# Reference

**COLLABORATORS**

| | *TITLE* :<br><br>Reference | |
|---|---|---|

| *ACTION* | *NAME* | *DATE* | *SIGNATURE* |
|---|---|---|---|
| WRITTEN BY | | July 31, 2024 | |

**REVISION HISTORY**

| NUMBER | DATE | DESCRIPTION | NAME |
|---|---|---|---|
| | | | |

# Contents

# Chapter 1

# Reference

## 1.1 Pure Basic Reference Manual

```
 ************************************************************
*                                                            *
*              PureBasic Reference Manual V2.32              *
*                                                            *
*                © 2001 - Fantaisie Software -               *
*                                                            *
************************************************************


      General Topics:

 Using the Editor
 Using the CLI compiler
 General Syntax Rules
 Variables and Types

Basic Keywords:

 For: Next
 Gosub: Return
 If: EndIf
 Repeat: Until
 Select: EndSelect
 While: Wend
 Others

Structure Options:

 DefType
 Dim
 NewList
 Structure: EndStructure

Procedure Support:

 Global
 Procedure: EndProcedure
 Shared
```

Various topics:



External Libraries:



Commands index:

## 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 accepts any standard ASCII characters, and loads and saves the file in the ASCII format. It uses all Amiga standard short cuts to edit the text:

```
Arrows Keys        : Move the cursor in the four directions
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 editor
AmigaRight + L: Load a new source code
```

You can use the Cut/Copy/Paste feature by dragging the mouse on a rectangular area around the text you want to select. The text is copied to the Amiga shared clipboard, so you can use this copied text in another application.

This editor has an auto-indent function which always keeps the cursor in the currently indented block, providing an easy source code editing.

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.
* Run: Run the previously compiled program without recompiling it.
* Debugger: A switch to toggle ON/OFF the debugger

* Structure Viewer: Display the full list of the structures which are available. This includes the AmigaOS strandard structures (loaded from the residents files) and your own structures. You can walk very

```
    easily through the structure by clicking on an entry or typing the
    structure name directly. Warning: structure names are case sensitive.
    The 'Parent' button is useful to get back to the previous structure.

  * Options:

    - Output processor: 68000/68020+. Change the format of the
      generated executable.

    - 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 turn it off while developping,
      as it increases 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 will slow down
      the executable generation a lot, so use it only when you need it.

    - Disable CLI output: Don't display the cli output window, useful
      if your program prints 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 the preferences for the actual file. Each file can
      have its own preferences.

  * CreateExecutable: Create a final executable. The optimizations are
    of course turned on automatically. It uses the options defined as
    described above.

  * Help on a keyword: display the documentation help topic about this
    command.


  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.

MC68020

```
  Switch: If this is set, it will enable the compilation for 68020+ processors
          and use special optimizations (to produce smaller and faster code)

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 will change.
These are:------

* Comments are marked by  ; . All text in the line 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.
To get access to a string or an array, use @ instaed of *.

Examples:

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

mouseX = *MyScreen\MouseX

Title.s = "Demo Window Example"
ResetTagList(#WA_Title , @Title)

Dim Hello.l(100)
ArrayHello.l = @Hello()
```

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, takes 1 byte in memory. Range: -128 to +127.
Word: .w, takes 2 bytes in memory. Range: -32768 to +32767
Long: .l, takes 4 bytes in memory. Range: -2147483648 to +2147483647

Unsigned Byte: .ub, takes 1 byte in memory. Range: 0 to 255
Unsigned Word: .uw, takes 2 bytes in memory. Range: 0 to 65535
Unsigned Long: .ul, takes 4 bytes in memory. Range: 0 to 4294967295

String: .s, takes as many bytes in memory as the string's length is.

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 with the given parameters. At each loop pass 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> the loop will be exited.

Example 1:

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

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

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. This is useful if 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 subroutine will be
  exited and the programme flow continues with the command that follows
  the Gosub call.

  "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:

```
  If 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 simulates a return without doing it really.
  If you don't use it, your program will crash.

  This function should be useless because a well constructed program
  don't need to use Goto's. 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 <expression2>"... "Case <expression3>"... values, and if one comes true it will execute the corresponding code and exit 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:

If 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 simulates an EndSelect without doing 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:

  "While" ... "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
  loop will never be entered and the programme 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 times.
```

## 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>[,<expression2>,..])

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 size and an another array cannot be classed as "dim" with the same name.

Example 1:

```
Dim MyArray.l(41)

MyArray(0) = 1
MyArray(1) = 2
```

Multi-dimensionned arrays are fully supported.

Example 2:

```
Dim ScreenBuffer.b(320,200)

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

## 1.15 NewList

Syntax:

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

Description:

"NewList" allows management of 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. Linked lists are always one-dimensioned. Structures can be attached by the \ option, see Structures .

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"
    ...


There is a possible way to share a memory emplacement inside structure. This
is known as 'Union' in the C/C++ langage. PureBasic supports fully the unions
to provide a complete support of the AmigaOS but it's not very recommended to
use them. It's complex and could generate many doubtful errors. There are the
'StructureUnion' and 'EndStructureUnion' keywords.

Example:

  Structure.person

    StructureUnion
      *FirstName.l      ;  These both names are exactly the same. They
      *AlternateName.l  ;  point to the same data, only the name is
    EndStructureUnion   ;  different

  EndStructure
```

## 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 varaible 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 position 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 if 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
  regulary updated and enhanced. 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:

  CallDebugger:

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

  Example:

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


  DisableDebugger:

    This will disable the debugger until EnableDebugger is called.


  EnableDebugger:

    This will enabled the debugger after a DisableDebugger call.
```

## 1.22 inlinedasm

```
  PureBasic allows you to include any 680x0 assembler commands
directly in the source code, as if it was a real assembler. And
it gives you even more: you can use directly any variables or
pointers in the assembler keywords, you can put any ASM commands
on the same line, ... All the assembler keywords are supported
from 68000 to 68060. Click  here  to have the full list
```

of legal ASM keyword and a quick description of them (extract of the PhxAss guide). Read the PhxAss guide to get more informations about their use...

You have to closely follow several rules if you want to include asm in a Basic code:

+ You must turn off the debugger when using the inlined ASM. Better to use the 'DisableDebugger' and 'EnableDebugger' command.

+ The used variables or pointers must be declared before to use them in an assembler keyword.

+ If you reference a label, you must put the letter 'p' before the name. This is because PureBasic adds a 'p' before a BASIC label to avoid conflicts with internal labels.

  Example:

```
LEA.l pMyLabel(pc),a0
...
MyLabel:
```

+ The errors in an asm part are not reported by PureBasic but by PhxAss. Just check your code if such an error happens.

+ The registers a2,a3,a4 must be always preserved. All others are free to use.

Example:

Inlined ASM example

## 1.23 internalindex

```
 ************************************************************
*                                                            *
*            PureBasic Internal Commands Index               *
*                                                            *
*              © 2001 - Fantaisie Software -                 *
*                                                            *
************************************************************
```

| Commands Index: | Area: |
|---|---|
| # | General Syntax Rules |
| ( | General Syntax Rules |
| ) | General Syntax Rules |
| * | Variables |
| .b | Variables |
| .l | Variables |

| | |
|---|---|
| .s | Variables |
| .ub | Variables |
| .ul | Variables |
| .uw | Variables |
| .w | Variables |
| : | General Syntax Rules |
| ; | General Syntax Rules |
| @ | Variables |
| \ | Structures |
| Byte | Variables |
| CallDebugger | Debugger |
| Case | Select: EndSelect |
| Default | Select: EndSelect |
| DefType | DefType |
| Dim | Dim |
| Else | If: Endif |
| EndIf | If: Endif |
| EndProcedure | Procedures |
| EndSelect | Select: EndSelect |
| EndStructure | Structures |
| EndStructureUnion | Structures |
| FakeEndSelect | Select: EndSelect |
| FakeReturn | Gosub: Return |
| For | For: Next |
| Forever | Repeat: Until |
| Global | Variables |
| Gosub | Gosub: Return |
| Goto | Others |
| If | If: Endif |
| IncludeBinary | Includes |
| IncludeFile | Includes |
| IncludePath | Includes |
| Long | Variables |
| NewList | LinkedLists |
| Next | For: Next |
| Procedure | Procedures |
| ProcedureReturn | Procedures |
| Repeat | Repeat: Until |
| Return | Gosub: Return |
| Select | Select: EndSelect |
| Shared | Variables |
| Step | For: Next |
| String | Variables |
| Structure | Structures |
| StructureUnion | Structures |
| To | For: Next |
| Until | Repeat: Until |
| Wend | While: Wend |
| While | While: Wend |
| Word | Variables |
| XIncludeFile | Includes |