A.A(11,11)
```

So we have 11X11 items. Function Len() works for strings and any object which have a length of items.

For **Base 1 arrays** we can use the Base 1 clause. Also we can use Print to print an array each value in a column, handling the new line automatic.

```
Dim Base 1, B(10)=2, C(30)=1

For i=1 to 10: Print B(i), :Next I: Print

Print B()   ' the same as the above line

Print C()
```

We can make an **empty Arra**y. We can **copy an array** to another array, we can change value of an item using ++, –, +=, -=, /=, *=, -! (change sign) and ~ (boolean invert)

```
Dim A(), B(10)=1

Print Len(A())=0

A()=B()

Print Len(A())=10

B(0)+=100

Print B(0)=101, A(0)=1
```

By Value Passing Array

Passing an array by value as array:

```
Module PassByValue (A()) {

      i=Dimension(A())

      if i=0 then Print "Zero Dimension Array": Exit

      P=1

        ' center to column using $(2) - try $(6) for center proportional text
```

```
        Print $(2),"Dim#", "Lower", "Upper", "total"
        Print $(0),      ' uaing , ro hold cursor to this line
        For j=1 to l
                P*=Dimension(A(),j)
                Print j, Dimension(A(),j,0), Dimension(A(),j,1), Dimension(A(),j)
        Next
        Print "Len(A()):"; Len(A()), "P:";P
}
Dim B(2 to 10, 4 to 20)=1
PassByValue B()
Dim B(0), C(), K(10)
PassByValue B()
PassByValue C()
PassByValue K()
```

By Reference Pass (Arrays, Array Items)

The following example display the pass by reference of an array, pass by value for array item, pass by reference array item. Passing by reference array item is a copy in copy out method. So if at the copy out the array change dimension the copy out process dropped without error.

```
Module PassArrayExample (&A(), &X, Y) {
        A(0)++
        X++
        Y++
        Print Y
}
Dim Base 0, k(10)=100, m(30)=10
```

```
Print k(0)=100, m(3)=10, m(4)=10

PassArrayExample &k(), &m(3), m(4)

Print k(0)=101, m(3)=11, m(4)=10
```

For Subroutines

Here we can't use copy in copy out process for m(3), but we can use another type, the passing of weak reference and use it in a string variable. Here we use the weak$ (we can use any name). The weak references can be used with a dot. So weak$.++ is like the m(3)++ but the name is the full name (including a specific part before m() which produced automatic at run time). To get the value we have to use Eval() so Eval(weak$) return the value. For string values inside a weak reference we use the Eval$(), so in one line:
A$="ok" : m$=weak$(A$): Print Eval$(m$.)="ok" ' true - look the dot in Eval$()

```
Dim Base 0, k(10)=100, m(30)=10

Print k(0)=100, m(3)=10, m(4)=10

PassArrayExample(&k(), weak$(m(3)), m(4))

Print k(0)=101, m(3)=11, m(4)=10

Sub PassArrayExample(&A(), weak$, Y)

      A(0)++

      weak$.++

      Y++

      Print Y

End Sub
```

About references. A variable or an array (not an array item) have a specific address, in a list of names for variables and arrays. So if name A has an address 100 and we make B as a reference of A , then the B has the same address 100. A reference can't change reference. We will see about Weak$() in Advanced Programming later.

Sorting 2D Arrays by multiple columns

This program is a mix of old and new style. We use a seed for the random generator, so we get each time the same random numbers.

We have two lambda functions. The Rand$ has a closure labels (a pointer to array, or auto array, or tuple). This return a field of 10 spaces with a name of a capital city. Because we choose to print Cities with proportional letters, and center justification the more spaces trimming from printing routine.

The second lambda has lambda inside and because we provide the (max) parameter (see the end of lambda), we execute it at once. It is a way to make an array which we provide the size of rows. We need the final lambda to retrieve a random number from 1 to max, and remove it from the array. We didn't fill the array to all numbers. Instead we think that each array item has a 0, which indicate "take index as number". When we remove an array item, when this has index less than max, then we copy the number from max, and reduce the max by one and redim the array to exclude one item. During the copy of number from top index to fill the removed number, we check if we have place something. If it is not the case, we have to push the index as number. From old style we use Gosub to numeric labels. These subroutines are just rows of statements with no local. Also we use a Sub End Sub where we want to pass arguments.

We will see a flashing (4 hz) "press any key…" using an Every { } stricture.

```
Form 80,40

Const max=36

\\ setup random generator to produse the same sequence

x=random(!12493587)

Rand$=Lambda$ labels=("LONDON","PARIS","BERLIN","ATHENS",
"ROME","SOFIA") ->{

        = Field$(labels#val$(random(0,5)), 10)

}

PickNumber=Lambda (m)-> {

    dim A(1 to m)

    =lambda A(), m -> {

        if m<1 then exit

        i=random(1, m)

        =if(A(i)=0->i, A(i))
```

```
            A(i)=if(i<m->if(A(m)=0->m,A(m)),0)
            m--
            if m=0 then A()=(,) else Dim A(1 to m)
        }
    }(max)

    Flush
    Dim A(1 to max, 1 to 3)
    Link A() to A$()
    Title("1. Populate Array")
    R=0
    refresh 5000
    for i=1 to max
        A(i,1):= PickNumber(), Rand$(), Random(1,6)*100
        Gosub 100
    next
    Refresh 20
    Gosub 240
    Title("2. Sort Ascending First Column")
    sort A(),1, max, 1,0
    R=20
    Gosub 200
    Title("3. Sort Ascending Second And Third Column, first Ascending")
    sort A(),1, max, 2,0,3,0,1,0
```

```
R=40
Gosub 200
Title("4. Sort Columns Third Descending, Second Ascending, first Ascending")
sort A(),1, max, 3,1,2,0,1,0
R=60
Gosub 200
End
100 Print Part  $(0), @(R),~(11),str$(A(i,1),"0000 "),@(R+5), ~(0, 7,7), $(6),A$
(i,2),~(15), @(R+15),$(0), ~(14), format$("{0::4}",A(i,3)) : Print
110 Return
200 Refresh 5000
210 For i=1 to max
220       Gosub 100
230 Next : Refresh 50
240 T=True
250 Every 1000/8 {
260       T~
270       Print Over $(6),~(If(T->7,15)),"PRESS LEFT MOUSE BUTTON"
280       If keypress(1) then Exit
290 }
300 Print Over
310 Return
320 Sub Title(A$)
330       Cursor 0,0: Double
340       Print Over ~(15),A$
```

```
350      Print
360      Normal
370 End Sub
```

Text Files

A way to store data to disk is through Text files, of Serial or Random access. In M2000 we can use Binary also (which we can see in Advanced Programming later). There is a special object the Document which save/load entire text file and a Log system to write to temporary folder a text file, which will see later.

Serial File Access

A text file of UTF16LE characters, or Unicode, means a WIDE file, using of 16bit minimum for a character. A text file of ANSI (using current Locale value) using of 8bit characters.

A file can be OPEN for four operations:

1. OUPUT: this operation erase the file if founded, we ge error if we can't write to media. Also we get error if the file is opened exclusive from elsewhere
2. INPUT: we get error if file not exist
3. APPEND: we get error if file not exist. We add to the end of file.
4. RANDOM: For a specific length we can read and write this length of information (the record) to file. A record can be a concatenation of fields of specific length too.

The statement give a number when we open the file.

```
OPEN file_name$ FOR INPUT EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR OUTPUT EXCLUSIVE  AS #variable_as_file_handle
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
```

We have to close using CLOSE #variable_as_file_handle

A file may OPEN two or more times we use INPUT and APPEND or RANDOM, without EXCLUSIVE clause.

If we want WIDE (UTF16LE) then

```
OPEN file_name$ FOR WIDE INPUT EXCLUSIVE AS #variable_as_file_handle
OPEN file_name$ FOR WIDE OUTPUT EXCLUSIVE AS #variable_as_file_handle
```

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

LEN for wide max is 32766/2, and for ANSI version 32766.

This is an example of standard (ANSI) file, for oupur lines of text and reading lines of text. We need EOF() function for finding End Of File to finish reading

```
Open "TextFileAnsi.txt" for output as #k

        Print "Line one"

        Print "Line two"

Close #k

Open "TextFileAnsi.txt" for Append as #k

        Print "Line Three"

        Print "Line four"

Close #k

Open "TextFileAnsi.txt" for input as #k

        While not EOF(#k)

                Line Input #k, aLine$

                Print aLine$

        End While

Close #k
```

CSV Files

The comma separated values (CSV) type of file, is a text file. The idea is to place lines of text, and for each text a comma to separate values. The original idea use numbers with dot for decimal numbers and quote marks (character 34 in Ansi and Unicode) for placing strings. To write the values M2000 use the Write statement. To read the values M2000 use the Input statement. Both statements have the variant with #fileHandler as first parameter

We can use encoding ANSI or UTF16LE. We can define the characters for value separator and for decimal point. Also we can exclude the quoting marks from string. And we can

convert the strings to JSON encoding strings, to include tabs, newline and other in a format using only plain ANSI characters.

Lets see how we program these properties of Write and Input statements

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.

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.

For ANSI files we have to use LOCALE to set the locale for conversion. A Local 1032 used for greek conversion.

```
Pen 11{Print "Ansi first"}

Print "Export csv using tab as seperator, and dot for decimal char"

open "testnochr34.csv" for output as k

Write With chr$(9), ".", ,1

Write #k, "Hello There", "And There", 1212.12, "Bye"

Close #k

Document docA$

Load.Doc docA$, "testnochr34.csv"   ' automatic get the file type

Print "The file as a string - tabs are expanded"

Pen 15 {report DocA$}

Print "ok..."

Print "Now we use Input # programming the separator and dot"
```

```
open "testnochr34.csv" for input as k
Input With chr$(9), ".", ,1
Input #k, A$, B$, C, D$
Close #k
Print "Results"
Print A$, B$, C, D$
Pen 11{Print "For UTF16LE wide chars"}
open "testnochr34.csv" for wide output as k
Write With chr$(9), ".", ,1   ' leave an optional value for JSON encoding
Write #k, "Hello There", "surrogate 𐤀", 1212.12, "1234 ברוכים הבאים abc"
Close #k
Document docA$
Load.Doc docA$, "testnochr34.csv"
Print "The file as a string - tabs are expanded"
Pen 15 {report DocA$}
open "testnochr34.csv" for wide input as k
Input With chr$(9), ".", ,1
Input #k, A$, B$, C, D$
Close #k
Print "Results"
Print A$, B$, C, D$
```

Random Access Text Files

A random access file, can be used to save text data as rows of records. We can put and get records with one opening. We can use the Wide version, or the Ansi, and we can use the Exclusive directive so only the running program can handle the file until closing it.

```
Const fieldlen=40, part1=30, part2=10
Def eRecord$
\\ for this example ensure that the test file not exist
Do
      DeleteFile("TextFileAnsi.dat")
Until not exist("TextFileAnsi.dat")
\\ create a data file of records
\\ each record has one big field
\\ Open "TextFileAnsi.dat" for wide random exclusive as #k len=fieldlen
Open "TextFileAnsi.dat" for random as #k len=fieldlen
      Put #k,field$("Kappa George",part1)+field$("1229998800", part2)
      Put #k,field$("Lambda Mary",part1)+field$("1225997861", part2)
      Put #k,field$("Delta Harris",part1)+field$("1225793462", part2)
      Print Records(#k)=3  ' # can be used from revision 17, ver 9.9.
      ListData()
      ChangePhone("1225793999", 3)
      ListData()
Close #k
Print Filelen("TextFileAnsi.dat")   ' 240 or 120 depends of WIDE clause exist
\\ no need for End statement. The Sub statement is like End.
Sub DeleteFile(a$)
      Dos "Del "+quote$(dir$+a$);   ' send command to a hidden dos box
```

```
        Wait 100
    End Sub
    Sub ChangePhone(NewPhone$, Where)
        Get #k, eRecord$, Where
        Insert part1+1, part2  eRecord$=field$(NewPhone$, part2)
        Put #k, eRecord$, Where
    End Sub
    Sub ListData()
        if records(#k)=0 then Print "Empty List" : exit sub
        Print "Rec No, Data"
        For i=1 to Records(#k)
            Get #k, eRecord$, i
            Print i, " "+eRecord$
        Next i
    End Sub
```

2d Drawing

See help file for graphic statements. Also see Appendix II for name of statemens
This is an example for Pythagoras tree. An example of recursion of a subroutine.

```
    MODULE Pythagoras_tree {
        CLS 5, 0  ' MAGENTA, NO SPLIT SCREEN
        PEN 14   ' YELLOW
        \\ code from zkl/Free Basic
        LET w = scale.x, h = w * 11 div 16
        LET w2 = w div 2, diff = w div 12
```

```
        LET TreeOrder = 6
        pythagoras_tree(w2 - diff, h -10, w2 + diff, h -10, 0)

        SUB pythagoras_tree(x1, y1, x2, y2, depth)

          IF depth > TreeOrder THEN EXIT SUB

          LOCAL dx = x2 - x1, dy = y1 - y2
          LOCAL x3 = x2 - dy, y3 = y2 - dx
          LOCAL x4 = x1 - dy, y4 = y1 - dx
          LOCAL x5 = x4 + (dx - dy) / 2
          LOCAL y5 = y4 - (dx + dy) / 2
          MOVE x1, y1
          DRAW TO x2, y2
          DRAW TO x3, y3
          DRAW TO x4, y4
          DRAW TO x1, y1
          pythagoras_tree(x4, y4, x5, y5, depth +1)
          pythagoras_tree(x5, y5, x3, y3, depth +1)

        END SUB
    }
    Pythagoras_tree
```

Database

This is an example of creating and using a simple database with one table and one row in this table

```
BASE "ALFA"  ' erase Dir$+"alfa.mdb" if found it

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

        Hello from Greece

        }

RETRIEVE "ALFA", "Employees", 1,"","""

READ MANY, ID, LASTNAME$, FIRSTNAMES$, MEMO$

PRINT $(4, 6),  "BASE:","ALFA"

PRINT  "NAME:",LASTNAME$ + " " + FIRSTNAMES$

PRINT "MEMO:",

REPORT  MEMO$

CLOSE BASE "ALFA"

PRINT $(0)
```

DataBase Statements
APPEND, BASE, COMPRESS, DB.PROVIDER, DB.USER, DELETE, EXECUTE, ORDER, RETRIEVE, RETURN, SEARCH, STRUCTURE, TABLE, VIEW

More for the Print Statement

All M2000 programs executed by an interpreter in the M2000 environment. This environment has a built in layer functionality. A layer is a visual object on screen, which have specific capabilities, such as printing text and drawing lines, circles, curves and fill with color areas. There are two cursors, the text cursor and the graphic cursor. The Print statement use the text cursor. The layer has rows of specific number of characters positions. The number of rows is the **Height**, and the number of characters positions in a row is the **Width**. So characters can be print in Height*Width places.

The Print statement has same clauses for changing the way of print, and some internal functions. Print always overwrite characters to the given background. When the Print need a new row then the rows scroll up and give a new clear row. The layer has two parts, the upper optional part which has no scroll functionality and the lower part with scroll functionality. The **CLS** statement clear the low part of layer. Also CLS has two optional parameters, the background color and the number of the top row where the second part start, so **CLS 0,0** clear with black the lower part which start from row 0, so there is no room for part one (non scrolling part). A **CLS ,-4** clear the lower part from height-3, with current background color, so we get four lines for scrolling text. Text cursor has 0,0 at top left corner. Also graphic cursor has 0,0 at top left corner. Graphics use unit twip, and **Scale.X** and **Scale.Y** are the number of twips for width and height of console.

There are two read only variables for text cursor, the **Row** and **Pos**, and two more for graphics, **pos.X** and **pos.Y**. We can locate the text cursor using **Cursor** statement, or using Print @() so to Print "Hello" at pos 20 and row 10 then:

```
Cursor 20,10

Print "Hello"
```

or

```
Print @(20,10), "Hello"
```

Both of the two statements above also insert a New Line after the "Hello".
The Print statement use Columns, so we can print numbers in four columns with right justification:

```
Print 1,2,3,4
```

We can break the column rule using the semi colon instead of coma. The following example print X=10 (if we place a coma instead a semi colon we get X=   10)

```
X=10

Print "X=";X
```

The rules above are the same as in BASIC language. But there are more specific for M2000 environment. The three internal functions ~(), @(), $():

The ~() used for change color for the column, change background color for column in the row and change color for a box outline the column in the row. The last two parameters are optional.

The @() used for moving the cursor anywhere and also for coloring the display column.

```
@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor)
@(X,Y, XlowerRightCorner,YlowerRightCorner, BackColor, Outlinecolor)  'make boxes with printing data
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$)   ' fill with an image...stretching
@(X,Y, XlowerRightCorner,YlowerRightCorner, BitmapOrFilename$,  1)  ' lkeep aspect ratio of image
```

So the above used to fill background in an area under the printing. For formatting purposes there is another internal function the $()

```
$(ColumnsJustifyID)
$(ColumnsJustifyID, ColumnsWidth)
$(StringWay2format$)
$(StringWay2format$ , ColumnsWidth)
$("") reset  StringWay2format$
```

Example using the StringWay2format, and setting of 10 character column.

```
Print $("#.000;(#.0000);\z\e\r\o", 10),123.21312,3123.45, -3, 0

Print $(""),  ' comma used to leave cursor to not insert the new line

Print 123.21312,3123.45, -3, 0
```

The ColumnsJustifyID 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

The ColumnsJustifyID 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

So using 4 and above we use Print with proportional text rendering. Also if the text in a string is RTL (right to left, like Arabian or Jewish letters) then the Print return to proportional to print properly the RTL.

The Print statement count diacritics symbols as expected. So maybe a string has a display length less than actual length. We can find the length of a string using LEN() and the display len using Len.Disp()

```
Print Len.Disp("ãž")  ' 2

Print Len("ãž")     '4
```

The three clauses, PART, OVER and UNDER. So Print Part, Print Over and Print Under change temporary to proportional $(4), and neither change the row. Also we can handle the ~() and $() temporary so after the execution of statement we get the old state.

The Print Over make the width as one column, and clear background except for one line at the base of the row. The Print Under first draw the horizontal line (the same that missing from Print Over) and insert a new line before print. We can move the start position to negative numbers, say @(-10) and change the column length as we print columns. So we can get columns with different lengths (and justification). Because we didin't get a new line in Print Part and Print Over we can use it to view large tables which not fit entire in layer and we can control the displaying part, by moving the start origin and redefine the columns.

HEX is the other PRINT statement, which is like Print but the numbers are returned as HEX values unsigned, so a value under 0 or over 0XFFFFFFFF, we get ???- and ???+

We can use ? Instead of Print. So:

```
? "This is a print statement"
```

A layer may have a **Linespace**, a space between rows, from 0 to some pixels, but we have to feed it as twips. We can convert pixels to twips using twipsX or twipsY, so 10*twipsX return the twips for 10 pixels.

A **Form 40,20** make the form 40 characters wide (width=40) and 20 rows (Height=20). This statement handle Linespace automatic.

A console form has Back Layer, Layer, 32 layers above. The Layer Is the standard layer. We can move all layers and change the size of each too. The layers above Back layer are masked by the Back Layer, so we may have a layer two times bigger from Back and scroll it inside Back, viewing a part of it. We use **Motion** statement and **Motion.X** and **Motion.Y** to read the current state of the top left corner. For Back layer we use **Motion.W** and variables **Motion.WX** and **Motion.WX**. The following example change the border of the console layer (changing the layer background behind)

```
Back {
      Cls 11.0
}
```

For each layer environment use am internal structure to save state. So we can change **Font** and font attribute like **italic** and **bold** (we can use variables to read **Fontname$**, **italic** and **bold**), We can use **Double** to get double width and height letters and **Normal** to return it to normal characters.

The 32 layers above standard layer can be used with **Layer** statement using the number 1 to 32. The 32 layer is always above layer 1. Layers can be used as sprites with magnify and rotate function plus a transparent color or a bitmap for a mask.to dedine the transparent bits. The following example show two layers, wait for a keypress, change the priority of layers using the Player statement (which used for sprites), and then swap variables l3 and l4 so now l3 has the real 4 value (always 4 is above 3). Finally we print something and wait for a keypress to hide them.

```
l3=3
l4=4
layer l3 {
      window 12, 6000,12000
      motion 7000, 2000
      cls 7, 0
      pen 0
      show
      print "ok.....3"
}
layer l4 {
      window 12, 6000,12000
      motion 6000,1000
      cls 15, 0
      pen 0
      show
      print "ok.....4"
      print
```

```
        print
}
\\ WAIT A KEY PRESS ANS STORE IT TO a$
a$=key$
player l3 swap l4
swap l3, l4
layer l4 {
        Print "This is 4"
}
layer l3 {
        Print "This is 3"
}

a$=key$

layer l4 {
        hide
}
layer l3 {
        hide
}
```

Another layer is the Printer layer which can be used to print pages. We can change orientation of a page before use it. Also we can set with Form statement the rows and character per row. We can't use Cls, but we can change page. Instead of scrolling a new empty line we get a new page. We use Printer { } to handle the printer page

```
Page 1  ' portrait - not form feed to printer
```

```
Printer {
        Form 64, 66
        Print "To printer"
        Page 1   ' portrait and form feed
        Print "Second Page to printer"
}
```

We can make more (100 more) making user forms, windows. The console form is a form (the size of Back) and maybe can be hide and we can use only forms (windows) and event programming. If we make Form1 then we have the Layer Form1 { } to use it.

See the Report statement (Help Report in M2000 console) which render multi line text, with justification and word wrap.

We can print one or more lines with justification using graphic cursor, with rotation and letter spacing using Legend statement.

Stop Execution

We can stop execution by keyboard using Escape, Ctlr+C and Break key. For Escape there is a statement Escape Off which disable it and Escape On which enable it. Ctrl+C open a msgbox to ask if you want to terminate the code. The same for Break key. Break key also erase all modules.

End Statement

The End statement in console terminate the m2000 environment. From code we can give the same using Set End, The Set statement send the rest line to console interpreter (or command line interpreter) which have some differences from the block of code interpreter.

A Test statement stop execution opening a form Control and we can continue, or execute a step, and we can see the code and the stack (third button from left). The Control form has 6 buttons (the three on the left and top are labels which are buttons also). The three at the right and top do three things: Execute a step, Execute at limited speed, stop execution.

We can use Control form to program some variables for inspection as we execute steps, or we can execute statements (use backspace to erase the line to change the mode, in the textbox at the bottom of form).

The control form can be used with GUI programs too. The modal forms can't disable the control form. Events not showing by control form, but threads showing indicating the thread number

HALT statement
We can use Halt to halt execution end go back to console input, and then we can write Continue to continue the execution. Statements are in global scope

Stop statement
We can use Stop statement which open a message box (default No) and if we click on yes we open in console the input prompt. We have to use exit to return to statements. Statements in Stop state are in local scope

Error Statement
We can produce an error using a string or a number or a string with a number in last chars on string. **Error 0** is the Fatal Error which open the windows message box and after we press ok the interpreter terminate.

Using Context Menu in Task Bar
We can close the Environment by choosing close in context menu in task bar. Also we can minimize the full screen from that context menu. Also Ctrl+F4 make the same. Its behave like a End statement. If we change the modules in module list the a message box ask if you want to cancel the exit of environment.

Starting a Program

To start a program you can execute first the environment, m2000.exe and then choose these options: To see what files (type of gsb) exist in user directory execute MODULES statement.

You may want to erase any program: use NEW
You may want to clear any global variable and static variables too, use CLEAR
You may want to flush stack of values, use FLUSH
You may want to reset Environment (stack not flushing) use Start or Start "","""
Reset can be done using Break button.

Manual execute a module

Load alfa (for alfa.gsb) or load "alfa beta" for "alfa beta,gsb". Press ctrl+N or Modules ? to see what modules you have and write the name of one and press enter.

Automatic execute a module

You can save the program with a series of statements at the end. When you load the program that statements also executed.

Start a program by double click on file icon

You can start a program (which can be start in automatic execution of a module), before you start environment. You have to confirm to OS that gsb opened with M2000.exe. Also in that case you have to use Show statement to show Console form because by default the

console open if we open manual the environment (starting it without passing a file to execute in command line), or when an Input statement need to use console for value input, so the console turn to show mode.

If you have a GUI program, may you don't need to open console, so you don't have to use Show. Also Show statement not only show the form but make it foreground window,

To end a GUI program you may use ctrl+F4, or the square on the top corner (in one corner by default the left but can be change to right).

Advanced Programming Style

More advanced examples.

The Pass By Reference &identifier vs Weak$(identifier)

This is the way to pass by reference using &indentifier from calling and from callee:

```
Module Callee {   ' or  Module Callee (&X) { X++}
        Read &X
        X++
}
M=10
Callee &M
Print M=11
```

About Weak references. A &A return a string which is a weak reference. So this is an example using Weak$. An identifier can be a reference only once, at definition. A string using it as weak reference can change reference.

```
Module aName {
        A=10
        Print &A=Weak$(A) ' true
        A$=Weak$(A)
        Print Eval(A$)=10
        A$.=30   ' see dot after A$
        Print A=30
        List
}
Call aName    ' call using Call
```

```
aName  ' call without using Call - See difference on the output of List
```

Not all **&indentifier** return a weak reference equal to weak$(). Almost all except of the functions. So a &funct1() is not weak$(func1()). The &func1() has the code of func1() and weak$(func1()) is the weak function.

A function reference also may include another weak reference, the object where belong. Object have to be a named group. Here we have the Alfa group. The .acc is private so we can't read from outside, the Alfa.acc not exist. Also Alfa not return a group like the standard group because has a value clause. So Alfa return the .acc value. Public function is the final function Alfa.Pow(). Module inner expect a function (or array if the indexes allow to be used in place of function). We can give any function, but here we give from a named group, so we get feedback about the number of use of it.

```
Group Alfa {
Private:
      acc=0
Public:
      Function Final Pow(X,Y) {
            =X**Y
            .acc++
      }
      Value {
            =.acc
      }
}
Module Inner(&c()) {
      Print c(2,3), c(4,2)
}
Inner &Alfa.Pow()
Print Alfa=2
```

Weak Reference for module

A module isn't first citizen, but we can produce a weak reference as string. Check following code:

```
Module Checkit {
      Module k {
            print number**2
      }
      k 10   ' print 100
      Module inner (a$) {
            a$. 10
            call a$, 10
      }
      inner weak$(k)
}
CheckIt
```

Weak Reference and Link to new Identifiers

Check this code for Functions, Arrays, Numeric Variable and String Variable, utilize the weak$() function. A Def function make one line function (no need for block)

```
Def Pow(X,Y)=X**Y
Pow$=weak$(Pow())
Print Function(Pow$,2,3)=8

\\ make a reference from a weak reference
Link weak Pow$ to P()
Print P(2,3)=8
```

```
Dim K(1 to 8, 1 to 8)=1
K(2,2)=100
Board$=weak$(K())
Print Array(Board$, 2, 2)=100

\\ make a reference from a weak reference
Link weak Board$ to K2()
Print K2(2,2)=100
K2(2,2)+=10
Print K(2,2)=110

X=10
X$=weak$(X)
Print Eval(X$)=10

\\ make a reference from a weak reference
Link weak X$ to XX
XX++
Print X=11

Z$="ok"
ZZ$=weak$(Z$)
Print Eval$(ZZ$.)="ok"
```

```
\\ make a reference from a weak reference

Link weak ZZ$ to ZZZ$

ZZZ$+="..."

Print Z$="ok..."
```

Static Variables

Static variables have some rules. We can create simple variables (no arrays) and only tuples, stacks, inventories (as pointers)from objects. We can't create references, except when we call a module or a function (not for simple functions and subroutines).

How static variables stored, in a language with dynamic allocation for modules and functions at the execution? The variables stored in the caller execution object. So this happen up to the root as we go back returning from call. If we have a global module and this has static variables then from different modules we call this module with different set of static variables.

```
\\ count from 10 to 15 rthen 10 to 15 and so on

\\ static variables can be passed by reference...

\\ only in modules and functions, but not in subs

\\ Using a copy in - copy out mechanism

Module CheckStatic {

        static a=9

        Module inner (&b) {

                b++

        }

        Inner &a

        Print a

        if a>=15 then clear ' clear variables including static variables

}
```

```
For i=1 to 20
      CheckStatic
Next I
```

Threads

A thread is a block of code with a handler (a number in a variable), with no name.
We can use any of two plans, using Thread.Plan:

- Thread.Plan concurrent
- Thread.Plan sequential (default)

In concurrent mode all commands in thread level are processed one step in time. If we have to process a block then all block processed without interruption. In this mode maybe two threads starts concurrent but not end at same time. Interval for threads aren't expanded (see next)

In sequential mode no thread allow any other thread to break it. This law sometimes break, especially if we call a command that wait for an input. Threads then run in the background. In this mode if a thread slow down then time interval recalculated. So we can place a command to fix this inside thread, for 100 millisecond, we must use THIS inside thread because the thread is out of thread pool when running, so giving a thread number interpreter miss the thread, but using THIS we make interpreter to communicate with process itself and alter internal variables – and these variables are checked from task manager, before putting again in threading pool, (thread this interval 100).
Any thread can Hold, Restart, Erased or we can set Interval. We can access these functions from thread using THIS, or out of thread using the number that we get in a variable when we construct the thread, and we say this a Handler.

Threads have own value's stack. Threads in same module share variables, arrays and modules and functions that are defined in that module. Threads are block of code that runs in time intervals. If a module ends then all threads in that module stopped and deleted (we can press Esc and we stopped them also). New variable in a thread is a new variable in module.

We can use Main.Task time { } as a main thread. We have to start one thread before, and this thread if exit erase threads, or if all treads other threads end then exit too.

```
m=100
thread {
      m--
      if m<1 then thread this erase
```

```
} as countdown interval 100

Main.Task 25 {

        Print m, "inside main task"

        if keypress(1) then exit

}

Threads ' just report

Print "ok"
```

We can use EVERY time { } but Every isn't thread so we loose right time slice for every thread. (Every is a block with a Wait command inside)

We can use Wait time to give time to run a thread.

There is the AFTER time { } an automatic thread to execute once after some milliseconds

Threads Example Dining Philosophers

There are 5 philosophers: Aristotle, Kant, Spinoza, Marx, Russell in a round table and there are 5 forks. Each philosopher may think or eat pasta. To eat has to get two forks. May eat any quantity of pasta, and then return to thinking placing the forks to table.

The problem is that if the five philosophers get one fork then they can't find a second fork, so the energy drop and they die. If each philosophers get the right fork then this happen and we say that we have a deadlock. But what happen if they change randomly the order of picking, so some time the pick the right first and other time pick the left first

The best strategy is to monitoring and counting the times when the forks are all out of table and then we have to decide how we place back a fork and from whom philosopher. If we choose to steady any philosopher pick the right fork first then we will see starving philosophers and maybe the death of one or more.

So we have 6 threads, 5 for philosophers and 1 as Main.Task where we display information in a period of 4hz (4 times per second), and checking the number of forks in table.

```
Module Dining_philosophers (whichplan) {

Form 80, 32

Const MayChangePick=Random(True, False)
```

```
Dim energy(1 to 5)=50
Document Doc$
Const nl$={
}
Print $(,12),  ' set column width to 12
Pen 14
Pen 15        {
        Doc$="Dining Philosophers"+nl$
        \\ we can change thread plan only if no threads defined
        if whichplan=1 then
                Doc$="Sequential threads - to execute exclusive one threads code"+nl$
                thread.plan sequential
                \\ need time_to_think>time_to_eat, but time_to_appear maybe the same for all
                time_to_think=150  ' one or more intervals
                time_to_eat=100 ' one interval to eat only
                time_to_appear=(150,150,150,150,150)
                Return time_to_appear, random(0,3):=300
        else
                Doc$="Concurrent threads  - to execute a statement or a block of code"+nl$
                thread.plan concurrent
                time_to_think=100  ' one or more intervals
                time_to_eat=50 ' one interval to eat only
                time_to_appear=(100,100,100,100,100)
```

```
                Return time_to_appear, random(1,4):=200
        end if
        Print #-2,Doc$
        Print @(0,2),"Press left mouse button to exit"
        Print Part $(1), time_to_appear
        Print under
}
Pen 13 {Print "Aristotle", "Kant", "Spinoza", "Marx", "Russell"}
enum philosopher {
        Aristotle, Kant, Spinoza, Marx, Russell
}
global enum forks {NoFork, Fork}
RoundTable =(Fork, Fork, Fork, Fork, Fork)
Getleft=lambda RoundTable (ph as philosopher) -> {
        where=(ph+4) mod 5
        = RoundTable#val(where)
        Return RoundTable, where:=NoFork
}
GetRight=lambda RoundTable (ph as philosopher) -> {
        where=ph mod 5
        =RoundTable#val(where)
        Return RoundTable, where:=NoFork
}
PlaceForks=lambda RoundTable (ph as philosopher) -> {
```

```
        Return RoundTable,  (ph+4) mod 5:=Fork,ph mod 5:=Fork

}

PlaceAnyFork=lambda RoundTable (ph as philosopher, &ForkL, &ForkR) -> {

        If ForkL=Fork then Return RoundTable,  (ph+4) mod 5:=Fork :
ForkL=NoFork

        If ForkR=Fork then  Return RoundTable, ph mod 5:=Fork :
ForkR=NoFork

}

ShowTable=lambda RoundTable -> {

        m=each(RoundTable)

        while m

                print if$(array(m)=NoFork->"No Fork", "Fork"),

        end while

        Print

}

noforks=lambda RoundTable -> {

        k=0

        m=each(RoundTable)

        while m

                if array(m)=NoFork then k++

        end while

        =k=5

}


def critical as long, basetick
```

```
Document page$

m=each(philosopher)

while m {

        \\ we make 5 threads

        \\ a thread has module scope (except for own static variables, and stack
of values)

        thread {

                if  energy(f)<1 then {

                                call PlaceAnyFork(f, ForkL, ForkR)

                                energy(f)=0

                                Page$=format$("{0::-12} - ",tick-basetick)+eval$(f)+" -
Die"+nl$

                                thread this erase

                } else  {

                                Page$=format$("{0::-12} - ",tick-basetick)+eval$(f)

                                Page$=if$(ForkL=NoFork or ForkR=NoFork->"
thinking",  " eating"+str$(eatcount))

                                Page$=if$(R->"- R", " - L")+nl$

                }

                if not think then

                        { \\ a block always run blocking all other threads

                                energy(f)++

                                eatcount--

                                if eatcount>0 then exit

                                Call PlaceForks(f) : ForkL=NoFork:ForkR=NoFork

                                eatcount=random(4,8)
```

```
                        if MayChangePick then R=random(-1,0)

                        think=true :thread this interval
time_to_think*random(1,5)

                }

        else.if energy(f)>70 or critical>5 then

                {

                        call PlaceAnyFork(f, &ForkL, &ForkR)

                        if energy(f)>70  then energy(f)=60

                }

        else.if R then

                        if ForkR=NoFork then ForkR=GetRight(f)

                        if ForkR=Fork and ForkL=NoFork then
ForkL=GetLeft(f)

                        if ForkL=Fork then think=false:thread this interval
time_to_eat else energy(f)--

        else

                        if ForkL=NoFork then ForkL=GetLeft(f)

                        if ForkL=Fork and ForkR=NoFork then
ForkR=GetRight(f)

                        if ForkR=Fork then think=false:thread this interval
time_to_eat else energy(f)--

        end if


} as a interval time_to_appear#val(m^)

\\ a is a variable which hold the number of thread (as returned from task
manager)

\\ so we can get 5 times a new number.
```

```
        \\ for each thread we make some static variables (only for each thread)
        \\ this statement execute a line of code in thread a
        thread a execute {
                \\ this executed on thread execution object
                static f=eval(m), think=true, ForkL=NoFork
                static ForkR=NoFork, eatcount=random(2,5)
                static R=-1
                if MayChangePick then  R=Random(-1,0)
        }
}
cls ,5  ' set split screen from fifth row
\\ Main.Task is a thread also. Normaly exit if no other threads running in background
\\ also serve a the wait loop for task manager (we can use Every 200 {} but isn't a thread, is a kind of a wait statement)
\\ tick return the counter from  task manager which used to triger threads
basetick=tick
\\ 4hz display results
MaxCritical=0
Main.Task 1000/4 {
        { \\ a block always run blocking all other threads
                cls
                Print Part $(1),$("####;\D\I\E;\D\I\E"),energy()
                Print Under
                Print "Table:"
```

```
                Call ShowTable()
                if noforks() then critical++  else critical=0
                MaxCritical=if(MaxCritical<critical->critical,MaxCritical)
                Print "noforks on table counter:";critical, "Max:";MaxCritical
                Print #-2,Page$
                Doc$=Page$
                Clear Page$
            }
            if critical>40 or keypress(1) then exit
        }
        threads erase
        Clipboard Doc$
        }
        Dining_philosophers Random(1,2)
```

Using Lambda functions

Dijkstra's_Algorithm

This algorithm show how we find the optimal path from on node to another in a Graph. Out Graph has nodes with one direction to some other nodes, and for each small path between two connected nodes we have a cost. The scope of this program is to show the Dijkstra's algorithm, and how we can put the Graph in a structures of tulples who have tuples also.

We use Infinite constant (a double value) and for Graph we place.
We use a tuple with a list of all nodes. For each node we have a tuple with the node name as string and then the paths, as tuples of two values, the node name to where we direct and the cost of it. If a node has no direction to any other node then we have to place Term a tuple with null string and null cost ("", 0). We look for **edge#val$(0)<>""** but we can use the advanced **not edge is term** which is the same here and return false if pointer in edge is different from pointer in term. The later need the term to be in scope, but the first comparison is more versatile because it is value oriented. So a tuple of (("",0),("",0)) has two tuples with same values but different pointers, and (term, term) has two tuples with

same pointer. As we see the Edges is a jagged array, has one level for nodes, and a second level for paths to nodes. Also note that using letters for nodes is like we use weak references. The path is hard structure by tuples, but the nodes have no physical pointers to nodes. Think about a structure where each node has a table of two things per row, the pointer to another node and the cost value. In that case we have many pointers to nodes, so a pointer for c has to included in all other nodes which points (as path) to c. And because a node may have from zero to many paths to nodes, we have to use a table of nodes and a private linked list of three values the pointer of destination node, a cost and a pointer to another node or null (no other node destination for the given starting node). To break this structure, we have to pass two times, one per node to erase each path from linked list and final to erase the table of nodes. So we didn't do such a thing, and we do just erasing of Edges, whith a cascade "message" to destroy object. Each embedded object send the same message from first to last object in the array, so at the return we get an empty Edges from objects. Only the term object survive because a second pointer exist In term variable. Because the objects have a counter of pointer references, when the term erased as value/object from Edges the counter from 2 turned to 1 and when term erased the counter turned to 0 and the object destroyed. So here we didn't do any special erasing of linked lists, (which are hard connected), because of the nature of tuple structure and the use of weak references, the names of nodes as strings.

```
Module Dijkstra`s_algorithm {

    const max_number=infinity

    GetArr=lambda (n, val)->{

        dim d(n)=val

        =d()

    }

    term=("",0)

    Edges=(("a", ("b",7),("c",9),("f",14)),("b",("c",10),("d",15)),("c",("d",11),("f",2)),
("d",("e",6)),("e",("f", 9)),("f",term))

    Document Doc$="Graph:"+{

    }

    ShowGraph()

    Doc$="Paths"+{

    }

    Print "Paths"
```

```
        For from_here=0 to 5
            pa=GetArr(len(Edges), -1)
            d=GetArr(len(Edges), max_number)
            Inventory S=1,2,3,4,5,6
            return d, from_here:=0
            RemoveMin=Lambda S, d, max_number-> {
                ss=each(S)
                min=max_number
                p=0
                while ss
                    val=d#val(eval(S,ss^)-1)
                    if min>val then let min=val : p=ss^
                end while
                =s(p!)  ' use p as index not key
                Delete S, eval(s,p)
            }
            Show_Distance_and_Path$=lambda$ d, pa, from_here, max_number (n) -
    > {
                ret1$=chr$(from_here+asc("a"))+" to "+chr$(n+asc("a"))
                if d#val(n) =max_number then =ret1$+ " No Path" :exit
                let ret$="", mm=n, m=n
                repeat
                    n=m
                    ret$+=chr$(asc("a")+n)
                    m=pa#val(n)
```

```
        until  from_here=n
        =ret1$+format$("{0::-4} {1}",d#val(mm),strrev$(ret$))
    }
    while len(s)>0
        u=RemoveMin()
        rem Print u, chr$(u-1+asc("a"))
        Relaxed()
    end while
    For i=0 to len(d)-1
        line$=Show_Distance_and_Path$(i)
        Print line$
        doc$=line$+{
        }
    next
next
Clipboard Doc$
End
Sub Relaxed()
    local vertex=Edges#val(u-1), i
    local e=Len(vertex)-1, edge=(,), val
    for i=1 to e
        edge=vertex#val(i)
        if edge#val$(0)<>"" then
            val=Asc(edge#val$(0))-Asc("a")
```

```
            if d#val(val)>edge#val(1)+d#val(u-1) then
                Return d, val:=edge#val(1)+d#val(u-1)
                Return Pa, val:=u-1
            end if
        end if
    next
end sub
Sub ShowGraph()
    Print "Graph"
    local i
    for i=1 to len(Edges)
        show_edges(i)
    next
end sub
Sub show_edges(n)
    n--
    local vertex=Edges#val(n), line$
    local e=each(vertex 2 to end), v2=(,)
    While e
        v2=array(e)
        line$=vertex#val$(0)  ' break the line for this paper
        line$+=if$(v2#val$(0)<>""->"+v2#val$(0)+format$(" {0::-2}",v2#val(1)),"")
        Print line$
        Doc$=line$+{
```

```
            }
        end while
    end sub
}
Dijkstra`s_algorithm
```

Export this:

```
Graph:
a->b  7
a->c  9
a->f 14
b->c 10
b->d 15
c->d 11
c->f  2
d->e  6
e->f  9
f
Paths
a to a   0 a
a to b   7 ab
a to c   9 ac
a to d  20 acd
a to e  26 acde
a to f  11 acf
b to a No Path
b to b   0 b
b to c  10 bc
b to d  15 bd
b to e  21 bde
b to f  12 bcf
c to a No Path
c to b No Path
c to c   0 c
c to d  11 cd
c to e  17 cde
c to f   2 cf
d to a No Path
d to b No Path
d to c No Path
d to d   0 d
d to e   6 de
d to f  15 def
```

```
e to a No Path
e to b No Path
e to c No Path
e to d No Path
e to e   0 e
e to f   9 ef
f to a No Path
f to b No Path
f to c No Path
f to d No Path
f to e No Path
f to f   0 f
```

Example RIPEMD-160

In the following example we use an algorithm for RIPEMD-160 (slow for an interpreter, but works). We use many Binary functions. The binary.add() add two 32 bit numbers and drop the overflow bit (if happen to produced). We place lambda functions in array items.

We make one lambda function which have closures other lambda functions. We use a buffer (see statement: **buffer clear message as byte*64**)

There are three REM code lines (which we break them so we get Print to get information, but we can delete the new line and leave them to REM to hide these lines)

```
\\ https://en.wikipedia.org/wiki/RIPEMD
\\ The original RIPEMD function was designed in the framework of the EU project RIPE (RACE Integrity Primitives Evaluation) in 1992
\\ RIPEMD-160 is 160bit cryptographic hash function.

Module Checkit {
      Function Prepare_RiPeMd_160 {
            Dim Base 0,  K(5), K1(5)
            K(0)=0x00000000, 0x5A827999, 0x6ED9EBA1, 0x8F1BBCDC, 0xA953FD4E
            K1(0)=0x50A28BE6,0x5C4DD124, 0x6D703EF3, 0x7A6D76E9, 0x00000000
            Dim Base 0,r(80), r1(80), s(80), s1(80)
            r(0)=0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
            r(16)=7, 4, 13, 1, 10, 6, 15, 3, 12, 0, 9, 5, 2, 14, 11, 8
            r(32)= 3, 10, 14, 4, 9, 15, 8, 1, 2, 7, 0, 6, 13, 11, 5, 12
```

```
r(48)=1, 9, 11, 10, 0, 8, 12, 4, 13, 3, 7, 15, 14, 5, 6, 2
r(64)=4, 0, 5, 9, 7, 12, 2, 10, 14, 1, 3, 8, 11, 6, 15, 13
k=r() : k*=4   ' k is a pointer to array. We have to multiply to make them offsets

r1(0)=5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, 10, 3, 12
r1(16)=6, 11, 3, 7, 0, 13, 5, 10, 14, 15, 8, 12, 4, 9, 1, 2
r1(32)=15, 5, 1, 3, 7, 14, 6, 9, 11, 8, 12, 2, 10, 0, 4, 13
r1(48)=8, 6, 4, 1, 3, 11, 15, 0, 5, 12, 2, 13, 9, 7, 10, 14
r1(64)=12, 15, 10, 4, 1, 5, 8, 7, 6, 2, 13, 14, 0, 3, 9, 11

k=r1() : k*=4

s(0)=11, 14, 15, 12, 5, 8, 7, 9, 11, 13, 14, 15, 6, 7, 9, 8
s(16)=7, 6, 8, 13, 11, 9, 7, 15, 7, 12, 15, 9, 11, 7, 13, 12
s(32)=11, 13, 6, 7, 14, 9, 13, 15, 14, 8, 13, 6, 5, 12, 7, 5
s(48)=11, 12, 14, 15, 14, 15, 9, 8, 9, 14, 5, 6, 8, 6, 5, 12
s(64)=9, 15, 5, 11, 6, 8, 13, 12, 5, 12, 13, 14, 11, 8, 5, 6

s1(0)=8, 9, 9, 11, 13, 15, 15, 5, 7, 7, 8, 11, 14, 14, 12, 6
s1(16)=9, 13, 15, 7, 12, 8, 9, 11, 7, 7, 12, 7, 6, 15, 13, 11
s1(32)=9, 7, 15, 11, 8, 6, 6, 14, 12, 13, 5, 14, 13, 13, 7, 5
s1(48)=15, 5, 8, 11, 14, 14, 6, 14, 6, 9, 12, 9, 12, 5, 15, 8
s1(64)=8, 5, 12, 9, 12, 5, 14, 6, 8, 13, 6, 5, 15, 13, 11, 11

Dim Base 0, T(5), TT(5)
T(0)=lambda ->binary.xor(binary.xor(number,number),number)
T(1)=lambda (B,C,D)->binary.or(binary.and(B,C), binary.and(binary.not(B), D))
T(2)=lambda ->binary.xor(binary.or(number, binary.not(number)), number)
```

```
            T(3)=lambda (B,C,D)->binary.or(binary.and(B,D), binary.and(C,binary.not(D)))
            T(4)=lambda ->binary.xor(number, binary.or(number, binary.not(number)))

            \\ no need for variables we read form stack with number
            TT(0)=lambda ->binary.xor(number, binary.or(number, binary.not(number)))
            TT(1)=lambda (BB,CC,DD)->binary.or(binary.and(BB,DD),
binary.and(CC,binary.not(DD)))
            TT(2)=lambda ->binary.xor(binary.or(number, binary.not(number)), number)
            TT(3)=lambda (BB,CC,DD)->binary.or(binary.and(BB,CC),
binary.and(binary.not(BB),DD))
            TT(4)=lambda ->binary.xor(binary.xor(number,number),number)

            \\ return of this function is a lambda function
            \\ all arrays are closures to this lambda
            =lambda K(),K1(),TT(), T(),r(),r1(), s(), s1() (&message$, ansi as
boolean=true, ansiid=1033)-> {
                  set fast!
                  def h0 = 0x67452301, h1 = 0xEFCDAB89, h2 = 0x98BADCFE
                  def h3 = 0x10325476, h4 = 0xC3D2E1F0
                  def i, j, l, padding, l1, blocks, acc, f64 as boolean=true, oldid
                  if ansi then oldid=locale : locale ansiid
                  \\ we use a buffer of 64 bytes
                  buffer clear message as byte*64
                  l=len(message$)*if(ansi->1,2 )
                  if binary.and(l,63)>55 then  padding=64
                  padding+= 64 - (l Mod 64)
                  l1=padding+l+1

                  f64=binary.and(l,63)<>0
```

```
                blocks=l1 div 64
rem
                Print "blocks:";blocks
                \\ now prepare the buffer
                PrepareBuffer()
                def decimal  A, B, C, D, E, AA, BB, CC, DD, EE, T, TT
                do
                A  = h0 : B  = h1 : C  = h2 : D  = h3 : E  = h4
                AA = h0 : BB = h1 : CC = h2 : DD = h3 : EE = h4
                for J=0 to 79 {
                        JJ=J DIV 16
                        PUSH binary.add(Binary.Rotate(binary.add(A,T(JJ)
(B,C,D),eval(message ,r(j) as long),k(jj)), s(j)), e)
                        A = E : E = D : D = Binary.Rotate(C, 10) : C = B : READ B
                        PUSH binary.add(Binary.Rotate(binary.add(AA,TT(JJ)
(BB,CC,DD),eval(message, r1(j) as long),k1(jj)),s1(j)),EE)
                        AA = EE : EE = DD : DD = Binary.Rotate(CC, 10) : CC = BB :
READ BB
                }
                push binary.add(h1, C, DD)
                h1 = binary.add(h2, D, EE)
                h2 = binary.add(h3, E, AA)
                h3 = binary.add(h4, A, BB)
                h4 = binary.add(h0, B, CC)
                Read h0
                blocks--
rem
                print over $(0,8), blocks : Refresh
                if blocks=0 then exit
                PrepareBuffer()
```

```
                always
rem
                print
                buffer ans as byte*20
                \\ we put ulong (long ar ulong in buffers)
                Return ans, 0:=h0 as long, 4:=h1 as long,8:=h2 as long, 12:=h3 as
long, 16:=h4 as long
                =ans
                if ansi then locale oldid
                set fast
                Sub PrepareBuffer()

                        if l-acc>=64 then
                                LoadPart(64)
                        else.if blocks=1 then
                                return message, 0:=string$(chr$(0),32)
                                if l-acc=0 and f64 then
                                        Return message, 56:=l*8 as long,
60 :=binary.shift(l,-29) as long
                                else
                                        Return message, l-acc:=0x80, 56:=l*8 as long,
60 :=binary.shift(l,-29) as long
                                        if l>acc then LoadPart(l-acc)
                                end if
                        else
                                Return message, l-acc:=0x80
                                LoadPart(l-acc)
                        end if
                End Sub
                sub LoadPart(many)
```

```
                \\ str$() convert to ansi, one byte per character
                \\ using 1033 as Ansi language
                if ansi then
                        Return message, 0:=str$(mid$(message$,1+acc, many))
                else
                        Return message, 0:=mid$(message$, 1+acc, many)
                end if
                acc+=many
            end sub
        }
    }
    Module TestHash (RIPEMD){
        Flush
        \\ push data to stack of values, as fifo (each entry append to end of stack)
        Data "b3be159860842cebaa7174c8fff0aa9e50a5199f","Rosetta Code"
        Data "9c1185a5c5e9fc54612808977ee8f548b2258d31",""
        Data "0bdc9d2d256b3ee9daae347be6f4dc835a467ffe","a"
        Data "8eb208f7e05d987a9b044a8e98c6b087f15a0bfc","abc"
        Data "5d0689ef49d2fae572b881b123a85ffa21595f36", "message digest"
        Data
"f71c27109c692c1b56bbdceb5b9d2865b3708dbc","abcdefghijklmnopqrstuvwxyz"
        Data "b0e20b6e3116640286ed3a87a5713079b21f5189"
        Data
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"
        Data "9b752e45573d4b39f4dbd3323cab82bf63326bfb", String$
("1234567890",8)
rem     Data "52783243c1697bdbe16d37f97f68f08325dc1528", String$("a",1000000)

        While not empty
            Read check$, text$
```

```
                    Print "RIPEMD160 for ";quote$(Left$(if$(len(text$)>30->left$
(text$,27)+"...",  text$),30))
                    \\ pass text$ by reference
                    Display(RIPEMD(&text$))
                End While

                sub Display(ans)
                    local answer$
                    for i=0 to len(ans)-1
                        answer$+=hex$(eval(ans,i),1)
                    next i
                    Print lcase$(answer$)
                    Print lcase$(answer$)=check$
                end sub
            }
            TestHash Prepare_RiPeMd_160()
}
Checkit
```

Object Oriented Programming

An OOP program use objects. M2000 isn't a pure object language. We can use objects when we want some functionality where the old style can't give.

Using Groups

We can compose groups from other groups. A class is a function which produce a group. We can merge definitions also: Groups may have nested groups. We can put Groups in arrays, inventories, stacks, and as closures in lambda functions.

```
        M=lambda (Z) -> {
            Group alfa {
                X=Z
```

```
        }
        =lambda alfa ->{
                =alfa.x
                alfa.x++
        }
}(100)
Print M()=100, M()=101
MM=M     ' save state to MM
Print M()=102, M()=103
Print MM()=102, MM()=103
```

The alfa group is a named group, and has a unique pointer which we can't use. The group act as a value. So now we change the rules. We use a pointer to a copy of alfa Group. Now we can't save the state to MM, because we just copy the pointer. So pointers to groups used to share state.

```
M=lambda (Z) -> {
        Group alfa {
                X=Z
        }
        =lambda a=Pointer((alfa)) ->{
                =a=>x
                a=>x++
        }
}(100)
Print M()=100, M()=101
MM=M
Print M()=102, M()=103
```

```
Print MM()=104, MM()=105
```

To share state between groups we can use a member as a pointer to group. Another way is to use a Superclass. A Superclass is an instance of a group with no public members, but include a hidden group and we can use it to make new groups and appl;y the pointer to hidden group which hold unique state. A group may have one SuperClass. A SuperClass can't have a SuperClass. Any nested group may have different Superclass. When we merge a group to another group we also replace Superclass if the group in right expression has own Superclass.

```
Superclass Alfa {
Unique:
        counter=0
Public:
        state=10
        Module doit (x) {
                .state<=x
                For SuperClass, this {
                        .counter++
                        \\ notice the double dot before state
                        Print .counter, ..state
                }
        }
}
A=Alfa
B=Alfa
A.doit 100     ' 1  100
B.doit 300     ' 2  300
A.doit 100     ' 3  100
```

```
A.doit 100     ' 4  100
B.doit 300     ' 5  300
Class Other {
        state2=100
        Function ShowCounter {
                for superclass {
                        =.counter
                }
        }
}
C=Other() with Alfa
C.doit 500     '6  500
Print C.ShowCounter()=6
```

In the example above we use the With operator (for groups only). We can use C=Alfa and C=Other() or the better C=Other() with Alfa (we can use anything which have a Group as return value. Here we use a class Other() and a superclass Alfa

Merging has some benefits, but how we get lesser members from a group? In the following example we make a basic and a likebasic class. The difference is that in basic we restrict the merging only to those members we have in the class. We use the Set function and we get from stack of values a group which only the members with same name assigned to current object. Also we make another class normal as basic. Now we combine definitions prior of making instance of class. So normal is a basic too. We can use many AS Class clause but here only for other classes (Without parenthesis).

```
Class Likebasic {
        x=10, y=29
Class:
        Module basic (.x, .y) {}
}
```

```
Class basic {
        x=10, y=29
        set {
                read this
        }
Class:
        Module basic (.x, .y) {}
}
Class normal as basic {
        Z=300
        Module Apply (&Basic) {
                Basic=This
        }
Class:
        Module normal (.x, .y) {}
}
B=basic(2, 3)
Print B.x=2, B.y=3
N=normal(500, 1000)
Print N.x=500, N.y=1000
N.Apply &B
Print B.x=500, B.y=1000
B=basic(2, 3)
B=N
```

```
Print B.x=500, B.y=1000

Print Valid(B.Z)=False

Z=LikeBasic(4,5)

Z=N

Print Z.x=500, Z.y=1000

\\ Now Z.Z exist

Print Valid(Z.Z)=True
```

See the example Inheritance as the last part of this unit.

Simple OOP Example

In the following example we use a group which return a string value. We have to place the name using $. Group alfa$ make two names, alfa anda alfa$. The first name used to access properties/methods of group.

We can make Private and Public parts. A group normally return a copy of it (a volatile group) unless we have a value function. We can still get the copy of it if we use Group$ (alfa$) or Group$(alfa).

```
Group alfa$ {
      Private:
            s$
      Public:
            many
            Value {
                  =.s$
            }
```

```
                Set (.s$) {
                        .many++
                }
        }

        Print alfa$, alfa.many=1
        z$=Group$(alfa$)
        Print z$, z.many=1

        Module Beta (&k$) {
                Print k$, k.many
                k$="ok2"
        }
        Beta &alfa$
        Print alfa$, alfa.many
```

Clasic Book Example

The class: part removed when the group change to volatile one. Because class is a function the Class Book make a local Book Group where module Book exist, and then return the Book Group as volatile group (nameless), with on;u the four final variables.

```
Class Book {
    \\ these are public but can get value one time
    Final title$, des$, price, pback
    \\ we use a class part, which means a part that exist only at contruction time
Class:
    Module Book (a$, b$, c as decimal, d as boolean) {
\\ we use fresh variables to catch a missing value
```

```
            .title$<=a$
            .des$<=b$
            .price<=c
            .pback<=d
        }
    }
    Class BookDB {
    Private:
        countme=0
        dim p()
    Public:
        module final AddBook {
            newdim =.countme+1
            dim .p(newdim)
            read .p(.countme)
            .countme++
        }
        module final ProcessPaperbackBooks (&ProcessBookDelegate) {
            if .countme<1 then break
            for i=0 to .countme-1 {
                if .p(i).pback then call  ProcessBookDelegate.processBook(.p(i))
            }
        }
    }
    group PrintTitle {
        function final processBook(AnyBook) { Report format$("    {0}", AnyBook.title$)}
```

```
}

group PriceTotaller {

      items, total

      function final processBook(AnyBook) {.items++ :  .total+=AnyBook.price }

      \\  look the <= operator, if we use = then we define local variables. Items and total are like global
but for group only

      module final zero { .total<=0 : .items<=0   }

      function final AveragePrice { if .items>0 then =.total/.items }

}

bookDB=bookDB()

report "First Book DB"



bookDB.AddBook Book("The C Programming Language",  "Brian W. Kernighan and Dennis M.
Ritchie", 19.95, true)

bookDB.AddBook Book("The Unicode Standard 2.0",  "The Unicode Consortium", 39.95, true)

bookDB.AddBook Book("The MS-DOS Encyclopedia", "Ray Duncan", 129.95, false)

bookDB.AddBook Book("Dogbert's Clues for the Clueless", "Scott Adams", 12.00, true)

bookDB.ProcessPaperbackBooks &PriceTotaller

report  "Average Paperback Book Price: $" + str$(PriceTotaller.AveragePrice(),"#.##")

bookDB.ProcessPaperbackBooks &PrintTitle



report "Second Book DB"    \\report prints text  proportionally with justifiation

PriceTotaller.zero

SecondbookDB=bookDB()

SecondbookDB.AddBook Book("Any Big Big and Big Tilte",  "Any Author", 3.45, true)

SecondbookDB.ProcessPaperbackBooks &PriceTotaller

report  "Average Paperback Book Price: $" + str$(PriceTotaller.AveragePrice(),"#.##")

SecondbookDB.ProcessPaperbackBooks &PrintTitle
```

Example Event Listener

This ia an example which we make all the code we need for events, without using event objects or light events in groups. This example used for the making of events in M2000 (was written before interpreter get events). Also this written before the introduction of pointers to groups. So we use weak references of "buttons" and by reference call for call back.

```
Module OOP {
      Set Fast
      cls 1, 0
      pen 14
      report 2, "Event Listener"
      report "Use keys 1, 2 and mouse left, and for exit mouse right"
      cls, 2
      class mouselistener {
            name$="MousePress",  yourkey,  auto
            dim ref$()
            module mouselistener {
                  read .yourkey
                  .name$<=format$("{0}({1})",.name$,.yourkey)
            }
            module register {
                  .auto++
                  dim .ref$(.auto)
                  read .ref$(.auto-1)   \\ weak reference
            }
            function CheckEvent {
```

```
            =Mouse=.yourkey
        }
    }
    class keylistener {
        name$="KeyPress", yourkey, auto
        dim ref$()
        module keylistener {
            read .yourkey
            .name$<=format$("{0}({1})",.name$,.yourkey)
        }
        module register {
            .auto++
            dim .ref$(.auto)
            read .ref$(.auto-1)   \\ weak reference
        }
        function CheckEvent {
            =keypress(.yourkey)
        }
    }
    class EventHandler {
        auto, autoevents, EnableEvents
        Dim Events()
        class widget {
            \\ in classes we can define variables without value
```

```
            ref$  \\ weak reference
            id, caption$, Enabled=True
            module linkme {
                  read .ref$
            }
            module widget {  \\ this is used as construction function, but is a
module
                  read .id
                  read .caption$
            }
            function Callback {
                  read some$
                  push .ref$ : read &ref  \\ this is the link
                  =ref.query(.id, some$, &this)
            }
      }
      function button {    \\ this is the button construction
        .auto++
        read txt$
        makeme =.widget(.auto,txt$)
        makeme.linkme &this
        =makeme   \\ we make a group a floating group to export
      }
      function EventList {
            handle=.autoevents  \\ we execute some commands..
```

```
                .autoevents++
                dim .Events(.autoevents)
                read .Events(handle)   \\ .. and then we can read function
parameter
                =handle
            }
            module register {
                read handler, ref$
                .Events(handler).register  ref$
            }
            function observe {
            if not .EnableEvents then =true : exit
                print "observe for events"
                for i=0 to .autoevents-1 {
                    for .Events(i) {
                        if  .checkevent() then {
                            for j=0 to .auto-1 {
                                for this {
                \\ we use a for this {} to make temporary variable
                \\ because we can't relink a link.  so we make a fresh here
                \\ the link statement to this:    push  .ref$(j) : read &Link
                                    link weak .ref$(j) to &link
                                    call link.callback(.name$)
                                }
                            }
```

```
                }
            }
        }
    }
     function query {
          read id, what$, &nicesmallgroup
          print under "Inside EventHandler";
          print under
          print  part "I get a query from a widget object ";id
          print
          print part "Say:";what$
          print
          for nicesmallgroup {
               Print part "I get the caption:";.caption$
               print
               =0
          }
     }
}
EventHandler=EventHandler()  \\ create handler
B1=EventHandler.button("Button1")  \\ create subjects
B2=EventHandler.button("Button2")
B3=EventHandler.button("Button3")
```

```
            for EventHandler {
                PressKeyOne=.Eventlist(keylistener(49))
                .Register  PressKeyOne, &B1  \\register event to subject
                .Register  PressKeyOne, &B2
                .Register .Eventlist(keylistener(50)) , &B1
                .Register .Eventlist(Mouselistener(1)) , &B3
                .EnableEvents=True
            }
            every 150 {
                if EventHandler.observe() then exit
                if mouse=2 then exit
            }
        }
        OOP
```

Example Rational Numbers

This is an example of a class Rational which make objects (groups) as rational numbers with operators. **(this program included in Info.gsb)**

```
        Module RationalNumbers {
            Class Rational {
                numerator as decimal, denominator as decimal
                gcd=lambda->0
                lcm=lambda->0
                operator "+" {
                    Read l
```

```
            denom=.lcm(l.denominator, .denominator)

.numerator<=denom/l.denominator*l.numerator+denom/.denominator*.numerat
or

            if .numerator==0 then denom=1

            .denominator<=denom

        }

        Operator Unary {

             .numerator-!

        }

        Operator "-" {

            Read l

            Call Operator "+", -l

        }

        Operator high "*" {

            Read l

            g1=.gcd(l.numerator,.denominator)

            g2=.gcd(.numerator, l.denominator)

            Push l.numerator/g1*.numerator/g2

            Push l.denominator/g2*.denominator/g1

            Read .denominator, .numerator


        }

        Function Inverse {

            if .numerator==0 then Error "Division by zero"
```

```
        ret=This
        sign=Sgn(ret.numerator) : if sign<0 then ret.numerator-!
        swap ret.numerator, ret.denominator
        if sign<0 then ret.numerator-!
        =ret
    }
    Operator high"/" {
        Read l
        call operator "*", l.inverse()
    }
    Function Power {
        Read pow as long
        ret=This
        ret.numerator<=.numerator^pow
        ret.denominator<=.denominator^pow
        =ret
    }
    Operator "=" {
        Read l
        Def boolean T=True, F=False
        if Abs(Sgn(l.numerator))+Abs(Sgn(.numerator))=0 then Push T: exit
        if Sgn(l.numerator) <>Sgn(.numerator) then Push F : exit
        pcomp=l/this
        PUSH pcomp.numerator=1 and pcomp.denominator=1
```

```
}
Operator  ">" {
      Read l
      Def boolean F
      if Abs(Sgn(l.numerator))+Abs(Sgn(.numerator))=0 then Push F: exit
      if Sgn(l.numerator)=0 then {
            PUSH .numerator>0
      } Else {
            pcomp=this/l
            PUSH pcomp.real>1
      }
}
Operator ">=" {
      Read l
      if Sgn(l.numerator)=0 then {
            PUSH .numerator>=0
      } Else {
            pcomp=this/l
            PUSH pcomp.real>=1
      }
}
Operator "<" {
      Read l
      Def boolean F
```

```
        if Abs(Sgn(l.numerator))+Abs(Sgn(.numerator))=0 then Push F: exit
        if Sgn(l.numerator)=0 then {
            PUSH .numerator<0
        } Else {
            pcomp=this/l
            PUSH pcomp.real<1
        }
    }
    Operator "<=" {
        Read l
        if Sgn(l.numerator)=0 then {
            PUSH .numerator<=0
        } Else {
            pcomp=this/l
            PUSH pcomp.real<=1
        }
    }
    Operator "<>" {
        Read l
        if Sgn(l.numerator)=0 then {
            PUSH .numerator<>0
        } Else {
            pcomp=this/l
            PUSH pcomp.real<>1
```

```
    }
}
Group Real {
    value {
        link parent numerator, denominator to n, d
        =n/d
    }
}
Group ToString$ {
   value {
        link parent numerator, denominator to n, d
        =Str$(n)+"/"+Str$(d,"")
    }
}
class:
Module Rational (.numerator, .denominator) {
    if .denominator=0 then Error "Zero denominator"
    sgn=Sgn(.numerator)*Sgn(.denominator)
    .denominator<=abs(.denominator)
    .numerator<=abs(.numerator)*sgn
    gcd1=lambda (a as decimal, b as decimal) -> {
        if a<b then swap a,b
        g=a mod b
        while g {
```

```
                    a=b:b=g: g=a mod b
                }
                    =abs(b)
            }
            gdcval=gcd1(abs(.numerator), .denominator)
            if gdcval<.denominator and gdcval<>0 then
                .denominator/=gdcval
                .numerator/=gdcval
            end if
            .gcd<=gcd1
            .lcm<=lambda gcd=gcd1 (a as decimal, b as decimal) -> {
                =a/gcd(a,b)*b
            }
        }
    }
    Print rational(-3,3)<>rational(-3,3)  ' false
    M=Rational(10, 150)
    N=Rational(2, 4)
    Print "M.real+N.real=";M.real+N.real
    Print "Z=M+N"
    Z=M+N
    Print 10/150@+2/4@
    Print "Z.real="; Z.real
    Print "(";M.numerator;"/"; M.denominator;") + (";N.numerator;"/";
N.denominator;") = (";Z.numerator;"/";Z.denominator;")"
```

```
Print M.tostring$+ " +"+N.tostring$+" ="+Z.tostring$
Print -10/150@+2/4@
Z=-M+N
Print "-"+M.tostring$+ " +"+N.tostring$+" ="+Z.tostring$
Print Z.numerator, Z.denominator, Z.numerator/Z.denominator
Print -10/150@+2/4@
Print Z.real
Z=M-N
Print Z.numerator, Z.denominator
Print 10/150@-2/4@
Print Z.real
Z=M*N
Print Z.numerator, Z.denominator
Print (10/150@)*(2/4@)
Print Z.real
Z=M/N
Print Z.numerator, Z.denominator
Print (10/150@)/(2/4@)
Print z.tostring$
Print Z.real
Print "Z power 2 = ";

Z=Z.Power(2)
Print Z.real
```

```
Print z.tostring$;" = ";eval(z.tostring$)
Print Z=Z
Print Z=N  ' false
Print Z=-Z  ' false
ZZ=-Z
Print ZZ=ZZ
Print -Z=-Z
Print Z.numerator, Z.denominator
Print Z.real, Z.tostring$
\\ Array of rational numbers
Dim K(100)=rational(1,1)
M=K(4)+K(3)
Print M.real
Print K(4).toString$

pk->(Z)
Print pk=>toString$+" +"
zzz=k(4)+pk
Print zzz.toString$+ " ="+K(4).toString$+" +"+pk=>toString$

zzz=Rational(10,1)+Rational(3,1)*Rational(2,1)
Print zzz.toString$, zzz.real=16
zzz=Rational(10,1)*Rational(3,1)+Rational(2,1)
Print zzz.toString$, zzz.real=32
```

```
}

RationalNumbers
```

Example Linked List

Another example using pointers to groups. We make a doubled linked list of objects, of type group. This program Linked_List1 is in info.gsb (a better version of Linked_List which also part of the same file - the modules file type of M2000). Now we have all the logic for a double linked list in one class, with private members. This class create the null object, has a node() function to create nodes. We can use own node class if we want to using different members for data but include the two pointers for the linked list.

```
\\ Second Version

\\ This example use pointers to groups.

\\ One class which make Node and Null class inside

\ \ Head and Tail are private


Module Checkit {
      Form 80, 50
      Class LList {
      Private:
            Group Head, Tail
            Class Null {}
      Public:
            Property Null {Value}
            Group GetHead {
                  Value {
                        link parent Head to Head
```

```
            ->Head
        }
    }
    Group GetTail {
        Value {
            link parent Tail to Tail
            ->Tail
        }
    }
    Function Node {
        Group Ret {
            group pred, succ
            dat=0
            Remove {
                Print "destroyed", .dat
            }
        }
        Ret.pred->.[Null]
        Ret.succ->.[Null]
        if match("N") Then Read Ret.dat
         ->(Ret)
    }
    Module PushTail(k as pointer) {
        if .Tail is .[Null] then {
```

```
            .Head<=k
            .Tail<=k
        } else {
            n=.Tail
            .Tail<=k
            k=>pred=n=>pred
            n=>pred=k
            k=>succ=n
        }
    }
    Function RemoveTail {
        n=.Tail
        if n is .Head then {
            .Head->.[Null]
            .Tail->.[Null]
        } Else {
            .Tail<=n=>succ
            .Tail=>pred=n=>pred
            n=>pred->.[Null]
        }
        for n, This {
            .succ->..[Null]
            .pred->..[Null]
        }
```

```
        =n
    }
    Module PushHead(k as pointer) {
        if .head is .[Null] then {
            .Head<=k
            .Tail<=k
        } else {
            n=.head
            .head<=k
            k=>succ=n=>succ
            n=>succ=k
            k=>pred=n
        }
    }
    Function RemoveHead {
        n=.Head
        if n is .Tail then {
            .Head->.[Null]
            .Tail->.[Null]
        } Else {
           .Head<=n=>pred
           .Head=>succ=n=>succ
            n=>succ->.[Null]
         }
```

```
    for n, This {
        .succ->..[Null]
        .pred->..[Null]
    }
    =n
}
Module RemoveNode(k as pointer) {
    pred=k=>pred
    succ=k=>succ
    if pred is succ then {
        if .head is k else Error "Can't remove this node"
        k=.RemoveHead()
        clear k
    } else {
        pred=>succ=succ
        succ=>pred=pred
    }
}
Module InsertAfter(k as pointer, n as pointer) {
    pred=k=>pred
    n=>pred=pred
    n=>succ=k
    pred=>succ=n
    k=>pred=n
```

```
        }
        Module DeleteAll {
            While not .IsEmpty(){
                For This {
                    A=.RemoveTail()
                    clear A
                }
            }
        }
        Remove {
            .DeleteAll
        }
        Function IsEmpty {
            = .Head is .[Null] or .tail is .[Null]
        }
    class:
        Module LList {
            .[Null]->.Null()
            .Head->.[Null]
            .Tail->.[Null]
        }
    }

    L=LList()
```

```
Null=L.Null
m->L.Node(100)
L.PushTail m
If not L.GetHead is Null then Print L.GetHead=>dat=100
for i=101 to 103 {
      m->L.Node(i)
      L.PushTail m
      Print "ok....", i
}
for i=104 to 106 {
      m->L.Node(i)
      L.PushHead m
      Print "ok....", i
}

Print "Use Head to display from last to first"
m=L.GetHead
do {
      Print m=>dat
      m=m=>pred
} Until m is Null
Print "ok, now find 3rd and remove it"
m1=L.GetHead
i=1
```

```
Index=3
While i<Index {
    if m1 is Null then exit
    m1=m1=>pred
    i++
}
If i<>Index then {
    Print "List has less than "; Index;" Items"
} Else {
    Print "First add one new node"
        newNode->L.Node(1000)
        L.InsertAfter m1, newNode
        L.RemoveNode m1
        clear m1  ' last time m1 used here
        newNode=Null
    Print "ok....."
}
Print "Use Tail to display from first to last"
m=L.GetTail
do {
    Print m=>dat
    m=m=>succ
} Until m is Null
```

```
If rnd<.5 Then
useother=True
While not L.IsEmpty(){
      For This {
            \\ we have to use a temporary variable name, here A
                  A=If(useother->L.RemoveTail(),L.RemoveHead())
                  ? A=>dat
                  useother~
                  \\ now we can try to perform removing
                  clear A
       }
}
Else
L.deleteAll
End if
Print "list is empty:"; If$(L.IsEmpty()->"Yes","No")
mm=0
for i=50 to 80 {
      m->L.Node(i)
      L.PushTail m
     mm++
}
m->Null
Print "Added ";mm; " nodes"
```

```
        Push Pointer((L))
    }
    Checkit
    Stack
    Print "Press a key for phase 2"
    Print "now Linked_list is a pointer to a group object"
    Push key$ : Drop
    Read Linked_List
    Null=Linked_List=>Null
    m=Linked_List=>GetHead
    While not m is Null
            Print m=>dat
            m=m=>pred
    End While
    List
    \\ we have to use Clear. Clear act only if the pointer is the last one.
    \\ Then Linlked_List Remove do a Clear for each node. If we didn't use any
    extenal pointer for these we get all destroyed.
    Clear Linked_list, Null, m
    Print " now all variables aren't pointers. They have an empty group"
    List
```

Inheritance

```
    \\ because deconstructor (Remove {}) is final
```

```
\\ and we want to apply a different deconstructor
\\ for all derived objects from GameObject
\\ intentionally we didn't apply the deconstructor in Class
\\ But see later how we can import this
class GameObject {
        \\ by default functions are virtual
        state=0
        Group PointerToGameObject->0&
        function update() {
                =2
        }
        function draw() {
                =3
        }
        function collide() {
                =4
        }
Class:
        Module GameObject (.state) {
                \\ this is the user constructor
                Print "You just create GameObject"
        }
}
\\ Visible, Solid, Movable compose GameObject
```

```
\\ Without using the constructor
\\ but their GameObject would be a different instance of class GameObject
class Visible as GameObject {
        \\ using final the function stay as is
        function final Draw() {
                =100
        }
}
class Solid as GameObject {
        function final collide() {
                =1000
        }
}
class Movable as GameObject  {
        function final update() {
                =10000
        }
}
\\ here we compose Player, Cloud, Building, Trap from other objects
class Player as Solid as Movable as Visible {
        Remove {
                Print "Player removed"
        }
class:
```

```
        Module Player (.PointerToGameObject) {}
}
Class Cloud as Movable as Visible {
        Remove {
                Print "Cloud removed"
        }
class:
        Module Cloud (.PointerToGameObject) {}
}
Class Building as Solid as Visible {
        Remove {
                Print "Building removed"
        }
class:
        Module Building (.PointerToGameObject) {}
}
Class Trap as Solid {
        Remove {
                Print "Trap removed"
        }
class:
        Module Trap (.PointerToGameObject) {}
}
Class TraitRemove {
```

```
        Remove {
            Print "GameObject Removed"
        }
    }
    \\ Game is a pointer to Group
    \\ Because we want to include Remove function
    \\ we can do that merging two objects
    \\ so Game is a GameObject but has a TraitRemove
    Game->(GameObject(1234) with TraitRemove())
    Print Game=>Update()=2, Game=>draw()=3, Game=>collide()=4
    \\ P, C, B, T are static objects here
    P=Player(Game)
    Print P.Update()=10000, P.draw()=100, P.collide()=1000
    C=Cloud(Game)
    Print C.Update()=10000, C.draw()=100, C.collide()=4
    B=Building(Game)
    Print B.Update()=2, B.draw()=100, B.collide()=1000
    T=Trap(Game)
    Print T.Update()=2, T.draw()=3, T.collide()=1000
    \\ All objects has the same pointer to Game
    Print P.PointerToGameObject=>state=1234
    Print C.PointerToGameObject=>state=1234
    Print B.PointerToGameObject=>state=1234
    Print T.PointerToGameObject=>state=1234
```

```
\\ When state of Game change, these objects can find it
Game=>state+=20000
Print P.PointerToGameObject=>state=21234
Print C.PointerToGameObject=>state=21234
Print B.PointerToGameObject=>state=21234
Print T.PointerToGameObject=>state=21234
\\ we use Parenthesis to get a pointer to a copy of P
\\ so we make a pointer to a new object with same state
\\ without parentesis we get a pointer to static group
PP->(P)
PC=Pointer((C))  ' this is the same as ->(C)
PB=Pointer((B))
PT=Pointer((T))
Print PP=>Update()=10000, PP=>draw()=100, PP=>collide()=1000
Print PC=>Update()=10000, PC=>draw()=100, PC=>collide()=4
Print PB=>Update()=2, PB=>draw()=100, PB=>collide()=1000
Print PT=>Update()=2, PT=>draw()=3, PT=>collide()=1000
Print PP=>PointerToGameObject=>state=21234
Print PC=>PointerToGameObject=>state=21234
Print PB=>PointerToGameObject=>state=21234
Print PT=>PointerToGameObject=>state=21234

\\ Remove function called for 5 pointers at the end of this module
\\ We can manule execute Remove for static groups if we want
```

```
\\ We have 4 static objects (in M2000 we say named groups)

Report "Removing using Clear Named Groups"

Clear P, C, B, T

Report "Automatic Removing Groups erasing Pointers when this module exit .
The last one is the GameObject"
```

Logging to File in Temporary Directory

Some time we want to export some values as the program run for review them later. We can do this by using the log files. We can also use the Test to run e module, so we can execute the module by a step by step manner. Using logging is a different approach, for finding errors, depending on values.

We can use text files for logging purposes. Logging is the procedure to save to file values or labels or both to indicate the state of program in various stages.

We can use three types of character encoding, the ANSI, the UTF-16LE, and the UTF-8. For simplicity we can pass labels in a string which are interpreted as variables. We can string pass expressions too. See the example bellow. For Utf-8 encoding we can read file using Document object.

```
Const programname$="MyProgram"

Text UTF-16 logging.txt {##STR$(TODAY+NOW,"YYYYMMDDHHNNSS")##

}

For var1=1 to 10

        Text UTF-16 logging.txt + {This is a line for LOG ##var1## for
##programname$##

        }

Next var1

\\ win temporary$+"logging.txt"  ' we can open the file in notepad

\\ or we can open using OPEN (not for UTF-8)

Open temporary$+"logging.txt" for wide input as #k

        Try {Seek #k, 3}  ' SKIP BOM
```

```
        While not EOF(#k)
            Line Input #k, aLine$
            Print aLine$
        End While
    Close #k

    \\ delete the log file
    Text logging.txt
```

BreakPoint

We can place statements to get breakpoints, an we can display a test form to see the executed code as executed.

```
    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  \\ wait for keypress
    }
    Test !
```

Final Examples

GAME 2048

This example has the color from the internal editor of M2000 Environment (convert to html when we place it to clipboard).

We use lambda functions to feed the Process() subroutine. We use an Inventory object to put the Empty Tiles and to pick one from there. Main loop loops while EmptyTiles has some empty tiles, so we can play.

The game has an array 4X4 in the display, but inside we have only one dimension array the Board(). We have four lambda objects, four functions which get a two dimension indexes, and return the actual one dimension for the board. Each function translate the 2D to 1D depends on the Direction as the name of it says, by using the appropriate formula.

An interest part of the code is the Gravity subroutine, which use a complex loop with three Continue statements. The important part of the sub is the assignment board(boardtile(k,i))=boardtile(j,i). Gravity finds the k for all tiles in the board always in the current direction which we choose as response of the each new tile.
Gravity plays two times. The first time to place all items without empty places. Next we make any doubling of tile so 2 2 2 2 change to 0 4 0 4. And then a Gravity again to get 0 0 4 4 where 0 is the empty tile. For each column, we check for empty tiles and we add to the EmptyTiles, appending the index of the empty tile. For each process we make a new EmptyTiles inventory, an empty, so for each column process we place only the tiles with 0, the empty tiles. When we fill all the board with tiles>0 the EmptyTiles will be empty so the game stopped.

```
Module Game2048 {
      \\ 10% 4 and 90% 2
      Def GetTlleNumber()=If(Random(1,10)=1->4, 2)
      \\ tile
      Def Tile$(x)=If$(x=0->"[    ]", format$("[{0::-4}]", x))
      \\ empty board
      BoardTileRight =lambda (x, y)->x+y*4
      BoardTileLeft=lambda (x, y)->3-x+y*4
      BoardTileUp=lambda (x, y)->x*4+y
      BoardTileDown=lambda (x, y)->(3-x)*4+y
      Dim Board(0 to 15)
      Inventory EmptyTiles
      \\ Score is a statement but we can use it as a variable too.
      Score=0
      \\ Win is also a statement but we can use it as a variable too.
      Win=False
```

```
ExitNow=False
BoardDirection=BoardtileRight
Process(BoardDirection)
\\ Split Rem lines to insert start condition to check valid moves
Rem : board(0)=2
Rem : board(1)=2, 2, 2 ' place to (1), (2), (3)
    While len(EmptyTiles) {
    NewTile()
    DrawBoard()
    Action=False
    do {
        a$=key$
        if len(a$)=2 then {
            Action=true
            Select case Asc(mid$(a$,2))
            Case 72
            BoardDirection=BoardTileUp
            Case 75
            BoardDirection=BoardTileRight
            Case 77
            BoardDirection=BoardTileLeft
            Case 80
            BoardDirection=BoardTileDown
            Case 79 ' End key
                ExitNow=True
            Else
            Action=false
            end select
        }
    } until Action
    If ExitNow then exit
    Process(BoardDirection)
}
If Win then {
    Print "You Win"
} Else {
    Print "You Loose"
}
Refresh 30
End
Sub Process(Boardtile)
Inventory EmptyTiles ' clear inventory
local where, i, j, k
For i=0 to 3
    Gravity()
    k=boardtile(0,i)
    For j=1 to 3
        where=boardtile(j,i)
```

```
            if Board(where)<>0 then {
                if board(k)=board(where) then {
                    board(k)*=2 : score+=board(where): board(where)=0
                    if board(k)=2048 Then Win=True : ExitNow=true
                }
            }
            k=where
        Next j
        Gravity()
        For j=0 to 3
            where=boardtile(j,i)
            if board(where)=0 then Append EmptyTiles, where
        Next j
    Next i
    End Sub
    Sub NewTile()
        local m=EmptyTiles(Random(0, len(EmptyTiles)-1)!)
        Board(m)=GetTlleNumber()
        Delete EmptyTiles, m
    End Sub
    Sub DrawBoard()
        Refresh 2000
        Cls
        Cursor 0, 10
        Local Doc$, line$
        Document Doc$
        Doc$=Format$("Game 2048 Score {0}", score)
        \\ Using Report 2 we use rendering as text, with center justify
        Report 2, Doc$
        Doc$={
        }
        Local i, j
        For i=0 to 3
            line$=""
            For j=0 to 3
                line$+=Tile$(Board(BoardTileRight(j, i)))
            Next j
            Print Over $(2), Line$
            Print
            Doc$=Line$+{
            }
        Next i
        Report 2, "Next:Use Arrows | Exit: Press End"
        Refresh
        ClipBoard Doc$
    End Sub
    Sub Gravity()
        k=-1
```

```
        for j=0 to 3 {
            where=boardtile(j,i)
            if k=-1 then if board(where)=0 then k=j : continue
            if board(where)=0 then continue
            if k=-1 then continue
            board(boardtile(k,i))=board(where)
            board(where)=0
            k++
        }
    End Sub
}
Game2048
```

Keyboard 005 - Piano

Playing music using keyboard. Press space bar for more volume. With a good keyboard you can play 6 note at once. You can find this program in info.gsb

```
Refresh 5000
Thread.Plan Sequential
Play 0 \\ clear music threads

dim note1$(10,3), note2$(10,3)
FillArray()

Refresh 5000
Move 0,0
Pen 14
Gradient 5,5
Cursor 0,0
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 --


}
Global kb$=" ", dur=300, org=5, f=0, vol$="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=scale.y/5
kk1=scale.y/4
gram=scale.y/20
DisplayNoRefresh(mm, kk, mm2, kk1)
refresh 5000
Scroll Split Height/2+1
Cursor 0, Height/2+1
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, dur, 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 "ok"
Threads
Thread L Interval 6000
Main.Task 10 {
    Display(mm, kk, mm2, kk1)
    Refresh 1000
    If KeyPress(asc("1")) Then Exit
    If KeyPress(asc("3")) Then org<=14 : dur<=100
    If KeyPress(asc("4")) Then org<=5 : dur<=300
    If KeyPress(asc("5")) Then org<=65 : dur<=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$ <=if$(dur=300->"V100", "V110")
    Print "--------"
}
Threads Erase
Print "End"
Scroll Split 0

Exit
Sub DisplayNoRefresh(p0,y0, p1, y1)
    Clavie(p0, y0, p1, y1, 0, ¬e1$())
    Clavie(p0-p1/2,y0,p1,y1*2/3, -1, ¬e2$())
End Sub
Sub Display(p0,y0, p1, y1)
    Clavie(p0, y0, p1, y1, 0, ¬e1$())
    Clavie(p0-p1/2,y0,p1,y1*2/3, -1, ¬e2$())
    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$, no \\ feed a row
            } else {
                 note2$(n-1,0):= n$, k$, no
            }
        }
}
End Sub
```

GUI Example using 3D rotating Graphics

This example use a build in MATH object which have many methods, which use multiple values. We use Buffer for values.

```
Set Fast !
smooth on
Rem :
Thread.plan sequential
\\ we can set Thread.plan before any thread created
Rem : Thread.plan concurrent
\\ concurrent execute one by one statements
\\ except those in { }
\\ also from Revision 52 Part {} as variable works fine with it
local counter1, counter2
Title "", 0 ' 0  to hide console
Set FAST !
\\ by api
Structure VecType {
        x As Double
        y As Double
        z As Double
}
\\ Program
Structure Variables {
    vRot1 As VecType
    vRot2 As VecType
    vRot3 As VecType
    vRot4 As VecType
    vBase As VecType
    vAxis As VecType
}
Buffer Clear Var As Variables
\\ utility function
VecAdr=Lambda Var (a$) -> {
    =Var(0,a$)
```

```
}
VecOff=Lambda Var, VecType (a$, b$) -> {
    =Var(0, a$, VecType(b$)!)
}
Class cLine {
    X1, Y1, X2, Y2, color
    Module Render {
        ' read Number from stack
        z=Round(Number,4)
        z1=Round(Number, 4)
        If z>=z1 Then {
            Move .X1, .Y1
            Width 3 {Draw to .X2, .Y2, .color}
            Circle Fill #aa33cc, z/40+200
        } Else {
            Move .X2, .Y2
            Circle Fill #aa33cc, z/40+200
            Width 3 {Draw to .X1, .Y1, .color}
        }
    }
Class:
    Module cLine (.color){
        If Match("NNNN") Then Read .X1, .Y1, .X2, .Y2
    }
}

\\ find address
vBase=VecAdr("vBase")
vBase.x=VecOff("vBase","x")
vBase.y=VecOff("vBase","y")
vBase.z=VecOff("vBase","z")
vRot1=VecAdr("vRot1")
vRot1.x=VecOff("vRot1","x")
vRot1.y=VecOff("vRot1","y")
vRot1.z=VecOff("vRot1","z")
vRot2=VecAdr("vRot2")
vRot2.x=VecOff("vRot2","x")
vRot2.y=VecOff("vRot2","y")
vRot2.z=VecOff("vRot2","z")
vRot3=VecAdr("vRot3")
vRot3.x=VecOff("vRot3","x")
vRot3.y=VecOff("vRot3","y")
vRot3.z=VecOff("vRot3","z")
vRot4=VecAdr("vRot4")
vRot4.x=VecOff("vRot4","x")
```

```
vRot4.y=VecOff("vRot4","y")
vRot4.z=VecOff("vRot4","z")
vAxis=VecAdr("vAxis")
Refresh 100
Declare Alfa Form
With Alfa, "Title", "Demo1", "UseIcon", True, "UseReverse", True ' icon now is in the left
side
Method Alfa,"MakeStandardInfo", 1 ' 1 for English
Def info$, AskRet
Function Alfa.InfoClick {
      Read New X
      If X=0 then
      after 100 {
                  Info$={
                        This is an example
                        of  using MATH object
                        (included in M2000 Interpreter)
                        }
                  AskRet=Ask(info$,"About Demo1","","")
      }
      end if
}
\\ a string to hold static background
screen$=""
disp=False
Inventory Depth
aLine=Each(Depth)
once=False
Function DepthSort() {
                  Inventory Queue Depth ' clear Depth, Then make keys As numbers
                  Append Depth, Eval(Var, vRot1.z As double):=1, Eval(Var, vRot2.z As
double):=2, Eval(Var, vRot3.z As double):=3
                  Sort Depth As number
}
Thread {
      part {
                  Method Math, "RotVectMult", 4, vRot1, vAxis, vRot1, dAngle
                  Push Eval(Var, vBase.y As double), Eval(Var, vBase.x As double)
                  \\ x is in top, y is after x
                  Over 2, 2 \\  copy two times from second, so double two top
                  Push Eval(Var, vRot4.x As double)+Number : Over 1, 2 \\ copy 2 times top only
                        Read Line1.X1, Line2.X1, Line3.X1
```

```
                Push Eval(Var, vRot4.y As double)+Number : Over 1, 2
                Read Line1.Y1, Line2.Y1, Line3.Y1
                Over 2, 4 \\ now original 2 values copied 4 times
                Line1.X2 = Eval(Var, vRot1.x As double)+Number
                Line1.Y2 = Eval(Var, vRot1.y As double)+Number
                Line2.X2 = Eval(Var, vRot2.x As double)+Number
                Line2.Y2 = Eval(Var, vRot2.y As double)+Number
                Line3.X2 = Eval(Var, vRot3.x As double)+Number
                Line3.Y2 = Eval(Var, vRot3.y As double)+Number
                call local depthsort()
                counter1++
                if random(20)>1 then exit
                Method Math, "Vector", vAxis,.4 -.8*rnd, 1.6, .2-.3*rnd
                Method Math, "UnitVect", vAxis
    } As once
} As Compute
dim All(1 to 3)
Layer Alfa {
    Window 12, 10000, 8000;
    Form 40, 20
    Line1=cline(#0000FF, scale.x/2, scale.y/2, scale.x/2, scale.y/2-2220 )
    Line2=cline(#FF0000, scale.x/2, scale.y/2, scale.x/2-2340, scale.y/2-60 )
    Line3=cline(#00FF00, scale.x/2, scale.y/2, scale.x/2-780, scale.y/2-1200 )
    All(1)->Line1
    All(2)->line2
    All(3)->line3
    Declare Math Math
    Method Math, "Vector", vBase,scale.x/2-1500, scale.y/2+1500, 1500 '  -1000
    Method Math, "Vector", vRot1, Line1.X2, Line1.Y2, -1000
    Method Math, "Vector", vRot2, Line2.X2, Line2.Y2, -1200
    Method Math, "Vector", vRot3, Line3.X2, Line3.Y2, 1700
    Method Math, "Vector", vRot4, Line1.X1, Line1.Y1, 0
    Method Math, "VecDiffMult", 4, vRot1, vBase, vRot1
    Inventory Depth=Eval(Var, vRot1.z As double):=1, Eval(Var, vRot2.z As double):=2,
Eval(Var, vRot3.z As double):=3
    Sort Depth As number
    Method Math, "Vector", vAxis, -.8, 1.6, .3
    Method Math, "UnitVect", vAxis
    Rad2Deg =Lambda pidivby180=pi/180 (RadAngle)->RadAngle / pidivby180
    dAngle =5
    Pen 0
    Cls 7
```

```
    Gradient 11, 13
    Move 0,0
    Cursor 0, Height-1
    Cls 7, Height-1
    Copy scale.x, scale.y to screen$
    Cursor 0,Height
    together=False
    ok1=true
    Thread {
        ok1~
    } as mm interval 1000/25
    tm=0
    Thread {
        static internal=1
        Move 0,0
        Copy 0,0 use screen$
        Part {
            tm=timecount
        } as ok1
        Print @(3,3); format$("{0:2:-5}",tm)
            Cursor 0,height-1


        Part {
                Part {
                 aLine=Each(Depth)
                 push Eval(Var, vRot4.z As double) : Over 1, 2
                 while aline {
                            pp=all(eval(aLine))
                            pp=>render Val(eval$(Depth, aLine^))
                 }
                 counter2++
                 } As disp
             Print Part $(5,Width/4), counter1, counter2,internal, $(7),Str$(Now ,
"hh:mm:ss" )
            Refresh 5000' execute together
        }  As together
        internal++
        Profiler
    } As PlayThis
}
\\ we set variables outside threads
State=False
```

```
blinking=False
Thread {
      If state Then disp~
} As blink
Thread blink Hold
ExitNow=False
Function Alfa.Unload {
     ExitNow=True
     threads erase
}
Function Alfa.Click {
               State~
               If State Then {
                     Thread compute Hold
                     Thread blink interval 1000/4
                     Thread blink Restart
                     blinking=true
               } Else {
                    disp=False
                    If blinking Then Thread blink Hold : blinking~
                    Thread compute Restart
               }
}
if IsWine then
     Thread PlayThis interval 1000/30
else
     Thread PlayThis interval 1000/120
end if
Thread compute interval 10
Wait 200
Method Alfa, "Show"
\\ Change Task.Main with Every
\\ Task.Main is a Thread
\\ ExitNow needed If we have Every in place of Task.Main
Task.Main 50 {
\\      If counter1>1000 Then Threads Erase : Exit
     If ExitNow Then Exit
}
Threads Erase
Wait 100 \ some delay here
Method Alfa, "CloseNow"
Declare Alfa Nothing
```

```
Declare Math Nothing
Print counter1, counter2
Title "Demo Complete"
Set Fast ' return to normal
```

GUI Example C# Editor

A C# editor. We can run the cs programs from editor. We use the color syntax capability of EditBox. Gui have menu and editor. We can choose bright or dark theme for EditBox.

```
Clear \\ Clear all variables/objects
Flush \\ Empty the stack
first$={using System;
        using System.Drawing;
        using System.Windows.Forms;

        class CSharpPendulum
        {
            Form _form;
            Timer _timer;

            double _angle = Math.PI / 2,
                   _angleAccel,
                   _angleVelocity = 0,
                   _dt = 0.1;

            int _length = 50;

            [STAThread]
            static void Main()
            {
                var p = new CSharpPendulum();
            }

            public CSharpPendulum()
            {
                _form = new Form() { Text = "Pendulum -  Γιώργος", Width = 400, Height =
200 };
                _timer = new Timer() { Interval = 30 };

                _timer.Tick += delegate(object sender, EventArgs e)
```

```
                {
                    int anchorX = (_form.Width / 2) - 12,
                        anchorY = _form.Height / 4,
                        ballX = anchorX + (int)(Math.Sin(_angle) * _length),
                        ballY = anchorY + (int)(Math.Cos(_angle) * _length);

                    _angleAccel = -9.81 / _length * Math.Sin(_angle);
                    _angleVelocity += _angleAccel * _dt;
                    _angle += _angleVelocity * _dt;

                    Bitmap dblBuffer = new Bitmap(_form.Width, _form.Height);
                    Graphics g = Graphics.FromImage(dblBuffer);
                    Graphics f = Graphics.FromHwnd(_form.Handle);

                    g.DrawLine(Pens.Black, new Point(anchorX, anchorY), new Point(ballX,
ballY));
                    g.FillEllipse(Brushes.Black, anchorX - 3, anchorY - 4, 7, 7);
                    g.FillEllipse(Brushes.DarkGoldenrod, ballX - 7, ballY - 7, 14, 14);

                    f.Clear(Color.White);
                    f.DrawImage(dblBuffer, new Point(0, 0));
                };

                _timer.Start();
                Application.Run(_form);
            }
        }
        }
title$="C# Editor"
typ$="cs"
W$=""
path$=""
Dir User
lookpath$=dir$
Declare NotePad Form
Declare Pad EditBox Form NotePad
Declare Inform1 Button Form NotePad
With Inform1, "Caption" as Informe$
Method Inform1, "Colors", 15, #FFA000
With Inform1, "Locked", True
Declare File1 Combobox Form NotePad
Declare Edit1 Combobox Form NotePad
```

```
Declare Run1 Combobox Form NotePad
Declare Help1 Combobox Form NotePad
With File1,"label","File", "listtext" As list$, "list" As list$() '
With Edit1,"label","Edit", "Mark", Color(255,100,0)
With Run1,"label","Run", "Mark", Color(255,100,0)
With Help1,"label","Help", "Mark", Color(255,100,0)


With NotePad, "Title" As Caption$, "Visible" As Visible, "TitleHeight" As tHeight, "Sizable",
True
With NotePad, "UseIcon", True, "UseReverse", True
With Pad, "Text" As Pad.text$, "ShowAlways", True,"NoColor", True,"SelLength" as
SelLength
With Pad, "ColorSet", 1, "linespace", 30, "NoCenterLineEdit", True ' we can tongle value
of  NoCenterLineEdit  using shift F9
Method Pad, "UserColorSet", false, , -#ff0000
Def TitleStr$(a$)=Ucase$(Left$(a$,1))+Mid$(a$,2)
Filename$=Dir$+"pendulum.cs"


Caption$=TitleStr$(File.Name$(Filename$)) +" - C# Editor"
Title "",0
Method NotePad,"MakeStandardInfo", 1
Method NotePad,"move", 2000, 4000, 8000, 4000
Layer NotePad {Cls #FFA000}
With File1,"MenuStyle", True, "MenuWidth", 3000
With Edit1,"MenuStyle", True, "MenuWidth", 3000,"menuEnabled" as Enabled()
With Run1,"MenuStyle", True, "MenuWidth", 3000,"menuEnabled" as Comp.Enabled()
With Help1,"MenuStyle", True, "MenuWidth", 3000
With File1, "MenuEnabled" As mEnable()
For This {
    mi$="MenuItem"  \\ is a temporary variable only for For This Block
    With File1, "MenuGroup","This"
    Method File1, mi$,"Open",True
    Method File1, mi$,"Save",True
    Method File1, mi$,""    \\  only  a line here
    Method File1,"MenuRadio","txt files",True,False
    Method File1,"MenuRadio","cs files",True,True

    Method File1, mi$,""
    Method File1, mi$,"Close",True
    Method File1, mi$,""
    Method File1, mi$,"Quit",True
```

```
    With Edit1, "MenuGroup","This"
    Method Edit1, mi$,"Cut",False
    Method Edit1, mi$,"Copy",False
    Method Edit1, mi$,"Paste",True
    Method Edit1, mi$,""
    Method Edit1, mi$,"Less Indent",True
    Method Edit1, mi$,"More Indent",True

    With Run1, "MenuGroup","This"
    Method Run1, mi$,"Compile test",true
    Method Run1, mi$,"Run test > out",False
    Method Run1, mi$,"Run test",False
    Method Run1, mi$,"Show Out.Txt",False
    Method Run1, mi$,""
    Method Run1, mi$,"Compile Final",true
    Method Run1, mi$,"Open Cmd.exe",true

    With Help1, "MenuGroup","This"
    Method Help1, mi$, "About", True
    Method Help1, mi$,""
    Method Help1, mi$,"Bright Themel",true
    Method Help1, mi$,"Dark Theme",true
}

if exist(Filename$) then
    Document BackUp$
    Load.Doc BackUp$, filename$
else
    Document BackUp$=first$
end if
Pad.text$=BackUp$
Function NotePad.Unload {
    Read New &Ok
    After 30 {Call local File1.DblClick(8) }
    Ok=True


}
Function Notepad.Resize {
    Layer NotePad { Cls Color(255, 160, 0) ,0}
```

```
    With NotePad, "Width" As NP.Width, "Height" As NP.Height, "TitleHeight" As tHeight
    tHeight1=theight*2
    Method File1,"move", twipsX*2, tHeight, twipsX*80, tHeight
    Method Edit1,"move", twipsX*2+twipsX*80, tHeight, twipsX*80, tHeight
    Method Run1,"move", twipsX*2+twipsX*160, tHeight, twipsX*80, tHeight
    Method Help1,"move", twipsX*2+twipsX*240, tHeight, twipsX*160, tHeight
    Method Inform1,"move", twipsX*2+twipsX*320, tHeight, twipsX*240, tHeight
    If NP.height>1000 Then {
        Method Pad,"move", twipsX*2, tHeight1, NP.Width-twipsX*5, NP.Height-tHeight1-
twipsx*3
        With Pad, "NoWrap" As NoWrap
        If Not NoWrap Then Method Pad,"Resize"
    }
}
Function Edit1.OpenMenu {
        Local X
        X=SelLength>0
        Enabled(0)=X
        Enabled(1)=X
}


Function Edit1.DblClick {
    Read Local Edit1index
    Select Case Edit1index
    Case 0
        {
        Method Pad,"mn1sub"
        Method Pad,"Resize"
        }
    Case 1
        Method Pad,"mn2sub"
    Case 2
        {
            Method Pad, "mn3sub"
            Method Pad,"GetFocus"
            Method Pad,"Resize"
        }
     Case 4
       {
            Method Pad,"PressKey", 9, 1
        }
     Case 5
```

```
        {
            Method Pad,"PressKey", 9, 0
        }
    End Select

}
Function Pad.Inform {
    Read New L, P
    Inform$=format$("{0}-{1}", L,P)
    Method Pad,"Show"
}
Function Pad.PopUp {
    Read Local X, Y
    Method Pad,"PopUpMenu", "",X , Y
}
function cs_choose{
     With Pad, "ColorCollection1", lcase$("|abstract|as|base|bool|break|byte|case|catch|
char|checked|class|const|continue|decimal|default|delegate|do|double|else|enum|event|
explicit|extern|false|finally|fixed|float|for|foreach|goto|if|implicit|in|int|interface|internal|is|
lock|long|namespace|new|null|object|operator|out|override|params|private|protected|public|
readonly|ref|return|sbyte|sealed|short|sizeof|stackalloc|static|string|struct|switch|this|throw|
True|try|typeof|uint|ulong|unchecked|unsafe|ushort|using|using|static|virtual|void|volatile|
while|add|alias|ascending|async|await|by|descending|dynamic|equals|from|get|global|
group|into|join|let|nameof|on|orderby|partial|remove|select|set|value|var|when|where|
yield|"),"NoColor", False, "UseCase", True,"ExtraFront", "_", "LineComment2", "",
"CommentSymbols","[/][/]", "ComSymbolsWidth", 2, "CommentLineLight", false,
"MultiLineComment1", "/*","MultiLineComment2", "*/"
    With Pad, "WordCharLeft"," ,+-=/*^()[];<>:?", "WordCharRight"," .,+-=/*^()
[];<>?:","OtherSymbols","@","UseColon",";"
    With Pad,"StartSymbols", "+-&|*=><,.1234567890()'"+chr$(34), "StringSep2","'",
"EnablePairs", True, "linespace", 30
    With Pad, "SearchWords", True, "nowrap", True, "AllowInsertBrackets", True,
"SpaceIndent", 2, "HighlightParagraph", True
    Method Pad, "ReColor"
    Method Pad, "Show"
}

Function Dark(){  '  13619071
    With Pad, "ColorSet", 0, "linespace", 60, "SelectionColor", -#FF0000,
"SelectionTextColor", -color(14), "HighLightColor", -Color(60,60,128)
     Method Pad, "Colors", -color(50,50,50),-color(14),,,-color(0)
```

```
      Method Pad, "Show"
}
Function Bright(){
      With Pad, "ColorSet", 1, "HighLightColor", -Color(225,255,255)
      Method Pad, "UserColorSet",0,,,,,,,,-#FF0000
      With Pad, "ColorSet", -1
      Method Pad, "ColorsReset"
      Method Pad, "ResetSelColors"
      Method Pad, "Show"
}


Function Run1Status() {
      if file.type$(filename$)="cs" then
            Comp.Enabled(0)=True
            Comp.Enabled(1)=False
            Comp.Enabled(2)=False
            Comp.Enabled(5)=True
      else
            Comp.Enabled(0)=False
            Comp.Enabled(1)=False
            Comp.Enabled(2)=False
            Comp.Enabled(5)=False
      end if
}
Function File1.MenuChecked {
Read New RadioIndex \\ 3 or 4
If RadioIndex =3 Then {
      After 30 { Static a$="."+File.type$(Filename$), b$=File.name$(Filename$)
      Filename$=File.path$(Filename$)+Left$(b$,Len(b$)-Len(a$))+".txt"
      Caption$=TitleStr$(File.Name$(Filename$)) +" - M2000 Pad"
      typ$="txt"
      Call local Run1Status()
      With Pad, "ColorCollection1", "","NoColor", True, "UseCase", False
      Pad.text$=Pad.text$
      Method Pad, "Show"
      }
} Else.If RadioIndex =4 Then {
     After 30 { local a$="."+File.type$(Filename$), b$=File.name$(Filename$)
      Filename$=File.path$(Filename$)+Left$(b$,Len(b$)-Len(a$))+".cs"
      Caption$=TitleStr$(File.Name$(Filename$)) +" - M2000 Pad"
      typ$="cs"
      Call local cs_choose()
```

```
      Call local Run1Status()
      }
}


}
Function File1.DblClick {
\\ we use functions As subs here
\\ each function and each module have a name (define a namespace)
\\ but here real module name is the File1.DblClick() host module
\\ this is done because is a Gui Event Service Function
\\ and is Call With Local tag (Call Local)
      Read New File1index
      Local cont, cont2, f$, NL$={
      }
      File1index++
      \\ Because we want some jumps..we use  On Goto
      \\ on Goto need here a block
      {
      On File1index Goto Open1, Save1, ExitNow, ExitNow, ExitNow, ExitNow, Save2,
ExitNow, Unload
Exitnow:
      Exit
Open1:
      If Pad.text$<>BackUp$ Then {
            If Ask("Save Changes first?",title$, "Yes","No")=1 Then Goto Save1
      }
     Layer NotePad {
          Open.file filename$,,"Load Text ("+typ$+") File",typ$
     }

     Method Pad,"GetFocus"
     Read f$
     If f$<>"" Then {
          Filename$=f$
          If Exist(F$) Then {
          Clear BackUp$
          Call Local Run1Status()
          Load.Doc BackUp$, f$

          Caption$=TitleStr$(File.Name$(Filename$)) +" - C# Editor"
          lookpath$=file.path$(Filename$)
          \\ silent means here without refresh
```

```
        With Pad, "SelStartSilent", 0, "SellengthSilent",0
        Pad.text$=BackUp$
        } Else Pad.text$="": Clear BackUp$
        Method Pad, "ReColor"
    }
    Exit
Save1:
    Layer NotePad {
        Save.As Filename$,,"Save Text ("+typ$+") File",typ$
    }
    If Not cont2 Then Method Pad,"GetFocus"
    Read f$ ' from save.as we get a value
    If f$="" Then Exit
    If Lcase$(File.type$(f$))<>typ$ Then f$=f$+"."+typ$
    If Exist(f$) Then If Ask(NL$+"Overwrite"+NL$+f$,title$, "Yes","No")<>1 Then Exit
    Try ok {
      Clear BackUp$
      BackUp$=Pad.text$
      Save.Doc BackUp$, f$ ' by default 2 - Utf-8
      filename$=f$
      lookpath$=file.path$(Filename$)
      Caption$=TitleStr$(File.Name$(Filename$)) +" - C# Editor"
    }
   If ok Else Beep
   If Not cont Then Exit
Save2:
    cont=True
    If Pad.text$<>BackUp$ Then {
        If Ask("Save Changes?",title$, "Yes","No")=1 Then Goto Save1
    }
    Clear BackUp$
    Pad.text$=""
    If Cont2 Then {
        Method NotePad, "CloseNow"
    } Else {
        FileName$=Dir$+"Untitled."+typ$
        Caption$=TitleStr$(File.Name$(Filename$)) +" - C# Editor"
        Method Pad, "Resize"
    }
    Exit
Unload:
    Cont2=True : Goto Save2
```

```
    }
}
function gecsc_folder {
        dir path$(0x24)+"Microsoft.NET\Framework\"
        dir ? dir$, "Choose Framework"
        path$=dir$
        dir user
}

thread {
    informe$=if$((tick mod 4 +1) ->"|", "/","-","\")
    If not Comp.Enabled(3) then {
        if exist(dir$+"test.bat") then {
            Informe$="ready to compile"
            Comp.Enabled(3)=True
            Comp.Enabled(0)=True
            Method Pad,"Show"
        }
    } else.if exist(lookpath$+"test.exe") then {
        Comp.Enabled(1)=true
        Comp.Enabled(2)=true
        informe$="Compiled"
        thread this hold
    }

} as lookEXE interval 1000
thread lookEXE hold
Function Run1.DblClick {
    Read New Run1index
    select case Run1index
    case 0
    {
        Comp.Enabled(0)=False
        Comp.Enabled(1)=False
        Comp.Enabled(2)=False
        Comp.Enabled(3)=False
        Informe$=""
        Call local gecsc_folder()
        if path$="" then Beep:Comp.Enabled(0)=True :exit
        if not exist(path$+"csc.exe") then beep : Comp.Enabled(0)=true : exit
        try {
```

```
        if exist(dir$+"test.bat") then dos "del "+quote$(dir$+"test.bat");
        if exist(dir$+"test.exe") then dos "del "+quote$(dir$+"test.exe");
    }
    local aa$
    Document aa$=Pad.text$
    dir user
    Save.doc aa$, "test.cs", 2 ' for UTF-8  see help

    if instr(aa$,"System.Windows.Forms")>9 then w$="win" else w$=""
    clear aa$
    aa$={setlocal
            set PATH=}+Path$+{;
            set LIB=%WINDIR%\Microsoft.NET\Framework\;
            csc.exe  /target:}+w$+{exe /out:test.exe test.cs
            endlocal
            }
    Save.doc aa$, "test.bat", 3 ' Ansi for bat
    after 300 {
        dir user
        dos "cd "+dir$+" && test.bat  > out.txt";
        after 200 {

            Thread lookEXE interval 100
            Thread lookEXE Restart
        }
        'Method Pad,"GetFocus"
    }
   after 500 {Comp.Enabled(0)=true}
}
case 1
    {
        thread lookEXE Hold
        After 40 {
        if w$="" then
            dos "cd "+dir$+" && test.exe > out.txt";
            Comp.Enabled(3)=True
         else
            win dir$+"test.exe"
         end if
        }
    }
case 2
```

```
        {
            thread lookEXE Hold
            After 40 {
            if w$="" then
                dos "cd "+dir$+" && test.exe"
                Comp.Enabled(3)=False
             else
                win dir$+"test.exe"
             end if
            }
        }
    case 3
     {
        try {win "out.txt"
        Thread lookEXE hold
        }
        informe$=""
    }
    case 5
    {
        local aa$
        Document aa$
        try {
            Call local gecsc_folder()
            if path$="" then exit
            if not exist(Filename$) then
                Call local File1.DblClick(1)
            else
                Load.doc aa$, Filename$
                if aa$<>Pad.text$ then Call local File1.DblClick(1)
                if instr(aa$,"System.Windows.Forms")>9 then w$="win" else w$=""
                clear aa$
            end if
            aa$={setlocal
                set PATH=}+Path$+{;
                set LIB=%WINDIR%\Microsoft.NET\Framework\;
                csc.exe  /target:}+w$+{exe /out:}+quote$(File.Name.only$(Filename$)
+".exe")+" "+quote$(File.Name$(Filename$))+{
                endlocal
            }
            Save.doc aa$, "final.bat", 3
            After 40 {
```

```
                    dos "cd "+shortdir$(dir$)+" && final.bat";
                }
            }
        }
        case 6
        dos "cd "+quote$(dir$)+" && cls "
        end select
}
Function Help1.DblClick {
     rem Read New Help1index \\ we get the number without variable
        select case Number
        case 0
        {
            Local A, info$
             Info$={
                  This is an example
                  of an editor for c#
                  written for M2000 Environment
                  use F1 to change wrap
                  }
             A=Ask(info$,title$,"","")
             Method Pad, "GetFocus"
        }
        Case 2
        Call Local Bright()
        Case 3
        Call Local Dark()
        End Select


}
Function Notepad.InfoClick {
     Read New X
     If X=0 Then Call Local Help1.DblClick(0) ' 0 for first menu item
}
Call Local Notepad.Resize()
\\ open As modal
Title "", 0
Call local cs_choose()
Call Local Bright()
Method NotePad,"Show" , 1
Declare Pad Nothing
Declare NotePad Nothing
```

Example Chess for Two

A thousand line program (you can find it in info.gsb as chessgame)

```
\\ George Karras, 2019
\\ Chess Example (a big one)
\\ Example Using sprites
\\ Rev. 4
chessfont$="Arial Unicode MS"
Font chessfont$
if not Fontname$="Arial Unicode MS" then
    chessfont$="DejaVu Sans"
end if
Font "Verdana"
Thread.plan sequential
Set Fast
Hide
window 12, window
if random(1, 3)=1 then
window 12, scale.x*random(6,9)/10,scale.y*random(6,9)/10;
end if
form 48,34
def thismode
thismode=mode
global const NoSound = False
Module NothingToMove {
    Layer {Print $(4)," Nothing to move",$(0);}
}
Module Proper {
    Layer {Print $(4)," Wrong color",$(0);}
}
Module Beep {
    Layer {Print $(4)," Not Possible",$(0);}
    if NoSound Else Beep
}
back {
    \\ we use a switch to alter the return code in Input ! variant, when we press Enter key
    \\ normally "-inp" return in Field  read only variable 1 when we press enter or down
arrow
    \\ using "+inp" we can get 13 for enter and 1 for down arrow
    Cls 0,0
    font "Times"
```

```
Pen 15
Mode thismode*5
cursor 0,height div 2
Report 2, "Wait...."
refresh 10000
Mode thismode


set switches "+inp"
Fkey Clear
Escape off
Cls #FFA000,0
Pen 14
bold 1
mode thismode
Def White$="PNBRQK", Black$="pnbrqk", WhiteDisp$="♙♘♗♖♕♔"
Def BlackDisp$="♟♞♝♜♛♚", empty$="12345678", disp$
disp$=WhiteDisp$+BlackDisp$
Def White_♔_file, White_♔_rank, Black_♚_rank, Black_♚_file
Def boolean White_♔_no_roke, Black_♚_no_roke
Def boolean White_no_left_roke, Black_no_left_roke
Def boolean White_no_right_roke, Black_no_right_roke
Def Halfmove_clock, Fullmove_number, threat, Clip$
Dim emptydisp$(1 to 8), BoardSq(1 to 8, 1 to 8)=(,)
Def en_passant_rank=0, en_passant_file=0
for i=1 to 8 :emptydisp$(i)=string$(" ",i):next i
Def board$, status$, oldI, color1, color2, C=14
color1=Color(209, 139, 71)
color2=Color(255, 206,158)

dim line$()
Def flashtime=300
Dim PastGames$(1 to 200)
Def freeSlot=0, cur=0, ok=true, k$, condition$
Def double ip, jp, ip1, jp1, si, sj, getone as boolean
Def st, fig$, tr, mx, my, lx,ly, key=0, mmx,mmy, mmb
Def movelogic as boolean=false, mvx, mvy
sa=(,) : sb=each(sa)
Double
OldI=Italic
Italic 1
Def upperlimit
Cursor 0,0
```

```
    Pen 15 {Report 2, "Chess Game for two"}
    Italic OldI
    Normal
    Move ! ' Move graphic cursor to character cursor  - Cursor !  the other way
    upperlimit=pos.y*1.6
    move 0, upperlimit*6/8
    Fill scale.x,scale.y-upperlimit*6/8, 3,5,1
    Set Fast !
    \\ calc based to height
    HalfWidth=(scale.y*.65) div 16
    def downlimit=0, White as boolean=True, fw
    DrawEmptyBoard((scale.x/2-HalfWidth*8),upperlimit, HalfWidth, 15)
    fw=HalfWidth*2-60
    \\ hold
    \\ set new game
    \\ -1 for no FEN
    def NoFEN(aGame$)=len(aGame$)<>len(filter$(aGame$,"/"))+7
    Inventory OnBoard
    Const NewGame$="rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0
1"
    Def ThisGame$
    \\ CHECK IF A STRING IS IN STACK (SO THIS MAYBE A FEN NOTATION FOR A
STARTING POSITION)
    if match("S") then
        For this {
            \\block for temporary definitions
            Local row$(), i, nok, m
            Read ThisGame$
            row$()=piece$(ThisGame$, chr$(13)+chr$(10))
            \\ we drop lines by redim the array (dim also is a "dim preserve")
            while len(row$())>1
                if row$(len(row$())-1)="" then dim row$(len(row$())-1) else exit
            end while
            if len(row$())>0 then
                m=each(row$())
                while m
                    if NoFEN(array$(m)) Then nok=true: exit
                end while
            End if
            if nok then Push ThisGame$ : ThisGame$=NewGame$ : exit
            PastGames$()=row$()
            Dim PastGames$(1 to 200)
```

```
            freeSlot=len(row$())
            ThisGame$=PastGames$(freeSlot)
        }
    else
        ThisGame$=NewGame$
    End if

    SetBoard(ThisGame$)
    RedrawBoard(True)
    sx=scale.x
    sy=scale.y-downlimit
    move 0,downlimit
    fill sx,sy, 5
    Layer {
        font "Verdanal"
        mode thismode, sx,sy
        motion center
        motion, downlimit
        Cls 5,0
        Pen 14
    }
    Refresh 60
    flush
    move$=""
    refresh
    \\ ctrl+F1 help
    About ! "How to play", 14000,9000,{Give one or more moves in one input line.
        Each move has a letter a number a letter and  final a number
        so:
                e2e4   or  e2-e4  or e2..e4
                (symbols other than aebcdefgh and 123456789 are white space)
        move something from e2 to e4, but this
                e2e3e7e5
        give two moves (so we can paste a number of moves)
        before a move a new FEN string compiled and copied to clipboard.
        if nothing exist in e2 then we get a beep sound. If a move break a rule then no move
happen and we get a beep. If King have a threat then we have to do a proper move to
eliminate threat otherwise we get a beep.
        We can press enter without giving a move, so we asked for ending the game or not.
        If we press Y then a new input start to get a FEN board notation, so we can use
ctrl+V to paste the string and pressing enter we get the new boad.
        * Castling work automatic. So if rules are ok wen can give a e1c1 for queen side
```

```
castling for white king.
        * En passant works fine
        * A pawn at last rank turn to Queen
        You can call this module passing a FEN string as parameter.
        George Karras
        }
    Thread {
        if control$<>"MAIN" then continue
        mmb=mouse
        If mmb=0 then continue
        mmy=mouse.y : mmx=mouse.x
        if mmy>downlimit then
            if mmx<scale.x*.8 then
                if mmb=2 then Field New 99 : Input End
            else
                if mmb=1 then Input End
            end if
        else.if mmy<downlimit and mmy>upperlimit then
            Field New if(mmy>((downlimit+upperlimit)/2)->1,-1)
            Input End
        end if

    } as Handler interval 100
    Thread Handler Hold
    Thread {
        if control$<>"MAIN" then continue

        if getone then
        move lx, ly
        Refresh 100
        sprite sprite$
        mx=mouse.x : my=mouse.y :mmb=mouse
        sb=each(BoardSq())

        St=(,)
        While sb
        sa=array(sb)
        if sa#val(3)-twipsX<=mx and sa#val(5)+twipsX>=mx and sa#val(4)-twipsY<=my and
sa#val(6)+twipsY>=my then St=sa : exit
        End While
        if len(st)>0 then
            si=st#val(10) : sj=st#val(11)
```

```
        if mmb=0 then
            keyboard chr$(ip+96)+chr$(jp+48)+chr$(si+96)+chr$(sj+48)+chr$(13)
            getone=false: refresh 100 : mouse.icon show
        else
            lx=mx : ly=my
            move lx, ly
            if st#val$(8)=" " or (si=ip and sj=jp) then
                sprite fig$, tr
            else
                sprite fig$, tr,-10,,80
            end if
            refresh 100
        end if
    else.if mmb=0 then
        RedrawBoard(?)
        getone=false
        mouse.icon show
        Thread Sp restart
    end if
    if not getone then Thread this hold
  } as pSp interval 1000/30
  Thread pSp hold
  Thread {
    if control$<>"MAIN" then continue
    if mouse=0 and movelogic then movelogic=false
    if mouse.y<upperlimit then mouse.icon 15 else mouse.icon 1

    if not movelogic then if mouse.y<upperlimit and mouse=1 then movelogic=true:
mvx=mousea.x : mvy=mousea.y:continue
    if movelogic then if mouse=1 then motion motion.wx- mvx+mousea.x, motion.wy-
mvy+mousea.y : continue

    mx=mouse.x : my=mouse.y
    if mouse=1 and my>downlimit then input end : Thread this hold
    if mouse=1 and not getone then{
    move mx, my
    if point=0 or point=#FFFFFF else exit
    sb=each(OnBoard)

    St=(,)
    While sb
    sa=eval(sb)
```

```
        if sa#val(3)<mx and sa#val(5)>mx and sa#val(4)<my and sa#val(6)>my then St=sa :
exit
        End While
        if len(st)=0 then exit
        move st#val(3)+30, st#val(4)+30
        refresh 1000
        tr=point
        copy fw,fw to fig$
        Image fig$ to fig$,120,120
        fill fw,fw, tr
        move mx, my
        lx=mx:ly=my
        ip=st#val(10):jp=st#val(11)
        sprite fig$, tr
        mouse.icon hide
        getone=true
        Thread pSp restart
        Thread this hold
        }
    } as Sp interval 100
    mouse.icon show
    isok=true
    {

        {
            if white then
                CheckThreat(white, White_♚_file, White_♚_rank, &threat)
            else
                CheckThreat(white, Black_♚_file, Black_♚_rank, &threat)
            end if
            if isok then
                freeSlot++
                GetBoard(&Clip$)
                if freeSlot>Len(PastGames$()) then Dim PastGame$(1 to 2*Len(PastGames$
()))
                PastGames$(freeSlot)=Clip$
            end if
            \\\test
            condition$=if$(White->"White move", "Black move") + if$(threat->" (check)","")+ if$
(Halfmove_clock>50->"(You can draw)","")+":"
            Layer {
                Print Part $(4,5), right$(string$(chrcode$(8199), 3)+str$
```

```
(Fullmove_number,""),3)+".", $(7,12),condition$,
        }

        White~
        if empty then
        Layer {
           Pen 15 {Input "",move$;}
        }
        \\\test !
        Thread Sp hold
        Thread pSp Hold
        mouse.icon 1
        getone=false
        if move$="" then

           Layer {
              wait 100
              Refresh 60
              profiler
              Every 1000/60 {
                 if timecount>flashtime then
                    profiler
                    Cls
                    Pen C {Double : Report 2,"End this Game ?" : Normal}
                    C=20-C
                    Cursor width, Height
                    Move ! \\ copy character cursor to graphic cursor
                    Legend "Use Y or Left Mouse Click  to exit | Right Mouse Click or N to
continue", FontName$, Mode*.7, 0,1, 1,twipsX
                 end if
                 k$=""
                 if keypress(0x1B) then k$="Y":exit
                 if keypress(0x4E) then exit
                 if keypress(0x59) then k$="Y":exit
                 if keypress(1) then k$="Y": exit
                 if keypress(2) then exit

              }
           }
           while not inkey$ ="" {Wait 1} 'drop key any
           If k$="Y" else
              \\ if stack has something then RedrawBoard mey use it (because read for
```

```
optional variable)
                \\ we can be sure we set the optional value using ?
                White~ :RedrawBoard(?): Layer {Cls}
                getone=false
                Thread Sp restart
                flush ' make empty true (so stack is empty for sure)
                loop ' set a flag for restart at end bracket of current block
            end if
        else
            move$=lcase$(move$)
            while len(move$)>0
                select case left$(move$,1)
                case "1" to "8"
                    data asc(move$)-48
                case "a" to "h"
                    data asc(move$)-96
                end select
                insert 1,1 move$=""
            end while
        end if
        isok=false
        if not empty then
            if stack.size mod 4 = 0 then
                try ok {
                    MakeAmove(&isok)
                    if not isok then flush : White~: exit
                    if not empty then
                    white~: refresh : wait 500 : loop
                    end if
                }
                Layer {Print}
                getone=false
                if isok else RedrawBoard(?)

                if ok then loop : Thread Sp restart
            else

                flush : White~ : loop : Thread Sp restart' loop processed at the end of the
block, so only a flag raised here
            end if
        end if
    }
```

```
        Thread Sp Hold
        Thread pSp Hold
        cur=freeSlot
        Clip$=PastGames$(cur)
        Layer {
            if cur=0 then cur=1
            if cur>freeSlot then cur=freeSlot
            if Clip$<>PastGames$(cur) then
            Clip$=PastGames$(cur)
            if len(Clip$)=len(filter$(Clip$,"/"))+7 and trim$(Clip$)<>"" then Back
{SetBoard(Clip$) :RedrawBoard(?)}
            end if
            Refresh 60
            Cls
            Cursor width, Height
            Move !
            Legend "Right Mouse Click or Esc to Quit | Left Mouse Click right of the FEN to
continue play | About Ctrl+F1", FontName$, Mode*.7, 0,1, 1,twipsX
            Cursor 0,0
            Report 2, "Replay the Game,(arrows u/d) or Start a new one setting a new FEN"
            Print Part $(7,7), "Board FEN: "
            Field New 13
            Thread Handler Restart
            Pen 15 {Input ! Clip$, width-7 len=100}
            Thread Handler Hold
            Report Clip$
            refresh 60
            if field=13 then exit
            if field=-1then cur-- : loop
            if field=99 then Clip$="": exit
            if field=1 then cur++ : loop
        }
        if len(Clip$)<>len(filter$(Clip$,"/"))+7 or trim$(Clip$)="" then SaveGame():Layer {Cls} :
exit
        if Clip$<>PastGames$(cur) then
            SaveGame()
            PastGames$(1)=Clip$:cur=1
        end if
        freeSlot=cur
        SetBoard(Clip$)
        RedrawBoard(?)
        Layer {Cls}
```

```
        Thread Sp restart
        Loop
    }
    Cls 0,0
}
hide
threads erase
wait 200
about ""
if module(info) then keyboard "info"+chr$(13)
Flush
set switches "-inp"
escape off
Window 12,0
form
form ;
About ! ""
end
Sub SaveGame()
if freeSlot=1 then exit sub
Local Out$
Document Out$ ' upgrade to Document
layer {
    Cls
    if ask("Copy the game to clipboard?","Finish","*Yes","No")=1 then
        Report "Wait..."
        for i=1 to freeSlot
            Out$+=PastGames$(i)
            if i<freeSlot then
            Out$={
            }
            end if
        next
        Clipboard Out$
        Save.Doc Out$, "LastGame.chess"
        Report "Done..."
        wait 300
    end if
}
end sub
Sub SaveGame1()
local Out$, i
```

```
Document Out$ ' upgrate to document
layer {
}
End Sub
Sub DrawEmptyBoard(leftmargin, topmargin, squarewidth, labelcolor)
   Local a=true, z=bold : bold 0
   Local l=squarewidth, k=2*l, k1=k*.85, N1=6, N=6, M=4, B=k*8, B1
   Local d=0, im=0, jm=0
   Repeat
   N=N1
   N1+=.25
   Until K1<size.Y("A",chessfont$, N1)
   topmargin-=l
   leftmargin-=l
   move leftmargin+l,topmargin+l
   B1=(l div 300)*twipsX
   step -B1,-B1
   B+=B1*2
   Pen 0 {
   Width b1 div 2+1 {
      color color1 , 1{Polygon 0, B, 0, 0, B, -B, 0, 0, -B}
   }
   M=N*.65
   For i=1 to 9
      d=leftmargin
      move d, topmargin
      if i<9 then
         step 0, k
         Pen labelcolor{Legend str$(9-i,""), chessfont$, M,0,2}
         step 0, -k
         else
         N=M
      end if
      for j=1 to 8
         If i<9 then
            step l, l
            color color1,1 {fill k,k, if(a->color1,color2)}
            step -l, -l
            BoardSq(j, 9-i)=(N, pos.x, pos.y, pos.x-l+twipsX, pos.y-l+twipsX, pos.x+l-twipsX,
pos.y+l-twipsX, k-twipsX*2, " ", a, j,9-i)
            a~
         else
```

```
            step k, k
            pen labelcolor {
               Legend mid$("abcdefgh",j,1), chessfont$, N, 0, 2
            }
         end If
         d+=k
         move d, topmargin
      next
      a~ : topmargin+=k
   next
   }
   bold z
   downlimit=topmargin+500
end sub
Sub MakeAmove( &ok, i, j, i1, j1)
Local z=bold, p$, p1$ : bold 0
Local t,t1, N=BoardSq(1,1)#val(0), again as boolean, playroke as boolean
ok=false
Local rule=true, threat as boolean
refresh 10000
Pen 0 {
   again=false
   t=BoardSq(i,j)
   t1=BoardSq(i1, j1)
   p$=t#val$(8)
   p1$=t1#val$(8)
   if p$=" " then NothingToMove : exit
   if t is t1 then NothingToMove : exit
   if p1$<>" " then if p1$<"♔" and p$<"♔" then Proper : exit
   if p1$<>" " then if p1$>="♔" and p$>="♔" then Proper : exit
   \\ white change logic here
   if not white and instr(WhiteDisp$, p$)=0 then Proper : exit
   if white and instr(BlackDisp$, p$)=0 then Proper :exit
   select case p$
   case "♔"
   {
      If not White_♔_no_roke then
         CheckThreat(not white, White_♔_file, White_♔_rank, &threat)
         if not threat then
            if i1=3 and j1=1 then
               if p1$=" " and BoardSq(2,1)#val$(8)=" " and BoardSq(4,1)#val$(8)=" " and not
White_no_left_roke then
```

```
                CheckThreat(not white, 4, 1, &threat)
                if not threat then push 1, 4, 1, 1 : again=true : playroke=true
            end if
        else.if i1=7 and j1=1 then
            if p1$=" " and BoardSq(6,1)#val$(8)=" " and not White_no_right_roke then
                CheckThreat(not white, 6, 1, &threat)
                if not threat then push 1, 6, 1, 8 : again=true : playroke=true
            end if
        end if
    end if
end if
if not playroke Then
    if abs(i-i1)>1 then rule=false: exit
    if abs(j-j1)>1 then rule=false: exit
    White_♚_no_roke=true
    White_no_right_roke=True
    White_no_left_roke=true
end if
CheckKing()
if threat then
    rule=false
    if playroke then drop 4
else
    White_♚_rank=j1
    White_♚_file=i1
end if
}
case "♚"
{
    If not Black_♚_no_roke then
        CheckThreat(not white, Black_♚_file, Black_♚_rank, &threat)
        if not threat then
            if i1=3 and j1=8 then
                if p1$=" " and BoardSq(2,8)#val$(8)=" " and BoardSq(4,8)#val$(8)=" " and not
Black_no_left_roke then
                CheckThreat(not white, 4, 8, &threat)
                if not threat then push 8, 4, 8, 1 : again=true : playroke=true
            end if
        else.if i1=7 and j1=8 then
            if p1$=" " and BoardSq(6,8)#val$(8)=" " and not Black_no_right_roke then
                CheckThreat(not white, 6, 8, &threat)
                if not threat then push 8, 6, 8, 8 : again=true : playroke=true
```

```
            end if
          end if
        end if
      end if
      if not playroke Then
        if abs(i-i1)>1 then rule=false: exit
        if abs(j-j1)>1 then rule=false: exit
        Black_♚_no_roke=true
        Black_no_right_roke=true
        Black_no_left_roke=true
      end if
      CheckKing()
      if threat then
        rule=false
        if playroke then drop 4
      else
        Black_♚_rank=j1
        Black_♚_file=i1
      end if
      }
  case "♔","♚"
  {
      if i1<>i and j1<>j then
        if abs(i1-i)<>abs(j1-j) then rule=false: exit
        jm=0
        if abs(i1-i)>1 then
          jm=j+sgn(j1-j)
          for im=i+sgn(i1-i) to i1-sgn(i1-i)
          if BoardSq(im,jm)#val$(8)<>" "  then jm=-1: exit for
          jm+=sgn(j1-j)
        next
        end if
      else
      jm=0:im=0
      if abs(i1-i)>1 then
        for im=i+sgn(i1-i) to i1-sgn(i1-i)
          if BoardSq(im,j)#val$(8)<>" " then jm=-1: exit for
        next
      else.if abs(j1-j)>1 then
        for jm=j+sgn(j1-j) to j1-sgn(j1-j)
          if BoardSq(i,jm)#val$(8)<>" " then im=-1 :exit for
        next
```

```
        end if
        if im=-1 or jm=-1 then rule=false:exit
        end if
        CheckIt()
    }
    case "♖"
    {
        If playroke then Black_no_right_roke=true : Black_no_left_roke=true :
Black_♚_no_roke=True : exit
        if i1<>i and j1<>j then rule=false: exit
        jm=0:im=0
        if abs(i1-i)>1 then
            for im=i+sgn(i1-i) to i1-sgn(i1-i)
                if BoardSq(im,j)#val$(8)<>" " then jm=-1: exit for
            next
        else.if abs(j1-j)>1 then
            for jm=j+sgn(j1-j) to j1-sgn(j1-j)
                if BoardSq(i,jm)#val$(8)<>" " then im=-1 :exit for
            next
        end if
        if im=-1 or jm=-1 then rule=false:exit
        CheckIt()
        if not rule then exit
        if i=1 and j=8 then Black_no_left_roke=true
        if i=8 and j=8 then Black_no_right_roke=true
    }
    case "♖"
    {
        If playroke then White_no_right_roke=true : White_no_left_roke=true :
White_♔_no_roke=True : exit
        if i1<>i and j1<>j then rule=false: exit
        jm=0:im=0
        if abs(i1-i)>1 then
            for im=i+sgn(i1-i) to i1-sgn(i1-i)
                if BoardSq(im,j)#val$(8)<>" " then jm=-1: exit for
            next
        else.if abs(j1-j)>1 then
            for jm=j+sgn(j1-j) to j1-sgn(j1-j)
                if BoardSq(i,jm)#val$(8)<>" " then im=-1 :exit for
            next
        end if
        if im=-1 or jm=-1 then rule=false:exit
```

```
        CheckIt()
        if not rule then exit
        if i=1 and j=1 then White_no_left_roke=true
        if i=8 and j=1 then White_no_right_roke=true
    }
    case "♖","♜"
    {
        if i1=i or j1=j then rule=false: exit
        if abs(i1-i)<>abs(j1-j) then rule=false: exit
        jm=0
        if abs(i1-i)>1 then
            jm=j+sgn(j1-j)
            for im=i+sgn(i1-i) to i1-sgn(i1-i)
                if BoardSq(im,jm)#val$(8)<>" "  then jm=-1: exit for
                jm+=sgn(j1-j)
            next
        end if
        if jm=-1 then rule=false:exit
        Checkit()
         }
    case "♘","♞"
    {
        if abs(i1-i)=abs(j1-j) then rule=false: exit
        if abs(i1-i)=0 or abs(j1-j)=0 then rule=false: exit
        if abs(i1-i)+abs(j1-j)<>3 then rule=false: exit
        Checkit()
    }
    case " ♙ "
    {
        if j1<=j then rule=false:exit
        if i<>i1 then if i1<i-1 or i1>i+1 then rule=false: exit
        if j>2 then if j1>j+1 then rule=false:exit
        if j=2 and j1>j+2 then rule=false:exit
        if i=i1 and p1$<>" " then rule=false:exit
        if i<>i1 and p1$=" "and not (en_passant_file=i1 and en_passant_rank=j1-1) then
rule=false:exit
        if i<>i1 and j1>j+1 then rule=false exit
        if en_passant_file=i1 and en_passant_rank=j1-1 then
            Checkit2()
        else
            Checkit()
        End if
```

```
        en_passant_file=0
        en_passant_rank=0
        if j1=8 then p$="♕" else if j1=j+2 then en_passant_file=i1:en_passant_rank=j1
    }
    case "♟"
    {
        if j1>=j then rule=false:exit
        if i<>i1 then if i1<i-1 or i1>i+1 then rule=false: exit
        if j<7 then if j1<j-1 then rule=false:exit
        if j=7 and j1<j-2 then rule=false:exit
        if i=i1 and p1$<>" " then rule=false:exit
        if i<>i1 and p1$=" " and not (en_passant_file=i1 and en_passant_rank=j1+1) then
rule=false:exit
        if i<>i1 and j1<j-1 then rule=false:exit
        if en_passant_file=i1 and en_passant_rank=j1+1 then
            Checkit2()
        else
            Checkit()
        End if
        en_passant_file=0
        en_passant_rank=0
        if j1=1 then p$="♛" else if j1=j-2 then en_passant_file=i1:en_passant_rank=j1
    }
    end select
    If not rule then beep : exit
    move t#val(3), t#val(4)
    return t, 8:=" "
    delete OnBoard, i*9+j
    if t#val(9) then fill t#val(7), t#val(7),color1 else fill t#val(7), t#val(7), color2

    t=t1
    move t#val(3), t#val(4)
    fill t#val(7), t#val(7),15
    if p1$<>" " then
        Halfmove_clock=0
        Return OnBoard, i1*9+j1:=t
    else
        Halfmove_clock++
        Append OnBoard, i1*9+j1:=t
    end if
    return t, 8:=p$
    move t#val(1), t#val(2): Legend p$, chessfont$, N, 0, 2,0
```

```
    move t#val(3)+twipsx*2, t#val(4)+twipsy*2
    if t#val(9) then floodfill ,,color1 else floodfill , , color2
    if again then read i, j, i1, j1 : Restart
    if p$<>"♙" and p$<>"♟" then en_passant_file=0 : en_passant_rank=0
    if white then Fullmove_number++
    Ok=true
}
bold z
refresh 10000
end Sub


Sub RedrawBoard(NoRefresh as boolean=False)
Local z=bold, p$ : bold 0
Local t, N=BoardSq(1,1)#val(0)
If NoRefresh Else refresh 10000
Pen 0 {
    For j=1 to 8
        For i=1 to 8
            t=BoardSq(i,j)
            move t#val(3), t#val(4)
            fill t#val(7), t#val(7),15
            p$=t#val$(8)

            if p$=" " else move t#val(1), t#val(2): Legend p$, chessfont$, N, 0, 2,0
            move t#val(3)+twipsx*2, t#val(4)+twipsy*2
            if t#val(9) then floodfill ,,color1 else floodfill , , color2
        Next
    Next
}
bold z
If NoRefresh Else refresh 60
end Sub
Sub GetBoard(&chessboard$)
chessboard$=""
local i, j,a$, spc, line$
for j=8 to 1
line$=""
for i=1 to 8
a$=BoardSq(i,j)#val$(8)
if a$<>" " then
if spc>0 then line$+=str$(spc,""): spc=0
```

```
line$+=mid$("PNBRQKpnbrqk", instr(" ♙♘♗♖♕♔♟♞♝♜♛♚", a$),1)
else
spc++
end if
next
if spc>0 then line$+=str$(spc,""): spc=0
if j>1 then chessboard$+=line$+"/" else chessboard$+=line$+" "
next
if white then chessboard$+="w " else chessboard$+="b "
if  White_♔_no_roke and Black_♚_no_roke then
   chessboard$+="-"
else
   if White_no_right_roke else chessboard$+="K"
   if White_no_left_roke else chessboard$+="Q"
   if Black_no_right_roke else chessboard$+="k"
   if Black_no_left_roke else chessboard$+="q"
end if
if not en_passant_file=0 then
   chessboard$+=" "+chr$(96+en_passant_file)+chr$(48+en_passant_rank)
else
   chessboard$+=" -"
end if
chessboard$+=str$(Halfmove_clock)
chessboard$+=str$(Fullmove_number)
End Sub
Sub SetBoard(chessboard$)
   Rem https://en.wikipedia.org/wiki/Forsyth–Edwards_Notation
   Clear OnBoard
   board$=leftpart$(chessboard$," ")
   if len(filter$(board$,"K"))<>len(board$)-1 Then Error "Problem with White King"
   if len(filter$(board$,"k"))<>len(board$)-1 Then Error "Problem with Black King"
   status$=ltrim$(rightpart$(chessboard$," "))
   white=left$(status$,1)="w"
   status$=ltrim$(mid$(status$,2))
   if left$(status$,1)="-" then
      White_no_right_roke=True
      White_no_left_roke=True
      White_♔_no_roke =True
      Black_no_right_roke=True
      Black_no_left_roke=True
      Black_♚_no_roke=True
```

```
    status$=ltrim$(mid$(status$,2))
else
    local L=len(status$)
    status$=filter$(status$,"K")
    White_no_right_roke= len(status$)=L
    L=len(status$) : status$=filter$(status$,"Q")
    White_no_left_roke= len(status$)=L
    White_♔_no_roke = White_no_right_roke and White_no_left_roke
    L=len(status$) : status$=filter$(status$,"k")
    Black_no_right_roke= len(status$)=L
    L=len(status$) : status$=filter$(status$,"q")
    Black_no_left_roke= len(status$)=L
    Black_♚_no_roke=Black_no_right_roke and Black_no_left_roke
    status$=ltrim$(status$)

End if
if left$(status$,1)="-" then
    en_passant_file=0
    en_passant_rank=0
    status$=mid$(status$,2)
else
    en_passant_file=Asc(left$(status$,1))-96
    en_passant_rank=Asc(Mid$(status$,2,1))-48
    if en_passant_rank=3 then en_passant_rank=4 else en_passant_rank=5
    status$=mid$(status$,3)
end if
Local m
Halfmove_clock=val(status$, "int", &m)
status$=mid$(status$,m)
Fullmove_number=max.data(val(status$, "int", &m), 1)
nl$={
}
for i=1 to 6
    board$=replace$(mid$(White$,i,1),mid$(WhiteDisp$,i,1), board$)
    board$=replace$(mid$(Black$,i,1),mid$(BlackDisp$,i,1), board$)
next
for j=1 to 8
    board$=replace$(str$(j,""),emptydisp$(j), board$)
next
line$()=piece$(board$,"/")
local t
dim line$(1 to 8)
```

```
   For j=1 to 8: For i=1 to 8
      t=BoardSq(i,j)
      return t, 8:=mid$(line$(9-j), i, 1)
      if t#val$(8)<>" " then Append OnBoard, i*9+j:=t
      if t#val$(8)="♔" then
         White_♔_rank=j
         White_♔_file=i
         if i<>5 and j<>1 then White_♔_no_roke=True : White_no_left_roke=true :
White_no_right_roke=true
      else.if t#val$(8)="♚" then
         Black_♚_rank=j
         Black_♚_file=i
         if i<>5 and j<>8 then Black_♚_no_roke=True: Black_no_left_roke=true :
Black_no_right_roke=true
      end if
   next : next
end Sub
Sub CheckIt()
      return t, 8:=" "
      return t1, 8:=p$
      if not white then
         CheckThreat(not white, White_♔_file, White_♔_rank, &threat)
      else
         CheckThreat(not white, Black_♚_file, Black_♚_rank, &threat)
      end if
      if threat then rule=false
      return t, 8:=p$
      return t1, 8:=p1$
end Sub
Sub CheckIt2()
      return t, 8:=" "
      return t1, 8:=p$
      local t2=BoardSq(en_passant_file, en_passant_rank)
      local p2$=t2#val$(8)
      return t2, 8:=" "
      if not white then
         CheckThreat(not white, White_♔_file, White_♔_rank, &threat)
      else
         CheckThreat(not white, Black_♚_file, Black_♚_rank, &threat)
      end if
      if threat then rule=false
      return t, 8:=p$
```

```
        return t1, 8:=p1$
        if threat then
           return t2, 6:=p2$
        else
           move t2#val(3), t2#val(4)
           if t2#val(9) then
              fill t2#val(7), t2#val(7),color1
           else
              fill t2#val(7), t2#val(7), color2
           end if
           Halfmove_clock=-1
        end if

end Sub
Sub CheckKing()
        return t, 8:=" "
        return t1, 8:=p$
        CheckThreat(not white, i1, j1,&threat)
        if threat then rule=false
        return t, 8:=p$
        return t1, 8:=p1$
end Sub
Sub CheckThreat(White as boolean, c, c1, &Yes)
local i=1, i1=8, j=1, j1=8, k=c, k1=c1
if white then
   local oKin$="♔",hor$="♖♕", dia$="♗♕", Kni$="♘"
else
   local oKin$="♔",hor$="♖♕", dia$="♗♕", Kni$="♘"
end if

' WhiteDisp$="♙♘♗♖♕♔"
' BlackDisp$="♟♞♝♜♛♚"
Yes=false
if c1<7 and white then
   if c>1 then
      if BoardSq(c-1, c1+1)#val$(8)="♟" then Yes=True :exit sub
   end if
   if c<8 then
      if BoardSq(c+1, c1+1)#val$(8)="♟" then Yes=True :exit sub
   end If
else.if c1>1 and not white then
   if c>1 then
```

```
      if BoardSq(c-1, c1-1)#val$(8)=" ♙ " then Yes=True :exit sub
   end if
   if c<8 then
      if BoardSq(c+1, c1-1)#val$(8)=" ♙ " then Yes=True :exit sub
   end If
end if
for k=max.data(c-1, 1) to min.data(c+1, 8)
for k1=max.data(c1-1, 1) to min.data(c1+1, 8)
if k1=c1 and k=c else if BoardSq(k, k1)#val$(8) =oKin$ then Yes=True : Exit Sub
next
next
for k=c to i
   if Instr(hor$,BoardSq(k,c1)#val$(8))>0 then Yes=True :exit sub
   if k<>c then if Instr(disp$,BoardSq(k,c1)#val$(8))>0 then exit for
next
if c>1 and c1<8 then {
k1=c1+1
for k=c-1 to i \\  look up left
   if Instr(Dia$,BoardSq(k,k1)#val$(8))>0 then Yes=True :exit sub
   if k<>c then if Instr(disp$,BoardSq(k,k1)#val$(8))>0 then exit for
   K1++
   if k1>8 then exit for
next
}
for k=c to i1
   if Instr(hor$,BoardSq(k,c1)#val$(8))>0 then Yes=True :exit sub
   if k<>c then if Instr(disp$,BoardSq(k,c1)#val$(8))>0 then exit for
next
if c<i1 and c1>1 then {
k1=c1-1
for k=c+1 to i1 \\  look down right
   if Instr(Dia$,BoardSq(k,k1)#val$(8))>0 then Yes=True :exit sub
   if Instr(disp$,BoardSq(k,k1)#val$(8))>0 then exit for
   K1--
   if k1<1 then exit for
next
}
for k1=c1 to j
   if Instr(hor$,BoardSq(c,k1)#val$(8))>0 then Yes=True :exit sub
   if k1<>c1 then if Instr(disp$,BoardSq(c,k1)#val$(8))>0 then exit for
next
if c1>j and c>1 then {
```

```
k=c-1
for k1=c1-1 to j \\  look down left
    if Instr(Dia$,BoardSq(k,k1)#val$(8))>0 then Yes=True :exit sub
    if Instr(disp$,BoardSq(k,k1)#val$(8))>0 then exit for
    k--
    if k<1 then exit for
next
}
for k1=c1 to j1
    if Instr(hor$,BoardSq(c,k1)#val$(8))>0 then Yes=True :exit sub
    if k1<>c1 then if Instr(disp$,BoardSq(c,k1)#val$(8))>0 then exit for
next
if c1<j1 and c<8 then {
k=c+1
for k1=c1+1 to j1 \\ look up right
    if Instr(Dia$,BoardSq(k,k1)#val$(8))>0 then Yes=True :exit sub
    if Instr(disp$,BoardSq(k,k1)#val$(8))>0 then exit for
    k++
    if k>8 then exit for
next
}
rem test "here"
for k=max.data(c-2,i) to min.data(c+2, i1)
    for k1=max.data(c1-2,j) to min.data(c1+2, j1)
        if Abs(k-c)+Abs(k1-c1)=3 then if BoardSq(k,k1)#val$(8)=Kni$ then Yes=True :exit sub
    next
next
end Sub
```

Appendix I

For the examples in Handle of Png File

\\ test.png

Binary {

iVBORw0KGgoAAAANSUhEUgAAAJIAAACWCAYAAAA16tGYAAAABmJLR0QAAAAAAAD5
Q7t/AAAACXBIWXMAAAABAAAAAQBPJcTWAAAAB3RJTUUH0QkeER8wXgfQggAAIABJ
REFUeJztvXmQJNd13vudc+7RCwNGkLYqSqMdV
tuSA7UeRsiNka3lSPFmhzQpbtrw82RF+sgU/S7RlyQyJtiRSEi0uEkkQBoRlAAyx
DNaZaczSM93T+1aV9573x82supWVWVXd0wPMUDgdGbl2Vlbmr75z7rk37wVetVft
VXvVXrVX7VV71V61V+1Ve9VetVftVXvVXrVXbbTRK30Bl67dvU335lO6Pee5tO0v
MUhjg7LVezQmQN8coP0lAqkWnGH34ELvzzBIavZdnmB9E4NUCc4427YCXGx1IJS3
Vx1Xse3yAOubDKSR8AyDZhygRt2vceDQLe7DpQzVNwFIm4KHxtw/bHkcqwNi1PI4
x+Z2aUF1GYM0ANC48Iya122rOndsw9RoGDSj5sOWcakAdRmCNBZAw+bj7KvaX/V5
sdW5qTIEw8DZKlx4pYG6jECqBaj88EfBU17eilLV2WaVpwzPKKhGuUO8UkBdBiCN
BKgOnP7t6WQLzeZeiBwg5qvV+yvh/RVQ3QHvmyBKQCQAGAAg4gF4FtlgpnVrkyUR
mTfGnGfmc9776fm5uVnMnlmvuOjNwlO3rW592Dy3lxeoSxikTQE0uDy5ZzeS5DXw
/kY4dxO8PwDvJwEYqEo4mvIJPqwHhsActkMVLJQfpyAiZlZmUhGTGTGraSM9Z609
xsxTs7OzU+1Tx5ei6xkXkLqp6riq81bNc3t5gLpEQbq7zp0Md1MTu6+AtTej07kD
zt0M73cBmgKkgBJACiIPQAHVHBwFkebzAph8DgJxFyIwE4jAzMRMRMRERBAREpFO
kiTLjUbjhIgcPT83d2zl2LNz2Bw4m5kwZBn9yxcfpksMpLFUqH+5tasBooNQfxec
fx282weFROdwALSrOiAF4CN4fFCfHKhCiYh7ihUUiuJlZmZmImahABYzgZiFSUS0
0WjMNRqNF51zj0+98MIUFmY2MBwMP+a2caGK57ldPKAuIZAqVagepObOFoheD+fe
CedeC9U0P8YD6qEo1Kd4GLn7oni9AEnzfRopUqxOBCIFE4EFIBCxEBGRiBRABZ1i
4rCNhZnJGJM1m82TxpiHnnv22aOYPbOaX2cdMOMsjwNV1RwXC6ZLBKSREMWTQXPn
W5Flfx3O3ZAfowBc8Fi5AoUbHyaiYt11jy+A6s21Oy8g4kiVCqCYAWZCUKECJirA
YWaJQCrmIiLUbDanjZj7jz780KPYWM6ia/djzjcD1MsK0yUA0t1lclrlMkiM1q63
oNP5HmTZQRARVAs4NF+umqrB6sFVgAQAHsz9sRLQAyqCqICKmTnnJsBETGJEcpok
AkmIyIgYarWa5wB87el7730C6JSvrwxQ3fl4LhClZfSWtxemVxCkTaiQbV0H4INw
2Ru68Y9q/gDU5cF0GR6f76uDCyDuPZh+RcrjJlIwIQdHl0UCmBnBpxEzUw4OCYvk
cVKOkxQgmZw2l2FdJycnT8zNzf3Z8YfvP4keDDFYWnHtoxTrFVGnVwikMSFq7mzB
+3ei03kfvNuV784AjW6wxuBk3WC6u96vTPtue0PGzCu7d+/2ExMTXtX7TifL9u7d
i1ar1UiSJF1aWtLz58+7Rx897HR+HiDuqZJwKMmJAEQMZmJjmEFsrClcW1eFCjPG
GBCJEbG5ShlhMTaxa41G46GHHnrwQcycWUM1PHVTWalGQYWKObYDplcApFqI+oGy
revh/IfgszsRkoTRDVRXsZwNbJvY1bntLW/RQ4cOre7bt2+NmdtZ1pm3SdppJJx5
R7pnt7FEsI2UW+sbDonlyY228sJyxs8/f3zjmWee0ePHj3k3v6B9JbgCJBGiECOx
iIQASYRNgCcGyhZMMbNlYiNGijk3Go1jMzMzX3nxwftO5vehTklHwTQMKlQs53Zh
ML3MIA2FqFgXJBN3Ics+DJ9dlStM5Mb6VCifujA5QP3t73zP+m233bZ862v3ubvu
nNx53YF0YtcOSozwhgjNLK24hXOznYWT053ZZoPl7Gzbpgk3Z85nye6dZqfz2jo3
0+Enn5tfPnLk8c7TTx/trE9PU1+cBAKMYTABIswibIxhLqLt4NvYGGOMEQ4gGWvC
QUZELBMLMSXMbEVErLUr3vuvPfzlLz+Bzmrx/UaBNEypynEUUK9QuBCYXkaQRkIU
Amrb+j+QZe+DOtuLb/oAyqJtrgtVa1f2He9//+r73vMG81feMKE3HWxM7t1tJLHY
ay03ACQAWFUXVGl2fsE9e26uM3V+PpOVNY8z5zaS6ZmOTxOZXFx2remZjnviyWfX
n3zySXf6yJFODpCCmCAcYiViwAiDmRCSkmKMIWExYkSstYVbE2Gxxhobq ZNh5kSC
qzNElOTbKE3TR7722c9+BdlahI7cFP944m31hYrx4idgG2B6mUAaC6IGTPMTyDpv
BpTQd+O0CJx74BQgtXZl7//IR9rf89dv2Xjbm3a0brmhsb/ZoATATgATAPbn80b+
ubOqeGF1zR9+abr9/NmZzvqJUxs6v+gbL53eoPUNn87MLfsvffFL66ePPLoOcLix
RRY8QFOU3gAxDGGCMZQ7MTIiXEAkxhhrjBFjjBExtrtD0lyJCqAMESVElBKRNBqN
Y3/+pS/9EebOrqPPbSOGK4ZpmNvbTOyErcBkNvsPF2jVEEnjSqj/IWTtW4Mro6xX
CtOSC+u6teyj/+Af+e9+z3X8tjdNtq67Jt3fSGgPgEMAdiPAE9s6gFkAx9fW/bNT
p9uzz7y47l4607ZPPLuq588v+KmpKf3617++vZ2PoHmpTpXyKcT86hke
BFXyzjETWllZQ32eEGCISAgwzGwBOCKyzKwi4nJFUiLyoSDIyszWOXfw3e9974eO
HDny+XNPPT7XuwfgfF7cy3i9mBdT4c48enFmlWl+/KXs2gbyRP0QpZMH0Ol8HD67
OS/Gl1SHeuqTz+/6rg9kH/m+t61+73v2XHXzwcZeZlyDAM8O9JQntlkAp53TqcVl
f/bYSxunH31y5fyXvz69fn7uPJ8+fQaHv/DHy6gMUhnoPgABAIVI+C4igAhDBDBG
IEJiDBsjbMSKTaxYa8UYY6y1hRoZY62xxqSFiyvUiZkTIkqKORGZJElmn3rqqS+c
eOSBs+gpUwFVWZ1iFzhu7FSjTJtTpYusSCMg4vTqIRBFsZAWN839vZ/6Gfc3v3c/
vePNE9e2mnIDgKsRIBIM2gqAeQDHl1f86ROnNk4cfnJ14cv3nV98+hf98f/N4q
hpeCAPj8Jku0icICM3cVynvAGFFVRqBPqNcsxTKzEFEmIpaIDDN7Ecly9wZmdszs
c0VSAMrM8N5f+frXv/49AL544pEhZtFTJcKgOhWqFN/r2Ao4xlCmu2kzMF1EkEZA
ZJp74f0PBYi6JbIoH1T84gJYyTUHO7/0cz+89OH37bnyxusbVyG4sKsw6MIKmwbw
3Pq6f+nkdGf6yefWVr/ywOLCl7/ywOrhP/2jJVTHFsWvGxgo7TgNnooU6gEWwLle

tYmqwLnMq8KLGBVl770AEGZ2CC7OM7MzIR7yxOxy96bMnOSurXBv4aYRqapeeeed
d753ZWXlf80e/cZCdE89+oEpql1ioFBx/JhubnyYLhJIlbX4MVANePcJeJdD1BdU
lyHK7vquD3R+9JPfvvG979mzf6LFb0BQoR01H76C4MqOnj7bOfq/H1k698Bjyyu/
8+nPrU4d/os1jCf/sQtAb714NgbwDjAW8B7wjnN1AgBxzmVGDKuqISJWVeNVDTNn
eXxkRMQbEWeMcSKixhgIIp9DpQBQzAEQEe399m//9vf8z+npP8b8uTUEGDIMKk/x
3eJ7X7aqmEmjYyN4xoPpYgfb5S8Zfg2c/hB8dmvPnVVC5BBcWefjHzvg/+obJ65l
ppsBXIN6FVoCcKbT0eeefmH9md/7k/NTn/3jr7cPf+GPl1AdV1RBVKNI8c3NFLCE
LAOsAZwLrQJUGVDOGypxrkTBpTEbhIDbFSCJMV6E1RiDPPjW3M0pESmFuj7K5wCw
/0Mf+9i7P3PPPV+E3yggKtxcYXVKU7YyTHGwXQHUcLsIwfYIl2ZbH0Jn43vHgCj7
sV/6Rf37P7hfXnuocROCK9uD6lioKJG9dO589uLDR1ZO/Nb/nJr9b//2X51FP0Bx
oDoMoihGGige9x4UJcH/JEn4nkICMMKSpGSMoTRNTZqmbK01zWaTk2CcJInNlxNj
TGrEWDGSinDKzCmzJMycAkiZOdXQRCYFkACabGy0H/+D/++ehwDXyb9XPBUpgrrv
PSpNsKXgu+qhXIANhQiwE38VWfb9gEqUE6qE6BP/+Oc3fuTuq/mWGxpvAHATAkRc
8aGzAJ7tdPTw08+vP/Ol+xZPfey7f/y5Iw/ctwSgk0/xje70Pqs7uXy7R/8DqSoJ
RcG4c2Dj4b3CWg/vHVg8gupokiRKRL7ZbHpmRpIkXoxBI001SRK11qoxBjaxKiKw
1IKhSsYYiAgZY4pKYZLQwoCTJNl/xWuuWz7+1DfmK+5HVVhRVprNWP4/byTgsdqD
LqZrKyvR9XDZx0LGGj4EF93SWR9EP/ijPzn7Ex8/kNxyQ+N2AAdRHw8tATg5v+ie
+OpfLL3w6c9OLf/2v/vXM+jBUfXLLCf1quqlyvFRRdIuN78BmCah0wEaDYJzUO/Z
O+87nU5mrRXnnDHGOCIyRsQbI46ZnTFGrbUqImqthYiAiAr3Ruh3awQFeXWsiuTg
wYNvuflt37bw3P1fO4fwAyumbOAa+y3+PgPfpub4kbaNIFW2KyqsCec/CO/2hmqP
bpVHVOEa3NzH/++f3fjJT1yb3HJD41YA16MeogUAx8/OZoc/+6W5Z/7Jr3xm9sTD
96+iX2Xq1GUz+RVUzPstWwOSFmFjg9BqAZ02qTHkvWci8sYYb60VZs5EJDHGemty
NbJWkyQBM8OIgRjJC4IBJGYmVSWE6h3yXggAe++b73jHO970/PPPf1XPnap6k6Vs
dUAUQXYcz5aPHVmK2ybXNlAFUk46vhdZ+6+hI23NHyr1JRp/6Kd/zv3Ex6+hb7mx
Wbiz/TUfOAvguZNn2o9+5vPnT/79H/ipEwunX1oH0EYPoMKlFW7LVSyPAmxYjql/
ch1F0lA455E2tNFsapqmyiLaSFMlIt9qtdBqtYJLs0YbjQbSNIW11htjKHd1ZA2T
ESJjCMlgEWJjmJipaC2Xt+3l3ddff3329CMPncf4bmvL2etg1S5um0B6Y139WWiU
lmV3Q7WJgWqObrLR/Y0f+Dudn/mRW+n2W5p3AHgtgCtqPmwWwNFnXlx/7Ld+f+al
n/rkT7+l/jioE02xeyvWqwLPYQFo1XLV5JG1PdlW4JzXvEifJIlnYt9oNnRiooUk
STRJEkxMTFCuRmStpUaaIE2EEkvMArKGNEmErGGyhokZJAIWIaIQMkEY0mpN7l3o
uDPnXzqxgX5V2axtArBBkLbBtQ11aQzVD+eN0jwG6s9CTLTvtjvbP/aJN+GOW5vX
oj4mcgDOAHj6iWfWjvzzX3vh1G/96r+cxmAwXXZtw3JGw8AYz62VTYwCgFtdJZem
zjnHzVZTRES9V7XWarPZRJlkmqsRjBCsBRLLJlz8VSdlInRAJOohAlwqd7xX8Yq2
c56cY2FG+u53v/vO5+6//95wKwautSjmx1NxTDwxem6uSAuM7eK2O9juV6PWrrdg
be129P2i1ffXn8H95l99/9o737ZjP4CbAeyqOfdZVTxx/+Hlx//pv3v01B996j8V
QfUwkDaTfASGg4Sa9fi7A4sznvddS945arVamJyc1DxTzdZanZycUCLWRqOBRmrb
iSWlENKEg+J0X5VjBpQBYlWIqhpVSOaJVZW9E+Oc63hlSSwdePdHPnrtl3/3v52s
uK6qADpD8EZ137WcMR/13S8UpFo1IgAG7fb7od4EcHwcWHeD63/ws7+w+gPfe+Wu
RsK3Ariy5oNmVfWph46sPvb9P/RfXzjxyAMr6AHURnUuZSt5k1FKNEqVCAD82ZNI
rjnIi4uLMMboxMSE37Nnjxhj1BiLNE2QJhaTEwJjKGskBCJqm9A6hUSKh0is4SUo
8Z7YeyXjQapknNOO80ac08w58B133HL7l/9g4izaK+V8Vxmkuu9YVqW64LtSlbZT
kfrVqLnzLqyv3xA+tGhj3S2lKUltfufvfmRf88A+ewgBosqae1V95unnN574xX9z
+HQEURVA4+aARiXgNqtG8T0AALRPHfe7b34d79+/HxMTE8zM2L9/vzabTTRS0ckW
Y3JCSBXUSAlGIMZwhzkokDAyr8hUNQMoU9WOc2qch/NeO87BOI+Oc5DMkRGhPd/5
0Y9e+/nf+o2XStdqovXyJKV5FUx936vuHmwHSIMltubOJjqd9+abosRjf4D9tz/2
js6bb2/tRSih7a049xKAYy+caD9xz3+fPpO7szJEceksdmlVanSxlKh8HAHAvn37
/NmzZ+mGG290wqKtVksnW4I0IUy0DCWWqNVkSixDVTmxTERBZYnA3kOcV9Hw5ozJ
MsqAAE8n88Z5Feco40wzx5A7X3/DLZ/fs+8c5s7G7jq+tmIqWhAU8VMMUHxsGaT4
+/XdkwsotfW5tXhiJI03odP5G+iqUBeebtH7wz/8l/M/8ncOTu7ZJa8HcC0Gs9az
AI6dPtt55FP/49zUL//Uz57AIDzl+GhYaexiQxSbAsDs1HG98+3fiiv27MH1118H
KHBgf4qJlmDHpNBEk6nVEEpT5lYjBNpJwpRYIiPEzODEEouAjBAbQ8wEYQnvZBIR
K1SYIACxtTRhG1fMPX/ksfht3rLVJR1HKXLFuXqpgAtVpEE1au1qwLl3hY/W0kMM
TQrTaw9l3/kd1/uD1yZNVLclWgDw0sKSe+xPv7pw5uf+4c8cR69IX+fSYkXaLoiw
9QbxdwMAbr/99uKc9NY7J6GqtHeP9VnmHRFRYpG/YEnMBHJeRQiZD+4syzLNVFW8
UqfT8UYNZZnzHedYiLw1Ilm74zNmb7KM5M133nzoTz+3dxaLs+XCRHFPyq6uUKVy
8F2oVDnwrlSlLSrSMDVq3oRO57vyrmOion5Pjb7vb/3g4j/6u4d2T7b4DiI6VPEB
U6tr/ht/9OX5p/72B/+vAqLNqtEmKyWrAKqvWxptjwF4DA9/bR5v+bZ30V95wyQd
2GfBDOzZZdBlia+6wlAjFUoTpjQhSiyxMLG1xEZIiEDWElvDzExihCS8lsnCBCNC
jNB6XIhYAOI04Ym5ZT13+vln2qULKqtL+Z5U3INxYsSgSlWVoFuxHkzq74L3KQAX
KVJRJeL33XZn56PfcyMO7LO7iWhfxbkWADx/5OjqyX/za1+Zx+g8UTkmGhYfDZty
+5Ru7+vMn9L/+E9v0leOLCsA3HJDE9dfk2DXDuN37TC+1WDfaLBPE/KJJd9qsm81
2acJZZMtyZoNzhopu2ZKLk0oSxPOEkudNKW2Nd1tLrHUSRP2aUL0re9464HQ8g4m
nzhalnwqb4vzTlOJ5cGSXJ9diGsrnzj0T9Tp3AYAUC2CuiJO8gDc29/+9sVve+uO
PQg5o3KAvQDg+Eun2y99/isL8w/+yR8uYrCmfhRMo5QIFXNc1C5fQitHAPBfe3CJ

0oR0316jSyuqOybFZ5l6Y9l5p+y9d+opa2deAPKZ08w59d5TxqLOOnXWqG93VIl8
Jo4skToiVSL1RIRr9ydX7bjh5uNLLz6zln9/m19K2cUV90fQC8DjwDsOuospDsKR
r28FpNruiAlMN8L7K0Hk+jt1yNsetXZl3/edNzcO7LNXoDpndHZ11R/9wr0LU7/w
oz9zHPUAjSqdjevO8HJ3kfdtb92hAPTFqXVcdYUgsYKlIY5PLHw7Y3UZucypESva
6ag3htQ5dZlDBninDCdCLsBDXpw6Zs3a5B0RHBG7yQlMvu1tb9v5hRef6UT3wSM8
72JebC8AKrZVxUblqXTP7qYLdW39J/f6RgAmiE83k10kH/13fvCDK99+1+Skql6N
QTWaBvDkQ0dWnv+13/jaLIbniMp5omH1ZZcMRNHn6w3XNdR7VVWvu3cGwWg1yDdS
QmrhUkO+2WDfSNnlLgutJrskZZ+7tK47TCy5NCGXWMrCMtNdb7nlSvRcVey+BIPu
Ld5HwECVSjmnNODituraBn3lzr27sLFxc9jLHuqLurXCxbn3/bVvMQf2pVcS0YHS
+VYAnJk5n535wtcXFh/8kz8sGqXFbm3cWKguHkJv/vIDlLu3Ppdw7dWpAtC5BaeT
E1bXN7w2G+zFMLKO8+zJqyqMkOtk8C4jJagaUW13vAPgmOCIoAB7hSrgFVB/YF9y
ZXrtoeMbJ4/FKlOoTqFCxasxZXjGVaNuSY7fif2bqC2ubdRPMOZ6qO5EX2OxXs8g
V7/+jdmbbp9Img3agfAWbGxz6219/t6HlqZ/+Vf+w1kMNhmNl7cSTAOvIESjbM8u
UQCYaAYH0UjIN1LxiYVaQ16EvBF4FvgkISdCLrHs04ScteSt6SqTFyFvDetEkxtv
fvObJ9BTH1NajpUoXo6hGjvovhDX1i913t+E0K4Y/aW18P7+XXfdtf6aq20LQBP9
jfcXVHXq1Jn2uc/8yalIN/1SkScqB9Kj6s/GcGevOERVn68AsHMyvDHSbIS5tayp
FbWWvTWERko+TaCJgSYJqTGkiWVtpOyT3MVZQz5N2BtDPk3I3HHHHZPod2VVLqwM
ULkEVwcUovmWXFt1HYzLbsyXPIh8DIM3oLv1lte09+yygsE3QGbXNzD9wOHlM7/1
q/+yeJu0TonqwLkk3dkY1ndNEy3WtXVFmjDaHYVNyHMGOCZ1nuC8d8xwvq2OLDwT
lAjqHHnvoYlVr0rqwwuc/uaD6R4AZ9GDqCihZegF3sUPNHZzg/nB/tIbUCrBbU9d
WzrZhPNX52sKjUsK6lvX3+Bee6jhJ5pkEBSpsAUAJ469tHHqzx5YWEE/MGWY6gLq
MYr5lxxEdcqkzQYFmFJCp6Owljw5VfbkhUU32g5ICM71vq9zpETwqqzWeu+VVdVh
3167Azv2GCzNxUX8wrUVcVOhQhn646a6eAmoKLldqGvLK2mbV0J1B/qbLnSnQ4cO
dW68Lp0Q4R0IVSLIv9S51TU38+iTK7P3/PI/OYf+xmgxOFFOaqy4CLh0IRppjTTc
x9SG55YE96YigBFWYVVmUmvIG0PeWvZGSBNL3gipEagxrI2U0xtufZ1Fv0uLXZuU
lqtipFGxEpAfhPEC7tpO1AGRAwBs3u9iqY6H/MGDB/Wa/ckO5r5OHma9x/Gp052T
9z64uIoeRIpqNaoCqJziL+xSh6g+F5dbIwldNieWYhWAMaTGCBILiIR+UY2EmMoY
UsmBsoa0kTLdeuutKYZDVBczAcPjo77vYICZYBp6A6ryCVeHt0y7JbZeI8Sqbvfu
3SvW4Kp8XyGx55ZX3NzjT6/O/Ydf/0/n0K9CsQLFCbXy8pCk4yULUWxDf8BJkrs3
E9IHRIDzICZo5lid82oNNLxg4ouXflUVEIFaBd9w6PoGgGX0q0zhvlzFthioYWmA
vkz3Vl1bvzoR5Q31tdcjbNFLLABr7ZoxtKKqy/nFz3qvz06dbh/7ygOL8/7syViB
YpiGxUHFMjDo1i5Vq1OigfVSzglGAGGokfAdmQk2ycdF4aBUzARjWI0QjKiKQK87
kDTQXxKrKpWVE5GI1kcV/QnYnhgJcG5v6Pi8C08PiI273ZkzZ1ZW1/SsKi0hdDNz
ZnHZL78wtTF/4sRJoNp9bbbaI7LLQo1qjSjvHk4VNsRJ4XWAPHYwJtT1m+DWyJjQ
vaU1wc2JQEUYRghXXmEbGA5RvA+odm/l/gWAqhhps99z4GTe7wAR9w3BACiIdMee
K/zKysr83EI20850DsCzWaZPnzjVPvbnDywufvY371nEoPrUQVNXMrvkXVpZYVB1
H6Nt8Su2xgQPklgCC1SYFADI77xBGGoNKVFQpQARlAi0c1ISVAfPVbmiOOAeO9BG
7NqGB9wju6lpgjDY6TmgS0uLnTRNqdPR9XZb19bb/vn5RTc7O9fxC4sbLdQnFsvu
bUjJrLBLE6KSVcWZfftUlVS1UCYCAjAAQuaIQDZ3ccIoBiMgZiVrACOq+UsE2mxw
gkEwpOI6pOK4Ua6tu7yVYLtqPWS0CzWKkpHXXX+9u/22W3YZA9Pp6PTyips+Od3x
Tz63vvrl4UeLjhuqJmB0UH25xEajrO/hUKiYg6oWwkR5MxRYQ+ScAmAiUQgYXpWM
V3hD8F7BzGRE1TmPxJJFvTsrAHIYBIdL11YZGxVmoo2jHkYdRAQiATHCK2t9owvh
yiuvTK/ZZ/c4h/bSSrZybjZbn53PGi9OrevhP/2jZQyWxkZlq2vguXTVKHJro0pA
hSLFasQAQETCrPBeQWEkFLICdHwYjCAjgJmUWSHc697SGl+uvY/X43ZIVe4shql8
/fH2baki6X0gQYEuRB7MOjc3552HtDOl02c7a+sbvnPi1IZ56eT5Oje2CXguKzWq
Cw/qIOJ4GUGp8q4Ewz4PEHKojGHy3kGY4BhgCu6w6EEQg3FS2W3VNRcp0jrDYKLt
ea+tO9oQE9Rpd11VFxcX/dnZTvv02fayc2gDkLOzmc6en41b4SG/WGB0yaxi22Wh
RkC9i+hOeVeBwZ8FcIA8bgpDegHedzssYcmhggu/YA6dMxMxlIiImarUqOpayrHR
wLWVrrvPLrSpbZgTopJazkcvl4QsU15a8cZ7TYkoPb/QoaNHjy6h36WhtDyuII0O
VvUQ6hSpCxR67ocjlWIALELsvHZdlDBRPuoXiBxCHwIA+pOMVepTp0bDYBpwbVX5
gVE3ZHDO4vLfUzwRiDA3N6eLiyuytu5SZppcWMr43MyyTh1+aAM9FYrjImC0e7vk
i/tAbZF/2FS4sHJAzBFgRWKAjIBEiLqDEIRUADj0YglhAsJB312jAAAfIUlEQVRb
zsM+s3xdZaiAakb6GLjQPFLYQLSRjwFbjGtWuDvo/IKurK7YjbZOtBq8G0Dz3Llz
8TnKRfmqHNHlbKPg6T7AUkzUN0Uuj6tACxMRAOY8RgIA75FhuBpWFfEL45r/GfiO
21L7L8Lr3cHxip9HGI5KIYwzZ87o/KLjhaUM0zMdevLJJ9cBaH6dcfG+WI/n8THI
5UvWhiQgyw+xKiFYB5MgUicEaFg4nC+881b4AyVicLujndLnjiqJARhwg2Ub+J9t
CbatTZazThZGWASCmobOWQjO0czMjHvh2LRbWNrrT506uTr16KOd/uYxhYerzFaX
gYrs0nRrNcX98npdnNIHTx5kF9UWRbwked8AAIrAG0RZ7maYicmTFfiFpWyj9NlA
PzRlq9s39H+3BSQRmQ9jmRHgQWAClADvAWY8++ijfnJyorOwsJBNT09HgBRtug2i
PjR93edcDlahREBNLITq+q6yChkMAhUfH9YVTEwMgkBBCK5Npme6IMXXUszLMFcd
U/W/A7bVhGTfCa21s2HcDBBYASXKISJ4BZhpauolnZk93z5z5kwG5nz8DqDETU3V
R+W1XZJqFFmVGlWVnPrcm6pKFAMVrkxit1YsM5M4rxwKy0TIlEQYnU5IQqpXTJ1u
b6CXC4qvrcrq3Fj5/8v7t6epLRHNMJMHOKT2vQtxkveAEYL3NDM1RTI5SW5hQSGi
XffnKWTE+1Wpyi51clZlsMsA9UFUUpu+WElVi+BacqAMUFYyEoUyCwgOnKsRqUKf

OjrlomuK5+Xr3IwNHH8hwXbXvPfTIsaFikNmYimGM89TqwQQwa2soDeUJ1DkmYp+
FwFbLs1dNkpUgii2YTB13VcOTAEKATCREkkEVJ9LY8qHrNA870QgImUJa9mRI0e2
Gipsio1tAWI+fn7GiFkNQ3CEXAdLPkRn6ECaIcIhPcAUlewIxoRrMPnr6cWwDJeR
VUBUW8QvTVIU6fPlYrtBb5iuWIkGYihVIoJyHlQwUa8d0dqG3zj29NMvy/3cng+Z
Ob3eaDbOBRESCqOVE4kI59myUIIT4TxjBlibr+fbgTBADABwOjRnsS3XvE02AqJa
FUKvBFZWo64by48xBVDFerScl94CgBSC7iD/AE5Nd5axPLfV+7UpJdsOkBQAGWOO
Sz5sedGfOBGo6EKKuiCJwBgBs+RV1ARjwiDDQDFATBiWod8uKYCAsZWoNi+ESG1y
Neq+BRurEQZfcuxruE9MrCEm6imVB/3FY8sr0bUNy89tNlwYON7U7aj55/JFdY2Z
T4hIx3vPqtQNElWVmZm996RE5CmvRAyjLjKcAwCGZwILwTnkL3WFYRnaq0C9Mr1i
8dIYENXCU0yFEpVcVxwbldWor211gExZlVhIJVOwhpQwMo/OV+59tJMfX1aXuvtW
pUJas73vXNvmP8+fPz+VJMkygYhzXSpMRFiCvwvzMHpLGCxP8vjJGIK1jCSPkRr5
W0vNnZerElUBFUMksVtDf4dYXaAw2EGW7S0TA8REyghl4PCnkNU1v37fff+76scW
K1O5brPqmKr/HbDtAkk3Th5bajQaJzgMlkEhu8qcu7ti5HLJh5IiY8JAwmxtcHFG
QlBubVAkgNBsBeWa3E3Ys68iZqrsq+miWd4EdivubMAtlZSoAMXm+8rwDAClqibM
IQoil8dJCO4NR46uzvuzJ+PnW65mQsVy1Xp5e+X+C8kjlf0richRY8wb8rFcjXLo
SBOKUKpQZe+8kBCTMeK9h6pnrxAC2DnHAMR7n6kII8uAyUlgbY3gPPG+a8mfPVk8
rJfVrVW0KyrmwzLWA1noomQWwVI3FWCVgYrUSNl7FfVBzbyCvSoy793/+OPnVvPP
LVc7VdVplhWp3MR54HaU/2crINXldvT83NyxnTt2zK+vrxcjPXZ/Och9OjP3pB1k
AHEgkHeejbXksoxySXfOGE6ShDNrXZqmWF1dRev6m2jXrl10+sgjW7j0LXzZeoCK
+bBM9YAa5e6sCKiHwhTFR30wqSqrkgAkYIRYs4ibAFlY0sUvfOEL5ZwcSusxKHV1
m77mfwZuk6nZUWdFwF0pcyvHnp3b99a3v9hut/dEzR6KX6UpMriqKqoqKmpUtQPA
wCBjZnEihpl9fhO9qvokSbjT6fgrrriCFhcXaW1tjW55xzv5Xe96l//1fwZcjMrb
MWvvR6lRGYQuJHlsVCzLGAoVwUSWSMU5sHqVvHAjyK/5/sNLc6snXihqw8do01UL
FqL9A7conl9ljFRZU59l7jFjjMvjbCvBTB4nJcxsWcQYY4y1VplkkSRJTKPRMNZa
k6YpNxoNNsawtZZtknCj0eBWq8WTk5N89dVX4+DBgwCAJ554gj7xj3+eKh761r9U
fxwEDA+ka4Pp8hQGpekFz0SUoAeIRc+N1U3Fj1GAMDYJ8uoRlJTWnIdsbLjs1//L
V9fzx1MHyLD3BRFtK5+jCkAA2zsWiQLQqRdemLrxW77l5Orq6vWlYqxBGO2nW6wt
folE5PKhzTMAbIyRPG0gzFyMwOhFxDvneGJiAhOTk3r77bcrAHzyZ17svq5DRbXL
Zi9+uAIV83GVKM4JSeTKbO9e9EFUlNIsEdn8XlTFSwKQyYeUEOcDWD4E2EQEev5E
+/wjX/1q3LFo/Gr7sNfc42OHQVRpW42R6i9mYabN/LqHmPkQ8piglyNRCyUHpozy
G8vMGQAnIqa4gUzsjTVGVbWAqNFo+HysMzXG6DUHrmFAcfDahl51he1+wQKoLdi4
sdC4EMUxYnlKEKlQrk5doHLgyqqVnzO4Mu+VQPnnq7L3YFXQv//NpxaxtijO9ewC
9OCo64ijTpXGCc4VeYwUHzTMivgoXq888XPPPnv0pptvnt7Y2LiKiEQBYSLDzEUd
kmFmC8ARUcbMrvgFSi9G8mmaegAuSaw0my2fpqm2Wi1N0xSg0Pn5zknjb72pQZ/7
8rx+97t3l2O4cW07IKpybd3YqFCj0mQLFUIFUOh3aUaVoow3jHqI9xCvoUQ8dbp9
7vd/53cE9a91obSvPMxGlQsEhkMEoNdz17hWB1P/h8yeWUtuu+3+Tif7AAhGclXK
4QnggDJiEma2OUiOmZ2IeGb2IuKstdYY44txXxuNVJthrFhMtgTCRKpKp6bb/pYb
mnEGtzYDX7JRAXW8XAVQMe9TojiQrgiiu2oUQdQHVH5Mn+sLAKl4T+K9Fx9g4gBR
6ADiV37t6TmszMdubVhfm1Xw1EFU5dr67m032P4Kpofc9IFSUZ17605PPfjgo61W
85yEoNvkwBgRsSlixhpjrY0DbxO4MZKmqW00UpumqSRJYtI0NWma2omJSbNjomV2
7bDmmv2JMKvs3WP5mv2JTLSYn3lxvaKtztiTlOZDg2cMJgn7gMmDaxO5tgT9Ls3G
LiyCqDLoDuchAZQAYvUhyPYuxHbHT26c/9177hEMB6EAxY25bdgLq30cbDVGin/x
1dP6UkZEXxeR7wFgmDljZktEGREnIlwM0emZ2eUq5EXE56NNq7UWaZp6YwySJPHG
GExMCIQVjZSxZ5dxBKW1defnFsgf2Gfx4tQ63XBdo/aXE9m4xXtgeC1+uQK2rigf
g1IE1APLqIQSBiDxHuIcjA/b2HuIAtLpqPvF/+exWWRrFr3YKI6Fql6Fr9oXd242
CqS+e3yhxf8qpeqWDJ66997HJiYmTgiLERHLxBIUiZNcmawxxooxibXW2iSYzS1J
EmOtTXNFShqpFSLYyQljrSEjrIYIZseEyI4JkUbK0khZTp9tx8o0rOZ8XNUpK1BV
rGPyEpdV1a7qlKZEVRMACRGl0f/Frqy7rKrGe5gwpm2R0FRRrwx4dnlt/2NPr577
7G/8hkE/JFWdlQ1zdfE7hsVxA8+07vlvtfg/SpXCxWlb5+fn/2znzp1/02UuYWGX
q5DLQcqIyIqIM8YUUyIiGquSMQbNNEFiCa0GdxJLyFx4hXmiJS5zoLV175eWve7e
WbS2HLvZybh1ZwNqlCdYu7X1cWyECjWKYyL0QxTDVoCZn7MLVVAk7eWNvFNeXHGr
n/zx/74CdAyqx6iLwaiLm8rQobQ+VI2A/rqYcWxkbFSa3PGH7z+ZJMlDYoSJKMnj
pYSZExAl1trUGJOKSCoiDWGJFSkxwawYSiZaYphhiWATy8YashttTVRhncJutL1Z
WcuMCGR+0Q2pp6qMb4YpTl/skicFLREIsRLlUBSTydUpyeOkJNpfB1G31r93zuAy
A0QQ78DOqTgPUYX+v/91+uSLD96XVNz7MkAxMOV+y8tuLT521AQAKocwScMD7cIe
QxjkbeymE0X1CJ1eWDx38ODBA6q6h4jIGOOZWY0ImFnD6NOGmFltYIVEYK0lYwwZ
Y6iRJmikjCQhMsK0Y9JQI2VlhqSpUJoQ7ZgQShKmZkrc6XhuNoTX28ppUv3mKqoD
7LFcW6FC+XKhHn2KkoNjiagMULytDFESndOGuEgFobMP8R7Ga4AIXunhb6yc+eEf
/hceG2sogVE1WkJ5NIVOabl83DBXOADThWS2h1Hq8wcTPnTmzNrMzMxXrrrqqg97
75u5MsXBtoamSaK5WwsNKcUgSRIYE3okC51shtcKALA1IBmBAMi8B6tX3eiQbzXY
r647TDRFV1Y9Jlo8KuAu/0D6guvozVYBQHmStejQoapytVziKorwSUXpbCD5WJTQ
Coi89+wUAiWGQqDKs/Od5U/++KdXsDhrMeia6gb+ieexMg0LtMcKuLejPVKdq+v7
Ui8+eN9J59xXmZmKX2QOVEpEKTOneV1csdwQIykRJcJoCCNRRYOZUiakREgUIAIw
REjW1l3qnLcuU9PpaGKEzUZbLTPZtXUtF6nL6tEXNMfLRVwTQZBGripFr0if1E35

/5aPrYPIBni6EIITGPjwDpvzYOeQ/dQ/+8aZF/7i67Z0r6vcWXI7WXXKMVKV26sq
9vfZVkCqOmEZoEpyH/5fn38iTdNHiuRkEStFU0pEKRE1mDll4oYwGgpNmalhDKUi
SIhgidDwXq2qNrJMEyIk6slmTtPMadIO2+xG2xsAdm09xCpR5Wk3EViUsqK4pYhr
0hyeGApbAUWax0KVQEXbTen/up9bXEsVROqJfYBIVJX+8++ePfGZ//hreeP2WliG
TYrqAYPqnl15+8Czl0OYpONYGZOhN1aVcIr5sAkAGPB04rkXpm5+/euvUdUrmJlE
BHkrSoiI5nERQitKAyNEiWWyBmQNkzFE1rBYS2QNsTFMRkjyRplibOjphaAsTGwM
i3NgY0mch7GG+uKewi3FmejcZdmoCWxZzfpAzOGKg+2+Yn8Jxr59cWBdB5F6lSxA
xPcfXj5194d+OuegMiYqxz1Vw7ZmFfviGKlOmarq4wAAMj5EQCng3kpTC4LPcPzc
7PHXvvaW61R1Z96UGwU8OVBkjKiIsDGExDKMARlDbAyRMcxGiBNLHGJ0oiQHKzHE
xCRExERqGOEY7yFWSJyDGNNrUB83a41bKtRAFCcUuwChlyPqVnHExfy6KartHwqR
huoQeuq5tXPf8d6fX0N7hUoPvyqoLsPUjpbjAafLwXb5vHUg9bk3GZuhrr2xDqDy
en3d1OqyLjmcuu66614DYDK8sRQmYwyMMSRi2BgD4a4SsREmkQCGNZRzR2wNsYTX
6YQIEjqfgmEiISZmBCXyCiNCxnsY72FFek1aSpWqheLEsYsUicNy0b0cR0UKVAVQ
BCNZVVgiMuHdtGqInAc9f3x95t0f+NXl9tmTReIxftCxwhTrVcoUl9TK0NUBVAYJ
GAxpLgikYYpU3lbEYgVMWJ2Z7rRNOn311VdfDWBn4eYiqCAiJEJsDZMwsRFia4Mi
5f1IU2JZjCGi4q258BadUYVYQwTAMAcFMsLsFSICSxSK0/m+ck6pXKLqVqhWVbSW
wCnHQlWlM6val5cy5cC6584CRN/1/fcsnn/miaKEVo6L4qnKpcUAlQeYLhf9xwm4
t0ORBtzbuHHSQL3VwumT6+tsz11zzTVXE9Fk3qqSAkCSSw6xCFHh+clysTEQI8RG
wMwsRkK4RczCDDFCoqrWSGhoz6FPofAAicSrGhEummdY5gGACuWJXVBXiWoSilVT
Nz+UuzALBHBC1jokGwF0IfJeOQvbuhCd+cZhi3p4qmCqGli6E82r3GBdVUotQME+
pVsACSi5N6AapvJ6lZvD4pmTG3MbndOHDh3aT0RBmVhIWAIdTCwCNsJsDBHnqlMo
FEBshIQ4dEAVIGJhhHZQRDBQGI9edQMRDFNoYShCFnmFqyoIOVAGUe18aV6oUgFR
rDa1KqQKq0rFeU0Oj/EexiIMlvVaO+aIM37qubVz7/7Ary7nSjQOQHUurY3xlagM
Ul3xP1p/bCuKBJTc22aC7jKADIBWZ6bbT794/MXX3X77Icx0RXirMvQ5IcwsQjlQ
xBS6EZDEEjNC4MwMNkQsTMIINeIEMhzWjZIaYRJmDUE0kSHquhjjPYxw4a6Qxy3d
GvlufqekSnEla50aGe+LZc1LhEHpnIMAapyS8Q6iUIGSyZwKALr/8PLp73jPz623
z52SigdcBqcMSdW2snsrb6vLZA91acG2DNJAdQkwvir1l+KK49ZX/FOHHz9+8xvu
MEmS7Cei8NoAM1EePlHIbod+TphMCLY1dDfNMMxkmMkQwMaQUUCIyRDIgPJWhUoW
oUbd5PFNrhIw3kOYyfYgC8txM49SZroumO66sfD/MKpBEb33RpXEA9Z7SN7K0TgP
8V6509HsP//uuWN3f+ingc4qYbg7qyrGl91YlWuLVahOiUapUW6f2mqwXdhIVQIG
YYqXy//HUEdHHztyZu+1r1nevXv3PiJK85d1Q0/2If7h2L2FqApGBEwUmvIyh4fP
XOSGIN7neSFV44M6WO9hPNQwwTjfdTldVwQgj2nIqiIp1CpaTgIk3XUDBIC8pyL+
kgAPTOZgNLwBYtSpeA+T5a8UAeCZ853lH/3Fx1/6t7/wrwzQKSAqB8DDXNgohSqr
0GaL+7Eadd0acEEgjVSleFsddIVxd786OvbUN+bXyJy79tprJ5h5tzCEmfJOySnv
7aSIiUlBTxUSupaOi/TIG5qFuMhrUCgAxqsaEAyUjAIWSsb5HDqnFkRWc9dEkcsr
gMmnHDQUAbRVJeNcaEcEQDIfYqMiJlIPcV6NVxKf1+p7BR7+xsqZj338t+fu+8PP
JHkpu6wQwyCqU59Rbq7KrcVVK2OpEXBBIAEjVKluXg68q1whZqeOrz/+3IsnDx48
mDVbk3uJKA2FODCzCjOzQkWYhAkRNAVARZOMUMSn/CFqoTZRu6EQE+VweHTVKo9p
bJ536hbbVWEJasPbrmqLuCfL1OavCBklEu9QBPLdBmrOh6YgPoBIC0udtV/9jenj
P/zJf65zLz5TNJUdVuUxDKIyUB0MQlOVyBzHrdWqEbC9IMVW59Kq9qPiWABgrC75
px958Pxix5+97vpDrcTSTiLkuezQfRcRmLpKFBQqV6BCiSR/iAZElvI3MAiwPgTa
Vr2Gd8TyEpXzsApY5wIk0G7ex4bSHaxzAUDv1ChgnCfjcxAVsN6F8zsP4xWceTXe
g33Yzxttr48+tXrmo//nb89+7lOfStBeiytN43lVFcY4bq0MWhnEqs8au7Y/2LaB
1HVvQHXcU6k2pf1V2/v+9/xLx9cf/tp9J1tXHVjcu/eKnUZoggkGoUsXIeKuEgFd
uML+bmOxvOcOhaU8+ahKRj0MiIzXkBsqYqQ8z1OUvIz3sN7DepD1qkZzcHxwieKd
FmAar+E1Ic3nmYOoV+NcaLZz4lR7/id+6ZGTP/sP/4XOnTxWNLcpu7FhAfaoHFHV
9jpVGgei2PL1/hdCqh7gJu3uMgx1pbO6BmRF++pynVb5lZwwb+607/vwh19zx+2H
Xjs5lXsSS2oMuzRhl1jyaUIuseStZZdYUmvIJTYMY24NqQjUCKm1DGugYYACQLrd
WoacAee96YuEfjGFaSAHhhxaIPSWRgT2Go5Rr5KPii3eK6l6OjmdnfvXv/7s3Kfv
ucfk9WVVgW3xcIvlONiuK96XIYrhKeeQ4mqUOjc3Kk7CRQAJKME0rLg/rDXiWM1c
UWSL9+xrfOf733/Vm++86dDunXJlYomtJU0suzQhbw15aznMDam18EZ6QBkTxn1l
BoXxYAM8MVAASEJjZGlpXosmhoI0dG4FQuh2T4gYpOx9+L7eKxOTbqw79/xUe/bf
/+ZTi7//6U8Lls4XN60MUF1RPM46j6NIdUX/Yao0boxUCVHx4LfBhqpSGaQ6oEa1
my43TgvTzr3JW97+9p3vePtbD1x3ILlqosU70oRhDak15E0OUpqwikCtIS9CavJB
8Yxhtfko1b3xYZUCWESE7nBVHMZ74m4XxPlXZQDk85cUDQHtzGNpRRfvP7w0d89/
+erGw/fea7EyD/QqPat+9XUQlaswhgEyCqJytUhdEnJYDgkXESRgDBe3GWUaBlRR
M1+as+y86ZbGt77jHbvefOeNVx3YI1wx2eJGEpp+eyNdaDSABORzZSYyhiEM5aA5
YbwzIoC0O3heNBojAfmwPQwFQJlDtrTs1p58bm3h9z737OoXv/hFWj3xAuf3vvxQ
YjdWVqMqlzas2L+ZXNK4SjS2SytsG0ECNuni6t6ZH2hsXzHFEJW3GwCcXnsoeetb
3zpxxxtu33HTwcbuvbvNzmaDE2vAjVQo9DZIGnrepXxk6kCMsRwoQVAoYrAA4Hyk
a+9UncKtrWft6Rm3+ODjyyt//rVHO/ff/wBnZ6aK7zqs5FPlzgqAyvAMi4/GhagK
pCoV2rRLK+xigITMx3VzsUJVwVQOyC0G1ascvPfOu2OPueHW19nbbrutcejga9Lr
DiTJ3j22tXNSGo2UbJqwCW2bAlgAPKCqSm6j7TtLK35jeqbTfuHE+vozz72UPf74
43r86FHG8lz5PlZVbMbteMoQlZuEVOWN4uC4rDJlV+ZRD1F87nEhir7TywYSMGa8

```
    VAVUrEplEOpKdlUglUEsQxq/KlUuVVYlSmMb1cbdl9brVKhKjYYV+8tQ1JXeqkpk
    ZRdZBdGw6pCREAEXnEeqssrcUmyjckp1VpPPGJrGr3uIo9yNq/m/UWpSBiR+kOPk
    h6pcWLk0VpUXGhUP1QXVo5KOY0EEhF/uxbTiAihaL7Z59N59G8diNSgHrAb9QBQ3
    LVak4vNcvh6rEkXnL/aNSqbWWRXgxXUC1cCWk5FVrq0MTJXy1KlPFUx1QTUq1kfa
    RVAkoEKVULFct63qmLov5iuOqYtRhpWWfM2xdceNim+KhxwDUBU01wXLVaW0qtio
    DNywWGgciCru83gdvV6EGCm2sYLvqgx4HM+Ug3Ez5nrVvliJhsVIQHWsVPWjqHAF
    lYH2sBhp3FJb3b6q9WEAjXJrY7u08s24iLYpmMZNXhZAjANVfFwVRKOC7brAu87G
    icHKqlaGKcuXx4FH0a9MdfHQOHEisAWIgJcFJGCbYCoDVQVSGaI6EKvOjei88bUU
    1mszVW1x7Bdvix9cHCdVucYqEKqUpgqkYQBdVIiAix9sl03RS9ZVWTkAr/rCRZsd
    jo4pAm5fWubScrG/CiTK95XBBvoVaRRI5QJFnSoNi7NigKqWYxWq+t8qeMaEaGv2
    MikSUBqAZpQy1alTOf9TF/fEilV1XNV5yipU5eLiay9bXWkt/lHUuTUtzaugqYun
    yufZjApVQLS1URReRpCAMWACBh/gKHdX5fqGQVaGJ4br5YiRYmiq1GkULHVqNgqg
    KpUHtgEi4GUHCRgBU7xcfqhloKrgqot/Rk1ADyZgECZE24HB+1YXH1U9xGFx0qhp
    mPuqAqgMEyqWc7uw8VxeAZCAITAV82GubhRM4wIjNecdFh8VxqV1X1qvi5OqlmP1
    KY4fdypDtJlYaNsgAl4xkIAtwlSlTsPAio+Rmv3DYCqsXlKrU6RiuUqZhgEVw1BV
    PTNKfUbFQRcVIuAVBamwgdRAsTwMqlFxVNVyeVu8DgwCVAUUMKhGhdWpUjk28dHx
    VUBVQTTMfQ0DCaVl9Ja3d2iySwAk4ALVaVQ8NWwO1MdGxTpQDVSdld1avK0qVqpS
    lmGKM476vCwqFNslAhIwBkzFfCtADVsuB9V1rg0Yfb+GBd1VpaZx3N5mAcKQOS4G
    RMAlBRJQMdhxHUjx8rhQjYKoCqoqmEZZGaJiPmqqAmWz8AwBCLhYEAGXHEiFVapT
    sbxVqDYzVZ2/6npiq4hDKh/0dk1156+4nosHUGGXKEjAmOpUzOuAqto2Cpyq8wKD
    96pYLz+k8sOsAqm8Pg4s46rPy6ZCsV3CIBW2KaCK+TiAjDqmal6+hsLqYKqajwvJ
    sPVh89xeHoAKuwxAKmxLQBXzUeo1znnKy1VW597q5uO4qUsaoMIuI5AKqwUqXh4X
    qrp9VfurPi+2UapULI8Cq25f1bkie2UAKuwyBKmwsYCKI7eiPFX3Z1yQ4m2bVaq6
    /6n4nFcWoMIuY5AKGwAKGB+qrSyPY+NAMAyuYcu5XRoAFfZNAFJsm4JqM/uGbYtt
    mCpVrW9mHy41eGL7JgMptpFQ1W2ruyfj3qu6hz0MmiHbLI14YvsmBqlslWABFw7O
    KBsXrNwuD3DK9pcIpLLVglW2rd6jMYG4PMEp219ikEbZ2KCNsG8OUEbZ/w8fiHbq
    OyituQAAAABJRU5ErkJggg==
} As Png4

\\ you can use it as Png4 or you can export to file
Open "test.png" for output as #k
        Put #k, Png4
Close #k
```

Appendix II

Help All show the following list. Most of them are identifiers and some of them are information (like Formating or Auto_Arrays)

INTERPRETER
ABOUT, AND, CLEAR, CLIPBOARD, DIV, DIV#, DOS, EDIT, END, FAST, FKEY, HELP, IS, KEYBOARD, LIST, LOAD, MOD, MOD#, MONITOR, NEW, OPERATORS, OR, PROTOTYPE, RECURSION.LIMIT, REM, REMOVE, SAVE, SCRIPT, SLOW, SORT, START, SWITCHES, TEST, VERSION, WIN, WRITER

MODULE COMMANDS
ERROR, ESCAPE, FUNCTION, HALT, INLINE, LINK, MODULE, MODULES, PIPE, STOP, SUB, THREAD, THREADS, USE

FLOW CONTROL
AFTER, BREAK, CALL, CASE, CONTINUE, DO, ELSE, ELSE.IF, EVERY, EXIT, FOR, GOSUB, GOTO, IF, LOOP, MAIN.TASK, ON, PART, PROFILER, REPEAT, RESTART, SELECT, THEN, THREAD.PLAN, TRY, UNTIL, WAIT, WHILE

STACK COMMANDS
COMMIT, DATA, DROP, FLUSH, OVER, PUSH, READ, REFER, SHIFT, SHIFTBACK, STACK

DEFINITIONS
AUTO_ARRAYS, BINARY, BUFFER, CLASS, CONST, DECLARE, DEF, DIM, DOCUMENT, ENUM, ENUMERATION, EVENT, GLOBAL, GROUP, INVENTORY, LAMBDA, LET, LOCAL, LONG, METHOD, PROPERTY, SET, STATIC, STOCK, SUPERCLASS, SWAP, WITH

DOCUMENTS
APPEND.DOC, EDIT.DOC, FIND, INSERT, LOAD.DOC, MERGE.DOC, OVERWRITE, SAVE.DOC, SORT(DOCUMENT), WORDS

FILE OPERATIONS
BITMAPS, CLOSE, DRAWINGS, FILES, GET, LINE INPUT, MOVIES, NAME, OPEN, PUT, SEEK, SOUNDS, WRITE

CONSOLE COMMANDS
BACK, BACKGROUND, BOLD, CHARSET, CLS, CURSOR, DESKTOP, DOUBLE, FIELD, FONT, FORM, FRAME, GRADIENT, GREEK, HIDE, HOLD, ICON, ITALIC, LATIN, LAYER, LEGEND, LINESPACE, LOCALE, MARK, MODE, MOTION, MOTION.W, NORMAL, PEN, REFRESH, RELEASE, REPORT, SCROLL, SHOW, WINDOW

SCREEN & FILES
?, HEX, INPUT, PRINT

OPERATORS IN PRINT
$(, @(, ~(

TARGET & MENU
CHANGE, MENU, SCAN, TARGET, TARGETS

DRAWING 2D
CIRCLE, COLOR, CURVE, DRAW, FILL, FLOODFILL, MOVE, POLYGON, PSET, SMOOTH, STEP, WIDTH

BITMAP COMMANDS
COPY, IMAGE, PLAYER, SPRITE

DATABASES
APPEND, BASE, COMPRESS, DB.PROVIDER, DB.USER, DELETE, EXECUTE, ORDER, RETRIEVE, RETURN, SEARCH, STRUCTURE, TABLE, VIEW

SOUNDS AND MOVIES
BEEP, CHOOSE.ORGAN, MOVIE, MEDIA, MUSIC, PLAY, SCORE, SOUND, SPEECH, TONE, TUNE, VOLUME

MOUSE COMMANDS
JOYPAD, MOUSE.ICON

BROWSER COMMANDS
BROWSER, TEXT, HTML

COMMON DIALOGUES
CHOOSE.COLOR, CHOOSE.FONT, DIR, OPEN.FILE, OPEN.IMAGE, SAVE.AS, SETTINGS, SUBDIR, TITLE

ARITHMETIC FUNCTIONS
#EVAL(, #FILTER(, #FOLD(, #MAP(, #MAX(, #MIN(, #POS(, #REV(, #SLICE(, #SORT(, #SUM(, #VAL(, ABS(, ARRAY(, ASC(, ASK(, ATN(, BACKWARD(, BANK(, BINARY.ADD(, BINARY.AND(, BINARY.NEG(, BINARY.NOT(, BINARY.OR(, BINARY.ROTATE(, BINARY.SHIFT(, BINARY.XOR(, BUFFER(, CAR(, CDATE(, CDR(, CEIL(, CHRCODE(, COLLIDE(, COLOR(, COMPARE(, CONS(, COS(, CTIME(, DATE(, DIMENSION(, DOC.LEN(, DOC.PAR(, DOC.UNIQUE.WORDS(, DOC.WORDS(, DRIVE.SERIAL(, EACH(, EOF(, EVAL(, EXIST(, EXIST.DIR(, FILE.STAMP(, FILELEN(, FLOOR(, FORWARD(, FRAC(, FREQUENCY(, FUNCTION(, GROUP(, GROUP.COUNT(, HIGHWORD(, HILOWWORD(, HSL(, IF(, IMAGE(, IMAGE.X(, IMAGE.X.PIXELS(, IMAGE.Y(, IMAGE.Y.PIXELS(, INKEY(, INSTR(, INT(, JOYPAD(, JOYPAD.ANALOG.X(, JOYPAD.ANALOG.Y(, JOYPAD.DIRECTION(, KEYPRESS(, LEN(, LEN.DISP(, LN(, LOCALE(, LOG(, LOWWORD(, MATCH(, MAX(, MAX.DATA(, MDB(, MIN(, MIN.DATA(, MODULE(, NOT, NOT_2, ORDER(, PARAGRAPH(, PARAGRAPH.INDEX(, PARAM(, POINT(, POINTER(, PROPERTY(, RANDOM(, READY(, RECORDS(, RINSTR(, ROUND(,

SEEK(, SGN(, SIN(, SINT(, SIZE.X(, SIZE.Y(, SQRT(, STACK(, STACKITEM(, TAB(, TAN(, TEST(, TIME(, UINT(, USGN(, VAL(, VALID(, WRITABLE(

STRING FUNCTIONS
#EVAL$(, #FOLD$(, #MAX$(, #MIN$(, #VAL$(, ADD.LICENSE$(, ARRAY$(, ASK$(, BMP$(, CHR$(, CHRCODE$(, DATE$(, DRIVE$(, DRW$(, ENVELOPE$(, EVAL$(, FIELD$(, FILE$(, FILE.APP$(, FILE.NAME$(, FILE.NAME.ONLY$(, FILE.PATH$(, FILE.TITLE$(, FILE.TYPE$(, FILTER$(, FORMAT$(, FUNCTION$(, GROUP$(, HEX$(, HIDE$(, IF$(, INPUT$(, JPG$(, LAZY$(, LCASE$(, LEFT$(, LEFTPART$(, LOCALE$(, LTRIM$(, MAX.DATA$(, MEMBER$(, MEMBER.TYPE$(, MENU$(, MID$(, MIN.DATA$(, PARAGRAPH$(, PARAM$(, PATH$(, PIECE$(, PIPENAME$(, PROPERTY$(, QUOTE$(, REPLACE$(, RIGHT$(, RIGHTPART$(, RTRIM$(, SHORTDIR$(, SHOW$(, SND$(, SPEECH$(, STACK$(, STACKITEM$(, STACKTYPE$(, STR$(, STRING$(, STRREV$(, TIME$(, TITLE$(, TRIM$(, TYPE$(, UCASE$(, UNION.DATA$(, WEAK$(

VARS READ ONLY
ABOUT$, APPDIR$, BROWSER$, CLIPBOARD$, CLIPBOARD.IMAGE$, CODEPAGE, COLORS, COMMAND$, COMPUTER$, CONTROL$, DIR$, DURATION, EMPTY, ERROR$, FIELD_as variable, FONTNAME$, GRABFRAME$, GREEK_variable, HEIGHT, HWND, INKEY$, INTERNET, INTERNET$, ISLET, ISNUM, KEY$, LAN$, LETTER$, MEMORY, MENU.VISIBLE, MENU_as variable, MENUITEMS, MODE_variable, MODULE$, MONITOR.STACK, MONITOR.STACK.SIZE, MOTION.WX, MOTION.WY, MOTION.X, MOTION.XW, MOTION.Y, MOTION.YW, MOUSE, MOUSE.KEY, MOUSE.X, MOUSE.Y, MOUSEA.X, MOUSEA.Y, MOVIE.COUNTER, MOVIE.DEVICE$, MOVIE.ERROR$, MOVIE.STATUS$, MOVIE_as variable, MUSIC.COUNTER, NOW, NUMBER, OS$, OSBIT, PARAMETERS$, PEN_variable, PLATFORM$, PLAYSCORE, POINT, POS, POS.X, POS.Y, PRINTERNAME$, PROPERTIES$, REPORTLINES, RND, ROW, SCALE.X, SCALE.Y, SPEECH as variable, SPRITE$, STACK.SIZE, TAB, TEMPNAME$, TEMPORARY$, THIS, THREADS$, TICK, TIMECOUNT, TODAY, TWIPSX, TWIPSY, USER.NAME$, VOLUME_as variable, WIDTH_as variable, X.TWIPS, Y.TWIPS

CONSTANTS
ASCENDING, BINARY_const, BOOLEAN, BYTE, CURRENCY, DATEFIELD, DESCENDING, DOUBLE_as constant, FALSE, FALSE _2, FORMATING _ANY TYPE, FORMATING_DATE AND TIME, FORMATING_NUMBERS, FORMATING_STRINGS, INFINITY, INTEGER, ISWINE, LONG_TYPE, MEMO, PI, SINGLE, TEXT_as constant, TRUE, TRUE _2, VERSION_as constant

PRINTINGS
PAGE, PRINTER, PRINTING, PROPERTIES

Table of Contents