

Reference

COLLABORATORS

	<i>TITLE :</i> Reference		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		January 23, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Reference	1
1.1	PureBasic Reference Manual	1
1.2	editor	2
1.3	Utilisation du compilateur	3
1.4	Règles générales	5
1.5	Information sur les variables	6
1.6	For : Next	7
1.7	Gosub : Return	8
1.8	If : EndIf	9
1.9	Repeat : Until	10
1.10	Select : EndSelect	11
1.11	While : Wend	12
1.12	Divers	13
1.13	deftype	13
1.14	dim	14
1.15	newlist	14
1.16	structures	15
1.17	global	16
1.18	shared	16
1.19	procedures	17
1.20	includes	18
1.21	debugger	19
1.22	inlinedasm	20
1.23	Commandes Internes	21

Chapter 1

Reference

1.1 PureBasic Reference Manual

```
*****
*
*      PureBasic: Manuel de référence V1.50
*
*      © 2000 - Fantaisie Software
*
*****
```

Informations générales:

Utiliser l'éditeur
Utiliser le compilateur
Règles de syntaxe
Variables et Types

Mots clefs du Basic:

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

Structures et définition:

DefType
Dim
NewList
Structure: EndStructure

Procédures:

Global
Procedure: EndProcedure
Shared

Bibliothèques externes:

Amiga	
App	
BitMap	
Chunky	
@{ "" LINK k }	Clipboard
@{ "" LINK k }	Commodity
@{ "" LINK k }	Drawing
Font	
File	
Gadget	
Linked List	
Menu	
Misc	
OS	
@{ "" LINK k }	Palette
@{ "" LINK k }	Picture
@{ "" LINK k }	Popup Menu
Requester	
Screen	
Sort	
Sound	
@{ "" LINK k }	String
@{ "" LINK k }	TagList
@{ "" LINK k }	ToolType
WbStartup	
Window	

```

@{ "" LINK k                }    Liste ←
    récapitulative:
Fonctions diverses:    @{ "" LINK k        }
@{ "" LINK k                }    Commandes ←
    externes
    Fonctions 'Include'      Commandes internes
    Asm 680x0 en ligne       Toutes Commandes
    Debugger
@{ "" LINK k                }

```

1.2 editor

Introduction:

L'éditeur du PureBasic a été spécialement conçu pour s'intégrer parfaitement à ce langage de programmation. Ainsi la compilation est plus rapide et plus facile qu'en ligne de commande. Il a déjà toutes les fonctions de base pour gérer le PureBasic, et bientôt viendront se greffer des fonctions supplémentaires comme la syntaxe colorée (en fonctions des mots clefs, des constantes...), l'aide en ligne pour chaque commande...

Utilisation:

L'éditeur PureBasic accepte n'importe quel caractère ASCII et les fichiers sauvegardés sont au format ASCII. Les raccourcis claviers standards sur Amiga sont utilisés:

```

Les flèches      : Déplace le curseur dans les quatres directions
Shift + Haut     : Page précédente
Shift + Bas      : Page suivante
Shift + Gauche   : Début de la ligne
Shift + Droite   : Fin de la ligne

Shift + Entrée    : Insertion d'une ligne au dessus de la ligne courante
Shift + Del       : Efface la partie droite de la ligne (par rapport au ←
                    curseur)
Shift + Backspace: Efface la partie gauche de la ligne (par rapport au ←
                    curseur)

```

Help: Affiche le fichier d'aide général du PureBasic (ce fichier)

Raccourcis des menus les plus importants:

```

AmigaDroite + S: Sauvegarde le code source courant
AmigaDroite + Q: Quitte le PureBasic
AmigaDroite + L: Charge un nouveau source code à partir d'un fichier

```

Vous pouvez utiliser le couper/copier/coller à la manière des autres éditeurs standards. Le texte est alors stocké dans le presse-papier de l'Amiga, et n'importe quelle application peut y avoir accès.

Cette éditeur a une fonction d'auto-indentation qui permet d'écrire un code lisible (correctement indenté) avec une facilité déconcertante.

Caractéristiques particulières:

Il y a un menu nommé 'Compilateur' et c'est par ce biais que vous allez contrôler le PureBasic.

- * Compiler/Executer: Compile le code source actuel et l'exécute.
- * Debugger: Active ou désactive le débogueur lors de l'exécution.
- * Options:
 - Compiler pour: 680x0/PowerPC. Permet de choisir le format de l'exécutable généré. Le format WarpOS est utilisé pour la version PowerPC.
 - Optimiser le code: Active les optimisations de l'exécutable lors de la compilation via l'éditeur pour avoir exactement le même fichier que lorsque l'on crée un exécutable final. Le débogueur doit être désactivé, sinon cette fonction est ignorée. La compilation est ralentie lorsque cette option est activée.
 - Fichier ASM commenté: Ajoute des commentaires (les lignes du programme) au fichier assembleur généré par le compilateur lors de la création d'un exécutable final. Ce fichier est localisé à l'endroit suivant: "PureBasic:Compilers/PureBasic.asm". Vous pouvez ainsi le modifier et le recompiler par la suite avec vos propres optimisations.
 - Disable CLI output: Empêche l'apparition de la fenêtre CLI, ce qui peut être pratique si votre programme n'utilise pas la ligne de commande.
 - Créer une icône: Lors de la création de l'exécutable final, une icône sera également créée avec le programme. L'icône utilisée est située à l'endroit suivant: "PureBasic:Compilers/Default_Icon.info". Vous pouvez la remplacer par celle de votre choix si nécessaire.
 - Sauver: Enregistre les préférences pour le fichier actuel. Chaque fichier peut avoir sa propre configuration.
- * CreateExecutable: Crée un exécutable final. Toutes les optimisations possibles sont activées automatiquement.

1.3 Utilisation du compilateur

Comment utiliser le compilateur du PureBasic:

Pour un résultat immédiat, tapez uniquement le mot PureBasic suivi du nom du fichier à compiler et le programme va s'exécuter automatiquement.

Voyons en détail les options du compilateur:

FILE

Texte: Nom du fichier source à compiler. Cet argument est obligatoire.

TO

Texte: Permet de générer un fichier exécutable à l'endroit donné (chemin + nom de l'exécutable). Seulement l'exécutable est généré, le programme n'est pas lancé.

NR or NORESIDENT

Switch: Permet de ne pas charger le fichier résident à la compilation pour gagner un peu de temps. Le fichier résident contient la déclaration d'une partie des structures et des constantes de l'AmigaOS.

PPC or POWERPC

Switch: Permet de produire un exécutable PowerPC (à la norme WarpOS). Cette option ne fonctionne pas entièrement pour l'instant. Mais si vous vous sentez capable de trouver les erreurs dans le code assembleur PowerPC généré, allez voir dans le fichier 'PureBasic:Compilers/PureBasic.asm'.

NC or NOCOMMENT

Switch: Permet d'enlever les commentaires du fichier assembleur généré par le compilateur. Cela diminuera le temps de compilation.

PRI or PRIORITY

Numeric: Permet de régler la priorité du compilateur. La valeur donnée doit être comprise entre -128 et +127. C'est une bonne idée de donner une valeur supérieure à 0 quand on utilise le compilateur sur des grands codes source.

CR or CREATERESIDENT

Switch: Utilisé pour générer un nouveau fichier résident à partir du fichier source (Toutes les constantes et les structures sont sauvegardées). Les fichiers résidents créé se nomment "Ram:ResidentFile" et "Ram:ResidentFile.struct".

STANDBY

Switch: Utilisé pour lancer le compilateur et le mettre en mode veille. Il attend des instructions extérieures pour continuer. Cette option sert uniquement à interfacer le compilateur avec un programme externe comme un éditeur. Ne l'utilisez pas pour le moment.

DB or DEBUGGER

Switch: Permet de lancer le debugger à l'exécution du programme et de surveiller ainsi son déroulement. Pour plus de détails sur son utilisation, consultez le chapitre debugger Debugger .

OPT or OPTIMIZATIONS

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

Exemples:

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

```
CLI:> PureBasic Sources:Exemple.pb TO Ram:Exemple.exe OPT PRI=10
```

1.4 Règles générales

Règles générales:

PureBasic a des règles globales bien établies qui ne varient pas durant son utilisation: ↵

- * Les commentaires sont signalés par un ; . Tout texte entré après ce point- ↵
virgule
est simplement ignoré par le compilateur.

Exemple:

```
If a = 10; Ceci est un commentaire..
```

- * Toutes les fonctions et commandes externes sont suivies de parenthèses sinon ↵
le mot
est considéré comme une variable (même pour les fonctions sans arguments).

Exemple: WindowID() est une fonction.
WindowID est une variable.

- * Toutes les constantes sont précédées par #

Exemple:

```
#Hello = 10 est une constante.  
Hello = 10 est une variable.
```

- * Tous les labels doivent être suivis par un :

Exemple:

```
Je_suis_un_label:
```

- * Une expression est quelque chose qui peut être évalué. Une expression peut ↵
utilisé n'importe
quel opérateur, et mixer des variables avec des constantes, des fonctions tant ↵
que tout
est du même type.

Exemples d'expressions valides:

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


- * Un nombre illimité de commandes peuvent être mis sur la même ligne en utilisant les : option.

Exemple:

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

- * Mots clefs utilisés dans ce manuel:

```
<variable> : une variable basique.  
<expression>: une expression, comme indiqué ci-dessus.  
<constant> : une constante numérique.  
<label> : un label.  
<type> : un type de données (basique ou structuré).
```

- * Dans ce manuel, toutes les rubriques sont listées dans l'ordre alphabétique, pour réduire le temps de recherche.

1.5 Information sur les variables

Déclaration des variables:

Pour déclarer une variable sous PureBasic, il suffit de taper son nom. La variable utilisera alors le type par défaut (normalement 'word', mais il peut être changé grâce à la commande DefType). Vous pouvez spécifier un type pour la variable au moment de la déclaration en lui ajoutant un '.' puis le nom du type. Les variables peuvent être déclarées n'importe quand, même au sein d'une expression complexe.

Exemple:

```
a.b ; Declaration d'une variable de type 'Byte'  
c.l ;  
  
c = a*d.w ; 'd' est déclarée dans l'expression
```

Pour déclarer une variable en tant que pointeur, il suffit de mettre une '*' devant le nom de la variable. Un pointeur est une variable de type 'Long' qui stocke une adresse mémoire. Il est généralement associé à une structure pour accéder à une zone de mémoire avec facilité.

Exemple:

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

Types de base:

PureBasic permet de déclarer les variables en utilisant différents types. Pour l'instant, seules les variables signées sont supportées à 100%. Les variables non-signées sont en cours de développement.

Types:

Byte: `.b`, consomme 1 octet en memoire. Valeurs possibles: de -128 à ↵
+127.

Word: `.w`, consomme 2 octets en memoire. Valeurs possibles: de -32768 à ↵
+32767

Long: `.l`, consomme 4 octets en mémoire. Valeurs possibles: de -2147483648 à ↵
+2147483647

Unsigned Byte: `.ub`, consomme 1 octets en mémoire. Valeurs possibles: de 0 à ↵
255

Unsigned Word: `.uw`, consomme 2 octets en mémoire. Valeurs possibles: de 0 à ↵
65535

Unsigned Long: `.ul`, consomme 4 octets en mémoire. Valeurs possibles: de 0 à ↵
4294967295

String: `.s`, chaîne de caractères.

Types structurés:

Ils sont contruits grâce aux structures. Pour plus d'informations, consultez le chapitre Structures

1.6 For : Next

Syntaxe:

```
For <variable> = <expression1> To <expression2> [Step <constant>]
```

```
... Contenu de la boucle
```

```
Next [<variable>]
```

Description:

Les commandes "For/Next" servent à créer une boucle au sein d'un programme, en tenant compte des paramètres donnés. A chaque boucle, la <variable> est incrémentée de 1 (ou de la valeur spécifiée après Step) et quand la valeur de la <variable> est égale ou supérieure à la valeur de l'<expression2> la boucle s'arrête et le programme continue après le Next.

Exemple 1:

```
For k=0 To 10
```

```
...
```

```
Next
```

Dans cet exemple, le programme va boucler 11 fois avant de continuer.

Exemple 2:

```
a = 2
b = 3

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

Ici, le programme va boucler 4 fois avant de continuer. Pourquoi ? Parce que 'k' est incrémenté de 2 (valeur du Step) donc la valeur de 'k' va prendre respectivement: 4, 6, 8 et 10. A la sortie de la boucle, 'k' sera égal à 10. Le 'k' après Next est destiné à confirmer que la boucle qu'on termine est bien la boucle 'For k'. Si ce n'était pas la bonne boucle, le ← compilateur aurait généré une erreur. C'est très utile quand on commence à imbriquer les boucles les unes dans les autres.

Exemple 3:

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

1.7 Gosub : Return

Syntaxe:

```
Gosub <label>

<label>:

    ... Code de la fonction

Return
```

Description:

Gosub veut dire en anglais 'Go to subroutine', c'est à dire: Aller à la sous routine. Une sous-routine est une partie de programme qui commence par un label, suivit par le code et qui se termine par un Return. Une fois que le Return est atteint, le programme revient à la position du Gosub et continue son execution. Les Gosub sont très utiles pour contruire une code structuré très rapide.

Exemple:

```
a = 1
b = 2

Gosub ComplexOperation

PrintNum(a)
End

ComplexOperation:

    a=b*2+a*3+(a+b)
    a=a+a*a

Return
```

Syntaxe:

```
FakeReturn
```

Description:

Si pour une raison ou pour une autre vous voulez sortir d'une sous routine sans passer par un 'Return' (en utilisant un Goto) vous devez au préalable informer le compilateur que le Return ne sera jamais exécuté. C'est pour cela que la fonction FakeReturn a été créée et vous devez impérativement l'utiliser dans ce cas là.

Exemple:

```
Main_Loop:
    ...

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

Return
```

1.8 If : EndIf

Syntaxe:

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

Description:

Ces fonction permettent de réaliser des branchements dans un programme, c'est à dire de prendre une direction ou une autre en fonction du resultat du test. If signifie 'Si' et Else signifie 'Sinon'. Donc littéralement on peut lire les tests comme ca: 'Si le résultat de l'expression est vrai alors on execute la partie de code correspondante, sinon on execute l'autre partie. Le mot clef Else est optionnel.

Les structures If-Else-Endif peuvent être imbriquée les unes dans les autres sans limite.

Exemple 1:

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

Exemple 2:

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

1.9 Repeat : Until

Syntaxe:

```
Repeat

... Programme ...

Until <expression>
or [Forever]
```

Description:

Le couple Repeat:Until permet de réaliser une boucle jusqu'à ce que l'< expression> devienne vraie. Vous pouvez utiliser le mot clef Forever à la place de Until pour faire une boucle qui ne se termine jamais.

Exemple:

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

Ce programme bouclera jusqu'à ce que 'a' soit supérieur à 100.

1.10 Select : EndSelect

Syntaxe:

```
Select <expression1>

Case <expression2>

  ...Code...

[Case <expression3>....]

  ...Code...

[Default]

  ...Code...

EndSelect
```

Description:

La combinaison de 'Select:Case:EndSelect' permet de faire rapidement plusieurs tests différents sur le resultat d'une expression donnée. Cette fonction est plus rapide et plus 'propre' qu'une succession de If:Else:Endif. Le mot clef Default est optionnel, il est utile lorsque l'on désire faire une opération par défaut lorsque que tous les autres tests ont échoués. Il peut y avoir un nombre illimité de Case.

Exemple:

```
a = 2

Select a

Case 1
  PrintN("a = 1")

Case 2
  PrintN("a = 2")

Case 20
```

```
PrintN("a = 20")

Default
PrintN("Je ne sais pas la valeur de 'a'")

End Select
```

Syntaxe:

```
FakeEndSelect
```

Description:

Si pour une raison ou pour une autre vous voulez sortir d'un Select sans passer par un 'EndSelect' (en utilisant un Goto) vous devez au préalable informer le compilateur que le EndSelect ne sera jamais exécuté. C'est pour cela que la fonction FakeEndSelect a été créée et vous devez impérativement l'utiliser dans ce cas là.

Exemple:

```
Main_Loop:
...
Select a

    Case 10
        ...

    Case 20
        FakeEndSelect
        Goto Main_Loop

EndSelect
```

1.11 While : Wend

Syntaxe:

```
While <expression>

... Programme ..

Wend
```

Description:

Le couple While:Wend permet de réaliser une boucle jusqu'à ce que l'expression devienne fausse. Une particularité de While:Wend est que si le résultat du test est faux dès le départ, alors le code contenu dans la boucle ne sera jamais exécuté (à l'inverse d'une boucle Repeat:Until où le code est

de toute façon exécuté au moins une fois).

Exemple:

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

Ce programme bouclera jusqu'à ce que la valeur de 'a' ne soit plus 10. Comme on change la valeur de 'a' quand 'b=10', alors ce programme bouclera 10 fois.

1.12 Divers

Liste des commandes diverses:

Goto

```
Goto <label>
```

Permet de faire un saut vers un label précis. Le programme se continuera après le label. Faites attention lors de l'utilisation de cette commande, car si vous l'utilisez mal, vous pouvez faire planter le système (problèmes de pile, ...). Néanmoins vous pouvez l'utiliser dans ces cas grâce aux commandes suivantes:

```
FakeReturn    : simule un 'Return' sans remonter au 'Gosub'
FakeEndSelect: simule un 'EndSelect'
```

1.13 deftype

Syntaxe:

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

Description:

DefType est utilisé pour définir le type de plusieurs variables en une seule ligne. Si aucune variable est spécifié après DefType, alors le type par défaut est modifié. Toutes les prochaines variables déclarées sans type explicite utiliseront ce nouveau type.

Exemple:

```
DefType.1
```

```
a = b+c
```

'a', 'b' et 'c' seront des variables de type 'Long' (.l), parce qu'on a changé le type par défaut par un 'Long'

Par contre si des variables sont déclarées, alors le type par défaut reste ↔
inchangé.
Seules les variables spécifiées utiliseront ce type.

Exemple:

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

'a', 'b', 'c' et 'd' seront du type 'Byte' (.b)

1.14 dim

Syntaxe:

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

Description:

Dim est utilisé pour déclarer un nouveau tableau vide. Un tableau sous le ↔
PureBasic
peut avoir n'importe quel type, qu'il soit structuré ou non. Un fois qu'un ↔
tableau
est déclaré, vous ne pouvez plus modifier sa taille, ni le redéclarer.

Exemple:

```
Dim MonTableau.l(41) ; Déclaration d'un tableau de 42 éléments (le 0 compte ↔  
aussi !)
```

```
MonTableau(0) = 1 ; On met la valeur 1 dans l'emplacement 1 du tableau...  
MonTableau(1) = 2 ; ... et la valeur 2 dans le n\textdegree{}2.
```

1.15 newlist

Syntaxe:

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

Description:

NewList permet de déclarer une nouvelle liste d'éléments dynamique, ↔
entièrement vide. Il n'y

a pas de limite pour le nombre d'éléments. L'avantage d'une liste par rapport à un tableau est multiple: possibilité d'insérer, supprimer, ajouter un élément, la consommation mémoire de la liste au début est quasi-nulle (alors qu'un tableau réserve tout l'espace nécessaire dès le départ). N'importe quel type peut être utilisé pour la liste.

Pour voir toute les commandes permettant la gestion des listes, cliquez [ici](#)

Exemple:

```
NewList mylist.l()
```

```
AddElement(mylist()) ; Ajoute un élément a notre liste
```

```
mylist() = 10 ; Donnons la valeur 10 à cet élément.
```

1.16 structures

Syntaxe:

```
Structure <name of structure>
```

```
... Structure content
```

```
EndStructure
```

Description:

Les structure sont très utile pour manipuler des objets précis, et accéder à des zones de mémoires partagée. On peut accéder à un champs d'une structure grâce au caractère '//'. On peut utiliser un type structuré pour définir un champ dans la nouvelle structure.

Exemple:

```
Structure Info
```

```
  Nom.s
```

```
  Prenom.s
```

```
  Age.l
```

```
  Anniversaire.l
```

```
EndStructure
```

```
Dim MesAmis.Info(100) ; Déclaration d'un tableau de 101 amis
```

```
MesAmis(0)\Nom      = "Andersson"    ; Remplissage de l'ami '0'...
```

```
MesAmis(0)\Prenom = "Richard"      ;
```

```
...
```

Il y a un moyen pour partager une zone de mémoire au sein d'une structure. ←
Cette
fonctionnalité est connue sous le nom d'<Union>' en C/C++. Le PureBasic ←
supporte
sans problème les unions dans les structures principalement pour être ←
compatible
à 100% avec l'AmigaOS. Utiliser les unions dans les structures n'est pas ←
vraiment
une bonne idée. C'est complexe et ça peut engendrer des erreurs inattendues. ←
Les
commandes servant à gérer les unions en PureBasic sont: StructureUnion et
EndStructureUnion.

Exemple:

```
Structure.person

    StructureUnion
        *FirstName.l      ; Ces 2 pointeurs ont exactement la même adresse
        *AlternateName.l  ; physique mais le programmeur pourra utiliser
    EndStructureUnion    ; 2 noms différents pour y accéder.

EndStructure
```

1.17 global

Syntaxe:

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

Description:

Le mot clef Global permet de rendre une ou plusieurs variables accessible dans n'importe quelle partie du programme (procédure ou non).

Exemple:

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

1.18 shared

Syntaxe:

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

Description:

Shared signifie 'Partagé' en français et ne s'utilise uniquement au sein d' une procédure pour permettre à une variable externe de pouvoir être utilisée dans cette procédure.

Exemple:

```
a.1 = 10
```

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

```
  a = 20
```

```
EndProcedure
```

```
myproc()
```

```
PrintN(Str(a)) ; Ecrira '20', parce que notre variable 'a' a été partagée.
```

1.19 procedures

Syntaxe:

```
Procedure[.<type>] nom(<variable1>[,<variable2>,...])  
  
... Code  
  
[ProcedureReturn value]  
EndProcedure
```

Description:

Une procédure est une partie de programme totalement indépendante du reste de l'application (avec ses propres variables, tableaux...), qui peut avoir des paramètres en entrée. Avec le PureBasic, vous pouvez utiliser la récurrence, c'est à dire une fonction qui s'appelle elle-même. Pour accéder aux variables ou tableaux du reste de l'application, vous pouvez utiliser les mot clés `global` ou `shared`.

De plus une procédure peut renvoyer un résultat si nécessaire. Vous devez alors spécifier le type du résultat (après le mot clef 'Procedure') et utiliser le mot clef 'ProcedureReturn' où bon vous semble dans la procédure.

Exemple:

```
Global Résultat.l
```

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

```
If nb1>nb2
    Résultat = nb1
Else
    Résultat = nb2
Endif

ProcedureReturn Résultat

EndProcedure

Résultat.l = Maximum(15,30)

PrintNumberN(Résultat)

End
```

1.20 includes

Syntaxe:

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

Description:

IncludeFile permet d'inclure un fichier source externe à l'endroit voulu.
XIncludeFile sert à la même chose mais le fichier inséré ne pourra plus être inséré une seconde fois.

Exemple:

```
code..
XInclude "Sources:myfile.pb" ; Ce fichier sera inclu
code..
XInclude "Sources:myfile.pb" ; Celui-ci sera ignoré, parce qu'il a déjà été ←
    inclu.
code..
```

Syntaxe:

```
IncludeBinary "filename"
```

Description:

IncludeBinary permet d'insérer un fichier de n'importe quel type (image, son, fichier binaire) tel quel dans le code.

Exemple:

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

Syntaxe:

```
IncludePath "chemin"
```

Description:

"IncludePath" permet de spécifier un chemin par défaut à utiliser pour toutes les prochaines fonctions 'Include'. C'est très pratique lorsque plusieurs fichiers à inclure se trouvent dans le même répertoire.
same directory:

Exemple:

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

1.21 debugger

Le debugger du PureBasic

Le debugger est un programme externe qui permet de contrôler l'exécution du programme en cours de développement et de prévenir les bugs éventuels. Il est indispensable de l'utiliser lors du développement ! Vous pouvez par exemple arrêter une boucle sans fin par simple click. Tout est visuel et très facile d'accès. De plus il est 100% multitâche et respecte le systèmes (pas de TrapVector ou d'interruptions..)

Fonctionnalités des boutons:

Stop

Arrête l'exécution du programme en cours et affiche la ligne qui devait être exécutée.

Cont

Continue l'exécution d'un programme préalablement arrêté.

Step

Arrête l'exécution d'un programme et avance pas par pas (instructions par instructions et non lignes par lignes) pour localiser avec exactitude l'origine d'un bug.

Trace

Permet d'exécuter un programme tout en affichant les lignes exécutées.

Exit

Arrête complètement le programme et retourne sous le compilateur, comme si

le programme s'était terminé correctement.

Les mot clés permettant de contrôler le debugger dans un programme:

CallDebugger:

Arrête l'exécution du programme et affiche la position courante du code.

Exemple:

```
If a=10
    CallDebugger    ; Le debugger apparaîtra
Else
    Ok=1
Endif
```

1.22 inlinedasm

Assembleur 680x0 en ligne:

----- Note Importante -----

Ce chapitre est uniquement consacré aux programmeurs maîtrisant le langage assembleur 680x0. Si vous ne le connaissez pas, vous pouvez passer votre chemin...

PureBasic vous permet d'inclure n'importe quelle commande assembleur 680x0 directement dans votre code source. Les commandes peuvent se situer à n'importe quel endroit dans le code. De plus vous pouvez utiliser les variables BASIC et les pointeurs directement dans les instructions 680x0 ! Vous pouvez aussi mettre autant d'instructions 680x0 que vous voulez sur la même ligne, à la manière des commandes standards du PureBasic. Tous les mots clés des processeurs 680x0s sont supportés, du 68000 au 68060. Cliquez [ici](#) pour avoir la liste complète des mots clés 680x0 disponibles. Pour avoir plus d'informations, veuillez vous reporter au guide de PhxAss (disponible sur Aminet en Freeware).

Si vous voulez inclure du code assembleur dans votre programme BASIC, vous devez respecter les règles suivantes:

- + Désactivez toujours le debugger du PureBasic.
 - + Les variables ou pointeurs utilisés dans les instructions 680x0 doivent être déclarés avant.
 - + Quand vous référencez un label, vous devez mettre la lettre 'p' devant le nom du label. C'est dû au fait que le compilateur
-

PureBasic met un 'p' devant chaque label, pour éviter tout conflit avec ses labels internes.

Exemple:

```
LEA.l pMonLabel(pc),a0
...
MonLabel:
```

- + Les erreurs qui proviennent d'un code asm mal écrit ne sont pas signalées par PureBasic mais par PhxAss. Si une telle erreur survient, veuillez vérifier votre code assembleur.
- + Les registres a2,a3,a4 doivent toujours être préservés entre 2 lignes en Basic. Tous les autres registres peuvent être utilisé sans contraintes. Le registre 'a7' est le registre de pile.

Exemple:

```
Inlined ASM example
```

1.23 Commandes Internes

```
*****
*
*      Liste des commandes internes du PureBasic      *
*
*      © 2000 - Fantaisie Software -                *
*
*
*****
```

Commandes disponibles:	Chapitre:
.b	Variables
.l	Variables
.s	Variables
.ub	Variables
.ul	Variables
.uw	Variables
.w	Variables
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