

## Reference

**COLLABORATORS**

	<i>TITLE :</i> Reference		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		January 19, 2025	

**REVISION HISTORY**

NUMBER	DATE	DESCRIPTION	NAME

# Contents

<b>1</b>	<b>Reference</b>	<b>1</b>
1.1	Pure Basic Reference Manual . . . . .	1
1.2	editor . . . . .	2
1.3	Using the CLI Compiler . . . . .	3
1.4	general_rules . . . . .	4
1.5	variables . . . . .	6
1.6	For : Next . . . . .	6
1.7	gosub_return . . . . .	7
1.8	if_endif . . . . .	9
1.9	Repeat : Until . . . . .	9
1.10	Select : EndSelect . . . . .	10
1.11	While : Wend . . . . .	11
1.12	others . . . . .	12
1.13	deftype . . . . .	12
1.14	dim . . . . .	13
1.15	NewList . . . . .	13
1.16	structures . . . . .	14
1.17	global . . . . .	14
1.18	shared . . . . .	15
1.19	procedures . . . . .	15
1.20	includes . . . . .	16
1.21	debugger . . . . .	17

---

# Chapter 1

## Reference

### 1.1 Pure Basic Reference Manual

```

*****
*                                                                 *
*          PureBasic Reference Manual V1.30                      *
*                                                                 *
*          © 2000 - Fantaisie Software -                        *
*                                                                 *
*****

          General Topics:          External Libraries:

Using the Editor          Amiga
Using the CLI compiler    App
General Syntax Rules      BitMap
Variables and Types       Chunky
Basic Keywords:          @{ "" LINK k          } Clipboard
                          @{ "" LINK k          } Commodity
                          @{ "" LINK k          } Drawing
                          Font
                          File
                          Gadget
                          Linked List
                          Menu
                          Misc
                          OS
                          @{ "" LINK k          } Palette
Structure Options:      @{ "" LINK k          } Picture
                          @{ "" LINK k          } Requester
                          DefType
                          Dim
                          NewList
                          Sound
                          String
                          @{ "" LINK k          } TagList
Procedure Support:      @{ "" LINK k          } ToolType
                          @{ "" LINK k          } WbStartup
                          Global
                          Window
                          Procedure: EndProcedure
                          Shared
                          Commands index:

```

```

Compiler Options:      @{ "" LINK k          }
                       @{ "" LINK k      }      Index
                       @{ "" LINK k          }
'Include' Functions
Debugger

```

## 1.2 editor

### Introduction:

The PureBasic editor has been created especially for the PureBasic programming language and has many special features especially designed for it. It will become more and more powerful and will support advanced editing like coloring syntax, word filling, online help...

### Basic usage:

The PureBasic editor accept any standard ASCII characters, and load and save the file in the ASCII format. It use all Amiga standard short cut to edit the text:

```

Arrows Keys          : Move the cursor in the four ways
Shift + Arrow Up     : One page up
Shift + Arrow Down   : One page down
Shift + Arrow Left   : Start of the line
Shift + Arrow Right  : End of the line

Shift + Enter        : Insert a line above the current line
Shift + Del          : Delete the right part of the line
Shift + Backspace    : Delete the left part of the line

```

Help: Bring up the PureBasic manual online help (this document)

Now, the most important menus shortcuts:

```

AmigaRight + S: Save the current source code without any request
AmigaRight + Q: Quit the PureBasic
AmigaRight + L: Load a new source code

```

### Special features:

There is a menu called 'Compiler' and it's by this way you could control the PureBasic. Menu items:

- \* Compile/Run: Compile the actual source code and run it.
- \* Debugger: Switch which toggle ON/OFF the debugger
- \* Options:
  - Output processor: 680x0/PowerPC. Change the format of the generated executable. The PowerPC version is for WarpOS.
  - Code optimizations: Turn on the optimizations while compiling in the editor, to have exactly the same executable than the

final one. Of course the debugger must be turned off, else this option is ignored. Better to turn it off while developping, as it increase the compile time...

- Commented ASM output files: Generate a commented asm file when you create a final executable. The file is located at "PureBasic:Compilers/PureBasic.asm". You can modify this file (optimize it) and recompile it with phxass. It slow down a lot the executable generation so use it only when you need it.
  - Disable CLI output: Don't display the cli output window, useful if your program print nothing to the cli.
  - Create an icon: An icon will be added to the created executable The icon is located at "PureBasic:Compilers/Default\_Icon.info" You can replace it by another one if you want
  - Save: Save theses preferences for the actual file. Each file can have their own preferences.
- \* CreateExecutable: Create a final executable. The optimizations are of course turned on automatically. It use the options defined as described above.

Other options like InsertFile, Print, Find, are classic one like on any other editors...

## 1.3 Using the CLI Compiler

Using the CLI compiler:

Type "PureBasic" followed by the source filename to compile. PureBasic will compile and launch the programme.

Compiler options:

FILE

String: This needs a source file name! This argument is needed or the compiler will generate an error.

TO

String: If specified, you must add the destination path, and the filename, to show where the executable must be created. Note: Only the executable is created in this case. The programme is not started.

NR or NORESIDENT

Switch: If this is set, it will not load the AmigaOS resident file. By default the compiler will load this file, thus increasing the compilation time.

PPC or POWERPC

---

Switch: If this is set, the compiler will generate an Amiga PowerPC executable for WarpOS. For the present, the result will read as an error. It can be tested ←

Please record the asm file, and the generated code, then send us the result! All data will be recorded and analysed in our continuous attempts to improve this option. Thank you for your assistance. :)

#### NC or NOCOMMENT

Switch: If this is set, it will produce a non-commented asm output, which is smaller and faster to assemble. This will decrease the compilation time.

#### PRI or PRIORITY

Numeric: A numeric value between -127 and +127, is required. It will determine the priority of the compiler. Example: PureBasic PRI=10 .. will give almost ←  
all  
of the cpu time to the compiler.

#### CR or CREATERESIDENT

Switch: This will compile the programme, and create a resident file with all structures and constants. The compiled file is located in "Ram:ResidentFile" and "Ram:ResidentFile.struct"

#### STANDBY

Switch: If this is set, the compiler is put into "sleep mode" and waits for on order through its message port. Please do not use it yet, as it is for use with the forthcoming editor.

#### DB or DEBUGGER

Switch: If this is set, it will compile the programme with debugger support. Shortly the debugger can be used to interrupt the programme. Please use it carefully and run it step by step...

#### OPT or OPTIMIZATIONS

Switch: If this is set, it will enable maximum optimization, and generate fast and small executables.

#### Examples:

```
PureBasic Sources:MypPog.pb DB PRI=10
```

```
PureBasic Sources:Example.pb TO Ram:Example.exe OPT PRI=10
```

## 1.4 general\_rules

### General Rules

PureBasic has established rules which never change.

---

These are:-----

- \* Comments are marked by `;` . All text entered after `;` is ignored by the compiler. ↵

Example:

```
If a = 10; This is a comment to indicate something.
```

- \* All functions must be followed by `(` or else it will not be considered as a function, (even for null parameter functions).

Example: `WindowID()` is a function.  
`WindowID` is a variable.

- \* All constants are preceded by `#`

Example:

```
#Hello = 10 is a constant.
Hello = 10 is a variable.
```

- \* All labels must be followed by `:`

Example:

```
I_am_a_label:
```

- \* An expression is something which can be evaluated. An expression can mix any variables, constants, or functions, of the same type.

Examples of valid expressions:

```
a+1+(12*3)
a+WindowHeight()+b/2+#MyConstant
a <> 12+2
b+2 >= c+3
```

- \* Any number of commands can be added to the same line by using the `:` option.

Example:

```
If OpenScreen(0,320,200,8,0) : PrintN("Ok") : Else : PrintN("Failed") : ↵
  EndIf
```

- \* Words used in this guide:

```
<variable> : a basic variable.
<expression>: an expression as explained above.
<constant> : a numeric constant.
<label> : a programme label.
<type> : any type, (standard or structured).
```

- \* In this guide, all topics are listed in alphabetical order to decrease any search time.

## 1.5 variables

Variables declaration:

To declare a variable in PureBasic type its name, or the type you want this variable to be. Variables do not need to be explicitly declared, as they can be used as "variables on-the-fly."

The "DefType" keyword can be used to declare mass variables.

Example:

```
a.b ; Declare some variables.  
c.l ;
```

```
c = a*d.w ; "d" is declared here within the expression!
```

To use a pointer, put \* before the variable name. A pointer is a long variable which stores an address. It is generally associated with a structured type. So, you can access the structure via the pointer.

Example:

```
*MyScreen.Screen = OpenScreen(0,320,200,8,0)  
  
mouseX = *MyScreen\MouseX
```

Basic types

PureBasic allows type variables. It now supports signed variables. Unsigned variables can be used, but this can result in an error, as this option is still in it's early stages.

Types:

Byte: .b, take 1 byte in memory. Range: -128 to +127.

Word: .w, take 2 bytes in memory. Range: -32768 to +32767

Long: .l, take 4 bytes in memory. Range: -2147483648 to +2147483647

Unsigned Byte: .ub, take 1 byte in memory. Range: 0 to 255

Unsigned Word: .uw, take 2 bytes in memory. Range: 0 to 65535

Unsigned Long: .ul, take 4 bytes in memory. Range: 0 to 4294967295

String: .s, take the string length into memory.

Structured types

Build structured types, via the Structures option. More information can be located in the "structures chapter."

## 1.6 For : Next

---

Syntax:

```
For <variable> = <expression1> To <expression2> [Step <constant>]
    ... Loop content
Next [<variable>]
```

Description:

The "For/Next" function is used to cause a loop within a programme within given parameters. At each loop the <variable> is increased by a factor of 1, (or of the "Step value" if a Step value is specified), when the <variable> value equals the <expression2> loop stop.

Example 1:

```
For k=0 To 10
    ...
Next
```

In this example, the programme will loop 11, time (0 to 10), then quit.

Example 2:

```
a = 2
b = 3

For k=a+2 To b+7 Step 2
    ...
Next k
```

Here, the programme will loop 4 times before quitting, (k is increased by a value of 2 at each loop, so the k value is: 4-6-8-10). The "k" after the "Next" indicates that "Next" is ending the "For k" loop. If another variable ←  
,  
is used the compiler will generate an error. It is useful when nesting some "For/Next" expressions.

Example 3:

```
For x=0 To 320
    For y=0 To 200
        Plot(x,y)
    Next y
Next x
```

## 1.7 gosub\_return

Syntax:

```
Gosub <label>
```

---

```
<label>:  
    ... Sub routine code  
  
Return
```

**Description:**

"Gosub" stands for "Go to sub routine." A label has to be specified after "Gosub" then the programme will continue at the label position until it encounters a "Return." When a return is reached, the programme is transferred below the Gosub.

"Gosub" is very useful when building fast structured code.

**Example:**

```
a = 1  
b = 2  
  
Gosub ComplexOperation  
  
PrintNum(a)  
End  
  
ComplexOperation:  
  
    a=b*2+a*3+(a+b)  
    a=a+a*a  
  
Return
```

**Syntax:**

```
FakeReturn
```

**Description:**

When you want to jump from a sub routine (with the command 'Goto') to another part in the code outside of this sub routine, you need to use a FakeReturn which simulate a return without do it really. If you don't use it, your program will crash.

This function should be useless because a well constructed program don't use Goto. But sometimes, for speed reason, it could help a bit.

**Example:**

```
Main_Loop:  
    ...
```

```
SubRoutine1:
  ...
  If a = 10
    FakeReturn
    Goto Main_Loop
  Endif

Return
```

## 1.8 if\_endif

Syntax:

```
If <expression>
  ...
[Else]
  ...
EndIf
```

Description:

The "If" structure is used to achieve tests, and/or change the programmes direction, if the test is true or false. The "Else" optional command is used to execute a part of code, if the test is false.

Any number of "If" structures can be nested together.

Example 1:

```
If a=10
  Nprint ("a=10")
Else
  Nprint ("a<>10")
EndIf
```

Example 2:

```
If a=10 and b>=10 or c=20
  If b=15
    nprint("ok")
  Else
    nprint("ok2")
  Endif
Else
  nprint("test failure")
Endif
```

## 1.9 Repeat : Until

---

Syntax:

```
Repeat
    ... Programme ...

Until <expression>
[or Forever]
```

Description:

This function will loop until the <expression> becomes true. Any number can be repeated. If an endless loop is needed then use the "Forever" keyword instead of "Until."

Example:

```
a=0
Repeat
    a=a+1
Until a>100
```

This will loop until "a" takes a value > to 100, (it will loop 101 times).

## 1.10 Select : EndSelect

Syntax:

```
Select <expression1>

Case <expression2>
    ...Code...

[Case <expression3>....]
    ...Code...

[Default]
    ...Code...

EndSelect
```

Description:

"Select" allows a quick choice. The programme will execute the <expression1> and keep its value in memory. It will compare this value to all of the "Case <expression> values," and if true it will execute the corresponding code and quit the "Select" structure. If none of the "Case" values are true, then the Default code, (if specified), will be executed.

---

Example:

```
a = 2

Select a

  Case 1
    PrintN("Case a = 1")

  Case 2
    PrintN("Case a = 2")

  Case 20
    PrintN("Case a = 20")

  Default
    PrintN("I don't know")

End Select
```

Syntax:

```
FakeEndSelect
```

Description:

When you want to jump from a select part (with the command 'Goto') to another part in the code outside of the Select, you need to use a FakeEndSelect which simulate an EndSelect without do it really. If you don't use it, your program will crash.

Example:

```
Main_Loop:
  ...
  Select a

    Case 10
      ...

    Case 20
      FakeEndSelect
      Goto Main_Loop

  EndSelect
```

## 1.11 While : Wend

Syntax:

```
While <expression>
```

---

```
... Programme ..
```

```
Wend
```

Description:

"Wend" will loop until the <expression> becomes false. A good point to keep in mind with a "While" test is that if the first test is false, then the ←  
programme  
will never enter the loop and will skip this part. A "Repeat" loop is executed at least once, (as the test is performed after each loop).

Example:

```
b = 0
a = 10
While a = 10
  b = b+1
  If b=10
    a=11
  Endif
Wend
```

This programme loops until the "a" value is <> 10. A change here when b=10, the programme will loop 10 time.

## 1.12 others

A list of other commands:

Goto

```
Goto <label>
```

This command is used to transfer the programme directly to the labels position ←

Be cautious when using this function, as incorrect use could cause a programme to crash...

## 1.13 deftype

Syntax:

```
DefType.<type> [<variable>, <variable>, ...]
```

Description:

If no <variables> are specified, "DefType" is used to change the "Default type ←  
"

for future untyped variables.

Example:

```
DefType.l
```

```
a = b+c
```

a, b and c will be signed long typed (.l) as no type is specified.

If variables are specified, "DefType" only declares these variables as "defined type" and will not change the default type.

Example:

```
DefType.b a,b,c,d
```

a,b,c,d will be signed byte typed (.b)

## 1.14 dim

Syntax:

```
Dim name.<type>(<expression>)
```

Description:

"Dim" is used to "size" the new arrays. An array in PureBasic can be of any types, including structured, and user defined types. Once an array is "dim" it cannot change it's time and another array cannot be classed as "dim" with the same name.

Example:

```
Dim MyArray.l(41)
```

```
MyArray(0) = 1
```

```
MyArray(1) = 2
```

## 1.15 NewList

Syntax:

```
NewList name.<type>()
```

Description:

"NewList" allows managed dynamic linked lists in PureBasic. Each element of

---

the list is allocated dynamically. There are no element limits, so there can be as many as needed. A list can have any standard or structured type.

To view all commands used to manage lists, please click [here](#)

Example:

```
NewList mylist.l()

AddElem(mylist())

mylist() = 10
```

## 1.16 structures

Syntax:

```
Structure <name of structure>

... Structure content

EndStructure
```

Description:

"Structure" is useful to define user type, and access some OS memory areas. Structures can be used to enable faster and easier handling of big data files. Structures are accessed with the "\" option. Structures can be nested.

Example:

```
Structure Info
  Name.s
  ForName.s
  Age.l
  Birthday.l
EndStructure

Dim myfriends.Info(100)

myfriends(0)\Name = "Andersson"
myfriends(0)\Forname = "Richard"
...
```

## 1.17 global

Syntax:

```
Global <variable> [,<variable>,...]
```

---

Description:

"Global" allows the variables to be used as Global, ie: they can be accessed inside a procedure.

Example:

```
Global a.l, b.b, c, d
```

## 1.18 shared

Syntax:

```
Shared <variable> [, <variable>, ...]
```

Description:

"Shared" allows a variable to share, or to be accessed, within a procedure.

Example:

```
a.l = 10
```

```
Procedure myproc()  
  Shared a
```

```
  a = 20
```

```
EndProcedure
```

```
myproc()
```

```
PrintN(Str(a)) ; Will print 20, as the variable has been shared.
```

## 1.19 procedures

Syntax:

```
Procedure[.<type>] name(<variable1>[, <variable2>, ...])
```

```
  ... Procedure code
```

```
  [ProcedureReturn value]
```

```
EndProcedure
```

---

**Description:**

A "Procedure" is a part of code independent from the main code which can have any parameters and it's own variables. In PureBasic, a recurrence is fully supported for the "Procedures" and any procedure can call it itself. To access main code variables, they have to be shared them by using "Shared" or "Global" keywords.

A procedure can return a result if necessary. You have to set the type after 'Procedure' and use the 'ProcedureReturn' keyword at any moment inside the Procedure.

**Example:**

```
Procedure.l Maximum(nb1.l, nb2.l)

    If nb1>nb2
        Result = nb1
    Else
        Result = nb2
    Endif

    ProcedureReturn Result

EndProcedure

Result.l = Maximum(15,30)

PrintNumberN(Result)

End
```

## 1.20 includes

**Syntax:**

```
IncludeFile "filename"
XIncludeFile "filename"
```

**Description:**

"IncludeFile" will include any named source file, at the current place in the code. "XIncludeFile" is exactly the same except it avoids having to include the same file many times.

**Example:**

```
XInclude "Sources:myfile.pb" ; This will be inserted.
XInclude "Sources:myfile.pb" ; This will be ignored along with all ↵
    subsequent
calls..
```

Syntax:

```
IncludeBinary "filename"
```

Description:

"IncludeBinary" will include the named file at the current place in the code.

Example:

```
IncludeBinary "Sources:myfile.data"
```

Syntax:

```
IncludePath "path"
```

Description:

"IncludePath" will specify a default path for all files included after the call of this command. This can be very handy when you include many files which are in the same directory:

Example:

```
IncludePath "Sources:Data/"  
  
IncludeFile "Sprite.pb"  
XIncludeFile "Music.pb"  
...
```

## 1.21 debugger

The PureBasic Debugger

The debugger is an external program which can control the execution of a programme. The provided debugger is limited and has few functions. ↩

Nevertheless

it is enough to debug a programme correctly. It will be regularly updated and bettered. If anyone wants to do their own debug utility, please contact us. ↩

The

debugger is 100% OS friendly and does not use interrupts or trap vectors.

A programme's execution can be stopped, and an analysis made to locate any faults! This can be very useful in case a programme falls into an endless loop ↩

.

Functions:

```
Stop
```

---

This will halt the execution then display the current code position.

Cont

This will continue a previously stopped programme.

Step

This button allows code to be inserted step by step, ie: line after line. It is very handy to locate any faults. ←

Trace

This button allows the user to read the code as the programme lines are displayed.

Exit

Exit: This quits the debugger; the compiler; and any programme in case of any problems or if an "endless loop" cannot be stopped in any other way.

The debugger's keywords in PureBasic:

STOP:

Stop: This invokes the "debugger" and freezes the programme immediately.

Example:

```
If a=10
  Stop ; The debugger will be invoked.
Else
  Ok=1
Endif
```