Oxygen Basic Compiler
Introduction

Oxygen Basic (o2H) is a Just-In-Time compiler assembler, deployed in a single DLL that takes source code strings and returns executable binary. It is an extension of the O2 assembler. This implementation can be embedded in any application to provide high performance programmability. The compiler can also generate conventional executables and dynamic link libraries (DLL).

Oxygen supports Object Oriented Programming with multiple inheritance and also single inheritance modes, more suited to COM. In addition to this O2H supports User-defined operators and operator sets for use with complex numbers, matrices, vectors, sets and other specialised forms. It can also compile & execute strings of source code at run time with its built in compile() function, opening new possibilities for functional programming.

The built in function set is small. but contains the necessary tools to build class / function libraries to any scale.

In addition to understanding Basic in the QBasic genre, Oxygen can read sufficient C syntax to deal with most C headers directly, eliminating the need to translate them into Basic.
types

void, sbyte, ubyte, byte, string, string2, wstring, bstring, gstr_, bstr, bstring2, char, wchar, cstr_, asciiz, zstring, zstring2, asciiz2, short, wide, long, int, integer, float, single, double, extended, quad, word, dword, ulong, uint, qword, any, sys, boolean, bool,

USE: to specify the types of variables and create them

EXAMPLE:

int x
dim as int x
dim x as int
var int x
**REMARKS:** C style instantiations is a simpler alternative to using 'Dim'.

**RELATED:** global  local  static

```
'------------------
'DIM VARIATIONS
'------------------

'------------------
'POST DEFINE TYPE
'------------------

dim i,j,k as long

'------------------
'PRE DEFINE TYPE
'------------------

dim as long i,j,k

'------------------
'MIXED TYPES
'------------------

dim as long i,j,k, as string s,t

'------------------
'MULTI LINE
'------------------

dim as long i,j,k,
    as string s,t

dim as long,
    i,
    j,
    k
```

---
'INITIAL VALUES
'==============

dim as long,
i = 1,
j = 2,
k = 42

'-------------------------
'SPREAD LINES AND COMMENTS
'=================================

dim as long,
i = 1, ' these can be spread over many lines

'-------
j = 2, ' with intervening comments

'-------
k = 42 '

'-----------------
'MULTIPLE ASSIGNMENTS
'=====================

dim as long a(10) => (2, 4, 6, 8, 10, 12, 42, 99)
print "Answer: " str a(7)

'-----------------
'SYNTAX VARIATIONS
'=================

dim long a(10) => (2, 4, 6, 8, 10, 12, 42, 99)
dim long a[10] => (2, 4, 6, 8, 10, 12, 42, 99)
long a[10] => (2, 4, 6, 8, 10, 12, 42, 99)
long a[10] <= (2, 4, 6, 8, 10, 12, 42, 99)
long a[10] = {2, 4, 6, 8, 10, 12, 42, 99}
long a[] = {2, 4, 6, 8, 10, 12, 42, 99}
long a =  
{
  2, 4, 6, 8, 10, 12, 42, 99
}
'------------------
'POINTERED VARIABLE
'==================

```
dim string s = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
dim byte b at strptr s
dim byte byref b : @b=strptr s
byte b at strptr s
byte *b = strptr s

'print b[7] '71 G
```

'-----------------'
'USING DYNAMIC MEMORY
'=====================

```
dim float f at getmemory 1024*4 : f={1.5, 2.5, 3.5}
'print f[2]
freememory @f 'release allocated memory
```

'------------------------'
'DIMS: GLOBAL, STATIC, LOCAL
'==========================

```
global int g=1 'visible to rest of program following t!

dim float f at getmemory 1024*4 : f={1.5, 2.5, 3.5}
'print f[2]
freememory @f 'release allocated memory

'------------------------'
'DIMS: GLOBAL, STATIC, LOCAL
'==========================

```
global int g=1 'visible to rest of program following t!

function f(p as int) as int
  static int s=0 'permanent storage
  local int l=100 'temporary storage
  s+=10
  return p+l+s+g
end function

print f(1000) '1111
print f(1000) '1121
```

'------------------------'
'LIMITING SCOPE
'===============

```
dim long a=16
```
scope
dim long a=1
'print a '1
...
end scope
'print a '16
inhstructures

**type**,  

**USE:** specify compound structure for a variable  

**RELATED:** *struct typedef*

`------------
'COMPOUND TYPES
'-------------

type color32

  r as byte  
  g as byte  
  b as byte  
  a as byte  
  =  
  rgba as long  'UNION

end type

'------------
'DERIVED TYPE:
'-------------

  type colortext

    txt as string  
    c as color32

end type


dim t as colortext

t.txt=`Color code `  
t.c.r=8  
t.c.b=16  
t.c.g=32  
t.c.a=64

print t.txt hex t.c.rgb`
'print "STRUCTURE:
" structureof color32

'-------------
'SYNTAX VARIATIONS
'-------------

type color32
  byte r
  byte g
  byte b
  byte a
  =
  long rgba  'UNION
end type

type color32
  byte r,g,b,a
  =
  long rgba  'UNION
end type

type color32 byte r,g,b,a = long rgba

type colortext string txt,color32 c

struct color32 {
  byte r,g,b,a
  =
  long rgba
}

typedef struct _color32 {
  byte r,g,b,a
  =
  long rgba
} color32, *pcolor32

typedef struct _color32 {
  union {
    struct {
      byte r,g,b,a
    }
    long rgba
  }
} color32, *pcolor32
} color32, *pcolor32

'#recordof color32
'#recordof _color32
'#recordof colortext
includes

\texttt{once},

\textbf{ACTION:} ensures that a file is included in the source code only.

\textbf{EXAMPLE:}
\begin{verbatim}
#include once ".././MinWin.inc"
\end{verbatim}

\textbf{RELATED:} include \#include includepath
equates

\$ , \%

REMARKS: $ and % are equivalent

RELATED: def #def macro #define

'$filename "t.exe"
'uses rtl64

'-----------------'
'EQUATES: $ or %'
'-----------------

'strictly speaking these are single-line macros

% a = 1
% b = 2

% c = 3
% c = 3
% c = 3

$d = 4
$d = "four"
%d = "four"

%= e b * c 'precalcuate e value '6
'#recordof e

% f = 8
% % f = 7 'default value 7 (unless previously defined)

% sp = ""
% cm = ","

'$ $ % prefixes and suffixes are ignored

print "" a cm b cm c cm d cm e cm f '1,2,3,four,6,8
'print %a cm %b cm %c cm %d cm %e cm f

'INCLUDING ARGUMENTS
arguments are represented by %1 .. %9

% display "value of %1: " %1

print display d
arrays
arrays

REMARKS: Only single dimensioned arrays are directly supported

RELATED: dim

'_filename "t.exe"
'uses rtl64

'-------------
'STATIC ARRAYS
'-------------

dim as long a(10)={2,4,6,8,10,12}
a(10)=a(1)+a(4)

print a(10)

'-------------
'DYNAMIC ARRAYS
'-------------

dim as long a at getmemory(10*sizeof(long)) : a={2,4,6, ...
freememory @a

dim as long a(10)={2,4,6,8,10,12}

'--------
'OVERLAYS
'--------

dim as string s = "ABCDEFGHIJ"
dim as byte b at strptr(s)
print str(b[3]) ": " chr(b[3])

'-----------------------
'MULTIDIMENSIONAL ARRAYS
'-----------------------

macro a(x,y) av(y*1024+x)
dim int av[1024*1024]
a(100,200)=42
print a(100,200) ;42

'--------
'INDEX BASE
'==========

dim int a[100]={10,20,30,40}
indexbase 1 'default: first element is indexed as 1
'print a[2] '20
indexbase 0
print a[2] '30

'--------
'PSEUDO ARRAYS
'============

dim av[100]

function a(int i,v) 'setter
  i*=2
  av[i]=v
end function

function a(int i) as int 'getter
  i*=2
  return av[i]
end function

a(7)=42 'this is interpreted as a(7,42)
'print a(7)
blocks

block, scope, skip, o2, (), RELATED: procedures macros

'------
'MACROS
'------

macro cube(v)
  v*v*v
end macro

#define cube(v) v*v*v
def cube %1 * %1 * %1
% cube %1 * %1 * %1
$ cube %1 * %1 * %1

'print cube 3

'MACRO MEMBERS
'=============

macro multiple(v)
  macro .two(v)
    v*v
  end macro
  macro .three(v)
    v*v*v
  end macro
  macro .four(v)
    v*v*v*v
  end macro
end macro

print multiple.four 3
macros

macro, define, def, deff, %, $,

RELATED: macro functions macro operators procedures
procedures

    function, sub, method, gosub,

RELATED: macro

'----------
'PROCEDURES
'==========

'SUBROUTINES

float v
goto ncube
cube: 'subroutine
'print v*v*v
ret
ncube:

v=3 : gosub cube

'SUBS AND FUNCTIONS
'=====================

sub cube(f as float, g as float)
    g=f*f*f
end sub

dim float a
cube 2,a
'print a

function cube(f as float) as float
    function=f*f*f
end function

function cube(f as float) as float
    return f*f*f
end function

'print cube 2

'METHODS (IN CLASSES)
'=====================

class multipliers
method cube(f as float) as float
    return f*f*f
end method
end class

dim multipliers m
'print m.cube 4

'ALTERNATIVE SYNTAX
'======================

'USE OF {...}
'=============

function cube(f as float) as float {return f*f*f}

function cube(f as float) as float {
  return f*f*f
}

float cube(f as float) {
  return f*f*f
}

float cube(float f) {
  return f*f*f
}

'print cube 2

'PASSING PARAMETERS BY REFERENCE
'=================================

dim as float f()={1,2,3,4,5}

function cubes(f as float, byval n as int) 'default byref
  indexbase 1
  int i
  for i=1 to n
    v=f[i]
    print v*v*v
  next
end function

function cubes(float*f, int n)
INDEXBASE 1
FLOAT V
INT I
FOR I=1 TO N
  V=F[I]
  PRINT V*V*V
NEXT
END FUNCTION

'cubes f(2),3
'cubes f, countof(f)

'OPTIONAL PARAMETERS
'=-=-=-=-=-=-=-=-

FUNCTION CUBES(F AS FLOAT, OPTIONAL BYVAL N AS INT)
  IF N=0 THEN N=1
  INDEXBASE 1
  FLOAT V
  INT I
  FOR I=1 TO N
    V=F[I]
    PRINT V*V*V
  NEXT
END FUNCTION

'cubes f(3)
'cubes f(3),1
'cubes float{1,2,3,4},countof 'PASSING LITERAL DATA SET

'DEFAULT PARAMETERS
'=-=-=-=-=-=-=-=-

FUNCTION CUBES(ARRAY F, INT N=2)
  INDEXBASE 1
  FLOAT V
  INT I
  FOR I=1 TO N
    V=F[I]
    PRINT V*V*V
  NEXT
END FUNCTION

'cubes f(3)
'cubes f(3),2
function cuber(int n, ...)  
    indexbase 0  
    float v  
    int i  
    for i=1 to n  
        v= (int) param[i]  
        print v*v*v  
    next  
end function  
'cuber 3, 2,3,4
conditionals

if, then, elseif, else, endif,

RELATED: selection loops

'--------------
'CONDITIONALS
'-------------

string s
int a=1, b=2

'SINGLE LINE FORMAT

if a>b then s="A>B" else s="A<=B"

'print s

'MULTI-LINE FORMAT

if a>b then
  s="A>B"
elseif a=b then
  s="A=B"
else
  s="A<B"
end if

'--------------
'SYNTAX VARIATIONS
'-----------------

if (a>b) {s="A>B"} elseif (a=b) {s="A=B"} else {s="A<B"}

if a>b {s="A>B"} elseif a=b {s="A=B"} else {s="A<B"}

if a>b {
  s="A>B"
} elseif a=b {
  s="A=B"
} else {
  s="A<B"
}
print "A=1 B=2"
" s
loops

do , while , exit , continue , wend , enddo ,

RELATED: iteration conditionals

'-----
'LOOPS
'======

dim a,b,c,d as long, s as string

'SIMPLE LOOPS
'-------------

a=4
'b=0
do
    b+=1
    if b>a then exit do
end do 'or enddo

b=0
do
    b+=1
    if b>a then exit do
loop

'-------------
'CONDITIONAL FORMS
'--------------

b=0
while b<=a
    b+=1
wend

b=0
while b<=a {b+=1}
b=0
do {b+=1} while b<a

b=0
do {b+=1} until b>=a

b=0
do
    b+=1
loop while b<a

b=0
do
    b+=1
loop until b>=a

b=0
do
    b+=1
    if b<a then continue do
    if b<a then repeat do
    if b>=a then exit do
    if b>=a then break
loop

b=0
do
    b+=1
    continue while b<a  'if/when/while
    continue until b>=a
    repeat until b>=a
    redo until b>=a
    redo until not b<a
    exit when b>=a       'if/when
    exit when not b<a
    break when b>=a      'if/when
end do

print "ok"
iteration

for, to, step, next,

RELATED: loops

'----------
'ITERATION
'----------

dim a, b, c, d, i as long
dim s as string = "QWERTY"

'checksum example

b=0
for i=1 to len(s)
  b+=asc(s,i)
next

b=0
for i=1 to len(s) step 1
  b+=asc(s,i)
next

b=0
for i=len(s) to 1 step -1
  b+=asc(s,i)
next

'-----------------
'SYNTAX VARIATIONS
'-----------------

b=0
for i=1 to len(s) step 1 {
  a=asc(s,i)
  b+=a
}

b=0
for i=1,len(s),1 {
  a=asc(s,i)
  b+=a
b=0
for (i=1, i<=len(s), i++) {
    a=asc(s,i)
    b+=a
}

#semicolon separator
b=0
for (i=1; i<=len(s); i++) {
    a=asc(s,i)
    b+=a
}
#semicolon comment

b=0
for (i=1, i<=len(s), i++) {
    a=asc(s,i)
    b+=a
}

def qu ""
print "Checksum for " qu s qu " = " b
selection
select , case , case , endselect ,

RELATED: conditionals

'------
'SELECT
'------

dim a as long, s as string
a=3

'COMPACT FORM
'---------

select a 'select case a

case 1 : s="A=1"
case 2 : s="A=2"
case 3 : s="A=3"
case else : s="A>3"

end select 'endsel

'GENERAL FORM
'----------

select a

case 1
  s="A=1"
case 2
  s="A=2"
case 3
  s="A=3"
case else
  s="A>3"

end select

'------------------
'SYNTAX VARIATIONS
select a {
    case 1
        s="A=1"
    case 2
        s="A=2"
    case 3
        s="A=3"
    case else
        s="A>3"
}

switch a {
    case 1
        s="A=1"
        break
    case 2
        s="A=2"
        break
    case 3
        s="A=3"
        break
    case else
        s="A>3"
        break
}

EXTENSIONS

select a
    case 1
        s="A=1"
    case 2
        s="A=2"
    case 3
        s="A=3"
    case 4,5,6

case 7 to 9

case 10 to <20

case else
    s = "A>3"

end select

print s
structures

type,

RELATED: classes class

'-----------------
'COMPOUND TYPES
'-----------------

type color32

    r as byte
    g as byte
    b as byte
    a as byte

    rgba as long 'UNION

end type

'-----------------
'DERIVED TYPE:
'-----------------

type colortext

    txt as string
    c as color32

end type

dim t as colortext

t.txt=`Color code`
t.c.r=8
t.c.b=16
t.c.g=32
t.c.a=64

print t.txt hex t.c.rgb

'print "STRUCTURE:
" structureof color32
SYNTAX VARIATIONS

type color32
    byte r
gbyte	g
    byte b
    byte a
    =
    long rgba 'UNION
end type

typedef struct _color32 {
    byte r, g, b, a
    =
    long rgba
} color32, *pcolor32

typedef struct _color32 {
    byte r, g, b, a
    =
    long rgba
} color32, *pcolor32
'#recordof color32
'#recordof _color32
'#recordof colortext
classes

class, objects, oop, has, of, from, inherits, virtual, pure, com, new, del,

RELATED: structures type

'--------
'CLASSES
'========

'METHODS (IN CLASSES)
'=====================

class multipliers
  method cube(f as float) as float
    return f*f*f
  end method
end class

dim multipliers m
'print m.cube 4

'ALTERNATIVE SYNTAX
'====================

class multipliers {
  method cube(f as float) as float {
    return f*f*f
  }
}

dim multipliers m
print m.cube 4
operators

**ACTION:** changes the state of an accumulator

**USE:** formulating expressions, in conjunction with operands

**EXAMPLE:** \(a \cdot b + c / 4\)

**REMARKS:** universal feature of maths and programming languages

**RELATED:** types
calling

\texttt{stdcall}, \texttt{cdecl}, \texttt{ms64}, \texttt{pascal},

\textit{ACTION}: determines how parameters are passed on the stack, when making a call
**void**

**USE:** specify a null type

**EXAMPLE:**
'Variables:
void * pv = getmemory(100 * sizeof float)
...
freememory pv
'

'In function headers:
function foo(byref v as void) as void ptr
void* foo(void*v)
'

'Procedures not returning a value:
void foo()
'

**REMARKS:** Void cannot be used directly.

**RELATED:** sys any types
sbyte

USE: specify a signed byte type (8 bits wide)

EXAMPLE: sbyte x=-10

REMARKS: limited to values ranging from -128 to 127 / 0x80 to 0x7F

RELATED: types
ubyte

USE: specify a byte type (8 bits wide)

EXAMPLE: ubyte semicolon=59

REMARKS: limited to values 0..255 / 0x00 to 0xFF

RELATED: types
**byte**

*USE:* specify a byte type (8 bits wide)

*EXAMPLE:*

```
byte colon=58
```

*REMARKS:* limited to values 0..255 / 0x00 to 0xFF

*RELATED:* types
**string**

**ACTION:** returns a string of characters

**USE:** string=string(length,character)

**EXAMPLE:**

string=string(4,"a")

**RESULT:**

s="aaaa"

**RELATED:** space nuls asc
string2

**USE:** specify a string type with 16-bit characters, supporting unicode

**EXAMPLE:**
```c
wstring sw
getfile "greek.txt", sw
```

**REMARKS:** strings are automatically destroyed when out of scope.

**RELATED:** unic types
**wstring**

**USE:** specify a string type with 16-bit characters, supporting unicode

**EXAMPLE:**
```cpp
wstring sw
getfile "greek.txt",sw
```

**REMARKS:** strings are automatically destroyed when out of scope.

**RELATED:** unic types
**bstring**

**USE:** specify a bstring type with 8-bit characters

**EXAMPLE:**
```
bstring s="name: 
... 
frees s
```

**REMARKS:** bstrings must be freed before going out of scope.

**RELATED:**  [bstring2 types](#)
gstr_
bstr

**USE:** specify a bstring type with 8-bit characters

**EXAMPLE:**

```c
bstr s="name: "
...
frees s
```

**REMARKS:** bstrings must be freed before going out of scope.

**RELATED:** bstring2 types
**bstring2**

*USE:* specify a bstring type with 16-bit characters, supporting unicode

*EXAMPLE:*

```c
bstring2 sw
getfile "greek.txt",sw
...
frees sw
```

*REMARKS:* bstrings must be freed before going out of scope.

*RELATED:* bstring types
char

**USE:** specify a string of ascii characters (8 bits wide)

**EXAMPLE:**
```c
char w="world"
char buf[1024]
buf=w
print "hello "+buf
```

**REMARKS:** similar to C char, but is not conflated with byte which is a numeric type

**RELATED:** wchar types
**wchar**

**USE:** specify a string of wide characters (16 bits wide)

**EXAMPLE:**
```c
wchar w="world"
wchar buf[1024]
buf=w
print buf
```

**RELATED:** char types
cstr_
**asciiz**

**USE:** specify a string of ascii characters (8 bits wide)

**EXAMPLE:**
```
asciiz w="world"
asciiz buf[1024]
buf=w
print "hello "+buf
```

**REMARKS:** similar to C char, but is not conflated with byte which is a numeric type

**RELATED:** wchar types
zstring

**USE:** specify a string of ascii characters (8 bits wide)

**EXAMPLE:**
```c
zstring w="world"
zstring buf[1024]
buf=w
print "hello "+buf
```

**REMARKS:** similar to C char, but is not conflated with **byte** which is a numeric type

**RELATED:** wchar types
**zstring2**

**USE:** specify a string of wide characters (16 bits wide)

**EXAMPLE:**
```
zstring2 w="world"
zstring2 buf[1024]
buf=w
print buf
```

**RELATED:** char types
asciiz2

**USE:** specify a string of wide characters (16 bits wide)

**EXAMPLE:**

```
asciiz2 w="world"
asciiz2 buf[1024]
buf=w
print buf
```

**RELATED:** char types
**short**

**USE:** specify a short integer (16 bits wide). Also used in conjunction with other types to halve the bit width

**EXAMPLE:**
```
short a
short int b
short short c 'an sbyte
```

**RELATED:** long types
**wide**

**USE:** specify a wide character string (16 bits wide). Also used in conjunction with other types to double the bit width

**EXAMPLE:**
- `wide s`
- `wide char sw`
- `wide float fw 'a double precision float`

**RELATED:** short types
long

USE: specify a long integer (32 bits wide). Also used in conjunction with other types to double the bit width

EXAMPLE: long i

RELATED: short types
**int**

*USE:* specify a long signed integer (32 bits wide)

*EXAMPLE:* 
```c
int i=0x7fffffff
```
integer

**USE:** specify a long signed integer (32 bits wide)

**EXAMPLE:**

```
integer i=0x7fffffff
```

**RELATED:** dword types
**float**

**USE:** specify a floating point variable (32 bits wide)

**EXAMPLE:**

```c
float f = 1/100
```

**REMARKS:** same as single

**RELATED:** single double extended types
**single**

**USE:** specify a floating point variable (32 bits wide)

**EXAMPLE:**

```
single f=1/100
```

**REMARKS:** same as float

**RELATED:** float double extended types
**double**

_**USE:**_ specify a double precision floating point variable (64 bits wide)

_**EXAMPLE:**_

double f=1/3

**RELATED:**  single  float  extended  types
**extended**

**USE:** specify an extended precision floating point variable (80 bits wide)

**EXAMPLE:**
extended e=1/3

**REMARKS:** this type holds the full precision of the pentium floating point processor (FPU)

**RELATED:** single float double types
**quad**

*USE:* specify a double precision signed integer (64 bits wide)

*EXAMPLE:*

quad q

*REMARKS:* In a 64 bit system, these are processed directly on the CPU.
In a 32 bit system, quads are passed to the FPU for processing

*RELATED:* qword int short sbyte types
word

USE: specify a short unsigned integer (16 bits wide)

EXAMPLE: word w=0xA000

RELATED: short int dword types
**dword**  
*USE:* specify a long unsigned integer (32 bits wide)  
*EXAMPLE:*
dword ui=0xA0000000

*RELATED:* word long int quad types
ulong

USE: specify a long unsigned integer (32 bits wide)

EXAMPLE:
ulong u=0xA0000000

REMARKS: same as uint

RELATED: uint word types
**uint**

**USE:** specify a long unsigned integer (32 bits wide)

**EXAMPLE:**
```c
uint u=0xA0000000
```

**REMARKS:** same as dword and ulong

**RELATED:** dword word byte types
qword

USE: specify a 64 bit operand in assembly code

EXAMPLE: fld qword d

REMARKS: Only used in assembly code, not Basic.

RELATED: integer quad types
any

**USE:** specify a parameter of uncertain type, nominally a signed integer of system width (32/64 bits wide)

**EXAMPLE:**
```
function f(any*a) {...}
```

**REMARKS:** Parameter of any type may be passed by-reference. Like C `void*`.

**RELATED:** `sys types`
sys

**USE:** specify a signed integer of system width (32/64 bits wide)

**EXAMPLE:**
sys i=42

**REMARKS:** this type is always wide enough to hold a pointer.

**RELATED:** any int quad types
**boolean**

*USE:* specify a variable to hold Boolean true/false states

*EXAMPLE:*
```
boolean t=true
if not t then ... 
```

*REMARKS:* Notionally a Boolean type, but in reality, it is an sbyte (8 bit signed integer)

*RELATED:* bool byte int any types boolean types
**bool**

**USE:** specify a variable to hold Boolean true/false states

**EXAMPLE:**
```
bool t=true
if not t then ...
```

**REMARKS:** Notionally a Boolean type, but in reality, it is a 32bit signed integer, as in C

**RELATED:** any types boolean int types
**dim**

**ACTION:** define a set of variables  
**USE:** create variables and arrays of variables with optional initial values  
**EXAMPLE:**

dim as string s="Hello World"

**RELATED:** redim let var
**local**

*ACTION*: define a local set of variables

*EXAMPLE*: 

```plaintext
local string s
```

*RELATED*:  dim static
**static**

**ACTION:** define a static set of variables, (persistent but invisible outside the block)

**EXAMPLE:**
```
static string s
```

**RELATED:** dim local
**type**

**ACTION:**  define a compound variable type

**EXAMPLE:**
```
    type rgbacolor
        red   as byte
        green as byte
        blue  as byte
        alpha as byte
    end type
```

**RELATED:**  `typedef` `struct` `class`
struct

**ACTION:** define a compound variable type (C Syntax)

**EXAMPLE:**
```
struct rgbacolor
{
    red   as byte
    green as byte
    blue  as byte
    alpha as byte
}
```

**RELATED:** type typedef class
typedef

**ACTION:** define a set of types (C syntax)

**USE:** create type definitions for creating other types

**EXAMPLE:**
```c
typedef Double *pDouble
```

**RELATED:** type struct class union enum
include

ACTION:  include source code from another file

USE: used for including header files and other units of source code

EXAMPLE: #include "rtl32.inc"

REMARKS: include and #include are the same

RELATED:  includepath embedfile
#include

**ACTION:** include source code from another file

**USE:** used for including header files and other units of source code

**EXAMPLE:**
```
#include "rtl32.inc"
```

**REMARKS:** include and #include are the same

**RELATED:** includepath embedfile
**includepath**

**ACTION:** define a filepath for source files specified by `include`.

**RELATED:** `include librarypath`
$  

**ACTION:** define an equate (which can be used as constants)

**RELATED:** equates macros `define`, `def`
ACTION: define an equate (which can be used as constants)

RELATED: equates macros #define, #def, def
def

**ACTION:**  define a low level macro

**EXAMPLE:**  

```
'DEFINE MACRO:
```

**REMARKS:** def and #def are the same

**RELATED:**  macro #define deff
#def

**ACTION:** define a low level macro

**EXAMPLE:**

```
'DEFINE MACRO:
```

**REMARKS:** def and #def are the same

**RELATED:** macro #define deff
**macro**

**ACTION:** define a high level macro

**USE:** many uses. Often a pseudo function for producing inline code instead of a call

```
---
'MACROS
'=====

macro cube(v)
  v*v*v
end macro

#define cube(v) v*v*v
def cube %1 * %1 * %1
%cube %1 * %1 * %1
%cube %1 * %1 * %1
stdin cube %1 * %1 * %1

'print cube 3

'MACRO MEMBERS
'=============

macro multiple(v)
  macro .two(v)
    v*v
  end macro
  macro .three(v)
    v*v*v
  end macro
  macro .four(v)
    v*v*v*v
  end macro
end macro

print multiple.four 3
```
RELATED: macros #define #def def deff
#define

**ACTION:** define a macro (C syntax)

**USE:** C preprocessor statements

**EXAMPLE:**
```c
#define X 32
```

**RELATED:** macros macro def ifdef ifndef
block

ACTION: start a block

RELATED: blocks scope exit repeat do while if select
scope

**ACTION:**
start a scope

**USE:**
create a block where variables and functions may be locally defined

**EXAMPLE:**

```plaintext
sys i=4

scope
    sys i=8
    'print "inner scope i=" i
end scope

'-----------------
'EQUIVALENT SYNTAX
'=================

scope
{
    sys i=8
    'print "inner scope i=" i
}

( sys i=8
    'print "inner scope i=" i
)

print "outer scope i=" i 'i=4
```

**REMARKS:** when the scope ends any definitions created within the scope will be forgotten.

**RELATED:** blocks block namespace
**o2**

**ACTION:** start a block of o2 machine code notation

**USE:** inserting inline machine code and other binary data. Direct use of o2 language (used by the o2 linker)

**EXAMPLE:**
```
o2 b8 00 01 00 00 'machine code for mov eax,256
```

**RELATED:** blocks
**deff**

**ACTION:** create an assembly code macro for the FPU (metatype -17)

**USE:** to create floating point maths functions

**EXAMPLE:**

deff sine fsin

**REMARKS:** Deff macros make use of the FPU. All float functions are defined this way. These are non-recursive macros and take no macro arguments. The expression parser is FPU-aware and takes care of passing parameters onto the FPU stack.

**RELATED:** macros def #define macro sin cos
**macro functions**

**USE:** invoke multi-line macros within expressions

```
'USING MACRO FUNCTIONS
'2D ARRAY WITH BOUNDARY CLIPPING
indexbase 0
int pix[800*600]
'
'SINGLE-LINE MACRO FUNCTION
'macro pix2d(x,y) pix(y*800+x)
'
'MULTI-LINE MACRO RETURNING A UNIQUE VARIABLE
'macro pix2d int* (v,x,y, vv)
===================================
'v  pixel pointer supporting read/write
'x  horizontal coordinate
'y  vertical coordinate
'vv sink pixel
if x>=0 and x<800 and y>=0 and y<600
   @v=@pix(y*800+x)
else
   int vv=0xffffffff 'value when out of bounds
   @v=@vv
end if
end macro
'
'TEST
pix2d(1,20)=0xaabbccdd
print hex pix2d(1,20)
print hex pix2d(800,10)
```

**REMARKS:** implements in-line functions

**RELATED:** macros macro operators
**Macro Operators**

**USE:** encode operator sets for different types (UDTs)

'------------------
'MACRO OPERATORS
'------------------

'PARTIAL DEMO

type vector3f
  float x,y,z
end type

macro vector3f_op
  macro ."save"(a,acc)
    a.x = acc.x  
    a.y = acc.y  
    a.z = acc.z  
  end macro
  macro ."load"(acc,a)
    acc.x = a.x  
    acc.y = a.y  
    acc.z = a.z  
  end macro
  macro ."neg"(acc,a)
    acc.x = -a.x  
    acc.y = -a.y  
    acc.z = -a.z  
  end macro
  macro ."+"(acc,a)
    acc.x += a.x  
    acc.y += a.y  
    acc.z += a.z  
  end macro
  macro ."-"(acc,a)
    acc.x -= a.x  
    acc.y -= a.y  
    acc.z -= a.z  
  end macro
  macro ."*"(acc,a)
    acc.x *= a.x  
    acc.y *= a.y  
    acc.z *= a.z  
  end macro
macro ."/"(acc,a)
    acc.x /= a.x
    acc.y /= a.y
    acc.z /= a.z
end macro

macro ."str" string (out,a)
    out=str(a.x)+","+str(a.y)+","+str(a.z)
end macro

'TESTS
'===== 
'#recordof "r.txt" vector3f_op 
dim vector3f A = {1,2,3} 
dim vector3f B = {10,20,30} 
dim vector3f C = {10,100,1000} 

print str(A) 
print str(A+B) 
print str(C*(A+B)) 

REMARKS: implementing operators for compound types: vectors, complex numbers, matrices, etc.

RELATED: macros macro functions
**function**

**ACTION:** define a function

**EXAMPLE:**

```plaintext
function triple(i as int) as int
  return i*3
end function
```

**RELATED:** procedures sub method
**sub**

**ACTION:** define a sub. (like a function but not returning a value)

**EXAMPLE:**
```
sub triple(i as int, j as int)
    j = i * 3
end sub
```

**RELATED:** procedures function method
method

**ACTION:** define a method. (a function or sub for objects)

**EXAMPLE:**
```
method triple(i as int) as int
    return i*3
end method
```

**RELATED:** procedures function sub
gosub

**ACTION:** call a labelled subroutine

**USE:** invoke local subroutines inside a procedure

```plaintext
'% filename "t.exe"
'uses rtl64

function f()
    int a=42
    int b

    'gosub g
    gosub g when a>0
    print b
    return

    g:
    b=a/2
    ret

end function

f

RELATED: procedures call goto
```
if
ACTION: start a conditional block with a test
USE: conditional execution
EXAMPLE: if a<b then a=b

RELATED: conditionals then else elseif endif while
then

**ACTION:** starts the conditional block where the prior test is met.

**USE:** conditional execution

**EXAMPLE:**
```
if a>b then print a
```

**RELATED:** conditionals if elseif endif
elseif

**ACTION:** make an alternative test if the previous condition was not met.

**USE:** conditional execution

**RELATED:** conditionals if else endif
else

**ACTION:** starts the alternative block where none of the prior conditions are met

**USE:** conditional execution

**EXAMPLE:**

```
if a>b then print a else print b
```

**RELATED:** conditionals if elseif endif
endif
end if

ACTION: end the conditional block
USE: conditional execution
EXAMPLE: if a>b then
        a=b
        end if

RELATED: conditionals if then elseif else end if
**do**

**ACTION:** start a block for repetition (looping)

**EXAMPLE:**
```
a=0
do
    ...
    a+=1 : if a=4 then exit do
end do
```

**RESULT:**
```
a=4 '4 loops
```

**RELATED:** loops while continue exit enddo loop
\textbf{ACTION:} start a block for conditional repetition

\textbf{EXAMPLE:}
\begin{verbatim}
a=0
while a<4
    a+=1
wend
\end{verbatim}

\textbf{RESULT:} a=4 '4 loops

\textbf{REMARKS:} while \textbf{id a combination of} do and \textbf{if}.

\textbf{RELATED:} loops do continue exit wend enddo
exit

**ACTION:** exit a **do while for (...)** block immediately

**USE:** leave a loop, usually after a condition has been met

**EXAMPLE:**

```
a=0  :  b=1
   do
     a+=1
     if a>4 then exit
     do
       b+=b
     end do
   end do
```

**RESULT:**

```
a=5  :  b=16
```

**REMARKS:** Also used to exit low-level nesting blocks.

**RELATED:** loops do while continue repeat
**continue**

**ACTION:** go back to the beginning of a **do**, **while** or **for** block

**USE:** to "short circuit" a loop

**REMARKS:** continue do **and** continue while **will loop back to the nearest** do or while at the same nesting level.

**RELATED:** loops do while for break exit wend enddo repeat
wend
end while, loop

**ACTION:** end a **while** block

**USE:** see: **while**

**RELATED:** loops do while continue exit
endo
end do, loop

**ACTION:** end a **do** repeating block

**USE:** see: **do**

**RELATED:** loops while do loop continue exit
for

ACTION: start an iteration block

RELATED: iteration to step next
to

ACTION: specify limit of an iteration

RELATED: iteration for step next
**step**

*ACTION:* specify increment of an iteration

*USE:* for iterator = start to end step step increment

*EXAMPLE:*  

```
for i=1 to 10 step 2 : ... : next
```

*RESULT:*  
i increases by 2 with each cycle : 1 3 5 7 9

*RELATED:* iteration for to next
next

**ACTION:** end iteration block

**RELATED:** iteration for to step
select

*ACTION*: Start a Case block (C style)

*RELATED*: selection case endsel end
case

ACTION: specify a case to match followed by actions to perform

RELATED: selection select endsel end
class
ACTION: define a class (structure and methods for objects)
EXAMPLE: 'DEFINE CLASS
class rgbacolor
    red  as byte
    green as byte
    blue as byte
    alpha as byte
    method in(sys r, sys g, sys b, sys a)
        red=r : green=g : blue=b : alpha=a
    end method
end class

RELATED: classes type typedef struct
has

**EXAMPLE:**
```
class person
    has mind
```

**REMARKS:** 'has' is used when the class is derived from more than a single parental class. ie: multiple inheritance. 'has' may be omitted. It is sufficient to provide the class/type name without this qualifier.

**RELATED:** of from extends class classes
REMARKS: 'of', and 'from' and extends are equivalent. They indicate derivation from a single parental class. COM interfaces us single inheritance

RELATED: has class classes
REMARKS: 'of', and 'from' and extends are equivalent. They indicate derivation from a single parental class. COM interfaces us single inheritance

RELATED: has class classes
virtual

**REMARKS:** objects of virtual classes are usually created by some kind of server or class factory. The client only receives a reference (pointer) to the object from the server and only knows how to invoke its methods. It assumes no knowledge of its inner workings.

**RELATED:** com external export class extern com class
com

**REMARKS:** COM is a protocol used by Microsoft to provide Object interfaces. Classes with this attribute are considered Virtual.

**RELATED:** virtual external export class of inherits extern virtual class
**new**

**ACTION:** create a dynamic object and call its constructor method.

**USE:** to create and initialise persistent objects

**EXAMPLE:**
new shape cuboid 1,1,1

**RELATED:** classes del redim dim
**del**

**ACTION:** Call a dynamic object's destructor method and disallocate its memory block.

**USE:** to delete dynamic objects

**EXAMPLE:**
```
del cuboid
```

**REMARKS:** 'del' may also be used to delete the content of individual strings and bstrings. Strings are set to null; the contents are released and removed from the garbage collector's lists.

**RELATED:** classes new
and
&
**RELATED:** and or xor operators

**REMARKS:** bitwise operator
| or |

**REMARKS:** bitwise operator

**RELATED:** and or= xor operators
xor

**REMARKS:** bitwise operator

**RELATED:** and or xor= operators
**ACTION:** return the integer value located by the address contained in the variable

**USE:** reading writing data, using pointers

**EXAMPLE:**

```plaintext
int a=42
print @a 'address of a
sys b 'sys' ensures an integer large enough to hold a
b=&a 'assign address of a to b
print *b '42
```

**REMARKS:** Unlike C, pointer resolution is normally handled implicitly.

**RELATED:** @ ? strptr addr
and=
&=

**REMARKS:** bitwise assign operator

**RELATED:** and or xor operators
\texttt{or=}
\texttt{|=}

\textbf{REMARKS:} bitwise assign operator

\textbf{RELATED:} \texttt{and or xor operators}
&&

**REMARKS:** logical operator

**RELATED:** && || ^^ operators
||

**REMARKS:** logical operator

**RELATED:** `& & || ^^` operators
^^

**REMARKS:** logical operator

**RELATED:** && || ^^ operators
&&=

**REMARKS:** logical assign operator

**RELATED:** && || ^^ operators
||=

**REMARKS:** logical assign operator

**RELATED:** && || ^^ operators
^^=

**REMARKS:** logical assign operator

**RELATED:** && || ^^ operators
**stdcall**

**ACTION:** determines how parameters are passed on the stack

**USE:** declaring external functions

**EXAMPLE:**

```c
! Sleep lib "kernel32.dll" stdcall (int msec)
```

**REMARKS:** this is the default calling convention on 32bit Windows platforms.

**RELATED:** calling cdecl ms64 pascal
cdecl

**ACTION:** determines how parameters are passed on the stack

**USE:** declaring external functions, and variadic functions

**REMARKS:** this is a common calling convention on 32bit platforms. The stack is cleaned up after the call by the caller.

**RELATED:** calling cdecl ms64 pascal callback
ms64

**ACTION:** determines how parameters are passed on the stack

**USE:** declaring external functions, and variadic functions

**REMARKS:** this is the default Windows calling convention on 64bit platforms.
  The first four parameters are passed in registers.
  The stack is cleaned up after the call by the caller.

**RELATED:** calling stdcall cdecl pascal
pascal

ACTION: determines how parameters are passed on the stack

USE: declaring external functions

EXAMPLE: ! Sleep lib "kernel32.dll" stdcall (int msec)

REMARKS: this is a legacy calling convention on 32bit platforms.

RELATED: calling stdcall cdecl ms64
space

ACTION:  returns a string of spaces

USE:  string=space length

EXAMPLE:  s=space 10

RESULT:  s="          ' 10 spaces (ascii 32)

REMARKS:  for wide strings use string(n,wchr 32) instead

RELATED:  nuls string ltrim rtrim
**nuls**

**ACTION:** returns the address of a string of null characters given the length.

**USE:** string=nuls length

**EXAMPLE:**

```plaintext
s=nuls 1000
```

**RESULT:**

`s` is a string of 1000 null characters

**REMARKS:** for wide strings use `string(n,wchr 0)` instead

**RELATED:** space string news frees getmemory
**asc**

**ACTION:** returns ascii encoding of a character in a string

**USE:** AsciiCode=asc(String,CharacterPosition)

**EXAMPLE:**

```plaintext
a=asc("ABCDEF",2)
```

**RESULT:**

```plaintext
a=66 'character='B'
```

**RELATED:** unic chr val len str
unic

**ACTION:** returns encoding of a character in a wide string

**USE:**
```
uni=unic(String,CharacterPosition)
```

**EXAMPLE:**
```
a=unic("ABCDEF",2)
```

**RESULT:**
```
a=0x0042 'character='B'
```

**RELATED:** asc wchr chr val len str
**redim**

**ACTION:** create or resize a dynamic array, preserving contents within range

**USE:** extend or reduce an array size at run-time

**EXAMPLE:**
redim string s(n)

**REMARKS:** to flush an array's contents, redim it with 0 elements first. But avoid doing this with arrays of strings; the orphaned strings are not garbage-collected until the end of the program, and will accumulate on each iteration where the redim reduces the number of elements.

**RELATED:** dim new
let

**ACTION:** defines a variable or object

**USE:** create a new variable or indirect object

**EXAMPLE:**

```javascript
let s="this is a string"
print typeof s 'result: 'string'
```

**REMARKS:** Similar to using 'dim' but the type is inferred from the assigned value.

**RELATED:** dim var
**var**

**ACTION:** define a set of variables  
**USE:** low-level dimensioning of variables  
**EXAMPLE:**  
```
var string s,t,u(64),v
```

**REMARKS:** var is normally only used internally. It accepts * for indirect variables  
and "at" for variable mapping. Arrays are also supported.

**RELATED:** dim
union
ACTION: define a union (C syntax)
USE: allows different variables to occupy the same space.
EXAMPLE:
union utype
{
    byte   b
    short  w
    long   i
}

RESULT: v.b=42 : v.w=42 : v.i=42

REMARKS: a union may also be nested inside a type.

RELATED: typedef type struct class enum
enum

**ACTION:** create an enumeration

**USE:** assign a numeric identity to a name

'------------
'ENUMERATIONS
'-----------

'SIMPLE ENUMERATION
'==================

```python
enum ManyThings
    shoes
    ships
    sealing_wax
    cabbages
    kings
end enum

def show "%1: " %1

print show cabbages
```

'_ENUMERATION FROM A BASE VALUE
'==============================

```python
enum ManyThings
    shoes=11
    ships
    sealing_wax
    cabbages
    kings
end enum

print show ships
```

'BITSWISE ENUMERATION
'=====================

'1 2 4 8 16 32 64 ...

```python
enum bit ManyThings
    shoes
```
ships
sealing_wax
cabbages
kings
end enum

'RENUMERATION USAGE
'--------------------

'Dim as ManyThings mt
ManyThings mt
mt=cabbages
print mt

REMARKS:  C syntax is supported for this construct. Also enum bit
 assigns values 1,2,4,8,16.. instead of 0,1,2,3,4..

RELATED:  typedef enum #define (%)
**embedfile**

**ACTION:** specify a file to be embedded in the data section

**USE:** store data file in a program, and get a pointer to it.

**EXAMPLE:**
```
sys dat : embedfile "data.txt",dat
```

**REMARKS:** Text or binary files may be embedded with this command. The data is read-only.
Either a sys or a bstring variable may be used to reference the data.
An upper limit of 255 files can be embedded.

**RELATED:** include strptr getfile
librarypath

**ACTION:** define a filepath for DLL files

**USE:** tells the computer where to find Dynamic Link Libraries required to run the program. (System DLLs do not require this.)

**RELATED:** library extern includepath
#ifdef

**ACTION:** include code if symbol already defined

**USE:** to allow blocks of code to be included or omitted at compile time

**EXAMPLE:**
```
#ifdef MSWIN
    #include "windows.inc"
#endif
```

**RELATED:** #if #elseif #endif
#ifndef

**ACTION:** include code if symbol already defined

**USE:** to allow blocks of code to be included or omitted at compile time

**EXAMPLE:**

    #ifdef MSWIN
        #include "windows.inc"
    #endif

**RELATED:** #if #elseif #endif
namespace

**ACTION:**  start a namespace

**USE:**  create a region where symbols locally defined

**EXAMPLE:**

```plaintext
int i=1

namespace aa
int i=10
namespace bb
int i=20
end namespace

print i + aa::i + bb::i '31

namespace bb
print i '20
end namespace
```

**RELATED:**  scope
**sin**

**ACTION:** returns the sine of a value given in radians

**USE:** Sine = sin(Radians)

**EXAMPLE:**

\[ v = \sin(\pi/6) \]

**RESULT:**

\[ v = 0.5 \]

**RELATED:** asin, cos, tan
**cos**

**ACTION:** returns cosine value (ratio of x/r) given angle in radians

**USE:** Cosine=cos(radians)

**EXAMPLE:**

\[ c = \cos(\pi/3) \]

**RESULT:**

\[ c = 0.5 \]

**RELATED:** sin tan
goto

**ACTION:** jump to a specified label in the code

**EXAMPLE:**

```plaintext
function cube(f as float) as float {return f*f*f}
```

**RELATED:** jmp gosub
end

**ACTION:** marks the end of a code block

**USE:** terminating various block structures

**EXAMPLE:**
```
if a>b then
  b=a
end if
```

**REMARKS:** An isolated 'end' terminates the program.

**RELATED:** block scope if do while select function sub method class macro def
loop
dend do, enddo

**ACTION:** end a **do** repeating block

**USE:** see: **do**

**RELATED:** loops **do** continue **exit**
break

**ACTION:** exit a switch block or do/while block

**RELATED:** case switch while do continue
end sel
end select

**ACTION:** end the select block

**RELATED:** selection select case
extends

**REMARKS:** 'of', and 'from' and extends are equivalent. They indicate derivation from a single parental class. COM interfaces us single inheritance

**RELATED:** has class classes
**external**

**REMARKS:** functions with the external attribute expect to be called from procedures external to Oxygen. Additional register management is enlisted to support external calls.

**RELATED:** extern com export callback com virtual function sub method
export

**REMARKS:** procedures with this attribute are visible to external callers when they are compiled as part of a DLL (Dynamic Link Library)

**RELATED:** external function sub extern
extern

**ACTION:** associate declared procedures with a calling convention and/or dll name

**EXAMPLE:**
```c
extern stdcall lib "kernel32.dll"
    declare function QueryPerformanceCounter(lpPerformanceCounter as quad)
    declare function QueryPerformanceFrequency(lpPerformanceFrequency as quad)
end extern
```

**REMARKS:** note there is no need to use an Alias in these declarations if you give the exact name of the procedures in their original case.

**RELATED:** lib library calling
xor=

**REMARKS:** bitwise assign operator

**RELATED:** and or xor operators
@  
**ACTION:** return the address of a variable  
**USE:** reading writing data of variables and arrays of variables, by reference  
**EXAMPLE:**

```
int a=42
print @a 'address of a
int *b 'indirect (pointer) variable
@b=@a 'coupled by address
print b '42
```

**REMARKS:** Unlike C, pointer resolution is handled implicitly. So the @ operator is required for manipulating pointers. It is similar to the & operator in C.

**RELATED:** * ? strptr addr
ACTION: return the integer value contained in the variable
USE: casting variables as integers
EXAMPLE: float f=100
        print hex(?f) 'display the hexadecimal form of float f

RELATED: cast * @ strptr addr
**strptr**

*ACTION:* return a string pointer

*USE:* obtain base address of string contents

*EXAMPLE:*

```plaintext
string s="Hello"
sys a=strptr s
```

*RELATED:* `typeof addr`
addr

**ACTION:** resolve address of a variable (assembler)

**USE:** load address of a variable to a register

**EXAMPLE:**
```
sys a
addr ecx, a
```

**RELATED:** @ * ? strptr
**ltrim**

**ACTION:** returns string with white space on the left trimmed off

**USE:** string=rtrim(string)

**EXAMPLE:**

s=ltrim "   abc"

**RESULT:**

s="abc"

**RELATED:** rtrim space str lcase ucase
**rtrim**

**ACTION:** returns string with white space on the right trimmed off

**USE:** string=rtrim(string)

**EXAMPLE:**

```plaintext
s=rtrim "abc   
```

**RESULT:**

```
s="abc"
```

**RELATED:** ltrim space lcase ucase
news

**ACTION:** allocates a bstring of null characters given the length in bytes

**USE:** bstring=news length

**EXAMPLE:**
```plaintext
b=news 1000
'...
frees b
```

**RESULT:** b contains address of a string of 1000 nul characters.

**REMARKS:** Oxygen strings are automatically freed but Bstrings must be explicitly freed when no longer needed.

**RELATED:** frees nuls space bstring getmemory
frees

**ACTION:** deallocates a bstring

**USE:** frees bstring freememory

**EXAMPLE:**
```
b=news 1000
'...
frees b
```

**RESULT:** b contains address of a string of 1000 nul characters.

**REMARKS:** Oxygen strings are automatically freed but Bstrings must be explicitly freed when no longer needed.

**RELATED:** news nuls freememory getmemory
getmemory

**ACTION:** allocate block of memory and return its base address

**USE:** address=getmemory bytes

**EXAMPLE:**

m=getmemory 8000

**RELATED:** freememory news frees
chr

**ACTION:** returns string of 1 character of ascii encoding (0..255)

**USE:**
```
string=chr(AsciiValue)
```

**EXAMPLE:**
```
s=chr(65)
```

**RESULT:**
```
s contains "A"
```

**RELATED:** wchr asc unic string
val
ACTION: returns numeric value of string
USE: value=val string
EXAMPLE: v=val "2.5"

RESULT: v=2.5

RELATED: tr hex asc
**len**

**ACTION:** returns length of string in characters

**USE:** length=len(string)

**EXAMPLE:**
\[
v=len("Hello")
\]

**RESULT:**
\[
v=5
\]

**REMARKS:** characters may be one or two bytes (wide characters)

**RELATED:** string space mid sizeof
str

**ACTION:** returns string representation of number

**USE:** String=str(value)

**EXAMPLE:**

```
s=str(-1.23456)  ' result: -1.23456
s=str(-1.23456,3)  ' result: -1.235
```

**REMARKS:** rounding is automatically applied before decimal places are truncated.

**RELATED:** hex val
wchr

**ACTION:** returns wide string of a 2 byte character (encoding 0..65535 / 0xffff)

**USE:** widestring=wchr(WideCharValue)

**EXAMPLE:**

wstring ws=wchr(65)

**RESULT:** ws contains unicode character 0x0041

**RELATED:** unic asc string
getfile

ACTION: copies file content to a string

USE: String=getfile FileName : getfile FileName,String

EXAMPLE: string s : getfile "t.txt",s

RESULT: s contains content of "t.txt"

RELATED: putfile print
library

**ACTION:** specify the name of a DLL library to associate with a set of procedure declarations

**EXAMPLE:**
```
lib "kernel32.dll"
```

**RELATED:** [declare](#)
#if

ACTION: include block of code if conditions are met

RELATED: #else #elseif #endif
#elseif

**ACTION:** include block of code if these alternative conditions are met

**RELATED:** #if #else #endif #fi
#endif

**ACTION:** end of conditional code inclusion block

**RELATED:** #if #elseif
**asin**

**ACTION:** returns angle in radians given the ratio \( y/radius \)

**USE:** angle=asin(YRRatio)

**EXAMPLE:**
\[
a=\text{asin}(0.5)
\]

**RESULT:**
\[
a=\pi/6
\]

**RELATED:** sin acos atan atan
\textbf{tan} \\
\textit{ACTION:} returns the tangent of a value given in radians \\
\textit{USE:} \( \text{tangent} = \text{tan}(\text{radians}) \) \\
\textit{EXAMPLE:} \( t = \text{tan}(a) \) \\
\textit{RESULT:} \( t = \text{ratio } y / x \) \\

\textit{RELATED:} \( \text{atan} \ \text{atn} \ \text{sin} \ \text{cos} \)
switch

**ACTION:** Start a Case block (C style)

**RELATED:** selection case endsel end
lib

**REMARKS:** lib may be specified in the Declare statement or it may be specified in an extern statement, referring to an entire group of declarations.

**RELATED:** library declare extern alias link at
cast

**ACTION:** change or specify the type of a variable temporarily, in an expression

**EXAMPLE:**

```plaintext
float f=100
print hex( cast(int) f) 'display the hexadecimal form of
print hex( (int) f )  'the same without using the 'cast
```

**RELATED:** convert ? union
**typecodeof**

**ACTION:**  return type code number of variables and literals.

**USE:**  obtain data for diagnostics or reflective programming

**RELATED:**  `typeof` `recordof` `sizeof` `offsetof` `spanof` `prototypeof` `#recordof`
lcase

**ACTION:** returns lowercase of string

**USE:**
```
string=lcase(string)
```

**EXAMPLE:**
```
s=lcase "ABCDEF"
```

**RESULT:**
```
s="abcdef"
```

**RELATED:** ucase ltrim rtrim asc
ucase

**ACTION:** returns uppercase of string

**USE:**

```plaintext
string=ucase(string)
```

**EXAMPLE:**

```plaintext
s=ucase "abcdef"
```

**RESULT:**

```plaintext
s="ABCDEF"
```

**RELATED:** lcase ltrim rtrim asc
freememory

ACTION: free previously allocated memory block

USE: freememory address

EXAMPLE: freememory m

RELATED: getmemory news frees
**hex**

**ACTION:** returns hexadecimal string representation of integer part of number

**USE:**

```
hexString=hex(value)
```

**EXAMPLE:**

```
print hex(14.4)    # result 'E'
print hex(14.4, 4) # result '000E'
```

**RELATED:** [str](#) [val](#)
**sizeof**

**ACTION:** return length of variable element (in bytes)

**USE:** nbytes = sizeof variable

**EXAMPLE:**

n = sizeof a

**RELATED:** offsetof spanof typeof typeidof
**putfile**

**ACTION:** saves a string to a file

**USE:** putfile FileName, String

**EXAMPLE:**

```
putfile "t.txt","Hello"
```

**RESULT:** A file named "t.txt" is created or overwritten containing

**RELATED:** getfile print
**print**

**ACTION:** Displays strings and numbers

**USE:** print String : print Number

**EXAMPLE:**
print "ABC" : print 123 : print "ABC: " 123 " DEF: " 456

**RESULT:**
ABC
123
ABC: 123 DEF: 456

**REMARKS:** strings can be combined with numbers as sys as the first element is a string

**RELATED:** val str hex putfile getfile
**declare**

**ACTION:** declare a procedure with its prototype (may be external or declared in advance)

**EXAMPLE:**
```
declare fun(float a,b,c,d) as float
declare function fun(float a,b,c,d) as float
```

**REMARKS:** There is a wide range of options for Declare statements. Please look at the examples and header files.

**RELATED:** ! library lib alias dim
#else

**ACTION:** alternative block of code to include if prior conditions are not met.

**RELATED:** #if #elseif #endif
acos

**ACTION:** returns angle in radians given the ratio x/radius

**USE:**

angle=acos(YRRatio)

**EXAMPLE:**

\( a = \text{acos}(0.5) \)

**RESULT:**

\( a = \frac{\pi}{3} \)

**RELATED:** cos asin atan atan
atn

**ACTION:** returns angle in radians given ratio y/x

**USE:** angle=atn(YXRatio)

**EXAMPLE:**
\[ a = \text{atn}(1) \times 4 \]

**RESULT:**
\[ a = \pi \]

**RELATED:** atan asin acos tan
atan

**ACTION:** returns angle in radians given the values of y and x

**USE:**

angle=atan(y,x)

**EXAMPLE:**

a=atan(0.5,sqr(0.75))

**RESULT:**

a=π/6

**RELATED:**  atn asin acos tan
alias

**REMARKS:** if the procedure name is an exact case match then it is not necessary to specify an Alias.

**RELATED:** library declare external library declare external
convert

ACTION: explicitly convert the type of an expression.

USE: pass values, in the required type, to an unprototyped function

EXAMPLE:
```plaintext
float f=2.5
sleep ( convert int ( f*1000 ) )
```

RELATED: cast
typeof
ACTION: return name of the variable type
USE: name=typeof variable
EXAMPLE: dim as long v : s=typeof v

RESULT: s="long"

RELATED: typeof codeof sizeof offsetof spanof recordof
recordof

**ACTION:** return record of compound (UDT) variable

**USE:** obtain data for diagnostics or reflective programming

**EXAMPLE:**
```plaintext
type vt long v, double d
dim as vt v : r=recordof v
```

**RESULT:**
```
r="
v 0 4 1 A0 , long
d 4 4 1 A0 , double
"
```

**REMARKS:** use #recordof to display the record during compilation.

**RELATED:** #recordof prototypeof sizeof offsetof spanof typeof typecodeof #recordof
offsetof

**ACTION:** return offset of variable from index register

**USE:** nbytes=offsetof variable

**EXAMPLE:**

n=offsetof a

**RELATED:** sizeof spanof typeof typecodeof recordof
spanof

**ACTION:** return span of array variable dimension

**USE:** nbytes=spanof variable dimension

**EXAMPLE:**
```
dim as long v(10) : n=spanof v
```

**RESULT:**
```
n=10
```

**RELATED:** sizeof offsetof typeof typecodeof recordof
prototypeof

**ACTION:** return prototype(s) of functions, subs and high level macros

**USE:** obtain data for diagnostics or reflective programming

**RELATED:** recordof sizeof offsetof spanof typeof typecoedof #recordof
#recordof

**ACTION:** return internal record of a declared entity.

**USE:** to check status at compile time.

**EXAMPLE:**

```
#recordof MyStructure
```

**REMARKS:** this command was introduced to aid debugging at compile time are often hard to trace.

**RELATED:** #blockdepth