Inside wxBasic

David Cuny

Draft November 30, 2001

Table of Contents

A Word Of Thanks....9
wxBasic At A Glance....10
What is wxBasic?....10
Is wxBasic Free?....10
General Overview....10
Variables....10
Scope Of Variables....11
Line Breaks....12
Arrays....12
Dynamic Arrays....12
Block Structures....13
Functions....13
Pass By Reference....14
Shrouding – Making Source Code Less Readable....14
Binding – Creating a Stand-Alone Executable....14
Writing a GUI Application....15
A Simple Application....15
An Example With Callbacks....15
The wxWindows Interface....16
wxBasic Tutorial....18
Running wxBasic Programs....18
Running wxBasic from the Command Line....18
Running wxBasic from Windows....18
Installation....19
Your First Program....19
Your First GUI Program....20
What Next?....21
The wxBasic Language Reference....22
Keywords....22
Operators....22
Assignment Operators....23
wxBasic Syntax....24
= (Assignment)....24
class....24
functionName....25
Abs....25
ACos....25
Argc....25
Argv....26
Asc....26
ASin....26
ATan....26
Chr$....27

Close....27
Common....27
Connect....27
Const....28
Continue....28
Cos....28
Date$....28
Declare....29
Delete....29
Dim....29
End....30
Erase....30
Eof....31
Fix....32
Floor....32
For ... Next 32
For ... End For....32
For Each ... Next....33
For Each ... End For....33
Frac....33
FreeFile....33
Function ... End Function 33
Hex$....34
If...ElseIf...Else...End If 34
In....35
Include....35
Indexes....35
Instr....35
Int....36
LBound....36
LCase$....36
Left$....36
Len....36
Length....36
Line Input #....37
Loc....37
Lof....37
Log....37
LTrim$....38
Mid$....38
New....38
Open....38
Option Explicit....39
Option NoConsole....39
Option QBasic....39

Print....40
Print #....40
Randomize....40
ReadByte....41
Reverse$....41
Right$....41
RInstr....41
Rnd....42
Round....42
RTrim$....42
Seek....42
Select ... End Select....42
Sgn....43
Shared....44
Shell....44
Sin(n)....44
Space$....44
Sqr....45
Str$....45
String$....45
Tan....45
Ticks....45
Time$....46
TypeOf$....46
UBound....46
UCase$....46
Val....47
While ... Wend....47
While ... End While....47
WriteByte....47
Overview of the Interpreter....48
The Lexer....48
The Parser....49
Parse Tree Nodes....50
Parse Trees....51
The Evaluator....52
The Inner Workings....53
Internal Values....54
The Data Stack....54
References on the Stack....55
Code Blocks....55
Complex Operations....56
Overview of the Modules....58
CORE.C....58
SHARED.H....58

ERROR.C....58
STACK.C....58
DATA.C....59
SYMBOL.C....60
VAR.C....60
ARRAY.C....61
NODE.C....61
BUILTIN.C....61
EVAL.C....62
LEXER.C....62
CLASS.C....62
CONNECT.CPP....64
Y_TAB.C....64
CONSOLE.CPP....64
The wxWindows Wrappers....66
Design Decisions....66
The Wrapper File Format....66
Comments: //....66
%{....67
%include....67
%enum....67
%typedef....67
%struct....68
%class....68
%ctor....69
%builtin....69
Simple Classes....70
Class Constructors....70
Class Destructors....71
Class Methods....72
Decoding The Method Type....73
virtual....73
static....73
const....73
* (pointer)....74
& (deref)....74
Decoding The Method Args....74
Optional Args....75
const....75
wxWindows DataTypes....76
Dereferencing....76
Pointers....76
Arrays....77
Calling The Method....77
wxWindows Methods....79

builtin (no class)....79
wxActivateEvent....80
wxApp....80
wxBitmap....80
wxBitmapButton....81
wxBoxSizer....81
wxBrush....81
wxBrushList....82
wxButton....82
wxCalculateLayoutEvent....82
wxCalendarCtrl....82
wxCalendarEvent....83
wxCheckBox....83
wxCheckListBox....83
wxChoice....83
wxClientDC....84
wxCloseEvent....84
wxColourDialog....84
wxComboBox....84
wxCommandEvent....85
wxControl....85
wxDC....85
wxDialog....87
wxDialUpEvent....87
wxDirDialog....87
wxDropFilesEvent....88
wxEraseEvent....88
wxEvent....88
wxEvtHandler....88
wxFileDialog....88
wxFlexGridSizer....89
wxFocusEvent....89
wxFont....89
wxFontDialog....89
wxFrame....89
wxGauge....90
wxGDIObject....90
wxGrid....90
wxGridSizer....94
wxIdleEvent....95
wxImage....95
wxIndividualLayoutConstraint....96
wxInitDialogEvent....96
wxJoystickEvent....96
wxKeyEvent....96

wxLayoutConstraints....97
wxListBox....97
wxListCtrl....97
wxListEvent....98
wxMDIChildFrame....99
wxMDIParentFrame....99
wxMemoryDC....99
wxMenu....99
wxMenuBar....100
wxMenuEvent....101
wxMenuItem....101
wxMessageDialog....101
wxMetafileDC....101
wxMiniFrame....101
wxMouseEvent....101
wxMoveEvent....102
wxNotebook....102
wxNotebookEvent....103
wxNotebookSizer....103
wxNotifyEvent....103
wxObject....103
wxPageSetupDialog....103
wxPaintDC....104
wxPaintEvent....104
wxPalette....104
wxPanel....104
wxPen....104
wxPlotWindow....105
wxPoint....105
wxPostScriptDC....105
wxPrintDialog....105
wxPrinterDC....105
wxProcessEvent....106
wxQueryLayoutInfoEvent....106
wxRadioBox....106
wxRadioButton....106
wxSashLayoutWindow....107
wxSashWindow....107
wxScreenDC....107
wxScrollBar....107
wxScrolledWindow....108
wxScrollEvent....108
wxScrollWinEvent....108
wxSingleChoiceDialog....108
wxSize....108

wxSizeEvent....109
wxSizer....109
wxSlider....109
wxSocketEvent....110
wxSpinButton....110
wxSpinCtrl....110
wxSpinEvent....110
wxSplitterWindow....111
wxStaticBitmap....111
wxStaticBox....111
wxStaticBoxSizer....111
wxStaticText....111
wxStatusBar....111
wxSysColourChangedEvent....112
wxTabCtrl....112
wxTabEvent....112
wxTextCtrl....112
wxTextEntryDialog....113
wxTimer....113
wxTimerEvent....114
wxToolBar....114
wxTreeCtrl....114
wxTreeEvent....116
wxUpdateUIEvent....116
wxWindow....116
wxWindowDC....119
wxWizardEvent....119
To Do and Wish Lists....120
Conversion Routines are in QBasic....120
Namespace....120
Bytecodes....120
User-Defined Types and Classes....121
Interactive Debugging....121
An IDE....121

A Word Of Thanks..

I'm not sure if there's a lot of demand for Yet Another Basic Interpreter, but here it is anyway. A number of people are to thank (or reprimand) for this project getting this far.

First, thanks for Everett (Rett) Williams for encouraging me to port my *Py* programming language to C. Somewhere along the line (much to his consternation) *Py* became *wxBasic*. So go the best laid plans of mice and men.

This project would not have been possible without a number of fine groups and their free software tools, including:

wxWindows for the great cross-platform wxWindows library. Hopefully wxBasic will make the wxWindows library more accessible to people.

wxBasic owes much of it's small size to the **UPX** file compression program.

Thanks to **Borland** for their free C++ compiler. It works great with the wxWindows library.

Thanks to Colin Laplace of Bloodshed Software for the **Dev C++ IDE**, and Mumit Kahn, Jan Jaap van der Heidjen, Colin Hendrix and the other Gnu coders for the **Mingw C++ compiler**.

Another thanks to Gnu for **Bison**, a free version of **YACC**.

Thanks are also due to Jeffrey Kingston for the **Basser Lout 2** typesetting program, which this document was initially set in. Thanks to the Gnu and AFPL folk for the **GhostScript** interpreter, and Russell Lang for **GSView,** the graphical GhostScript viewer which I used to convert to document to PDF format.

I'm especially indebted to Brian Kernighan and Rob Pike for the books **The Unix Programming Environment** and **The Practice of Programming**. The core of wxBasic is built around the ideas they presented in these texts.

wxWindows:	http://www.wxwindows.org
Borland Compiler:	http://www.borland.com/bcppbuilder/freecompiler/
GNU Bison:	http://www.gnu.org/software/bison/bison.html
UPX:	http://wildsau.idv.uni-linz.ac.at/mfx/upx.html

wxBasic At A Glance

What is wxBasic?

wxBasic makes it easy for to write cross-platform applications. It combines the simplicity of Basic with the rich toolkit of wxWindows.

- Free (Lesser Gnu Public License)
- Cross-platform GUI support via wxWindows library. Runs in Windows and Linux
- Easy to learn, based on Basic
- Interpreted
- Small footprint (fits on a floppy)

Is wxBasic Free?

Yes. wxBasic is released under the LGPL (*Lesser Gnu Public Licence*).

General Overview

wxBasic borrows heavily from BASIC. It is case-insensitive.

Variables

By default, wxBasic will create a variable for you when it sees an assignment:

```
' This will create a variable called 'newVar'
newVar = "create me"
```

There are no types; all variables and arrays are variants. In general, wxBasic will attempt to automatically cast a variable to the correct type. For example:

```
' create variables
this = "123"
that = 456

' this is automatically converted to a number
Print that + this
```

Scope Of Variables

When a variable is created, it is *scoped* to the current context. That is, it is visible to anything in the current scope or greater. The widest scope is the module:

```
newVar = 12
```

The next scope is the routine:

```
Function myRoutine()
    newVar = 12
End Function
```

When wxBasic encounters a variable in a routine, it checks to see if there is a variable scoped to the routine of the same name and use that.

```
' This is declared at the module level
Dim myVar = "module variable"

Function myRoutine()
    ' This is declared at the routine level
    Dim myVar = "routine variable"

    ' use the local routine version of myVar
    myVar = "new value"
End Function
```

If not, it checks to see if there is a module variable and uses that.

```
' This is declared at the module level
Dim myVar = "module variable"

Function myRoutine()
    ' use the module version of myVar
    myVar = "new value"
End Function
```

If there is no routine or module version of the variable, one will be created in the current scope.

```
Function myRoutine()
    ' create a variable scoped to the routine
    myVar = "new value"
End Function
```

Variables scoped to routines are only visible within the routines that they are scoped to:

```
Function myRoutine()
    myVar = 12
    ' myVar is visible here
End Function
```

```
' myVar is invisible here
```

You can prevent wxBasic from creating variables with the Option Explicit statement. With Option Explicit, you will need to declare your variables before use:

```
Option Explicit
Dim newVar = "create me"
```

If you use Option Explicit, your module level variables will be hidden from your routines unless you specifically declare them visible with the Shared keyword:

```
Option Explicit
Dim myVar = "module variable"

Function myFunction()
    Shared myVar
    myVar = "new value"
End Function
```

Line Breaks

Each line can contain one or more statements. Statements can be split into multiple lines following an operator. For example:

```
' several statements on a line
this = 12 : that = 34 : theOther = 56

' a line split into several lines
result = this + that /
    theOther + 12
```

Arrays

Arrays in wxBasic use [] instead of (). For example:

```
Dim list[10,32]
```

Dynamic Arrays

In addition to the standard declared arrays, wxBasic supports *dynamic* arrays. If you declare an array without listing the indexes:

```
Dim myArray[]
```

wxBasic will treat it as a *dynamic* array. Dynamic arrays use strings as their indexes:

```
myArray["cat"] = "Chester"
```

You can use numeric values for indexes, but they will be stored internally as strings. So the following declarations are equivalent:

```
myArray[1,2,3] = 23
myArray["1,2,3"] = 23
```

You can use the `For Each` construct to interate through dynamic arrays:

```
For Each key In myArray
    Print key, myArray[key]
Next
```

Block Structures

In addition to the standard BASIC structures:

```
For ... Next
While ... Wend
```

you can use the Algol inspired forms:

```
For ... End For
While ... End While
```

Functions

wxBasic borrows the return statement from C. In addition to the standard method of returning values from functions:

```
Function add( a, b )
    add = a + b
End Function
```

you can also write:

```
Function add( a, b )
    Return a + b
End Function
```

Like C, wxBasic exits the function at the point the return statement is executed. Also like C, you can choose to simply ignore the result of a function, and treat it like a `Sub`:

```
' Ignore the return value of Len()
```

```
Len("123")
```

Pass By Reference

All types are passed by reference, which means changing a parameter passed to a routine effects the value in the caller:

```
Function swap( a, b )
    tmp = a
    a = b
    b = a
End Function
```

Shrouding – Making Source Code Less Readable

You can use the program `shroud.wx` to make to *shroud* your code. It will go through and replace user-defined routines and variables with non-descriptive names. The syntax is:

wxbasic shroud.wx *sourcefile destfile*

A couple points to keep in mind:

- This is still in alpha development
- It doesn't handle `included` files yet
- Tokens starting with "wx" will *not* be shrouded.
- Strings matching routine names *will* be shrouded.

Any token starting with "wx" will be left alone, and any string that matches the name of a

Binding – Creating a Stand-Alone Executable

You can use the program `bind.wx` to make to a stand-alone executable program. Binding a program will attach your source code to the end of the wxbasic executable. When wxbasic launches, it will check for a tag at the end of the file, to see if an source file has been attached. If it has, it will load and execute that file.

The syntax is:

```
wxbasic bind.wx wxbinary sourcefile destfile
```

For example:

```
wxbasic bind wxbasic.exe freecell.wx freecell.exe
```

would bind the file `freecell.wx` to the wxBasic binary file `wxbasic.exe`, creating the

stand-alone binary file `freecell.exe`. You might want to bind your source file before shrouding it.

Keep in mind:

- This is still in alpha development
- It doesn't handle `included` files yet

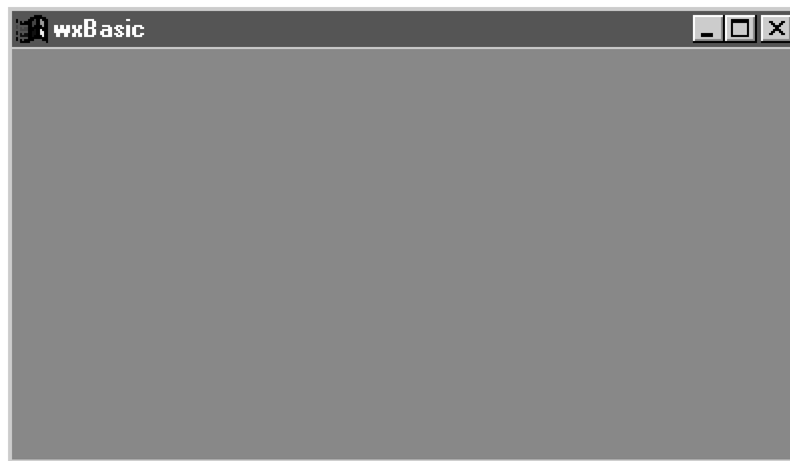
Writing a GUI Application

A Simple Application

Here's perhaps the simplest wxBasic GUI application:

```
' create a window
frame = New wxFrame( 0, -1, "wxBasic" )

' show the frame
frame.Show( True )
```



This program creates a window of the class `wxFrame`, and then processes events until the window is closed.

An Example With Callbacks

Here's a slightly more complex example, which attaches an action to a button:

```
' create a window
frame = New wxFrame( 0, -1, "wxBasic" )

' create a button in the window
```

```
button = New wxButton( frame, -1, "Press Me" )

' create a callback function
Function ShutDown( event )
    End
End Function

' attach the callback
Connect( button, wxEVT_COMMAND_BUTTON_CLICKED, "ShutDown" )

' show the frame
frame.Show( True )
```

In addition to creating a `wxFrame`, this example also creates a `wxButton` called `button`.

The `Connect` routine associates a *callback* called `ShutDown` to the button.

> A *callback* is an action associated with an event. Typically, the windows and controls in the windows (often called widgets, for "window gadgets"_ are handled by the operating system (in Windows) or a library (in Linux). A *callback* is the glue between the widgets and the wxBasic application
>
> When an *event* occurs in the system (such as a mouse click, window resize, and so on), wxBasic looks to see if the application has registered a *callback routine* to be triggered. If it has, wxBasic wraps that event into a `wxEvent` object, and then activates that callback.
>
> Callback are registered using the `Connect`() function.

When `button` is pushed, a `wx_COMMAND_BUTTON_SELECTED` event is triggered by wxWindows, and the callback `ShutDown` is executed.

The wxWindows Interface

wxBasic attempts to mimic wxWindow's C++ interface. This means that, allowing for the fact that wxBasic isn't C++, most of the documentation on wxWindows can be applied t to wxBasic in a rather trivial manner.

wxBasic attempts to follow imitate the C++ interface with respect to wxWindows object. wxWindows objects are created with the `New` keyword, which returns a handle to the new

object:

```
frame = New wxFrame( 0, -1, "wxBasic", wxPoint( 10, 10 ),
    wxSize( 300, 200 ) )
```

and are destroyed with the `Delete` keyword:

```
Delete Frame
```

If an object is created without the `New` keyword, that object acts as if it were created "on the stack", and will be destroyed when the routine that created it exits. wxBasic will issue an error if you do this to an object that inherit from `wxWindow`.

In general, you can access an object's wxWindows method by writing:

```
object.method()
```

For example:

```
frame.Show(True)
frame.SetSize( 10, 20 )
```

You are limited to a single level of nesting with this form. For example, you can't write something along the lines of:

```
// probably not a real method anyway...
x = frame.getParent().getPositionX()
```

Instead, you would have to write:

```
// only one method per call
p = frame.getParent()
x = getPositionX()
```

wxBasic Tutorial

Pardon our dust... This is very much a work in progress.

Running wxBasic Programs

wxBasic programs are written as plain text files. You can use any sort of editor, as long as you save in plain text (`.TXT`*) format. To run a program written in wxBasic, you only have to have the wxBasic executable, wxbasic.exe (in Linux, wxbasic) in the current directory, or somewhere in your path. If there are any includefiles, they must be in the same directory as the source file.*

Running wxBasic from the Command Line

If you use the Windows command line, you can simply type:

```
WXBASIC filename
```

or in Linux,

```
./wxbasic filename
```

where *filename* is the name of the file you want to run. For example, to run a program called myprog.wx, you would write:

```
wxbasic myprog.wx
```

in Windows, or:

```
./wxbasic myprog.wx
```

in Linux.

There is no requirement that wxBasic programs end with .wx; any sort of extention like.`BAS`, or even none at all, is fine.

Running wxBasic from Windows

You can also drag and drop wxBasic programs onto the `wxbasic.exe` interpreter.

The simplest method is just to associate an extention, such as .wx or .bas with the `wxbasic.exe` executable. Then you can just double-click files with that extention, and Windows will automatically launch the wxBasic interpreter.

Installation

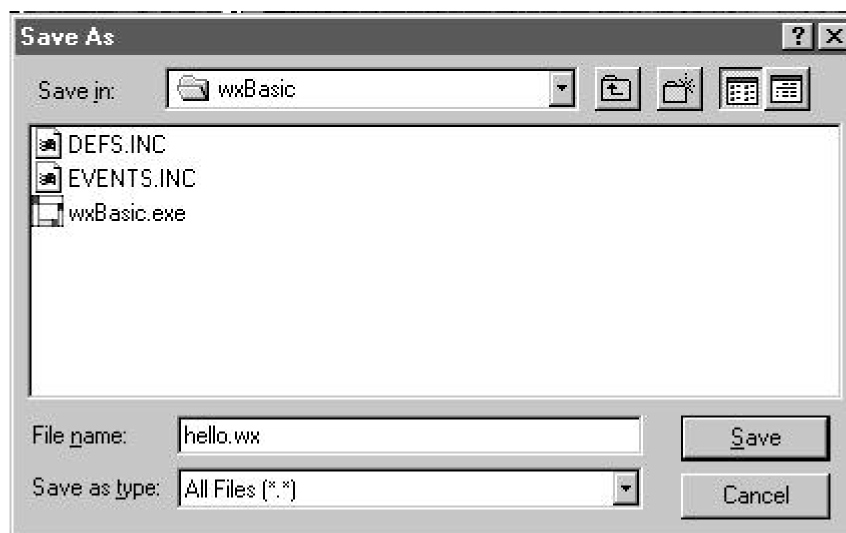
The only file that needs to be installed is the wxBasic executable.

Your First Program

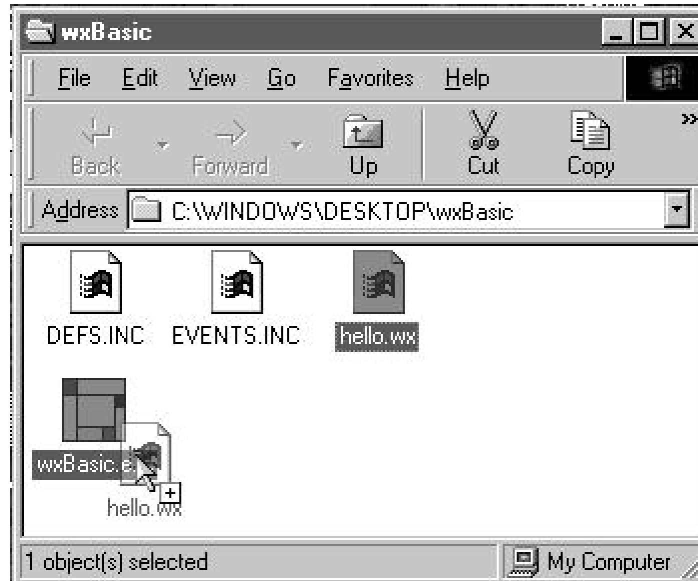
Bring up the text editor of your choice (for example, Notepad), and enter in the following:

```
print "Hello, World!"
```

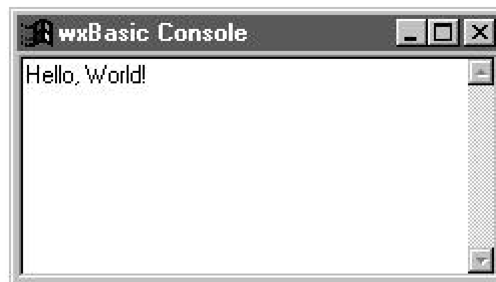
Save the file as hello.wx into the wxBasic directory. If you are using Windows, be sure to set the file type to `All Files (*.*)`, or the editor will automatically give the file a `.TXT` extention.



To run the program, open the wxBasic directory and drag the file `hello.wx` onto the wxBasic executable:



A console window will appear with the message in it:



Your First GUI Program

Bring up the text editor of your choice (for example, `Notepad`), and enter in the following:

```
frame = New wxFrame( 0, -1, "wxBasic" )
frame.Show( True )
```

Save the file as `helloGui.wx` into the wxBasic directory. If you are in Windows, be sure to set the file type to `All Files (*.*)`, or the editor will automatically give the file a .txt extention.

Drag and drop the `helloGui.wx` program onto the wxbasic.exe file, and the window will appear:

What Next?

From this point, you should explore the demos that come with wxBasic. For full documentation of the wxWindows library, download the help file at:

http://www.wxwindows.org

The wxBasic Language Reference

Keywords

The following keywords are reserved:

And	Append	As	Case	Close	Const
Continue	Declare	Delete	Dim	Else	ElseIf
End	Erase	Exit	Explicit	For	Function
If	Include	Input	Inv	Line	Mod
New	Next	Not	Open	Option	Or
Output	Print	QBasic	Return	Select	Shl
Shr	Step	Sub	Then	To	Until
Wend	While	Xor			

Operators

Here are the operators that can be used to create expressions in wxBasic. They are listed in order of operator precedence:

expression ^ *expression*	power
- *expression*	unary minus
expression * *expression*	multiplication
expression \ *expression*	integer division
expression / *expression*	division
expression Shl *expression*	shift left
expression Shr *expression*	shift right
expression Mod *expression*	modulus
expression Inv *expression*	inverse
expression + *expression*	addition
expression & *expression*	string concatenation
expression - *expression*	subtraction
expression = *expression*	equality
expression <> *expression*	inequality
expression != *expression*	inequality (alternate form)
expression < *expression*	less than
expression > *expression*	greater than
expression <= *expression*	less or equal to
expression >= *expression*	greater or equal to
Not *expression*	logical not
expression And *expression*	logical And
expression Or *expression*	logical Or
expression Xor *expression*	logical Xor
expression | *expression*	bitwise Or

Assignment Operators

In addition to the standard assignment form:

```
variable = expression
```

wxBasic also supports the assignments:

```
variable += expression   ' same as variable = variable + expression
variable -= expression   ' same as variable = variable - expression
variable *= expression   ' same as variable = variable * expression
variable /= expression   ' same as variable = variable / expression
variable &= expression   ' same as variable = variable & expression
```

wxBasic Syntax

= (Assignment)

lval = expression

Assigns value of expression to *lval*, which can be a variable or array item.

```
' Set the message text
message = "wxWindows is cool!"

'Numbers from 1 to 3
Dim a[3]
a[1] = "One"
a[2] = "Two"
a[3] = "Three"
```

class

object = class(arglist)

Create a *wxWindows* object on the stack. This is the same as creating it with `New`, but the object will be destroyed when the routine it was created in is destroyed.

This form is typically used in a function call, where it is more convenient than creating a variable to hold the result.

Always use `New` when creating controls and windows - never create them on the stack, or your program will probably crash when the object is automatically destroyed.

The stack can only contain a limited number of objects. The current maximum is 1024. If you are in a loop and might possibly create a large number of objects, you should probably use `New` and explicitly `Delete` the objects after use.

In the following example, `wxFrame` and `wxPoint` are created "on the stack".

```
' Create a new window
frame = new wxFrame( 0, -1, "My Window",
```

```
        wxPoint(50, 50), wxSize(250, 140))
```

functionName

```
lval = functionName( arglist )
functionName( arglist )
functionName
```

Calls *functionName*. If the result is not assigned, it is discarded.

```
    'Calculate sin of .5
    n = Sin(.5)

    'Display a message, discard result
    messageBox( "Hello, World!", "Hello", 0 )
```

Abs

```
n2 = Abs(n1)
```

Returns the absolute value (positive value) of *n1*.

```
' n is 1
n = Abs( 1 )
```

ACos

```
n2 = ACos(n1)
```

Return an angle with cosine equal to *n1*. The argument must be in the range -1 to +1 inclusive.

```
    ' n is 3.141592654
    n = ACos( -1 )
```

Argc

```
n2 = Argc()
```

Return the number of arguments passed to a routine. Used with routines that take variable numbers of parameters. This can only be called within a `Function` or a `Sub`.

```
    Function myFunction( ... )
        Print "myFunction was passed "; Argc(); " parameters"
        For i = 1 To Argc()
            Print "Parameter "; i; " is "; Argv(i)
        Next
```

```
End Function
```

Argv

n2 = Argc(*n1*)

Return the value of parameter *n1* passed to a routine. Used with routines that take variable numbers of parameters.

```
Function myFunction( ... )
    Print "myFunction was passed "; Argc(); " parameters"
    For i = 1 To Argc()
        Print "Parameter "; i; " is "; Argv(i)
    Next
End Function
```

Asc

n = Asc(*string*)

Return the ASCII code of the first character in *string*. Empty strings will return a value of zero.

```
' set n to 65
n = Asc("A")
```

ASin

n2 = ACos(*n1*)

Return an angle with sine equal to *n1*. The argument must be in the range -1 to +1 inclusive. A value between -PI/2 and +PI/2 (radians) will be returned.

```
' n is -1.570796327
n = ASin( -1 )
```

ATan

n2 = ATan(*n1*)

Return an angle with sine equal to *n1*. A value between -PI/2 and PI/2 (radians) will be returned.

```
' n is 0.785398
n = ATan( 1 )
```

Chr$

```
string = Chr$(n)
```

Returns a 1 character string with `ASCII` code character \n.

```
' s = "A"
s = Chr$(65)
```

Close

```
Close( file number )
Close
```

Close *file number*. If no file is specified, all files are closed.

```
' Print "Hello, wxBasic" to the file "temp"
Open "temp" For Output as #1
Print #1, "Hello, wxBasic"
Close #1
```

Common

```
Common name[]
Common name[ maxValue ] [[...]] [, ... ]
Common name[ minValue To maxValue ][[...]] [, ...]
Common name [ = expression ] [ , ... ]
```

Create a global array or variable named *name*. These are globally accessible, and do not need to be declared as `Shared` when used in a `Sub` or `Function`.

Common can only be used outside of `Sub` and `Function`. Inside of `Sub` or `Function`, use `Dim`. See `Dim` for further details.

```
' Create an array
Common myGlobalArray[3 To 10][5]

' Create a global variable and assign it
Common myGlobalVar = 100
```

Connect

```
Connect( control id, identifier, event type, function name )
Connect( control id, event type, function name )
```

Connects *event type* to callback function. The function *function name* must take a

single argument, the `wxEvent`. In the case menus and others where the *control id* is not equal to the identifier, an *identifer* is required.

```
' Connect the menu wxID_EXIT event to the OnQuit callback
Connect( frame, wxID_EXIT, wxEVT_COMMAND_MENU_SELECTED, "OnQuit" )

' Connect the mouse left down event on the panel
' to the onLeftDown routine
Connect( panel, wxEVT_LEFT_DOWN, "onLeftDown" )
```

Const

`Const` *`name`* `=` *`expression`* `[, ... ]`

Creates constant \name with value of *expression*. Multiple constants can be declared on one line. Constants are global, and do not have to be declared with a `Shared` statement.

```
'Create a constant for PI
Const PI = 3.14159271
```

Continue

`Continue`

Jump back to the top of the loop. This works only for `For` and `While` loops.

```
'Don't process numbers greater than 5
For i = 1 to 10
    If i > 5 Then
        Continue
    End If
    Print i
End For
```

Cos

`n2` `= Cos(`*`n1`*`)`

Return the cosine of *n1*, where *n2* is in radians.

```
' n is 1
n = Cos( 0 )
```

Date$

`string` `= Date$()`

Returns date string in `MM:DD:YYYY` format. You can also use `Date()`.

```
' s holds the current date
s = Date$( 0 )
```

Declare

```
Declare Function functionName( arglist )
Declare Sub subName( arglist )
```

Create a forward reference to *functionName* or *subName*. This allows the routine to be referenced in code before it is actually defined with `Function` or `Sub`.

You still can't actually *execute* the routine until it is actually declared. If you need to encounter problems with forward references that `Declare` won't fix, try adding `Option QBasic`.

```
Option QBasic

Declare mySub()

' Option QBasic causes these statements to be deferred
mySub()
End

Sub mySub()
    Print "This is my sub"
End Sub
```

Delete

```
Delete object
```

Delete a `wxWindows` object. Only `Delete` objects that were created with `New`. Objects created "on the stack" are automatically destroyed when the routine that created them exits. Refer to `class`() for details.

```
' Destroy a window
Delete myWindow
```

Dim

```
Dim name[]
Dim name[ maxValue ] [[...]] [, ... ]
Dim name[ minValue To maxValue ][[...]] [, ...]
Dim name [ = expression ] [ , ... ]
```

Create an array or variable named *name*. Arrays can have up to 5 indexes, and are initialized to zeros. Variables can optionally be assigned by `Dim` statements.

If no indexes are specified, a *dynamic array* will be created. This is an array that is similar to those implemented in Awk. Indexes can be any value – numeric or strings – but they will be stored internally as strings. For example:

```
a[1] = "one"
```

and

```
a["1"] = "one"
```

are identical. You can have as many indexes as you wish, but the result is simply combined into a single string. For example:

```
a["this",22] = "some value"
```

is the same as writing:

```
a["this,22"] = "some value"
```

If you request an index from a dynamic array that has not been assigned, it will return an empty string "".

Dynamic arrays also differ from static arrays in that that elements can only be passed by value, not by reference.

```
' Create an array
Dim a[3 To 10, 5]

' Create a variable and assign it
Dim myVar = 100
```

End

```
End
```

Terminates program execution.

```
'Exit program
End
```

Erase

Erase *array*[]
Erase *array*[...]

If no indexes are specified, erases all the array. For static arrays, this means resetting all the values to zero. For dynamic arrays, this means deleting all the keys and and values. For example:

```
'Create a static array
Dim a[10]

' Initialize it
For i = 1 To 10
    a[i] = i
End For

' Reset it to zeros
Erase a[]
```

If indexes are given, only the specified index will be reset. For static arrays, this is the same as setting the value to zero:

```
'Create a static array
Dim a[10]

' Initialize it
For i = 1 To 10
    a[i] = i
End For

' Same as a[3] = 0
Erase a[3]
```

On the other hand, if the array is dynamic, the key and value are actually removed, so the array shrinks:

```
'Create a dynamic array
Dim a[]

' Initialize it
For i = 1 To 10
    a[i] = i
End For

' Remove element from array
Erase a[3]
```

Eof

n = `Eof(` *file handle* `)`

Returns true if at end of *file handle* has been reached.

```
' Read until the end of file
```

```
Open "readme.txt" For Input as #1
While Not Eof( 1 )
    Line Input #1, text
    Print text
Wend
Close #1
```

Fix

`n2 = Fix( n1 )`

Returns integer portion of *n1*. Same as `Floor`().

```
' n is 3
n = Fix( 3.1415927 )
```

Floor

`n2 = Floor( n1 )`

Returns integer portion of *n1*. Same as `Fix`().

```
' n is 3
n = Floor( 3.1415927 )
```

For ... Next

For ... End For

```
For variable = start To finish [Step increment]
   commands
   [Exit For]
   [Continue]
End For | Next
```

Iterate *variable* from *start* through *finish*, incrementing by *increment*. The *commands* are executed each iteration through the loop.

```
'Print the odd numbers from 1 to 10
For i = 1 To 10 Step 2
    Print i
End For
```

For Each ... Next

For Each ... End For

```
For Each variable In array
   commands
   [Exit For]
   [Continue]
End For | Next
```

Iterate through *array,* sequentially assigning the key from each *array* index to *variable.*

```
' Create a dynamic arrays
Dim number[]

' Fill it with values
number[1] = "one"
number[2] = "two"
number[3] = "three"

' Display the keys and values
For Each key in number
    Print key, number
End For
```

Frac

```
n2 = Frac( n1 )
```

Returns non-integer portion of *n1*.

```
' n is 0.1415927
n = Fix( 3.1415927 )
```

FreeFile

```
n = FreeFile()
```

Returns next available file number, or zero if no slot is free.

```
' open file "myfile.txt" on next available slot
fileNum = FreeFile()
Open "myfile.txt" For Output As #fileNum
```

Function ... End Function

```
Function functionName( parmlist [...] )
    Shared variable [, ...]
    Dim variable [, ...]
    commands
```

```
    [Exit Function]
    [Return Function]
    [functionName = expression]
End Function
```

Create a user-defined function named *functionName*. Functions can optionally return values, specified by the `Return` statement. The use elipsis (...) indicates the the function can take a variable number of arguments. You can mix required with optional args.

The total number of arguments can be found by calling function `Argc`(), and the function `Argv`() returns a particular argument passed to the caller.

```
    'sum the args passed
    Function sumNumbers( a )
        total = 0
        For i = 1 to Argc
            total = total + Argv( i )
        End For
        Return sum
    End Function
```

Hex$

```
string = Hex$(n)
```

Returns the hexidecimal representation of the number *n*. Only the integer portion of the *n* will be used.

```
    ' n is "c"
    n = Hex$( 12 )
```

If...ElseIf...Else...End If

```
If test Then
   commands
[ElseIf test Then
   commands]
[Else
    commands]
End If
```

Conditionally execute code. There can be any number of `ElseIf` clauses. `ElseIf` can also be spelled `ElsIf`.

```
    'Determine if a number is odd or event
    If Floor(a / 2) = a / 2 Then
        Print a; " is Even"
    Else
        Print a; " is Odd"
```

```
End If
```

In

expression `In` *array*`[]`

Returns `True` if *expression* is a key in the dynamic array `array[]`. Using a static array will result in an error.

```
'Check if key is an index in array a[]
If key In a[] Then
   Print key; " is a valid index in a[]"
End If
```

Include

`Include` *filename*

Insert *filename* into source code. This statement cannot be placed in a structure.

```
'Include the file "defs.inc"
Include "defs.inc"
```

Indexes

count `= Indexes(` *array*`[] )`

Return the number of indexes in *array*.

```
' Display information about array's indexes
count = Indexes( a[] )
Print "There are "; count; " indexes in the array"
For i = 1 To count
    bottom = LBound( a[], i )
    top = UBound( a[], i )
    Print "Index "; i; " is from "; bottom; " to "; top
Next
```

Instr

n `= Instr(` *substring*, *string* `)`

Returns position of substring in string. If no match, returns zero.

```
' sets n to 2
n = Instr( "Basic", "wxBasic" )
```

Int

```
n2 = Int( n1 )
```

Returns integer portion of *n1*.

```
' set n to 12
n = Int( 12.44 )
```

LBound

```
n = LBound( array[], index )
```

Returns the lower bound of an *array*'s *index*.

```
' Display information about array's indexes
count = Indexes( a[] )
Print "There are "; count; " indexes in the array"
For i = 1 To count
    bottom = LBound( a[], i )
    top = UBound( a[], i )
    Print "Index "; i; " is from "; bottom; " to "; top
Next
```

LCase$

```
string2 = LCase$( string1 )
```

Returns string *string1* in lower-case.

```
' s is set to "wxbasic"
s = LCase$("wxBasic")
```

Left$

```
string2 = Left$( string1, n )
```

Returns the leftmost *n* characters from string string1.

```
' s is set to "wx"
s = Left$( "wxBasic", 2 )
```

Len

Length

```
n = Len( string )
```

n = Length(*string*)

Returns length of *string*.

```
' n is set to 7
n = Len("wxBasic")
```

Line Input

Line Input # *file handle*, *string*

Assign *string* to the next line of text in file *file handle*.

```
' Read the first 10 lines from "readme.txt" into an array
Dim text[10]
Open "readme.txt" For Input As #1
For i = 1 To 10
    Line Input #1, text[i]
Next
Close #1
```

Loc

position = Loc(*file handle*)

Returns position of the file being read. This only works if the file is opened in `Input` mode. This is the same as `Seek()`.

```
' Get current file position
Print "Current file position is "; Loc(1)
```

Lof

n = Lof(*file handle*)

Returns the length of the file. This only works if the file is opened in `Input` mode.

```
' Get  length of file myfile.txt
filename = "myfile.txt"
Open filename For Input As #1
Print "The file "; filename; " is "; Lof(1); " bytes long"
Close( 1 )
```

Log

n2 = Log(*n1*)

Returns natural log of *n1*.

```
' n is set to 2.302585
n = Log( 10 )
```

LTrim$

string2 = LTrim(*string1*)

Returns *string1* with whitespace (spaces and tabs) removed from left side.

```
' s is set to "wxBasic"
s = LTrim$( "     wxBasic" )
```

Mid$

string2 = Mid$(*string1*, *start*, *length*)

Returns substring of *string1* starting at *start*, *length* characters long.

```
' s = "Ba"
s = Mid$("wxBasic", 3, 2 )
```

New

handle = New *class*(*arglist*)

Create a new instance of a `wxWindows` object belonging to `class`. Refer to the `wxWindows` documentation for details of classes. Objects created with `New` should be destroyed with the `Delete` statement.

Note that in the example, the `wxPoint` and `wxSize` objects are created "on the stack". Refer to *class*() for details.

```
' Create a new window
frame = new wxFrame( 0, -1, "My Window",
    wxPoint(50, 50), wxSize(250, 140))
```

Open

Open *filename* For Input | Output | Append As #*filenumber*

Opens file *filename* for reading or writing. `Input` opens a file for reading, `Output` opens a file for writing, and `Append` opens an existing file for writing, appending the output.

```
' Print 10 numbers to the file "temp.txt"
Open "temp.txt" For Output As #1
```

```
For i = 1 to 10
    Print #1, i
Next
Close #1
```

Option Explicit

`Option Explicit`

Ordinarily, wxBasic creates variables automatically when a value is assigned to one. `Option Explicit` requires that variables be explicitly declared with a `Dim` or `Option Shared` statement.

This should be placed as the first statement in your file. It effects all following statements.

```
' Require all variables be explicitly declared
Option Explicit
```

Option NoConsole

`Option NoConsole`

Ordinarily, wxBasic will send output of the `Print` statement to an emulated console window. Option NoConsole will prevent the console from appearing.

```
' Prevent console from appearing
Option NoConsole
```

Option QBasic

`Option QBasic`

Ordinarily, wxBasic acts like `C` - each statement not in a `Function` or `Sub` is executed when it is encountered.

`Option QBasic` defers the execution of these statements until after the entire file has been parsed. Basically, it allows you to declare the main section of your code *before* the routines that it uses have been declared.

Note that `Constant` is *not* deferred. See also `Declare`.

```
' Example 1: not deferred

Sub mySub()
    Print "This is my sub"
End Sub
```

```
' The main routine
mySub()
End


' Example 2: Deferred
Option QBasic
Declare mySub()

' The main routine
mySub()
End

Sub mySub()
    Print "This is my sub"
End Sub
```

Print

`Print` *`expression`* `[,][;][` *`expression`* `]`

Evaluate and print expressions, and send output to the console window. Expressions can be seperated by a semicolon {;} or a comma {,}. A comma will place a tab between the two expressions. Placing a semicolon at the very end will prevent a line return from being printed.

```
'Print the current value of a
Print "a = "; a
```

Print

`Print #` *`file handle`*`,` *`expression`* `[,][;][` *`expression`* `]`

Evaluate and print expressions to *file number*. This works the same as `Print`, but sends the output to the requested file.

```
' Print "Hello, wxBasic" to the file "temp"
Open "temp" For Output as #1
Print #1, "Hello, wxBasic"
Close #1
```

Randomize

`Randomize( [`*`n`*`] )`

Seeds pseudo-random number generator with *n*. If *n* is not given, it will use the current time as the seed.

' Seed the random number generator based on the time
Randomize(Timer)

ReadByte

```
n = ReadByte( file handle )
```

Returns byte from current position in *file handle*.

```
' Display "myfile.txt" byte by byte
Open "myfile.txt" For Input As #1
While Not Eof( 1 )
    Print Seek( 1 );,;
    Print Chr$(ReadByte(1))
End While
Close(1)
```

Reverse$

```
string2 = Reverse$( string1 )
```

Returns \string1 reversed.

```
' Set s to "4321"
s = Reverse("1234")
```

Right$

```
string2 = Right$( string1, n )
```

Returns the rightmost \n characters from *string1*.

```
' Set s to "Basic"
s = Right$( "wxBasic", 5 )
```

RInstr

```
n = RInstr( string, substring )
```

Returns position of *substring* in *string*, searching from the right.

```
' set n to 3
n = RInstr( "wxBasic", "Basic" )
```

Rnd

n2 = Rnd(*n1*)

Returns random number between 1 to *n*.

```
' Set n to a random number between 1 and 10
n = Rnd( 10 )
```

Round

n2 = Round(*n1*)

Returns *n1*, rounded down to the nearest integer.

```
' Set n to 12
n = Round( 12.3 )
```

RTrim$

string2 = RTrim$(*string1*)

Returns *string1* with whitespace removed from right side.

```
' Set s to "wxBasic"
s = RTrim$( "wxBasic    " )
```

Seek

position = Seek(*file handle*)
success = Seek(*file handle*, *position*)

Returns the current position in *file handle*. If the optional argument *position* is included, moves to that position in the file, returning `True` if succeeding. See also `Loc()`.

```
' Print "myfile.txt" byte by byte
Open "myfile.txt" For Input As #1
While Not Eof( 1 )
    Print Seek( 1 );
    Print Chr(GetByte(1))
End While
Close(1)
```

Select ... End Select

Select Case *expression*

```
Case tests
    commands
[Case tests
    commands ]
[Case Else
    commands]
End Select
```

Conditionally execute code, depending on if expression matches tests. Tests can be strung together with commas, and the forms are:

lowValue `To` *highValue*
`Is` = *expression*
`Is` <> *expression*
`Is` > *expression*
`Is` < *expression*
`Is` <= *expression*
`Is` >= *expression*

```
' Print the name of a number
Select Case n
Case 1
    Print "One"
Case 2
    Print "Two"
Case 3
    Print "Three"
Case 4, 5, 6
    Print "Four, Five or Six"
Case Is < 10
    Print "Less than 10"
Case Else
    Print "Too big!"
End Select
```

Sgn

```
n2 = Sgn( n1 )
```

Returns 1 if *n1* is positive, -1 if it's negative, and 0 if it's zero.

```
' Print sign of n
Select Case Sgn( n )
Case 1
    Print "The number is positive"
Case -1
    Print "The number is negative"
Case 0
    Print "The number is zero"
End Case
```

Shared

Shared *variable* [, *variable*...]

Use variables declared at the module level.

```
' Declare a module level variable
Dim counter = 0

Sub incrementCounter()
    ' Make the counter shared
    Shared counter

    ' Increment the counter
    counter = counter + 1
End Sub
```

Shell

Shell(*command string*)

Passes *command string* to shell to be executed. This command is operating system dependant.

```
' In DOS, delete all files ending with .TMP
Shell( "del *.tmp" )
```

Sin(n)

n2 = Sin(*n1*)

Returns sine of *n1*.

```
' Set n to 0.841471
n = Sin( 1 )
```

Space$

string2 = Space$(string1)

Returns string built of *n* spaces.

```
' Set s to "   " (three spaces)
s = Space$( 3 )
```

Sqr

n2 = Sqr(*n1*)

Returns square root of *n1*.

```
' Set n to 3
n = Sqr( 9 )
```

Str$

string = Str$(*n*)

Returns string representation of *n*.

```
' Set s to "123"
s = Str$( 123 )
```

String$

string = String$(*count*, *char*)

Returns *string* length *count* made up of *char*. *char* can also be a numeric ASCII code.

```
' Set s to "---" (three dashes)
s = String$( 3, "-" )
```

Tan

n2 = Tan(*n1*)

Returns tangent of angle *n1*.

```
' Set n to 0.557408
n = Tan( 0 )
```

Ticks

n = Ticks()

Returns number of ticks since beginning of program. Ticks are operating system dependant.

```
' Set n to number of ticks to perform loop
startTicks = Ticks()
For i = 1 To 1000
```

```
Next
n = Ticks() - startTicks
```

Time$

string = Time$()

Returns current time in `HH:MM:SS` format.

```
' Set s to current time
s = Time$()
```

TypeOf$

string = TypeOf$(*expression*)

Returns name of datatype *expression* evaluates to. Types are “`number`”, “`string`”, “`object`” and “`unknown`”.

```
' Set s to "string"
s = TypeOf$( "wxBasic" )
```

UBound

number = UBound(*array*[], *index*)

Returns the upper bound of *array*'s *index*.

```
' Display information about array's indexes
count = Indexes( a[] )
Print "There are "; count; " indexes in the array"
For i = 1 To count
    bottom = LBound( a[], i )
    top = UBound( a[], i )
    Print "Index "; i; " is from "; bottom; " to "; top
Next
```

UCase$

string2 = UCase$(*string1*)

Returns string converted to upper case.

```
' Set s to "WXBASIC"
s = UCase$( "wxBasic" )
```

Val

```
n = Val( string )
```

Returns value of number contained in string.
If the string has no value, the result is zero.

```
' Set n to 12
n = Val( "12" )
```

While ... Wend

While ... End While

```
While expression
    commands
    [Exit While]
    [Continue]
Wend | End While
```

Execute *commands* while *expression* remains true.

```
'Count from 1 to 10
i = 1
While i <= 10
    Print i
    i = i + 1
Wend
```

WriteByte

```
WriteByte( file handle, n )
```

Write byte *n* to current position in *file handle*. This is primarily used for writing binary files, since strings can't contain `ASCII 0`.

```
' Binar copy file
Open "myfile.txt" For Input As #1
Open “newfile.txt” For Output As #2
While Not Eof( 1 )
    WriteByte( 1, ReadByte(2))
End While
Close
```

Overview of the Interpreter

At a high level, the interpreter performs the following functions:

- The *lexer* reads source code, breaking it into *tokens*.
- The *interpreter* then matches the tokens against the *grammar,* generating an internal representation for the code in the form of a *parse tree*.
- The *parse tree* consists of a series of connected *nodes*, each representing some action or data, tagged with an identifying *opcode*.
- The *parse tree* is then traversed by the *evaluator*, which performs the appropriate *action* for each *opcode* it encounters.

The Lexer

The job of the lexer is to break strings of text into *tokens*, and feed them to the *parser*. The lexer for wxBasic can be found in `lexer.c`.

Each token is tagged with an identifier telling what class it belongs to, as well as an optional value.

Ignoring keywords and symbols, the following types of tokens are recognized by the lexer:

- `Float` is a numeric value. All numbers scanned by the lexer are returned in this format, even if they are integers.

- `String` is a "string" of alpha-numeric characters. Strings are delimited by double quotes. Special characters are include by using a backslash, such as `\n` (newline) or `\t` (tab).

- `FunctionName` is a user-defined function.

- `BuiltinName` is a built-in function found in the `Symbol` table. See `builtin.c` for examples.

- `ConstantName` is a user-defined constant, found in the `Symbol` table.

- `VariableName` is a user-defined variable, found in the `Symbol` table.

- `ArrayName` is a user-defined array, found in the `Symbol` table.

- `ClassName` is a wrapped C++ class, found in the `Symbol` table. Refer to `wrap.cpp` for examples.
- `MethodName` is an non-numeric token prefixed with a `'.'` character, such as `.setVisible` in `myWindow.setVisible( True )`.
- `Undefined` is any token not falling into one of the prior categories.

`Float`, `String` and `MethodName` can all be identified by some characteristics (quotes, digits, etc.). All other types are identified by looking them up in the `Symbol` table.

A token's value can be one of the following, except for node, which is used exclusively by the parser.

```
%union {
    int     iValue;         /* integer value */
    Number  fValue;         /* float value */
    char    *strValue;      /* string pointer */
    Node    *node;          /* parse node */
    Symbol  *symbol;        /* symbol */
}
```

The Parser

The parser attempts to match tokens against gramatical patterns. The parser is written in `Yacc`[1].

The grammar file for wxBasic can be found in `wxbasic.y`. This file is processed by `Yacc` to create the parser, found in `y_tab.c`. The top level grammar for a wxBasic program is recursively defined as:

```
program:
      program statements
    | /* nothing */
    ;
```

After wxBasic parses a block of statement, it evaluates it, and then frees the memory held by the representation. The definition of statements is:

```
statements:
      End If
```

1 YACC is an acronym for *Yet Another Compiler Compiler*, a popular tool for such tasks. Strictly speaking, I used Bison, the free GNU clone of Yacc.

```
        | End For
        | End Function
        | End While
        | End Select
        | Next
        | Wend
        | sep statements
        | statement statements
        | /* nothing */
        ;
```

The `'|'` represents a logical "Or". Note that all the structure terminators are defined here. This allows a structure to be defined as:

```
    | While expr sep statements sep
```

Encountering an `End` will cause the parsing of that block to terminate.

The definition of statement is the meat and potatoes of the grammar:

```
    statement:
          '~' expr '(' ')' sep
        | '~' expr error
        | '~' expr '(' error
        | expr sep
        | ConstantName Eq
        | FunctionName '('
        | FunctionName

        etc...
```

For each grammatical pattern, there is a corresponding action to take when that grammar is encountered. For example, here's the grammar for a number:

```
    | Float
        { $$ = floatNode( OpFloat, $1, NULL, NULL ); }
```

The first line contains the grammar pattern, while the following lines contain the C code to execute. The `$1` refers to `Float`, the matched token.

Parse Tree Nodes

In wxBasic, parse tree nodes have five elements: `opcode`, `value`, `left` node, `right` node and `trace`. There is a `next` field defined, but it not currently being used.

- `opcode` tells what action is associated with the node.
- `value` is the optional value associated with the node.
- `left` is the optional left branch of the node, pointing to another node. If the branch is empty, it holds a.`NULL` value.

- `right` is the optional right branch of the node, pointing to another node. If the branch is empty, it holds a `NULL` value.
- `trace` points back to the source code

Readers paying attention at this point will notice that each node is simply a binary tree - a node with two leaves. So the value

```
123
```

would look like:

```
node->opcode = OpFloat;
node->value = 123;
node->left = NULL;
node->right = NULL;
```

(For the sake of space, fields that are not used will be ignored in the examples.)

Parse Trees

Expressions are essentially recursively defined. For example, here's the definition of multiplication from the YACC file:

```
| expr '*' optSep expr
    { $$ = opNode( OpMul, $1, $4 ); }
```

`optSep` is an optional seperator that allows expressions to span multiple lines. Multiplication is encoded in a node as:

```
node->opcode = OpMul;
node->left = $1;
node->right = $4;
```

So writing:

```
12 * 42
```

would be encoded (more or less) as:

```
Node *left_node, *right_node, *parent_node;

left_node->opcode = OpFloat;
left_node->value = 12;

right_node->opcode = OpFloat;
right_node->value = 42;

parent_node->opcode = OpMul;
parent_node->left = left_node;
```

```
    node->right = right_node;
```

As wxBasic is parsed, a parse tree representing the code is created. It is the task of the evaluator to execute that parse tree.

The Evaluator

In most languages, the parse tree structure merely serves as an intermediate structure which is then converted into a more efficient form, such as bytecodes.

This is *not* the case with wxBasic. Rather than transforming the parse tree, it executes it directly. This task is accomplished by the `eval()` routine. Basically, it's just a large switch statement:

```
void eval( Node *node )
{
    switch( node->op ) {
    case NULL:
        break;

    case OpAdd:
        /* code for OpAdd */
        break;

    case OpAnd:
        /* code for OpAnd */
        break;

    case OpArgList:
        /* code for ArgList */
        break;

    etc...
```

wxBasic uses a stack to pass and store values. Here's a simplified version of the evaluator, with only `OpFloat` and `OpMul` implemented:

```
void eval( Node *node )
{
    Number  n1, n2;

    switch( node->op ) {
    case OpFloat:
        pushNumber( node->value.fValue );
        break;

    case OpMul:
        eval( node->left );
```

```
            eval( node->right );

            n2 = popNumber();
            n1 = popNumber();

            pushNumber( n1 * n2 );
            break;
        }
    }
```

Given our example from above:

```
    12 * 42
```

encoded as:

```
    left_node->opcode = OpFloat;
    left_node->value = 12;

    right_node->opcode = OpFloat;
    right_node->value = 42;

    parent_node->opcode = OpMul;
    parent_node->left = left_node;
    node->right = right_node;
```

The node passed for evaluation would be `parent_node`, the parent node. `OpMul` is evaluated. The left hand node, `left_node`, is evaluated first:

eval(node->left);

This executes the `OpFloat`, which places the value `12` on the stack. Then right hand node `right_node` is then executed, which places the value `42` on the stack.

```
    eval( node->right );
```

The values are then popped off the stack and stored into local variables:

```
    n2 = popNumber();
    n1 = popNumber();
```

Finally, they are multiplied, and the result placed back on the stack for another operator to process.

```
    pushNumber( n1 * n2 );
```

The Inner Workings

Internal Values

The basic datatypes that wxBasic uses are:

- `Number`, a C `double`.
- `String`, a C `char*`.
- `Array`, a pointer to a C `*Array` structure.
- `Variant`, a pointer to a C `*Variant` structure.

The actual *value* is stored in the `Value` data structure, which is a `union` of the above types. This value is placed into a `Variant` data structure. The possible types that a `Variant` datatype can hold are:

• `DATA_UNDEFINED`.	Unassigned value.
• `DATA_REF`.	Data passed by reference, stored in `Variant*`.
• `DATA_NUMBER`	Numeric data, stored in `Number`.
• `DATA_STRING`.	String data, stored in `String`.
• `DATA_ARRAY`	Array data, pointed to by `*Array`.
• `DATA_OBJECT`	Not used yet.

The Data Stack

Data is stored in wxBasic on a LIFO (Last In, First Out) stack. As items are placed on the stack, it expands upwards, toward the top of stack:

```
first, second, third
```

Most operations (such as addition, subtraction, comparison, and the like) are performed using data on the stack. For example:

```
print 123 + 456
```

would first cause `123` to be pushed on the stack:

```
123
```

Then `456` would be pushed on top:

```
123, 456
```

The addition operation would take the top items from the stack, and replace them with the result:

```
579
```

And finally, the `Print` statement would fetch the result from the stack for display.

Since the stack can hold `Variant` data, strings are the other literal values that it can hold:

```
123, "hello, world"
```

References on the Stack

In addition to holding literal values, values can be passed by reference on the stack. For example, in the following code:

```
function foo( bar )
    grill( bar )
end function
```

The call to `grill`() passes the value stored in the variable bar by reference. Instead of placing the *value* of `bar` on the stack, the reference *points back* to the original value:

```
123, "hello, world!", address of bar
```

In the case where a reference points to a reference, each reference *always* points back to the original reference. This avoids having to walk a chain of references in order to resolve a value.

Code Blocks

A code block is a series of commands. For example:

```
a = 12
b = 33
c = a + b
```

is a *block* containing three instructions, to be executed in order. This is accomplished by treating the nodes as if they were a linked list. The `OpComma` opcode is used for this purpose. Basically, actions are chained together in a binary tree. For example:

```
command 1
command 2
command 3
```

Would be encode as:

```
n1->opcode = OpComma
n1->left = pointer to command 1
n1->right = n2
```

```
n2->opcode = OpComma
n2->left = pointer to command 2
n2->right = n3

n3->opcode = OpComma
n3->left = pointer to command 3
n3->right = NULL
```

The implementation of `OpComma` in `eval`() uses a `while` statement to avoid the overhead of recursion.

Complex Operations

Most operations are a bit more complex that the `OpAdd` operation, but just combinations of data items glued together in linked list chains (via `OpComma`) or binary trees. For example, here is a typical `Print` statement:

```
Print 12+23, "foo"
```

The grammar of the `Print` statement is:

```
PRINT [expressionlist] [{; } ,}]
```

where *`expressionlist`* is a list of one or more expressions to print.

The `Yacc` implementation (in `wxbasic.y`) is:

```
| Print printlist sep
    {   $$ = opNode( OpPrint, NULL, $2 ); }
```

The `left` branch holds the `file number` to print to. `NULL` indicates the output will be routed to the console. The `right` branch holds the `printlist`, which is a list of items to print, along with their seperators.

Each element to be printed is stored in a `node`. The `left` branch of these nodes points to the value to print. The `value` field of the node holds a flag indicating what type of seperator to print after the value, `PRINT`, `PRINT_TAB` or `PRINT_NEWLINE` . The `right` branch points to the next node in the list:

```
node->opcode = OpPrint
node->value = seperator flag
node-> = node containing value to print
node->right = node containing next item to print
```

The implementation in `Yacc` is straightforward. The constants are used to flag what type of seperator to print after the data element is displayed:

```
printlist:
        ';' printlist
          { $$ = $2 }

     | ',' printlist
          { $$ = intNode( OpPrintValue, PRINT_TAB, NULL, $2 ); }

     | expr ',' printlist
          { $$ = intNode( OpPrintValue, PRINT_TAB, $1, $3 ); }

     | expr ';' printlist
          { $$ = intNode( OpPrintValue, PRINT, $1, $3 ); }

     | expr
          { $$ = intNode( OpPrintValue, PRINT_NEWLINE, $1, NULL ); }

     | /* nothing */
          { $$ = NULL; }
     ;
```

An `If ... Then` statement is similarly coded. The tests are chained together using `OpComma`. The `left` leaf of the node contains the test, and the `right` node the action to take if the test evaluates to true:

```
node->opcode = OpIf
node-> = node containing test
node->right = node containing action
```

Overview of the Modules

CORE.C

This is the toplevel program. The main code is placed in wxBasic, so it can optionally compile it without wxWindows.

The code that tries to determine the current working directory isn't likely to be portable, but it appears to work under Windows and Linux.

Also, this is where the application determines if the application is a bound executable. It takes a look at the last 10 bytes, looking for a tag of the form:

```
//nnnnnnnn wxbind
```

where `nnnnnnnn` is the decimal offset into the file where the source file begins.

SHARED.H

Forward references of types, global variables. For example, `NINDEX` is the maximum number of indexes an array can have.

I don't think `NOPTARGS` is used anymore; it should probably be removed.

ERROR.C

Rewrites of `printf`, `malloc` and `free`. There are two versions of the memory routines - 'safe' ones that check their values and abort on errors, and 'debugging' versions that check to make sure the application isn't misusing memory.

The debugging routines basically keep a linked list of allocated memory chunks. Nothing really fancy. `#define __DEBUG__` in `core.c` to activate them. Debugging mode adds a *lot* of overhead and slows down the program, so don't use it in production.

STACK.C

Integer LIFO stack routines, used by `YACC` to keep track of the control structure (`For`, `If`, `Case`). So if you write something like:

```
For i = 1 To 10
Wend
```

The interpreter can issue a nice message like

```
Expected "End For", not "End While"
```

instead of the unhelpful sort of "unmatched paren" error that C issues.

It's also used by `YACC` to track of the number of parameters passed to a routine. Some (but not all) parameter counts are checked at compile time.

DATA.C

The basic datatypes that wxBasic recognizes. They are:

- `Number`, a `double`.
- `String`, a `char*`.
- `Array`, a `*Array`.
- `Variant`, a `*Variant`.

The data is stored in a `Variant` type, which is a `union` of the above types.

All data (except for arrays) is stored on the stack. Local variables are placed on the stack, and grow upward. Global variables grow down. If the two collide, wxBasic will be forced to shut down. It can't dynamically resize the stack because globals are stored at *absolute* addresses. If I used negative values, I could fix that, but it would incur an overhead of having to resolve offsets.

The variable `localstart` keeps track of where the current local data is. The layout is:

- `localstart-1`: return value of function
- `localstart+0`: count of parameters
- `localstart+1`: first local variable
- `localstart+2`: second local variable, etc...

Data is retrieved from the stack via the routines `popNumber()` and `popString()`. These routines automatically convert data to the expected types. So if a `string` was expected but a number was on the top of stack, the `number` will be converted into a `string`.

The routine `getStackPointer()` resolves a `DATA_REF`. Initially, I only allowed data to be passed by value. The addition of passing by reference broke a lot of code that just took a look directly at the stack. I recently fixed a bug in the `CASE` statement of this type.

Data is placed on the stack via `pushNumber()`, `pushStringCopy()` and `pushString()`.

SYMBOL.C

This is the wxBasic dictionary. The following types of symbols exist:

`SYM_KEYWORD`	Reserved keyword
`SYM_VARIABLE`	Variable name
`SYM_ARRAY`	Array name
`SYM_CONSTANT`	Constant name
`SYM_BUILTIN`	Builtin routine name
`SYM_FUNCTION`	User defined function name
`SYM_CLASS`	Wrapped wxWindows class name

Symbols are stored in a linked list, accessed through the prior field. `lastSymbol` contains the last symbol in the list. At some point, I'll probably change this to a hash table.

The scope of a symbol determines what it belongs to. For example:

```
function foo( a, b )
   c = 12
end function
```

would have:

```
foo->scope = NULL
a->scope = foo
b->scope = foo
c->scope = foo
```

Symbols with a NULL scope are global. Children are linked to their parents via a linked list:

```
foo->child = a
a->sibling = b
b->sibling = c
c->sibling = NULL
```

VAR.C

This is where simple (non-array) variables are defined and accessed. All simple variables are created on the stack. Constants are simply variables that cannot be altered. The creation routines are:

```
createVar( char *name, int symType, int unique )
```

```
createConst( char *name )
createParm( char *name, int symType )
```

The variables can be fetched and set via `setVar()` and `getVar()`. These routines resolve references.

ARRAY.C

Arrays are basically just arrays of `Variant` data. Since arrays can be passed to routines, they have to keep track of the number of indexes they have, as well as the upper and lower bounds of each index. Range checking of arrays passes to routines is done at runtime.

Array indexes can be passed by reference, which leads to some of the most oblique code in wxBasic.

You can't currently `ReDim` arrays, but I'll get around to fixing that. I'm a bit nervous that a user will pass an array index, and then `ReDim` the array to a smaller size. Reference the index, and *boom*. However, you can use Dynamic arrays safely, since they never return values by reference.

I may get over this.

“Dynamic” arrays are currently implemented as arrays of `ArrayCell`. Resizing in handled by `eRealloc()`. The current implementation is horribly inefficient, and will probably change with the next release. I'll either go with a hashtable sort of scheme, or (more likely) sort the data and use a binary search.

NODE.C

These routines are used by `YACC` to build an internal represenation of the source code. This is discussed elsewhere in the documentation, so I'll just mention that I've added (but not implemented) a next field, which should make `OpComma` obsolete.

BUILTIN.C

The functions that are "built in". Numbers are passed to routines on the stack, in reverse order. Here's a typical routine:

```
/* basACos: returns arccos of number */
void basACos()
{
    pushNumber( acos( popNumber() ) );
```

```
}
```

EVAL.C

These routines execute the internal representation of the code. The main routine is `eval()`, which is a huge case statement. Not suprisingly, it's recursive.

As an example, here's how division is implemented:

```
case OpDiv:
    eval( node->left );
    eval( node->right );

    n2 = popNumber();
    n1 = popNumber();

    /* should add division by zero check here... */
    pushNumber( n1 / n2 );
    break;
```

At some point, I'll probably replace the switch case statement with a bunch of function pointers.

The `OpCase` (and associated opcodes) is probably the hairiest opcode.

LEXER.C

This implements `yylex`, which is the lexer used by `YACC`. Nothing much special here. I could have used `LEX` to do the job, but I've heard that it's slow, and there are some ugly hacks in my lexer.

Anything that starts with the character `'.'` is assumed to be a method name. Strings start with a `"' character. Numbers start with a digit. Otherwise, the string is looked up in the symbol table to be resolved.

CLASS.C

This implements the interface to C++ classes. There are four structures:

- `WrappedObject` Pointer to a C++ object
- `WrappedClass` Description of a C++ class
- `WrappedMethod` Description of a C++ class method
- `Callback` Callbacks associated with a WrappedObject

All objects are stored in the `objectList` array. This is currently an array with a fixed size, so if too many objects are created, *boom*.

Each WrappedObject contains:

- `pointer` Pointer to C++ object, stored as an `int`
- classIndex Index to classList, an array of `WrappedClass`
- `id` Id assigned to C++ object by wxWindows
- `lastCallback` Pointer to `Callback` linked list

A `WrappedObject` is created and destroyed with the routines `addObject()` and `runDestructor()`.

The routine `popPointer()` is used to retrieve an object pointer from the stack. Currently, these are stored as `DATA_NUMBER`, but they should really be stored as `DATA_OBJECT`. The `popPointer()` routine ensures that the pointer belongs to the expected class.

Class information is held in the fixed length `classList` array. Being fixed shouldn't be a problem, since the number of wxWindows classes is known at compile time. Each `WrappedClass` contains:

- name Class name.
- super Index in classList of the superclass; zero if none.
- lastMethod Pointer to class methods, stored in linked list of WrappedMethod

Each WrappedClass has a linked list of class methods, pointed to by the lastMethod field in WrappedClass:

- `name` Name of the method
- `hash` Hashed name of the method
- `minArgs` Minimum number of arguments the method takes
- `maxArgs` Maximum number of arguments the method takes
- `routine` Pointer to wrapped method
- `prior` Next method in linked list

Because methods aren't resolved until runtime, the hash of the name is kept to make searching the linked list faster. In theory I could use as hash table here, but I don't know that it's any faster.

Objects that are not created by the `New` keyword are created `on the stack', so to speak. This emulates the C++ behavior of being able to create an unassigned object which is automatically destroyed when the routine exits. For example:

```
button = new wxButton( panel, -1, "Button", wxPoint(10, 10) )
```

creates a `wxPoint` object `on the stack'. The stack `createStack` is used to keep track of these objects. When a routine is called, a zero is placed on the top of the `createStack` to mark the start.

When an object is created `on the stack', it's index is placed on the `createStack` as well. When the routine exits, it calls `clearCreateStack()`, which successively pops the `createStack` and destroys all the objects on it until the terminating zero is reached.

CONNECT.CPP

This is the glue that connects wxWindows events to wxBasic. A pointer to a wxBasic routine is attached to a wxWindows event via the `::Connect` routine.

When an object receives an event, the routine `runCallback`() is activated. The event is tagged with the appropriate class wrapper, the `*Symbol` pointing to the wxBasic callback is retrieved, and the routine is executed.

Y_TAB.C

Code generated by `YACC`. The source file is `wxbasic.y`.

CONSOLE.CPP

This emulates a console, so the `Print` statement has something that can be printed to. The printing code is implemented in the routine `eConsole()`, in `error.c`.

The console is simply a `wxFrame` with a `wxTextCtrl` in it. The only reason for defining a class was to set up the callback to clear `consoleExists` when the console window is closed.

The following global variables are used:

- `consoleExists` If non-zero, the console already exists, and `theConsoleText` control can be written to.
- `theConsoleText` Pointer to the wxTextCtrl in the consoleClass control.

When the console window is closed, it is destroyed. This allows wxBasic to fall out of the main loop when the main window is closed. Otherwise, it would hang around as an

invisible task in the background.

It is possible to crash wxBasic by closing a window with an active timer, and leaving the console window open. The application will remain running, and the timer will attempt to run in the non-existant window.

The wxWindows Wrappers

The wrappers for wxBasic are automatically generated by a QBasic program called `wrap.bas`. It's about 900 lines long - a lot of it is hacked together, and it should be rewritten at some point. It's quite specialized, and only I'd be quite suprised if it worked with anything other than wxWindows method prototypes.

I'm in rewriting the code in `Awk`, but it's on hold for the moment, as other things need to be done first.

The Qbasic program wrap.bas reads the file text file `class.i` , which contains the prototypes for the class methods. These prototypes were gathered by cutting and pasting from the help files. If there's a better way to gather the information, I'd sure like to know about it.

Design Decisions

Since the wrappers could accept a variable number of parameters, it seemed logical to pass the parameters on a stack. The stack accepts only two datatypes: numbers and strings. Values passed back from the wrappers are passed by value. This means that methods that pass their values back through pointers in the parameter list will not wrap well.

The routines for pushing and popping values on the stack are defined in `data.c`.

The Wrapper File Format

The parser for the wrappers takes the view that if it doesn't know what an object is, it must be a pointer to some class. This isn't always the case: there are enums, typdefs and structs that all look like classes. To distinguish these, the `class.i` file includes not only classes, but `enums`, `typedefs` and `structs`. The basic file layout is:

- `enums`
- `typedefs`
- `structs`
- `classes`

Comments: //

```
// comment
```

The input file uses the C++ comment style `'//'`, starting in the first column. For example:

```
// this is a comment
```

%{

```
%{
```

This causes all C code between the `%{` and `%}` line to be written verbatim to the wrapper file.

%include

```
%include filename
```

The `%include` statement causes the include statement to be added to the wrapper code. If the filename is included more than one time, repeated instances are ignored.

For example:

```
%include "wx/calctrl.h"
```

%enum

```
%enum enum
```

This is used to prevent enums from being treated as classes. For example:

```
%enum wxSocketNotify
```

%typedef

```
%typedef name alias
```

This causes declarations of *name* to be replaced with *typedef*. For example:

```
%typedef wxNotebookPage wxWindow
```

causes `wxNotebookPage` to be replaced with `wxWindow`. This is typically used to reflect where wxWindows uses `typedefs` internally. These constructs might not be portable over various platforms.

%struct

```
%struct name
```

Structs are treated the same as classes - a statement:

```
int _struct = addClass( 0, "struct" );
```

is generated. For example, the following declares `wxResourceTable` to be a struct:

```
%struct wxResourceTable
```

%class

```
%class [%alias alias] name [,superclass]
```

where the *superclass* is optional. Currently, the `superclass` must be declared before the subclass. For example:

```
%class wxPoint
```

This creates a variable that holds the class index:

```
int _class = addClass( super, name );
```

It also causes the default destructor to be declared:

```
void <class>_del()
{
    delete (<class> *)me;
}
```

For example, the `wxPoint` class would create:

```
int _wxPoint = addClass( 0, "wxpoint" );

void wxPoint_del()
{
    delete (wxPoint *)me;
}
```

This is the behavior assigned to the classes' Delete method. The destructor is assumed to take no arguments, and return no values. The method is linked in with the `addMethod()` routine, which takes the class index, method name, function name, minimum number of args, and maximum number of args:

```
    addMethod( _wxPoint, "del", wxPoint_del, 0, 0 );
```

A *dummy* class is one that is declared so that the wrapper generator recognizes the name

as a pointer, but the actual class is never wrapped. The format is:

For example:

```
%class wxAcceleratorTable
```

%ctor

```
%ctor class alternateName( arglist )
```

Some classes have more than one constructor. Because wxBasic will coerce data to the expected type, it can only have a single default constructor.

In order to accomodate additional constructors, the `%ctor` flag is used. This creates a class that inherits the current class, and uses the alternate constructor.

For example, wxPen has several constructors. To accomodate them, they are given seperate names, and prefixed with `%ctor`:

```
%class wxPen, wxGDIObject
%include <wx/pen.h>

wxPen()
...
// alternate constructors
%ctor wxPenFromColour(const wxColour& colour, ... )
%ctor wxPenFromColor(const wxColour& colour, ... )
%ctor wxPenFromColourName(const wxString& colourName, ... )
%ctor wxPenFromColorName(const wxString& colourName, ... )
%ctor wxPenFromBitmap(const wxBitmap& stipple, ... )
%ctor wxPenCopy(const wxPen& pen)
```

The alternate constructors can be used as if they were a unique class:

```
' Create a blue pen
myPen = new wxPenFromColor( "Blue", 1, wxSOLID )
```

%builtin

```
%builtin type functionName( arglist )
```

The `%builtin` declaration allows wrappers to be created for methods that don't belong to a particular class. This declaration is done a single time, followed by all the methods that have no class.

For example (the ... here is for brevity, and is not part of the syntax):

```
%builtin int wxMessageBox(const wxString& message, ... )
```

creates a wrapper for the `wxMessageBox` that can be called as it were a native wxBasic function.

In addition to creating the standard wrapper, it creates a dictionary entry using the `addBuiltin()` routine:

```
addBuiltin( "wxmessagebox", builtin_wxMessageBox,  1,  6 );
```

Simple Classes

There are a number of 'simple' classes that don't inherit from `wxObject`. These can safely be created on the stack. `wxPoint` is a typical declaration:

```
%class wxPoint
wxPoint(int x, int y)
~wxPoint()
int x
int y
```

Currently, *attributes* (like `wxPoint`'s *x* and *y*) are not supported, and are ignored.

Class Constructors

The declaration of a class constructor is:

```
class ( arglist )
```

For example:

```
wxPoint(int x, int y)
```

The name of the class must match the name of the most recently declared class. There can only be one constructor for each class. This is the method that is assigned to `New`.

The constructor is assumed to always return a wrapped pointer of the created classes' object.

```
    // wxPoint(int x, int y)
    void wxPoint_new()
    {
        wxPoint *returns;
        // pop int y
        int y = (int)popNumber();
        // pop int x
```

```
        int x = (int)popNumber();

        // call wxPoint
        returns = new wxPoint(x, y);

        pushNumber( addObject( _wxPoint, (int)returns ) );
    }
```

The variable `argCount` contains the number of arguments that are passed. The routine assumes that the argument count is correct (the acceptable range is stored by `addMethod`). The routines `popNumber()` and `popString()` don't do type checking, since they automatically coerce their arguments to the right type. They are then cast to the proper types. For example:

```
const unsigned char blue = (unsigned char)popNumber();
```

and:

```
wxString colourName = popString();
```

`popPointer()` *does* check to make sure the pointer belongs to the proper class, and will generate an error if the class is wrong:

```
wxClassInfo *info = (wxClassInfo *)popPointer( _wxClassInfo);
```

If there routine accepts a variable number of args, the code will pop the proper number from the stack, or assign the default value:

```
coord yoffset = (argCount > 2 ? (coord)popNumber() : 0);
coord xoffset = (argCount > 1 ? (coord)popNumber() : 0);
wxList *points = (wxList *)popPointer( _wxList);
```

Class Destructors

If a class has a specific destructor, it is specified with the form:

```
~<class> ()
```

For example:

```
~wxObject()
```

This creates a destructor function of the form:

```
void class_dtor()
{
    ((class *)me)->~class();
    pushNumber( (Number)0 );
```

```
}

For example:

void wxPoint_dtor()
{
    // call ~wxPoint
    ((wxPoint *)me)->~wxPoint();

    // result is ignored
   pushNumber( (Number)0 );
}
```

The variable me contains the object's pointer.

Class Methods

All the statements following a `%class` statement not comments or starting with `'%'` are assumed to be method prototypes for that class. For example:

```
%class wxSize
...
GetWidth()
```

creates the wrapper:

```
void wxSize_GetWidth()
{
    int returns;
    returns = ((wxSize *)me)->GetWidth();
    pushNumber( (Number)returns);
}
```

Two special variables are used in methods:

- `me`: Pointer to object
- `argCount` : number of args being passed

All method wrappers are void - parameters are passed through a stack. Values are pushed onto the stack with `pushNumber()` and `pushString()`, and popped off with `popNumber()`, `popString()` and `popPointer()`. For example:

```
const unsigned char blue = (unsigned char)popNumber();
wxString colourName = popString();
wxClassInfo *info = (wxClassInfo *)popPointer( _wxClassInfo);
```

`popNumber()` and `popString()` will attempt to coerce the value on the stack to the proper datatype. However, `popPointer()` *requires* a number, and checks to make sure that the class belongs to, or is derived from the specified class.

All methods return a value on the stack. `Void` methods push a zero on the stack, which is typically discarded by the caller:

```
// result is ignored
pushNumber( (Number)0 );
```

`addMethod()` then links the method to the class:

```
addMethod( _wxSize, "getwidth", wxSize_GetWidth,  0,  0 );
```

The parameters to `addMethod()` are:

- class index
- method name (in lower case)
- routine name
- minimum number of args
- maximum number of args

Decoding The Method Type

virtual

The keyword `virtual` is ignored.

static

The keyword `static` is ignored.

const

The keyword `const` in the function return type causes the variable returns to be declared as as:

```
const type returns;
```

instead of:

```
type returns;
```

signed/unsigned

This is added to the type declaration of returns. For example:

```
signed int foo()
```

will create the declaration:

```
signed int returns;
```

* (pointer)

The `*` is prepended to the returns declaration. For example:

```
foo **bar()
```

will create the declaration:

```
foo **returns;
```

& (deref)

The `&` causes returns to be declared as a pointer and prefixes the return value with a dereference operator. For example:

```
%class myClass
foo &bar()
{
   return grill();
}
```

will create the routine:

```
void myClass_bar()
{
    foo *returns;
    returns = &grill();
    pushNumber( addObject( _foo, (int)returns ) );
}
```

Decoding The Method Args

Optional Args

An `=` in the arg will cause the argument to be read as optional, with the value following the `=` as the value to assign. The variable `argCount` holds the number of values on the stack, and controls what objects are popped off. For example:

```
const wxString& name = "panel"
long style = wxTAB_TRAVERSAL
```

will be converted to:

```
wxString name = (argCount > 5 ? popString() : "panel");
long style = (argCount > 4 ? (long)popNumber() : wxTAB_TRAVERSAL);
```

If the `type` is a reference pointer and is not a wxString, a dereference operator is prepended to the argument. For example:

```
const wxSize& size = wxDefaultSize
```

will be converted to:

```
const wxSize *size = (argCount > 3 ?
    (wxSize *)popPointer( _wxSize) :
     &wxDefaultSize);
```

const

The `const` declaration will cause the parameter to be declared as a `const`. For example:

```
const wxObject& clone
```

will be converted to:

```
const wxObject *clone = (wxObject *)popPointer( _wxObject);
```

The exception to the rules are arrays and `wxString`. They both ignore the `const` declaration. For example:

```
const wxString& colourName
```

will be converted to:

```
wxString colourName = popString();
```

wxWindows DataTypes

Any datatype that starts with "wx" is assumed to be a wxWindows datatype. Non-native datatypes will not check the pointer type returned by popPointer. For example:

```
double *y
```

will be converted to:

```
double *y = (double *)popPointer( 0 );
```

Whereas a wxWindows datatype will check the type. For example:

```
wxWindow* window
```

will be converted to:

```
wxWindow *window = (wxWindow *)popPointer( _wxWindow);
```

Dereferencing

Datatypes preceeded by the dereference operator & will have it removed, and take the general form:

```
(datatype *)popPointer( _datatype )
```

For example:

```
const wxObject& clone
```

will be converted to:

```
const wxObject *clone = (wxObject *)popPointer( _wxObject);
```

Non-native datatypes will not have their type checked by popPointer(). The main reason for having to declare structs is to prevent popPointer() from flagging these as errors.

Pointers

Datatypes *preceeded* by the pointer operator(s) * will have them removed and take the general form:

```
(datatype *)popPointer( _datatype )
```

The code can only handle two levels of indirection. For example:

```
wxWindow* window
```

will be converted to:

```
wxWindow *window = (wxWindow *)popPointer( _wxWindow);
```

The exception is `*wxString`, which takes the form:

```
wxString name = popString();
```

For example:

```
const wxString& colourName
```

will be converted to:

```
wxString colourName = popString();
```

Arrays

Arrays are not currently supported. They will basically create a dummy placeholder of the form:

```
popNumber();
type name[1];
```

For example:

```
pop wxPoint points[]
```

will be converted to:

```
popNumber();
wxPoint points[1];
```

There is a procedure in `class.c` called `popStringList()` that was designed to convert delimited strings into wxString arrays, but the was never fully implemented.

Calling The Method

The actual call to the method is fairly simple, once the proper casts have been set up. Parameters are pushed onto the stack in reverse order (the last argument is on the top of the stack). The variable `me` holds the pointer to the object, so the call is basically:

```
returns = ((class*)me)-> method();
```

If the return value is a pointer, it will be wrapped with a call to `addObject()`:

```
addObject( class_index, (int)returns )
```

For example:

```
wxBrush& GetBackground()
```

will be converted to:

```
void wxDC_GetBackground()
{
    wxBrush *returns;
    returns = &((wxDC *)me)->GetBackground();
    pushNumber( addObject( _wxBrush, (int)returns ) );
}
```

wxWindows Methods

This is an autogenerated list of classes and methods that are implemented with wxBasic. For details on the behavior of these methods, please refer to the wxWindows documentation.

Methods followed by a comment *alternate constructor* indicate an alternate constructor for the class. For example, an alternate constructor for the `wxBitmap` class is `wxEmptyBitmap`. It can be called as if it were a class name:

```
myBitmap = new wxEmptyBitmap(10, 10, 2)
```

Because wxBasic is loose with it's typecasting (converting numbers and strings to the propert types on demand), it can't determine which constructor to select if more than one option is available. Alternate constructors solve this problem.

Also, some classes, like `wxBrush.GetColour`, have aliases (`wxBrush.GetColor`). In these instances, the documentation will reference the alias following the main function.

Finally, the wrappers do not currently support arrays.

With those exceptions, the wxWindows methods behave as documented in the wxWindows documentation.

builtin (no class)

wxBeginBusyCursor(cursor)
wxBell()
wxCreateDynamicObject(className)
wxDisplaySize(width, height)
wxEnableTopLevelWindows(enable)
wxEndBusyCursor()
wxError(msg, title)
wxExecute(command, sync, callback)
wxExit()
wxFatalError(msg, title)
wxFindMenuItemId(frame, menuString, itemString)
wxFindWindowAtPoint(pt)
wxFindWindowAtPointer(pt)
wxFindWindowByLabel(label, parent)
wxFindWindowByName(name, parent)
wxGetHomeDir()
wxGetMousePosition()
wxGetOsDescription()
wxGetOsVersion(major, minor)

wxHandleFatalExceptions(doIt)
wxIsBusy()
wxMessageBox(message, caption, style, parent, x, y)
wxNewId()
wxNow()
wxPostEvent(dest, event)
wxRegisterId(id)
wxSafeYield(win)
wxShell(command)
wxSleep(secs)
wxTrap()
wxUsleep(milliseconds)
wxWakeUpIdle()
wxYield()

wxActivateEvent

GetActive()
wxActivateEvent(eventType, active, id)

wxApp

CreateLogTarget()
Dispatch()
ExitMainLoop()
GetAppName()
GetClassName()
GetStdIcon(which)
GetTopWindow()
GetUseBestVisual()
GetVendorName()
Initialized()
MainLoop()
Pending()
SendIdleEvents()
SetAppName(name)
SetClassName(name)
SetExitOnFrameDelete(flag)
SetTopWindow(window)
SetUseBestVisual(flag)
SetVendorName(name)

wxBitmap

Create(width, height, depth)
GetDepth()

GetHeight()
GetMask()
GetPalette()
GetSubBitmap())
GetWidth()
LoadFile(name, type)
Ok()
SaveFile(name, type, palette)
SetDepth(depth)
SetHeight(height)
SetMask(mask)
SetWidth(width)
wxBitmapCopy(bitmap) (alternate constructor)
wxEmptyBitmap(width, height, depth) (alternate constructor)
~wxBitmap()

wxBitmapButton

Create(parent, id, bitmap, pos, size, style, validator, name)
GetBitmapDisabled()
GetBitmapFocus()
GetBitmapLabel()
GetBitmapSelected()
SetBitmapDisabled(bitmap)
SetBitmapFocus(bitmap)
SetBitmapLabel(bitmap)
SetBitmapSelected(bitmap)
wxBitmapButton(parent, id, bitmap, pos, size, style, validator, name)
~wxBitmapButton()

wxBoxSizer

CalcMin()
GetOrientation()
RecalcSizes()
wxBoxSizer(orient)

wxBrush

Copy(brush) (alternate constructor)
GetColour() (alias GetColor)
GetStipple()
GetStyle()
Ok()
SetColour(colour) (alias SetColor)
SetColour(colourName)

SetColour(red, green, blue)
SetStipple(bitmap)
SetStyle(style)
wxBrush(colour, style)
wxBrushFromBitmap(stippleBitmap) (alternate constructor)
wxBrushFromColorName(colourName, style) (alternate constructor)
wxBrushFromColourName(colourName, style) (alternate constructor)
~wxBrush()

wxBrushList

FindOrCreateBrush(colour, style)
wxBrushList()

wxButton

Create(parent, id, label, pos, size, style, validator, name)
GetDefaultSize()
GetLabel()
SetDefault()
SetLabel(label)
wxButton(parent, id, label, pos, size, style, validator, name)
~wxButton()

wxCalculateLayoutEvent

GetFlags()
GetRect()
SetFlags(flags)
SetRect(rect)
wxCalculateLayoutEvent(id)

wxCalendarCtrl

Create(parent, id, date, pos, size, style, name)
EnableHolidayDisplay(display)
EnableMonthChange(enable)
EnableYearChange(enable)
GetAttr(day)
GetHeaderColourBg() (alias GetHeaderColorBg)
GetHeaderColourFg() (alias GetHeaderColorFg)
GetHighlightColourBg() (alias GetHighlightColorBg)
GetHighlightColourFg() (alias GetHighlightColorFg)
GetHolidayColourBg() (alias GetHolidayColorBg)
GetHolidayColourFg() (alias GetHolidayColorFg)
ResetAttr(day)
SetAttr(day, attr)

SetDate(date)
SetHeaderColours(colFg, colBg) (alias SetHeaderColors)
SetHighlightColours(colFg, colBg) (alias SetHighlightColors)
SetHoliday(day)
SetHolidayColours(colFg, colBg) (alias SetHolidayColors)
wxCalendarCtrl(parent, id, date, pos, size, style, name)
~wxCalendarCtrl()

wxCalendarEvent

GetDate()
GetWeekDay()
wxCalendarEvent(cal, type)

wxCheckBox

Create(parent, id, label, pos, size, style, val, name)
GetValue()
SetValue(state)
wxCheckBox(parent, id, label, pos, size, style, val, name)
~wxCheckBox()

wxCheckListBox

Check(item, check)
IsChecked(item)
wxCheckListBox(parent, id, pos, size, n, choices[], style, validator, name)
~wxCheckListBox()

wxChoice

Append(item)
Clear()
Create(parent, id, pos, size, n, choices[], style, validator, name)
FindString(string)
GetClientData(n)
GetColumns()
GetSelection()
GetString(n)
GetStringSelection()
Number()
SetClientData(n, data)
SetColumns(n)
SetSelection(n)
SetStringSelection(string)
wxChoice(parent, id, pos, size, n, choices[], style, validator, name)
~wxChoice()

wxClientDC

wxClientDC(window)

wxCloseEvent

CanVeto()
GetLoggingOff()
SetCanVeto(canVeto)
SetLoggingOff(loggingOff)
Veto(veto)
wxCloseEvent(commandEventType, id)

wxColourDialog

Create(parent, data)
GetColourData() (alias GetColorData)
ShowModal()
wxColourDialog(parent, data) (alias wxColorDialog)
~wxColourDialog()

wxComboBox

Append(item)
Clear()
Copy()
Create(parent, id, value, pos, size, n, choices[], style, validator, name)
Cut()
Delete(n)
FindString(string)
GetClientData(n)
GetInsertionPoint()
GetLastPosition()
GetSelection()
GetString(n)
GetStringSelection()
GetValue()
Number()
Paste()
Remove(from, to)
Replace(from, to, text)
SetClientData(n, data)
SetInsertionPoint(pos)
SetInsertionPointEnd()
SetMark(from, to)
SetSelection(n)
SetValue(text)

wxComboBox(parent, id, value, pos, size, n, choices[], style, validator, name)
~wxComboBox()

wxCommandEvent

GetClientData()
GetExtraLong()
GetInt()
GetSelection()
GetString()
IsChecked()
IsSelection()
SetClientData(clientData)
SetExtraLong(extraLong)
SetInt(intCommand)
SetString(string)
wxCommandEvent(commandEventType, id)

wxControl

Command(event)
GetLabel()
SetLabel(label)

wxDC

BeginDrawing()
Blit(xdest, ydest, width, height, source, xsrc, ysrc, logicalFunc, useMask)
CalcBoundingBox(x, y)
Clear()
CrossHair(x, y)
DestroyClippingRegion()
DeviceToLogicalX(x)
DeviceToLogicalXRel(x)
DeviceToLogicalY(y)
DeviceToLogicalYRel(y)
DrawArc(x1, y1, x2, y2, xc, yc)
DrawBitmap(bitmap, x, y, transparent)
DrawCheckMark(x, y, width, height)
DrawCheckMarkRect(rect)
DrawEllipse(x, y, width, height)
DrawEllipticArc(x, y, width, height, start, end)
DrawIcon(icon, x, y)
DrawLine(x1, y1, x2, y2)
DrawLines(n, points[], xoffset, yoffset)

DrawLinesList(points, xoffset, yoffset)
DrawPoint(x, y)
DrawPolygon(n, points[], xoffset, yoffset, fill_style)
DrawPolygonList(points, xoffset, yoffset, fill_style)
DrawRectangle(x, y, width, height)
DrawRotatedText(text, x, y, angle)
DrawRoundedRectangle(x, y, width, height, radius)
DrawText(text, x, y)
EndDoc()
EndDrawing()
EndPage()
FloodFill(x, y, colour, style)
GetBackground()
GetBackgroundMode()
GetBrush()
GetCharHeight()
GetCharWidth()
GetClippingBox(x, y, width, height)
GetFont()
GetLogicalFunction()
GetMapMode()
GetOptimization()
GetPen()
GetPixel(x, y, colour)
GetSize(width, height)
GetTextBackground()
GetTextExtent(string, w, h, descent, externalLeading, font)
GetTextForeground()
GetUserScale(x, y)
LogicalToDeviceX(x)
LogicalToDeviceXRel(x)
LogicalToDeviceY(y)
LogicalToDeviceYRel(y)
MaxX()
MaxY()
MinX()
MinY()
Ok()
ResetBoundingBox()
SetBackground(brush)
SetBackgroundMode(mode)
SetBrush(brush)
SetClippingRegion(region)
SetClippingRegionXY(x, y, width, height)
SetDeviceOrigin(x, y)

SetFont(font)
SetOptimization(optimize)
SetPalette(palette)
SetPen(pen)
SetTextBackground(colour)
SetTextForeground(colour)
SetUserScale(xScale, yScale)
StartDoc(message)
StartPage()
wxDC_GetSizeX()
wxDC_GetSizeY()
~wxDC()

wxDialog

Centre(direction)
Create(parent, id, title, pos, size, style, name)
EndModal(retCode)
GetReturnCode()
GetTitle()
Iconize(iconize)
IsIconized()
IsModal()
SetModal(flag)
SetReturnCode(retCode)
SetTitle(title)
Show(show)
ShowModal()
wxDialog(parent, id, title, pos, size, style, name)
~wxDialog()

wxDialUpEvent

IsConnectedEvent()
IsOwnEvent()
wxDialUpEvent(isConnected, isOwnEvent)

wxDirDialog

GetMessage()
GetPath()
GetStyle()
SetMessage(message)
SetPath(path)
SetStyle(style)
ShowModal()

wxDirDialog(parent, message, defaultPath, style, pos)
~wxDirDialog()

wxDropFilesEvent

GetFiles()
GetNumberOfFiles()
GetPosition()

wxEraseEvent

GetDC()
wxEraseEvent(id, dc)

wxEvent

GetEventObject()
GetEventType()
GetId()
GetSkipped()
GetTimestamp()
SetEventObject(object)
SetEventType(typ)
SetId(id)
SetTimestamp(timeStamp)
Skip(skip)
wxEvent(id)

wxEvtHandler

wxEvtHandler()
~wxEvtHandler()

wxFileDialog

GetDirectory()
GetFilename()
GetFilterIndex()
GetMessage()
GetPath()
GetStyle()
GetWildcard()
SetDirectory(directory)
SetFilename(setfilename)
SetMessage(message)
SetPath(path)
SetStyle(style)

SetWildcard(wildCard)
ShowModal()
wxFileDialog(parent, message, defaultDir, defaultFile, wildcard, style, pos)
~wxFileDialog()

wxFlexGridSizer

wxFlexGridSizer(rows, cols, vgap, hgap)

wxFocusEvent

wxFocusEvent(eventType, id)

wxFont

GetDefaultEncoding()
GetFaceName()
GetFamily()
GetPointSize()
GetStyle()
GetUnderlined()
GetWeight()
SetDefaultEncoding(encoding)
SetFaceName(faceName)
SetFamily(family)
SetPointSize(pointSize)
SetStyle(style)
SetUnderlined(underlined)
SetWeight(weight)
wxFont(pointSize, family, style, weight, underline, faceName, encoding)
~wxFont()

wxFontDialog

GetFontData()
ShowModal()
wxFontDialog(parent, data)
~wxFontDialog()

wxFrame

Centre(direction)
Command(id)
Create(parent, id, title, pos, size, style, name)
CreateStatusBar(number, style, id, name)
CreateToolBar(style, id, name)
GetClientAreaOrigin()

GetMenuBar()
GetStatusBar()
GetTitle()
GetToolBar()
Iconize(iconize)
IsIconized()
IsMaximized()
Maximize(maximize)
SetIcon(icon)
SetMenuBar(menuBar)
SetStatusBar(statusBar)
SetStatusText(text, number)
SetStatusWidths(n, widths)
SetTitle(title)
SetToolBar(toolBar)
wxFrame(parent, id, title, pos, size, style, name)
~wxFrame()

wxGauge

Create(parent, id, range, pos, size, style, validator, name)
GetBezelFace()
GetRange()
GetShadowWidth()
GetValue()
SetBezelFace(width)
SetRange(range)
SetShadowWidth(width)
SetValue(pos)
wxGauge(parent, id, range, pos, size, style, validator, name)
~wxGauge()

wxGDIObject

wxGDIObject()

wxGrid

AppendCols(numCols, updateLabels)
AppendRows(numRows, updateLabels)
AutoSize()
AutoSizeColumn(col, setAsMin)
AutoSizeColumns(setAsMin)
AutoSizeRow(row, setAsMin)
AutoSizeRows(setAsMin)
BeginBatch()

BlockToDeviceRect(topLeft, bottomRight)
CalcCellsExposed(reg)
CalcColLabelsExposed(reg)
CalcRowLabelsExposed(reg)
CanDragColSize()
CanDragGridSize()
CanDragRowSize()
CanEnableCellControl()
CellToRect(coords)
CellToRect(row, col)
ClearGrid()
ClearSelection()
CreateGrid(numRows, numCols, selmode)
DeleteCols(pos, numCols, updateLabels)
DeleteRows(pos, numRows, updateLabels)
DisableCellEditControl()
DisableDragColSize()
DisableDragGridSize()
DisableDragRowSize()
DrawTextRectangle(dc, wxString, wxRect, hAlign, vAlign)
EnableCellEditControl(enable)
EnableDragColSize(enable)
EnableDragGridSize(enable)
EnableDragRowSize(enable)
EnableEditing(edit)
EnableGridLines(enable)
EndBatch()
GetBatchCount()
GetCellAlignment(row, col, horiz, vert)
GetCellBackgroundColour(row, col)
GetCellFont(row, col)
GetCellHighlightColour()
GetCellTextColour(row, col)
GetCellTextFont()
GetCellValue(coords)
GetCellValue(row, col)
GetColLabelAlignment(horiz, vert)
GetColLabelSize()
GetColLabelValue(col)
GetCols()
GetColSize(col)
GetColumnWidth(col)
GetCursorColumn()
GetCursorRow()
GetDefaultCellAlignment(horiz, vert)

GetDefaultCellBackgroundColour()
GetDefaultCellFont()
GetDefaultCellTextColour()
GetDefaultColLabelSize()
GetDefaultColSize()
GetDefaultRowLabelSize()
GetDefaultRowSize()
GetDividerPen()
GetEditable()
GetEditInPlace()
GetGridCursorCol()
GetGridCursorRow()
GetGridLineColour()
GetLabelBackgroundColour()
GetLabelFont()
GetLabelSize(orientation)
GetLabelTextColour()
GetLabelValue(orientation, pos)
GetNumberCols()
GetNumberRows()
GetRowLabelAlignment(horiz, vert)
GetRowLabelSize()
GetRowLabelValue(row)
GetRows()
GetRowSize(row)
GetScrollPosX()
GetScrollPosY()
GetSelectionBackground()
GetSelectionForeground()
GetTable()
GetTextBoxSize(dc, lines, width, height)
GetViewHeight()
GetViewWidth()
GridLinesEnabled()
HideCellEditControl()
InsertCols(pos, numCols, updateLabels)
InsertRows(pos, numRows, updateLabels)
IsCellEditControlEnabled()
IsCellEditControlShown()
IsCurrentCellReadOnly()
IsEditable()
IsInSelection(coords)
IsInSelection(row, col)
IsReadOnly(row, col)
IsSelection()

IsVisible(coords, wholeCellVisible)
IsVisible(row, col, wholeCellVisible)
MakeCellVisible(coords)
MakeCellVisible(row, col)
MoveCursorDown(expandSelection)
MoveCursorDownBlock(expandSelection)
MoveCursorLeft(expandSelection)
MoveCursorLeftBlock(expandSelection)
MoveCursorRight(expandSelection)
MoveCursorRightBlock(expandSelection)
MoveCursorUp(expandSelection)
MoveCursorUpBlock(expandSelection)
MovePageDown()
MovePageUp()
SaveEditControlValue()
SelectAll()
SelectBlock(topLeft, bottomRight)
SelectBlock(topLeft, bottomRight, addToSelected)
SelectBlock(topRow, leftCol, bottomRow, rightCol)
SelectBlock(topRow, leftCol, bottomRow, rightCol, addToSelected)
SelectCol(col, addToSelected)
SelectRow(row, addToSelected)
SetCellAlignment(align, row, col)
SetCellAlignment(row, col, horiz, vert)
SetCellBackgroundColour(col)
SetCellBackgroundColour(colour, row, col)
SetCellBackgroundColour(row, col,))
SetCellFont(row, col,))
SetCellHighlightColour())
SetCellTextColour(col)
SetCellTextColour(row, col,))
SetCellTextColour(val, row, col)
SetCellTextFont(fnt)
SetCellTextFont(fnt, row, col)
SetCellValue(coords, s)
SetCellValue(row, col, s)
SetCellValue(val, row, col)
SetColAttr(col, attr)
SetColFormatBool(col)
SetColFormatCustom(col, typeName)
SetColFormatFloat(col, width, precision)
SetColFormatNumber(col)
SetColLabelAlignment(horiz, vert)
SetColLabelSize(height)
SetColLabelValue(col,))

SetColMinimalWidth(col, width)
SetColSize(col, width)
SetColumnWidth(col, width)
SetDefaultCellAlignment(horiz, vert)
SetDefaultCellBackgroundColour())
SetDefaultCellFont())
SetDefaultCellTextColour())
SetDefaultColSize(width, resizeExistingCols)
SetDefaultRowSize(height, resizeExistingRows)
SetEditable(edit)
SetGridCursor(row, col)
SetGridLineColour())
SetLabelAlignment(orientation, align)
SetLabelBackgroundColour())
SetLabelFont())
SetLabelSize(orientation, sz)
SetLabelTextColour())
SetLabelValue(orientation, val, pos)
SetMargins(extraWidth, extraHeight)
SetReadOnly(row, col, isReadOnly)
SetRowAttr(row, attr)
SetRowHeight(row, height)
SetRowLabelAlignment(horiz, vert)
SetRowLabelSize(width)
SetRowLabelValue(row,))
SetRowMinimalHeight(row, width)
SetRowSize(row, height)
SetSelectionBackground(c)
SetSelectionForeground(c)
SetSelectionMode(selmode)
SetTable(table, takeOwnership, selmode)
ShowCellEditControl()
StringToLines(value, lines)
UpdateDimensions()
wxGrid(parent, id, pos, size, style, name)
XToCol(x)
XToEdgeOfCol(x)
XYToCell(x, y,))
YToEdgeOfRow(y)
YToRow(y)
~wxGrid()

wxGridSizer

wxGridSizer(cols, rows, vgap, hgap)

wxIdleEvent

MoreRequested()
RequestMore(needMore)
wxIdleEvent()

wxImage

AddHandler(handler)
CleanUpHandlers()
ConvertToBitmap()
Copy()
GetBlue(x, y)
GetData()
GetGreen(x, y)
GetHandlers()
GetHeight()
GetMaskBlue()
GetMaskGreen()
GetMaskRed()
GetPalette()
GetRed(x, y)
GetSubImage(rect)
GetWidth()
HasMask()
InitStandardHandlers()
InsertHandler(handler)
LoadFile(name, type)
LoadMimeFile(name, mimetype)
Mirror(horizontally)
Ok()
RemoveHandler(name)
Replace(r1, g1, b1, r2, g2, b2)
Rotate(angle, rotationCentre, interpolating, offsetAfterRotation)
Rotate90(clockwise)
SaveFile(name, type)
SaveMimeFile(name, mimetype)
Scale(width, height)
SetData())
SetMask(hasMask)
SetMaskColour(red, blue, green)
SetPalette(palette)
SetRGB(x, y, red, green, blue)
wxEmptyImage(width, height) (alternate constructor)
wxImage(image)
wxImageFromBitmap(bitmap) (alternate constructor)

wxImageFromData(width, height, data, static_data) (alternate constructor)
wxImageFromFile(name, type) (alternate constructor)
wxNullImage() (alternate constructor)
~wxImage()

wxIndividualLayoutConstraint

Above(otherWin, margin)
Absolute(value)
AsIs()
Below(otherWin, margin)
LeftOf(otherWin, margin)
PercentOf(otherWin, edge, per)
RightOf(otherWin, margin)
SameAs(otherWin, edge, margin)
Set(rel, otherWin, otherEdge, value, margin)
Unrained()
wxIndividualLayoutConstraint()

wxInitDialogEvent

wxInitDialogEvent(id)

wxJoystickEvent

ButtonDown(button)
ButtonIsDown(button)
ButtonUp(button)
GetButtonChange()
GetButtonState()
GetJoystick()
GetPosition()
GetZPosition()
IsButton()
IsMove()
IsZMove()
wxJoystickEvent(eventType, state, joystick, change)

wxKeyEvent

AltDown()
ControlDown()
GetKeyCode()
GetPosition()
GetX()
GetY()
HasModifiers()

MetaDown()
ShiftDown()
wxKeyEvent(keyEventType)

wxLayoutConstraints

wxLayoutConstraints()

wxListBox

Append(item)
Clear()
Create(parent, id, pos, size, n, choices[], style, validator, name)
Delete(n)
Deselect(n)
FindString(string)
GetClientData(n)
GetSelection()
GetSelections(selections)
GetString(n)
GetStringSelection()
Number()
Selected(n)
SetClientData(n, data)
SetFirstItem(n)
SetSelection(n, select)
SetString(n, string)
SetStringSelection(string, select)
wxListBox(parent, id, pos, size, n, choices[], style, validator, name)
~wxListBox()

wxListCtrl

Arrange(flag)
ClearAll()
Create(parent, id, pos, size, style, validator, name)
DeleteAllItems()
DeleteColumn(col)
DeleteItem(item)
EditLabel(item)
EnsureVisible(item)
FindItem(start, str, partial)
FindItemAtPos(start, pt, direction)
FindItemData(start, data)
GetColumn(col, item)
GetColumnWidth(col)

GetCountPerPage()
GetImageList(which)
GetItem(info)
GetItemCount()
GetItemData(item)
GetItemPosition(item, pos)
GetItemRect(item, rect, code)
GetItemSpacing(isSmall)
GetItemState(item, stateMask)
GetItemText(item)
GetNextItem(item, geometry, state)
GetSelectedItemCount()
GetTextColour() (alias GetTextColor)
GetTopItem()
HitTest(point, flags)
InsertColumn(col, info)
InsertColumnInfo(col, heading, format, width)
InsertImageItem(index, imageIndex)
InsertImageStringItem(index, label, imageIndex)
InsertItem(info)
InsertStringItem(index, label)
ScrollList(dx, dy)
SetBackgroundColour(col) (alias SetBackgroundColor)
SetColumn(col, item)
SetColumnWidth(col, width)
SetImageList(imageList, which)
SetItem(info)
SetItemData(item, data)
SetItemImage(item, image, selImage)
SetItemPosition(item, pos)
SetItemState(item, state, stateMask)
SetItemText(item, text)
SetSingleStyle(style, add)
SetStringItem(index, col, label, imageId)
SetTextColour(col) (alias SetTextColor)
SetWindowStyleFlag(style)
wxListCtrl(parent, id, pos, size, style, validator, name)
~wxListCtrl()

wxListEvent

Cancelled()
GetCode()
GetColumn()
GetData()

GetImage()
GetIndex()
GetItem()
GetLabel()
GetMask()
GetOldIndex()
GetPoint()
GetText()
wxListEvent(commandType, id)

wxMDIChildFrame

Activate()
Create(parent, id, title, pos, size, style, name)
Restore()
wxMDIChildFrame(parent, id, title, pos, size, style, name)
~wxMDIChildFrame()

wxMDIParentFrame

ActivateNext()
ActivatePrevious()
ArrangeIcons()
Cascade()
Create(parent, id, title, pos, size, style, name)
GetActiveChild()
GetClientSize(width, height)
GetClientWindow()
GetToolBar()
SetToolBar(toolbar)
Tile()
wxMDIParentFrame(parent, id, title, pos, size, style, name)
~wxMDIParentFrame()

wxMemoryDC

SelectObject(bitmap)
wxMemoryDC()

wxMenu

Append(id, item, helpString, checkable)
AppendItem(menuItem)
AppendMenu(id, item, subMenu, helpString)
AppendSeparator()
Break()
Check(id, check)

Delete(id)
DeleteItem(item)
Destroy(id)
DestroyMenuItem(item)
Enable(id, enable)
FindItem(itemString)
GetHelpString(id)
GetLabel(id)
GetMenuItemCount()
GetMenuItems()
GetTitle()
Insert(pos, item)
IsChecked(id)
IsEnabled(id)
Remove(item)
RemoveById(id)
SetHelpString(id, helpString)
SetLabel(id, label)
SetTitle(title)
UpdateUI(source)
wxMenu(title, style)
~wxMenu()

wxMenuBar

Append(menu, title)
Check(id, check)
Enable(id, enable)
EnableTop(pos, enable)
FindMenu(title)
FindMenuItem(menuString, itemString)
GetHelpString(id)
GetLabel(id)
GetLabelTop(pos)
GetMenu(menuIndex)
GetMenuCount()
Insert(pos, menu, title)
IsChecked(id)
IsEnabled(id)
Refresh()
Remove(pos)
Replace(pos, menu, title)
SetHelpString(id, helpString)
SetLabel(id, label)
SetLabelTop(pos, label)

wxMenuBar(style)
~wxMenuBar()

wxMenuEvent

GetMenuId()
wxMenuEvent(type, id)

wxMenuItem

Check(check)
Enable(enable)
GetHelp()
GetId()
GetLabel()
GetLabelFromText(text)
GetSubMenu()
GetText()
IsCheckable()
IsChecked()
IsEnabled()
IsSeparator()
SetHelp(helpString)
wxMenuItem(parentMenu, id, text, help, isCheckable, subMenu)
~wxMenuItem()

wxMessageDialog

ShowModal()
wxMessageDialog(parent, message, caption, style, pos)
~wxMessageDialog()

wxMetafileDC

wxMetafileDC(filename)
~wxMetafileDC()

wxMiniFrame

Create(parent, id, title, pos, size, style, name)
wxMiniFrame(parent, id, title, pos, size, style, name)
~wxMiniFrame()

wxMouseEvent

AltDown()
Button(button)
ButtonDClick(but)

ButtonDown(but)
ButtonUp(but)
ControlDown()
Dragging()
Entering()
GetLogicalPosition(dc)
GetPosition()
GetX()
GetY()
IsButton()
Leaving()
LeftDClick()
LeftDown()
LeftIsDown()
LeftUp()
MetaDown()
MiddleDClick()
MiddleDown()
MiddleIsDown()
MiddleUp()
Moving()
RightDClick()
RightDown()
RightIsDown()
RightUp()
ShiftDown()

wxMoveEvent

GetPosition()
wxMoveEvent(pt, id)

wxNotebook

AddPage(page, text, select, imageId)
AdvanceSelection(forward)
Create(parent, id, pos, size, style, name)
DeleteAllPages()
DeletePage(page)
GetImageList()
GetPage(page)
GetPageCount()
GetPageImage(nPage)
GetPageText(nPage)
GetRowCount()
GetSelection()

InsertPage(index, page, text, select, imageId)
RemovePage(page)
SetImageList(imageList)
SetPadding(padding)
SetPageImage(page, image)
SetPageSize(size)
SetPageText(page, text)
SetSelection(page)
wxNotebook(parent, id, pos, size, style, name)
~wxNotebook()

wxNotebookEvent

GetOldSelection()
GetSelection()
SetOldSelection(page)
SetSelection(page)
wxNotebookEvent(eventType, id, sel, oldSel)

wxNotebookSizer

GetNotebook()
wxNotebookSizer(notebook)

wxNotifyEvent

IsAllowed()
Veto()
wxNotifyEvent(eventType, id)

wxObject

GetClassInfo()
GetRefData()
IsKindOf(info)
Ref(clone)
SetRefData(data)
UnRef()
wxObject()
~wxObject()

wxPageSetupDialog

GetPageSetupData()
ShowModal()
wxPageSetupDialog(parent, data)
~wxPageSetupDialog()

wxPaintDC

wxPaintDC(window)

wxPaintEvent

wxPaintEvent(id)

wxPalette

Create(n, red, green, blue)
Ok()
~wxPalette()

wxPanel

Create(parent, id, pos, size, style, name)
GetDefaultItem()
InitDialog()
SetDefaultItem(btn)
wxPanel(parent, id, pos, size, style, name)
~wxPanel()

wxPen

GetCap()
GetColour()
GetJoin()
GetStyle()
GetWidth()
Ok()
SetCap(capStyle)
SetColour(colour)
SetColour(colourName)
SetColour(red, green, blue)
SetJoin(join_style)
SetStyle(style)
SetWidth(width)
wxPen()
wxPenCopy(pen) (alternate constructor)
wxPenFromColor(colour, width, style) (alternate constructor)
wxPenFromColorName(colourName, width, style) (alternate constructor)
wxPenFromColour(colour, width, style) (alternate constructor)
wxPenFromColourName(colourName, width, style) (alternate constructor)
~wxPen()

wxPlotWindow

Add(curve)
Delete(curve)
Enlarge(curve, factor)
GetAt(n)
GetCount()
GetCurrent()
GetUnitsPerValue()
GetZoom()
Move(curve, pixels_up)
RedrawEverything()
RedrawXAxis()
RedrawYAxis()
SetCurrent(current)
SetEnlargeAroundWindowCentre(aroundwindow)
SetScrollOnThumbRelease(onrelease)
SetUnitsPerValue(upv)
SetZoom(zoom)
wxPlotWindow(parent, id, pos, size, flags)
~wxPlotWindow()

wxPoint

wxPoint(x, y)
~wxPoint()

wxPostScriptDC

GetResolution()
SetResolution(ppi)
wxPostScriptDC(printData)
~wxPostScriptDC()

wxPrintDialog

GetPrintDC()
GetPrintDialogData()
ShowModal()
wxPrintDialog(parent, data)
~wxPrintDialog()

wxPrinterDC

wxPrinterDC(printData)

wxProcessEvent

GetPid()
wxProcessEvent(id, pid)

wxQueryLayoutInfoEvent

GetFlags()
GetOrientation()
GetRequestedLength()
GetSize()
SetAlignment(alignment)
SetFlags(flags)
SetOrientation(orientation)
SetRequestedLength(length)
SetSize(size)
wxQueryLayoutInfoEvent(id)

wxRadioBox

Create(parent, id, label, point, size, n, choices[], majorDimension, style, validator, name)
Enable(enable)
EnableItem(n, enable)
FindString(string)
GetItemLabel(n)
GetLabel(item)
GetSelection()
GetString(n)
GetStringSelection()
Number()
SetItemLabel(n, label)
SetSelection(n)
SetStringSelection(string)
Show(show)
ShowItem(item, show)
wxRadioBox(parent, id, label, point, size, n, choices[], majorDimension, style, validator, name)
~wxRadioBox()

wxRadioButton

Create(parent, id, label, pos, size, style, validator, name)
GetValue()
SetValue(value)
wxRadioButton(parent, id, label, pos, size, style, validator, name)
~wxRadioButton()

wxSashLayoutWindow

GetAlignment()
GetOrientation()
SetAlignment(alignment)
SetDefaultSize(size)
SetOrientation(orientation)
wxSashLayoutWindow(parent, id, pos, size, style, name)
~wxSashLayoutWindow()

wxSashWindow

GetMaximumSizeX()
GetMaximumSizeY()
GetMinimumSizeX()
GetMinimumSizeY()
GetSashVisible(edge)
HasBorder(edge)
SetMaximumSizeX(min)
SetMaximumSizeY(min)
SetMinimumSizeX(min)
SetMinimumSizeY(min)
SetSashBorder(edge, hasBorder)
SetSashVisible(edge, visible)
wxSashWindow(parent, id, pos, size, style, name)
~wxSashWindow()

wxScreenDC

EndDrawingOnTop()
StartDrawingOnTop(window)
StartDrawingOnTopRect(rect)
wxScreenDC()

wxScrollBar

Create(parent, id, pos, size, style, validator, name)
GetPageSize()
GetRange()
GetThumbPosition()
SetScrollbar(position, thumbSize, range, pageSize, refresh)
SetThumbPosition(viewStart)
wxScrollBar(parent, id, pos, size, style, validator, name)
~wxScrollBar()

wxScrolledWindow

CalcScrolledPosition(x, y, xx, yy)
CalcUnscrolledPosition(x, y, xx, yy)
Create(parent, id, pos, size, style, name)
EnableScrolling(xScrolling, yScrolling)
GetScrollPixelsPerUnit(xUnit, yUnit)
GetViewStart(x, y)
GetVirtualSize(x, y)
IsRetained()
PrepareDC(dc)
Scroll(x, y)
SetScrollbars(pixelsPerUnitX, pixelsPerUnitY, noUnitsX, noUnitsY, xPos, yPos, noRefresh)
SetTargetWindow(window)
wxScrolledWindow(parent, id, pos, size, style, name)
~wxScrolledWindow()

wxScrollEvent

GetOrientation()
GetPosition()
wxScrollEvent(commandType, id, pos, orientation)

wxScrollWinEvent

GetOrientation()
GetPosition()

wxSingleChoiceDialog

GetSelection()
GetSelectionClientData()
GetStringSelection()
SetSelection(selection)
ShowModal()
~wxSingleChoiceDialog()

wxSize

GetHeight()
GetWidth()
Set(width, height)
SetHeight(height)
SetWidth(width)
wxSize(width, height)
~wxSize()

wxSizeEvent

GetSize()
wxSizeEvent(sz, id)

wxSizer

Add(width, height, option, flag, border, userData)
AddSizer(sizer, option, flag, border, userData)
AddWindow(window, option, flag, border, userData)
CalcMin()
Fit(window)
GetMinSize()
GetPosition()
GetSize()
Layout()
Prepend(width, height, option, flag, border, userData)
PrependSizer(sizer, option, flag, border, userData)
PrependWindow(window, option, flag, border, userData)
RecalcSizes()
Remove(nth)
RemoveSizer(sizer)
RemoveWindow(window)
SetDimension(x, y, width, height)
SetItemMinSize(pos, width, height)
SetMinSize(width, height)
SetSizeHints(window)
SetSizerMinSize(sizer, width, height)
SetWindowMinSize(window, width, height)
~wxSizer()

wxSlider

ClearSel()
ClearTicks()
Create(parent, id, value, minValue, maxValue, point, size, style, validator, name)
GetLineSize()
GetMax()
GetMin()
GetPageSize()
GetSelEnd()
GetSelStart()
GetThumbLength()
GetTickFreq()
GetValue()

SetLineSize(lineSize)
SetPageSize(pageSize)
SetRange(minValue, maxValue)
SetSelection(startPos, endPos)
SetThumbLength(len)
SetTick(tickPos)
SetTickFreq(n, pos)
SetValue(value)
wxSlider(parent, id, value, minValue, maxValue, point, size, style, validator, name)
~wxSlider()

wxSocketEvent

GetClientData()
GetSocket()
GetSocketEvent()
wxSocketEvent(id)

wxSpinButton

Create(parent, id, pos, size, style, name)
GetMax()
GetMin()
GetValue()
SetRange(min, max)
SetValue(value)
wxSpinButton(parent, id, pos, size, style, name)
~wxSpinButton()

wxSpinCtrl

Create(parent, id, value, pos, size, style, min, max, initial, name)
GetMax()
GetMin()
GetValue()
SetRange(minVal, maxVal)
SetValue(text)
wxSpinCtrl(parent, id, value, pos, size, style, min, max, initial, name)

wxSpinEvent

GetPosition()
SetPosition(pos)
wxSpinEvent(commandType, id)

wxSplitterWindow

Create(parent, id, pos, size, style, name)
GetMinimumPaneSize()
GetSashPosition()
GetSplitMode()
GetWindow1()
GetWindow2()
Initialize(window)
IsSplit()
ReplaceWindow(winOld, winNew)
SetMinimumPaneSize(paneSize)
SetSashPosition(position, redraw)
SetSplitMode(mode)
SplitHorizontally(window1, window2, sashPosition)
SplitVertically(window1, window2, sashPosition)
Unsplit(toRemove)
wxSplitterWindow(parent, id, point, size, style, name)
~wxSplitterWindow()

wxStaticBitmap

SetBitmap(label)

wxStaticBox

Create(parent, id, label, pos, size, style, name)
wxStaticBox(parent, id, label, pos, size, style, name)
~wxStaticBox()

wxStaticBoxSizer

GetStaticBox()
wxStaticBoxSizer(box, orient)

wxStaticText

Create(parent, id, label, pos, size, style, name)
GetLabel()
SetLabel(label)
wxStaticText(parent, id, label, pos, size, style, name)

wxStatusBar

Create(parent, id, style, name)
GetFieldRect(i, rect)
GetFieldsCount()
GetStatusText(ir)

SetFieldsCount(number, widths)
SetMinHeight(height)
SetStatusText(text, i)
SetStatusWidths(n, widths)
wxStatusBar(parent, id, pos, size, style, name)
~wxStatusBar()

wxSysColourChangedEvent

wxSysColourChangedEvent()

wxTabCtrl

Create(parent, id, pos, size, style, name)
DeleteAllItems()
DeleteItem(item)
GetCurFocus()
GetImageList()
GetItemCount()
GetItemData(item)
GetItemImage(item)
GetItemRect(item, rect)
GetItemText(item)
GetRowCount()
GetSelection()
HitTest(pt, flags)
InsertItem(item, text, imageId, clientData)
SetImageList(imageList)
SetItemData(item, data)
SetItemImage(item, image)
SetItemSize(size)
SetItemText(item, text)
SetPadding(padding)
SetSelection(item)
wxTabCtrl(parent, id, pos, size, style, name)
~wxTabCtrl()

wxTabEvent

wxTabEvent(commandType, id)

wxTextCtrl

AppendText(text)
CanCopy()
CanCut()
CanPaste()

CanRedo()
CanUndo()
Clear()
Copy()
Create(parent, id, value, pos, size, style, validator, name)
Cut()
DiscardEdits()
GetInsertionPoint()
GetLastPosition()
GetLineLength(lineNo)
GetLineText(lineNo)
GetNumberOfLines()
GetSelection(from, to)
GetValue()
IsModified()
LoadFile(filename)
Paste()
PositionToXY(pos, x, y)
Redo()
Remove(from, to)
Replace(from, to, value)
SaveFile(filename)
SetEditable(editable)
SetInsertionPoint(pos)
SetInsertionPointEnd()
SetSelection(from, to)
SetValue(value)
ShowPosition(pos)
Undo()
WriteText(text)
wxTextCtrl(parent, id, value, pos, size, style, validator, name)
XYToPosition(x, y)
~wxTextCtrl()

wxTextEntryDialog

GetValue()
SetValue(value)
ShowModal()
wxTextEntryDialog(parent, message, caption, defaultValue, style, pos)
~wxTextEntryDialog()

wxTimer

IsOneShot()
IsRunning()

Notify()
SetOwner(owner, id)
Start(milliseconds, oneShot)
wxTimer(owner, id)
~wxTimer()

wxTimerEvent

GetInterval()

wxToolBar

AddControl(control)
AddSeparator()
DeleteTool(toolId)
DeleteToolByPos(pos)
EnableTool(toolId, enable)
GetToolBitmapSize()
GetToolClientData(toolId)
GetToolEnabled(toolId)
GetToolLongHelp(toolId)
GetToolPacking()
GetToolSeparation()
GetToolShortHelp(toolId)
GetToolSize()
GetToolState(toolId)
Realize()
SetToolBitmapSize(size)
SetToolLongHelp(toolId, helpString)
SetToolPacking(packing)
SetToolSeparation(separation)
SetToolShortHelp(toolId, helpString)
ToggleTool(toolId, toggle)
wxToolBar(parent, id, pos, size, style, name)
~wxToolBar()

wxTreeCtrl

AddRoot(text, image, selImage, data)
AppendItem(parent, text, image, selImage, data)
Collapse(item)
CollapseAndReset(item)
Create(parent, id, pos, size, style, validator, name)
Delete(item)
DeleteAllItems()
EditLabel(item)

EnsureVisible(item)
Expand(item)
GetChildrenCount(item, recursively)
GetCount()
GetFirstChild(item, cookie)
GetFirstVisibleItem()
GetImageList()
GetIndent()
GetItemData(item)
GetItemImage(item, which)
GetItemSelectedImage(item)
GetItemText(item)
GetLastChild(item)
GetNextChild(item, cookie)
GetNextSibling(item)
GetNextVisible(item)
GetParent(item)
GetPrevSibling(item)
GetPrevVisible(item)
GetRootItem()
GetSelection()
GetSelections(selection)
GetStateImageList()
HitTest(point, flags)
InsertItem(parent, previous, text, image, selImage, data)
InsertItemBefore(parent, before, text, image, selImage, data)
IsBold(item)
IsExpanded(item)
IsSelected(item)
IsVisible(item)
ItemHasChildren(item)
PrependItem(parent, text, image, selImage, data)
ScrollTo(item)
SetImageList(imageList)
SetIndent(indent)
SetItemBackgroundColour(item, col) (alias SetItemBackgroundColor)
SetItemBold(item, bold)
SetItemData(item, data)
SetItemFont(item, font)
SetItemHasChildren(item, hasChildren)
SetItemImage(item, image, which)
SetItemSelectedImage(item, selImage)
SetItemText(item, text)
SetItemTextColour(item, col) (alias SetItemTextColor)
SetStateImageList(imageList)

SortChildren(item)
Toggle(item)
Unselect()
UnselectAll()
wxTreeCtrl(parent, id, pos, size, style, validator, name)
~wxTreeCtrl()

wxTreeEvent

GetCode()
GetItem()
GetLabel()
GetOldItem()
GetPoint()
wxTreeEvent(commandType, id)

wxUpdateUIEvent

Check(check)
Enable(enable)
GetChecked()
GetEnabled()
GetSetChecked()
GetSetEnabled()
GetSetText()
GetText()
SetText(text)
wxUpdateUIEvent(commandId)

wxWindow

AddChild(child)
CaptureMouse()
Center(direction)
CenterOnParent(direction)
CenterOnScreen(direction)
Centre(direction)
CentreOnParent(direction)
CentreOnScreen(direction)
Clear()
ClientToScreen(x, y)
Close(force)
ConvertDialogPointToPixels(pt)
ConvertDialogSizeToPixels(sz)
Destroy()
DestroyChildren()

Enable(enable)
FindFocus()
FindWindow(id)
Fit()
GetBackgroundColour() (alias GetBackgroundColor)
GetBestSize()
GetCaret()
GetCharHeight()
GetCharWidth()
GetChildren()
GetClientSize()
GetConstraints()
GetDropTarget()
GetEventHandler()
GetExtraStyle()
GetFont()
GetForegroundColour() (alias GetForegroundColor)
GetGrandParent()
GetId()
GetLabel()
GetName()
GetParent()
GetPosition(x, y)
GetPositionTuple()
GetRect()
GetScrollPos(orientation)
GetScrollRange(orientation)
GetScrollThumb(orientation)
GetSize(width, height)
GetTextExtent(string, x, y, descent, externalLeading, font)
GetTitle()
GetToolTip()
GetUpdateRegion()
GetValidator()
GetWindowStyleFlag()
InitDialog()
IsEnabled()
IsExposed(x, y)
IsExposedPoint(pt)
IsExposedRect(rect)
IsRetained()
IsShown()
IsTopLevel()
Layout()
LoadFromResource(parent, resourceName, resourceTable)

Lower()
MakeModal(flag)
Move(x, y)
MoveXY(pt)
PopEventHandler(deleteHandler)
PopupMenu(menu, pos)
PopupMenuXY(menu, x, y)
PushEventHandler(handler)
Raise()
Refresh(eraseBackground, rect)
ReleaseMouse()
RemoveChild(child)
Reparent(newParent)
ScreenToClient(pt)
ScreenToClientXY(x, y)
ScrollWindow(dx, dy, rect)
SetAcceleratorTable(accel)
SetAutoLayout(autoLayout)
SetBackgroundColour(colour) (alias SetBackgroundColor)
SetCaret(caret)
SetClientSize(size)
SetClientSizeWH(width, height)
SetConstraints(raints)
SetCursor())
SetDimensions(x, y, width, height, sizeFlags)
SetDropTarget(target)
SetEventHandler(handler)
SetExtraStyle(exStyle)
SetFocus()
SetFont(font)
SetForegroundColour(colour) (alias GetForegroundColor)
SetId(id)
SetName(name)
SetPosition(size)
SetScrollbar(orientation, position, thumbSize, range, refresh)
SetScrollPos(orientation, pos, refresh)
SetSize(width, height)
SetSizeHints(minW, minH, maxW, maxH, incW, incH)
SetSizer(sizer)
SetTitle(title)
SetToolTip(tip)
SetValidator(validator)
SetWindowStyle(style)
SetWindowStyleFlag(style)
Show(show)

TransferDataFromWindow()
TransferDataToWindow()
Validate()
WarpPointer(x, y)
wxGetClientSizeTuple(width, height)
wxWindow(parent, id, pos, size, style, name)
~wxWindow()

wxWindowDC

wxWindowDC(window)

wxWizardEvent

GetDirection()
wxWizardEvent(type, id, direction)

To Do and Wish Lists

There are lots of things that still remain to be implemented, and a number of things that would be nice, but are hardly critical. The following is in no particular order:

Conversion Routines are in QBasic

The routines to create a number of support files, such as `wrap.cpp`, are written in Qbasic. This isn't especially terrible, since it works just fine. Butit should probably be coded in something else, like C++ or even wxBasic itself so that it's more portable.

I'm currently working on an version using `Awk`, but it's incomplete.

Namespace

Namespaces solve the problem of when function and variable names in included libraries conflict with those already in use.

Under a namespace scheme, you could assign a namespace to an include file:

```
Include "myLibrary" as mylib
```

and then refer to routines and variable in that file with the namespace prefix:

```
mylib.someFunction()
```

At the moment, there is no namespace implemented, but I'd like to add it.

Bytecodes

wxBasic doesn't generate very efficient code. I don't know if this is a problem or not. My first and foremost goal was *get the damned thing done*. There's no value in having a language fast and efficient if it's incomplete and broken.

It's also not clear how much of an issue this is. Computer processors are getting faster and faster, and most of the time, a GUI application is just sitting in an idle event loop.

Also, the code is still in a constant state of flux. At some point when it settles down, I'll think about generating bytecodes instead of nodes.

User-Defined Types and Classes

User-defined types are conspicious by their absence in wxBasic. At one point I had implemented them, but the addition of passing parameters by reference broke all that code.

Since wxBasic already supports C++ classes, it would *appear* to make sense to add user-defined classes as well. This may well be the case, but I'd like to get version 1.0 completed and released before adding this sort of complexity.

Interactive Debugging

It would be nice to be able to see what line is currently being executed and the values of the variables at that point. If you look at the Symbol data structure, you'll see that there is a.sibling field. This allows you to access things like the names of all the variables associated with a function.

It would also be nice to be able to modify the values of variables. All this would be dependant on tying a trace window, and possibly and IDE to wxBasic.

Don't hold your breath for this stuff. It's certainly feasible, but not in the immediate future.

An IDE

At a minimum, it would be nice to have some sort of editor for wxBasic. I'm thinking that I could hack something together from ScITE, along the lines of Python's IDLE. Since ScITE is being wrapped as a wxWindow control, I might even be able to write the IDE in wxBasic.