FutureBasic 5

Reference Manual

http://www.4toc.com/fb

Preface

FutureBasic (FB) is a high-level procedural programming language, in fact a whole "Integrated Development Environment" (IDE), for the Apple Macintosh® computer system. it is capable of creating standalone Universal Binary applications (Mach-O executables) which can run natively on either PPC or the latest Intel Macintosh computers.

It is a compiled BASIC dialect allowing easy access to the graphical user interface and file system of the MacOS. In providing structures such as nestable records and arrays, and various forms of subroutines (local functions) and callbacks, it is quite as powerful as C however with the syntax of BASIC and a less strict variable typing. The current version allows direct passthrough of both C and Objective-C source code. FutureBasic is a front-end to the acclaimed open-source gcc Unix compiler which ships with all new Macintoshes, and will allow translation to either C or Objective-C code for onward compiling behind the scenes into a finished MacOS application program!

FutureBasic features an editor, compiler, debugger, profiler, project manager, documentation, and code samples.

In January 2008, Staz Software released FutureBasic as freeware. Simultaneously, the FBtoC project was initiated to modernize FutureBasic and its components.

FBtoC was initially a standalone application and can still be used as such, but is now well integrated with the FutureBasic 5 editor.

FutureBasic version 5 and FBtoC are being actively developed and maintained by a team of volunteers whose members have collectively contributed thousands of man-hours in what is tantamount to a labor of love for their chosen programming language.

This FBtoC website (http://www.4toc.com/fb/) hosts both the FBtoC Project and the FutureBasic Freeware downloads. Downloads and their executables are freeware, but the source code and rights to distribute are reserved to the respective authors (the FBtoC team and Staz Software).

The FBtoC team welcomes feedback and may be contacted by subscribing to the FutureBasic mailing list (http://freegroups.net/groups/FutureBasic/) by posting a message with "FBtoC" in the subject line. The FutureBasic mailing list archives and subscription information are maintained at that location.

FBtoC Team

History

FutureBasic, known simply as "FB" to its advocates, began life in the mid-1980s as ZBasic, which was created by Andrew Gariepy and envisioned as a cross-platform development system. Before long, the cross-platform aspects were dropped in favor of focusing on Macintosh development.

ZBasic acquired a devoted following of developers who praised its ease of use and the tight, fast code produced by the compiler (a legendary labor involving extensive use of hand-built 68K assembly language code).

In 1992 and as the next major step after ZBasic version 5, Zedcor Inc., the company of the Gariepy brothers Andy, Mike, Peter and friends based in Tucson Arizona presented FutureBasic (later called FBI).

In 1995, Staz Software led by Chris Stasny, acquired the rights to market FutureBasic. Chris Stasny started this business with an upgraded version, namely FBII, and with his own development, a CASE tool namely the Program Generator (PG PRO) became available.

The transition from 68k to PowerPC CPUs was a lengthy process that involved a complete rewrite of the editor by Chris Stasny and an adaptation of the compiler by Andy Gariepy. This was undertaken during Apple's darkest days when the further existance of the Mac and Apple itself was in the news every week.

The result of their effort was a dramatically enhanced IDE called FB^3, and was released in September 1999. It featured, among many other things, a separate compiler application and various open hence modifiable runtimes, inline PPC assembly, simplified access to the Macintosh Toolbox API, as well as an expanded library of built-in functions.

Major update releases introduced a full-featured Appearance Compliant runtime written by Robert Purves and the Carbon compliance of generated applications. Once completely carbonized to run natively on MacOS X, the FutureBasic IDE was called FB4 and first released in July 2004.

Based in Diamondhead Mississippi, Staz Software was severely hit by Hurricane Katrina in September 2005 and development pace was slowed. This was at a time when major effort was required to keep the IDE up to date with Apple's evolution towards the Intel-based Macintosh.

More recently, an independent team of volunteer FutureBasic programmers developed a cross-compiler (FBtoC) that allows FutureBasic to generate applications as Universal Binaries through the use of the open source gcc compiler which is included with each copy of Apple's MacOS X system software.

On January 1, 2008, Staz Software announced that FutureBasic version 4 would henceforth be freeware and FBtoC 1.0 was made available at the http://www.4toc.com/fb website.

Conventions

In the syntax descriptions that appear in the remainder of this manual, the following conventions apply:

- Items in italics represent placeholders which should be replaced as indicated in the description;

- Items in bold text represent literal text that you should enter exactly as shown;

- Items in plain non-italic text represent literal text that you should usually enter exactly as shown; however, the following characters should not be entered, but have special meanings explained below:

   [   ]   {   } | . . .

- Items enclosed in square brackets [ ] are optional;

- Items separated by vertical bars | and enclosed by curly brackets { } represent a list from which one item should be chosen;

- Items separated by vertical bars | and enclosed by square brackets [ ] represent a list from which one or zero items should be chosen;

- An elipsis (. . .) indicates that the preceding item may be repeated an indefinite number of times.

Example: Consider the following syntax description template:

bob [, {bill | ron [, rick]}]

This template matches each of the following:

bob
bob, bill
bob, ron
bob, ron, rick

@fn **function**

Syntax:
functionAddress `=` **@fn** `FunctionName`

Description:
Returns a memory address which can be used to access the function specified by FunctionName. `FunctionName` must be the name of a function which was defined or prototyped at some earlier location in the source code, in a `local fn`, `def fn <expr>`, or `def fn <prototype>` statement.

The address returned by `@fn` can be used in a `def fn using` statement, or as a parameter to a Toolbox function that expects the address of a callback function.

See Also:
def fn using

#define statement

Syntax:
#define *NewTypeName* **as** *OldTypeName*
#define *NewTypeName* **as** { **pointer to** |@|^|.} *OldTypeName*
#define *NewTypeName* **as** { **Handle to** |@@|^^|..} *OldTypeName*

Description:
The **#define** statement is one way to create a name for a variable type (the other way to do so is to use the begin record statement). NewTypeName can be any new name you like that is different from the names of all existing types. OldTypeName is the name of an existing type; this can either be a built-in type such as Rect or int, or a type which you created previously, in a begin record statement or in another **#define** statement. After the **#define** statement, you can declare variables of the new type using dim statements, and you can pass NewTypeName to the sizeof and typeof functions.
If you use the first syntax, NewTypeName essentially becomes a synonym for OldTypeName. If you use the other two syntaxes, then variables of the new type are recognized as pointers or handles to structures of OldTypeName. This is the only way to create a type name for pointers or handles to other types.

Note:
#define is non-executable, so you can't change its effect by putting it inside a conditional execution structure such as long if...end if. A non-executable statement inside a #if block will only be compiled if the condition following the #if is met. Otherwise it will be ignored.

See Also:
begin record; sizeof; typeof; dim

#else   **statement**

See:
#if

#endif   **statement**

See:
#if

#if   **statement**

Syntax:
#if *condition*
  [statementBlock1]
[**#else**
  [statementBlock2]]
#endif

Description:
You can use the **#if** statement to conditionally include or exclude selected lines of code from the compiled version of your program. This is useful if you need to maintain two or more slightly different versions of your program; **#if** allows you to maintain them both within the same source file.
If the condition following **#if** is evaluated as "true" or non-zero, then the statements in *statementBlock1* are included in the compilation and the statements (if any) in *statementBlock2* are ignored by the compiler. If the condition is evaluated as "false" or zero, then the statements in *statementBlock1* are ignored by the compiler, and the statements (if any) in *statementBlock2* are included in the compilation.
The **#if** statement must be matched by a following **#endif** statement.

condition must have one of the following forms:

- *constExpr*
- {def | ndef} *_symbolicConstant*

constExpr is a "static integer expression." A static integer expression is a valid expression which consists only of:

- integer literal constants
- previously-defined symbolic constant names
- operators (like +, -, *, /, >, ==, !=)
- parentheses

In particular, it can't contain variables or function references. If you use this form of **#if**, then the condition will be evaluated as "true" if the expression's value is nonzero.
_symbolicConstant stands for a symbolic constant name.
def _symbolicConstant is evaluated as "true" if the indicated constant was previously defined, regardless of its value.
ndef _symbolicConstant is evaluated as "true" if the indicated constant was not previously defined.

Example:
Because **#if** can cause lines (including non-executable lines) to be completely ignored by the compiler, you can use it to control such things as the declaration of variables, program labels, constants, and even entire functions.

```
_myDebugStuff = 0 // 0 or 1
#if _myDebugStuff
 local fn DebugPrint( a as long )
 ...
 end fn
#endif

_dimensions = 3
#if ( _dimensions == 3 )
 def fn Diag!( a, b, c ) = sqr( a*a + b*b + c*c )
#else
 def fn Diag!( a, b ) = sqr( a*a + b*b )
#endif
```

abs   **function**

Syntax:

positiveValue = **abs** (*expr*)

Description:

The **abs** function returns the absolute value of the numeric expression *expr*, which may be either integer or floating point.

The absolute value of a number is its distance from zero. Thus, the number 3 has an absolute value of 3, while the number -12.34 has an absolute value of 12.34. The absolute value of zero is zero.

acos   **function**

Syntax:

radianAngle = **acos**(*expr*)

Description:

Returns the arccosine of *expr* in radians. In other words, if *expr* represents the cosine of some angle, then **acos**(*expr*) returns the angle. The returned angle will be in the range of 0 to pi. **acos** returns a double-precision result.

See Also:

sin; cos; tan; atn; asin

acosh   **function**

Syntax:
result = **acosh** (*expr*)

Description:
Returns the inverse hyperbolic cosine of *expr* . This is the (positive-valued) inverse of the cosh function, so that **acosh**(cosh(x)) equals abs(x). **acosh** returns a double-precision result.

See Also:
cosh; asinh; atanh

and   **operator**

Syntax:
result = exprA { **and** *| &&} exprB*

Description:
Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **and** operator performs a bitwise comparison of each bit in *exprA* with the bit in the corresponding position in *exprB* . The result is another 32-bit quantity; each bit in the result is determined as follows:

Bit value in exprA	Bit value in exprB	Bit value in result&
0	0	0
1	0	0
0	1	0
1	1	1

Example:
In the following example, expressions are evaluated as true or false before a decision is made for branching. The logical expression `time>7` is true, and is therefore evaluated as -1. The expression `time<8.5` is false, and is therefore evaluated as 0. Then the bitwise comparison `(-1)` **and** `(0)` is performed, resulting in zero. Finally, the `long if` statement interprets this zero result as meaning "false," and therefore skips the first `print` statement.

```
time = 9.5
long if ( time > 7 and time < 8.5 )
   print "It is time for breakfast!"
xelse
   print "We have to wait 'til noon to eat!"
end if
```

The example below shows how bits are manipulated with **and**:

```
defstr long
print bin$(923)
print bin$(123)
print "--------------------------------"
print bin$(923 and 123)
```

program output:
```
00000000000000000000001110011011
00000000000000000000000001111011
--------------------------------
00000000000000000000000000011011
```

Note:
In a statement like `if` *expr1* `and` *expr2* `then`..., it is possible for " *expr1* `and` *expr2* " to be false even though each individual *expr* is evaluated as true. Consider this example:

```
JoeIsHere = 16
FredIsHere = 2
if JoeIsHere then print "Joe's here" else print "Joe's gone"
if FredIsHere then print "Fred's here"¬
   else print "Fred's gone"
long if JoeIsHere and FredIsHere
 print "They're both here"
xelse
 print "They're not both here!"
end if
```

program output:
```
Joe's here
Fred's here
They're not both here!
```

This strange result happens because the expression "16 **and** 2" evaluates to 0, which is then interpreted as "false" by the `long if` statement. This wouldn't have happened if we had set `JoeIsHere` to -1 and `FredIsHere` to -1, because the expression "`-1` **and** `-1`" evaluates to -1.

See Also:

nand; nor; not; xor; or; Appendix D - Numeric Expressions

annuity   **function**

Syntax:

annuityFactor = **annuity**(*rate* , *periods*)

Description:

Returns the double-precison annuity factor for the given interest rate and number of periods. The parameters *rate* and *periods* are double precision variables, and the returned value *annuityFactor* is also a double precision value. The interest rate should be expressed as a fraction of 1; for example, 5.2 percent should be expressed as 0.052.

Note:

annuity uses the following formula:

$$annuityFactor = \frac{1-(1+rate)^{periods}}{rate}$$

See Also:

compound

appearance button   **statement**

Syntax:
appearance button [-] *btnNum*[, [*state*][, [*value*][,¬
[*min*][, [*max*][, [*title$*][, [*rect*][, [*type*]]]]]]]

Description:
The **appearance button** statement puts a new control in the current output window, or alters an existing control's characteristics. After you create a button using the **appearance button** statement, you can use the `dialog` function to determine whether the user has clicked it. You can use the `button close` statement if you want to dispose of the button without closing the window.

When you first create a button with a specific ID (in a given window), you must specify all the parameters up to and including *type*. If you later want to modify that button's characteristics, execute `button` again with the same ID, and specify one or more of the other parameters (except *type*, which cannot be altered). The button will be redrawn using the new characteristics that you specified; any parameter that you don't specify will not be altered.

btnNum	a positive or negative integer whose absolute value is in the range 1 through 2147483647. The number you assign must be different from all other scroll bars or buttons in that window. Negative values build invisible buttons. Positive values build visible buttons.
state	The state may be: `_grayBtn` (0/disabled) `_activebtn` (1/default/active) `_markedBtn` (2/selected)
value, *min*, *max*	generally an integer value for the initial, minimum, and maximum values of a control
title$	a string expression. This parameter also serves to set the text of buttons defined with `_kControlStaticTextProc` or `_kControlEditUnicodeTextProc`.
rect	a rectangle in local window coordinates. You can express it in either of two forms: *(x1,y1)-(x2,y2)* Two diagonally opposite corner points. *@rectAddr* `Pointer` variable which points to a `Rect` type.
type	any of the many types listed in the following text.

Things To Keep In Mind
On button creation, default values supplied for missing parameters (if any) are:

state	`_activeBtn`
value	1
min	0
max	1
title$	null string

You can hide the control with either `button -1` or `appearance button -1` and you can deactivate the control with either `button 1`, `_grayBtn` or `appearance button 1`, `_grayBtn`. Buttons are commonly hidden and revealed as tab panes are brought into or removed from view. The same is true of panes that are changed in response to items such as group pop-up placards.
To read an appearance button's value, use either `x = button( id )` or
`x = button( id, _FBGetCtlRawValue )`

Summary of Appearance Helpers:
The following utility routines will help access information regarding the new appearance buttons:
`actualSize =` **fn ButtonDataSize**`( btnID, part, tagName )`
`theString =` **fn ButtonTextString$**`( btnID )`

Button Types:

While it is not our intention to document completely all of Apple's controls, a few possible types are shown below with examples on how each might be implemented

Push Buttons:
Common push buttons are shown below.
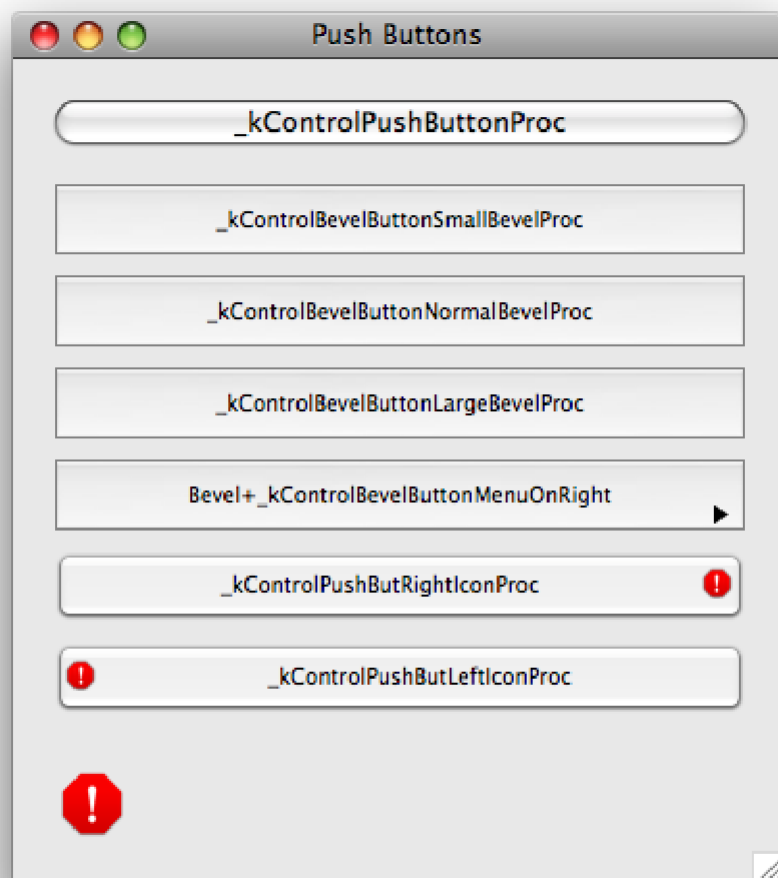
Push buttons

So that you may see how each button was displayed, the following code shows that source used to generate the displays.

```
/*
appearance button [-] btnNum[, [state][, [value][,¬
   [min][, [max][, [title$][,[rect][, [type]]]]]]]
*/
appearance button btnNum,_activeBtn,0,0,1,¬
   "_kControlPushButtonProc",@r,_kControlPushButtonProc
appearance button btnNum,_activeBtn,0,0,1,¬
   "_kControlBevelButtonSmallBevelProc",@r,¬
   _kControlBevelButtonSmallBevelProc
appearance button btnNum,_activeBtn,0,0,1,¬
   "_kControlBevelButtonNormalBevelProc",@r,¬
   _kControlBevelButtonNormalBevelProc
appearance button btnNum,_activeBtn,0,0,1,¬
   "_kControlBevelButtonLargeBevelProc",@r,¬
   _kControlBevelButtonLargeBevelProc
// "value" is menu ID
appearance button btnNum,_activeBtn,101,0,1,¬
   "Bevel+_kControlBevelButtonMenuOnRight",@r,¬
   _kControlBevelButtonSmallBevelProc + ¬
   _kControlBevelButtonMenuOnRight
// max value is cicn ID
appearance button btnNum,_activeBtn,0,0,256,¬
   "_kControlPushButRightIconProc",@r, ¬
   _kControlPushButRightIconProc
appearance button btnNum,_activeBtn,0,0,256,¬
   "_kControlPushButLeftIconProc",@r,¬
   _kControlPushButLeftIconProc
// get rect from pict to determine button size
h = fn GetPicture(256)
long if  h
  pR;8 = @h..picFrame%
 OffsetRect(pR,-pR.left,-pR.top)
```

```
  OffsetRect(pR,r.left,r.top)
// control "value" is pict ID
    appearance button btnNum,_activeBtn,256,0,1,¬
      "_kControlPictureProc",@pR,_kControlPictureProc
end if
```

Not all possible push buttons (and their variations) are shown here. For example, the control that displays an arrow to indicate the presence of a menu was built with a small bevel. It would have been created with a large bevel by using

```
_kControlBevelButtonLargeBevelProc + _kControlBevelButtonMenuOnRight
```

Other button types that you may wish to investigate are:

```
_kControlIconProc
_kControlIconNoTrackProc
_kControlIconSuiteProc
_kControlIconSuiteNoTrackProc
_kControlPictureNoTrackProc
```

Using Buttons to Group or Separate:

FutureBasic buttons (which are Control Manager controls) can be grouped together, placed in placards or separated by lines. The following example creates buttons on a plain white background so that youmay more easily see the drawing that is implemented by the control definition. We begin with the source code statements used to create the buttons.

```
appearance button btnNum,_activeBtn,0,0,1,¬
    "_kControlGroupBoxTextTitleProc",@r,¬
    _kControlGroupBoxTextTitleProc
appearance button btnNum,_activeBtn,0,0,1,¬
    "_kControlGroupBoxSecondaryTextTitleProc",@r,¬
    _kControlGroupBoxSecondaryTextTitleProc
appearance button btnNum,_activeBtn,1,0,1,¬
    "_kControlGroupBoxCheckBoxProc",@r,¬
    _kControlGroupBoxCheckBoxProc
appearance button btnNum,_activeBtn,1,0,1,¬
    "_kControlGroupBoxSecondaryCheckBoxProc",@r,¬
    _kControlGroupBoxSecondaryCheckBoxProc
// min value is menu ID
appearance button btnNum,_activeBtn,1,101,1,¬
    "",@r,_kControlGroupBoxPopUpButtonProc
appearance button btnNum,_activeBtn,1,101,1,¬
    "",@r,_kControlGroupBoxSecondaryPopUpButtonProc
appearance button btnNum,_activeBtn,1,0,1,¬
    "",@r,_kControlPlacardProc
appearance button btnNum,_activeBtn,1,0,1,¬
    "",@r,_kControlSeparatorLineProc
```
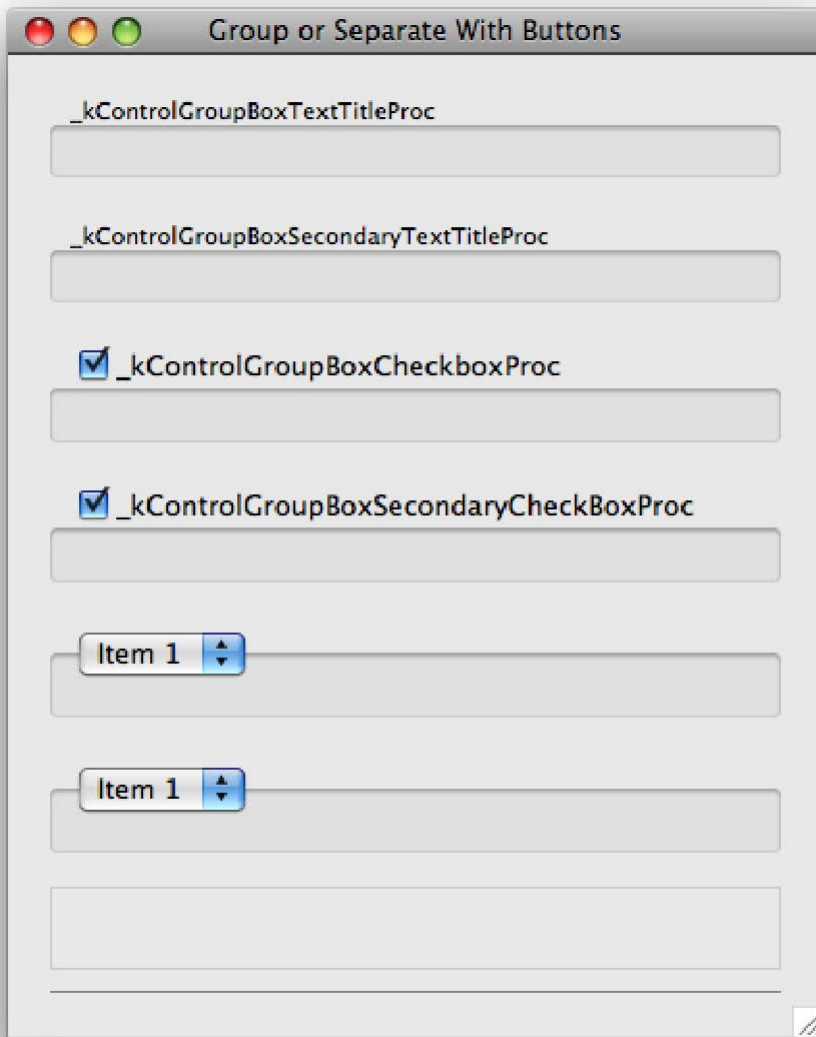
Groups and Separators in MacOS X

Embedding Buttons:

Part of the strength of Appearance Manager buttons is that one button may be embedded in another. By disabling or hiding the parent button (called a super control), all embedded controls would automatically be disabled or hidden. Each window has a primary control known as a root control. The following example builds a window with a parent radio group button. Inside of that parent are three radio buttons. We can determine which of the three buttons has been selected by getting the value (via the `button()` function) of the parent button.

```
dim r as Rect
dim pR as Rect
dim h as Handle
dim btnNum as long
dim err as OSStatus
// setup
_btnHt = 20
_btnWd = 80
_btnMargin = 8
btnNum = 1
// create a window
SetRect(r,0,0,_btnWd_btnMargin_btnMargin,120)
appearance window 1,,@r
err = fn SetThemeWindowBackground (window( _wndPointer ),¬
   _kThemeActiveDialogBackgroundBrush,_zTrue )
// button #1 is the papa button
// note that the parent button has sufficient space so that
// it holds all embedded buttons within its own rectangle
SetRect(r,_btnMargin,_btnMargin,¬
   _btnMargin_btnWd,(_btnMargin_btnHt)*3)
appearance button btnNum,_activeBtn,0,0,1,¬
   "",@r,_kControlRadioGroupProc
btnNum ++
SetRect(r,_btnMargin,_btnMargin,_btnMargin_btnWd,¬
   _btnMargin_btnHt)
```

```
appearance button btnNum,_activeBtn,0,0,1,¬
   "Radio 1",@r,_kControlRadioButtonProc
def EmbedButton(btnNum,1)
btnNum ++ : offsetrect(r,0,_btnHt_btnMargin)
appearance button btnNum,_activeBtn,0,0,1,¬
   "Radio 2",@r,_kControlRadioButtonProc
def EmbedButton(btnNum,1)
btnNum ++ : offsetrect(r,0,_btnHt_btnMargin)
appearance button btnNum,_activeBtn,0,0,1,¬
   "Radio 3",@r,_kControlRadioButtonProc
def EmbedButton(btnNum,1)
local fn handleDialog
  dim as long action,reference
  action = dialog(0)
  reference = dialog(action)
  long if action = _btnclick
  MoveTo(8,100)
  print "Current Button "; button(1);
   end if
end fn
on dialog fn handleDialog
do
 HandleEvents
until gFBQuit
```


Embedded Buttons

Check Boxes

Other than the obvious differences in physical appearance, check boxes generally follow the same guidelines as they have for many years. One notable exception to this rule is the ability to create a mixed check box. This box contains a dash instead of a check mark to show that part, but not all, of the current selection has a specific feature. This adds a new possible maximum value of 2

(`_kControlCheckBoxMixedValue = 2`) to the control's range.

Possible check box values now include:
```
_kControlCheckBoxUncheckedValue
_kControlCheckBoxCheckedValue
_kControlCheckBoxMixedValue
```
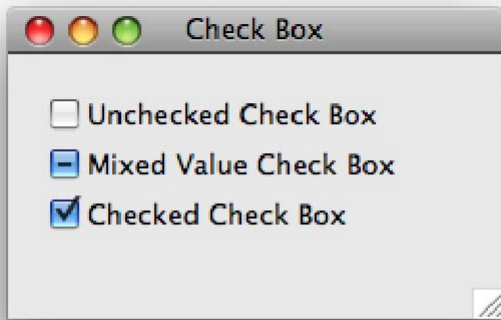
Check Boxes

The buttons in the screen shot above were created using the following lines of code:

```
appearance button btnNum,_activeBtn,¬
   _kControlCheckBoxUncheckedValue,0,¬
   _kControlCheckBoxMixedValue,¬
   "Unchecked Check Box",@r,_kControlCheckBoxProc
appearance button btnNum,_activeBtn,¬
   _kControlCheckBoxMixedValue,0,¬
   _kControlCheckBoxMixedValue,¬
   "Mixed Value Check Box",@r,_kControlCheckBoxProc
appearance button btnNum,_activeBtn,¬
   _kControlCheckBoxCheckedValue,0,¬
   _kControlCheckBoxMixedValue,¬
   "Checked Check Box",@r,_kControlCheckBoxProc
```

Note:
You cannot use `button btnNum,state` to tick and untick group buttons of type `_kControlGroupBoxCheckBoxProc` and `_kControlGroupBoxSecondaryCheckBoxProc`. Use `appearance button btnNum,,state-1` instead. (`button btnNum,0` and `button btnNum,1` will however inactivate and activate the button respectively).

Wait States:
The Appearance Manager provides several methods for telling the user that your application is busy with a task. These include chasing arrows, and finite and indeterminate progress bars.
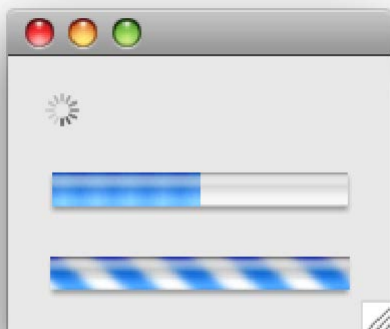
Wait States

The chasing arrows control is easy to create and is self maintaining. Each time your program scans for events, the arrows are animated. The following statement creates a chasing arrows control:

```
appearance button btnNum,_activeBtn,0,0,1,,@r,¬
   _kControlChasingArrowsProc
```

Progress bars are also easy to create, but you need to keep a couple of things in mind. First, the progress bar operates in a range of -32,768 to +32,767. If your task involves a greater number of steps, you will have to calculate a ratio to keep things within range. Second, the progress bar is updated by your program. This is as easy as setting a new value for the button, but it is code that you must write.

The minimum and maximum values for the control become the minimum and maximum values for the progress bar. The initial and current value show the current rate of progress. In the example above, the button was created using the following source:

```
appearance button btnNum,_activeBtn,50,0,100,,@r,¬
   _kControlProgressBarProc
```

The minimum value was zero; maximum was 100. At the time of creation, the control value was 50, so the indicator shows colorization half

way across the bar. If we wanted to indicate that the next step had been completed, we would use the following code:

appearance button `btnNum,,51`

Indeterminate progress bars are more complex. After the button is created, you must set the control's internal data to a new value. The following code shows how:

```
appearance button btnNum,_activeBtn,1,0,1,,@r,¬
   _kControlProgressBarProc
dim b    as Boolean
dim err as OSStatus
b = _true
err = fn SetControlData(btnNum, 0,¬
   _kControlProgressBarIndeterminateTag,sizeof(Boolean), @b)
```

Range Selectors (Sliders and Arrows):

There are many variations of the slider. Each begins with the simple type constant of `_kControlSliderProc`. Additional parameters are added to this constant to add features to the control. The following constants are available for slider variations:

```
_kControlSliderLiveFeedback
_kControlSliderHasTickMarks
_kControlSliderReverseDirection
_kControlSliderNonDirectional
```

To create a slider that uses an upward pointing indicator and has tickmarks, the following type would be used:

```
_kControlSliderProc + ¬
   _kControlSliderHasTickMarks + ¬
   _kControlSliderReverseDirection
```

Vertical sliders are created by building the button with a vertical dimension that is greater than the horizontal dimension. The control definition handles the new orientation automatically.
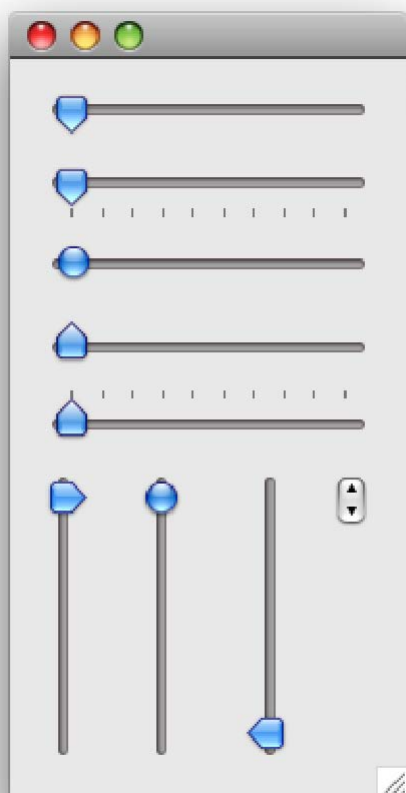
Sliders and Little Arrows

The following source lines show how this display was created:

```
appearance button btnNum,_activeBtn,1,1,10,,@r,¬
   _kControlSliderProc
appearance button btnNum,_activeBtn,10,1,10,,@r,¬
   _kControlSliderProc_kControlSliderHasTickMarks
appearance button btnNum,_activeBtn,1,1,10,,@r,¬
   _kControlSliderProc_kControlSliderNondirectional
appearance button btnNum,_activeBtn,1,1,10,,@r,¬
   _kControlSliderProc_kcontrolSliderReverseDirection
appearance button btnNum,_activeBtn,10,1,10,,@r,¬
   _kControlSliderProc_kControlSliderHasTickMarks +¬
   _kcontrolSliderReverseDirection
// vert orientation
appearance button btnNum,_activeBtn,10,1,10,,@r,¬
```

```
    _kControlSliderProc
```
appearance button `btnNum,_activeBtn,10,1,10,,@r,¬`
```
    _kControlSliderProc_kControlSliderNondirectional
```
appearance button `btnNum,_activeBtn,10,1,10,,@r,¬`
```
    _kControlSliderProc_kControlSliderHasTickMarks +¬
    _kcontrolSliderReverseDirection
```
appearance button `btnNum,_activeBtn,0,0,1,,@r,¬`
```
    _kControlLittleArrowsProc
```

When sliders are created, the number of tick marks is set by the initial value of the control. After the control is created, the value is reset to the control minimum. Sliders range from a minimum value of -32,768 to a maximum of +32,767. The number of tick marks is something that you need to determine by balancing the size of the control against the range of the control's minimum/maximum value.

The little arrows (shown in the screen shot above) are used to increment and decrement a related visual counter (usually an edit field with a specific range of numbers). The current version of the MacOS X control definition is intolerant of variations in the value used for the height of this type of control. Our tests show that it must be exactly 22 pixels tall. Other values offset the arrows inside of the beveled area or, in more extreme cases, can place the arrows entirely outside of the beveled area.

Pop-Up Menus:
There are two distinct types of pop-up menus: beveled, and standard. Both are valid types and the particular use of one over the other is something that should be guided by your individual application and by Apple's Human Interface Guidelines. Beveled buttons are created as follows:

appearance button `btnNum,_activeBtn,menuID,0,1,¬`
```
    "Bevel+_kControlBevelButtonMenuOnRight",¬
    @r,_kControlBevelButtonSmallBevelProc + ¬
    _kControlBevelButtonMenuOnRight
```
When bevel-button menus are created, the initial value for the control is the resource ID number of the menu. Three specific `button` function commands may be used to extract information from the control.

```
handle       = button(btnNum, _FBgetBevelControlMenuHandle)
currentItem  = button(btnNum, _FBgetBevelControlMenuVal)
previousItem = button(btnNum, _FBgetBevelControlLastMenu)
```
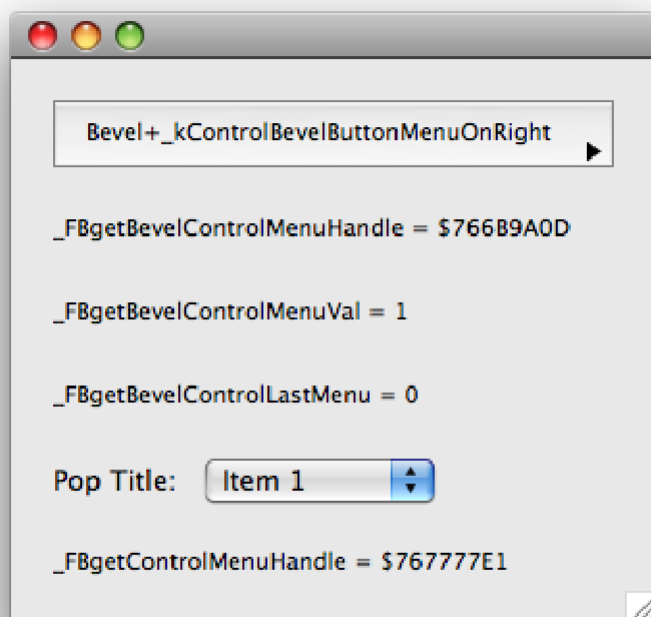

Pop-Up Menu Buttons

Standard pop-up buttons follow slightly different syntax. When creating, the minimum value specifies the menu resource ID and the maximum value is the width of the title for the menu. Passing in a menu ID of -12345 causes the popup not to try and get the menu from a resource. Instead, you can build the menu and later stuff the menuhandle field in the popup data information (using `def SetButtonData (btnNum, _kControlMenuPart, _kControlPopupButtonMenuHandleTag, sizeof( handle), @yourMenuHndl )`). You can pass -1 in the *max* parameter to have the control calculate the width of the title on its own instead of guessing and then tweaking to get it right. It adds the appropriate amount of space between the title and the popup. A maximum value of zero means, "Don't show the title."
After creation you might need to change the value, minimum and maximum to the correct settings for your pop-up menu with:

appearance button `btnNum,,value,min,max`

The standard pop-up button menu in the above illustration was created with the following code:

appearance button `btnNum,_activeBtn,0,101,-1,"Pop Title:"¬`
```
    ,@r,_kControlPopUpButtonProc
```
A single `button` function provides access to the menu handle. Remember: standard and beveled pop-up menus do not use the same

`button` function constants.

```
menuHandle = button(btnNum, _FBgetControlMenuHandle)
```

You can retrieve the current pop-up menu item with:

```
mItem = button(btnNum)
```

Tab Buttons:

Tab buttons will require more work than other controls. This stems from the fact that tabs are really several controls that act in unison. First is the main tab control. When this is created you specify the number of tabs that will be present by setting the max value of the control. It is generally better to create the tab button invisibly (by using a negative button reference number) then show it by issuing a `button` statement with the positive version of the reference number.

After the initial shell is built for the tab, you must set the title for each tab using `def SetButtonData` . Then a user pane is inserted for each tab. These are embedded in the tab shell using `def EmbedButton`. Buttons that will reside in each user pane are created and embedded in the user pane.

When a dialog event is encountered, the value of the tab shell button corresponds to the position of the clicked tab in the tab list. Your program must loop through each user pane and show or hide them so that the display matches the clicked tab.

Study the following example to see how a working tab button is created. Be sure to note the simple `dialog` handler that maintains the buttons.
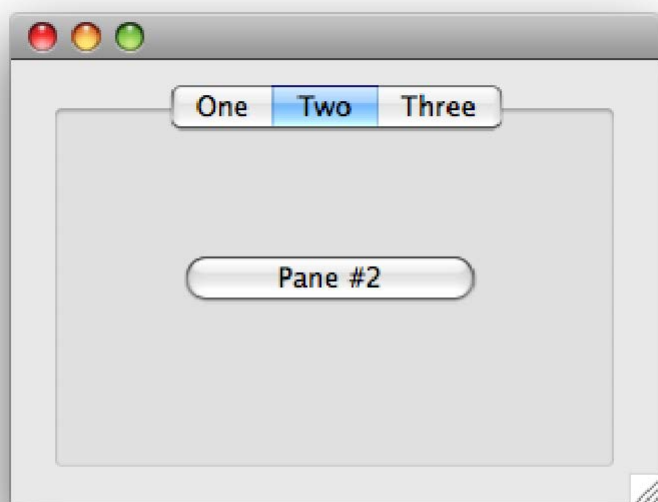
Tab Buttons

There are many styles of tab buttons:

```
_kControlTabLargeProc
_kControlTabSmallProc
_kControlTabLargeNorthProc
_kControlTabSmallNorthProc
_kControlTabLargeSouthProc
_kControlTabSmallSouthProc
_kControlTabLargeEastProc
_kControlTabSmallEastProc
_kControlTabLargeWestProc
_kControlTabSmallWestProc
```

This example uses `_kControlTabSmallProc`, but you should experiment with other types to see the results.

dim `r`   **as Rect**

dim `x`   **as long**

dim `btnNum`   **as long**

dim `infoRec` **as** `ControlTabInfoRec`

```
// Names of the individual tabs
_tabCount = 3
```

dim `tabTitles$(_tabCount)`

```
tabTitles$(1) = "One"
tabTitles$(2) = "Two"
tabTitles$(3) = "Three"
// create a window
```

SetRect`( r, 0, 0, 300, 200 )`

appearance window `1, "Tabs", @r, _kDocumentWindowClass`

def SetWindowBackground`( _kThemeActiveDialogBackgroundBrush,¬`
  `_zTrue )`

```
/*
Button 100 is the tab 'shell'. In this example, it is made to be the full size of the window, less a
small margin. Buttons 1, 2, & 3 will be the embedded user panes that contain information to be
displayed for each tab.
A tab control is usually built as invisible. This is because the information contained in the tabs will
be modified as the control is being constructed. Making it visible after all modifications have been
completed provides a cleaner window build.
```

```
*/
_tabBtnRef = 100
_btnMargin = 8
InsetRect( r, _btnMargin, _btnMargin )
appearance button -_tabBtnRef, 0, 0, 2, _tabCount,, @r,¬
  _kControlTabSmallNorthProc
/*
Fix the tab to use a small font. This is not a requirement, but it is information which many will find
useful. */
dim cfsRec as ControlFontStyleRec
cfsRec.flags = _kControlUseSizeMask
cfsRec.size = 9
def SetButtonFontStyle( _tabBtnRef, cfsRec )
/*
Adapt a rectangle that can be used for the content area of each tab.
*/
InsetRect( r, _btnMargin, _btnMargin )
r.top += 20
// Loop thru the tabs and set up individual panes
for x = 1 to _tabCount
    infoRec.version    = _kControlTabInfoVersionZero
    infoRec.iconSuiteID = 0
    infoRec.Name       = tabTitles$(x)
    def SetButtonData ( _tabBtnRef, x, _kControlTabInfoTag, ¬
  sizeof( infoRec ), infoRec )
    /*
    Each of these panes is a button that is embedded in the
    tab button. The first    one will be visible. All others will
    be invisible because information from only one tab at a time
    can be viewed.
    Remember: negative button reference numbers make
    invisible buttons.
    Once a new pane button (_kControlUserPaneProc) is created,
    it is embedded into the larger tab button.
    */
  if x != 1 then btnNum = -x else btnNum = x
    appearance button btnNum,,¬
    _kControlSupportsEmbedding,,,, @r,¬
    _kControlUserPaneProc
    def EmbedButton( x, _tabBtnRef )
next
/*
Now we have a tab shell (_tabBtnRef = 100) and in it we have embedded three tab panes (1,2, and 3). To
demonstrate how these can contain separate info, we will place a simple button in each of the three
panes.
    Button 10 in pane 1
    Button 20 in pane 2
    Button 30 in pane 3
*/
InsetRect( r, 32, 32 )
r.bottom = r.top + 24
r.right = r.left + 128
appearance button 10, _activeBtn,,,,¬
  "Pane #1", @r, _kControlPushButtonProc def EmbedButton( 10, 1 )
OffsetRect( r, 8, 8 )
appearance button 20, _activeBtn,,,,¬
  "Pane #2", @r, _kControlPushButtonProc def EmbedButton( 20, 2 )
OffsetRect( r, 8, 8 )
appearance button 30, _activeBtn,,,,¬
  "Pane #3", @r, _kControlPushButtonProc
def EmbedButton( 30, 3 )
button _tabBtnRef, 1 // make visible
/*
Only one event (a button click in the tab shell button)
gets a response from out dialog routine. The value returned
(1,2, or 3) corresponds to buttons 1,2, or 3 that were
embedded into the tab parent.
Our only action is to show (button j) or hide (button -j)
the proper tab pane. All controls embedded in those panes
will automatically be shown or hidden.
*/
local fn doDialog
  dim as long action, reference, j
  action    = dialog(0)
  reference = dialog(action)
  long if action == _btnClick and reference == _tabBtnRef
  for j = 1 to _tabCount
      long if j == button(_tabBtnRef)
    button j
   xelse
    button -j
      end if
  next
  end if
```

end fn
on dialog fn `doDialog`
do
  HandleEvents
until `gFBQuit`

See Also:
button&; button function; button close; scroll button; dialog function

appearance window statement

Syntax:
appearance window `[-]` *wNum*`[,` `[`*title$*`][,` `[`*rect*`][,` `[`*windowClass*`][,` `[`*windowAttributes*`] ]]]]`

Description:
Use this statement to do any of the following:

- Create a new screen window;
- Activate (highlight and bring to the front) an existing window;
- Make an existing window visible or invisible;
- Alter the title or rectangle of an existing window.

The **appearance window** statement closely follows the older `window` statement, but is used primarily for the creation of windows. You may freely mix `window` functions and `window` statements with windows that are created via the **appearance window** statement. For instance, after creating a window with the **appearance window** statement, you could determine its size with the `window(_width)` and `window(_height)` functions.
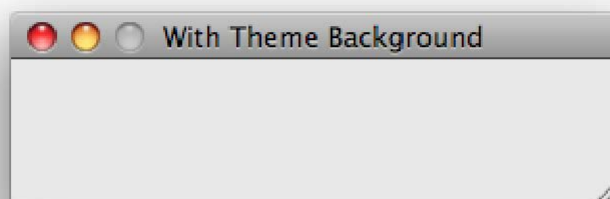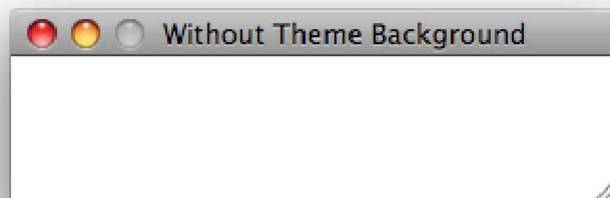
Windows with and without theme backgrounds

The parameters for **appearance window** should be specified as follows. They are interpreted slightly differently depending on whether you are creating a new window or altering an existing one.

- *wNum* - a positive or negative integer whose absolute value is in the range 1 through 2147483647.
- *title$* - a string expression.
- *rect* - a rectangle in global screen coordinates. You can express it in either of two forms:

(x1,y1)-(x2,y2) Two diagonally opposite corner points.
@rectAddr& Long integer expression or `pointer` variable which points to an 8-byte struct such as a `Rect` type
  windowClass - an unsigned long integer that specifies the Macintosh window class. To create a *windowClass* variable use the following syntax:
dim `wc` **as** `WindowClass`

windowClass	Description
`_kAlertWindowClass`	I need your attention now.

_kMovableAlertWindowClass		I need your attention now, but I'm kind enough to let you switch out of this app to do other things
_kModalWindowClass		system modal, not draggable
_kMovableModalWindowClass		application modal, draggable
_kFloatingWindowClass		floats above all other application windows. Available in OS 8.6 or later
_kDocumentWindowClass		document windows
_kDesktopWindowClass		the desktop
_kHelpWindowClass		help windows
_kSheetWindowClass		sheets
_kToolbarWindowClass		floats above docs, below floating windows
_kPlainWindowClass		plain
_kOverlayWindowClass		overlays
_kSheetAlertWindowClass		sheet alerts
_kAltPlainWindowClass		plain alerts

windowAttributes - this unsigned long integer describes the features and widgets available to a window such as a close box, grow box, or a collapse box. You can dimension a windowAttributes variable as follows:

dim wa **as** WindowAttributes

windowAttributes		Description
_kWindowNoAttributes		none
_kWindowCloseBoxAttribute		close box
_kWindowHorizontalZoomAttribute		horizontal zoom
_kWindowVerticalZoomAttribute		vertical zoom
_kWindowFullZoomAttribute		standard zoom
_kWindowCollapseBoxAttribute		collapse box (sends to MacOS X dock)
_kWindowResizableAttribute		grow box
_kWindowSideTitlebarAttribute		title on side for floating window
_kWindowNoUpdatesAttribute		does not receive update event
_kWindowNoActivatesAttribute		does not receive activate event
_kWindowToolbarButtonAttribute		has a toolbar button in title bar
_kWindowNoShadowAttribute		no drop shadow
_kWindowLiveResizeAttribute		resize events repeatedly sent while window is being sized
_kWindowStandardDocumentAttributes		close box, zoom box, collapse box, grow box
_kWindowStandardFloatingAttributes		close box, collapse box

To Create a New Window

- Specify a *wNum* value such that abs(*wNum*) is different from the ID number of any existing window. A new window is created and is assigned an ID number of abs(*wNum*). You can use the window's ID number later to identify the window in other FutureBasic statements and functions. If *wNum* is negative, the window is created invisibly; it's sometimes useful to create a window invisibly if it will contain controls, edit fields and graphics that may take a long time to build. You can use the window statement again to make an invisible window visible (see below). When you create a new window, it becomes the current output window. If you create it visibly, it also becomes the current active window.
- *title$* assigns a string to the window's title bar (if the window has a title bar). If you omit this parameter, the window will be created without a title.
- *rect* specifies the initial size and location of the window's content rectangle. Note that *rect* does not include the window's frame. This parameter is interpreted in a special way if you specify an upper-left coordinate of (0,0) in *rect*; in this case, the window is centered in the screen, and its width and height are determined by the right and bottom coordinates of *rect*. Note that this special interpretation applies only when you're creating a new window. If you omit this parameter, a window of a "default" size and location is created.
- *windowClass* specifies the layer in which a window will reside.

- *windowAttributes* specifies the types of window widgets (close box, grow box) that a window will include.

See Also:

MinWindow; MaxWindow; get window; window close; window output; window function

append   **statement**

Syntax:
append [#] *fileID*

Description:
This statement moves the file mark, in the currently-open file indicated by *fileID*, to the end-of-file position (without overwriting any of the existing data in the file). This causes subsequent file output statements such as `print#, write` and `write file#` to append data to the end of the file.

Example:
In the following, note that we open the output file using the "R" method. This is because opening with the "O" method causes the "end-of-file" mark to move to the beginning of the file, effectively erasing any existing data.
```
A$ = "TESTING..."
open "R", 1, @fSpec // open an existing file
append #1 // set file pointer to end
write #1, A$;25 // add data to end of file
close #1
```

See Also:
files$; open; close; read#

apple menu   **statement**

Syntax:
apple menu *string$*

Description:
This statement inserts one or more items at the top of the Application Menu, and separates them from the existing Application Menu items with a grey dividing line. The *string$* parameter contains the text of the item(s); to add multiple items, separate them with semicolons in *string$*. Certain "meta characters" in *string$* have special interpretations; see the menu statement for more information.
After you add items to the Apple Menu, you can use the menu function to detect when the user selects one of the items you've added.
If you execute **apple menu** more than once, any items you added to the Apple Menu previously will be completely replaced.

Example:
apple menu "About This Program;About Me..."
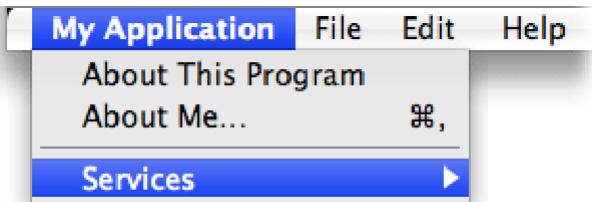
See Also:
menu function; menu statement; on menu fn; HandleEvents

asc   **function**

Syntax:

charCode = **asc**(*string$*)

charCode = **asc**(*container$$*)

Description:

Returns the ASCII character code for the first character in *string$* or *container$$*. If *string$* (or *container$$*) has zero length, then **asc**(*string$*) returns zero.

A character code is a numeric value in the range of 0 through 255 that represents a specific character in the American Standard Code for Information Interchange (ASCII). The ASCII character codes between 32 and 127 represent standard characters which generally remain the same from one font family to another. Codes greater than 127 represent different sets of characters depending on the font. Codes below 32 usually represent non-printing "characters."

Note:

If the string is a single-character literal, such as "G", consider using the underscore-literal syntax instead, as in this example:

charCode = _"G"

The above code executes much faster than charCode = **asc**("G").

You can use the following syntax to return the ASCII code of the n–1-th character in a string variable:

charCode = stringVar$[n]

See Also:

chr$; Appendix F - ASCII Character Codes

asin   **function**

Syntax:

radianAngle = **asin** (*expr*)

Description:

Returns the arcsine of *expr* in radians. In other words, if *expr* represents the sine of some angle, then **asin** (*expr*) returns the angle. The returned angle will be in the range of -π/2 to +π/2 radians (which corresponds to -90 to +90 degrees). **asin** returns a double-precision result.

See Also:

sin; cos; tan; atn; acos

asinh function

Syntax:
result = **asinh**(*expr*)

Description:
Returns the inverse hyperbolic sine of *expr*. This is the inverse of the sinh function, so that **asinh**(sinh(x)) equals x. **asinh** returns a double-precision result.

See Also:
sinh; acosh; atanh

atan   **function**

Syntax:
radianAngle = **atan** (*expr*)

Description:
atan is a synonym for the atn function.

atanh   **function**

Syntax:
result = **atanh**(*expr*)

Description:
Returns the inverse hyperbolic tangent of *expr*. This is the inverse of the `tanh` function, so that **atanh**`(tanh(x))` equals `x`. **atanh** returns a double-precision result.

See Also:
tanh; acosh; asinh

atn   **function**

Syntax:

radianAngle = **atn**(*expr*)

Description:

Returns the arctangent of *expr* in radians. In other words, if *expr* represents the tangent of some angle, then **atn**(*expr*) returns the angle. The returned angle will be in the range of -π/2 to +π/2 radians (which corresponds to -90 to +90 degrees). **atn** returns a double-precision result.

See Also:

sin; cos; tan; asin; acos; atan

beep   **statement**

Syntax:
beep

Description:
This statement produces a system beep as defined by the Sound panel of System Preferences. Useful for alerting the user that the application needs attention.

See Also:
sound

begin enum   **statement**

Syntax:
begin enum `[[not] output] [start [,inc]]`
  `_constName1 [= staticExpression1]`
  `_constName2 [= staticExpression2]`
end enum

Description:
This statement begins a block of "enumerated constant" definition lines. The block must be terminated with the **end enum** statement. All of the constants defined in this block are global, regardless of where in the program the block appears.
The `begin enum...end enum` block is "non-executable," which implies that it won't be repeated or skipped if it appears within any kind of "conditional execution" block, such as `for...next`, `long if...end if`, `do...until`, etc. (but it can be conditionally included or excluded if it appears inside a `#if` block).
Each _constName represents a symbolic constant name that has not previously been defined, and each *staticExpression* represents an integer expression which consists only of:

- integer literal constants;
- previously-defined symbolic constant names;
- operators (like +, -, *, /, >, =);
- parentheses

In particular, it can't contain variables or function references.
The **begin enum** block assigns values to each of the _constName symbolic constants as follows:

- If the _constName is followed by `=` *staticExpression*, then _constName is assigned the value of *staticExpression*;
- If the _constName is not followed by `=` *staticExpression*, then _constName is assigned the value of the _constName in the line above it, plus the value of *inc*;
- If the very first _constName is not followed by `=` *staticExpression*, then it's assigned the value of *start*.

The *start* and *inc* parameters, if included, must each be a static integer expression. The default value of *start* is 0, and the default value of *inc* is 1.

Example:
In the following, the dwarves are assigned values of 1 through 7; `_snowWhite` is assigned the value 100, and `_thePrince` is assigned the value 101.

begin enum `1`
  `_docDwarf`
  `_sneezy`
  `_grumpy`
  `_sleepy`
  `_dopey`
  `_happy`
  `_bashful`
  `_snowWhite = 100`
  `_thePrince`
end enum

Output Option:
The **output** option, introduced in FB 5.6.1, improves the readability of translated C code. Without this option, _constants and static expressions in FB source are translated to 'magic numbers' whose names are lost.

Consider this FB code:
begin enum
  `_foo = 3`
  `_bar = 42`
end enum
`if ( _foo == _bar ) then ...`

The C translation has magic numbers:
  `if ( 3 == 42 ) { ... }`

Here's the same example, with the `output` option added:

begin enum output
```
   _foo = 3
   _bar = 42
```
end enum
```
if ( _foo == _bar ) then ...
```

In the C translation, an enum statement is output for each constant, allowing its name to be used later:
```
   enum { foo = 3 };
   enum { bar = 42 };
   ...
   if ( foo == bar ) { ... } // symbols instead of numbers
```

Not Output Option:

The **not output** option improves the readability of translated C code in the same way as **output**. This form is used for system constants already known to the compiler, mainly those in FB Headers files. The C translation doesn't contain enum statement for the constants, because they would cause compile-time duplicate definition errors.

begin globals   **statement**

Syntax:
begin globals
  `[statements including variable declarations]`
end globals

Description:
The **begin globals** and **end globals** statements indicate the beginning and end, respectively, of a section of global variable declarations. A global variable is one which is "visible" to all parts of the program that follow its declaration (except **local mode** functions). It maintains its value when local function are entered or exited. Global variables are set to zero at program startup.

FB places an implicit **begin globals** statement at the beginning of your program. That means that, by default, all variables declared in "main" are global. You must include an **end globals** statement in "main" if you want any of the variables declared in "main" to be local to "main."

By judicious placement of `begin globals...end globals` statements, you can also create variables which are considered "global" to some local functions but not to others.
begin globals and **end globals** are "non-executable" statements, so you can't change their effect by putting them inside a conditional execution structure such as `long if...end if`. However, you can conditionally include or exclude them from the program by putting them inside a `#if` block.
You may include any number of `begin globals...end globals` pairs in your program. You may also include `begin globals...end globals` pairs in local functions. They must occur in matched pairs when they occur within a local function, and should normally be in matched pairs when they occur in the "main" part of your program (the "main" part consists of those lines which are outside of all local functions). When you include a `begin globals...end globals` section in "main," it should not enclose any local functions, or variables may be scoped in unpredictable ways.
When a variable's first appearance within "main" occurs within a `begin globals...end globals` section, that variable is declared as global to all local functions which appear below that section.

See Also:
end globals

begin record   **statement**

Syntax:
begin record *TypeName*
  `recDefnBlock`
end record

Description:
Begins the definition of a "record" type. The record type definition must end with an `end record` statement.
A `begin record...end record` block is non-executable, so you can't change its effect by putting it inside a conditional execution structure such as `long if...end if`. However, you can conditionally include or exclude it from the program by putting it inside a `#if` block.
The record type defined in the `begin record...end record` block is global in scope, and can be used anywhere below where the block appears.
TypeName is a name that identifies the record type. This name must be unique among all defined record types in your program.
`recDefnBlock` is a block of one or more `dim` statements. These `dim` statements have a syntax which is identical to that of an ordinary `dim` statement. However, instead of declaring variables, these `dim` statements declare the names and types of the fields in this type of record. The field names do not need to be unique to this type of record (that is, a different record type could use some of the same field names as this record type). A field can be of any data type, including a previously-defined record type. A field may also be an array of elements of any type.

After a record type has been defined using `begin record...end record`, it can be used just like any other data type. This means that you can declare:

- variables of type *TypeName*;
- arrays of type *TypeName*;
- fields in other record types as having type *TypeName*.

Anywhere below the **end record** statement, you can use the `dim` statement, along with the `as` keyword, to declare a variable, array or field of type *TypeName*. For example, if you have defined a record type called `Address`, then you can do the following:
dim `myHouse` **as** `Address, yourHouse` **as** `Address`
dim `relatives(15)` **as** `Address`
begin record `EmployeeInfo`
 dim as Str63 `name`
 dim `9 socSecNo$`
 dim `20 hobbies$[9]`
 dim `empAdress` **as** `Address`
end record
After you have declared a variable of a given record type, then you can use the "embedded dot" syntax to refer to individual fields within the record. Using the above example:
dim `mySecretary` **as** `EmployeeInfo`
`mySecretary.socSecNo$ = "456-78-9999"`

Arrays of records
When you use arrays of records, you always write the array subscript at the end of the expression, whether the expression indicates an entire record or one of its fields.
Example:
begin record `StudentInfo`
 dim as Str63 `firstName, lastName`
 dim `1 finalGrade$`
end record
dim `myStudents(35)` **as** `StudentInfo`
`'This represents the final grade of myStudent #14:`
`myStudents.finalGrade$(14)= "B"`

Arrays inside records
An array field is declared in a similar fashion to an ordinary array declaration, except that square brackets are used instead of parentheses. Brackets are used for both dimensioning and accessing the array's elements.
Example:

          `StudentInfo`

begin record
  dim as Str63 `firstName, lastName`
  dim `grades[100] // note square brackets`
end record
dim as `StudentInfo myStudents(35)`
`myStudents.grades[1](5) = 96`
In the final line of this example, the grade element number 1 of student number 5 is set to 96.

See Also:
dim; begin union; sizeof

begin union   **statement**

Syntax:
begin record *recordName*
 dim *statements* . . .
   begin union
   dim *statements*
   end union
 dim *statements* . . .
end record

Description:
A union specifies two or more variables whose storage begins at the same offset in the record. The following example sets aside two equal offsets within a record for variables of differing sizes:

begin record RecordWithUnion
 dim as long beforeUnion
 begin union
  dim as UInt8 inUnion1
  dim as Str255 inUnion2
  end union
end record
dim as RecordWithUnion myTest
myTest.inUnion2 = "COW"
print myTest.inUnion1

The variable myTest.inUnion1 is a single byte which occupies the same space as the first byte in the string myTest.inUnion2. In this case, myTest.inUnion1 happens to be the length byte of the string and the print statement will produce "3". Such an overlap is not necessary and the two values may have no relation to one another except that they start at the same location in memory.

When FutureBasic encounters a **begin union** statement, all **dim** statements up to the **end union** statement are examined and the largest item in the union determines the amount of space set aside by the compiler. In the example above, the union would occupy 256 bytes since the largest element in the union is a 256 byte Pascal string.

See Also:
dim; begin record

BeginCCode **statement**

Syntax:
BeginCCode
 C statements
EndC

Description:
Marks the beginning of a block of C language statements. The block must be terminated with the **EndC** statement. The C statements are copied untranslated into the C source code, then compiled by the C compiler. **BeginCCode** is intended to replace and has advantages over the old **#if def _PASSTHROUGH / #endif** syntax. The **#if def _PASSTHROUGH / #endif** syntax continues to be available.

Unlike **#if def _PASSTHROUGH / #endif**, **BeginCCode** does *not* interfere with editor indentation and C keywords within the block are not highlighted.

#if def _PASSTHROUGH
// FBtoC sees this; FutureBasic does not
// FBtoC passes everything, except comments, untranslated to the compiler
// passed C code goes in current function or main()
#endif

could be written as:

BeginCCode
// FBtoC passes everything, except comments, untranslated to the compiler
// passed C code goes in current function or main()
EndC

See Also:
BeginCFunction; BeginCDeclaration; EndC; #if

BeginCDeclaration   **statement**

Syntax:
BeginCDeclaration
  C declarations or preprocessor directives
EndC

Description:
Marks the beginning of a block of C language statements and is typically used to pass an #include statement to a translated .h file, for example:

BeginCDeclaration
 #include <OpenGL/glu.h>
EndC

The block must be terminated with the **EndC** statement. The C statements are copied untranslated into the C source code, then compiled by the C compiler.

Example:

C functions

// declarations of all kinds (these go in *.h)
BeginCDeclaration
  void FooBeep();
EndC

// definitions of functions and global vars (these go in *.c)
BeginCFunction
  void FooBeep()
  {
    SysBeep(1);
  }
EndC

toolbox FooBeep() // declare symbol for FBtoC

In your FB source, FooBeep may now be used instead of the keyword beep.

C global variables

Defined constants (_myConst, _versionString$) are in general preferred to literals (42, "1.3"), for reasons of readability and ease of maintenance. The example shows how to obtain the CFString equivalent of _versionString$. A global CFStringRef is initialized at compile time, and given the "const" attribute so that it cannot be modified at run time.

// declarations of all kinds (these go in *.h)
BeginCDeclaration
  extern const CFStringRef kFooVersionString;
EndC

// definitions of functions and global vars (these go in *.c)
BeginCFunction
  const CFStringRef kFooVersionString = CFSTR("1.3");
EndC

system CFStringRef kFooVersionString // declare symbol for FBtoC

In your FB source, kFooVersionString may now be used instead of @"1.3" or fn CFSTR("1.3"), with exactly the same effect.

Objective-C

compile as "Objective-C"
// declarations of all kinds (these go in *.h)
BeginCDeclaration
```
@interface FooThing : NSObject {
  NSWindow *window;
}
- (void)buildWindow;
@end
void BuildWindow( void );
```
EndC

// definitions of functions and global vars (these go in *.m)
BeginCFunction
```
@implementation FooThing
- (void)buildWindow {
  window = [[NSWindow alloc] initWithContentRect:NSMakeRect( 0, 0, 300, 300 )
          styleMask:NSTitledWindowMask
          backing:NSBackingStoreBuffered defer:NO];
  [window center];
  [window setTitle:@"FooThing"];
  [window makeKeyAndOrderFront:nil];
}
@end
void BuildWindow( void )
{
  FooThing *ft = [[FooThing alloc] init];
  [ft buildWindow];
  [ft release];
}
```
EndC

// declare symbols for FBtoC
```
toolbox BuildWindow
toolbox fn NSApplicationLoad = Boolean
```

// main
```
fn NSApplicationLoad()
BuildWindow()
RunApplicationEventLoop()
```

See Also:
BeginCCode; BeginCFunction; EndC; #if

BeginCFunction   **statement**

Syntax:
BeginCFunction
 C statements
EndC

Description:
Marks the beginning of a block of C language statements. The block must be terminated with the **EndC** statement. The C statements are copied untranslated into the C source code, then compiled by the C compiler. **BeginCFunction** is intended to replace and has advantages over the old **#if def _PASSTHROUGHFUNCTION / #endif** syntax. The **#if def _PASSTHROUGHFUNCTION / #endif** syntax continues to be available.

Unlike **#if def _PASSTHROUGHFUNCTION / #endif**, **BeginCFunction** does *not* interfere with editor indentation and C keywords within the block are not highlighted.

#if def _PASSTHROUGHFUNCTION
// FBtoC sees this; FutureBasic does not
// FBtoC passes everything, except comments, untranslated to the compiler
// the C code, typically a function definition, goes before main()
#endif

could be written as:

BeginCFunction
// FBtoC passes everything, except comments, untranslated to the compiler
// the C code, typically a function definition, goes before main()
EndC

Example:

C functions

// declarations of all kinds (these go in *.h)
BeginCDeclaration
 void FooBeep();
EndC

// definitions of functions and global vars (these go in *.c)
BeginCFunction
 void FooBeep()
 {
  SysBeep(1);
 }
EndC

toolbox FooBeep() // declare symbol for FBtoC

In your FB source, FooBeep may now be used instead of the keyword beep.

C global variables

Defined constants (_myConst, _versionString$) are in general preferred to literals (42, "1.3"), for reasons of readability and ease of maintenance. The example shows how to obtain the CFString equivalent of _versionString$. A global CFStringRef is initialized at compile time, and given the "const" attribute so that it cannot be modified at run time.

// declarations of all kinds (these go in *.h)
BeginCDeclaration

```
   extern const CFStringRef kFooVersionString;
```
EndC

// definitions of functions and global vars (these go in *.c)
BeginCFunction
```
   const CFStringRef kFooVersionString = CFSTR( "1.3" );
```
EndC

```
system CFStringRef kFooVersionString // declare symbol for FBtoC
```

In your FB source, kFooVersionString may now be used instead of @"1.3" or fn CFSTR("1.3"), with exactly the same effect.

Objective-C

compile as "Objective-C"

// declarations of all kinds (these go in *.h)
BeginCDeclaration
```
   @interface FooThing : NSObject {
     NSWindow *window;
   }
   - (void)buildWindow;
   @end
   void BuildWindow( void );
```
EndC

// definitions of functions and global vars (these go in *.m)
BeginCFunction
```
   @implementation FooThing
   - (void)buildWindow {
     window = [[NSWindow alloc] initWithContentRect:NSMakeRect( 0, 0, 300, 300 )
            styleMask:NSTitledWindowMask
            backing:NSBackingStoreBuffered defer:NO];
     [window center];
     [window setTitle:@"FooThing"];
     [window makeKeyAndOrderFront:nil];
   }
   @end
   void BuildWindow( void )
   {
     FooThing *ft = [[FooThing alloc] init];
     [ft buildWindow];
     [ft release];
   }
```
EndC

// declare symbols for FBtoC
```
toolbox BuildWindow
toolbox fn NSApplicationLoad = Boolean
```

// main
```
fn NSApplicationLoad()
BuildWindow()
RunApplicationEventLoop()
```

See Also:
BeginCCode; BeginCDeclaration; EndC; #if

bin$ **function**

Syntax:
*binString$* = **bin$**(*expr*)

Description:
This function returns a string of zeros and ones representing the binary value of *expr*, in twos-complement integer format (the native format in which integers are stored). If `defstr byte` is in effect, an 8-character string will be returned. If `defstr word` is in effect, a 16-character string will be returned. If `defstr long` is in effect, a 32-character string will be returned.

Example:
The chart below shows the results of **bin$** on some integer values. (If a non-integer *expr* is used, **bin$** converts it to an integer before generating the string.) The chart assumes that `defstr word` is in effect.

expr	**bin$** (*expr*)
1	0000000000000001
-1	1111111111111111
256	0000000100000000
-256	1111111100000000

To convert a string of binary digits into an integer, use the following technique:

```
intVar = val&("&X" + binaryString$)
```

`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. Byte variables can handle a `binaryString$` up to 8 characters in length; short-integer variables can handle a `binaryString$` up to 16 characters in length; long-integer variable can handle a `binaryString$` up to 32 characters in length.

See Also:
hex$; oct$; uns$; defstr byte/word/long; Appendix C - Data Types and Data Representation

bit   **function**

Syntax:

bitValue = **bit**(*bitPos*)

Description:

This function returns an integer whose binary representation has the bit in position *bitPos* set to "1", and all other bits set to "0". Bit positions are counted from right to left: a *bitPos* value of zero corresponds to the rightmost ("least significant") bit. The maximum allowable value for *bitPos* is 31, which corresponds to the leftmost bit in a long-integer value. **bit** is useful in conjunction with "bitwise operators" like and and or, for setting and testing the values of particular bits in a quantity.

Note:

If `_bitPos` is a symbolic constant name, then you can use `_bitPos%` (note the "%") as a synonym for **bit**(`_bitPos`).

The following expression evaluates as `_zTrue` (-1) if bit "n" is set in `testValue&`:

`(testValue&` **and** `bit`(n)) != 0`

The following assignment sets bit "n" in `testValue&` to 1:

`testValue& = (testValue&` **or** `bit`(n))`

The following assignment resets bit "n" in `testValue&` to 0:

`testValue& = (testValue&` **and not** `bit`(n))`

See Also:

bin$; and; or; not; Appendix C - Data Types and Data Representation

BlockFill & LongBlockFill   **statement**

Syntax:
BlockFill (*startAddr&* , *numBytes&* , *byteValueExpr*)
LongBlockFill (*startAddr&* , *numBytes&* , *byteValueExpr*)

Description:
Fills each byte in a range of memory with the value specified in *byteValExpr*. The *startAddr&* parameter indicates the first memory address to fill, and *numBytes&* indicates the number of bytes in the range. **BlockFill** and **LongBlockFill** are identical.

See Also:
BlockMove; string$; space$

BlockMove **statement**

Syntax:
BlockMove *sourceAddr, destinationAddr, numberBytes*

Description:
Copies a range of bytes which begin at address *sourceAddr*, to the address range that begins at address *destinationAddr*. The *numberBytes* parameter specifies the number of bytes which are to be copied. The copying works correctly even if the source and destination ranges overlap.

Example:
```
dim as short src(9), dst(9) // 10-element arrays
// Copy one array to the other:
BlockMove @src(0), @dst(0), 10*sizeof( short )
```

See Also:
let; varptr

box statement

Syntax:
box [**fill**] [h, v] **to** h1, v1 [**to** h2, v2 ...]

Description:
Draws a rectangle which has diagonally opposite corners at coordinates (*h,v*) and (*h1,v1*). The rectangle's frame is drawn using the current pen size, mode, pattern and color for the current window or printer. If the **fill** parameter is specified, then the rectangle is filled with the current window pattern.
If the **to** keyword is specified more than once, then several rectangles are drawn, each using (*h,v*) as one corner, and the coordinates following **to** as the diagonally opposite corner.
If the *h* and *v* parameters are omitted, then the rectangle's originating point is set to one of the following:

- The (*h,v*) coordinates of the most recent **box** statement (in any window) that actually specified the *h* and *v* parameters;
- The last point specified in the most recent plot statement (in any window);
- (0,0), if no plot statement has yet been executed, and no prior box statement that specified *h* and *v* has yet been executed.

See Also:
circle; plot; pen

button close   **statement**

Syntax:
button close *btnID*

Description:
The button, scroll bar or other control specified by *btnID* is removed from the current output window. This is one of two ways you can dispose of a button: the other way is to close the window, which automatically closes all the controls in it.

Example:
button close 1
button close _myRadioBtn2
button close btnNum

See Also:
button; scroll button; appearance button

button   **function**

Syntax:

buttonState = button(*btnNum* [, *selector*])

Description:

Returns 0 (_grayBtn) if the button is inactive (for example in a non-front window), otherwise returns the button's 32-bit signed value.

The extended button function offers access to many of the features found in appearance buttons. Use the selectors in conjunction with the button reference number to obtain information. For example, to get the minimum value for button number 10, the code would be:

min = button(10, _FBGetCtlMinimum)

If the **button** statement is called and the *btnNum* parameter refers to a button that does not exist, you will see the message, "Button() called for non-existant button." If an improper selector is used, you will see the message, "Bad parameter for Button()." The following table lists possible values for the button statement's selector.

Selector	Description
< zero	Get the reference number of the nth embedded sub control. The absolute value of this is used as the index. Example: subControlRef = button(_mySuperControl,-3) : rem get 3rd embedded control's ref num
_FBGetCtlRawValue	The control's 32-bit signed value.
_FBGetCtlMinimum	Minimum value allowed for this control.
_FBGetCtlMaximum	Maximum value allowed for this control.
_FBGetCtlPage	Page up/down value for a scroll bar.
_FBGetRootControl	Root control of the window.
_FBcountSubControls	Count how many controls are embedded in this super control.
_FBGetSuperControl	Get (parent) super control's ref num.
_FBGetControlDate	Fills the global Pascal string gFBControlText$ with the text version of the control's date. It also fills the global record gFBControlSeconds with the control's date/time record. See "Time and Date Buttons" under the appearance button statement.
_FBgetControlTime	Fills the global Pascal string gFBControlText$ with the text version of the control's time. It also fills the global record gFBControlSeconds with the control's date/dime record. See "Time and Date Buttons" under the appearance button statement.
_FBgetBevelControlMenuHandle	Gets the menu handle of a beveled pop-up menu button.
_FBgetBevelControlMenuVal	Gets the current selection of a beveled pop-up menu button.
_FBgetBevelControlLastMenu	Gets the previous selection of a beveled pop-up menu button.
_FBgetControlMenuHandle	Gets the menu handle of a standard pop-up menu button.
_FBgetControlMenuID	Gets the current selection of a standard pop-up menu button.

See Also:

button; scroll button

button statement

Syntax:
To create a button:
button *btnNum, state, title, btnRect* [, *btnType*]
To alter a button:
button *btnNum* [,[*state*][,[*title*][, *btnRect*]]]
To hide a button:
button *-btnNum*

Description:
A simplified form of the appearance button statement. The **button** statement puts a new button in the current output window, or alters an existing button's characteristics. After you create a button using the **button** statement, you can use the dialog function to determine whether the user has clicked it. You can use the button close statement if you want to dispose of the button without closing the window. You may hide an existing button using the **button** statement with a negative button reference number. You can deactivate the control with either:
button 1, _grayBtn
or ...
appearance button 1, _grayBtn
When you first create a button with a given ID (in a given window), you must specify all the parameters up to and including *btnRect* (the *btnType* parameter is optional; it defaults to _push). If you later want to modify that button's characteristics, execute **button** again with the same ID, and specify one or more of the other parameters (except *btnType*, which cannot be altered). The button will be redrawn using the new characteristics that you specified; any parameter that you don't specify will not be altered.

Parameter	Description
btnNum	An ID number that you assign when you create the button and that you refer to when altering or closing the button. The number you assign must be different from the btnNum of all other existing buttons, scroll bars and edit fields in the current window.
state	Sets the state of the button. See button function for details.
title$	The text that appears inside the button (in the case of push buttons) or to the right of the button (in the case of checkboxes and radio buttons).
btnRect	The button's enclosing rectangle. This can be specified in either of two ways: (x1,y1)-(x2,y2) Coordinates of two diagonally opposite points @rectAddr& Address of an 8 byte rectangle structure.
btnType	Specifies the type of button: _push (1) push button (default type) _checkBox (2) check box _radio (3) radio button _shadow (4) framed push button If you add the constant _useWFont to any of the above types (except _shadow) the button's title will be drawn using the window's current font ID, font size, and font style. Any subsequent change you make to the window fontID, font size, or font style will be reflected in the button's title when it is redrawn. If you don't specify _useWFont (or if the button type is _shadow) the title is drawn using the system font.

See Also:
appearance button; scroll button

button& **function**

Syntax:
controlRef = **button&**(*btnNum*)

Description:
For the button, scroll button or edit field specified by *btnNum* in the current output window, this function returns an opaque reference suitable for passing to Toolbox functions.

Example:
dim as ControlRef c
c = **button&**(1)
HideControl(c)

See Also:
button

ButtonTextString$ function

Syntax:
*pString$* = **fn ButtonTextString$**(*btnNum*)

Description:
Extracts the text from a text control or edit field.

```
appearance button 1,,,,,,(10,10)-(200,200), _kControlEditTextProc
// put text into edit field button
def SetButtonTextString( 1, "Hello" )
// extract text from edit field button
pString$ = fn ButtonTextString$(1)
// pString$ now equals "Hello"
```

See Also:
appearance button

call <toolbox>   **statement**

Syntax:
[**call**] *ToolboxProcName*[(arg1, arg2...)]

Description:
This statement calls a Carbon Toolbox procedure. A Toolbox procedure (as opposed to a Toolbox function) performs an operation without returning a value. To execute a Toolbox function, use the fn statement instead.
ToolBoxProcName must be previously defined in a toolbox statement.
If the procedure requires parameters, include them in a list surrounded by parentheses. (If the procedure does not take any parameters, then the parentheses are optional.)

Example:
dim as Rect myRect
SetRect(myRect, 10, 10, 200, 150)
or
call SetRect(myRect, 10, 10, 200, 150)
FutureBasic reserves memory locations 8234650 through 8234657 (for example) for the "myRect" structure. When **SetRect** is executed, FutureBasic passes 8234650 to the Toolbox as the first parameter.

See Also:
fn <toolbox>

See the select case statement.

CFIndexSort   **function**

Syntax:
fn CFIndexSort (*whichIndex*)

Description:
The parameter that you pass to the array is the index number. Pass zero if you are using only one index$ array.

Items in the array are sorted in ascending order (from least to greatest).

Note:
This works only if you have included CFIndex.incl.

See Also:
clear <index>; index$ D; index$ I; index$ function; index$ statement; indexf; mem

---

chr$ **function**

Syntax:
*character$* = chr$(*expr*)

Description:
Returns a 1-character string consisting of the character whose ASCII code is given by *expr* mod 256.

Example:
```
print chr$( 65 )chr$( 66 )chr$( 67 )
```
program output:
ABC

See Also:
asc; string$; Appendix F - ASCII Character Codes

circle **statement**

Syntax:
circle [**fill**] *x , y , radius* [**to|plot** *startAngle , angleSize*]

Description:
Draws a circle, an arc or a wedge in the current foreground color, pen pattern and pen size. If a circle or wedge is drawn using the **fill** keyword, the circle or wedge will be filled with the current pen pattern.

If only the *x*, *y*, and *radius* parameters are specified, then a complete circle is drawn, with its center at point (*x , y*) and having a radius of *radius* pixels.
If the **to** keyword is specified, then a wedge (two radii plus an arc) is drawn. The first radius is drawn in the orientation specified by *startAngle*, which is measured in units of "brads" (see below). Angles are measured counterclockwise starting from the "3-o'clock" position, which corresponds to zero brads. The *angleSize* parameter specifies the angular width of the wedge (also in brads); the wedge always extends counterclockwise from the *startAngle* position. Note that the width of the "wedge" may be greater than a half-circle, in which case the "wedge" looks more like a Pac-Man.
If the **plot** keyword is specified, then an arc is drawn without any radii. The position and size of the arc are the same as when the **to** keyword is specified. If both the **plot** keyword and the **fill** keyword are specified, then the **circle** command does nothing.
"Brads" are an angular unit in which a full circle corresponds to 256 brads. A brad is therefore slightly larger than a degree (to be exact, it's 360/256 of a degree). A half circle therefore equals 128 brads, and a right angle equals 64 brads.
Note:
If you use values outside the range 0..255 for *startAngle* and/or *angleSize*, then values modulo 256 will be used.

See Also:
fill

clear <index>   **statement**

Syntax:
Release the memory used by existing `index` arrays:
1. **clear** `-1`
2. **clear index$** [*indexID*]

Description:
Syntax (1) releases the memory occupied by all existing `index$` arrays. Syntax (2) releases memory occupied by the *indexID* array. If you omit the *indexID* parameter, then `index$` array #0 is used.

See Also:
index$ statement; index$ D; index$ I; indexf; mem

See the **local** statement

clear lprint   **statement**

Syntax:
clear lprint

Description:
If output has been routed to the printer, **clear lprint** forces the Printing Manager to print the current page, without closing the print job. The other ways to print the current page are to execute `close lprint` or to exit the program, both of which also cause the print job to be closed.

Note:
Before using **clear lprint** or `close lprint`, you must route printing to the screen. The following fragment shows how this might be done.

route `_toScreen`
clear lprint
route `_toPrinter`

See Also:
page statement; close lprint; def lprint; route _toPrinter; route _toScreen

close **statement**

Syntax:
close `[[#]`*fileNum* `[,[#]`*deviceID2* `...]]`

Description:
Closes one or more files or devices (usually a serial port) previously opened with the `open` statement. If no file or deviceID is specified, all open files and devices are closed.
Closing a file forces any remaining bytes in the output buffer to be written to disk, and allows you to re-use the number specified in *fileNum* or *deviceID* (for a subsequent `open` statement).

See Also:
open

close lprint   **statement**

Syntax:
close lprint

Description:
After output has been routed to the printer, **close lprint** informs the Print Manager that the print job is complete. The current page is printed and the print job closed. You should execute a route _toScreen statement immediately before or after calling **close lprint**.

Note:
Before using clear lprint or **close lprint**, you must route printing to the screen. Closing the printer while output is pointing to the printed page is equivelant to cutting off a limb while you are sitting on it. The following fragment shows how this might be done.

route _toScreen
close lprint

See Also:
clear lprint; def lprint; route _toPrinter; ROUTE_toScreen

cls   **statement**

Syntax:
cls 〔 **line** | **page** 〕

Description:
cls resets the current window's clipping region to encompass the entire window, clears the entire contents of the window to the background pattern and color (usually white), and resets the pen to a position near the upper-left corner (0, 0) of the window.
cls line clears a rectangle as high as the current font, from the current pen position to the right side of the window. This is handy for clearing text from the current line only. The pen position is not affected.
cls page clears the text from the pen position to the right side of the window (as **cls line** does), then clears the entire window below the pen position. The pen position is not affected.

color   **statement**

Syntax:
color [=] *colorExpr*

Description:
The **color** statement sets the current window's foreground color to one of eight old-style "Basic QuickDraw" colors. *colorExpr* should equal one of the following:

```
0 (_zWhite)
1 (_zYellow)
2 (_zGreen)
3 (_zCyan)
4 (_zBlue)
5 (_zMagenta)
6 (_zRed)
7 (_zBlack)
```

Note:
The **color** statement does not change the appearance of anything that's already in the window; the new color will appear the next time you draw text or a QuickDraw shape in the window.

For greater control over the foreground color, use the `long color` statement.

See Also:
long color

compile   **statement**

Syntax:
compile [as] "Objective-C"

Description:
Instructs the compiler to compile the translated source code using the Objective-C compiler instead of the standard compiler. The Objective-C compiler is essential if the source code contains Cocoa or Objective-C statements.

compile shutdown   **statement**

Syntax:
compile shutdown *"optional string literal"*

Description:
You may have code designed for a specific circumstance known at compile time. If the circumstance is not present, **compile shutdown** would abort the compile process with an error and show the offending line complete with the optional quoted remark.

Example
#if ndef _FBtoC
  compile shutdown "This requires FBtoC"
#endif

CompilerVersion   **function**

Syntax:
versionNum = **CompilerVersion**

Description:
This function returns the current build number of FBtoC. The number is internally set before each FutureBasic package is shipped. The purpose of this function is to determine whether or not a specific functionality is present before a particular command is executed.
If, for example, you wanted to invoke a routine that was not added to FBtoC until build 143, you would check the version as follows:

```
long if ( CompilerVersion < 143 )
 print "You need a newer version of FBtoC to run this."
xelse
 // new operation goes here.
end if
```

compound   **function**

Syntax:
compoundFactor = **compound**(*rate*, *periods*)

Description:
Returns the compounding factor for the given interest rate and number of periods. The parameters *rate* and *periods* are double precision variables, and the returned value *compoundFactor* is also a double precision value. The interest rate should be expressed as a fraction of 1; for example, 5.2 percent should be expressed as 0.052.

Note:
compound uses the following formula:
compoundFactor = $(1 + rate)^{periods}$

See Also:
annuity

Constant declaration   **statement**

Syntax:
_constantName = *staticExpr*

Description:
This is a non-executable statement which assigns the value of *staticExpr* to the symbolic constant indicated by _constantName. The _constantName should indicate a name which was not defined anywhere previously in the program, and which is different from all pre-defined FutureBasic symbolic constant names. The name must be preceded by an underscore '_' character. The *staticExpr* must be a "static integer expression"; it cannot contain variables or function references.

An error occurs if you attempt to assign a different value to an existing constant name, without preceding the new assignment by override. It is not an error to reassign the same value to an existing constant name.

Another way to assign values to symbolic constant names is with the begin enum statement.

See Also:
override; begin enum; Appendix C - Data Types and Data Representation

cos   **function**

Syntax:

result = **cos**(*expr*)

Description:

Returns the cosine of *expr*, where *expr* is given in radians. The returned value will be in the range -1 to +1. **cos** returns a double-precision result.

Note:

To find the cosine of an angle degAngle which is given in degrees, use the following:

```
result = cos( degAngle * pi / 180.0 )
```

where pi is 3.141592654...

See Also:

acos; sin; tan

cosh

Syntax:
result = **cosh** (*expr*)

Description:
Returns the hyperbolic cosine of *expr*.
cosh returns a double-precision result.

Note:

cosh (*x*) is: $\dfrac{e^{x} + e^{-x}}{2}$

See Also:
acosh; sinh; tanh; exp

csrlin   **function**

Syntax:
currentLine = **csrlin**

Description:
This function returns the number of the "current" text line (that is, the text line which contains the current pen position) for the current window. The text line at the top of the window is considered Line #0.
csrlin does not necessarily reflect the number of text lines which have actually been displayed. It is calculated based on the current pen position and the size of the current font.

Example:
```
dim as long i
window 1
text _monaco, 16
cls
for i = 1 to 10
 print "csrlin = "; csrlin
next
```

See Also:
pos

cursor **statement**

Syntax 1:
cursor *cursorID*

Description:
cursorID is interpreted as the resource ID of a 'crsr' (first choice) or a 'CURS' resource (alternative choice), and this statement changes the current cursor to the indicated cursor. The following cursor resources are always available:
```
_arrowCursor (0)
_iBeamCursor (1)
_crossCursor (2)
_plusCursor  (3)
_watchcursor (4)
```

Note:
If you've designed a csrs or a CURS resource of your own which you want to make available to your program, do the following:

1. Assign a positive resource ID number, 128 or higher, to the resource.
2. Copy the resource to the resources file which you reference in your program's resources statement.

You can then use the **cursor** statement to activate your cursor within the program.

Syntax 2:
cursor *cursorID, _themeCursorStatic*
cursor *cursorID, _themeCursorAnimate* // treated as _themeCursorStatic

Here, cursorID is one of:
```
_kThemeArrowCursor
_kThemeCopyArrowCursor
_kThemeAliasArrowCursor
_kThemeContextualMenuArrowCursor
_kThemeIBeamCursor
_kThemeCrossCursor
_kThemePlusCursor
_kThemeWatchCursor
_kThemeClosedHandCursor
_kThemeOpenHandCursor
_kThemePointingHandCursor
_kThemeCountingUpHandCursor
_kThemeCountingDownHandCursor
_kThemeCountingUpAndDownHandCursor
_kThemeSpinningCursor
_kThemeResizeLeftCursor
_kThemeResizeRightCursor
_kThemeResizeLeftRightCursor
```

cvi   **function**

Syntax:

integerTypeVar = **cvi**(*stringVar*)

Description:

This function converts the bytes in *stringVar* into an integer value which has the same internal bit-pattern that *stringVar* has. If *stringVar* consists of 4 or more bytes, only its first 4 bytes are considered. If *stringVar* consists of 1, 2 or 3 bytes, then **cvi**(*stringVar*) returns an 8-bit, 16-bit or 24-bit integer, respectively. If *stringVar* is a null string, then **cvi**(*stringVar*) returns zero.

This function is useful for finding the integer form of such things as file types, creator signatures and resource types. For example:

```
typeString = "text"
theType = cvi( typeString )
```

After executing the above, `theType` is then suitable for passing to a Toolbox routine which requires an OSType parameter. `theType` will also have the same value as the integer constant `_"text"`.

The size (in bytes) of the value returned by **cvi** depends on the length of *stringVar*. It does not depend on the current setting of `defstr byte/word/long`. Therefore, if you want to assign the return value of **cvi** to a short integer variable, you must make sure that *stringVar* is not longer than 2 bytes; otherwise, you'll get an unexpected value in your short integer variable. Similarly, if you want to correctly assign **cvi**'s return value to a byte variable, you should make sure that *stringVar* is not longer than 1 byte.

The `mki$` function is the inverse of **cvi**. Note, however, that the output of `mki$` does depend on the current setting of `defstr byte/word/long`.

Note:

If *stringVar* is 1 byte long, then **cvi**(*stringVar*) returns the same value as `asc`(*stringVar*).

See Also:

asc; mki$; defstr byte/word/long

data   **statement**

Syntax:
data *item1* [, *item2* ...]

Description:
This statement is used to list data constants (numbers or strings, quoted or unquoted) to be accessed by the read statement. Each item must be a numeric or string constant; string constants may be quoted or unquoted. The items are separated by commas. Leading spaces (between a comma and the item that follows it, or between the **data** keyword and the first item) are ignored; therefore, if you wish to represent a string item which contains commas and/or leading spaces, you must enclose it in quotes. Trailing spaces in an unquoted string item are not ignored; they're considered part of the string.

To represent a numeric item in a **data** statement, you can use a decimal, hex, octal or binary constant.

You can have as many **data** statements in your program as you like, and as many or as few items in each **data** statement as you like. The only restriction is that the total number of data items (in all **data** statements) must be at least enough to satisfy all read requests. You can use the restore statement to allow **data** items to be read more than once.

data statements are global in scope: this means that any read statement (whether it's in a local function, or in "main") can access any **data** statement (whether it's in a (possibly different) local function, or in "main"). **data** statements are "non-executable," which means you can't change their effect by putting them inside a conditional execution structure such as long if...end if. However, you can conditionally include or exclude them from the program by putting them inside a compile long if block.

Note that everything between the **data** keyword and the end of the line is considered part of the **data** statement. In particular, this means that you cannot use the ":" separator to put another statement after the **data** statement on the same line, and you cannot put a comment after the **data** statement on the same line.

See Also:
read; restore

date$ **function**

Syntax:
**date$** [(formatString$)]

Description:
**date$** returns a string based on the current date. It may be used both with and without a formatString$.

Using **date$** *without* a formatString$ returns the current date as a string containing two digit numerals each for month, day and year separated by slash marks, specifically in "MM/DD/YY" format.

Using **date$** *with* a formatString$ returns a string based on the current date and formatted based on formatString$.

The formatString$ must contain Unicode Date and Time symbols as shown below and in Appendix I - Data & Time Symbols.

Example:

print date$
print date$("EEE, MMM d, yyyy")
print date$("MMMM d, yyyy")
print "This is day ";**date$**("D");" of the year"

01/14/10
Thu, Jan 14, 2010
January 14, 2010
This is day 14 of the year

More Date & Time symbols can be found in Appendix I - Data & Time Symbols.

See Also:
time$; Appendix I - Data & Time Symbols

dec   **statement**

Syntax:
dec (*intVar*)
numericVariable – –
numericVariable – = *valueToSubtract*

Description:
This statement decrements *intVar* by 1 (or by *valueToSubtract*); that is, it subtracts 1 from the value of *intVar*, and stores the value back into *intVar*. *intVar* must be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. If *intVar* is already at the minimum value for its variable type, then **dec** (*intVar*) will cycle it back to its maximum value. As of Release 3, FutureBasic supports the use of -= to decrease the value of a variable by a specified amount.

Example:
dec (x&)
and...
x& – = 1
and...
x& – –
...are all equivalent to:
x& = x& – 1
The following expressions are also equivalent:
x& = x& – 100
x& – = 100

Note:
The – = syntax may not be used for arrays of strings, containers, or records where arrays are involved, only numeric values may take advantage of this syntax.

See Also:
inc; dec long/word/byte

dec long/word/byte   **statement**

Syntax:
dec {**long** | **word** | **byte**} (*addr&*)

Description:
This statement decrements the long integer, short integer or byte which begins at the specified address in memory; that is, it subtracts 1 from the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its minimum possible value, then the statement will cycle it back to its maximum value.

Example:
dec long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) - 1
Also:
dec word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) - 1

See Also:
dec; inc; inc long/word/byte

dec long/word/byte statement

Syntax:
dec { **long** | **word** | **byte** } (*addr&*)

Description:
This statement decrements the long integer, short integer or byte which begins at the specified address in memory; that is, it subtracts 1 from the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its minimum possible value, then the statement will cycle it back to its maximum value.

Example:
dec long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) - 1
Also:
dec word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) - 1

See Also:
dec; inc; inc long/word/byte

dec long/word/byte **statement**

Syntax:
dec {**long** | **word** | **byte**} (*addr&*)

Description:
This statement decrements the long integer, short integer or byte which begins at the specified address in memory; that is, it subtracts 1 from the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its minimum possible value, then the statement will cycle it back to its maximum value.

Example:
dec long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) - 1
Also:
dec word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) - 1

See Also:
dec; inc; inc long/word/byte

def fn <expr>   **statement**

Syntax:
def fn functionName [(var1 [,var2 ...])] [**as type**] = expr

Description:
This statement defines a "one-line" function. You can refer to the function in later parts of your program by using an expression in this form:

def fn functionName [(parm1 [,parm2 ...])]

This expression returns the value of expr from the function definition.

The Def Fn <expr> statement should not appear inside any Local function.

functionName can be any valid FB identifier which is different from any other function name defined in your program. functionName can optionally end with a type-identifier (such as "as double", etc.) to indicate the data type that the function returns (and hence expr should be of the same type). If none is specified, the function returns a long-integer value.

You can optionally include a formal parameter list in the function definition: this is a list of variable names (var1, var2, etc.) separated by commas and enclosed in parentheses, which immediately follows functionName. Usually, expr will contain references to these parameter variables. When you call a function that has a formal parameter list, you pass values to it in an actual parameter list (parm1, parm2, etc.). These values are then assigned to var1, var2, etc., and are used in evaluating expr.

var1, var2, etc. must be "simple" variables: they cannot be array elements, records, nor record fields. parm1, parm2, etc. can (with some exceptions) be any kinds of expressions, as long as the data type of each parm expression is compatible with its corresponding var variable in the formal parameter list. The number and order of the items in the actual parameter list must exactly match the number and order of the items in the formal parameter list (if any).

The variables in the formal parameter list are either global (if they were previously declared within a Begin Globals...End Globals section), or they are "local to main." In either case, this means that the values which get assigned to those variables (when you call the function) persist even after the function returns its value. You need to keep this in mind if you later execute some statement in "main" (outside of all Local functions) which contains one of those variables.

expr may contain other variables besides those which appear in the formal parameter list. All variables in expr are either global (as declared within a Begin Globals...End Globals block) or are local to main.

Example:
def fn Area(r **as single**) **as single** = pi * r * r
...
local fn Circle6
  a = **fn** Area(6.0)
  print a
end fn

The function Fn Area calculates the area of a circle, when the radius of the circle is passed as a parameter. We are assuming that the variable pi is a global variable or a "local to main" variable whose value has previously been set to 3.14159... as required.

When we call the Circle6 function, the value 113.079 gets assigned to the local variable a. As a side effect of calling Fn Area(6.0), the value of the "local to main" variable r is changed to 6.0.

Note:
Def Fn <expr> is a "non-executable" statement, which means you cannot affect the definition of the function by placing Def Fn <expr> after Then or Else (in an If statement), nor by placing it inside any kind of "conditional execution" block such as Long If...End If, While...Wend, For...Next, etc. However, you can affect the function definition (at compile time) by placing Def Fn <expr> inside a Compile Long If block.

Def Fn does not work for threaded functions.

See Also:

local fn; def fn <prototype>

def fn <prototype>   **statement**

Syntax:
def fn functionName [(*var1* [, *var2* ...])]

Description:
This statement declares a prototype for a local fn. The functionName and the argument list (*var1*, *var2* etc.) must match those of
some local fn whose definition appears later in the source code stream.
A local fn must either be defined (using a local fn...end fn block) or prototyped (using **def fn** <prototype>) before it can be
referenced in any fn <userFunction> statement. By prototyping the function early in the code, you can call fn <userFunction>
above the spot where the local fn...end fn block actually appears. This frees you from concerns about how to order your local fn
blocks in the code.

Example:
In the following excerpt, fn myFn1 calls fn myFn2, and fn myFn2 calls fn myFn1. This kind of construction would be difficult to
implement without prototyping the functions.
def fn myFn1(x **as long**)
def fn myFn2(x **as long**)
do
  input "Enter a number", k
    print fn myFn1(k)
until k = 0
local fn myFn1(x **as long**)
  if x **mod** 1 **then** z = 3 * x **else** z = **fn** myFn2(x) + 6
end fn = z
local fn myFn2(x **as long**)
  if x **mod** 1 **then** z = **fn** myFn1(x) – 1 **else** z = x / 2
end fn = z

See Also:
fn <userFunction>

def fn using <address>   **statement**

Syntax:
def fn FunctionName [(*var1* [,*var2* ...])] **using** *fnAddress*

Description:
This statement associates the name FunctionName with the routine which is located at the address given by *fnAddress*. You can refer to FunctionName in later parts of your program by using an expression in this form:
  fn FunctionName [(parm1 [, parm2 ...])]
This expression will call the function referenced by *fnAddress*, and will return the value (if any) that the referenced function returns.
fnAddress must be a pointer variable. Before you can call fn FunctionName, you must make sure that the *fnAddress* variable contains the address of a local fn or a def fn <expr>, as returned by the @fn function. You must not use the address of a label location (as returned by the line function or the proc function).
If the name of the function referenced by *fnAddress* ends with a type-identifier suffix, then functionName must end with the same type-identifier suffix.
If the function referenced by *fnAddress* has a parameter list, then you must include a parameter list (*var1* [, *var2* ...]) in the **def fn using** <fn address> statement. The number, order, and data types of the parameters in the **def fn using** <fn address> list must match the number, order, and data types in the parameter list of the referenced function.
The **def fn using** <fn address> statement is useful in cases where your program must decide at runtime which of several similar functions should be executed in a given instance.

Note:
def fn using <fn address> is a "non-executable" statement, which means you cannot change its effect by placing it after then or else (in an if statement), nor by placing it inside any kind of "conditional execution" block such as long if...end if, while...wend, for...next, etc. However, you can conditionally include or exclude it by placing it inside a #if / #endif block.
It is possible to choose at run time which function FunctionName shall be associated with, and even to dynamically change that association from one function to another. This is done by dynamically setting the **fnAddress** variable to the addresses of different functions at run time.

See Also:
local fn; end fn; @fn; def fn <prototype>

def lprint   **statement**

Syntax:
def lprint

Description:
This statement initializes the printer driver and displays the printer "Job Dialog." The exact appearance of the Job Dialog depends on which printer is currently selected, but it typically prompts the user to enter a range of page numbers to be printed, along with other information. Your program should execute the **def lprint** statement when the user clicks a "Print..." button or selects a "Print..." menu item in your program.

Note:
After your program executes **def lprint**, you can use the `prCancel` function to determine whether the user cancelled the print job. If the user did not cancel, then you can use the `prHandle` function to determine the page range and the number of copies that the user requested.

See Also:
clear lprint; close lprint; lprint; prHandle

def open   **statement**

Syntax:
def open `[=]` *typeCreatorString$*

Description:
Use this statement to specify the file type and/or creator signature to be assigned to files which are subsequently opened for output by the **Open** statement. *typeCreatorString$* should be a string expression whose length is either 4 or 8 characters. The first 4 characters will be used as the file type for newly opened files; the second group of 4 characters (if any) will be used as the creator signature for newly opened files. The specified *typeCreatorString$* remains in effect until another **Def Open** statement is executed.

A Macintosh file may have 4-character file type and a 4-character creator signature (See Note 2). The file type is usually used to signify the general format of the file's contents. If you're creating a file which is to be opened by another application, you should use a file type which the other application recognizes. Otherwise, you can make up a custom file type for your program's private use. By default (i.e., if your program hasn't yet executed a **Def Open** statement), FutureBasic assigns the file type "????" to files opened for output.

A file's creator signature usually signifies which application created the file. The Finder uses the file's creator signature to determine such things as: which application to launch when the user double-clicks the file's icon. The Finder uses the file's creator signature, in combination with the file's type, to determine the icon to display for the file. By default, if your program hasn't yet executed a **Def Open** statement with an 8-character *typeCreatorString$* parameter, FutureBasic assigns the creator signature "????" to files opened for output.

Example:
Def Open `"ttrottxt"`
Open `"O", #1, "test file",@parentFolderRef // an FSRef cannot be created for a non-existent file, so a filename is supplied`
Print `#1, "This is a test file"`
Close `#1`

The program above creates a file whose type is "ttro" and whose creator signature is "ttxt". The Finder recognizes such a file as a SimpleText "read-only" text file. The file will appear with an appropriate icon, and the Finder will launch SimpleText when the user double-clicks the file's icon.

Note:
(1) **Def Open** only applies to files which are opened with the "O" option. It does not affect the type nor creator signature of files which are opened for input only ("I") or for random access ("R").
(2) Apple has recommended use of Uniform Type Identifiers to replace Type/Creator. See Apple's "Introduction to Uniform Type Identifiers" for more information.

See Also:
files$; open

def page **statement**

Syntax:
def page

Description:
This statement displays the printer "Style Dialog." The exact appearance of the Style Dialog depends on which printer is currently selected, but it typically prompts the user to select portrait or landscape mode, paper type, and other information. Your program should execute the **def page** statement when the user selects a "Page Setup..." menu item.

Note:
After your program executes **def page**, you can use the `prCancel` function to determine whether the user clicked the Style Dialog's "Cancel" button, and you can use the `prHandle` function to determine the page setup preferences that the user requested.

See Also:
clear lprint; close lprint; lprint; def lprint; prHandle

def tab   **statement**

Syntax:
def tab [=] *fieldWidth*

Description:
This statement sets the default print field width to *fieldWidth* characters. This affects the number of space characters which are output when a comma is encountered in subsequent **Print**, **Print#** and **lprint** statements. When an item in such a statement follows a comma, FutureBasic outputs enough space characters so that each new item begins in a new print field which is *fieldWidth* characters wide. *fieldWidth* must be between 1 and 255. When the program starts, the default print field width is 16 characters.

The value of the print field width affects comma-delimited items whether they're sent to a disk file or to a display device.

See Also:
width

def using   **statement**

Syntax:
def using `[ = ]` *intlMoneyFormat&*

Description:
This statement specifies how the **Using** function should interpret certain characters in its pattern string; this affects the format of the string returned by **Using**. The *intlMoneyFormat&* is specified by four characters packed into a long integer. The characters represent: the decimal point, the thousands separator, the list separator, and the currency symbol. After you execute **Def Using**, subsequent numbers formatted with **Using** are formatted as follows:

- A leading dollar sign in the pattern string is replaced with the currency symbol.
- Commas in the pattern string are replaced with the thousands separator.
- A period in the pattern string is replaced with the decimal point.

Example:
The following selects the U.S. numeric format (this is also the default if no **Def Using** statement has been executed):
Def Using `= _".,;$"`
The following selects the international numeric format as set by the system or selected by the user (this is the recommended numeric format to use):
Def Using `= [[Fn GetIntlResource(0)]]`

Note:
The international numeric format returned by **Fn GetIntlResource** may include a currency symbol which consists of more than one character. **Def Using** will pick up only the first character of the currency symbol.

Fn GetIntlResource is deprecated by Apple as of the introduction of MacOS X 10.5. The current recommended method to obtain items such as the currency symbol or decimal separator is to use **Fn CFLocaleGetValue** with an appropriate key such kCFLocaleDecimalSeparator or kCFLocaleCurrencySymbol.
See Also:
using

def<type>   **statement**

Syntax:
defsng   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defdbl   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defstr   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defint   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
deflong *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]

Description:
A **def<type>** statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The **def<type>** statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared that means you can override the effect of a **def<type>** statement by putting a type-identifier suffix (like "`%`" or "`&`") at the end of a name, or by declaring the name in a `dim` statement using an `as` clause.
Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A **def<type>** statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the letter parameter it's paired with.)
def<type> statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use **def<type>** statements within your program, they should appear near the top of your source code. In particular, if a **def<type>** statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.
def<type> statements are "non-executable," which implies that they should not appear within any kind of "conditional execution" block, such as `for...next`, `long if...ENDIF`, `do...until`, etc. (but they may be conditionally included or excluded if you put them inside a `compile long if` block).
The following table indicates the default type applied by each of the `def<type>` statements.

statement	default type	equivalent suffix
defsng	single precision	!
defdbl	double precision	#
defstr	string	$
defint	integer	%
deflong	long integer	&

See Also:
defstr long/word/byte; Appendix C - Data Types and Data Representation

def<type>   **statement**

Syntax:
defsng   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*]...]
defdbl   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*]...]
defstr   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*]...]
defint   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*]...]
deflong *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*]...]

Description:
A **def<type>** statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The **def<type>** statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared that means you can override the effect of a **def<type>** statement by putting a type-identifier suffix (like "`%`" or "`&`") at the end of a name, or by declaring the name in a `dim` statement using an `as` clause.
Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A **def<type>** statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the letter parameter it's paired with.)
def<type> statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use **def<type>** statements within your program, they should appear near the top of your source code. In particular, if a **def<type>** statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.
def<type> statements are "non-executable," which implies that they should not appear within any kind of "conditional execution" block, such as `for...next`, `long if...ENDIF`, `do...until`, etc. (but they may be conditionally included or excluded if you put them inside a `compile long if` block).
The following table indicates the default type applied by each of the `def<type>` statements.

statement	default type	equivalent suffix
defsng	single precision	!
defdbl	double precision	#
defstr	string	$
defint	integer	%
deflong	long integer	&

See Also:
defstr long/word/byte; Appendix C - Data Types and Data Representation

def<type> statement

Syntax:
defsng   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defdbl   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defstr   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defint   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
deflong *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]

Description:
A **def<type>** statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The **def<type>** statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared that means you can override the effect of a **def<type>** statement by putting a type-identifier suffix (like "`%`" or "`&`") at the end of a name, or by declaring the name in a `dim` statement using an `as` clause.
Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A **def<type>** statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the letter parameter it's paired with.)
def<type> statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use **def<type>** statements within your program, they should appear near the top of your source code. In particular, if a **def<type>** statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.
def<type> statements are "non-executable," which implies that they should not appear within any kind of "conditional execution" block, such as `for...next`, `long if...ENDIF`, `do...until`, etc. (but they may be conditionally included or excluded if you put them inside a `compile long if` block).
The following table indicates the default type applied by each of the `def<type>` statements.

statement	default type	equivalent suffix
defsng	single precision	!
defdbl	double precision	#
defstr	string	$
defint	integer	%
deflong	long integer	&

See Also:
defstr long/word/byte; Appendix C - Data Types and Data Representation

def<type> statement

Syntax:
defsng *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defdbl *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defstr *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defint *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
deflong *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]

Description:
A **def<type>** statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The **def<type>** statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared that means you can override the effect of a **def<type>** statement by putting a type-identifier suffix (like "`%`" or "`&`") at the end of a name, or by declaring the name in a `dim` statement using an `as` clause.
Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A **def<type>** statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the letter parameter it's paired with.)
def<type> statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use **def<type>** statements within your program, they should appear near the top of your source code. In particular, if a **def<type>** statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.
def<type> statements are "non-executable," which implies that they should not appear within any kind of "conditional execution" block, such as `for...next`, `long if...ENDIF`, `do...until`, etc. (but they may be conditionally included or excluded if you put them inside a `compile long if` block).
The following table indicates the default type applied by each of the `def<type>` statements.

statement	default type	equivalent suffix
defsng	single precision	!
defdbl	double precision	#
defstr	string	$
defint	integer	%
deflong	long integer	&

See Also:
defstr long/word/byte; Appendix C - Data Types and Data Representation

def<type>   **statement**

Syntax:
defsng   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defdbl   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defstr   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
defint   *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]
deflong *letter1* [– *thruLtr1*] [, *letter2* [– *thruLtr2*] . . .]

Description:
A **def<type>** statement specifies the data type for subsequently occurring variables, arrays and record fields whose names begin with any of a specified set of letters. The **def<type>** statement only applies to variables, arrays and record fields whose types are not otherwise explicitly declared that means you can override the effect of a **def<type>** statement by putting a type-identifier suffix (like "`%`" or "`&`") at the end of a name, or by declaring the name in a `dim` statement using an `as` clause.
Each of the *letter* and *thruLtr* parameters should be a letter of the alphabet. A **def<type>** statement applies to those simple variables, arrays and record fields whose names begin with *letter1*, *letter2*, etc. Whenever a *thruLtr* parameter is included, the statement also applies to names that start with any letter in the range between *letter* and *thruLtr* (inclusive). (For that reason, a *thruLtr* parameter should always occur later in the alphabet than the letter parameter it's paired with.)
def<type> statements are global in scope (that is, they apply to names declared both inside and outside of local functions). If you use **def<type>** statements within your program, they should appear near the top of your source code. In particular, if a **def<type>** statement appears farther down in the source than some variable, array or record field that it applies to, the results could be unpredictable.
def<type> statements are "non-executable," which implies that they should not appear within any kind of "conditional execution" block, such as `for...next`, `long if...ENDIF`, `do...until`, etc. (but they may be conditionally included or excluded if you put them inside a `compile long if` block).
The following table indicates the default type applied by each of the `def<type>` statements.

statement	default type	equivalent suffix
defsng	single precision	!
defdbl	double precision	#
defstr	string	$
defint	integer	%
deflong	long integer	&

See Also:
defstr long/word/byte; Appendix C - Data Types and Data Representation

defstr long/word/byte **statement**

Syntax:
defstr {**long** | **word** | **byte**}

Description:
These statements affect the way several integer-to-string functions format their return values. Basically, they affect whether the string functions interpret their arguments as 32-bit, 16-bit or 8-bit integers. The statements are global in scope, and apply to all subsequent calls to the affected string functions, until another **defstr** {**long** | **word** | **byte**} statement is executed. When the program starts, **defstr long** is in effect. The following table shows how the statements affect the return values of the various string functions.

	bin$	hex$	oct$	uns$	mki$
defstr long (Default)	returns 32 characters	returns 8 characters	returns 11 characters	10 characters; adds 2^{32} if arg < 0	returns 4 characters
defstr word	returns 16 characters	returns 4 characters	returns 6 characters	5 characters; adds 2^{16} if arg < 0	returns 2 characters
defstr byte	returns 8 characters	returns 2 characters	returns 3 characters	3 characters; adds 256 if arg < 0	returns 1 character

When **defstr byte** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -255 to +255. Likewise, when **defstr word** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -65535 to +65535.

See Also:
bin$; hex$; oct$; uns$; mki$

defstr long/word/byte   **statement**

Syntax:
defstr { **long** | **word** | **byte** }

Description:
These statements affect the way several integer-to-string functions format their return values. Basically, they affect whether the string functions interpret their arguments as 32-bit, 16-bit or 8-bit integers. The statements are global in scope, and apply to all subsequent calls to the affected string functions, until another **defstr** { **long** | **word** | **byte** } statement is executed. When the program starts, **defstr long** is in effect. The following table shows how the statements affect the return values of the various string functions.

	bin$	hex$	oct$	uns$	mki$
defstr long (Default)	returns 32 characters	returns 8 characters	returns 11 characters	10 characters; adds 2^{32} if arg < 0	returns 4 characters
defstr word	returns 16 characters	returns 4 characters	returns 6 characters	5 characters; adds 2^{16} if arg < 0	returns 2 characters
defstr byte	returns 8 characters	returns 2 characters	returns 3 characters	3 characters; adds 256 if arg < 0	returns 1 character

When **defstr byte** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -255 to +255. Likewise, when **defstr word** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -65535 to +65535.

See Also:
bin$; hex$; oct$; uns$; mki$

defstr long/word/byte   **statement**

Syntax:
defstr {**long** | **word** | **byte**}

Description:
These statements affect the way several integer-to-string functions format their return values. Basically, they affect whether the string functions interpret their arguments as 32-bit, 16-bit or 8-bit integers. The statements are global in scope, and apply to all subsequent calls to the affected string functions, until another **defstr** {**long** | **word** | **byte**} statement is executed. When the program starts, **defstr long** is in effect. The following table shows how the statements affect the return values of the various string functions.

	bin$	hex$	oct$	uns$	mki$
defstr long (Default)	returns 32 characters	returns 8 characters	returns 11 characters	10 characters; adds 2^{32} if arg < 0	returns 4 characters
defstr word	returns 16 characters	returns 4 characters	returns 6 characters	5 characters; adds 2^{16} if arg < 0	returns 2 characters
defstr byte	returns 8 characters	returns 2 characters	returns 3 characters	3 characters; adds 256 if arg < 0	returns 1 character

When **defstr byte** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -255 to +255. Likewise, when **defstr word** is in effect, the string functions may not return the expected result if the integer argument lies outside of the range -65535 to +65535.

See Also:
bin$; hex$; oct$; uns$; mki$

delay   **statement**

Syntax:
delay *count*

Description:
This statement causes program execution to pause for *count* milliseconds (a millisecond is a thousandth of a second). Delay timing is accurate to the nearest "tick," which is about 16.7 milliseconds.

Example:
To pause a program for approximately 5 seconds, use the following code:
delay 5000
The following built-in constants are useful for producing delays of various durations:

```
_sec        = 1000   '(1 second)
_secHalf    = 500    '(1/2 second)
_seqQuarter = 250    '(1/4 second)
_secTenth   = 100    '(1/10 second)
_sec60th    = 17     '(about 1 tick)
_secTick    = 17     '(about 1 tick)
```

Note:
The **delay** statement ties up the CPU, so that no other processes can execute while the **delay** is occurring (this limitation does not apply if you're using the "FB Lite" runtime). If you intend to implement a very long delay, then consider instead using a timing loop that includes calls to HandleEvents.

See Also:
timer; time$; date$; HandleEvents

dialog   **function**

Syntax:
evnt%|& = **dialog**(0)
id%|& = **dialog**(*evnt*)

Description:
Please note that the dialog() function is not recommended, use CarbonEvents instead. The **dialog** function returns information about the next event (if any) in FutureBasic's internal dialog queue. An FutureBasic dialog event consists of two parts: an "event type" number, which is returned when you call **dialog**(0), and "event id" number, which is returned by **dialog**(*evnt*) (where *evnt* is the event type number which was returned by **dialog**(0)). The "event type" identifies what kind of event occurred, and the "event id" provides some secondary information about it.

In the Appearance runtime, dialog functions return long integers instead of shorts.

Note that the name "dialog function" is a bit of a misnomer. It has nothing to do with dialog boxes or dialog controls. FutureBasic dialog events consist of a wide range of events, mostly related to the user's interaction with the currently active window.

Normally, you will call **dialog**(0) and **dialog**(*evnt*) from within a dialog-event handling function that your program has designated via the on dialog statement. Your dialog-event handling function should then respond to the event as appropriate.

To make sure that events are properly posted to the dialog queue, your program should call HandleEvents periodically. Among other things, HandleEvents checks the system event queue; translates any applicable system events into FutureBasic dialog events and posts them into the dialog queue; and calls your dialog-event handling function (once per dialog event). HandleEvents will also call your dialog-event handling function if your program has posted an event of type _userDialog (see the dialog statement).

The following pages list the various types of events returned by the **dialog** function. In these tables, "Event Type" refers to the number returned by **dialog**(0), and "ID" refers to the number returned by **dialog**(*evnt*).

Window Events

Event Type	Description	ID
_wndclick (3)	User clicked in the content area of an inactive window. See notes below.	Window ID of the clicked window
_wndClose (4)	User clicked the go away box of an active window. See notes below.	Window ID of the clicked window.
_wndRefresh (5)	A window has been resized, or made visible, or a previously obscured part of the window has been uncovered.	Window ID of the window needing a refresh.
_wndDocWillMove (9)	User clicked in the title bar of a window and is about to drag it.	ID of a _preview event
_wndActivate (18)	A previously inactive window has been made active, or a previously active window has been made inactive (The active window is the frontmost window and is usually highlighted differently from inactive windows.) See notes below.	If ID is positive, the window with that ID number has been made active. If ID is negative, the window whose ID number is abs(ID) has been made inactive.
_wndZoomIn (8)	User clicked in the zoom box of an active window which is currently in the zoomed out state. See notes below.	Window ID of the clicked window.
_wndZoomOut (9)	User clicked in the zoom box of an active window which is currently in the zoomed in state. See notes below.	Window ID of the clicked window.
_wndResized (30)	This is an updated version of the older preview event and tells your program that a window was resized. *(Appearance Manager only)*	Window ID of the resized window.
_inToolBarButton (13)	User clicked in the toolbar attached to a window. *(Carbon only)*	Window ID of the window owning the toolbar.

About _wndClick and _wndActivate events.
When the user clicks on the content region of an inactive window, FutureBasic posts a _wndClick event, and activates the window automatically when your dialog handler returns to the event loop.

When the user clicks on the structure region of an inactive window, FutureBasic posts a _wndActivate event, and activates the window automatically when your dialog handler returns to the event loop.

When you execute a window statement for an inactive window, FutureBasic posts a _wndActivate event, and activates the window

immediately.

A `_wndActivate` event can also be generated in response to other actions: for example, when the active window closes and forces another window to become active; or when your application is brought from the background to the foreground. `_wndActivate` events often occur in pairs: one with a positive "ID" value (an inactive window has become active), and one with a negative "ID" value (the previously active window has become inactive).

Note: Whenever FutureBasic activates a window (for any reason), the newly-active window also becomes the output window (i.e., all new drawing and text commands are sent to that window). If you want to make sure that your program's output doesn't inadvertently get re-directed to the newly activated window, then your program should watch for `_wndActivate` events, and respond by setting the output window (via the `window output` statement, or the `SETPORT` or `SETGWORLD` Toolbox procedures) as appropriate.

Note: Window activation, `_wndActivate` events and `_wndClick` events occur somewhat differently if one or more window has its `_keepInBack` attribute set. See the [window](#) statement for more information.

About `_wndClose`, `_wndZoomIn` **and** `_wndZoomOut` **events**
When the user clicks on the active window's "close box," FutureBasic generates a `_wndClose` event, but does not automatically close the window (your program should execute a `window close` statement in response to the `_wndClose` event, if you want the window to close). When the user clicks on the active window's "zoom box," FutureBasic generates a `_wndZoomIn` or `_wndZoomOut` event, and automatically resizes the window the next time your program calls `HandleEvents`.

Button/Scrollbar Event

Event Type	Description	ID
`_btnClick` (1)	User clicked a button or moved a scrollbar thumb.	Button ID number or scrollbar ID number. **NOTE**: If the toolbar button is clicked in an Appearance Manager window, the button ID returned is `toolBarBtn` (-1)

Contextual Menu Events

Event Type	Description	ID
`_cntxtMenuClick` (24)	The user has clicked in a window with the control key modifier pressed. See the [menu](#) statement for additional information.	The window number.

Key Press Events
Note: This event is not reported if there is an active edit field in the current output window or if you are running under the Appearance Compliant runtime. To capture keypress events in the active edit field, use the `tekey$` function. To make all edit fields in the current window inactive, use the `EDITFIELD0` statement.

Event Type	Description	ID
`_evCmdKey` (24)	User pressed a command-key combination which does not match any active menu command key equivalent	ASCII value of the key that was pressed.
`_evKey` (16)	User pressed a key (or key combination) which can be mapped to an ASCII character.	ASCII value of the key (or key combination) that was pressed.

Cursor (Mouse Pointer) Events
Note: These events might occasionally be missed if the rate at which the cursor moves is too fast compared to the rate at which `HandleEvents` is called.
Note: These events are reported for the output window and for floating windows.

Event Type	Description	ID
`_cursOverBtn` (21)	The cursor is over a button. This is the same as the old `_cursOver`, but is valid only for buttons.	Number of button entered
`_cursOverNothing` (29)	The cursor moved off of a control.	Always zero

User Dialog Events

Event Type	Description	ID
`_userDialog` (23)	The program posted a custom event using the `dialog =` *expr%* statement.	The value of *expr%* that was specified when the event was posted.

Disk Insert Events (Not Supported)

Event Type	Description	ID
_evDiskInsert (17)	A floppy disk, tape, or other removable storage medium was inserted into a drive. (Note: some removable-media devices use their own reporting mechanism and won't generate a disk-insert event.	The drive ID number. (Does not function in Appearance Runtime)

Edit Field Events (Not Supported)
Note: these events apply only to editable (non-static) edit fields, and only in the current output window.

Event Type	Description	ID
_efClick (2)	User clicked in an inactive edit field, and FutureBasic activated the field.	Edit field ID of the clicked field.
_efReturn (6)	User pressed return while a "noCR" type edit field was active.	Edit field ID of the active field.
_efTab (7)	User pressed the tab key while an edit field was active	Edit field ID of the active field.
_efShiftTab (10)	User pressed shift-tab while an edit field was active	Edit field ID of the active field.
_efClear (11)	User pressed the clear key while an edit field was active.	Edit field ID of the active field.
_efLeftArrow (12)	User pressed the left arrow key when the insertion point in the active edit field was already to the left of the first text character.	Edit field ID of the active field.
_efRightArrow (13)	User pressed the right arrow key when the insertion point in the active edit field was already to the right of the last text character.	Edit field ID of the active field.
_efUpArrow (14)	User pressed the up arrow key when the insertion point in the active edit field was already in the top line of text.	Edit field ID of the active field.
_efDownArrow (15)	User pressed the down arrow key when the insertion point in the active edit field was already in the bottom line of text.	Edit field ID of the active field.
_efSelected (26)	A user click or tab key press has changed the focus. (Appearance Manager only)	Edit field ID of the activated field.

Picture Field Events (Not Supported)
Note: This event is only reported for picture fields whose type parameter equals _framed, _framedNoCR, _noFramed, or _noFramedNoCR.

Event Type	Description	ID
_efClick (2)	User clicked in a picture field.	Picture field ID of the clicked field.
_pfClick (25)	User clicked in a picture field. (Appearance Manager only)	Picture field ID of the clicked field.

Cursor (Mouse Pointer) Events (Not Supported)
Note: These events might occasionally be missed if the rate at which the cursor moves is too fast compared to the rate at which HandleEvents is called.
Note: These events are reported for the output window and for floating windows.

Event Type	Description	ID
_cursOverEF (27)	The cursor is over an edit field in the active document window or in any floating window.	Number of edit field entered
_cursOverPF (28)	The cursor is over a picture field in the active document window or in any floating window.	Number of picture field entered

Multi-process Events (Not Supported)

Event Type	Description	ID
		ASCII value of the key (or key combination) that was pressed.ID is set to one of the

`_mfEvent` `(19)`	An event associated with process-switching has occurred. In System 6, these events were associated with the MultiFinder program; hence the letters "MF".	following constants: _mfResume (1) _mfSuspend (2) _mfClipboard (3) _msMouse (4) See the notes below for more information.
`_FBQuitEvent` `(31)`	Your application has received a message to quit. This may be from an Apple Core event or by the user selecting Quit from the application menu.	zero

ID values for "`_MFEvent`" events

`_mfResume` : Your program has been brought to the front, and the clipboard contents were not altered while your program was in the background.

`_mfSuspend` : Your program is being moved to the background or hidden.

`_mfClipboard` : Your program has been brought to the front, and the clipboard contents were altered while your program was in the background. It should be noted that this event cannot be reported under Carbon. Apple suggests the use of the Toolbox function `GetCurrentScrap` for the purpose.

`_mfMouse` : User moved the mouse cursor to an area outside of a special "mouse region" specified by your program. Before this event can be reported, your program must create a region (in global coordinates), and then execute the following statements:

poke long `event-8, tickFrequency%`

poke long `event-4, regionHandle&`

where `regionHandle&` is a handle to the global mouse region, and `tickFrequency%` indicates the desired delay in ticks between successive postings of system events (usually you should set this to 1). The `_mfMouse` event is posted repeatedly for as long as the cursor remains outside of the region. You can later specify a different mouse region by `poke`'ing a different region's handle into `event-4`, or you can disable the event as follows:

poke long `event-4, _nil`

For as long as a region remains the "active" mouse region, you must not dispose of that region's handle.

Preview Events (Partially Supported)

Event Type	Description	ID
`_preview` `(22)`	Usually indicates that some other related event is about to be posted.	"ID" is set to one of the following constants: _premenuclick (1) _prewndgrow (2) _wndmoved (3) _wndsized (4) _efchanged (5) _preefclick (6) _prewndzoomin (7) _prewndzoomout (8) _wnddocwillmove (9) See notes below for more information.

ID values for "`_preview`" events

`_preMenuClick` : User clicked on a menu in the menubar. This event is reported before the menu actually opens.

`_preWndGrow` : User clicked on the active window's size box. This event is reported as soon as the mouse is pressed down.

`_wndMoved` : User clicked on the active window's title bar (or its frame, in Mac OS 8.x - Mac OS 9.x). This event is reported after the mouse button is released.

`_wndSized` : User resized the window. This event is reported after the size change. See `_wndResized` for a new dialog event invoked by the Appearance Manager runtime.

`_efChanged` : User selected the "Clear", "Cut" or "Paste" option from the Edit menu. This applies only when the Edit menu was created by the `edit menu` statement. The event is reported before the text actually changes.

`_preEfClick` : User clicked in a non-static edit field in the active window. This event is reported whether the clicked field is the active field or not. The event is returned after the mouse button is released, but before the `_efClick` event (if any) is reported.

`_preWndZoomIn` : User clicked in the active window's Zoom box (while the window was in the "zoomed-out" state). This event is reported after the mouse button is released, but before the window actually changes size (and before the `_wndZoomIn` event is reported).

`_preWndZoomOut` : User clicked in the active window's Zoom box (while the window was the "zoomed-in" state). This event is reported after the mouse button is released, but before the window actually changes size (and before the `_wndZoomOut` event is reported).

Supported Preview Events in FB5 are:

`_preMenuClick` : User clicked on a menu in the menubar. This event is reported before the menu actually opens.

`_preWndGrow` : User clicked on the active window's size box. This event is reported as soon as the mouse is pressed down.

`_wndMoved` : User clicked on the active window's title bar (or its frame, in Mac OS 8.x - Mac OS 9.x). This event is reported after the mouse button is released.

`_wndDocWillMove` : User started a window drag as reported by the kEventWindowDragStarted Carbon event.

Note:

You can use the `event%` function and the `event&` function to retrieve more details about an event returned by the **dialog** function.

See Also:
HandleEvents; on dialog fn

dialog   **statement**

Syntax:>
dialog `=` *expr%*

Description:

This statement posts a `dialog` event of type `_userDialog` to FutureBasic's internal `dialog` queue. This allows your program to post "custom events" to itself. After you post a `_userDialog` event, you can use the `dialog` function to subsequently read the event from the queue (normally you'll do this from within the dialog-event handling function designated by an `on dialog` statement). If you retrieve the event from the queue as follows:

`evnt` = **dialog**`(0)`
`id` = **dialog**`(evnt)`

then `evnt` will be set to the value `_userDialog` (which equals 23), and `id` will be set to the value of *expr%* that you specified in the **dialog** statement.

To learn how to pass additional information associated with an event of type `_userDialog`, see the descriptions of the `event%` and `event&` statements, and the `event%` and `event&` functions.

See Also:

dialog function; on dialog; HandleEvents

dim   **statement**

Syntax:
dim as *userType* *declaration1* [, *declaration2* . . .]
dim as predefinedType *declaration1* [, *declaration2* . . .]
dim *varName* **as** *userType*
dim *varName* **as predefinedType**

Description:
dim is a non-executable statement that allows the compiler to determine how much storage space should be allocated for the declared variables, arrays and record fields, and identifies their data types. **dim** can also be used to affect the relative storage locations of the declared variables, arrays and fields. The basic syntax is the same for dims within a record and outside a record. There are some exceptions such as arrays inside records, see help for begin record.
A declaration can have any of the following forms:
Simple variables:
    { *varName* | [*maxLen*] *stringVar$*}
    untypedVar as *predefinedType*
    untypedVar as *userType*
Records:
    untypedVar as *recordType*
    untypedVar . *constant2*
Arrays:
    varName | [*maxLen*] *stringVar$*}(*maxSub1*[, *maxSub2* . . .])
    untypedVar (*maxSub1* [, *maxSub2* . . .]) as *predefinedType*
    untypedVar (*maxSub1* [, *maxSub2* . . .]) as *userType*
    untypedVar . *constant2* (*maxSub1* [, *maxSub2* . . .])
Pointers:
    untypedVar as {pointer to|^|@|.}{ *predefinedType* | *recordType*}
Handles
    untypedVar as {Handle to|^^|@@|..}{ *predefinedType* | *recordType*}
Memory alignment declarations:
    {%|&|&&|&&&| . *constant*}

- *maxLen*, *maxSub1*, *maxSub2* and *statExpr* are (non-negative) static integer expressions.
- *constant* is a (non-negative) literal integer or a symbolic constant; but if a symbolic constant is used, its initial underscore character is omitted.
- *stringVar$* is a variable name that ends with a "$" type-identifier suffix.
- *varName* is a variable name that may optionally end with a type-identifier suffix.
- *untypedVar* is a variable name that does not end with a type-identifier suffix.
- *userType* is a type name defined in a previous begin record statement or #define statement.
- *recordType* is a type name defined in a previous begin record statement.
- *predefinedType* is one of the following: char, [unsigned] byte, [unsigned] word, [unsigned] short, [unsigned] int, UInt16, SInt16, [unsigned] long, UInt32, SInt32, UInt64, SInt64, Rect, Handle, RgnHandle, Str255, Str63, Str31, Str15, double, single.

If a **dim** statement appears within a begin globals...end globals block, then the scope of the declared variables and arrays is global. If it appears within the scope of a local function (but not within a begin globals...end globals block), then the scope of the declared variables and arrays is local to that function or procedure block. If **dim** appears outside of all local functions and procedure blocks (and outside of any begin globals...end globals block), then the scope of the declared variables and arrays is local to "main."

Your program can use as many **dim** statements as you like. The following statement:
dim a, b&, c$, d#
is equivalent to this pair of statements:
dim a, b&
dim c$, d#
Certain structures must always be declared in a **dim** statement, which must appear somewhere above the first line where the structure is used. These structures include:

- Arrays (unless declared in an xref or xref@ statement)

- Record variables
- Variables of user-defined types
- Pointer variables and Handle variables

Storage space for FutureBasic's built-in types is allocated as follows:

byte integers (`` ` ``, `` `` ``), `byte`, `char`, `Boolean`	1 byte
short integers (`%`, `%` `` ` ``), `short`, `int`, `SInt16`, `UInt16`	2 bytes
long integers (`&`, `&` `` ` ``), `long`, `SInt32`, `UInt32`	4 bytes
long long integers `SInt64`, `UInt64`	8 bytes
`Point`	4 bytes
single precision (`!`), `single`	4 bytes
double precision (`#`), `double`	8 bytes
`Rect`	8 bytes
`Str255`	256 bytes
`pointer`	4 bytes
`Handle`	4 bytes

(Note: the storage space for variables can vary between CPU devices. When in doubt, use the `sizeof` function to make a definite determination of the size of the variable.)

The storage space allocated for a string variable depends on the value of the `maxLen` parameter (which cannot exceed 255). If `maxLen` is omitted, then the most recent `maxLen` specified in the same **dim** statement is used. String variables declared using the `as Str255` clause always have a `maxLen` value of 255.

Once `maxLen` has been determined for a given string variable, the actual number of bytes allocated for the variable is:

- `maxLen` + 1 bytes, if `maxLen` is odd;
- `maxLen` + 2 bytes, if `maxLen` is even;

Your program should not assign a string longer than `maxLen` characters to a string variable. The storage space for a record variable equals the sum of the lengths of the record's fields, or the value of `constant2` (in bytes).

Storage space for an array is calculated as follows: If `elSize` is the size in bytes of a single array element, then the space allocated for the entire array is given by the following expression:

```
array size = elSize * (maxSub1 + 1) * (maxSub2 + 1) * ...
```

All the elements in an array are stored in contiguous locations in memory. If the array is multi-demensional, then the rightmost dimensions change most rapidly as you step through the elements' locations in memory. For example, if you declare an array as follows:

dim `p%(3, 2)`

Then the elements of `p%()` are stored in this order in memory:

```
p%(0,0)
p%(0,1)
p%(0,2)
p%(1,0)
p%(1,1)
p%(1,2)
p%(2,0)
p%(2,1)
p%(2,2)
p%(3,0)
p%(3,1)
p%(3,2)
```

Aliased variables

Aliased variables are no longer supported. For example, the following syntax cannot be used:

dim as int`;0,` hi **as byte** `,` lo **as byte**

See Also:

begin globals...end globals; begin record...end record

See the dynamic statement.

DisposeH   **statement**

Syntax:
DisposeH (*handle&*)

Description:
If *handle&* represents a valid handle to a relocatable memory block, this statement disposes of the block, and sets the value of *handle&* to zero (*handle&* must be a long-integer variable or a `Handle` variable).
If *handle&* does not represent a valid handle, the statement sets the value of *handle&* to zero, but otherwise does nothing.

Note:
Never use **DisposeH** on a resource.

See Also:
kill field

do **statement**

Syntax:
do [*statementBlock*] **until** *expr*

Description:
The **do** statement marks the beginning of a "do-loop," which must end with an **until** statement. *statementBlock* represents a block of zero or more executable statements, possibly including other do-loops. When a **do** statement is encountered, FutureBasic executes the statements (if any) in *statementBlock*, and then evaluates *expr*. If *expr* is zero, then the process is repeated. The statements in *statementBlock* are repeatedly executed until *expr* evaluates to a nonzero value, at which point the loop exits and the next statement after **until** is executed. Typically, *expr* is an expression involving logical operators, which is evaluated either as _zTrue (-1) or _false (0). See the if statement for more information about *expr*.
Note that the statements in *statementBlock* always executed at least once. If you want to use a looping structure which may possibly skip over the *statementBlock* without executing it, consider using a while/wend loop.

See Also:
for...next; while...wend; if

dynamic **statement**

Syntax:
[**dim**] **dynamic** *arrayName*(*maxSub1*[*, maxSub2* ...]) [**as** *dataType*]

Description:
The **dynamic** syntax is an alternate version of `dim` that allows for arrays that can grow as needed. The constant expression used in the parenthesis is ignored. Dynamic arrays may only be created and used as global arrays. Do not attempt to dimension them inside of a `local fn`.

Example:
```
// 1000 elements max
dynamic myIntArray(1000)
// 2 gig elements max
dynamic hugeEmployeeRecAry(_maxLong) as employeeRec
// 32,767 elements max
dynamic arrayOfRects(_maxInt) as Rect
```
The *maxSub1* , *maxSub2* etc. values must be positive static integer expressions. However, since **dynamic** does not actually allocate any memory, the declared subscripts are used somewhat differently than in a `dim` statement. The second and subsequent subscripts (if any) determine the internal structure of the array, and space for them will be fully allocated for each element dynamically referenced in the first subscript. But the value of the first subscript (*maxSub1*) is ignored, and may be arbitrarily set to any value greater than zero. You can actually reference array elements greater than *maxSub1*, so long as adequate RAM is available to allocate the memory required.

Auto Grow
To obtain the next index for a dynamic array:

```
begin globals
dynamic MyDynArray(1) as long
end globals
dim nextIndex as long
MyDynArray(567) = 1
nextIndex = fn DynamicNextElement( dynamic( MyDynArray ) )
// nextIndex is 568
```

Note:
Dynamic arrays may only be global in nature and may not be dimensioned inside of a `local fn`.

WARNING:
Dynamic arrays may not be used to dimension an array of handles.

See Also:
fn DynamicNextElement; kill dynamic; read dynamic; write dynamic

DynamicInsertItems   **statement**

Syntax:
DynamicInsertItems (*gMyDynamicArray, Where&, HowMany&, FillPtr&*)

Description:
This function shifts item *Where&* and all subsequent items in *gMyDynamicArray*, *HowMany&* positions higher, to leave *HowMany&* new items beginning at position *Where&*. The total number of items in the array increases by *HowMany&* (or more - see below), and the inserted items (beginning at *Where&*) are filled with data located at *FillPtr&*, or with zeros if *FillPtr&* is 0.

Note:
Whether currently populated or not, a dynamic array must have held data at some point before being passed to this function.

gMyDynamicArray is any FutureBasic Dynamic Array previously dimensioned using `dynamic` or `dim dynamic`.

Where& is the array position at which the first item will be inserted. It must be >= 0. *Where&* will normally be less than the current number of items in the array, but can be greater. If it is greater, enough new empty items will be inserted to provide *HowMany&* items beginning at item *Where&*. For example, if you have 10 items (0-9) in the array, and insert 2 items beginning at item #13, your array will hold the original 10 items, followed by 3 empty items (10-12), and the 2 inserted items (13-14) for a total of 15 items.

HowMany& specifies the number of items to be inserted. It also represents the number by which the index of any specific item higher in the array will increase.

FillPtr& is an address where new data are waiting to be inserted into *gMyDynamicArray*. These must be in the same format, with the same size elements as *gMyDynamicArray*. In OS 9, if you pass a dereferenced handle as *FillPtr&*, you should first lock the handle. **DynamicInsertItems** does not check to ensure there is adequate data to copy.

See Also:
DynamicRemoveItems; dynamic

fn DynamicNextElement function

Syntax:
nextIndex = **fn DynamicNextElement** (**dynamic** (*gArray*))

Description:
This function returns 1 + the highest used index of a dynamic array.

Example:
begin globals
dynamic gMyDynArray(1) **as long**
end globals
dim nextIndex **as long**
gMyDynArray(567) = 1234
nextIndex = **fn** DynamicNextElement(**dynamic**(gMyDynArray))
// nextIndex is 568

See Also:
DynamicInsertItems; dynamic; DynamicRemoveItems

DynamicRemoveItems   **statement**

Syntax:

DynamicRemoveItems (**Dynamic** (*gMyDynamicArray*) *, First&, HowMany&, SavePtr&*)

Description:

This function deletes *HowMany&* items from *gMyDynamicArray*, beginning with item *First&*. Any subsequent items will shift down to replace the removed items, resulting in an array of *HowMany&* fewer items. Each subsequent item will have its index reduced by *HowMany&*. If *SavePtr&* is 0, the data will be expunged without warning or recourse. If a pointer is passed in *SavePtr&*, the data to be removed will first be copied to that address.

Note:

Whether currently populated or not, a dynamic array must have held data at some point before being passed to this function.

gMyDynamicArray is any FutureBasic Dynamic Array previously dimensioned using `dynamic` or `dim dynamic`.

First& is the array position of the first item to be removed. It must be >= 0. **def DynamicRemoveItems** will not remove more items than exist in the array. For example, if your array holds 10 items (0-9) and you attempt to remove 5 items beginning with item 8, 2 items (8-9) will be removed instead and the array will be left with 8 items (0-7).

HowMany& specifies the number of contiguous items to be removed. If there are too few items following *First&,* only the number available will be removed. *SavePtr&* is an address to which the items being removed from *gMyDynamicArray* will be copied.

SavePtr& must point to an allocated memory block (or variable) of adequate size to hold all data being removed. **DynamicRemoveItems** does not check to ensure there is adequate space. If there are fewer than *HowMany&* items available to remove, only the number removed will be copied to *SavePtr&*. To remove data without saving it, pass 0 in *SavePtr&*.

See Also:

DynamicInsertItems; dynamic

edit field statement

Syntax:
edit field [#] *idExpr* [,[*text*][,[*rect*][,[*type*],¬ [*efClass*],[*keyFilterProc*]]]]
edit field – *idExpr*
edit field 0

Description:
Use this statement to perform any of the following actions in the current output window:

- Create a new edit field
- Activate an existing editable (non-static) edit field
- Modify the characteristics of an existing edit field
- Deactivate all editable fields in the window
- Disable a field

In the new Appearance Manager runtime, edit fields are actually buttons. FutureBasic creates special controls called user panes that allow styled text to be present. It is important to note that items created with the **edit field** statement are different from those created with the `appearance button` statement. Text items made with `appearance button` do not accommodate multiple runs of styled text, proper scrolling, and other necessary items. Text items created with the **edit field** statement have most of the features found in Standard BASIC edit fields with the added advantages of Appearance Manager compliance.

To disable an Appearance Manager field, use **edit field** with a negative value. To gray out field #22, you would use the statement **edit field** –22.

In FutureBasic, an edit field can be either static or non-static. A static edit field contains text that is used for display purposes only; the user cannot edit the contents of such a field. A non-static field is editable; the user can use mouse and keyboard commands to edit the field's contents. It is possible for the program to change a field's type from static to non-static (or vice-versa); this is sometimes useful when you want to temporarily inhibit the user from editing some text. The Appearance Manager adds the ability to create "copy only" fields in which text may be selected and copied to the desk scrap but may not be edited.

Key Filter Procs
It is possible to filter key presses directed at fields in the Appearance Manager. The Appearance Manager has the ability to direct every keypress destined for a target field to a special procedure which you establish in your program and point to at the time the field is created. The following fully functional example creates a simple numeric filter that limits field entry to the digits zero through nine.

```
local fn numeralFilter
  dim k as Str15
  k = tekey$
  long if k >= "0" and k <= "9"
    tekey$ = k
  xelse
   select asc( k )
     case < 32 : rem allow for control keys
      tekey$ = k
     case else
       beep
    end select
  end if
end fn
dim @ filter as pointer
dim r as Rect
window 1
SetRect( r, 10, 10, 450, 60 )
filterFN = @fn numeralFilter
edit field 1, "Numbers Only", @r,,, filterFN
do HandleEvents until 0
```

A non-static (editable) field can be either active or inactive. In the frontmost window, an active field contains the blinking insertion point or a highlighted selection; it is the field whose contents the user is currently editing. At most one field in the window can be active at any given time; whenever a field becomes active, all other editable fields in the window become inactive. It is also possible to inactivate all of the editable fields in a window.

If your window contains an active edit field, then your program should call `HandleEvents` periodically (it's a good idea for your program to do this in any case). This will allow the user's keypresses and mouse events to be transmitted to the active field as appropriate, and will cause the field's contents to be updated correspondingly. Your program can use the `edit$` function or the `get field` statement to check the contents of a field at any time.

FutureBasic automatically refreshes (redraws) both static and non-static edit fields as necessary, unless the window's `_noAutoClip` feature is set.

To create a new edit field: Specify a positive or negative number in *idExpr*, such that `abs(`*idExpr*`)` is different from the ID numbers of all other existing edit fields or picture fields in the current window. The *rect* parameter is required in this case. If *idExpr* is positive, then a "non-styled" edit field is created, and is assigned an ID number equal to *idExpr*. If *idExpr* is negative, then a "multistyled" edit field is created, and is assigned an ID number equal to `abs(`*idExpr*`)`. A "non-styled" edit field adopts the text characteristics (font family, size, style and color) that were in effect for the window at the time the field was created, and all the text in that field has those characteristics. A "multistyled" edit field can contain different pieces of text which each have different text characteristics. NOTE: When you create a new non-static edit field, the new field becomes the active field in that window.

When you create a new edit field, any parameters that you omit have these default values:

- *text* defaults to a null string (i.e., the field is empty)
- *type* defaults to `_framedNoCR` (this is one of the editable types)
- *efClass* defaults to zero, which is one of the left-justified classes.

To activate an existing non-static edit field: Specify the ID number of the existing field in *idExpr*. You don't need to specify any other parameters, unless you also wish to alter some of the field's characteristics.

To modify the characteristics of an existing edit field: Specify the ID number of the existing field in *idExpr*, and specify one or more of the other parameters. Any parameter that you omit won't have its characteristic changed. Note that if the field is editable (non-static), this command will also make it the active field.

If the only thing you want to change about the field is its contents, then you can alternatively use the `edit$` statement. The `edit$` statement will not make the field active.

To inactivate all edit fields in the window: Use the **edit field** `0` syntax.

To make a static field editable: You can Specify any of the editable types in the *type* parameter.

Note: Changing a static field to an editable field also makes the field active.

To make an editable field static: First, deactivate the field (either by executing **edit field** `0` or by activating a different field), and then specify any of the static types in the *type* parameter.

The following sections explain the use of the various parameters.

text

This parameter can be specified in any of the following forms:

- `string$` -- Any string expression.
- `&ZTXThandle&` -- An "`&`" symbol followed by a handle to a "`ZTXT`" structure.
- `%TEXTresID%` -- A "`%`" symbol followed by the resource ID number of a "`text`" resource.
- `#container$$` -- A "`#`" symbol followed by a container variable indicates that the contents of the container should be copied to the field. If the container has a length that is greater than 32,767, then only the first 32,767 bytes are copied to the field.

The field's contents are completely replaced by the specified text. The various forms of this parameter are interpreted the same way as in the `edit$` statement; see the edit$ statement for more information.

rect

This parameter specifies the "view rectangle" for the field. No text will be drawn outside of this rectangle. This parameter can be specified in either of the following forms:

- `(x1%, y1%)-(x2%, y2%)` -- These coordinates specify two diagonally opposite corners of the rectangle.
- `rectAddr&` -- A pointer or long integer expression. This is interpreted as the address of a standard 8-byte "`rect`" structure.

type

This is an integer which specifies several characteristics about the field.

Please note that not all of the old FBII style field types are available for the Appearance Manager Runtime. The following list shows all of the types that are *not* available for Appearance projects:

```
_statFramedGray
_statNoFramedGray
_statFramedInvert
_statNoFramedInvert
_statFramedInvert + _hilite
_statNoFramedInvert + _hilite
```

The following 24 types only are supported, (each in two forms, with and without `_usePlainFrame`, making 48 altogether):

```
_copyOnlyFramed                    _copyOnlyFramed_autoGray
_copyOnlyNoFramed                  _copyOnlyNoFramed_autoGray
_framedNoCR                        _framedNoCR_autoGray
_framed                            _framed_autoGray
_noFramedNoCR                      _noFramedNoCR_autoGray
_noFramed                          _noFramed_autoGray
_framedNoCR_noDrawFocus            _framedNoCR_autoGray_noDrawFocus
_framed_noDrawFocus                _framed_autoGray_noDrawFocus
```

_noFramedNoCR_noDrawFocus   _noFramedNoCR_autoGray_noDrawFocus
_noFramed_noDrawFocus   _noFramed_autoGray_noDrawFocus
_statFramed   _statFramed_noAutoGray
_statNoFramed   _statNoFramed_noAutoGray

Editable (non-static) types:

_framedNoCR (1)	Frame is drawn. "Return" key does not advance line, but generates an _efReturn event. This is the default type.
_framed (2)	Frame is drawn. "Return" key advances insertion point to next line, does not generate an _efReturn event.
_noFramedNoCR (3)	Like _framedNoCR, but no frame is drawn.
_noFramed (4)	Like _framed, but no frame is drawn.
_statFramed (5)	Frame is drawn.
_statNoFramed (7)	No frame is drawn.
_statFramedGray (9)	Frame is drawn; frame and text are dimmed.
_statNoFramedGray (11)	No frame is drawn; text is dimmed.
_statFramedInvert (13)	Frame is drawn; text and background colors are inverted.
_statNoFramedInvert (15)	No frame is drawn; text and background colors are inverted.
_statFramedInvert+hilite (29)	Frame is drawn; text and background are highlighted using system highlight colors.
_statNoFramedInvert+hilite (31)	No frame is drawn; text and background are highlighted using system highlight colors.
_noDrawFocus (256)	Use this option if you don't want the focus rectangle outlining the active field.
_noAutoGray (512)	When the window goes to the background, text in this field will not be grayed.
_autoGray (1024)	Text is automatically grayed when the window goes to the background.
_copyOnlyFramed (2048)	The user may selct and copy text from this field but may not edit the text.
_copyOnlyNoFramed (2051)	This is the same as _copyOnlyFramed without the frame.
_usePlainFrame (4096)	When added to a _framed or _framedNoCR type, gives an old-fashioned but crisp rectangular frame instead of the trendy fuzz obtained with DrawThemeEditTextFrame. The runtime currently requires a frame/CR constant to be supplied as well as the special _usePlainFrame. Thus you will have to specify: _framedNoCR_usePlainFrame or _framed_usePlainFrame

efClass
This parameter must be within the range 0 through 255 (0 through 536,870,912 in the Appearance Manager). It serves two purposes:

- It determines how the lines of text will be positioned in the field (i.e., left-justified, right-justified or centered);
- For editable fields, it assigns a user-defined "class number" to the field. The number you specify will subsequently be returned by the window(_*efClass*) function when the field is active.

If *efClass* is zero, then the text will be left-justified. Otherwise, text justification is determined by the low-order two bits in *efClass* (given by efClass mod 4), as follows:

If efClass **mod** 4 is:	Then the text is:
zero	Right justified.
_leftJust (1)	Left justified.
_centerJust (2)	Centered.
_rightJust (3)	Right justified.

If you just want to set the field's text justification, and you don't care about its "class," then the easiest thing to do is just to set *efClass* to one of the constants _leftJust, _centerJust or _rightJust.

Note:
An edit field cannot contain more than 32,767 characters of text.
The **edit field** statement is not the only means by which an inactive field can become active. If the user clicks on an inactive editable field, FutureBasic automatically activates the field the next time your program executes the HandleEvents statement.

See Also:
HandleEvents; edit text; edit$ statement; edit$ function; read field

edit field close   **statement**

Syntax:
edit field close `[#]` *fieldID% | &*

Description:
This statement removes the specified field from the current output window. *fieldID&* can refer either to an edit field or a picture field. Note that all fields in a window are closed automatically whenever the window is closed. The contents of the field are no longer accessible by the `edit$` function or the `get field` statement after the field has been closed.

See Also:
edit field; edit$ function

edit menu   **statement**

Syntax:
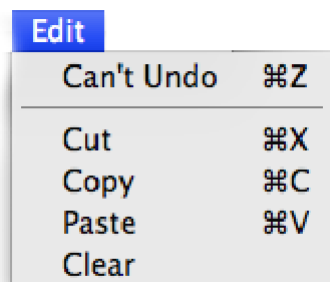**edit menu**   *menuParm*

Description:
This statement builds an edit menu (see illustration) with the following characteristics:

- If an edit field is active in the current output window, then the menu's Cut, Copy, Paste and Clear options move text between the edit field and the "TextEdit Scrap" (a private clipboard area) in the standard ways. If the active field is a multistyled field, then style information is also moved.
- If a multistyled edit field is active in your application's output window, then FutureBasic exchanges text and style information between the TextEdit scrap and the clipboard whenever your application is moved to the front or to the back (this allows you to cut & paste text to and from other applications).

If *menuParm* is greater than zero, then it's interpreted as a menu ID number, and the menu shown in the illustration is put into the specified position (almost always "2") in the menu bar.
If an edit field is active in the current window, then selecting the Edit menu's Cut, Copy, Paste and Clear options will not generate FutureBasic Menu events (FutureBasic will handle these options transparently to your program). If the user selects any other item from the Edit menu, a Menu event will be generated.
If there is no active edit field in the current window, then all items in the Edit menu will generate Menu events.

If you wish to turn off automatic handling of Edit menu events, use:
edit menu   0

See Also:
menu function; menu statement; on menu; HandleEvents; edit field

edit text **statement**

Syntax:
edit text [*#optionalFieldNum%* ,][*font*]¬ [, [*size*][, [*style*][, [*mode*][, *foreRGB*][, *backRGB*]]]]

Description:
If the active edit field or the field specified by *#optionalFieldNum%* in the current output window is a multistyled edit field, then **edit text** applies the specified text characteristics to any text which is currently selected. Also, any new text which is subsequently inserted at the current insertion point or within the current selection range will be displayed using the specified text characteristics.
If the active field (or the field specified by *#optionalFieldNum%*) is not a multistyled field, then **edit text** causes the specified text characteristics (except color) to be applied to all the text in the field.
The parameters have the following meanings. If you omit any parameters, the corresponding text characteristics won't be altered. Multistyled fields not supported.

#optionalFieldNum	field number that should accept the font changes. When omitted, the active edit field is used.
font	font family ID
size	font size, in points
style	bold, italic, underline, etc. See the text statement for more information.
mode	This parameter is currently ignored.
foreRGB	An RGB color record for the text color which could be dimensioned as: **dim** `foreRGB` **as** `RGBColor`
backRGB	An RGB color record for the background color which could be dimensioned as: **dim** `backRGB` **as** `RGBColor`

See Also:
text; SetSelect; edit field

edit$ **function**

Syntax:

*fieldContents$* | *container$$* = **edit$** (*fieldID*)
*fieldContents$* | *container$$* = **edit$** (*fieldID* , *lineNumber*)
*fieldContents$* | *container$$* = **edit$** (*fieldID* , -1)
*fieldContents$* | *container$$* = **edit$** (*fieldID* , *selStart* , *selEnd*)

Description:

This function returns the contents of the specified edit field in the current output window. If the edit field contains more than 255 characters of text and a string is specified to receive the information, then only the first 255 characters are returned. Where a container is the specified target, there is more than enough room to hold the contents of a field. Use the get field statement if you need to retrieve style information along with the text of a styled edit field.

If a single parameter is used, the **edit$** function attempts to return as much of the entire edit field as will fit into the target variable.

If a second parameter of -1 is added, the function returns the current selection in whole or part depending on the size of the target variable. A second parameter that consists of a numeric value specifies the line number that is to be returned.

Where three parameters are used, the second and third values are used to specify the starting and ending points of text to be captured. If the target variable is a string instead of a container, no more that 255 bytes of information can be returned.

In all cases, FutureBasic ensures that variables are not overflowed. This is important because we are working with three different sizes of items here.

Item	Size
Pascal String	255 bytes + length byte
Edit Field	32,767 bytes
Container	2 gigabytes

If *fieldID* refers to a picture field, then the **edit$** function returns the pictID$ string that was specified in the picture field statement. You can use this to identify the handle or resource that contains the picture.

If there is no edit field nor picture field with an ID of *fieldID* in the current output window, the **edit$** function returns a null string.

See Also:

edit field; edit$ statement; window(_efNum)

edit$ **statement**

Syntax:

edit$(*efID*) = ¬
  string$* | *&ZTXThandle&* | *%TEXTresID%*| *#container$$¬
  font , *size* , *style* , *mode* , *red* , *green* , *blue*

edit$(*efID,lineNumber*) = ¬
  string$* | *&ZTXThandle&* | *%TEXTresID%*| *#container$$¬
  font , *size* , *style* , *mode* , *red* , *green* , *blue*

edit$(*efID* , -1) = ¬
  string$* | *&ZTXThandle&* | *%TEXTresID%*| *#container$$¬
  font , *size* , *style* , *mode* , *red* , *green* , *blue*

edit$(*efID,selStart,selEnd*) = ¬
  string$* | *&ZTXThandle&* | *%TEXTresID%*| *#container$$¬
  font , *size* , *style* , *mode* , *red* , *green* , *blue*

Description:

This statement replaces all or a portion of the text in the edit field whose ID is *efID* in the current output window. *string$* is any string expression. *ZTXThandle&* is a handle to a "ZTXT" block (a "ZTXT" block contains text and (optionally) style information. *TEXTresID%* is the resource ID number of a "text" resource in an open resource file. A container is a variable that can hold up to 2 gigabytes of information. In the case of edit fields, the information is truncated at 32,767 characters which is the maximum number of characters allowed in an edit field. If the field is a multistyled field, and you use the *&ZTXThandle&* syntax, then any style information found in the "ZTXT" block is applied to the field's text.

If a single parameter is used, the **edit$** statement attempts to replace as much of the edit field text as will fit (up to 32K).

If a second parameter of -1 is added, the function replaces the current selection in whole or part depending on the size of the source variable. A second parameter that consists of a numeric value greater than or equal to zero specifies the line number that is to be replaced. Where three parameters are used, the second and third values specify the starting and ending points of text to be replaced.

In all cases, FutureBasic insures that the field is not overflowed.

When the style parameter is used, the runtime toggles specific style attributes instead of setting them. For instance, the bold style may be toggled on then off again by using the same style parameter of _boldbit%. If you wish to reset the selection to a particular style regardless of its current state, then begin by using a style parameter of zero before resetting the desired bits.

Inserted text may take on specific font characteristics as described by new parameters. This feature is available starting with Release 4. You may describe a font (by number), a size, style, and text mode. RGB colors are also definable using the red, green and blue parameters. Any of these parameters may be eliminated as long as a comma is retained as a place holder. The following example places "Mississippi" in the third line of the field using red text in a Monaco font.

edit$(4,3) = "Mississippi",_monaco,,,-1

Example:

edit$(1) = "555-2467"
edit$(2) = &myZTXThndle&
edit$(3) = %-350
**edit$**(4,-1) = ReplaceCurrSel$
**edit$**(5,12) = ReplaceLine12$
edit$(6,10,20) = "Replace characters 10 thru 20 with this."
**edit$**(7,-1) = ReplaceCurrSel$,newFont,newSize,newStyle
**edit$**(8) = #myBigContainer$$

Colors

Colors can be specified either as red,[green,[blue]] or @rgb, where rgb is an RGBColor record. Note: the runtime distinguishes between a red value and an rgb address by taking any value from 0 to 65535 to mean "red", with any other value meaning "address of rgb record". Thus you cannot use a negative red values and must use the unsigned integer equivalent, for example 45536 not -20000.

See Also:

edit$ function; edit field; read field; window(_efNum)

else **statement**

See the if or long if statement.

end   **statement**

Syntax:
end

Description:
This statement calls your stop-event handling routine (if you've designated one using the `on stop` statement), then closes all open files and ports, disposes of all handles and pointers, releases all resources, and stops execution of the program.
Note:
FutureBasic always inserts an implicit **end** statement following the last executable line in your program. You don't need to explicitly include an **end** statement unless you want the program to end somewhere before that last line is reached.

See Also:
stop; system statement; shutdown

end enum statement   **statement**

See the begin enum statement.

See the local fn statement.

end globals statement   **statement**

See the begin globals statement.

end if statement   **statement**

See the if or long if statement.

end record statement   **statement**

See the begin record statement.

end select statement   **statement**

See the select statement.

EndC statement

Syntax:
EndC

Description:
Required block termination keyword for **BeginCCode**, **BeginCFuntion**, or **BeginCDeclaration**.

See Also:
BeginCCode; BeginCFunction; BeginCDeclaration; #if

eof **function**

Syntax:
endReached = **eof** (*fileID*)

Description:
This function returns _zTrue if the "file mark" associated with the specified open file is positioned at the end of the file. The "file mark" is an internal pointer which is used by all operations which read from or write to the file; it indicates where in the file the next data should be read from or written to. You can move the file mark explicitly using the record statement; the file mark is also advanced automatically every time you do a read or write operation. Typically, you use the **eof** function when reading data sequentially, to determine when no more data can be read from the file.

Note:
If your program attempts to read data past the end of the file, FutureBasic returns an _endOfFile error to your program.

See Also:
error function; error statement; on error fn; on error fn/gosub; open; close; record; rec; pos; lof

erf# & erfc#   **function**

Syntax:
error# = **fn erf#** (*z#*)
complementaryError# = **fn erfc#** (*z#*)

Description:
ERF is shorthand for error function. ERFC stands for complementary error function. The error function Erf[z] is the integral of the Gaussian distribution given by

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_{0}^{z} e^{-t^2} dt$$

The complementary error function ERFC[z] is
erfc (z) = 1 - **erf** (z)
The error functions are located in the file named "Subs Float Addns.Incl" (Path: FutureBasic Extensions/Compiler/Headers). To provide access to these functions you must use the following statement in your program.
include "Subs Float Addns.Incl"
FutureBasic will automatically locate the file and compile it with your project. The **ERF** and **ERFC** functions are provided as local functions and expect a double precision (#) value as the incoming parameter. The return value is also a double precision (#) number.

erf# & erfc#   **function**

Syntax:
error# = **fn erf#** (*z#*)
complementaryError# = **fn erfc#** (*z#*)

Description:
ERF is shorthand for error function. ERFC stands for complementary error function. The error function Erf[z] is the integral of the Gaussian distribution given by

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_{0}^{z} e^{-t^2} dt$$

The complementary error function ERFC[z] is
erfc (z) = 1 - **erf** (z)
The error functions are located in the file named "Subs Float Addns.Incl" (Path: FutureBasic Extensions/Compiler/Headers). To provide access to these functions you must use the following statement in your program.
include "Subs Float Addns.Incl"
FutureBasic will automatically locate the file and compile it with your project. The **ERF** and **ERFC** functions are provided as local functions and expect a double precision (#) value as the incoming parameter. The return value is also a double precision (#) number.

error   **function**

Syntax:
errorInfo = **error**

Description:
This function returns information about the most recent disk IO error encountered in your program. It relates only to IO errors in the FutureBasic runtime; it does not include string errors, numeric errors, or error codes returned by MacOS Toolbox functions.

The **error** function returns an OSStatus value such as 0 (_noErr), -35 (_nsvErr), -43 (_fnfErr), -5000 (afpAccessDenied).

Note that these kinds of errors will normally cause your program to stop executing. If you want to trap and handle these errors within your program, you must use the on error fn and on error end statements.

Note:
After retrieving the error information with the **error** function, you should reset FutureBasic's internal error register by executing the error = _noErr statement.

See Also:
error statement; on error fn; on error gosub

error   **statement**

Syntax:
error = _noErr

Description:
This statement clears FutureBasic's internal error register after an error has occurred. If your program has an error-handling function (defined using the on error fn statement), then you should execute this statement within that function, after you have used the **error** function to retrieve information about the error.

See Also:
error function; on error fn/gosub

event   **function**

Syntax:
eventRecPtr& `=` **event**

Description:
This function returns a pointer to a system event record. If your program has designated a system event-handling routine using the `on event` statement, then your routine should check the contents of this record every time the routine is called. The record will contain a description of the event that triggered the call to your system event-handling routine.

If you store the record pointer into a long integer or `pointer` variable (e.g., *eventRecPtr&* `=` **event**), then you can examine the individual fields of the event record as follows:
`type% = eventRecPtr&.evtNum%`
This represents the event type, which will be one of the following constants:

`type%`	Description
`_nullEvt (0)`	No event occurred
`_mButDwnEvt (1)`	mouse button pressed
`_mButUpEvt (2)`	mouse button released
`_keyDwnEvt (3)`	key pressed
`_keyUpEvt (4)`	key released
`_autoKeyEvt (5)`	key repeatedly held down
`_updatEvt (6)`	window needs updating
`_diskInsertEvt (7)`	disk inserted
`_activateEvt (8)`	window was brought to front or moved to back (see the `evtMeta%` field to determine which)
`_osEvt (15)`	operating system events (suspend, resume, mouse moved)
`_kHighLevelEvent (23)`	high-level events (includes Apple Events)

`message& = eventRecPtr&.evtMessage&`
This contains additional information about the event, which varies depending on the type of event that occurred.

`type%`	message&
`_keyDwnEvt, _keyUpEvt, autoKeyEvt`	bits 0-7 = ASCII char; bits 8-15 = key code; for ADB keyboards, bits 16-23 = keyboard's ADB address.
`_updatEvt, activateEvt`	pointer to the window
`_diskInsertEvt`	bits 0-15 = drive ID; bits 16-31 = error code
`_osEvt`	high-order byte = 1 for suspend/resume events; high-order byte = 250 for mouse-moved event. For suspend/resume events: bit 0 = 0 for suspend; bit 0 = 1 for resume. For resume event: bit 1 = 1 if clipboard was changed.
`_kHighLevelEvent`	Class of events to which the high level event belongs. This is used with the `_evtMouse` field to identify the specific type of high-level event received.

`ticks& = eventRecPtr&.evtTicks&`
This gives the tickcount value (time since system startup, in ticks) when the event occurred. You can compare this against the current value of `fn TickCount`, to determine how long ago the event occurred.
`mousePt;4 = eventRecPtr& + _evtMouse`
For low level events (i.e., all types except `_kHighLevelEvent`), this gives the mouse cursor location, in global coordinates, at the time the event occurred. `mousePt` should be `dim`'ed as a 4-byte record.
`highLevelEvtID& = eventRecPtr&.evtMouse&`
For high-level events (i.e., of type `_kHighLevelEvent`), this gives the high-level Event ID, which together with `message&` identifies the type of high-level event.
`modKeys% = eventRecPtr&.evtMeta%`
Contains information about the state of the modifier keys (Control, Shift, etc.) and the mouse button, at the time the event occurred. For activate events, bit 0 = 1 if the window should be activated; bit 0 = 0 if the window should be deactivated. The state of the modifier keys and

mouse button is determined by specific bit values in this short integer; see the `event%` function for more information.

"Clearing" an Event

After you've finished looking at an event record and you exit your system event-handling routine, FutureBasic examines that same event record to determine whether the event can be translated into one of the "FB events" that your program typically detects using the `dialog` function, the `menu` function, etc. For example, if FutureBasic sees an event record in which a `_mButDwn` event has occurred, it checks whether the mouse was clicked inside a button, and tracks the mouse in preparation for a possible `dialog` event of type `_btnClick`. If you've handled the event completely within your `on event` routine, and you don't want FutureBasic to do anything else with the event record afterwards, then you should set the event type to `_nullEvt` before you exit your `on event` routine. You can do this as follows:

```
eventRecPtr&.evtNum% = _nullEvt
```

This will "fool" FutureBasic into thinking that the event was a null event, and the record will subsequently be ignored.

See Also:

on event; HandleEvents

exit <label>   **statement**

Syntax:
exit "label"

Description:
This statement causes the program to jump to the statement following the indicated label. Unlike the goto statement, the **exit** <label> statement makes sure that any loops, long if blocks, local fn's, etc., that are being "jumped out" of get properly closed, so that the stack will be in a consistent state.

If an **exit** "Label" is in the main program, the "Label" must be inside the main. If an **exit** "Label" is in a local fn, the "Label" must be inside the same local fn. If an **exit** "Label" is nested inside one or more select [case] statements, the "Label" must be after the last end select.

Any exceptions to the above conditions could result in substantial penalties or crashes.

See Also:
goto; exit fn

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit fn   **statement**

Syntax:
exit fn

Description:
When used inside a `local fn` function, this statement causes the program to jump immediately to the `end fn` statement. The function then exits, passing back the value (if any) specified in the `end fn` statement. This is useful when, for example, you wish to break out of a loop and quit the function immediately; **exit fn** is a safer way to do this than using something like `goto`.

Note:
You should not use the **exit fn** statement outside of a `local fn` function.

See Also:
local fn; end fn; goto; exit <label>

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a for/next loop, while/wend conditional, do/until conditional, or in a select case structure, this statement causes the program to jump immediately to the line following the next, wend, until, or end select statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exit <structure>   **statement**

Syntax:
exit case
exit do
exit for
exit next
exit until
exit wend
exit while

Description:
When used inside a `for/next` loop, `while/wend` conditional, `do/until` conditional, or in a `select case` structure, this statement causes the program to jump immediately to the line following the `next`, `wend`, `until`, or `end select` statement. This is useful when, you wish to break out of a loop because a certain condition has been met. **exit** is a safe way to do it.

Example:

for x = 1 **to** 10
   if fn button then exit for
next x

See Also:
for; while; Do; Select Case

exp   **statement**

Syntax:
result# = **exp**(*expr*)

Description:
Returns the value of the transcendental number "e" raised to the power of *expr*. The number "e" is the base of natural logarithms, and is approximately equal to 2.718281828. The **exp** function is used extensively in probability theory and in applied sciences.
exp is the inverse of the log function; that is: **exp**(log(x)) equals x. **exp** always returns a double-precision result.

See Also:
log; log10; log2

FBCompareContainers   **function**

Syntax:

result& = **fn FBCompareContainers** (*a$$* , *b$$*)

Description:

This function returns a result that represents how container *a$$* compares to container *b$$*. If the *result&* is zero, the containers are identical. A negative result (-n&) provides the character position at which container *a$$* was found to be less than container *b$$*. A positive result give the character position where container *a$$* became greater than container *b$$*.

Result	Indicates
Negative	container *a$$* < container *b$$*
Zero	container *a$$* = container *b$$*
Positive	container *a$$* > container *b$$*

Note:

With this function, containers are evaluated by ASCII (not numeric) values.

FBCompareHandles function

Syntax:

result& = **fn FBCompareHandles** (*a&* , *b&*)

This function returns a result representing a comparison of the contents of handle *a&* with the contents of handle *b&*. If *result&* is zero, the contents of the handles are identical. If *result&* is negative, – *result&* indicates the byte position at which handle *a&* was found to be less than handle *b&*. If *result&* is positive, it indicates the byte position at which handle *a&* was found to be greater than handle *b&*. The first position in the handle is byte number 1 (not zero). Two handles which differ from the very first byte will return a positive or negative 1 as a result.

Result	Indicates
Negative	handle *a&* < handle *b&*
Zero	handle *a&* = handle *b&*
Positive	handle *a&* > handle *b&*

FBGetControlRect **function**

Syntax:
ignored = **fn FBGetControlRect**(*cHndl&*, *rect*)

Description:
Before MacOS X, we were able to extract the content rectangle of a button using the following code:

```
// rem does not work in MacOS X
dim @t,l,b,r
BlockMove button&(_btnRefNum)+_contrlrect,@t,8
```

Carbon programs do not offer the ability to peek into structures like control records. FutureBasic provides this utility function for use in any program so that you may extract the rectangle regardless of the current version of system software.

```
dim @t,l,b,r // old style rectangle
fn FBGetControlRect(button&(_thebutton), t)
```

or...

```
dim r as Rect // new style rectangle
fn FBGetControlRect(button&(_thebutton), r)
```

FBGetScreenRect   **function**

Syntax:
ignored = **fn FBGetScreenRect** (*rect*)

Description:
Before MacOS X, we were able to extract the content rectangle of a window using the following code:

```
// rem does not work in MacOS X
dim @t,l,b,r
BlockMove window(_wndPointer)+portRect,@t,8
```

This no longer works because the window pointer and the grafport have been separated into different structures. You can substitute this simple function in your programs and it will work in all supported versions of the system software.

```
dim @t,l,b,r // old style rectangle
fn FBGetScreenRect(t)
```

or...

```
dim r as Rect // new style rectangle
fn FBGetScreenRect(r)
```

FBGetSystemName$ **function**

Syntax:
*name$* = **fn FBGetSystemName$**(*nameType*)

Description:
This function returns the computer name or the user name according to the *nameType* parameter. *nameType* is one of the following
constants: _FBComputerName, _FBLongUserName or _FBShortUserName.
To make this routine available to your program, you must include the header file "Util_ComputerNames.Incl".

Example:
```
include "Util_ComputerNames.Incl"
Print "Computer Name: """;¬
  fn FBGetSystemName$(_FBComputerName);""""
Print "Long User Name: """;¬
  fn FBGetSystemName$(_FBLongUserName);""""
Print "Short User Name: """;¬
 fn FBGetSystemName$(_FBShortUserName);""""
Do
 HandleEvents
Until 0
```

files$ **function**

Syntax:

(1) For Selecting a File to Open
*fileName$* = **files$**(_FSSpecOpen, [typeListString$], [promptString$], [@]fSpecVar)
*fileName$* = **files$**(_FSRefOpen, [typeListString$], [promptString$], [@]fsRefVar)
*fileName$* = **files$**(_CFURLRefOpen, [typeListString$], [promptString$], [@]cfURLRefVar)

(2) For Selecting a Folder
*folderName $* = **files$**(_FSSpecFolder, [typeListString$], [promptString$], [@]fSpecVar)
*folderName $* = **files$**(_FSRefFolder, [typeListString$], [promptString$], [@]fsRefVar)
*folderName $* = **files$**(_CFURLRefFolder, [typeListString$], [promptString$], [@]cfURLRefVar)

(3) For Selecting a File Name and Folder where a file may be Saved
*fileName$* = **files$**(_FSSpecSave, [typeListString$], [promptString$], [@]fSpecVar)
*fileName$* = **files$**(_FSRefSave, [typeListString$], [promptString$], [@]fsRefVar)
*fileName$* = **files$**(_CFURLRefSave, [typeListString$], [promptString$], [@]cfURLRefVar)

(4) Returns the file type of the last file returned by the files$(open) functions (see option 1 choices)
*fileType$* = **files$**

Description:
This function is an optional step in the general process to select, open, read/write and close files. It provides three basic functionalities for selecting files and folders and one function to return a file type. These basic functions are:

(1) Ask the user to select a file to open
(2) Ask the user to select a folder
(3) Ask the user to provide a file name and select a folder where a file may be saved.

Within each of the three options the programmer may return a reference to the file as a variable of type FSSpec, FSRef or CFURLRef. The reference allows the programmer to work with the file or folder chosen by the user.

**files$** prompts the user via standard Navigation Services dialogs to select an existing file and/or existing folder (and provide a name if the save option is used). If the user selects a file/folder, then the file's name is returned in *fileName$*, and a reference to the file (either FSRef, FSSpec or CFURLRef) is returned for the programmer's use. If the user cancels the dialog, then the function returns an empty (zero-length) string and the reference does not contain a valid value.
The types of files that appear in the dialog may be limited by specifying up to four file types in *typeListString$*. For example, if "TEXTPICT" is passed in typeListString$, then only files of type "text" and type "PICT" will be available for selection. If *typeListString$* is an empty string, or the parameter is omitted, then all file types will be available for selection.

(4) Returns the file type of the last file returned by a files$(open) function, as a 4-character string. In some cases this will not return a value: If the user clicked "cancel" in response to the last File Open dialog, or if the files$(open) function has never yet been executed or the file doesn't have a file type (which is common for modern MacOS X files), then the **files$** function returns an empty (zero-length) string. In some cases it's useful to express the file type as a 4-byte long integer rather than as a string. Use the mki$ function and the cvi function to convert between these two forms. Apple has recommended use of Uniform Type Identifiers to replace Type/Creator. See Apple's "Introduction to Uniform Type Identifiers" for more information.

Note:
Using **files$** option 1 to select a file does not actually open the selected file. Use the open statement if you need to open the file.
The reference returned is only valid while your program operates and not after it quits. The reference should not be saved to refer to a file/folder at later date. If you need to keep track of a file's location over time, create and save an alias record for the file. The NavDialog() function is a more versatile alternative to files$(). For instance it can create sheets.

See Also:
Appendix A - File Object Specifiers; NavDialog

fill   **statement**

Syntax:
fill *h* , *v*

Description:
This statement fills the area around pixel coordinates (*h,   v*) (in the current output window) with the current color and pen pattern. All the contiguous pixels which have a color equal to the original color at (*h,   v*) are included in the fill.

Example:
This little program paints a red ring:
color = _zBlack
circle 110,110,100
circle 110,110,80
color = _zRed
fill 100, 15

See Also:
color; circle

FinderInfo   **function**

Syntax:

Move all waiting items to arrays or simple variables
```
countVar = maxAcceptableentries
```
action = ¬
   FinderInfo(*countVar%, nameVar$, typeVar&, dirRefNumVar%*)

Find out how many items are waiting to be picked up
```
countVar = 0
action = ¬
```
   FinderInfo(*countVar%, nameVar$, typeVar&, dirRefNumVar%*)

Pick up an indexed item from the list
```
countVar = negativeIndex
action = ¬
```
   FinderInfo(*countVar%, nameVar$, typeVar&, dirRefNumVar%*)

Gather a list of file spec records
```
countVar = maxAcceptableentries
action = ¬
```
   FinderInfo(*countVar%, @FSSpec[(array)], @OSType&[(array)], dirRefNumVar%*)

Clear the list
fn ClearFinderInfo

Description:

If the user launched your application by double-clicking a document icon, or by dragging document icon(s) to your application's icon, or by selecting document icon(s) and then selecting "Open" or "Print" from the Finder's "File" menu, then you can use the **FinderInfo** function to determine which document file(s) were involved, and whether they should be opened or printed.

You should call **FinderInfo** once, soon after your program starts. You should also check during null events to see if additional files have been added to the list. This can take place when an `_openDoc` event is sent from another application or when the user drags a file onto the icon of your running application.

action - The result of the **FinderInfo** function is one of the following:

action constant	Description
_finderInfoOpen (0)	This file should be opened
_finderInfoPrint (1)	This file should be printed
_finderInfoErr (2)	An error occurred. One possible reason is that the program attempted to retrieve an indexed item that was out of range.

countVar - This variable is used to send a value to and receive a result from **FinderInfo**

countVar	Description
< zero	An index into the list. The first item is -1, the next is -2, etc.
zero	The return value will be placed in countVar. It will be the total number of items available. This is reset to zero when you call Fn ClearFinderInfo.
> zero	In this case, countVar indicates the maximum number of entries that your program can accept. If you dimension an array to hold 10 elements, then the maximum would be 11 (10 + element zero). If you use 1, then the information can be placed in simple variables.

The parameters for **FinderInfo** are used to both send and receive values. In order to send a value of "1" for the count, you must first set the variable, then check it on return.
```
count% = 0
action = FinderInfo(count%,fName$,fType&,vRefNum%)
print "There are" count%" files in the queue."
```

nameVar$$	This must be a "short string" simple variable or array element.

	If you specify a maximum greater than 1 in countVar%, then you must specify an array element in nameVar$, and the array must be dimensioned at exactly 31 characters per string. The names of the documents are returned into consecutive elements in the array, starting at the element you specify. If you specify 1 in countVar%, then you can use a simple string variable for nameVar$, dimensioned to at least 31 characters. The name of the document is returned in this variable.
typeVar&	This must be a long integer simple variable or array element. If you specify a maximum greater than 1 in countVar%, then you must specify an array element in typeVar&. The 4-byte document type codes are returned into consecutive elements in the array, starting at the element you specify. If you specify 1 in countVar%, then you can use a simple long integer variable for typeVar&. The type code of the document is returned in this variable.
dirRefNumVar%	This must be a short integer simple variable or array element. If you specify a maximum greater than 1 in countVar%, then you must specify an array element in dirRefNumVar%. The directory reference numbers for the documents are returned into consecutive elements in the array, starting at the element you specify. If you specify 1 in countVar%, then you can use a simple short integer variable for dirRefNumVar%. The directory reference number for the document is returned in this variable. A document's directory reference number indicates what directory the document is in. This will either be a volume reference number (in which case the document is in the volume's root directory), or a working directory reference number.

Note:
Before your application can support Finder-launched documents, you need to set up certain special tags in the "Info.plist" file in your application's bundle.

See Also:
on FinderInfo

fix   **function**

Syntax:

wholeNum# = **fix** (*expr#*)

Description:

This function returns a whole number representation of *expr#* (it strips off digits to the right of the decimal point). Although **fix** always returns an integer, the number it returns is considered to be a double-precision floating-point value.

See Also:

frac; int

FlushWindowBuffer **statement**

Syntax:
FlushWindowBuffer [*wNum* | *_FBAutoFlushOff* | *_FBAutoFlushOn*]

Description:
Under MacOS X, all drawing to a window is intercepted and stored ("buffered") by the Window Server. The Window Server normally transfers the drawing to the screen only when your program executes a `HandleEvents` statement. You can force an early update with **FlushWindowBuffer**.

If *wNum* is omitted, or is 0, the current output window is flushed by the MacOS X Window Server. If *wNum* is non-zero, window *wNum* is flushed.

The default behavior of the FutureBasic runtime is to flush the current output window each time a `print` statement is performed. You can control that behavior: **FlushWindowBuffer** `_FBAutoFlushOff` will turn off the automatic flushing. You still can force the flushing for a specific window with **FlushWindowBuffer** `wNum`. **FlushWindowBuffer** `_FBAutoFlushOn` will restore the automatic flushing.

The **FlushWindowBuffer** command has no effect unless the program is running under MacOS X. Because of coalesced updates in MacOS X 10.4 Macho-O binaries, this does nothing if called too soon (0 or 1 tick) after the previous call.

Example:

```
// In this example, there is no HandleEvents,
// and so FlushWindowBuffer is needed to
// make the drawing visible under MacOS X.
window 1
plot 0,0 to 500,500
FlushWindowBuffer

until fn button // wait for mouse-down
```

fn <userFunction>   **function**

Syntax:

[*result* =] **fn** functionName[(*param1* [,*param2* ...])]

Description:

Executes the user function specified by functionName, and optionally returns a numeric or string result. The user function must be one which was defined or prototyped at an earlier location in the program. A user function is defined using local fn statement. A user function is prototyped using the def fn <protoType> statement. If the user function returns a value, you can use **fn <userFunction>** as part of a numeric or string expression, as in this example:

count% = 3 * **fn** NumFish%(x) + 7

If the user function does not return a value, then you should use **fn <userFunction>** as a standalone statement.

If the function definition includes a list of parameters, then you must provide the same number of parameters (in param1, param2, etc.) when you call the function, and the parameters that you pass must be of compatible types. The compatible types are summarized here (not all of these are available for all kinds of functions; see the individual description of local fn):

Formal variable type (in FN definition)	Compatible types (in Fn call)
signed/unsigned byte (var`; var``)	Any numeric expression[1,2]
signed/unsigned short integer (var%;var%`)	Any numeric expression[1,2]
signed/unsigned long integer (var&;var&`)	Any numeric expression[1,2]
pointer variable (p As Pointer [To unType])	A record variable, or a long-integer expression[6]
single/double precision floating point (var!; var#)	Any numeric expression[2]
string variable(var$)	Any string expression[3]
address reference (@adr&; @p As Pointer [To unType])	Any variable (of any type), or a long-integer expression preceded by "=".[4]
array declaration (tableau[suffixe](dim1[,dim2...]))	Base array element (arr[suffix](0[,0...]))[5]

Notes:

1. Non-integer values are rounded to integers before being moved into integer or pointer variables.

2. If you pass a numeric value that's outside the range of the formal variable type, you may get an unexpected result, or you may get an overflow error.

3. If you pass a string value that is longer than the maximum size of the formal variable, the string will be truncated.

4. If you specify a variable here, the variable's address will be copied into the formal parameter (addr& or p). If you specify a long-integer expression preceded by "=", then the value of that expression is copied into addr& or p.

5. The array must be a numeric or string array (not an array of records). All of the array's elements are accessible to the function. Any changes that are made to the array's elements within the function will also affect the array outside of the function. Note that if the array specified in the formal fn definition has a different type or different dimensions from the array you pass when you call the function, you may get unexpected results, or even a crash. (Be sure you know what you're doing before you try this!)

6. If you specify a record variable here, the record's address will be copied into the formal parameter (p). If you specify a long-integer expression, then the value of that expression is copied into p.

If the function definition does not have a list of parameters, then you must not include any parameters (nor parentheses) when you call the function.

In most cases, the parameters that you specify in **fn <userFunction>** are "passed by value." That means that the user function receives a private copy of the parameter's value; if the function changes that copy, it doesn't affect the value of the parameter that was used in the fn call.

In a few cases, the parameters that you specify in **fn <userFunction>** are "passed by reference." That means that the user function receives the address of the parameter that you specified. If the function changes the contents at that address, it will affect the value of the parameter you passed. Parameters are passed by reference when you use the following kinds of formal parameter declaractions in the function definition:

- pointer (p as pointer [to someType]). (Parameter is passed by reference if you specify a record variable when you call the function.)
- address reference (@addr&; p as pointer [to someType]). (Parameter is passed by reference if you specify a variable when

you call the function.)

- array declaration (`arr[suffix](dim1[,dim2...])`)

You can also pass the address of a variable or array by using the `varptr` function (`varptr(var)` or `@var`) when you call the function (if you do this, then specify a long integer variable or a pointer variable as the formal parameter in the fn definition). This is another way to give the function direct access to the memory comprising the variable or array, allowing it possibly to change its value.

Note that there is no way to pass the contents of a record directly to a function. To give the function access to a record's contents, either pass the record by reference, or pass the record's address directly (passing `varptr(recVar)` or `@recVar` when you call the function).

A **fn <userFunction>** call may appear anywhere below the place where the function is defined (or prototyped). It can appear in the "main" scope of the program, or inside other functions. It may even appear inside the very function that it is calling--this allows you to implement so-called "recursive" functions (functions which call themselves).

See Also:
local fn; def fn <prototype>

fn <toolbox> **function**

Syntax:

result = **fn** ToolboxFunctionName [*modifiers*] ¬
    [([{ *#addrExpr1&* | *arg1* }[,{ *#addrExpr2&* | *arg2* }...]])]

Description:

This function executes a Toolbox function as defined in Inside Macintosh. A Toolbox function (as opposed to a Toolbox procedure) returns a value.

ToolboxFunctionName must be the name of a Macintosh Toolbox function. FutureBasic recognizes the names of hundreds of Toolbox functions and procedures; advanced programmers can also use the toolbox statement and the TBALIAS statement to add new Toolbox function/procedure names.

The use of the *modifiers*, *addrExpr&* and *arg* parameters is identical to their use in the call <toolbox> statement. See the description of the call <toolbox> statement for more information.

See Also:

call <toolbox>

for   **statement**

Syntax:
for *indexVar* = *firstValue* **to** *lastValue* [**STEP** *stepValue*]
   [*statementBlock*]
next [*indexVar*]

Description:
The **for** statement marks the beginning of a "for-loop," which must end with a next statement. A for-loop is useful when you want to repeat the execution of a block of statements for a particular number of times. This is what happens when a for-loop is encountered:

1. The value of *firstValue* is assigned to *indexVar* (*indexVar* must be a simple numeric variable).
2. The statements in *statementBlock* are executed. *statementBlock* can contain any number of statements, possibly including other for-loops (but note that any for-loop that's "nested" within *statementBlock* should not use the same *indexVar* as the "outer" for-loop).
3. The value of *indexVar* + *stepValue* is assigned to *indexVar*. (If you omit the **STEP** *stepValue* clause, then incremental value defaults to 1.)
4. The new value of *indexVar* is compared with *lastValue*, to see whether the loop should be repeated:
   If *stepValue* is positive, then repeat the loop (go to Step 2) if *indexVar*<=*lastValue*.
   If *stepValue* is negative, then repeat the loop (go to Step 2) if *indexVar*>=*lastValue*.

For example, consider this loop:

```
for n = 3 to sqr(x!) STEP 2
  :
next
```

In the above, the sqr function is called after each iteration of the loop. Assuming that the value of x! doesn't change within the loop, we are needlessly recalculating the same sqr value at each iteration. It would be much faster to do it this way:

```
sqrx! = sqr(x!)
for n = 3 to sqrx! STEP 2
  :
next
```

Here the sqr function is called only once.

Implementation changes:
A design mistake in FutureBasic version 4 and earlier, apparently inherited from Applesoft BASIC, has been corrected that made for/next loops always execute at least once. Compatibility with legacy FutureBasic code can be obtained by overriding a special predefined constant as shown below.

```
dim as long j
  for j = 1 to 0
  print "Never get here"
next

override _forLoopsAlwaysExecuteAtLeastOnce = _true

  for j = 1 to 0
  print "Get here" // legacy FutureBasic behavior
next

override _forLoopsAlwaysExecuteAtLeastOnce = _false

  for j = 1 to 0
  print "Never get here"
next
```

Example:
Sometimes it's useful to exit a for-loop "early," after some condition within *statementBlock* has been met. The standard way to do this is to

use `exit for`.

```
for p = 1 to maxStrings
   long if strArray$(p) = searchString$
      found = _zTrue
      theIndex = p
      exit for 'force early exit from loop
   end if
next
```

Note:

The `while` and `do` statements provide other useful kinds of loop structures.

See Also:

while; do; exit for

frac   **function**

Syntax:
fractionValue# = **frac** (*expr*)

Description:
This function returns the fractional portion of the floating-point expression given by *expr*. If *expr* is negative, then the value returned by **frac** will also be negative.

Example:
print frac (78245.1096)
program output:
0.1096

See Also:
fix; int

get preferences   **statement**

Syntax:
get preferences *prefFileName$, prefRecord*

Description:
This statement locates the preference file *prefFileName$* in the preferences folder at: ~/Library/Preferences and loads the contents of the file into the preferences record *prefRecord*.

Example:
// The example assumes the preference file has been created with the example on the Put Preferences help page.
begin record prefsRecord
dim as   Str31   name
dim as   SInt32 aNumber
end record

dim as prefsRecord gMyPref

get preferences "MyPrefs", gMyPref

print gMyPref.name
print gMyPref.aNumber

do
HandleEvents
until (gFBquit)

See Also:
put preferences; kill preferences; menu preferences

get window   **statement**

Syntax:
get window *wndNum , [@]windowRefVar*

Description:
This statement obtains an opaque reference to the window specified by *wndNum*. The reference value is returned into *windowRefVar*, which must be a WindowRef variable. Most Toolbox functions that deal with windows take a WindowRef argument.

Note:
The inverse is `fn Wptr2WNum()` which takes a WindowRef argument and returns the corresponding FB wndNum.

See Also:
window statement; window function

GetProcessInfo   **function**

Syntax:
GetProcessInfo *index%, processName$* [*, PSN*]

Description:
A "Process" is something that is currently running on your computer; this includes, but is not limited to things like applications, control strip extensions, and background applications.

The index parameter in this call indicates which process is to be queried. An index value of -1 means that the front process is used. This is generally the running FutureBasic application that you created.

Index values of zero or higher represent running processes. You may climb this list, examining processes as you go, until the process name comes back as a null string. At that point, you have exhausted the system's list of processes and you can quit searching.

The process serial number is an 8 byte value (2 long integers) that holds a unique value which cannot be used by any other concurrent process. You create a process serial number as follows:

dim psn **as** ProcessSerialNumber

The following example shows how to display a list of running processes.

```
dim indx&
dim ProcessName$
dim psn as ProcessSerialNumber
GetProcessInfo -1,ProcessName$
print "My name is:""";ProcessName$;"""
print " indx","0x-------PSN------ ","Process Name"
indx& = 0
do
  GetProcessInfo indx&,ProcessName$,psn
    long if ProcessName$[0]
      print indx&,"0x";hex$(psn.highLongOfPSN);
      print hex$(psn.lowLongOfPSN),ProcessName$
    end if
  inc(indx&)
until len(ProcessName$) == 0
```

See Also:
SendAppleEvent

globals   **statement**

Syntax:
globals *"filename1"* [, *"filename2"* . . .]

Description:
This statement behaves identically as the `include` statement. The keyword **globals** is maintained for backwards compatibility with earlier versions of FutureBasic. To make your program easier to read, you may typically use the **globals** statement to include files which define global variables, constants, record structures, etc., while using the `include` statement to include functions, etc. However, the **globals** statement and the `include` statement are completely interchangeable.

See Also:
include

gosub **statement**

Syntax:
gosub { *lineNumber* | *"statementLabel"* }

Description:
Executes the subroutine located at the indicated line number or statement label. The subroutine should include a `return` statement; `return` causes execution to continue at the statement following the **gosub** statement.
gosub is an outdated method for executing routines; it's generally a better idea to encapsulate your routines in a `local fn` function. However, there are a couple of possible advantages to using **gosub**:

- Routines called using **gosub** may execute somewhat faster than local functions.
- You can create a "private" subroutine inside a local function, and use a **gosub** within that local function to call the subroutine. The variables used in the subroutine will have the same scope as the local function. This may be a good way to execute certain repetitive tasks within the local function.

Example:
Subroutines can be executed in a "nested" fashion; i.e., one subroutine may call another. FutureBasic keeps track of where each `return` statement should "return" to.
print `"First line."`
gosub `"sub1"`
print `"Fifth line."`
end
`"sub1"`
print `"Second line."`
gosub `"sub2"`
print `"Fourth line."`
return
`"sub2"`
print `"Third line."`
return
program output:
First line.
Second line.
Third line.
Fourth line.
Fifth line.

Note:
It is not possible for a **gosub** statement inside a local function to jump to a subroutine located outside of that function; it is similarly not possible for a **gosub** statement in "main" to jump to a subroutine located inside a local function. **gosub** programming is not recommended, a **local fn** should be used as a replacement.

As of FutureBasic version 5.4.8, the new implementation of gosub/return does not support optimized compilation. If your code uses gosub/return and needs optimization, you will have to replace every subroutine by an ordinary local fn.

See Also:
return; fn <userFunction>; local fn

goto **statement**

Syntax:
goto { *lineNumber* | *"statementLabel"* }

Description:
Causes program execution to continue at the statement at the indicated line number or statement label. The target statement must be within the same "scope" as the **goto** statement (i.e., they must both be within the "main" part of the program, or they must both be within the same local function). Also, you should never use **goto** to jump into the middle of any "block" statement structures (such as `for...next`, `select...end select`, `long if...end if`, etc.).

goto is sometimes useful in the "main" part of the program, to branch around certain structures. However, excessive use of **goto** can lead to code that is difficult to read and maintain. The use of **goto** is generally discouraged; FutureBasic's other branching and looping structures offer a better solution.

See Also:
local fn; gosub; for; while; do; long if; select case

HandleEvents **statement**

Syntax:
HandleEvents

Description:
HandleEvents performs a number of important functions that affect the user's experience. It examines the system event queue, as well as FutureBasic's internal event queues, to see whether any recent events have occurred for your program, that have not yet been handled. If any such events are found, **HandleEvents** removes them from the queue and responds to them. **HandleEvents** also performs the important function of turning over control to the Process Manager. The Process Manager oversees the execution of all processes on your Macintosh; once it has control, the Process Manager may allow another application to run for a short time before returning control to your application.
HandleEvents responds to some kinds of user actions by calling functions that you have designated in your program. It responds to other kinds of user actions in predetermined, "automatic" ways.

"Automatic" responses by `HandleEvents`

- Allows the Process Manager to bring another process to the front, if the user has selected it in the Applications menu or clicked in one of its windows.
- Opens menus and tracks selection, if user has clicked on the menu bar.
- Activates an inactive window, if the user has clicked on the window's structure region (e.g. its title bar). (This action is inhibited if the window's `_keepInBack` attribute is set.)
- Handles dragging & resizing of the active window.
- Performs "standard" handling of mouseclicks and keystrokes in the currently active edit field (if any).
- Highlights & tracks various objects when they're clicked (e.g. buttons, window close box, etc.)
- For any window that requires updating: redraws all buttons, scrollbars, edit fields and picture fields (unless the window's `_noAutoClip` feature is set). Also redraws certain parts of the window's structure region.
- If the user presses cmd-period, and no `on break fn` function has been identified, then **HandleEvents** displays a dialog asking whether the user wants to stop or to continue. If the user elects to stop, FutureBasic then calls your designated `on stop fn` function (if any), and then halts the program.

NOTE: You can inhibit and/or alter these responses by trapping low-level events (especially the `_mButDwnEvt` event) in a system event-handling function. See below for more details.

"Programmed" responses by `HandleEvents`
There are many kinds of common user actions, such as button clicks and menu selections, which you will want to explicitly handle with program statements. When you write a function that is to handle events of a certain type, you designate it as an event-handling function by executing statements like `on dialog fn <functionName>`, or `on menu fn <functionName>`. Once you've designated your event-handling function(s) this way, **HandleEvents** will examine recent user actions to determine whether any of them are of the kind that your function(s) can handle. If any such events are found, **HandleEvents** calls the appropriate event-handling function once for each such event. See the descriptions of the various `on <eventType>` statements, to learn what types of user actions can be handled.
If you haven't identified a function to handle a certain class of user actions, then **HANDLEVENTS** just ignores actions of that class. For example, if you have not identified any function with the `on dialog` statement, then **HANDLEVENTS** will ignore button clicks and other similar actions. **HandleEvents** will still perform the "automatic" responses listed above, however.

Intercepting system events
There may be times when you need greater control over how **HandleEvents** responds to certain events. For example, you may want to inhibit or alter some of the "automatic" responses that **HandleEvents** normally performs. To do this, you should designate one of your functions as a "system event-handling function," by using the `on event` statement. Once you've designated such a function, **HandleEvents** calls that function first , before it executes any of its "automatic" responses and before it calls any of the other event-handling functions you may have designated. **HANDLEVENTS** either passes a system event to your function (if there's an event in the queue), or it passes a "null event" to your function (if there are no events in the queue).
After your system event-handling function returns, **HandleEvents** continues to handle that same event, unless it was a null event. Depending on what the event was, **HandleEvents** may perform some of its "automatic" responses, or it may call another one of your event-handling functions. If you don't want **HandleEvents** to continue handling the event after your system event-handling function exits, then you need to "trick" FutureBasic into thinking that the event was a null event. You do this by executing a line like the following, in your system event-handling function, after you've handled the event:
`theEvent&.evtNum% = _nullEvt`

where `theEvent&` is a pointer to the event record.

In order to provide the user with snappy response to actions, and to share execution time with other processes, your program should call **HandleEvents** as often as possible. Most well-designed programs contain a "main event loop" which calls **HandleEvents** repeatedly for as long as the program is executing, allowing **HandleEvents** to call the various event-handling functions as events occur.

Flushing events

Flush the event queue using: **call FlushEvents**(_everyEvent, 0)

Events trapping

The default behaviour of HandleEvents is now to block (i.e. not return) until an event is dispatched.

In FutureBasic version 4, the default behaviour was to return after every 2 ticks (1 tick = 1/60 s) even when no events occurred. In effect a steam of null events was generated 30 times a second, allowing polling but wasting CPU time.

This (typically unwanted) activity could be suppressed by:

poke long event - 8, 0xFFFFFFFF // no null events, thanks

In FutureBasic version 5 onwards, it is no longer necessary to suppress null events in this way. The old FutureBasic version 4 behaviour, if required, can be restored as shown:

poke long event - 8, 2 // null events every 2 ticks, like FutureBasic version 4
do
  fn PollRegularlyForSomething
  HandleEvents
until gFBQuit

See Also:

on <eventType> statements; dialog statement/function; menu function; mouse; event function; tekey statement/function

HandShake   **statement**

Syntax:
HandShake *portID*, *handshakeType*

Description:
Sets the handshaking parameter for the open serial port specified by *portID* (which can be either _modemPort or _printerPort). The handshaking parameter determines how i/o operations will be negotiated when your program subsequently writes to or reads from the specified port. *handshakeType* can take any of the following values:

Constant	Description
_none (0)	No handshaking. This is the default value.
_CTS (1)	CTS (Clear To Send) hardware handshaking. When the computer is ready to send a block of data, it sets the CTS signal. The computer must then wait for a DTR signal from the other device, before sending the data.
_DTR (2)	Sets the DTR (Data Terminal Ready) signal on.
_DTRToggle (-2)	Sets the DTR signal off.
_XON (-1)	XON/XOFF software handshaking. Each device sends an XON character when it's ready to receive data, and sends an XOFF character if its buffer fills up.

To determine the best handshaking format for a device, see the device's manual.

Note:
You must open the serial port (use the open "C"... statement) before executing the **HandShake** statement.
Powerbooks require an initialization setting of _none to preset the serial port properly.

See Also:
open "C"; lof

hex$ **function**

Syntax:
*hexString$* = **hex$**(*expr*)

Description:
This function returns a string of hexadecimal digits which represent the integer value of *expr*. The returned string will consist of either 2, 4 or 8 characters, depending on which of `defstr byte`, `defstr word` or `defstr long` is currently in effect. Note that if the value of *expr* is too large to fit in a hex string of the currently selected size, the string returned by **hex$** will not represent the true value of *expr*.
In FutureBasic, integers are stored in standard "2's-complement" format, and the values returned by **hex$** reflect this storage scheme. You need to keep this in mind when interpreting the results of **hex$**, especially when *expr* is a negative number. For example: **hex$**`(-3)` returns "`FD`" when `defstr byte` is in effect; "`FFFD`" when `defstr word` is in effect; and "`FFFFFFFD`" when `defstr long` is in effect.

Note:
To convert a string of hex digits into an integer, use the following technique:
`intVar` = **val&**(`"&H"` + `hexString$`)
`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. Byte variables can handle a `hexString$` up to 2 characters in length; short-integer variables can handle a `hexString$` up to 4 characters in length; long-integer variables can handle a `hexString$` up to 8 characters in length.

See Also:
oct$; bin$; DEFSTRBYTE/word/long; val&

if   **statement**

Syntax 1:

if *expr* **then** { *dest1* | *statement1* [: *statement2* ...] } ¬
    [**else** { *dest2* | *statement3* [: *statement4* ...] }]

Syntax 2:

if *expr*
    [*statementBlock1*]
[**else** | **xelse**
    [*statementBlock2*]]
end if

Description:

Conditionally executes one or more statements, or jumps to an indicated line, based on the value of `expr`. If *expr* is evaluated as "true" or as nonzero, then the program either jumps to the line indicated by *dest1*, or it executes *statement1*, *statement2*, etc. If *expr* is evaluated as "false" or as zero, and you've included the `else` clause, then the program either jumps to the line indicated by *dest2*, or it executes *statement3*, *statement4*, etc. Each of the *statement*'s can be any executable statement except a "block" statement such as `for`, `while`, `do`, etc.

expr can be either a numeric expression, a "logical" expression, or a string. A logical expression normally contains "data comparison" operators, and can be evaluated as being either "true" or "false." Here are some examples of logical expressions:

```
x! > 19.7
myName$ = "RICK"
6*7 <= 42
```

In FutureBasic, numeric expressions and logical expressions are interchangeable. When a numeric expression is used in the context of a logical expression, then it's considered "true" if it's nonzero, or "false" if it's zero. For example:

if `x+3` **then** `beep`

Here, the `beep` will be executed if and only if `x+3` is not zero.

When a logical expression is used in the context of a numeric expression, then it's evaluated as -1 if it's true, or as 0 if it's false. For example:

```
found = (fileName$ = seekName$)
```

Here, if `fileName$` equals `seekName$`, the value -1 is assigned to `found`; otherwise, `found` is assigned a value of 0.

You can use the operators `and`, `or`, `not` within *expr*. Note, however, that these three are considered to be arithmetic operators, not logical operators. This can lead to some unexpected results if you're not careful. For example, this expression:

`firstNumber&` **and** `secondNumber&`

may evaluate to zero (false), even when both `firstNumber&` and `secondNumber&` are each nonzero (true). When you wish to use `and`, `or`, or `not` in the context of a logical expression, you should use operands which always evaluate either to -1 or to 0. For example:

`firstNumber& <> 0` **and** `secondNumber& <> 0`

This expression behaves "logically," because (`firstNumber&<>0`) is always -1 or 0; and likewise (`secondNumber&<>0`) is always -1 or 0.

The *expr* can also be a string. When a string is used in the context of a logical expression, it's evaluated as "true" if and only if the length of the string is greater than zero.

Note:

The **if** statement is a one-line structure. To create a conditional structure spanning multiple lines, use the `long if` statement.

Use caution when comparing floating point values to zero or to whole numbers. The following expression may not evaluate as expected:

```
if x# = 1
```

In this statement, the compiler compares the value in x# to an integer "1". Since SANE and PPC math both use fractional approximations of numbers, the actual value of x#, though very close to one, may actually be something like 0.99999999 and therefore render unexpected results.

See Also:

long if; and; or; not; select case

inc   **statement**

Syntax:

inc (*intVar*)

intVar ++

numericVariable += IntegerValToAdd

Description:

This statement increments *intVar* by 1; that is, it adds 1 to the value in *intVar*, and stores the result back into *intVar*. *intVar* must be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. If *intVar* is already at the maximum value for its variable type, then **inc** (*intVar*) will cycle it back to its minimum value.

Example:

inc (x&)

and...

x& ++

and...

x& += 1

...are equivalent to:

x& = x& + 1

The following expressions are also equivalent.

x& = x& + 100
x& += 100

Note:

The += syntax may not be used for arrays of strings, containers, or records (though it may be used in the concatenation of simple strings or containers). Where arrays are involved, only numeric values may take advantage of this syntax. This syntax is valid for values other than 1.

See Also:

dec; inc long/word/byte

inc long/word/byte   **statement**

Syntax:
inc {long|word|byte} (*addr&*)

Description:
This statement increments the long integer, short integer or byte which begins at the specified address in memory; that is, it adds 1 to the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its maximum possible value, then the statement will cycle it back to its minimum value.

Example:
inc long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) + 1
Also:
inc word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) + 1

See Also:
inc; dec long/word/byte

inc long/word/byte   **statement**

Syntax:
inc {long|word|byte} (*addr&*)

Description:
This statement increments the long integer, short integer or byte which begins at the specified address in memory; that is, it adds 1 to the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its maximum possible value, then the statement will cycle it back to its minimum value.

Example:
inc long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) + 1
Also:
inc word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) + 1

See Also:
inc; dec long/word/byte

inc long/word/byte   **statement**

Syntax:
inc {long|word|byte} (*addr&*)

Description:
This statement increments the long integer, short integer or byte which begins at the specified address in memory; that is, it adds 1 to the value in memory and stores the result back into the addressed location. If the long integer, short integer or byte is already at its maximum possible value, then the statement will cycle it back to its minimum value.

Example:
inc long (myAddr&)
...is equivalent to:
poke long myAddr&, **peek long**(myAddr&) + 1
Also:
inc word (myAddr&)
...is equivalent to:
poke word myAddr&, **peek word**(myAddr&) + 1

See Also:
inc; dec long/word/byte

include   **statement**

Syntax:
include { *"<path>"* | *alisResID* }

Description:
When the compiler encounters an **include** statement, it looks for a text file indicated by *<path>* or *alisResID*, reads the FutureBasic statements contained in that file, and effectively inserts those statements into the source code stream. *<path>* can be either a full or partial pathname; if you use a partial pathname, it's assumed to be relative to the folder containing the "parent" source file (i.e., the source file that has the **include** statement). *<path>* may refer either to a text file, or to an alias file whose target is a text file. *alisResID* must be the resource ID number of an "`alis`" resource whose target is a text file. This resource should exist in the resource fork of the "parent" source file.
include files may be "nested"; that is, an **include**'d file may contain references to other **include** files.

Example:
Suppose we create a file "`Assign.INCL`" which contains the following lines of text:
```
a = 3
b = 7
```
Now suppose we write a program like this:
dim `x(100)`
include `"Assign.INCL"`
`c = a + b`
When we compile this program, the result will be identical to this:
dim `x(100)`
```
a = 3
b = 7
c = a + b
```

Special treatment for C source and C header files (*.c and *.h):

include "SomeFile.c"
include "SomeFile.h"

Such files are copied to the build_temp folder. A #include statement is inserted in the translated FutureBasic code. This feature provides an alternative to #if def _PASSTHROUGHFUNCTION for mixing C with FutureBasic code.

Special treatment for C static libraries (*.a):

include "MyLib.a"

The include statement copies the library file to the build_temp folder; you must also place the name of the library file in the preferences 'More compiler options' field [this causes it to be linked]. The example below is for a library MyLib that exports one symbol (MyLibFunction).

include `"MyLib.a"`
BeginCDeclaration
```
// let the compiler know about the function
void MyLibFunction( void ); // in lieu of .h file
```
EndC
```
// let FBtoC know about the function
```
toolbox `MyLibFunction()`
```
MyLibFunction() // call the function
```

include resources "SomeFile.someextension":

The file indicated is copied from the FutureBasic source folder to the application's Contents/Resources/ directory, unless the extension is .nib in which case it is copied to Contents/Resources/en.lproj/.
This statement is the standard way to copy your sound (for example *.aiff), and image (for example *.icns) files into the application package.
Nib files are handled by this statement, as an alternative to dragging them to your project window in FutureBasic version 5.

include library

MacOS X frameworks may be specified with the 'include library' statement, which has two forms:

include library "Framework/Header.h"
include library "Framework" // optional short form, expanded internally to: include library "Framework/Framework.h"

```
// tell the compiler the framework and header
```
include library `"AddressBook/AddressBookUI.h"`
```
// tell FBtoC the functions
```
toolbox fn `ABPickerCreate() = ABPickerRef`
`...`

The effect of 'include library' is to insert the appropriate #include preprocessor directive in the translated C file, and to pass the appropriate linker command to the compiler.

The QuickTime framework is #included and linked by default; your source code does not need include library "QuickTime".

OpenGL is automatically #included and linked, if your project uses one of the relevant Headers files such as Tlbx gl.incl.

Note:
FutureBasic's Project Manager is generally a more convenient way to combine the source code from several different files. However, there are some advantages to using the **include** statement, such as the ability to conditionally include files.
FutureBasic does not allow a given file to be included more than once in the source stream. If a second **include** reference is made to a given file, that **include** statement will be ignored.

index$ **function**

Syntax:
*stringVar$* = **index$**(*element* [*, indexID*])

Description:
This function returns an element from one of the special **index$** string arrays (see the index$ statement and the clear <index> statement for information about how **index$** arrays are created). The *indexID* parameter specifies which of the **index$** arrays (0 through 9) to return an element from; if this parameter is omitted, **index$** array 0 is used. The *element* parameter specifies which element to get. If the indicated **index$** array has not yet been initialized by the clear <index> statement, or if *element* specifies an element that has not yet been assigned any value, then **index$** returns an empty (zero-length) string.

See Also:
CFIndexSort; clear <index>; index$ D; index$ I; index$ statement; indexf; mem

index$ D   **statement**

Syntax:
index$ D (*element* [, *indexID*])

Description:
This statement deletes an element from one of the special `index$` string arrays (see the index$ statement and the clear <index> statement for information about how `index$` arrays are created). The *indexID* parameter specifies which of the `index$` arrays (0 through 9) to delete an element from ; if this parameter is omitted, `index$` array 0 is used. The *element* parameter specifies which element to delete.
The **index$ D** statement does not merely assign a null string to the indicated element. Instead, it moves all of the subsequent array elements in memory, in order to fill the "gap" left by the deleted element. As shown in the diagram, this also affect the element numbers by which the moved strings are identified.
This statement is the complement of `index$ I`, which inserts an element into the array.

Example:
This illustrates the difference between deleting an element using **index$ D**, and "clearing" an element by assigning a null string to it.

INDEX$

element #	contents
4	Sandy
5	Joshua
6	Carrie

Before

element #	contents
4	Sandy
5	Carrie

After

INDEX$(5) = ""

element #	contents
4	Sandy
5	Joshua
6	Carrie

Before

element #	contents
4	Sandy
5	
6	Carrie

After

See Also:
CFIndexSort; clear <index>; index$ I; index$ function; index$ statement; indexf; mem

index$ I   **statement**

Syntax:
index$ I (*element* [, *indexID*]) = *stringExpr$*

Description:
This statement inserts a new element into one of the special `index$` string arrays, and assigns the value of *stringExpr$* to the new element (see the index$ statement and the clear <index> statement for information about how `index$` arrays are created). The *indexID* parameter specifies which of the `index$` arrays (0 through 9) to insert an element into; if this parameter is omitted, `index$` array 0 is used. The *element* parameter specifies where in the array to insert the string.

Unlike the `index$` statement, the **index$ I** statement does not merely replace the contents of the indicated element. Instead, all of the strings at position *element* and beyond are moved in memory, to open a space in which to insert the new string. As shown in the diagram, this also affect the element numbers by which the moved strings are identified.

This statement is the complement of `index$ D`, which deletes an element from the array.

Example:
This illustrates the difference between inserting an element using **index$ I**, and replacing an element using the `index$` statement.

INDEX$ I(5) = "Riley"

element #	contents
4	Sandy
5	Joshua
6	Carrie

element #	contents
4	Sandy
5	Riley
6	Joshua
7	Carrie

Before   **After**

INDEX$(5) = "Riley"

element #	contents
4	Sandy
5	Joshua
6	Carrie

element #	contents
4	Sandy
5	Riley
6	Carrie

Before   **After**

See Also:
CFIndexSort; clear <index>; index$ D; index$ function; index$ statement; indexf; mem

indexf   **function**

Syntax:

foundElement = **indexf** (*string$* [, *startElement* [, *indexID*]])

Description:

This function searches the index$ array specified by *indexID*, for a string which contains the characters specified by *string$*. The search begins at the element specified by *startElement*. If you omit the *startElement* parameter, the search begins at element zero (i.e., at the beginning of the array). If you omit the *indexID* parameter, index$ array #0 is searched.

The search is case-sensitive. If a match is found, **indexf** returns the element number of the matching element; otherwise, it returns -1. If you specify an index$ array which does not exist (because no space for it has been allocated using the clear <index> statement), **indexf** returns -1.

Example:

elementNumber = **indexf** ("Modem")

This code would find a match with the following index$ elements:

"Modem"
"Modemizing"
"my Modem"

The code would not find a match with these elements:

"modem"
"MODEM"
"horse feathers"

See Also:

CFIndexSort; clear <index>; index$ D; index$ I; index$ function; index$ statement; mem

inkey$ **function**

Syntax:
*stringVar$* = **inkey$**

Description:
This function tests whether there is a keypress event pending in the event queue (this happens if the user has pressed a key and no program statement has yet detected it). If there is such an event on the queue, **inkey$** removes the event from the queue and returns a 1-character string indicating what key was pressed. If there is no keypress event pending, **inkey$** returns a null (zero-length) string.
Note that **inkey$** is a rather old-fashioned way to check for keypresses. It will not work reliably if your program calls `HandleEvents` regularly, because `HandleEvents` also checks for keypress events and removes them from the event queue. If your program uses `HandleEvents`, then you should check for keypresses either by trapping the `_evKeyDIALOG` event (if there are no active editable text fields in the current window or if you are running in the Appearance Compliant runtime).

Note:
The pressed key character is not automatically displayed on the screen. Use the `print` statement if you want to display the character.
The value returned by **inkey$** is affected by whether the Shift key, Option key, etc. were down. However, these "modifier" keys will not generate any keypress event by themselves; they must be pressed in combination with some character key in order for **inkey$** to detect the keypress. If you want to detect whether a certain modifier key was down when an event happened, use the `event%` function (or the `_evtMeta` field of the `event` record). If you want to detect whether a modifier key is currently down, use the Toolbox function GetKeys.

See Also:
HandleEvents; dialog function; tekey$ function; on dialog

inkey$ <ioChannel> **function**

Syntax:
*stringVar$* = **inkey$**(*deviceID*)

Description:
This function reads a single character from an open serial port or an open file, and returns it as a 1-character string. *deviceID* should equal either _modemPort or _printerPort (see the open "C" statement), or should be the fileID number of an open file (see the open statement).
The function returns an empty (zero-length) string in the following situations:

- There are no characters currently in the input buffer of the specified serial port
- The end of the specified file has been reached.

Note:
This function is similar to the read# deviceID, stringVar$;1 statement. However, the read# statement generates an error if you attempt to read past the end of a file.

See Also:
inkey$; read#; open; open "C"

input   **statement**

Syntax:
input [*"prompt"*;] *var1* [, *var2* ...]

Description:
This statement displays an optional prompt in the current output window, then waits for the user to enter data from the keyboard. The entered data is displayed in the window (to the right of the prompt, if any) as the user types it. When the user finishes entering the data, the data are assigned to the variables *var1*, *var2* etc., and the program then continues execution at the next statement. This is a simple (but old-fashioned) way to let the user interact with the program.

Specifier	Description
"prompt";	If you specify this parameter, the literal string inside the quotes is displayed as a prompt. The input cursor is displayed just to the right of the prompt.
var1[,var2...]	These must be string or numeric variables (not record variables). The data that the user enters are assigned to these variables according to the rules explained below.

Variable Assignment
The **input** statement expects the user to enter data as a sequence of data items separated by commas or tabs as in this example:
```
23, green, -15.7
23 [tab] green [tab] -15.7 [cr]
```
Subsequent text will refer to the more common comma delimiter, but works also with tabs.
Each one of the comma-delimited items is assigned (after some conversion, if necessary) to a separate variable in the *var1* [, *var2* ...] list. If the user enters more items than the number of variables in the list, the extra entered items are ignored. If the user enters fewer items than the number of variables in the list, then zeros or null strings are assigned to the extra variables.
The items may undergo some conversion before being assigned to the variables:

- If the variable is a string variable, leading spaces are stripped from the item (trailing spaces are not stripped).
- If the variable is a numeric variable but the entered item is a string that can't be interpreted as a number, then 0 is assigned to the variable.
- If the variable is a numeric variable and the entered item is numeric, then it is subject to the same kinds of conversion as if the item were assigned via an assignment statement. For example, the item's value may be rounded to an integer, or excess digits of precision may be dropped.

Quoted Items
By surrounding an entered item with double quotes, the user can exercise more control over how the item is assigned to a (string) variable. FutureBasic always strips the surrounding quotes from the item before assigning the string to the variable; however:

- Leading and trailing spaces inside the quotes are preserved;
- A comma that occurs inside the quotes is considered part of the item, rather than a delimiter between items.

Therefore, if the user needs to input an item which has leading blanks, or which contains a comma, then he or she should surround the item with double quotes when typing it in. To assign an arbitrary line of entered text to a string variable without the need for enclosing quotes, use the line input statement.

edit field vs. input
input is an old-fashioned way of getting keyboard input. A more appropriate and "Macintosh-like" way to get input is to use the edit field statement.

See Also:
input#; line input; edit field; edit$ function

input# **statement**

Syntax:
input# *deviceID* , *var1* [, *var2* ...]

Description:
This statement reads text data from the open file or serial port specified by *deviceID*, and stores the data into the specified variables. The variables *var1*, *var2* etc. must be numeric, container, or string variables (not record variables).

If *deviceID* equals zero, then **input#** reads data from the keyboard. **input#** 0 , var1 [, var2...] is identical to **input** var1 [, var2 ...]. The data in the file (or coming in through the serial port) should be formatted as text items delimited by commas and/or carriage-returns. Each item is assigned to a separate variable. Some data conversion may occur during the assignment; see the input statement for more details.

If *deviceID* specifies a file, then **input#** reads a line of text from the file, beginning at the current "file mark" position (which is usually at the beginning of the line), and ending when a carriage-return character is encountered, or the end of the file is encountered, or 255 characters have been read, whichever occurs first. The file mark is then advanced to a position just past the last character read.

input# then attempts to assign each of the comma-delimited items in the text line to one of the variables (*var1*, *var2* etc.) in the variable list. If there are more items in the text line than variables in the list, the extra items are discarded. If there are fewer items in the text line than variables in the list, then zeros or null strings are assigned to the extra variables.

If *deviceID* specifies an open serial port (i.e., if its value is _modemPort or _printerPort), then **input#** behaves in a similar way, except that the concepts of "file mark" and "end of file" generally don't apply.

input# is the slowest method of reading data from disk. For greater speed, try using read# instead. (Note however that **input#** and read# generally interpret the data in the file differently.)

input# can read the data created by print#. Note, however, that if you want to create data items that are delimited by commas, you must explicitly print comma characters between them. print# does not automatically insert commas between the items it puts out.

Note:
If the file mark is already at the end of file when you execute **input#**, FutureBasic generates an "Input past end of file" error. To prevent this situation, check the value of **eof** (deviceID) before executing **input#**.

See Also:
input; line input#; open; read#; write; print#; eof

instr   **function**

Syntax:

foundPosition = **instr** (*startPos* , ¬
    targetString$* | *targetContainer$$, ¬
    searchString$* | *searchContainer$$)

Description:

This function searches for the first occurrence of *searchString$* or *searchContainer$$* within *targetString$* or *targetContainer$$*, starting at character position *startPos* . (If *startPos* is less than 1, it's treated as 1.) If a match is found, the function returns the character position (1..255) within *targetString$* or *targetContainer$$* where the match begins. If no match is found, the function returns zero. The string search is case-sensitive.

Note:

instr always returns zero in these cases:

- when *startPos* is greater than `len(` *targetString$* `)`
- when `len(` *searchString$* `)` is zero.

See Also:

indexf; mid$; left$; right$

int   **function**

Syntax:
nearestInteger& = **int** (*expr#*)

Description:
Returns the value of *expr#* rounded to the nearest integer.

Note:
int returns a "long integer" value, which means that *expr#* should be within the range -2147483648 through +2147483547. To obtain the integer part of numbers which are outside this range, use the `fix` function. (Note however that `fix` truncates the fraction part rather than rounding to the nearest integer. In general, `fix` and **int** don't return the same values.)

See Also:
fix; frac

InvalRect   **function**

Syntax:
ignored = **fn InvalRect** (*rect*)

Description:
Before Carbon became a part of the Mac toolbox, we were able to use a toolbox procedure called `InvalRect` to mark a portion of the current window as an area that needed to be refreshed during the next update. This call will not work in MacOS X (or in the Carbon version of OS 9). Our substitute, **fn InvalRect**, will work in versions 7 through X without additional coding required on your part.

kill **statement**

Syntax:
kill *path$* `[` , *refNum%* `[` , *dirID&* `]]`

Description:
This statement deletes a (closed, unlocked) file or folder from the disk. The *path$* parameter can be either a full or partial pathname; the *refNum%* parameter can be a working directory reference number, a volume reference number, a drive ID number, or zero. The *dirID&* parameter is a directory ID number of the item's parent directory. If *dirID&* is not specified, then the current `PARENTID` value (if nonzero) is used instead. **kill** resets the `PARENTID` value to zero after the item is deleted.

The easiest way to use this statement is to specify the item's name in *path$*, the volume reference number of its volume in *refNum%*, and the ID number of its parent directory in *dirID&*. For other ways to use these parameters, see Appendix A - File Object Specifiers.

Note:
You cannot delete a folder if it has any contents, nor if the folder is "open." A folder is considered "open" if your program has executed any statement which returned a working directory reference number for the folder, and has not subsequently closed it using the `close FOLDER` statement.

kill dynamic statement

Syntax:
kill dynamic *arrayName*

Description:
This statement releases the memory allocated for use by a dynamic array. The array may still be used after the **kill dynamic** statement is executed and will begin to grow once again as information is added.

See Also:
dynamic; read dynamic; write dynamic

kill field   **statement**

Syntax:
kill field *handle&*

Description:
This statement disposes of the specified handle. This means that the memory block referenced by *handle&* gets released, and the value in *handle&* is thereafter no longer a valid handle. **kill field** is commonly used with handles returned by `get field` or `read field`, but it can dispose of any kind of handle. However, you should specifically not use it to dispose of resources, regions, window controls, and other "standard" kinds of Macintosh objects which are created and managed by the MacOS. Instead, you should use the appropriate Toolbox routine (ReleaseResource, DisposeRgn, DisposeControl, etc.) to dispose of such objects.
kill field is similar to the Toolbox call `DisposeHandle`, except that it (like the `DisposeH` statement) checks for `_nil` handles and sets the *handle&* variable to zero.

See Also:
DisposeH; read field

kill picture   **statement**

Syntax:
kill picture *pictureHandle&*

Description:
This statement calls the Toolbox procedure KillPicture, which releases the memory occupied by the picture which is specified by *pictureHandle&*. This should be a handle that your program created using the `picture on`, `picture off` statements (or the Toolbox routines OpenPicture and ClosePicture). You should not use **kill picture** to release a picture handle that was created by other means (for example, a picture resource handle).

Warning: Do not use **kill picture** to kill a picture that is currently being displayed in a picture field. You must close the picture field first (using the `edit field close` statement) before calling **kill picture**.

Warning: You should make sure that *pictureHandle&* does not equal zero before calling **kill picture**. Disposing of a "nil handle" can cause problems on some older systems.

See Also:
picture statement; picture on/off; edit field close

kill preferences   **statement**

Syntax:
kill preferences *prefFileName$*

Description:
This routine locates and deletes a file with the specified name in the preference folder. If the file does not exist, this statement does nothing.
A full example of using the new PREFERENCE commands can be found on the Put Preferences reference page.

See Also:
put preferences; get preferences; menu preferences

kill resources   **statement**

Syntax:
kill resources *"resType", resID%,* [*"resType", resID%...*]

Description:
This statement will store specified resource types and IDs in an internal list. When one of these resources is encountered by the compiler, it will not be added to the built application.
There are two important reasons for a call like this. The first is that resources normally included in the runtime shells may be considered unnecessary by a programmer may easily be deleted from the finished product. The second is that multiple resource files may be used and unnecessary resources may be eliminated programmatically.

Example:
To prevent a picture resource with an ID of 8001 from being added to the compiled application, the syntax would be:
kill resources "PICT", 8001

See Also:
resources

left\$ and left\$\$ **function**

Syntax:
subString\$ = **left\$**(*string\$* , *numChars*)
subContainer\$\$ = **left\$\$**(*container\$\$* , *numChars*)

Description:
This function returns a substring or subcontainer consisting of the leftmost *numChars* characters of *string\$* or *container\$\$*. If *numChars* is greater than the length of *string\$* or *container\$\$*, then **left\$** returns the entire string or container. If *numChars* is zero, then **left\$** returns an empty (zero-length) string.

Note:
You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:
c\$\$ = c\$\$ + **left\$\$**(a\$\$,10)
Use:
c\$\$ += **left\$\$**(a\$\$,10)

See Also:
mid\$ function; right\$

len   **function**

Syntax:

stringLength = **len** (*string$* | *container$$*)

Description:

This function returns the number of characters contained in *string$* or *container$$*. In the case of an empty string, zero is returned.

Note:

To determine the maximum number of characters that can be put into a string variable, use:

   sizeof (stringVar$) - 1

The maximum number of characters allowed in a container is theoretically 2 gigabytes, but is normally limited by available memory.

See Also:

sizeof

let statement

Syntax:

1. [**let**] *var* = *expr*
2. [**let**] *var;length* = *address&*

Description:

The **let** statement assigns a value to the variable *var*, replacing whatever value *var* had before. Note that the **let** keyword is optional.

- If you use Syntax 1, the value of *expr* is assigned to *var*:
- If *var* is a numeric variable, then *expr* can be any numeric expression; if *expr* is outside the range or precision that can be stored in *var*, then the expression will be appropriately converted.
- If *var* is a pointer variable, then *expr* can be _nil (zero), or another pointer variable of the same type, or any valid address expression.
- If *var* is a Handle variable, then *expr* can be _nil (zero), or another Handle variable of the same type, or any valid address expression whose value is a handle.
- If *var* is a string variable, then *expr* can be any string expression. You should make sure that the length of *expr* does not exceed the maximum string size that will fit into *var*.
- If *var* is a "record" declared using **dim** var **as** recordType, then *expr* must be a record variable of the same type as *var*.

If you use Syntax 2, then *length* bytes are copied into *var*, from the memory location starting at *address&*. The *length* parameter must be a static integer expression (i.e., it cannot contain any variables). Note that FutureBasic does not check whether *length* actually equals the size of *var*. If *length* is too small, an incomplete value will be copied into *var*; if *length* is too big, data will be copied into addresses beyond *var*'s location in memory (this can be dangerous).

See Also:

dim; begin record; BlockMove; BlockFill; Constant declaration statement

line input   **statement**

Syntax:
line input [@(*col*, *row*) | %(*h*, *v*)] [*"prompt";*] *stringVar$*

Description:
This statement behaves similarly to the input statement, except that the entire line of text entered by the user (including any commas, quotes and leading spaces) is stored into the single variable *stringVar$*. See the input statement for a description of the parameters that precede *stringVar$*.

Note:
input and **line input** are considered rather "old-fashioned" ways to interact with the user. Programs with good user-interface design usually utilize Edit Fields instead.

See Also:
input; line input#; edit field

line input# statement

Syntax:
line input# *deviceID , stringVar$*

Description:
This statement reads a line of text data from the open file or open serial port specified by *deviceID*, and stores the data into the string variable *stringVar$*.

If *deviceID* equals zero, then **line input#** reads data from the keyboard. **line input#**`0,stringVar$` is identical to **line input** `stringVar$`.

If *deviceID* specifies a file, then **line input#** reads a line of text from the file, beginning at the current "file mark" position (which is usually at the beginning of the line), and ending when a carriage-return character is encountered, or the end of the file is encountered, or 255 characters have been read, whichever occurs first. **line input#** then assigns the entire string of characters to *stringVar$*. the file mark is then advanced to a position just past the last character read.

If *deviceID* specifies a serial port (i.e., if its value is `_modemPort` or `_printerPort`), then **line input#** behaves in a similar way, except that the concepts of "file mark" and "end of file" generally don't apply.

Note that **line input#** is similar to `input#`, except that special characters such as commas, quotes and leading spaces are not interpreted as data item delimiters, but instead are copied directly into *stringVar$*.

Note:
If the file mark is already at the end of the file when you execute **line input#**, FutureBasic generates an "Input past end of file" error. To prevent this situation, check the value of `eof(deviceID)` before executing **line input#**.

See Also:
input#; line input; eof; open

loc   **function**

Syntax:
result = **loc**(*deviceID*)

Description:
This function returns one of two things, depending on the value of *deviceID*.

- If *deviceID* is the file ID number of an open file, the function returns the current location of the file mark as an offset from the beginning of the current record. For example, if **loc**(fileID) returns zero, the file mark is located at the beginning of the record. The file mark indicates where in the file the next input or output operation will occur.
- if *deviceID* specifies an open serial port (i.e., if its value is _modemPort or _printerPort), then the function indicates the status of the carrier signal. If returns _zTrue if the carrier is detected, or _false if the carrier is not detected.

Note:
You can use **loc** along with rec to determine the exact location of the file mark within the file.

See Also:
rec; record; open; lof; HandShake; open "C"

local   **statement**

Syntax:
[**clear**] **local** [**MODE**]

Description:
This statement is an alternative way to indicate the beginning of the scope of a local function. If used, it must appear somewhere above the local fn statement. All non-global variables which are declared between the **local** statement and the local fn statement have a scope local to the function. Adding the **clear** and/or **MODE** keywords has the following additional effects:

- The **clear** keyword causes all of the function's local variables and arrays (except parameter-list variables) to be initialized to zeros, null strings or empty records, each time the function is called. Otherwise, the variables will have unpredictable initial values. You can accomplish the same effect by adding the **clear** keyword to the local fn statement.
- The **MODE** keyword prevents the use of global variables within the function. That is, all variables inside the function will be local variables, even those which have the same names as global variables. This is a good practice when you're writing a function that you might wish to use in a number of different projects, because it removes the possibility of the function's local variables being misinterpreted as globals.

Note:
dim is the only kind of statement that you should put between the **local** statement and the local fn statement. Executable statements placed between **local** and local fn will never be executed.
You cannot declare any of the the variables in the function's parameter list using a dim statement after the **local** statement.
A compiler preference allows you to fill local fns with $A5A5 for debugging. With this item checked, all functions that do not begin with **clear local** have every variable filled with this value. It's a great debugging tool.

See Also:
local fn; end fn; dim; begin/end globals

Syntax:
[**clear**] **local fn** *functionName* [(*arg1* [, *arg2* ...])]
    [*statementBlock*]
end fn [= *expr*]

Description:
This statement marks the beginning of an FutureBasic local function. The end of the local function is marked by the end fn statement. A local function is a named collection of statements which can be accessed and executed as a unit by referring to the function's name (see the fn <userFunction> statement). All variables and arrays referenced in a local function (except those explicitly declared as "global") are local to the function, which means they do not have any influence outside of the function; any identically-named variables which appear outside of the function are actually different variables, and occupy a different place in memory, than the function's local variables. (An exception to this rule occurs when an array is listed as one of the function's formal parameters; see more about this below.) When your program "calls" (executes) a local function, you can pass data into the function by means of its parameter list (also called its argument list), and you can receive a value back from the function by means of its return value. Local functions allow you to encapsulate complex programming tasks; they're a fundamental and extremely powerful programming construct.
In addition to marking the beginning of the function, the **local fn** statement also declares the function's name, the data type of its return value (if any), and the number and types of its input parameters (if any). A local function can be placed anywhere in the program, except inside another local function; you should also not place a local function inside a "block" structure such as long if...end if, etc. The statements in *statementBlock* can contain anything except the following:

- A local statement;
- Another local function.

Adding the **clear** keyword causes all of the function's local variables and arrays (except the parameter variables *arg1*, *arg2* etc.) to be initialized to zeros, null strings or empty records, each time the function is called. Otherwise, the local variables and arrays will have unpredictable initial values. (You can accomplish the same effect by using the local statement with the clear keyword; see the local statement for more information.)
The functionName must be unique; that is, it must be different from the name used in any other **local fn**, def fn using statement anywhere else in the program. If the function is to return a value, then you should specify the data type of the return value by including an appropriate type-identifier suffix at the end of functionName. For example, a local function which is to return a string value might be declared as follows:
   local fn FullName$(idNum&)
The default data type of a function's return value is "long integer"; if the function is to return a long integer value, you can either include the "&" type-identifier suffix or omit it. No type-identifier suffix should be appended to functionName if the function does not return a value.
In order to execute the statements in statementBlock, your program must "call" the function using the fn <userFunction> statement. The fn <userFunction> statement can appear anywhere in your program, as long as the function it calls is either defined (using the **local fn** statement) or prototyped (using the def fn <prototype> statement) somewhere above the fn <userFunction> statement. This restriction is required in order to allow your program to compile; however, the actual order of execution of your program's statements is not affected by where you place your **local fn**'s.

Function Parameters
Each of the parameters *arg1*, *arg2* etc. can have any of the following forms:

Parameter form	Description
SimpleVar	A simple numeric or string variable. Cannot be a record variable, a record field nor an array element.
ptrVar As Pointer [To unType]	(See below.)
@longVar&	longVar& is a simple long-integer variable.
@ptrVar As Pointer [To unType]	(See below.)
tableau(dim1[,dim2...])	array is a numeric or string array (not an array of records), and dim1,

The parameters in the **local fn** statement are called the function's "formal arguments." They must not be global variables. You should not declare the formal argument variables in a dim statement; they are implicitly declared by the **local fn** statement.
When your program calls the function, the arguments passed to it in the fn <userFunction> statement are called the "actual arguments."

They must match the function's formal arguments in number, and they must be of "compatible types" (see `fn <userFunction>` for more information). Each time the function is called, values are assigned to its formal arguments as follows:

- If the formal argument is a simpleVar, the value of the actual argument is copied into simpleVar.
- If the formal argument is of the form `ptrVar as pointer [to someType]`, then the actual argument should be either a record variable or a long-integer expression. In the first case, the record's address is copied into ptrVar; in the second case, the long integer's value is copied into `ptrVar`.
- If the formal argument is of the form `@longVar&`, or `@ptrVar as pointer [to someType]`, then the actual argument must either be a variable (possibly a record variable), or a long integer expression preceded by "=". In the first case, the variable's address is copied into `longVar&` or `ptrVar`. In the second case, the value of the long integer expression is copied into `longVar&` or `ptrVar`.
- If the formal argument is `array(dim1 [,dim2 ...])`, then the actual argument must be the base element of an array of the same type, which has the same number of dimensions. The base element is the element in which all subscripts are set to zero. The entire array is then accessible to the local function, and (important!) any changes made to the array's elements within the function will persist after the function exits.

If the local function has no parameters, you should omit the parentheses after `functionName`.

Passing an Array of Unknown Size
Sometimes it's useful to write a local function which operates on an array passed in its parameter list, without knowing in advance the size of the passed array. For example, suppose you wish to write a function which sorts the elements of a long integer array, and you want it to work regardless of the declared size of the passed array.

When you declare an array as a formal parameter, FutureBasic ignores the value of the array's first declared dimension in the **local fn** statement. For example, suppose we have a function defined like this:

```
local fn  SetElements(anArray&(1,7), max&)
  'Set each element to 1492 in the array:
 for i&  = 0 to max&
  for j = 0 to 7
      anArray&(i&,j) = 1492
  next
 next
end fn
```

When we pass a long integer array to `fn SetElements`, the passed array can have any size as its first declared dimension, as long as it has a second dimension declared as 7. For example:

```
dim arrayOne&(1250,7), arrayTwo&(465,7)
fn SetElements(arrayOne&(0,0), 1250)
fn SetElements(arrayTwo&(0,0), 465)
```

Within the function, we can safely manipulate elements in the array as long as the subscripts we use don't exceed the declared dimensions of the actual array that was passed. Thus, in `fn SetElements`, we can set the first subscript in `anArray&` to values much greater than 1, even though `anArray&` was "declared" with dimensions (1,7).

Note: you do not have equal freedom with the second, third, etc. dimensions of the parameter array. If the array is multi-dimensional, the second and subsequent dimensions must be declared with the same values in both the "formal" array parameter (in the **local fn** statement) and the external `dim` statement that declares the actual passed array.

Returning a Value
If you specify an *expr* in the **end fn** statement, the function will "return" the value of *expr*. This can be any expression which is compatible with the type-identifier suffix (if any) that appears in `functionName`. When your function "returns" a value, it means that you can reference the function (using `fn <userFunction>`) as part of a string or numeric expression, and the function's return value will be substituted in the expression. For example:

```
    maxPuppets = 6 * fn storeCount%(x)
```

Here, if `fn storeCount%(x)` returns a value of 7, then the value 42 will be assigned to `maxPuppets`.

The Lifespan of Local Variables
The memory space for a function's local variables is reserved when the function is called. This memory is released after the **end fn** statement is executed. Therefore, you should never make reference to a local variable's address after the function has finished executing; in particular, you should never pass a local variable's address back to the routine that called the function. For example:

```
'DON'T do THIS!
local fn myFunction&(x,y,z)
  dim r#
 r# = sqr(x*x + y*y + z*z)
end fn = @r#
    rAddr& = fn myFunction&(x,y,z)
```

After the preceding is executed, `rAddr&` points to an area of memory (the old address of `r#`) which is no longer reserved, and which should not be used.

On the other hand, it is permissible to pass a local variable's address into another local function. This works because the first local function has not yet finished executing when it calls the second local function. Therefore, the memory space holding the first function's local variables

is still reserved intact while the second function executes.

```
'THIS IS OKAY:
local fn FirstFn
 dim 255 myString$
   'Pass address of local var into another fn:
  fn SecondFN(@myString$)
end fn
   :
local fn SecondFn(strAddr&)
  BlockMove @gString$, strAddr&, len(gString$)+1
end fn
```

Recursive functions

You can have several functions executing simultaneously, in the sense that one function can call a second function, which can call a third, and so on. If you design your function calls in such a way that a function can call a function that is already executing, then you have a "recursive function." The most obvious (but not the only) example of a recursive function is any function which calls itself. When that happens, we say that two (or more) "instances" of the function are executing simultaneously.

In FutureBasic, every currently executing "instance" of a local function maintains its own private set of local variables, and they don't interfere with the local variables of any other executing instance of that function. Calling a function recursively is very much like calling a "different" function which just happens to contain exactly the same program lines.

Although recursive functions may at first seem like a bizarre concept, they are perfectly acceptable, and often very useful. For example, here is a short program which prints all the permutations of the characters contained in a given input string; note that `FNpermute_r` calls itself. It would be very difficult to write such a program without using recursive functions.

```
'Function prototypes:
def fn Permute(aString$)
def fn permute_r(prefix$, suffix$)
input "Enter a word: "; theWord$
fn Permute(theWord$)
end
local fn Permute(aString$)
    'Prints all permutations of the letters in aString$
  fn permute_r("", aString$)
end fn
local fn permute_r(prefix$, suffix$)
   'Prints all permutations of prefix$+suffix$
   'that start with prefix$
  long if suffix$ = ""
  print prefix$
 xelse
  for i = 1 to len(suffix$)
      'Move the i-th letter of suffix$ over to newprefix$:
      newprefix$ = prefix$ + mid$(suffix$, i, 1)
      newsuffix$ = left$(suffix$,i-1) + mid$(suffix$,i+1)
      'Now print all permutations that
      'start with newprefix$
   fn permute_r(newprefix$, newsuffix$)
  next
   end if
end fn
```

Returning Multiple Values

The `end fn` statement can return only a single numeric or string expression. But many times, it's useful to have a local function which can return more than one value. The way to accomplish this is through the function's parameter list. If you give the function access to the address of some external variable or array, then the function can alter the contents at that address, effectively modifying the value of that variable or array.

There are three ways to pass an address to your function:

- If you pass an entire array (using the `array(dim1 [,dim2 ...])` syntax in the formal parameter list), then your function implicitly has access to the address of the passed array. Any changes you make to the array's elements inside your function are actually made to the external array, so the changes persist after the function exits.
- If you use the `@var` syntax in the function's formal parameter list, and specify a variable when you call the function, then the variable's address is copied into var. Your function can then modify the contents at that address.
- You can explicitly pass any address into a long integer or `pointer` formal parameter.

See Also:
fn <userFunction>; local; @fn; def fn <prototype>

locate   **statement**

Syntax:
locate   *h,v*

Description:
This statement moves the pen position in the current output window to text column *h* and text row *v*, based on the current font family and font size. The pen's horizontal placement is based on an "average" character width; you can't count on this position to encompass exactly *h* characters unless you are using a mono-spaced font. **locate** 0 , 0 places the pen at the upper-leftmost character position in the window.

See Also:
csrlin; pos

lof   **function**

Syntax:
numRecords = **lof**(*deviceID* [, *recordLength*])

Description:
If *deviceID* is the ID number of an open file, **lof** returns the total number of records in the file. If *deviceID* equals `_modemPort` or `_printerPort` (and the specified device is open), **lof** returns the total number of records currently in the device's input buffer. If there is a "partial" record at the end of the file (or input buffer), it is included in the count.

The returned record count is based on the record length given in *recordLength*, if it's specified. If this parameter is omitted, **lof** uses the record length that was specified in the `open` statement when the file or the port was opened. If *recordLength* is omitted and no record length was specified in the `open` statement, a default record length of 256 is used.

To determine the total number of bytes in the file or in the serial input buffer, use **lof**`(deviceID,1)`.

See Also:
record; rec; loc; open

log   **function**

Syntax:
naturalLog# = **log**(*expr*)

Description:
Returns the natural logarithm of *expr*. The natural logarithm uses the transcendental number "e" as its base. **log** always returns a double-precision result.
log is the inverse of the `exp` function. That is: **log**`(exp(x))` equals `x`.

Note:
To find the logarithm of *expr* for an arbitrary base `n`, use this formula:
`theLog#` = **log**`(expr)`/**log**`(n)`

See Also:
exp; log10; log2

log2 **function**

Syntax:
base2Log# = **log2** (*expr*)

Description:
Returns the base-2 logarithm of *expr*. **log2** always returns a double-precision result.

See Also:
log; log10

Log10   **function**

Syntax:
commonLog# = **log10** (*expr*)

Description:
Returns the common logarithm of *expr*. the common logarithm uses "10" as its base. **log10** always returns a double-precision result.

See Also:
log; log2

long color **statement**

Syntax:
long color *bluePart* , *greenPart* , *redPart* [, *foregroundFlag*]

Description:
Sets the foreground color or background color for the current output window. Each of the three color components can range from 0 (darkest) to 65535 (lightest), and they combine to make any conceivable shade. If you set *foregroundFlag* to `_zTrue`, or omit the parameter, then **long color** sets the foreground color. If you set *foregroundFlag* to `_false`, then **long color** sets the background color.

Note:
long color does not immediately change the appearance of the window. If you set the foreground color, the new color will appear the next time you draw text or a QuickDraw shape (it won't affect the color of anything that's already drawn). If you set the background color, the new color will appear the next time you erase all or part of the window (for example, with the `cls` statement).
You can use the Toolbox procedures GetForeColor and GetBackColor to find out the current foreground or background color for the current window.

See Also:
color; pen

long if   **statement**

Syntax:
[**long**] **if** *expr*
    [*statementBlock1*]
[**xelse**
    [*statementBlock2*]]
end if

Description:
The **long if** statement marks the beginning of an "if-block," which must be terminated with the end if statement. The *expr* can be either a logical expression (such as: personCount>17), a numeric expression, or a string. A numeric expression is counted as "true" if it evaluates to a nonzero value. A string is counted as "true" if its length is greater than zero.
If *expr* is "true," then only the statements in *statementBlock1* are executed, and execution then continues at the first statement after end if. if *expr* is "false," then only the statements in *statementBlock2* (if any) are executed, and execution then continues at the first statement after end if.
statementBlock1 and *statementBlock2* may contain any number of executable statements, and may even include other "nested" if-blocks.

Note:
To conditionally execute just a single statement, consider using the if statement instead. To conditionally execute statement blocks based on more complex conditions, use the select case statement.
Use caution when comparing floating point values to zero or to whole numbers. The following expression may not evaluate as expected:
long if x# = 1
In this statement, the compiler compares the value in x# to an integer "1". Since SANE and PPC math both use fractional approximations of numbers, the actual value of x#, though very close to one, may actually be something like 0.99999999 and therefore render unexpected results.

See Also:
if; and; or; not; select case

lprint **statement**

Syntax:
lprint [@(*col*, *row*)|%(*h*, *v*)] [*itemList*]

Description:
This statement sends a line of text to the printer. The @(*col*, *row*) and %(*h*, *v*) options specify where on the page the line should be printed (see the print statement); if you don't specify one of these, the line is printed at the current pen position of the printing grafPort (this is usually just under the previously-printed line).
The **lprint** statement is equivalent to the following group of lines:
route _toPrinter
print [@(col,row)|%(h,v)] [itemList]
route _toScreen
lprint is inefficient if you are printing many lines to a page, because it reroutes the output each time **lprint** is executed. In such cases, it's better to execute a sequence of print statements, with the entire sequence preceded by a single route _toPrinter statement and followed by a single route _toScreen statement.

Note:
You should execute clear lprint or close lprint in order to cause the printed page to be put out, after you have finished printing to it.

See Also:
print; clear lprint; close lprint; route

MaxWindow   **statement**

Syntax:
MaxWindow *h , v*

Description:
Sets a limit on how large the user can make the current output window. After **MaxWindow** is executed, the user will not be able to drag the window's size wider than *h* pixels nor taller than *v* pixels (these dimensions refer to the window's "content region"; they don't include the "structure region" (frame) of the window). If the current output window's dimension(s) exceed *h* and/or *v*, **MaxWindow** will not shrink it immediately. The window will shrink the next time the user drags its grow box.

See Also:
MinWindow

maybe **function**

Syntax:

trueOrFalse = **maybe**

Description:

This function is a special random number generator that returns either _zTrue (-1) or _false (0), with equal probability. Before your program calls **maybe** for the first time, you should execute the randomize statement to "seed" the random number generator.

See Also:

rnd; randomize

mem function   **function**

Syntax:

info = **mem** (*expr*)

Description:

Specify one of the following values in *expr* to get information about the application's heap memory or index$ array:

expr	Value	Value returned by mem(expr)
_maxAvail	-1	Returns the size (in bytes) of the largest available block of contiguous free memory in the heap. Also forces all purgeable resources to be removed from memory, and may move relocatable memory blocks.
_freeBytes	-2	Returns the total number of free bytes in the heap. This memory may be spread over several (non-contiguous) free blocks. This number always greater than or equal to the number returned by mem(_maxAvail).
indexID + _numElem	10-19	The element number of the highest element in this array that has been assigned a string, plus 1. Equals zero if no element has a string assigned to it. This number is also affected by the `index D` and `index I` statements.

See Also:

index$ statement; indexf; index$ I; clear <index>

menu   **function**

Syntax:
selectedMenu = **menu**(_menuID)
selectedItem = **menu**(_itemID)

Description:
If you have designated a menu-event handling routine in your program (using the `on menu` statement), then **menu**(_menuID) returns the menu number, and **menu**(_itemID) returns the item number, of the menu item most recently selected by the user. Your menu-event handling routine should check these values each time it's called.
To give the user continual access to the menu bar, your program should execute `HandleEvents` periodically. `HandleEvents` checks for recent clicks on the menu bar, and responds by opening the menu and tracking the mouse's movement. Finally, `HandleEvents` calls your menu-event handling function if the user selects a menu item.

Menu Numbers
With the exception of the Apple Menu, the Help Menu and the Application Menu, the menus on the menu bar are numbered in increasing order from left to right. In most cases, they will be numbered consecutively starting with 1. You use the `menu` statement to assign menu numbers to the menus your program creates.
The number of the Apple Menu equals the constant `_AppleMenu`. If your program adds new items to the Apple Menu, the **menu** function can detect when the user selects those items. Other items in the Apple Menu are handled by the Finder, and your program can't detect when the user selects those. You use the `apple menu` statement to add items to the Apple Menu.
The number of the Help Menu equals the constant `_kHMHelpMenuID`. If your program adds new items to the Help Menu, the **menu** function can detect when the user selects those items. Other items in the Help Menu are handled by the Help Manager, and your program can't detect when the user selects those. To add new items to the Help Menu, you use the Toolbox routines HMGetMenuHandle and AppendMenu (see the `menu` statement for an example of how to do this).
Your program can't directly detect an item selected in the Application Menu; that's handled by the Finder. However, your program can detect when another application has been brought to the front. See the `dialog` function for more details.
Your program can also detect when the parent item that pops out a hierarchical menu is selected. This is turned on and off by a constant in the file named UserFloatPrefs which is located in the User Libraries folder. One (undesirable) side effect of enabling this feature is that a menu grayed by setting its title to a disabled state will produce menu events in inactive items. To use the old method (this is the default state) of ignoring hierarchical items, remark out the line in that reads...
`_FBEnableMenuChoice = _zTrue`
To enable the new feature, remove the remarks and allow the constant to be defined. This declaration is in the file named UserFloatPrefs which is located in the User Libraries folder.
Hierarchical menus have their own menu numbers that are different from their "parent" menu's number. You can use the **menu** function to detect selection in hierarchical menus that your program creates.
Pop-up menus are considered to be window controls (like buttons), and are therefore not detected by the **menu** function.

Item Numbers
Menu items are numbered consecutively from top to bottom, starting with 1. Note that a grey dividing line between items has its own item number, even though it can't be selected. It's important to remember this when assigning and interpreting item numbers. Items within hierarchical menus are also numbered consecutively starting with 1.
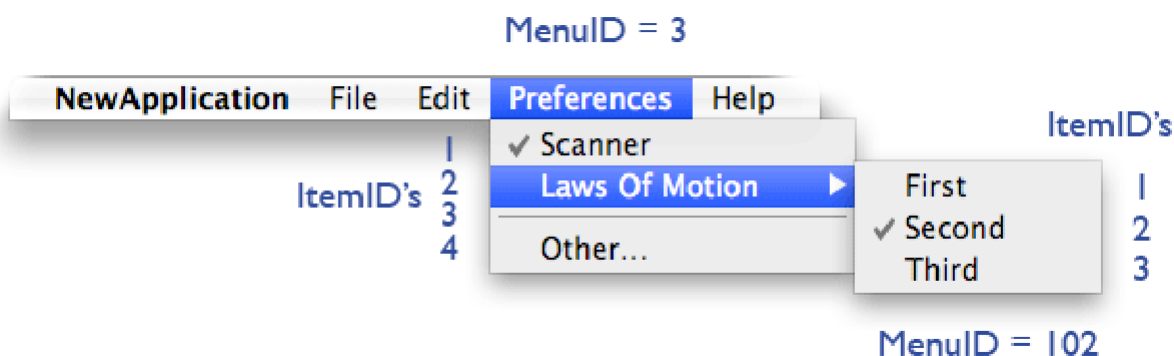
See Also:
menu statement; apple menu; HandleEvents; on menu

menu   **statement**

Syntax:
To create or alter a menu:
menu *menuID* `,` *itemID* `,` *state* `[,` *string$* `[,` *commandID* `]]`
To unhighlight the menu bar: **menu**

Description:
Use this statement to do any of the following:

- Add a new menu to the menu bar.
- Enable or disable a menu.
- Add a new item to an existing menu.
- Enable or disable a menu item.
- Add or remove a checkmark from a menu item.
- Change the text of a menu item.
- Specify a hierarchical submenu to be attached to a menu item
- Unhighlight the menu bar.

To add a new menu to the menu bar:

- Set the *menuID* parameter to a number which is not already in use by an existing menu. Use a number in the range 1 through 31.
- Set the *itemID* parameter to zero.
- Set the *state* parameter either to `_enable` or `_disable`, depending on whether you want the menu to be initially enabled or dimmed (you can change this state later).
- Set the *string$* parameter to the text that you want to appear as the new menu's title.
- Set the *commandID* parameter to the desired commandID for the menu item.

This creates a new empty menu (see below to learn how to add items to the menu). The value you choose for *menuID* will determine the new menu's position on the menu bar; menus are automatically positioned from left to right in increasing order of their *menuID* numbers. Almost always, you'll want to assign your menus consecutive numbers starting with 1.
To enable or disable (dim) an existing menu:

- Set the *menuID* parameter to the ID number of an existing menu.
- Set the *itemID* parameter to zero.
- Set the state parameter to `_enable` or `_disable`.
- Do not specify the *string$* parameter (if you do, all the menu's items will go away!)

To add a new item to an existing menu:

- Set the *menuID* parameter to the ID number of an existing menu.
- Set the *itemID* parameter to a positive number which is not being used by any other item in the menu. This number determines the item's position in the menu; items are numbered consecutively from top to bottom starting with 1. If you "skip" an item, then either a blank space or a grey dividing line will appear in that position, depending on what version of System software you're using. Note that a grey dividing line between items has its own item ID number. You can create a grey dividing line by using the meta character "-" in the *string$* parameter.
- Set the *state* parameter to `_enable`, `_disable` or `_checked`, depending on what you want the item's initial state to be (you can change this state later).
- Set the *string$* parameter to the text that you want to appear in the item. Note that when you're adding a new item, certain special characters in *string$* won't appear in the item text but have other special meanings. Consult the "Meta Characters" table below.

To enable, disable (dim), or checkmark an existing item:

- Set the *menuID* and *itemID* parameters to an existing item in an existing menu.
- Set the *state* parameter to `_enable`, `_disable` or `_checked`. Note that setting state to `_enable` or to `_disable` will remove any existing checkmark on the item.

To change the text of an existing item:

- Set the *menuID* and *itemID* parameters to an existing item in an existing menu.

- Set the *string$* parameter to the desired text. Note that when you change the text of an existing item, all the characters in *string$* will appear in the item text, and none will be interpreted as "meta characters."

To specify a hierarchical submenu to be attached to a menu item:

- Set the *menuID* parameter to the ID number of an existing menu; this is the "parent" menu which will contain the submenu.
- Set the *itemID* parameter to a positive number which is not being used by any other item in the menu. This is the "parent" item to which the submenu will be attached.
- Set the *state* parameter to the ID number of the submenu. This should be a number in the range 32 through 235 which is not being used by any other menu.
- Set the *string$* parameter to a string which ends with these two characters: `"/" + chr$(&1B)`.

Note: The above procedure will attach the submenu to the parent menu item, but it doesn't install the submenu. To install the submenu, you also need to call the Toolbox procedure `InsertMenu`. See the examples below.

To unhighlight the menu bar:

- Execute the **menu** statement without any parameters. The menu bar is automatically highlighted every time the user selects a menu item, and it remains highlighted until your program unhighlights it. By unhighlighting the menu bar, your program lets the user know that the action associated with that menu item has completed.

Meta Characters

The characters in this table have special meanings when they appear in the *string$* parameter when you're adding a new menu item. Note that when you change the text of an existing item, all the characters in *string$* will appear in the item text, and none will be interpreted as meta characters. The exception to this rule is a string that starts with a minus sign. The minus sign is a flag used by most menu definitions do draw a divider line. If your item needs to contain a minus sign, you may still display the item properly if you put a space before the character.

Meta character	Effect
;	When it appears by itself, ";" creates a grey dividing line. When it appears as a delimiter in a list (e.g., "item1;item2"), each of the items in the list becomes a separate menu item. You can use this fact to add several new menu items with just a single Menu statement.
(	When it appears in an item that follows a semicolon, "(" initially disables (dims) the item.
/	The character following "/" becomes a command-key equivalent for the menu item. Or, if the character following "/" is Chr$(&1B), it indicates that this menu item has a submenu.
!	When "!" appears in an item that follows a semicolon, the character following "!" is displayed as a "mark" on the left side of the menu item.
-	Creates a grey dividing line. Any other characters in the item string are ignored.
<	The letter following "<" is interpreted as a text attribute to be applied to the menu item. Use one of the following letters: B=Bold O=Outlined U=Underlined I=Italic S=Shadowed

Creating Hierarchical Menus

You can use the following function to add a new menu item and attach a new hierarchical menu to it. You should set `childMenuID` to some number in the range 32 through 235 which is not being used by any existing menu.

```
local fn MakeHierMenu(parentMenuID,parentMenuItem,¬
    itemString$,childMenuID)
  title$ = "!"+chr$(childMenuID)+itemString$ + "/" + chr$(&1B)
  menu parentMenuID,parentMenuItem,,title$
  call InsertMenu(fn NEWMENU(childMenuID,""), -1)
end fn
```

After you have called `fn MakeHierMenu`, you can use the **menu** statement to add new items to the hierarchical menu (set the *menuID* parameter to the value of `childMenuID`).

Items in the Apple Menu

You should use the `apple menu` statement to add items to the top of the Apple Menu. After adding these items, you can use the **menu** statement (with the *menuID* parameter set to `_appleMenu`) to alter the items (for example to enable or dim them).

Items in the Help Menu

You can add items to the bottom of the Help Menu by getting a handle for the Help Menu and then calling the `AppendMenu` procedure. You also need to find out the item number of your first Help item for use by your menu event handler (any existing items are handled by the Help Manager):

dim as int `OSErr, @ firstCustomHelpItem`
dim as Handle `@ hmHandle`

```
#if carbonlib
    OSErr = fn HMGETHELPMENU(hmHandle, firstCustomHelpItem)
#else
    OSErr = fn HMGETHELPMENUHANDLE(hmHandle)
    firstCustomHelpItem = fn COUNTMITEMS(hmHandle)+1
#endif
call AppendMenu(hmHandle, "My Help")
```

After adding items to the Help Menu, you can use the **menu** statement (with the *menuID* parameter set to `_kHMHelpMenuID`) to alter the items.

Note:
Do not use the **menu** statement to add new items to the Help Menu; use `AppendMenu` instead.

Removing Menus
Call the `DeleteMenu` procedure to remove a menu created by the **menu** statement:
call DeleteMenu(menuID)
This may cause other menus in the menu bar to slide to the left to fill the gap; however, they still retain their original menu ID numbers.

Removing Menu Items
To remove all the items from a menu you created, use the **menu** statement, specifying zero in the *itemID* parameter, and specifying a menu title in the *string$* parameter.
To remove an individual item, use the `GetMHandle` function and the `DelMenuItem` procedure:
call DELMENUITEM(**fn** `GETMHANDLE(menuID), itemID)`
Note that this will renumber any items below the deleted item, as they move up to fill in the gap. Menu item numbers are always numbered consecutively starting with 1.

Example:
The following lines create a complete menu which also contains a hierarchical menu. This example makes use of the `MakeHierMenu` function defined above.
```
menu 3,0,_enable,"Game"
menu 3,1,_enable,"See High Scores/H"
menu 3,2,_enable,"Reset High Scores/R"
menu 3,3,_disable,"-"
fn MakeHierMenu(3,4,"Scenarios",100)
'Items in hierarchical menu:
menu 100,1,_checked,"Level 1"
menu 100,2,_enable,"Level 2"
menu 100,3,_enable,"Level 3"
'It takes two menu statements to include a
'special character like "!" in the text:
menu 3,5,_enable,"dummy"    'This adds the item
menu 3,5,_enable,"Play Now!"    'This alters the item
```
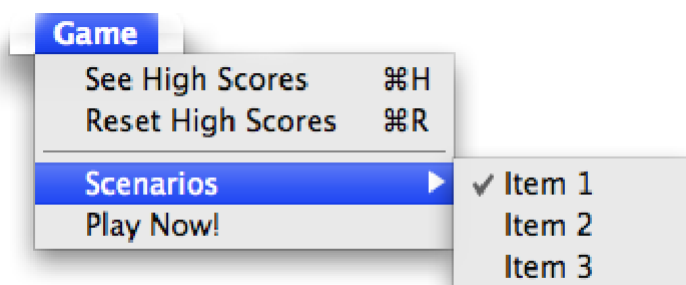

Contextual Menus
At about the time that the Appearance Manager came on to the scene, programmers began to use contextual menus. A contextual menu appears when the user clicks at a specific area in a window while holding down the control key. When this type of action takes place, you will receive (Appearance Manager Runtime only) a `dialog(0)` message of `_cntxtMenuClick`. `dialog(_cntxtMenuClick)` will be the window number of the window. At this point you may need to react by showing a menu under the cursor.
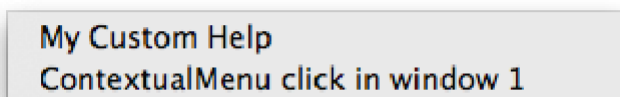
The following function builds and displays a menu and might be called in reaction to a contextual menu click.
local fn `DoContextMenu( wNum as long )`
  dim @ `selectionType` **as long**

```
  dim @ menuID as short
  dim @ menuItem as short
  dim mHndl as Handle
  dim err as OSStatus
  dim helpItemString as Str255
  mHndl = fn NEWMENU(255, "X")
  long if mHndl
    InsertMenu( mHndl, -1 )
    AppendMenu( mHndl, ¬
      "ContextualMenu click in window" + str$( wNum ) )
    helpItemString = "My Custom Help"
    err = fn CONTEXTUALMENUSELECT( mHndl, ¬
      #gFBTheEvent.where, _nil, _kCMHelpItemNoHelp, ¬
      @helpItemString, #_nil, @selectionType, ¬
      @menuID, @menuItem )
    /*
    In this function, we don't actually do anything with the
    selectionType, menuID, or menuItem returned, but we
    could react to it right here
    */
    DisposeMenu( mHndl )
  end if
end fn
```

Resource Menus
Resource menus are no longer supported.

See Also:
menu function; on menu fn; apple menu

menu preferences   **statement**

Syntax:
menu preferences *menuID* , *itemID*

Description:
This command, used in conjuction with other menu statements, moves the preference item to the standard MacOS X location in the Application menu. Prior to calling the menu preference statement, a menu must be created with the preference item. When the menu preference statement executes, it not only moves the preference item to the Application menu but removes the preference item from the menu where it was created (which is traditionally the edit menu but it could be a different menu). When the preference item is selected during program execution, the menu choice is converted to the specified *menuID* and *itemID* for use by the program. See the example below.

History:
Prior to OSX, the preference menu item was located at the bottom of the Edit menu. In OSX, the preference menu item is located in the application menu. The Menu Preference statement was created to reduce coding and allow FutureBasic programmers to use the same menus in OS 9 and MacOS X without changing their FutureBasic source code.

Notes:
[1] The *itemID* must be the last item of the edit menu (or wherever the preferences item is created).
[2] This command assumes menus are created programmatically using either FB's Menu statement or toolbox calls. A menu bar and menus created as a Nib with Apple's Interface Builder does not need a Menu Preferences statement because the preference menu item is in the nib.

Example:

```
local fn DoMenu
  if ( menu( _menuID ) == 2 and menu( _itemID ) == 10 ) then Print "Pref item selected"
end fn

apple menu "About MyApp"

menu 1, 0, _enable,"File"

edit menu 2                          // Create Edit menu with standard copy, paste etc. items
menu 2, 10, _enable,"Preferences/,"   // Create Preference as item #10 to make sure it is last on the
edit menu
menu preferences 2, 10               // Move item #10 from menu #2 ( preference item ) to the
application menu

on menu fn DoMenu

do
   HandleEvents
until ( gFBQuit )
```

See Also:
put preferences; get preferences; kill preferences

mid$ and mid$$ **function**

Syntax:
*subString$* = **mid$**(*string$* , *startPos* [, *numChars*])
subContainer$$* = **mid$$**(*container$$, *startPos* [, *numChars*])

Description:
This function returns a substring or subcontainer of *string$* or *container$$*, consisting of characters which begin at position *startPos* within *string$* or *container$$*. If you specify *numChars*, then a maximum of *numChars* characters are returned; otherwise, all the characters from *startPos* to the end of *string$* or *container$$* are returned. If *startPos* is less than 1, then it's treated as 1. If *startPos* is greater than the length of *string$* or *container$$*, then a null (zero-length) string is returned.

Note:
You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:
```
c$$ = c$$ + mid$$(a$$,10)
```
Use:
```
c$$ += mid$$(a$$,10)
```

Example:
```
print mid$("Rick Brown", 2, 3)
myContainer$$ = "Rick Brown"
print mid$(myContainer$$, 2, 3)
print mid$("Rick Brown", 6)
```
program output:
```
ick
ick
Brown
```

See Also:
mid$ statement; left$; right$; instr

mid$ and mid$$ **statement**

Syntax:
**mid$** (*stringVar$* , *startPos* , *numChars*) = *replaceString$*
**mid$$** (*container$$* , *startPos* , *numChars*) = ¬
   replaceString$* / *contnr$$

Description:
This statement updates *stringVar$* (which must be a string variable) or *container$$* (a container variable), deleting a subpart from *stringVar$* or *container$$* and replacing it with an equal number of characters from the left side of *replaceString$*. The subpart to be replaced begins at position *startPos* within *stringVar$* or *container$$*. In the following code fragments, containers and strings work the same. The number of characters replaced equals the smallest of these quantities:

- *numChars*
- len(*replaceString$*)
- len(*stringVar$*) – *startPos* + 1

Under the following circumstances, **mid$** does nothing:

- When *stringVar$* or *replaceString$* is empty;
- When *startPos* is less than 1 or greater than len(*stringVar$*) ;
- When *numChars* is less than 1.

Note:
You may not use complex expressions that include containers on the right side of the equal sign.

Example:
```
x$ = "abcdefgh"
y$ = "abcdefgh"
z$ = "abcdefgh"
mid$(x$,2,3) = "1234"
print x$
mid$(y$,2,5) = "1234"
print y$
mid$(z$,7,4) = "1234"
print z$
program output:
a123efgh
a1234egh
abcdef12
```

See Also:
mid$ function; left$; right$; instr

MinWindow **statement**

Syntax:
MinWindow *h , v*

Description:
Sets a limit on how small the user can make the current output window. After **MinWindow** is executed, the user will not be able to drag the window's size narrower than *h* pixels horizontally nor shorter than *v* pixels vertically (these dimensions refer to the window's "content region"; they don't include the "structure region" (frame) of the window). If the current output window's dimension(s) are smaller than *h* and/or *v*, **MinWindow** will not expand it immediately. The window will expand the next time the user drags its grow box.

See Also:
MaxWindow

mki$ function   **function**

Syntax:
*string$* = **mki$**(*intExpr*)

Description:
mki$ ("MaKe Integer string") returns a string which has the same internal bit pattern as *intExpr*; each character in the returned string will represent 8 bits from *intExpr*. The returned string will have a length of 1, 2 or 4 characters, depending on which of defstr byte, defstr word or defstr long is currently in effect. If defstr byte is in effect, you should make sure that *intExpr* is within the range of numbers that can be expressed in a single byte; similarly, if defstr word is in effect, you should make sure that intExpr is within the range of numbers that can be expressed in a "word"-length (2-byte) integer.
mki$** is useful for translating the 4-letter file types, creator codes, resource types, etc. that are frequently used in MacOS Toolbox routines. These codes are typically transmitted in the form of long-integer values; by using the **mki$ function you can translate these long integers into strings for display purposes (be sure to set defstr long before doing this).
If defstr byte is in effect, **mki$** returns the same thing as the chr$ function.

Note:
When defstr long is in effect and 4-character strings and long-integers are being converted, **mki$** is essentially the inverse of the cvi function. Note, however, that the behavior of cvi does not depend on the the current setting of defstr byte/word/long.

See Also:
cvi; defstr byte/word/long; chr$; str$; val

mod operator

Syntax:

remainder = *expr* **mod** *modulus*

Description:

The **mod** operator subtracts from abs(*expr*) the largest multiple of abs(*modulus*) which is less than or equal to abs(*expr*), and returns the result as remainder. If *expr* is negative, then a negative result is returned in remainder.

Note that if *expr* and modulus are both integers, the result of **mod** is just the remainder of the integer division operation *expr* / *modulus*.

See Also:

Appendix D - Numeric Expressions

mouse(_down) function

Syntax:
buttonStatus = **mouse**(_down)

Description:
If your program does not do any kind of event-trapping using the HandleEvents statement, then you can use the **mouse**(_down) function to determine whether the mouse button is currently down. In these circumstances, **mouse**(_down) returns _zTrue is the button is down, or _false otherwise.

The **mouse**(_down) function will not work if your program traps events using HandleEvents. Since most well-designed programs trap events this way, the **mouse**(_down) function is probably of limited usefulness. See the mouse <event>functions to learn how to respond to mouse clicks using event trapping.

Note:
You can also use the Toolbox function fn button to determine whether the mouse is currently down. fn button works regardless of whether your program uses event trapping.

See Also:
mouse <position>; mouse <event>; on mouse

mouse <event> **function**

Syntax:
clickType = **mouse**(0)
locationInfo = **mouse**(*locationType*)

Description:
If you have designated a mouse-event handling routine using the `on mouse` statement, then the **mouse** `<event>` functions return information about a mouse click event. Your mouse-event handling routine should check the **mouse**(0) function, and possibly the **mouse**(locationType) functions, each time your routine is called.

The **mouse** **<event>** functions will not report a mouse click that occurs inside an active control (such as a button or scrollbar), or in an edit field or picture field, or anywhere outside the active window's content region. Such mouse clicks are handled by other routines, such as your dialog-event handling routine (see the [dialog](#) function), or your menu-event handling routine (see the [menu](#) function).

mouse(0) function
The **mouse**(0) function indicates whether a single, double or triple-click occurred. It will usually return one of the following values:

Mouse(0)	Description
_click1nDrag (-1)	single click, and mouse is still down.
_click2nDrag (-2)	double click, and mouse is still down.
_click3nDrag (-3)	triple click, and mouse is still down.

In rare cases, the user may have time to both click the mouse and release it before your program detects the click. This can happen, for example, if your program runs a long time between successive calls to `HandleEvents`. In that case, **mouse**(0) may return one of the following values:

Mouse(0)	Description
_click1 (1)	single click, and mouse is already released.
_click2 (2)	double click, and mouse is already released.
_click3 (3)	triple click, and mouse is already released.

If you just want to detect the click, and you don't care whether the user released the mouse button before your mouse-event handling routine was called, then your routine can just check `abs(`**mouse**(0))`, which will always return 1, 2 or 3.

mouse(locationType) functions
To detect where the mouse pointer was at the instant it was clicked, call the **mouse**(_lastMHorz) and **mouse**(_lastMVert) functions within your mouse-event handling routine. The values returned by **mouse**(_lastMHorz) and **mouse**(_lastMVert) are usually the same as those returned by **mouse**(_horz) and **mouse**(_vert) (see the [mouse <position>](#) functions), but they may be different, especially if the mouse is being moved quickly.

If **mouse**(0) returns a positive value (indicating that the mouse was both clicked and released before your mouse-event handling routine was called), then you may also be interested in the values returned by **mouse**(_releaseHorz) and **mouse**(_releaseVert). These values tell you where the mouse pointer was at the instant the mouse button was released. If **mouse**(0) returned a negative value, then **mouse**(_releaseHorz) and **mouse**(releaseVert) are meaningless.

mouse Window (Appearance Manager)
A new selector helps your program determine where the mouse is located:
`wndNum` = **mouse**(_mouseWindow)
...will return the FutureBasic window reference number of the window over which the mouse is positioned. The window does not need to be active when this is used.

Click Sequencing
FutureBasic reports a mouseclick event as soon as it can after the mouse button has been pressed down. If the user executes a double-click, FutureBasic interprets it first as a single-click event and then (once the second click happens) as a double-click event. Both "events"

will be reported to your mouse-event handling routine. Similarly, if the user executes a triple-click, FutureBasic will first report a single-click event, then a double-click event, and finally a triple-click event.

You should take this into account when writing your mouse-event handling routine. The example program "doubleClick.BAS" handles single-clicks and double-clicks; like most well-designed programs, its interface is designed so that the effects of a single-click are included in the effects of a double-click.

Waiting for the Mouse Up

In most cases, FutureBasic will call your mouse-event handling routine while the mouse button is still being held down. But in some situations, your routine may need to track the mouse's motion until the button is released. You can use the Toolbox function `fn STILLDOWN` to determine when the user releases the mouse button.

See Also:

mouse(_down); mouse <position>; on mouse; HandleEvents

mouse <position>   **function**

Syntax:
horzPosition = **mouse**(_horz)
vertPosition = **mouse**(_vert)

Description:
If your program does not do any kind of event-trapping using the HandleEvents statement, then **mouse**(_horz) and **mouse**(_vert) return the mouse pointer's current horizontal and vertical pixel coordinates, relative to the upper-left corner of the current output window. If your program does use HandleEvents, then the **mouse** <position> functions indicate where the mouse was the last time mouse(0) was accessed (see the mouse <event> functions).

Example:
```
window 1
print "Move the mouse around. Press Cmd-period to quit."
do
 HandleEvents
   dummy = mouse(0) 'To activate the mouse <position> fn's
   xNew = mouse(_horz): yNew = mouse(_vert)
   long if xNew <> xOld or yNew <> yOld
     locate 0,1: cls line
     print xNew, yNew
     xOld = xNew: yOld = yNew
   end if
until _false
```

Note:
Use the mouse <event> functions to determine the mouse's position at the instant it was clicked.

See Also:
mouse(_down); mouse <event>

nand **operator**

Syntax:

result& = exprA { **nand** | ^& } *exprB*

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **nand** operator sets each bit in *result* when the bit in *exprA* is set and the corresponding position in *expB* is cleared. This can be thought of as a `not and` expression. The result is another 32-bit quantity; each bit in the result is determined as follows:

Bit Value in expr	Bit Value in expr	Bit Value in expr
0	0	0
1	0	1
0	1	0
1	1	0

See Also:

and; nor; not; xor; or; Appendix D - Numeric Expressions

NavDialog   **function**

NavDialog Functions

NavDialog(dialogType [+ options], message, typeList | defaultSaveName, callbackFn, userData)

The NavDialog function is similar to the FutureBasic files$ keyword but provides more functionality. Using Navigation Services, the details are hidden in the C runtime, so it is easy to call NavDialog (and several other NavDialogxxxxxx helper functions) as shown in the NavDialog demos.

The parameters

dialogType
_kNavDialogGetFile
_kNavDialogPutFile
_kNavDialogChooseFolder
_kNavDialogChooseFile
_kNavDialogChooseVolume
_kNavDialogChooseObject

options
The programmer may control what the user sees in the dialog and how it is presented. Option constants are combined with the dialogType parameter.

_kNavDialogSheet	requests a sheet dialog attached to a parent window
_kNavDialogMultiple	allows the user to select multiple files
_kNavDialogInvisible	allows the user to see and select invisible files
_kNavDialogSupportPackages	allows the user to see packages
_kNavDialogOpenPackages	allows the user to open packages (like MacOS X application packages)

typeList
Used in _kNavDialogGetFile and _kNavDialogChooseFile dialog types. This parameter allows the programmer to filter by file type. The typeList parameter can be a null string. Same operation as similar parameter in FutureBasic's files$

defaultSaveName
Only appropriate with the _kNavDialogPutFile dialog type. This parameter can be a null string. If supplied, this name is automatically supplied in the "save as.." dialog. The user can obviously replace it with a name of their choosing. Same operation as similar parameter in FutureBasic's files$

callbackFn
The address of a programmer created function. This parameter is always required - see below for more details.

userData
Optional data that can be sent to the callbackFn - The NavDialog_demo uses it to pass a window number but could easily pass a windowRef or other data

Overview of NavDialog Process (order of operation)

(1) Prior to the actual call of NavDialog(), the code establishes:

(a) The dialogType - in other words, getting a file, putting a file, choosing a file
(b) The options, userData, defaultSaveName as described above in parameters
(c) The callbackFn - this is where the code processes the user file/folder selection(s)
(d) If a filterFn is used, NavDialog_SetFilterFn is called with the name of the FN that will filter the files (also see NavDialog_SetFilterFn below)

(2) NavDialog() is called. It interacts with the user and collects the files(s)/folders selected

(3) While (2) is executing the filterFn, if used, is busy deciding which files to show to the user in the dialog

(4) When the user dismisses the dialog (presumably having completed their selections), NavDialog passes control to the callbackFn where the selected files are processed

(5) When the callbackFn ends, control returns to the next sequential instruction after the NavDialog() call. Note: If you specify a sheet window, the NavDialog function returns immediately.

The callback function

The programmer identifies a callback function in their own code. This is done simply by passing the address of the FN in the NavDialog call (i.e. @fn MyGetFileHandler). This callback function (FN) is called by NavDialog to process the results of the dialog interaction with the user. So this is where the program would process the file(s) or folder the user picked in the dialog. Example callback function:

```
local fn MyGetFileHandler( reply as ^NavReplyRecord, userData as pointer )
dim as FSSpec spec
NavDialog_GetItemFSSpec( #reply, 1, _false, @spec )
// do something with file spec
end fn
```

The callback function always takes two parameters: NavReplyRecord and userData.

Ancillary functions to assist with retrieving results

NavDialog_GetItemCount	returns the number of items selected by the user. Useful if the program allows multiple file selection with _kNavDialogMultiple
NavDialog_GetItemFSSpec	returns the FSSpec of a file/folder selected
NavDialog_GetItemFSRef	returns the FSRef of a file/folder selected
NavDialog_CopyItemCFURLRef	returns the CFURLRef of a file/folder
NavDialog_GetSaveFileNameAsPascalString	what it says
NavDialog_CopySaveFileName	returns the saveFileName as a CFStringRef

Optional helper functions

A few helper functions are provided to add features to NavDialog.

NavDialog_AddUTIPascalString	If used, must be called before NavDialog(). UTIs are Uniform Type Identifiers and are the modern filtering method. UTIs are broader and more powerful than OSTypes (i.e. TEXT PICT etc.). An overview can be found in the Apple docs at "Introduction to Uniform Type Identifiers Overview" (/Reference Library/Guides/Carbon/Data Management/Uniform Type Identifiers Overview).
NavDialog_SetFilterFn	If used, must be called before NavDialog() and establishes the name of the filter function (i.e. filterFn). A filter function limits which files the user sees. In the NavDialog_demo the filter function uses UTIs to filter.
NavDialog_UTIConformsTo	May be used in a filter callback. Used to filter by UTIs. The filter checks to see if the UTI of the item passed (first parameter) "conforms" (or is a member of) the UTI group named in the second parameter. If the item is a member, it returns _true (else _false) so it can be included in the list presented to the user.
Other Functions	Many other features/functions are available for use and can be found in Subs Files.incl but are not documented here.

See Also:
Appendix A - File Object Specifiers

See the for statement.

nor   **operator**

Syntax:

result& = *exprA* { **nor** | ^| } *exprB*

Description:

Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **nor** operator sets each bit in *result* when the corresponding bits in both *exprA* and *exprB* are cleared or when the bit in *exprA* is set and the corresponding bit in *exprB* is cleared. This can be thought of as a `not or` expression.The result is another 32-bit quantity; each bit in the result is determined as follows:

Bit Value in expr	Bit Value in expr	Bit Value in expr
0	0	0
1	0	1
0	1	0
1	1	1

See Also:

and; nand; not; xor; or; Appendix D - Numeric Expressions

not   **operator**

Syntax:
value = **not** *expr*

Description:
The **not** operator interprets *expr* as an integer, and returns another integer in whose internal bit pattern all the bits are flipped to their opposite state (i.e., all 1's are changed to 0; and all 0's are changed to 1). Coincidentally, because of the way that integers are stored in FutureBasic, the value returned by **not** *expr* equals: $-(expr + 1)$.

One common use for **not** is to reverse the sense of an expression whose value equals _zTrue (-1) or _false (0). Note that (**not** _zTrue) returns _false, and (**not** _false) returns _zTrue. You must be careful when using **not** with "true" values other than -1. For example:
testValue = 35
if testValue **then beep** 'This produces a beep
if not testValue **then beep** 'But so does this!

This program produces two beeps, because in the second if statement, "**not** testValue" produces the value -36, which is still interpreted as "true" by the if statement.

Another common use for **not** is to help you set or reset individual bits in a bit pattern. For example:
      pattern& = pattern& **and not bit**(7)
This sets bit 7 in pattern& to zero, and leaves all of pattern&'s other bits alone.

See Also:
and; or; xor

oct$ **function**

Syntax:
*octalString$* = **oct$**(*expr*)

Description:
This function is a string of octal (base-8) digits which represent the integer value of *expr*. The returned string will consist of either 3, 6 or 11 characters, depending on which of `defstr byte`, `defstr word` or `defstr long` is currently in effect. Note that if the value of *expr* is too large to fit in the currently selected `defstr` size, the string returned by **oct$** will not represent the true value of *expr*.
In FutureBasic, integers are stored in standard "2's-complement" format, and the values returnet by **oct$** reflect this storage scheme. You need to keep this in mind when interpreting the results of **oct$**, especially when *expr* is a negative number. For example: **oct$**(-3) returns "775" when `defstr byte` is in effect; "777775" when `DESTR word` is in effect; and "77777777775" when `defstr long` is in effect.

Note:
To convert a string of octal digits into an integer, use the following technique:
    `intVar = ` **val&**`("&o" + octalString$)`
`intVar` can be a (signed or unsigned) byte variable, short-integer variable or long-integer variable. See Appendix C - Data Types and Data Representation, to determine the range of values that can be stored in different types of integer variables.

See Also:
hex$; bin$; defstr byte/word/long; val&

offsetof **function**

Syntax:

byteOffset = **offsetof** (*fieldName* **in** { *recordType* | *recVar* })

Description:

Use this function to find where a particular field begins within a record. **offsetof** returns the field's offset as a number of bytes past the beginning of the record.

The *recordType* is the name of a "record" type as defined in a `begin record` statement; *recVar* is a variable declared as a "record" type; *fieldName* is the name of a field within that "record" type.

The value passed as *fieldName* is seen by the compiler as a constant. You do not use type designator suffixes like $,&,#, etc.

See Also:

sizeof; typeof; begin record; Appendix C - Data Types and Data Representation

on dialog   **statement**

Syntax:
on dialog {**fn** userFunction|**gosub**{ lineNumber| "stmtLabel"}}

Description:
This statement designates a particular function or subroutine as a dialog-event handling routine. A dialog-event handling routine is called in response to a number of different kinds of user actions and internal events; see the dialog function for more information.
After a dialog event occurs, FutureBasic does not call your designated routine immediately. Instead, your program continues executing until a HandleEvents statement is reached. At that time, HandleEvents will call your designated routine once for each dialog event that occurred; your designated routine should examine the dialog(0) and dialog(evnt) functions to get information about the event. If you have not designated any dialog-event handling routine, FutureBasic ignores events of this kind.

Note:
If you use the **on dialog fn** userFunction syntax, then userFunction must refer to a function which was defined or prototyped at an earlier location in the source code. Your dialog-event handling function should not take any parameters, nor return a result.

See Also:
dialog function; dialog statement; HandleEvents

on error end   **statement**

Syntax:

on error end Description:

There are two possible outcomes when using this statement and they depend on other factors in your program. If you have not established any other error handling routine, then you may use this routine to turn off all error checking. Errors such as file errors will be ignored. It will be your responsibility to track them manually after each file access statement by checking the function `error`. This concept is demonstrated in the example below.

A second use involves programs where you have set up your own error handling routines. You may toggle between FutureBasic's error handling and your program's built-in error handlers by using **on error end** to turn off FutureBasic's handlers and use the ones in your program. Alternatively, you may use `on error return` to reinstate FutureBasic's handlers.

Example:

```
// Manual, line-by-line error handling
print "This program will produce a file error"
print "that is completely ignored."
on error end
open "I",#1,"this file does not exist"
print
print "The error has occurred and was not flagged."
print "The error number is"; error and &FF
print "In file number"; error >> 8
```

Note:

If you turn off error checking (**on error end**) and you get an error with `x = error`, then your program must clear the error variable with `error = _noErr`

See Also:

on error fn; on error return; error

on error fn / gosub **statement**

Syntax:
on error {**fn** userFunction{(fileID, errorCode)} | **gosub**{lineNumber | "stmtLabel"}}

Description:
This statement designates and enables the routine that FutureBasic will call when certain kinds of errors occur. There may only be one **on error** vector. If you use a second call to **on error fn**, the new routine replaces the old version in subsequent calls. However you can deactivate or reactivate error trapping as often as you need. Using the on error end/on error return statements you can switch between the default behavior and your error handler.

Example:
```
/*
// Standard On Error handler without parameters
local fn myErr
stop "error happened"
end fn
*/

// optional On Error handler accepts two parameters
local fn myErr(id as long, e as long)
stop "id:"+str$(id)+" - e:"+str$(e)
end fn

dim as long test

on error fn myErr

read #2, test
```

Note:
If you use the **on error** fn userFunction syntax, then userFunction must refer to a function which was defined or prototyped at an earlier location in the source code. The error handling function accepts two optional parameters, fileID and errorCode, but doesn't return any result.
errorCode will be set to the error code issued for the failure (i.e. -38 is file not open, -39 end of file error, -43 file not found etc.)

See Also:
on error end; on error return; error function

on error return   **statement**

Syntax:
on error return

Description:
Use this statement to reinstate FutureBasic's standard error checking routines. After this statement is invoked, FutureBasic will display an error dialog and halt the program when a file error is encountered. If your program has established its own `on error fn` vector, it will be ignored. If you wish to return control to your internal routine or turn off FutureBasic's error checking, use `on error end`.

See Also:
on error fn; on error end; error function

on event   **statement**

Syntax:
on event {**fn** `userFunction`|**gosub**{*lineNumber*|*"stmtLabel"*}}

Description:
This statement designates a particular function or subroutine as a system-event handling routine. A system-event handling routine is called in response to any event which the MacOS puts into the event queue designated for your program. This includes various kinds of low-level events such as mouseclicks and keypresses, as well as high-level events such as Apple Events. See the event function for more information.

After a system event occurs, FutureBasic does not call your designated routine immediately. Instead, your program continues executing until a `HandleEvents` statement is reached. At that time, `HandleEvents` will call your designated routine once for each system event that occurred; your designated routine should examine the `event` function to get information about the event.

If there are no events in the system queue when your program executes `HandleEvents`, FutureBasic calls your designated routine once, passing it a "null" event in the `event` record.

Even if you don't designate a system-event handling routine, FutureBasic often uses system events to determine whether other kinds of interesting events have occurred. For example, if the queue contains a system event of type `_mButDwnEvt` (indicating that the user has pressed the mouse button), FutureBasic checks whether the mouse was clicked inside a button, or in the menu bar, or in the "close box" of a window, etc., and may generate an event such as a dialog event or a menu event that your program can detect in other event handling routines.

By designating a system-event handling routine, your program can "intercept" events like `_mButDwnEvt`, before FutureBasic has a chance to interpret them and report them to your other event handling routines. (When a system event occurs, FutureBasic always calls your system-event handling routine first, before any of your other designated event handling routines.) This allows your program to customize the way it responds to system events, in case FutureBasic's "standard" responses don't meet your needs. If you handle an event within your system-event handling routine, you can inhibit FutureBasic from further interpreting the event by setting the `_evtNum` field in the event record to `_nullEvt` before returning from your routine, as illustrated here:

local fn `DoEvent`
 `evtPtr&` = **event**
 select case `evtPtr&.evtNum%`
  `'[handle the event as desired in here]`
 end select
 `'"Hide" the event from further handling by FutureBasic:`
 `evtPtr&.evtNum% = _nullEvt`
end fn

Another good reason to designate a system-event handling routine is so that your program can respond to high-level events such as Apple Events.

Note:
If you use the **on event fn** `userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your system-event handling function should not take any parameters, nor return a result.

See Also:
event

on FinderInfo statement

Syntax:

on FinderInfo {**fn** `userFunction`|**gosub**{ *lineNumber*| *"stmtLabel"*}}

Description:

You establish this vector before entering your event loop. When a file is dropped onto your application's icon or one of your applications files is double-clicked from the Finder, the specified routine is called with information on how to open the file.

The routines are set up to handle up to 1024 files at a time. If this number is insufficient, you will need to change the dim statement in the header file named "Subs Files.Incl". There are three global values maintained for this vector.

`gFBFndrInfoCount`	The number of files pending in the queue.
`gFBInfoSpec(1024) as FSSpec`	An array of file spec records. There is one file spec record for each file that needs to be opened or printed.
`gFBInfoAction%(1024)`	A boolean value that is zero if the file is to be opened and non-zero if it is to be printed.

Example:

This routine establishes a function that is called when a file is dropped onto the compiled version of the application. It also shows how to determine if a file is to be printed or opened.

```
/*
   Build the application,
   then drop a text file on to it
*/
local fn MyOpenFile ( fs as ^FSSpec )
   print "FileName: ";fs.name
end fn
local fn MyFinderInfo
   dim as FSSpec fs
   dim as short @ count, action, j
   dim as OSType @ fType
   count = 0 // set to ask "How many?"
   action = FinderInfo( count, fs, fType ) // FSSpec &
OSType
   long if ( count > 0 ) // at least one file wants in
for j = 1 to count // process them all
  count = -j
  // FSSpec & OSType
  action = FinderInfo( count, fs, fType )
  if ( action == _finderInfoOpen ) and ¬
    ( fType == _"text" ) ¬
    then fn MyOpenFile( fs )
next
fn ClearFinderInfo // in Subs Common.Incl
   end if
end fn
on FinderInfo fn MyFinderInfo
menu 1,0,1, "File"
menu 1,1,1, "Quit/Q"
window 1
do
   HandleEvents
until 0
```

See Also:

FinderInfo; open; Appendix A - File Object Specifiers; Appendix H - Printing; resources

on menu   **statement**

Syntax:
on menu {**fn** `userFunction`|**gosub**{*lineNumber*|*"stmtLabel"*}}

Description:
This statement designates a particular function or subroutine as a menu-event handling routine. A menu-event handling routine is called in response to the user selecting an item from a menu. This includes menu items that your program puts into menus on the menu bar, but it doesn't include items in pop-up menus; see the menu function for more information.
When the user clicks on the menu bar, FutureBasic does not open up the menu immediately. Instead, your program continues executing until a `HANDLEVENTS` statement is reached. If the mouse button is still down at that time, `HandleEvents` then opens the menu, tracks the user's selection, then calls your menu-event handling routine if the user selected a menu item. Your routine should examine the `menu(_menuID)` and `menu(_itemID)` functions to get information about the event.

Note:
If you use the **on menu fn** `userFunction` syntax, then `userFunction` must refer to a function which was defined or prototyped at an earlier location in the source code. Your menu-event handling function should not take any parameters, nor return a result.

See Also:
HandleEvents; menu function

on mouse																		**statement**

Syntax:
on mouse {**fn** userFunction|**gosub**{ lineNumber| "stmtLabel"}}

Description:
This statement designates a particular function or subroutine as a mouse-event handling routine. A mouse-event handling routine is called in response to a mouseclick which occurs inside the content region of the currently active window (but not inside any buttons, scrollbars, edit fields nor picture fields).
After such a mouseclick occurs, FutureBasic does not call your designated routine immediately. Instead, your program continues executing until a HandleEvents statement is reached. At that time, HandleEvents will call your designated routine once for each mouseclick event that occurred; your designated routine should examine the mouse <event> functions to get information about the event.

Note:
If you use the **on mouse fn** userFunction syntax, then userFunction must refer to a function which was defined or prototyped at an earlier location in the source code. Your mouse-event handling function should not take any parameters, nor return a result.
If your program does not use HandleEvents, you can use the mouse(_down), mouse(_lastMVert), mouse(_lastMHorz) and many other position functions (outlined in the FutureBasic Mouse Group of the constants document) to track mouse activity.

See Also:
mouse <event>; mouse(_down); mouse <position>; HandleEvents

on timer statement

Syntax:
on timer(*interval*) **fn** userFunction

Description:
This statement designates a particular function as a timer-event handling routine. A timer-event handling routine is called periodically according to a time interval that you specify.

Setting *interval* to a nonzero value causes timer events to be initiated. If *interval* is positive, it specifies the timer interval in seconds. If *interval* is negative, then abs(*interval*) specifies the interval in ticks (a tick is approximately 1/60 second). Fractional values of *interval*, if positive, are allowed. Setting *interval* to zero does not initiate timer events; in this case, you can use the timer statement to initiate timer events later in your program.

After timer events have been initiated, FutureBasic checks its internal timer whenever a HandleEvents statement is executed. If FutureBasic checks its timer and finds that at least *interval* seconds (or abs(*interval*) ticks) have elapsed since the last time your designated routine was called, it calls your designated routine again.

Timer firings are not queued; they are lost if your application does not handle events for times greater than the *interval*.

Note:
You can use the timer statement to change the timing interval.

See Also:
timer; HandleEvents

open statement

Syntax:

open *"method[fork]",fileID,ref,recLen*

Description:

This statement opens a file so that you can read from it and/or write to it. If you specify a *method* other than `"I"`, the **open** statement also creates the file if it doesn't already exist.

The parameters are interpreted as follows:

method

Specify one of the following letters:

I	Open for input (reading) only. The file must already exist. Other processes may read from the file (but not write to it) while it's open with this method.
O	Open for output (writing) only. If the file already exists, all of its current contents will be destroyed. You have exclusive access to the file (no other process can read from it nor write to it) while it's open with the "O" method. Does *NOT* support the "resource fork" open.
R	Open for "random access." You can either read from or write to the file. The "file mark" (which indicates where the next read or write operation will occur) is placed initially at the beginning of the file. If you write to the file, you only replace those bytes which you're writing; the rest of the file's contents are unaffected. You have exclusive access to the file.
A	Open for "append." This is just like method "R", except the file mark is placed initially at the end of the file. This method is normally used when you want to add data to the end of an existing file.
N	Open for non-exclusive random access. This is just like method "R", except that other processes may read from the file while you have it open. Your process is the only one allowed to write to the file.

fork

Specify one of the following letters:

D	Open the "data fork." This is the default value.
R	Open the "resource fork." You should only specify this value if you're doing something like copying an entire resource fork from one file to another, or examining the internal structure of the resource fork. You should not use the Open statement when you want to read or write individual resources; in that case, use a Toolbox function such as HOpenResFile instead. Do not use the Open statement to create a new resource fork; use a Toolbox function such as HCreateResFile instead.

fileID

Specify a number in the range 1 through 255 which is not being used by any other currently open file. You can use this number to identify the open file in statements and functions such as `read#`, `write`, `eof`, `lof`, etc. The *fileID* number is associated with the file until you close the file.

ref

The 'ref' can be either an FSSpec, FSRef or CFURLRef. The FSRef and CFURLRef are opaque references to the file name and location to be opened. The FSSpec contains the filename and location. Appendix A - File Object Specifiers, describes the composition of each 'ref' type. Open syntax uses any of FSSpec, FSRef or CFURLRef and is identical except when using the "O" (output) option with an FSRef. Examples are:

open "I", 1, @spec	An FSSpec record is used
open "I", 1, @ref	An FSRef object is used
open "O", 3, fileName$, @parentFolderRef	An FSRef cannot be created for a non-existent file, so a filename and the FSRef of the parent directory must be supplied
open "I", 2, @url	A CFURL object is used.
open "A", 5, @ref	An FSRef object is used.
open "R", 2, @ref	An FSRef object is used.

open "I", 2, @url	A CFURL object is used.

 recLen

This value indicates the length of the records in the file; naturally, it's most useful when the file consists of fixed-length records. The value you specify is used when you execute statement and functions such as `record`, `rec`, `loc` and `lof`. If you omit this parameter, a default value of 256 is used. If the file doesn't consist of fixed-length records, it's often most convenient to set *recLen* to 1.

See Also:

close; inkey\$; input#; print#; read#; read file; read field; write; write file; write field; record; rec; loc; lof; eof; files\$

open "C" **statement**

Syntax:

open `"C"`, *portID,baud* ¬

   `[,[` *parity* `][,[` *stopbit* `][,[` *wordLength* `][,` *buffer* `]]]]`

This statement opens a serial communications port (the modem port or the printer port) so that your program can write to or read from a serial device. The optimal values for the various parameters depend on the device and the desired communications protocol; see the device's manual for more information. The parameters are interpreted as follows:

nbsp;*portID*

Set this either to `_modemPort` or to `_printerPort` or to any port sepcified to a maximum -8. (Ports are numbered from -1 for the printer port to -8.) The `_modemPort` value also usually works to communicate with a built-in modem. Some Macintosh computers provide different values. A Powerbook generally uses `_modemPort` as the infra red port. and `_printerPort` as the internal modem. USB adapters such as the Keyspan adapter will provide different values if the device is connected before booting as opposed to plugging it in after the computer is running.

nbsp;*baud*

Set this to one of the following values: 110; 300; 1200; 1800; 2400; 3600; 4800; 7200; 9600; 19200; 38400; 57600, 115200, 230400.

nbsp;*parity*

Set this to one of the following values: `_noParity`; `_oddParity`; `_evenParity`. The default value is `_noParity`.

nbsp;*stopbit*

Set this to one of the following values: `_oneStopBit`; `_twoStopBits`; `_halfStopBit` (1.5 stop bits). The default value is `_oneStopBit`.

nbsp;*wordLength*

Set this to one of the following values: `_fiveBits`; `_sixBits`; `_sevenBits`; `_eightBits`. The default value is `_sevenBits`. Note: do not set this parameter to the values 5, 6, 7 or 8: these are different from the values of the symbolic constants.

nbsp;*buffer*

Set this to a number in the range 1 through 32,768. This parameter indicates how many bytes to allocate for an input buffer. The input buffer stores data that is being received, even when the program is not reading it, allowing the program to process data while data is being received in the background. The default value for *buffer* is 4096 bytes. To determine the number of unread characters currently in the buffer, use `lof(portID,1)`.

Reading Data

To read incoming data from an open serial port, use the same commands that you would use to read data from a file; e.g., `input#`, `read#`, etc. Since it's difficult to predict when (if ever) the data will come in, it's best to design your program so that it won't get "stuck" on a single statement waiting for incoming data. Instead, you should execute a loop that periodically checks whether there is any data to read. This will allow your program to proceed with other activities while it's waiting; or to quit waiting if too much time has elapsed.

There are basically three ways to check whether there is any data available in the buffer:

- You can check the value of `lof(portID,1)`. This will return zero if no data is available to read; otherwise, it returns the number of bytes waiting to be read.
- You can use the `read#` *portID, stringVar$;*`0` statement. By specifying "`;0`", you instruct the `read#` statement to return immediately if there is no data available; if there is data, the statement reads all the characters currently in the input buffer (up to the maximum allowable length of *stringVar$*), and puts them into *stringVar$*. You can use `len(`*stringVar$*`)` after the `read#` statement to determine how many (if any) characters were read.
- You can use the `inkey$(`*portID*`)` function. It will either return one character from the buffer, or a null string if the buffer was empty.

Writing Data

To write data out to an open serial port, use the same commands that you would use to write data to a file; e.g., `print#`, `write`, etc.

FutureBasic Runtime Globals

FutureBasic has several reserved global variables. (See Subs Files.Incl in Header folder)

```
gFBHasComTB% //true if comm toolbox is used...
gFBSerialPortCount% //number of com port
gFBSerialName$(n) //serial port name
gFBSerialInName$(n) //input buffer name
gFBSerialOutName$(n) //output buffe name
gOSXSerialInited //[!]0 if serial inited under MacOS X
```

After any communications port has been opened or after you make your own call to the runtime `fn FBInitSerialPorts`, you may refer to `gFBSerialPortCount%` for the total number of devices (maximum 8). `gFBSerialOutName$(n)` contains the name of the device. With this in mind, the serial ports can best be referred to by name rather than number when multiple ports are present or when USB devices are in use for the purpose of emulating serial ports.

To search all available communication ports use the folling lines. This is especially important if a USB/serial port adapter is inserted after the program has started.

```
gFBSerialportCount% = 0
// This is for MacOS X
long if system(_sysVers) => 1000
  gOSXSerialInited = _false
end if
```

See Also:

close; HandShake; loc; lof; input#; read#; read file; read field; inkey$; print#; write; write file; write field

open "UNIX"   **statement**

Syntax:
open "UNIX" *, fileID,UNIXcommand$*
Revised:
August, 2002 (Release 7)

Description:
Mac MacOS X brings a wealth of commands with its UNIX foundation. These commands are easy to access from FutureBasic. You may establish one or several simultaneous channels by opening them as files. Each channel should have a unique file number. There is a theoretical maximum of 255 files or channels that may be opened at once.

Once the channel has been opened, you read from the file to accept the resulting list from UNIX. (Exception: some UNIX commands perform a task without responding to the user, so no file reading is required.) When the operation is complete, close the channel as you would any file; with the close statement.

There are literally hundreds of books written about UNIX commands. Suffice it to say that if the Mac supports a specific UNIX command, it may be accessed through **open "UNIX"**

Example:
```
dim as Str255 a
window 1,, (0,0)-(600,550)
text _monaco, 10
// print a title-string then list a directory
open "UNIX", 2, "echo ""Root Directory""; ls -l"
do
    line input #2, A
    print a
until eof(2)
close 2
do
    HandleEvents
until 0
```

Note:
open "UNIX" works only in MacOS X.

See Also:
line input; close

or   **operator**

Syntax:
result& = *exprA* { **or** | ||} *exprB*

Description:
Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **or** operator performs a "bitwise comparison" of each bit in *exprA* with the bit in the corresponding position in *exprB*. The result is another 32-bit quantity; each bit in the result is determined as follows:

Bit Value in expr	Bit Value in expr	Bit Value in expr
0	0	0
1	0	1
0	1	1
1	1	1

The **or** operator can also be used to join two "condition clauses" for use in statements like `if`, `while` and `until`. For example:
  if `n > 17` **or** `myName$ = "Smith"` **then beep**
This statement produces a beep if either `n > 17` is true, or `myName$="Smith"` is true, or both.
Even when it's used to join condition clauses, the **or** operator still does a "bitwise comparison." This happens because FutureBasic actually assigns a numeric value to every condition clause, depending on whether the clause is true or false. for example, the clause `n>17` is evaluated as -1 if it's true, or as 0 if it's false. Conversely, any numeric expression is judged as "true" if it's non-zero, or as "false" if it's zero.

Example:
In the following example, expressions are evaluated as true or false before a decision is made for branching. The logical expression `state$="IL"` is true, and therefore evaluated as -1. The expression `state$="CA"` is false, and is therefore evaluated as 0. Then the bitwise comparison `(-1)or(0)` is performed, resulting in -1. Finally, the `long if` statement interprets this -1 result as meaning "true," and therefore executes the first `print` statement.
```
state$ = "IL"
long if state$ = "IL" or state$ = "CA"
  print "Okay"
xelse
  print "Invalid state"
end if
```
The example below shows how bits are manipulated with **or**:
```
defstr long
print bin$(923)
print bin$(123)
print "--------------------------------"
print bin$(923 or 123)
```
program output:
```
00000000000000000000001110011011
00000000000000000000000001111011
--------------------------------
00000000000000000000001111111011
```

See Also:
not; and; xor; Appendix D - Numeric Expressions

output   **statement**

Syntax:
output [**file**] *"filePath"*

Description:
Use this statement to specify a name for the application file that's created when you "Build" a project. If your program doesn't contain an **output** statement, then FutureBasic will use a default name (as set in your preferences) when you select "Build" (B from the "Command" Menu).
The *filePath* can be:

- A simple file name. The application file is saved in your project folder.
- A relative path name (starting with a colon). The path is relative to your project folder.
- A full path name. The application file is saved in the specified folder.

The **output** statement can appear anywhere in your program, but most commonly it appears somewhere near the beginning.

Note:
output is a non-executable statement, so you can't affect its operation by putting it inside conditional execution structures like `long if...end if`. You can, however, conditionally include or exclude it using `compile long if`.

See Also:
compile; resources

override   **statement**

Syntax:
override _constantFoo = newValue
override _constantBar$ = "new value"

Description:
The override statement may be used to change the value of a constant or string constant.

Note:
When you override a constant, any code compiled after the override is affected. Constants are not variables. They are only examined at compile time. It is therefore not possible to override a constant that is used in the runtime since FutureBasic has already compiled the entire runtime before your override is ever encountered.

page   **function**

Syntax:
lineCount = **page**

Description:
This function returns a count of the text lines which have been sent to the printer on the current page. Its value is incremented each time a carriage-return character is sent to the printer, and is reset to zero each time a printed page is ejected.

See Also:
csrlin; close lprint; clear lprint; route

Syntax:
page

Description:
This statement is identical to the `clear lprint` statement.

See Also:
clear lprint

page lprint   **statement**

Syntax:
page lprint

Description:
This statement prints the content area of the current output window to the selected printer. The picture is a bit mapped copy of screen pixels. Before the printing starts, the user is offered a page setup dialog.

See Also:
def lprint; route

peek   **function**

Syntax:
byteValue` = **peek** [**Byte**](address&)
shortIntValue% = **peek Word**(address&)
longValue& = **peek Long**(address&)

Shorthand syntax:
byteValue` = |address&|
shortIntValue% = {address&}
longValue& = [address&]

Description:
The **peek** functions look at the 1, 2 or 4 bytes of data which begin at *address&*, and return them as a byte integer, short integer or long integer value, respectively. The *address&* should be a long integer expression, or a Pointer or Handle variable. The value returned by a **peek** function will be interpreted either as a signed or unsigned value, depending on what type of variable it's assigned to. If the value is not assigned to any variable, it's usually interpreted as a signed value.

See Also:
Poke; varptr

pen   **statement**

Syntax:
pen [*penWidth*][, [*penHeight*][, [*visible*][, [*mode*][, *pattern*]]]]

Description:
This statement alters the characteristics of the drawing "pen" in the current output window. The pen characteristics affect the appearance of QuickDraw shapes (lines, ovals, rectangles, etc.) that are subsequently drawn in the window. If you omit any parameter, the corresponding characteristic is not altered. The parameters are interpreted as follows:
nbsp;*penHeight* and *penWidth*
These specify the height and width of the pen in pixels. They must be positive integers. Taller, wider pen sizes produce thicker lines and borders.
nbsp;*visible*
If you set this to _false, subsequent drawing won't be visible on the screen (but it will still be "recorded," if you have turned on picture recording (see the picture on statement)). If you set *visible* to _true, subsequent drawing will be visible.
nbsp;*mode*
This determines how the pen behaves when you draw over existing images in the window. Usually you will use one of the following constants:

_patCopy	_transparent
_patOr	_addOver
_patXor	_addPin
_patBic	_subPin
_notPatCopy	_adMax
_notPatOr	_subOver
_notPatXor	_adMin
_notPatBic	_blend

nbsp;*pattern*
This determines the pattern that will be used to draw lines, and to frame or fill shapes. Specify a number in the range 0 through 37 to get one of the following system patterns:

Note:
To change the pen's color, use the color or long color statement. To change the appearance of text, use the text statement.

See Also:
plot; box; circle; fill; color; long color; text

picture   **function**

Syntax:
pictureHandle& `=` **picture**

Description:
This function returns a handle to the picture recorded with the most recent pair of `picture on/picture off` statements. This handle is the same handle returned by:

    picture off `, pictureHandle&`

You can specify this handle in the `picture` statement when you want to draw the picture. You can also pass the picture handle to any of a number of Toolbox routines which require a picture handle as a parameter.

Note:
Your program is responsible for releasing the memory occupied by pictures created with the `picture on/picture off` statements. You should normally use the `kill picture` statement to do this, once you're finished using the picture handle. However, if you turn the picture into a resource (using the Toolbox routine `AddResource` or PG's `FNpGreplaceRes`) then you should not dispose of the picture.

See Also:
picture on/off; picture statement

picture   **statement**

Syntax:
picture `[(`*h1*`,`*v1*`)][-(`*h2*`,`*v2*`)]` `[,`*pictureHandle&*`]`

Description:
This function draws a picture in the current output window, or to the printer if output is currently routed to the printer. If you specify *pictureHandle&*, the picture referenced in that handle is drawn; this can be a handle returned by the `picture` function, or a `PICT` resource handle, or any other valid picture handle. If you don't specify *pictureHandle&*, the picture which was recorded by the most recent pair of `picture on/picture off` statements is drawn.
The `(`*h1,v1*`)` and `(`*h2,v2*`)` parameters specify the upper-left and lower-right corners of a rectangle. If you specify both `(`*h1,v1*`)` and `(`*h2,v2*`)`, the picture is scaled to fit the indicated rectangle. If you specify only `(`*h2,v2*`)`, the picture is scaled to fit the rectangle `(0,0)-`
`(`*h2,v2*`)`. If you specify only `(`*h1,v1*`)`, the picture is drawn unscaled, with its upper-left corner at `(`*h1,v1*`)`. If you specify neither corner, the picture is drawn unscaled, using the same frame rectangle that was used to record the picture.

See Also:
picture on/off; picture function

picture on/off   **statement**

Syntax:
picture on `[(`*h1*`,`*v1*`)][-(`*h2*`,`*v2*`)]`
picture off `[,`*pictHandleVar*`&]`

Description:
The **picture on** statement initiates "picture recording" for the current output window. The **picture off** statement turns off picture recording. while picture recording is on, all drawing commands and text-display commands which are sent to the window are "recorded" in a special data structure called a picture. The internal format of a picture is identical to that of a resource of type "`PICT`". After picture recording is turned off, you can display or print the picture, or attach it to a picture field, or save it to disk (as a "`PICT`" resource or a "`PICT`" file).
Every picture has a frame, an imaginary rectangle which is stored as part of the picture's data structure. Usually, but not always, the picture itself is contained within the frame. The frame is used as a reference to determine how the picture should be scaled and positioned when the picture is later displayed or printed. The picture recording is cropped to the recording window's clip region, which may or may not be the same as the frame rectangle.
In the **picture on** statement, the *(h1,v1)* and *(h2,v2)* parameters specify the upper-left and lower-right corners of a rectangle. If you specify both *(h1,v1)* and *(h2,v2)*, they define the picture's frame. Otherwise, the frame is determined as follows:
If you specify only `(`*h1,v1*`)`, it becomes the frame's upper-left corner; the frame's lower-right corner is set to the current lower-right corner of the window. If you specify only `(`*h2,v2*`)`, the picture's frame is set to the rectangle `(0,0)-(`*h2,v2*`)`. If you specify neither `(`*h1,v1*`)` nor `(`*h2,v2*`)`, the picture's frame is set to the window's current rectangle.
In the **picture off** statement, a handle to the recorded picture is returned in *pictHandleVar&*, if it's specified. *pictHandleVar&* must be a long-integer variable, or a `Handle` variable. The value returned in *pictHandleVar&* is the same as the value returned by the `picture` function.
By default, drawing commands are not visible on the screen while they're being recorded. If you want the picture to be visible while you're recording it, you must call the Toolbox procedure `SHOWPEN` after you start recording. If you do this, `SHOWPEN` must be "balanced" by a call to the `HIDEPEN` procedure, before you turn off picture recording. For example:
picture on
call SHOWPEN `'show while recording`
`'[execute drawing commands here]`
call HIDEPEN `'balance the call to SHOWPEN`
picture off
Only one window can have picture-recording enabled at any given time; you cannot "nest" **picture on/picture off** pairs. If you switch output windows while picture recording is on, any drawing commands sent to the new output window will not be recorded.
You cannot "temporarily" turn off the recording of a picture. Each call to **picture off** completes a picture, and each call to **picture on** starts a brand new picture. However, it is possible to effectively "append" one picture to another, by "inserting" an old picture inside a new one. For example:
picture on
circle `50,50,45`
picture off`, circlePict&`
`:`
`'"Append" more drawing commands:`
picture on `'start a new picture`
picture `,circlePict& 'This gets recorded in new pict box 20,20 to 50,50`
`:`
picture off`, pictHandle& 'contains circle & box`

Note:
Your program is responsible for releasing the memory occupied by pictures created with the **picture on/picture off** statements. Use the `kill picture` statement to do this, once you're finished using the picture handle.
Drawing commands which plot icons are not recorded in pictures.

See Also:
picture function; picture statement; kill picture

FutureBasic 5

plot   **statement**

Syntax:
plot *h* , *v*
plot to *h* , *v*
plot *h1* , *v1* **to** *h2* , *v2* [**to** *h3* , *v3* ...]

Description:
This function draws a "point," a line, or a series of connected lines, in the current output window, using the current pen size, pen mode, pen pattern and foreground color.
If you use the first syntax, a single "point" is drawn. This will actually be a little rectangular block whose dimensions are the same as the pen's width and height, with its upper-left corner at point (*h,v*) .
If you use the second syntax, a line is drawn, having one endpoint at (*h,v*) . The line's other endpoint will be one of the following:

- The last point specified in the most recent **plot** statement (in any window);
- The (*h,v*) coordinates of the most recent box statement (in any window) that actually specified the h and *v* parameters;
- (0 , 0) , if no **plot** statement has yet been executed, and no prior box statement that specified h and *v* has yet been executed.

If you use the third syntax, a line or a series of connected lines is drawn, with endpoints at the specified points.

See Also:
pen; box; color; long color

poke **statement**

___

Syntax:
poke [**byte**] *address&, byteExpr*
poke word *address&, shortIntExpr*
poke long *address&, longIntExpr*

Shorthand syntax:
```
| address&, byteExpr
% address&, shortIntExpr
& address&, longIntExpr
```

Description:
These statements copy the value in *byteExpr*, *shortIntExpr* or *longIntExpr* into the 1, 2 or 4 memory bytes (respectively) which start at location *address&*. The *address&* should be a long integer expression, or a `pointer` variable.

See Also:
peek; varptr; BlockMove; let

pos function

Syntax:

numCharacters = **pos**(*deviceType*)

Description:

This function returns a number which means different things depending on the value of *deviceType*. Use one of the following values for *deviceType*.

nbsp;_anyDev

pos(*_anyDev*) returns information about the number of characters sent by the print statement to screen windows or to the device specified by the most recent route statement. The value of **pos**(*_anyDev*) is incremented whenever you print a character (other than a carriage-return) to any open window or to the route'd device. The value of **pos**(*_anyDev*) is reset to zero whenever any of the following happens:

- You send a carriage-return character to any window or route'd device (this is usually the final character sent by a print statement); or:
- The text in any window or route'd device reaches the right margin and wraps around to the next line; or:
- You open a new window with the window statement; or:
- You start a new print job with the route _toPrinter statement.

Note that **pos**(*_anyDev*) often, but not always, represents the number of characters on the current line of text. However, because FutureBasic does not maintain separate **pos** values for separate windows, the value returned by **pos**(*_anyDev*) may represent the characters on a line in the current window, or on a line in a different window, or even the sum of the characters on lines in multiple windows.

nbsp;_printerDev

pos(*_printerDev*) returns the number of characters printed so far on the current line of text sent to the printer. The value of **pos**(*_printerDev*) is incremented whenever you send a chacter (other than carriage-return) to the printer; the value is reset to zero whenever you send a carriage-return character to the printer, or the text reaches the right margin and wraps around to the next line.

nbsp;_diskDev

pos(*_diskDev*) returns information about characters sent to open files. The value of **pos**(*_diskDev*) is incremented whenever you send a character (other than carriage- return) to any open file; the value is reset to zero whenever you send a carriage-return character to any open file. Note that if you have more than one file open, the value returned by **pos**(*_diskDev*) reflects the sum of the characters sent to all the files you're writing to. If you write a total of more than 32767 characters (to all open files) without writing a carriage-return character, the number returned by **pos**(*_diskDev*) is invalid.

Note:

To determine the current horizontal pen position (in pixels), use the window(_penH) function.

See Also:

csrlin; width; page function; window function

prCancel **function**

Syntax:
userCancelled = **prCancel**

Description:
You should examine the value of **prCancel** after executing the `def lprint` statement. **prCancel** returns `_zTrue` if the user pressed the "Cancel" button in the Print Job dialog; or `_false` if the user pressed the "OK" button. If **prCancel** returns `_zTrue`, your program should not continue with the print operation.

See Also:
def lprint; prHandle; route

prHandle   **function**

Syntax:

pRecH& = **prHandle**

Description:

This function returns a handle to the print record. The print record (also called a "TPrint record") contains useful information about the printer and about the current print job (if any). For a complete description of the contents of the print record, see the "Printing Manager" chapter of Inside Macintosh: Imaging with QuickDraw. A few of the more useful fields from this record are listed here:

 pRecH&..prInfo.rPage

This gives the print page rectangle. You can get the width and height of the page as follows:

```
pageWidth = pRecH&..prInfo.rPage.right% - ¬
pRecH&..prInfo.rPage.left%
pageHeight = pRecH&..prInfo.rPage.bottom% - ¬
pRecH&..prInfo.rPage.top%
```

Use these numbers to determine how much room is available for text and graphics. The page rectangle is affected by the user's selections in the "Page Setup" dialog: for example, if the user selects "landscape" mode, the page width will be greater than the page height.

 pRecH&..prJob.iFstPage%; pRecH&..prJob.iLstPage%

These numbers indicate the first and last pages which the user wants to print. Your program should not print any pages outside of this range.

 pRecH&..prJob.iCopies%

The number of copies to print. This is the number of times your program should send the selected pages to the printer. Note that many newer printer drivers will always set this number to 1; such printer drivers will handle multiple copies internally.

Note:

prHandle is partially emulated for Carbon. FutureBasic creates a 120 byte handle and fills in a few standard rectangles and other values so that programs won't be so likely to break during the transition period to MacOS X. But these emulations are not something that you should depend on in all future versions of the IDE.

See Also:

def lprint

print   **statement**

Syntax:
print `[@(`*col*`,`*row*`)|%(`*h*`,`*v*`)]` [*printItem* `[{,|;}[`*printItem*`]...]]`
print `[@(`*col*`,`*row*`)|%(`*h*`,`*v*`)]` `[Point [{,|;}[Point]...]]`

Description:
Use this statement to put text out to the current window or to the currently `route`'d device. The text is printed using the window's or printer's current font ID, font size, text style, text mode and foreground color (see the `text`, `color` and `long color` statements). The parameters are interpreted as follows:
When the item to be printed is a point, FutureBasic makes special provisions during the print process.
dim `mousePos` **as Point**
call GETMOUSE`(mousePos)`
print `mousePos`
The output looks like this `(194x,167y)`
 `@(`*col,row*`)|%(`*h,v*`)`
This specifies where the first printed character should appear within the window or the printed page. If you use the `@(`*col,row*`)` variant, then *col* and *row* represent the text column and row where the first character should appear; the exact pixel location depends on the current font ID and font size. If you use the `%(`*h,v*`)` variant, then *h* and *v* represent horizontal and vertical pixel positions; the first printed character is positioned with its lower-left corner at point `(`*h*`,`*v*`)`. If you don't specify either variant, printing begins at the window's or printer's current pen position.
 printItem
This can be any of the following:

printItem	Description
a string expression	The string is printed. If the string includes a carriage-return character (ASCII character 13), the character causes the pen to move down to the beginning of the next line.
a numeric expression	The decimal value of the number is printed. A space character is always printed after the number; if the number is non-negative, a space character is also printed before the number. FutureBasic formats the number in a reasonable way; if you need the number to appear in a special format, use string functions such as Using, Hex$, Str$, etc.
a Pointer or Handle variable	The variable's value is interpreted as an address, which is then printed as a numeric expression (see above).
Tab(position)	Sufficient space characters are printed until the current line contains position -1 characters. (If there are already more than position -1 characters on the line, Tab does nothing.) This is usually done to help line up several lines of text into neat columns. Note that this effect looks best if you're using a monospaced font.
Spc(numSpaces)	numSpaces space characters are printed. This has the same effect as printing the string expression Space$(numSpaces).

 `{,|;}`
You must use a comma or a semicolon to separate each pair of *printItem*'s; you can also optionally put a comma or semicolon at the end of the **print** statement (following the last *printItem*).
A comma causes space characters to be printed until the total number of characters on the line is a multiple of the current "tab field width." Commas are usually used to help line up several lines of text into neat columns, or just to put some space between consecutive *printItem*'s.
A semicolon does not cause any spaces to be inserted between consecutive *printItem*'s. Use this when you want *printItem*'s to be printed as close together as possible.
Normally, **print** moves the pen down to the beginning of the next line after the last *printItem* is printed. However, if you put a comma or a semicolon at the end of the **print** statement, the pen remains to the right of the last *printItem*, and is not moved down to the next line. This allows you to continue printing on the same line using a subsequent **print** statement.
Note that you can use **print** without any parameters, like this:
print
This simply causes the pen to move down to the beginning of the next line; it effectively "prints" a carriage-return character. This is useful for putting blank line(s) between other lines of text.

Line wrap, scrolling and page-eject

By default, if the *printItem*'s reach the right edge of the window or the printer page, the **print** statement automatically "wraps the line"; that is, it moves the pen down to the beginning of the next line to continue printing the remaining *printItem*'s. However, this behavior can be altered using the `width` statement; see the width statement for more information.

If you're printing to a window, and the **print** statement causes the pen to move below the bottom of the window, the window's contents are scrolled up so that the newly printed text will be visible.

If you're printing to the printer, and the **print** statement causes the pen to move below the bottom of the printer page, the page is ejected and printing continues at the top of the next page.

Note:

Text which is displayed using the **print** statement is not automatically refreshed, unless you're using the "FB Lite" runtime. To display text which is automatically refreshed, consider using Edit Fields (see the edit field statement).

See Also:

text; color; long color; width; using; space$; edit field

print using   **statement**

See the print statement and the using function.

print#   **statement**

Syntax:

print# *deviceID*, [*printItem* [{ , | ; } [*printItem*] ...]]

Description:

This statement writes information formatted as text to the open file or serial port specified by *deviceID*. The list of *printItem*'s is interpreted and formatted the same way as in the `print` statement. **print#** normally writes a carriage-return character (ASCII character 13) after writing the final *printItem*; to inhibit this behavior, put a comma or a semicolon at the end of the **print#** statement.

print# is typically used to write data which is to be viewed later in a text editor or word processing program; or to write data which is to be read later by the `input#` statement. It generally formats its output differently than the `write` and `write file` statements, which are better suited for transferring the contents of memory directly to the device. For example, consider this sample program fragment:

```
a% = -1623
print #1, a%
write #1, a%
```

In this example, the `print#` statement formats the number as text, and puts out 7 bytes, as follows:

00101101 (ASCII code for "-")

00110001 (ASCII code for "1")

00110110 (ASCII code for "6")

00110010 (ASCII code for "2")

00110011 (ASCII code for "3")

00100000 (ASCII code for a space character)

00001101 (ASCII code for a carriage-return character)

On the other hand, the `write` statement puts out the binary contents of `a%`. In memory, the short integer `-1623` is stored as 1111100110101001; therefore, the `write` statement puts out these two bytes:

11111001 10101001

See Also:

print; input#; write; write file; open; route

pstr$ **function**

Syntax:
*string$* = **pstr$**(*address&*)

Description:
This function returns the string which is located at the indicated *address&* in memory; *address&* must be a long-integer expression or a pointer variable.

The data at *address&* should be a string in "Pascal format," which is the string format used by FutureBasic string variables and by MacOS Toolbox string parameters. In Pascal format, the first byte is interpreted as a number in the range 0 through 255 which indicates the length of the string's text; this length byte is immediately followed by the text of the string.

See Also:
pstr$ statement; str#

pstr$ **statement**

Syntax:
**pstr$**(*addressVar&*) = *string$*

Description:
This statement changes the value of *addressVar&*, setting it to the address of *string$*. *addressVar&* must be a long-integer variable or a pointer variable; *string$* must be a string variable or a literal string in quotes.

If *string$* is a string variable, the **pstr$** statement is equivalent to this:
    addressVar& = @string$
If *string$* is a literal string in quotes, then *addressVar&* is set to an address in FutureBasic's heap where the literal string is stored. The **pstr$** statement is the only way to obtain the address of a literal quoted string.

Note:
The **pstr$** keyword can also be used with the read statement, to obtain the address of a string specified in a data statement.

See Also:
pstr$ function; read

put preferences   **statement**

Syntax:
put preferences *prefFileName$,   prefRecord*

Description:
This statement writes the contents of the preferences record *prefRecord* to the file name in *prefFileName$.* The file is created in the preferences folder at: ~/Library/Preferences. If the file does not exist, it is created.

Example:

begin record prefsRecord
dim as   Str31   name
dim as   SInt32   aNumber
end record

dim as prefsRecord gMyPref

gMyPref.name = "my test name"
gMyPref.aNumber = 123456

put preferences "MyPrefs", gMyPref

do
HandleEvents
until (gFBquit)

See Also:
get preferences; kill preferences; menu preferences

random statement

Syntax:
random [**IZE**] [*expr*]

Description:

This statement "seeds" the random number generator: this affects the sequence of values which are subsequently returned by the rnd function and the maybe function.

The numbers returned by rnd and maybe are not truly random, but follow a "pseudo-random" sequence which is uniquely determined by the seed number. If you use the same seed number on two different occasions, you'll get the same sequence of "random" numbers both times. For example:

cls
for i = 1 to 2
 randomize 325
 for j = 1 **to** 10
  print rnd(50),
 next
 print
next

The program above seeds the random number generator twice with the same number (325). If you run this program, you'll find that it produces the same sequence of 10 random numbers after each seeding.

Seeding the random number generator with a pre-specified number can be useful in cases where you specifically want to produce a repeatable sequence of random numbers. In most cases, however, you will probably prefer a sequence that is unpredictable. In that case, you should omit the *expr* parameter. When you execute **random** without any *expr* parameter, the system's current time is used to seed the random number generator. Since the system clock changes very quickly, it's essentially impossible to predict what value will be used as the seed-this is the best way to get the "most random" random numbers.

Normally, you will execute **random** only once in your program, unless you wish to repeat a specific sequence of random numbers. If you don't execute **random** at all, the random number generator is seeded with the current tick count. Remember that re-seeding with the same number will cause your program to generate the same sequence of random numbers each time it's run.

See Also:
rnd; maybe

randomize   **statement**

This is a synonym for the random statement.

ratio   **statement**

Syntax:
ratio *h* , *v*

Description:
This statement affects the width and height of shapes subsequently drawn with the `circle` statement. By suitably setting the *h* and *v*
parameters, you can draw ellipses with any aspect ratio.
Each of *h* and *v* can range in value from -128 to +127. The *h* parameter affects the width of the ellipse, and the *v* parameter affects its
height. When you subsequently execute a `circle` statement with a radius parameter of `radius`, a circle or ellipse is drawn which has
these dimensions:
Shape's width = `2 * radius * (1 + h /128)`
Shape's height = `2 * radius * (1 + v /128)`
We can notice a few consequences of these formulas:

- If *h* and *v* are equal, a circle is drawn.
- If *h* and *v* are different, an ellipse is drawn.
- If *h* > *v*, the ellipse is wider than it is tall.
- If *v* > *h*, the ellipse is taller than it is wide.
- Negative values of *h* or *v* cause the corresponding dimension to shrink. Positive values cause the corresponding dimension to grow
  (up to about twice its original size).
- Specifying **ratio** `0,0` "resets" the ratios: subsequent `circle` statements will draw a circle with the indicated radius.

After you execute a **ratio** statement, the indicated *h* and *v* values remain in effect for all subsequent `circle` statements (in all windows),
until another **ratio** statement is executed.

See Also:
circle

read **statement**

Syntax:
read { *var* | **pstr$** (*addressVar&*) } [, { *var* | **pstr$** (*addressVar&*) } ...]

Description:
This statement reads one or more of the items listed in one or more `data` statements. If you specify *var* (which must be either a numeric variable or a string variable), the data item's value is stored into *var*. If you specify **pstr$** (*addressVar&*) , the item is interpreted as a string and its address is stored into *addressVar&* (which must be a long-integer variable or a `pointer` variable).
Each *var* or **pstr$** that you specify causes one data item to be read. The first time your program executes a **read** statement, the first item in your program's first `data` statement is read. Every time a *var* or **pstr$** is encountered in any **read** statement, the next data item is read, until all items in all your program's `data` statements have been exhausted. The number of *var* or **pstr$** specifications in a **read** statement does not need to match the number of items in a `data` statement; however, the total number of read requests should not exceed the total number of items in all `data` statements (unless you use the `restore` statement, which allows you to re-use data from previous `data` statements).

Example:
data 1,2
data 3,4
data 5,6
for i = 1 **to** 2
  read a, b, c
  print a, b, c
next
program output:
1 2 3
4 5 6

See Also:
data; restore

read dynamic statement

Syntax:
read dynamic *deviceID,arrayName*

Description:
Use **read dynamic** to read the contents of a dynamic array from a disk file. Before executing this read statement, you must dimension the dynamic array. The following example creates and fills a dynamic array, writes the array to disk, then reads it back into memory.

```
dim x
dynamic myAry(_maxLong)
for x = 1 to 100
  myAry(x) = x
next
open "O",#1,"Dynamic Array Test"
write dynamic #1,myAry
close #1
kill dynamic myary
open "I",#1,"Dynamic Array Test"
read dynamic #1,myAry
close #1
for x = 1 to 10
 print myary(x)
next
kill "Dynamic Array Test"
```

See Also:
dynamic; write dynamic

read field statement

Syntax:
read field [#]*deviceID,handleVar*

Description:
This statement creates a new relocatable block in memory, reads data from the open file or serial port specified by *deviceID* into the new block, and returns a handle to the block into *handleVar*, which must be a long-integer variable or a Handle variable.
The data in the file (or coming in through the serial port) must be in a particular format in order to be read properly. The first 4 bytes (at the current "file mark" position) must be a long integer which indicates the size of the block to create. This should be immediately followed by the data which is to go into the block. This is the format in which the write field statement writes to a file; almost always, the data you read with read field will have been created using a write field statement.

Note:
Your program is responsible for disposing of the handle returned in *handleVar* when you're finished using the block. You can use a statement such as DisposeH or kill field to do this.

See Also:
write field; read file; open; kill field; DisposeH

read file   **statement**

Syntax:
read file [#]*deviceID,address&,numBytes&*

Description:
This statement reads *numBytes&* bytes from the open file or serial port specified by *deviceID* (starting at the current "file mark" position), and copies them into memory starting at the address specified by *address&*. This is the fastest way to read large amounts of data from a file; it's also well suited for reading data whose format you may not know in advance.

Example:
This program fragment quickly loads an array with the data read from a file. It's assumed that the binary image of the array was previously saved to the file using a statement like write file (see the example accompanying the write file statement).
```
_maxSubscript = 200
dim myArray%(_maxSubscript)
arrayBytes& = (_maxSubscript+1) * sizeof(int)
read file #1, @myArray%(0), arrayBytes&
```

Note:
If **read file** attempts to read past the end of the file (because *numBytes&* was too large), FutureBasic generates an error.

See Also:
open; read#; read field; write file

read# **statement**

Syntax:
read# *deviceID* , { *recVar* | *numVar* | *strVar$* ; *len* } ¬
[, { *recVar* | *numVar* | *strVar$* ; *len* } . . .]

Description:
This statement reads data from the open file or serial port specified by *deviceID*, and stores the data into the indicated variable(s).
You can read the data into record variables (*recVar*), into numeric variables (*numVar*) or into string variables (*strVar$*), or any combination of these. If you specify *recVar* or *numVar*, the statement reads a number of bytes equal to the size of the variable, and stores the bytes directly into the variable's location in memory, without doing any data conversion. If you specify *strVar$;len*, the statement reads *len* bytes, and stores them into *strVar$* as a Pascal string of length *len* (*len* can be any numeric expression, but its value should not exceed 255). The read operation begins at the current location of the "file mark," and the file mark is advanced as each item is read. If **read#** attempts to read past the end of the file, FutureBasic generates an error.
Because **read#** copies the file's bytes directly into memory without conversion, it's best suited for reading "binary" information, such as that created by the write statement. To read file data which is formatted as text, it's usually better to use the input# statement. To read an arbitrary number of bytes into a block of memory, use the read file statement.

See Also:
write; input#; read file; eof; open; sizeof

rec   **function**

Syntax:
currentRecord = **rec** (*fileID*)

Description:
This function returns the record number of the record where the "file mark" is currently located, in the open file specified by *fileID*. The first record in the file is considered Record #0.

To calculate the record number, **rec** uses the record length that was specified in the open statement when the file was opened. If no record length was specified, a default length of 256 bytes is used. If you specified a record length of "1" when you opened the file, **rec** returns the file mark's byte position within the file.

Note:
To determine the file mark's offset from the beginning of the record, use the loc function.

See Also:
record; loc; open

FutureBasic 5

record   **statement**

Syntax:
record [#] *fileID* , *recordNum* [, *positionInRecord*]

Description:
This statement sets the "file mark" position, in the open file specified by *fileID*. The position of the file mark determines the location in the file where the next input or output operation will occur.

If you omit the *positionInRecord* parameter, **record** places the file mark at the beginning of the record indicated by *recordNum* (the first record in the file is considered Record #0). If you specify *positionInRecord*, the file mark is placed at an offset of *positionInRecord* bytes past the beginning of the indicated record.

record uses the record length that was specified in the open statement when the file was opened. If no record length was specified, a default length of 256 bytes is used. If you specified a record length of "1" when you opened the file, you can set the file mark to a particular byte offset from the beginning of the file using a statement like this:

  record [#]fileID, byteOffset&

Note:
The file mark position is also moved automatically after every input or output operation on the file.

See Also:
rec; loc; open; lof

rem statement

Syntax:

```
[statement]  rem [remarks]
[statement]  '[remarks]
[statement]  //[remarks]
[statement]  /*[blockRemarks]*/ [statement]
```

Description:

The **rem** statement (and its variations) provide a way for you to insert remarks into the source code which are ignored by the compiler. Remarks have absolutely no effect on how your program runs, but they can be very useful in helping readers to understand how your code works.

If you use the **rem** keyword, or the apostrophe (') or double-slash (//) token, everything following it on the same line is treated as a remark; therefore, it's not possible to put a non-remark statement after remarks on the same line.

The apostrophe and double-slash variations work identically to each other. In the FutureBasic Editor, remarks that begin with the apostrophe, the double-slash, or the "/*" token are automatically tabbed to the Remarks column of the Editor window if they're preceded by a statement. Remarks that begin with **rem** are not tabbed.

When you use the "/*" and "*/" tokens, the "/*" indicates the beginning of the remarks, and the "*/" indicates the end of remarks. Because it uses two tokens, this variation allows you to do some things you can't do with other variations. Specifically:

You can write remarks that span multiple lines, using only one pair of tokens. For example:

```
/* This is the first line of remarks.
The remarks continue on this line.
This is the last line of remarks. */
```

One disadvantage of using the "/*" and "*/" tokens is that you have to make sure you don't inadvertently include a "*/" token in the middle of your remarks. If you do, the compiler will assume that the remarks end there. For example:

```
/* This is the first line of remarks.
We have /*accidentally*/ put a remark-closing token
in the middle of the remarks.
*/
```

When the compiler encounters this, it interprets the "*/" token following "accidentally" as the end of the remarks, and it attempts to compile the remainder of that line and the following line, leading to compile errors.

Note:

You cannot put remarks after a data statement on the same line. Everything following the data keyword on the same line is considered part of the data statement.

See Also:

data

rename **statement**

Syntax:
rename *oldPath$* {**to** | **,**} *newFileNameOnly$* [**,** *refNum%* [**,** *dirID&*]]

Description:
Changes the name of a file or folder from the name specified in *oldPath$* to the name specified in *newPath$*. The *oldPath$* and *newPath$* parameters are used along with *refNum%* and *dirID&* to determine the location of the item to be renamed: see Appendix A - File Object Specifiers, for more information. You cannot use **rename** to move an item into a different directory: *oldPath$* and *newPath$* must both refer to items in the same directory.

See Also:
Appendix A - File Object Specifiers

call <resource>; button

call <resource>; button

restore   **statement**

Syntax:
restore [*expr*]

Description:
This statement is used in conjunction with the data and read statements. It resets an internal pointer which tells FutureBasic where to find the next data item to read. This allows your program to read the same data item(s) more than once if necessary.

If you omit the *expr* parameter, the data pointer is reset to point to the first item in the first data statement. If you specify *expr*, the data pointer is reset to point to the item immediately following the *expr*-th item. Thus **restore** 1 points to the second item; **restore** 2 points to the third item; and so on.

Example:
```
data Able, Baker
data Kane, Dread
data Echo
for i = 1 to 5
  read x$
  print x$,
next
print
restore 3
read x$, y$
print x$, y$
```
program output:
```
Able Baker Kane Drea Echo
Dread Echo
```

See Also:
read; data

return **statement**

Syntax:
return [*"label"*]

Description:
You should include at least one **return** statement in every subroutine that is called by a gosub statement. **return** causes the subroutine to "exit"; that is, it causes execution to continue at the statement following the gosub that called the subroutine.
You may also return to a specific location using **return** `"label"`. This pops the return address from the stack, then jumps to the requested address.

See Also:
gosub

right$ and right$$ **function**

Syntax:
*subString$* = **right$** (*string$,numChars*)
s*ubContainer$$* = **right$$** (*container$$,numChars*)

Description:
This function returns a string or container consisting of the rightmost *numChars* characters of *string$* or *container$$*. If *numChars* is greater than the length of *string$* or *container$$*, the entire *string$* or *container$$* is returned. If *numChars* is less than 1, an empty (zero-length) string is returned.

Note:
You may not use complex expressions that include containers on the right side of the equal sign. Instead of using:
c$$ = c$$ + **right$$**(a$$,10)
Use:
c$$ += **right$$**(a$$,10)

Example:
print right$("Nebraska", 3)
program output:
ska

See Also:
left$; mid$; instr

rnd   **function**

Syntax:
randomInteger% = **rnd**(*expr%*)

Description:
This function returns a pseudo-random long integer uniformly distributed in the range 1 through *expr&*. The *expr&* parameter should be greater than 1, and must not exceed 65536. If the value returned is to be assigned to a 16-bit integer (*randomInteger%*), *expr&* should not exceed 32767. The actual sequence of numbers returned by **rnd** depends on the random number generator's "seed" value; see the random statement for more information. Note that **rnd**(1) always returns the value 1.

Example:
To get a random integer between two arbitrary limits *min* and *max*, use this statement:
randomInteger = **rnd**(max - min + 1) + min - 1
(Note: max - min must be less than or equal to 65536.)
To get a random fraction, greater than or equal to zero and less than 1, use this statement:
frac! = (**rnd**(65536)-1)/65536.0
To get a random long integer in the range 1 through 2,147,483,647, use this statement:
randomInteger& = ((**rnd**(65536) - 1)<<15) + **rnd**(32767)

See Also:
random; maybe

route   **statement**

Syntax:
To re-route text and graphics:
route `[#]_toPrinter`
route `[#]_toScreen`
route `[#]_toBuffer [+ 0... 4 ]`
To re-route text:
route `[#]` *serialPort*
route `[#]` *fileID*

Description:
This statement causes text printed by subsequent `print` statements to be sent to the indicated device. If you specify `_toScreen` or `_toPrinter`, subsequent drawing commands are also sent to the indicated device. If you specify `_toBuffer`, only text printing commands are sent to the indicated device.

Using route _toPrinter
route `_toPrinter` opens a new print job unless a print job is already open. When you first open a new print job, the printer adopts the font and graphics characteristics that are in effect in the current screen window. Subsequent `print` statements, as well as graphics commands such as `box`, `circle` and `plot`, are sent to the printer. Subsequent statements which affect the appearance of text and graphics, such as `text`, `pen` and `color`, apply to the printed output. Subsequent QuickDraw Toolbox commands such as `FrameRect` apply to the printed output.
Note: To actually put out the current printer page, use the `clear lprint` statement after sending text and drawing commands to the printer.

Using route _toScreen
After using **route** to direct output to the printer, or to a serial device or a file, you can use **route** `_toScreen` to re-direct output back to the screen window again.
Note: Statements which affect the appearance of text and graphics, such as `text`, `pen` and `color`, apply separately to the screen window and to the printer. The settings in one device don't affect the settings in the other, except when you first open a new print job.

Using route serialPort and route fileID
These statements cause text printed by subsequent `print` statements to be sent to the specified open device. They don't affect the destination of graphics commands. This group of statements:
route `deviceID`
print `itemList1`
print `itemList2`
`:`
print `itemListN`
route `_toScreen`
has the same effect as this group of statements:
print `#deviceID, itemList1`
print `#deviceID, itemList2`
`:`
print `#deviceID, itemListN`

Using route _toBuffer
You can use the FutureBasic **route** `_toBuffer` statement to print information directly to a handle. You may use any one of five handles by setting the route statement in the range of `_toBuffer` through `_toBuffer + 4`. When routed to a buffer, only text commands are sent to the handle. Graphic commands are ignored.
Information printed to a buffer ends up in one of five handles stored as globals in the FutureBasic runtime. This array is named `gFBbuffer(n)` where `n` is a numeric expression in the range of 0...4. The following example prints text to buffer number 2.
route `_toBuffer + 2`
print `"Hello"`
print `"Goodbye"`
route `_toScreen`
When this snippet has been executed, `gFBbuffer(2)` will contain a handle that points to a moveable block of 14 bytes:
`Hello<CR>Goodbye<CR>`

Note that carriage returns are added or suppressed in writing to a buffer just as they would be in writing to a file or to the screen.
If you want to dispose of a buffer, use `DisposeH`.
DisposeH`(gFBbuffer(2))`

Note:
The `window` statement and the `window output` statement implicitly execute a **route** `_toScreen` statement. Also, any user action which activates a screen window (such as clicking on its title bar) will implicitly cause a **route** `_toScreen`. If you want to make sure that output is not inadvertently directed to the screen window, you should check for `_wndActivate` events in your dialog-event handling routine, and re-direct the output as necessary.

See Also:
open; print; print#; clear lprint; close lprint; text; pen; color; long color; box; circle; plot; on dialog

run   **statement**

Syntax:
run *path$* [*, refNum%* [*, dirID&*]]

Description:
This statement launches the application specified by *path$*, *refNum%* and *dirID&*, and puts it into the foreground, making it the active process. The program that launched the application does not quit, but is put into the background.

See Appendix A - File Object Specifiers, to see how the *path$*, *refNum%* and *dirID&* parameters are interpreted.

scroll **statement**

Syntax:

scroll *(h1 , v1) – (h2 , v2) , hPixels% , vPixels%*

Description:

This statement scrolls the pixels in the rectangle *(h1 , v1) – (h2,v2)*, by a horizontal distance of *hPixels%* and a vertical distance of *vPixels%*, in the current output window. Positive values of *hPixels%* scroll to the right; positive values of *vPixels%* scroll downward; use negative values to scroll in the opposite direction(s). **scroll** is often used in conjunction with scroll bars to view a graphic that is too large to fit entirely in the window.

scroll generates a `_wndRefresh` event, which your program can detect in its dialog-event handling routine the next time a `HandleEvents` statement is executed. Your dialog-event handling routine should respond to this event by redrawing the portion of the window that was left blank by the scroll. If you don't redraw this area explicitly, it will be filled with the window's current background color and pattern.

See Also:

scroll button; on dialog

Syntax:
scroll button `[#]`*idExpr* ¬
  `[,[`*current*`][,[`*min*`][,[`*max*`][,[`*page*`][,[`*rect*`][,`*type*`]]]]]]`

Description:
The **scroll button** statement puts a new scrollbar in the current output window, or alters an existing scrollbar's characteristics. The parameters are interpreted as follows:

Parameter	Description
idExpr	An integer which identifies the scrollbar. If Abs(idExpr) is different from the ID numbers of all buttons and all other scrollbars in the current window, a new scrollbar is created, and is assigned an ID number equal to Abs(idExpr). If Abs(idExpr) equals the ID of an existing scrollbar, the scrollbar's characteristics are altered.
current	This sets the current "value" of the scrollbar, which, along with min and max, determines the position of the scrollbar's "thumb." It must be greater than or equal to min, and less than or equal to max.
min	The minimum value that the scrollbar can have. For vertical scrollbars, this corresponds to a thumb position at the top of the bar; for horizontal scrollbars, it corresponds to a thumb position at the left side of the bar. min must be in the range -32768 through +32767.
max	The maximum value that the scrollbar can have. For vertical scrollbars, this corresponds to a thumb position at the bottom of the bar; for horizontal scrollbars, it corresponds to a thumb position at the right side of the bar. max must be in the range -32768 through +32767, and must be greater than min.
page	This specifies the amount by which the scrollbar's value should change when the user clicks in the areas between the thumb and the scrollbar's end-arrows. Must be non-negative.
rect	For scrollbars of type _scrollOther, the rect parameter specifies the rectangle that defines the size and position of the scrollbar. You can specify it in either of two ways: (x1,y1)-(x2,y2) - Coordinates of two diagonally opposite points. rectAddr& - Address of an 8-byte rectangle structure. If the specified rectangle is wider than it is tall, the scrollbar becomes a horizontal scrollbar. If the rectangle is taller than it is wide, the scrollbar becomes a vertical scrollbar. The standard recommended width for a vertical scrollbar (or height for a horizontal scrollbar) is 16 pixels. Note: For scrollbars of type _scrollHorz or _scrollVert, the rect parameter is interpreted differently. See below for more details.
type	Specify one of the following: _scrollOther: The scrollbar occupies the rectangle specified in the rect parameter. _scrollVert: The scrollbar occupies the right edge of the window, and is resized as the window is resized. If you specify a rect parameter when creating the scrollbar, the top of the scrollbar is offset from the top of the window by y1 pixels. _scrollHorz: The scrollbar occupies the bottom edge of the window, and is resized as the window is resized. If you specify a rect parameter when creating the scrollbar, the left side of the scrollbar is offset from the left side of the window by x1 pixels. Note: _scrollVert and _scrollHorz scrollbars can only be put into windows of type _doc, _docZoom and _docNoGrow. If you try to create a _scrollVert or _scrollHorz scrollbar in any other type of window, the scrollbar won't appear.

To Create a New Scrollbar:

- Choose an *idExpr* value such that `abs(`*idExpr*`)` is different from the ID's of all existing buttons and scrollbars in the window.

Choose initial values for current, min, max and page. All of these parameters are optional; any of them that you omit will have the following default initial values:

- *current* = 0
- *min* = 0
- *max* = 255
- *page* = 16

- If creating a _scrollOther scrollbar, specify the *rect* parameter. This parameter is optional if you're creating a _scrollVert or _scrollHorz scrollbar.
- Specify the type. This parameter is optional; its default value is _scrollOther.

To Alter an Existing Scrollbar:

- Set *idExpr* to the ID number of an existing scrollbar in the window.
- If you want to change any of the current, min, max or page values, specify the corresponding parameters. Any of these that you omit won't have their values changed.
- If you want to change the rectangle of a _scrollOther scrollbar, specify the new rectangle in the rect parameter. If you omit this parameter, the rectangle won't change. NOTE: the rect parameter is ignored when you're altering a _scrollVert or _scrollHorz scrollbar.
- You can't alter the type of an existing scrollbar. This parameter is ignored if the scrollbar already exists.

To Activate or De-activate a Scrollbar:
You can use the button statement to activate (highlight) or de-activate (dim) an existing scrollbar.

- To activate it, use: button scrollbarID, _activeBtn
- To de-activate it, use: button scrollbarID, _grayBtn

Using the Scrollbar
To make the scrollbar useable, your program must call HandleEvents periodically. Among other things, HandleEvents tracks the motion and clicking of the mouse in the scrollbar, and moves the thumb in response to these user actions. Your program can also move the thumb explicitly by setting the *current* parameter in the **scroll button** statement.
Whenever the user moves the thumb, a dialog event of type _btnClick is generated. The "id" value for this event equals the scrollbar's ID. You can determine the thumb's current position using the button function:
thumbPosition = **button**(scrollBarID)

Note:
To remove a scrollbar, use the button close statement:
button close scrollBarID
To find out information about a scrollbar, use the button& function to get the scrollbar's control record.

See Also:
button&; button function; button statement; dialog function

select case or select switch **statement**

Syntax 1:
select [**case**] *targetExpr*
   case *testExpr* [, *testExpr* ...] [, *max32 testExpr*]
[statementBlock]
   [**case** *testExpr* [, *testExpr* ...] [, *max32 testExpr*]
[statementBlock]...]
   [**case else**
[statementBlock]]
end select

Syntax 2 (introduced in FB 5.6.1):
select switch *integerTargetExpr*
   case *compileTimeIntegerConstantExpr* [, *compileTimeIntegerConstantExpr* ...] [, *max32 compileTimeIntegerConstantExpr*]
[statementBlock]
   [**case** *compileTimeIntegerConstantExpr* [, *compileTimeIntegerConstantExpr* ...] [, *max32 compileTimeIntegerConstantExpr*]
[statementBlock]...]
   [**case else**
[statementBlock]]
end select

Syntax 3:
select [**case**]
   case *booleanExpr* [, *booleanExpr* ...] [, *max32 booleanExpr*]
[statementBlock]
   [**case** *booleanExpr* [, *booleanExpr* ...][, *max32 booleanExpr*]
[statementBlock]...]
   [**case else**
[statementBlock]]
end select

Description:
The **select case** statement marks the beginning of a "select block," which must end with an **end select** statement. You can use a select block to conditionally execute a block of statements based on a number of tests that you specify. When there is only one test, it's often more convenient to use a `long if...[xelse]...end if` block. A select block is useful when there are two or more conditions to test. If you use Syntax 1, *targetExpr* must be a numeric or string expression. Each *testExpr* has the following syntax:

   [=|<>|<|<=|>|>=] *expr*

where *expr* is an expression of the same type as *targetExpr*. When the select block executes, *targetExpr* is compared against each *testExpr* in order. If the *testExpr* does not include a comparison operator (<, >, etc.), then *targetExpr* is compared for equality with *expr*; otherwise *targetExpr* is compared with *expr* using the indicated operator.
If the comparison of *targetExpr* with *expr* is true, the `statementBlock` (if any) following that **case** statement is executed; then execution continues at the first statement after **end select**. If none of the comparisons in any **case** statement is true, the `statementBlock` (if any) following the **case else** statement is executed; then execution continues at the first statement after **end select**. Each `statementBlock` can consist of any number of executable statements, possibly including other **select...end select** blocks.
If you use Syntax 2, each *booleanExpr* must be a numeric expression that can be evaluated as "true" (nonzero) or "false" (zero). Typically, *booleanExpr* will be an expression that includes operators such as `and`, `or`, `>`, `<`, etc. When the select block executes, each *booleanExpr* is evaluated in order, until one is found that is nonzero ("true"). Then the `statementBlock` (if any) under the corresponding **case** statement is executed; then execution continues at the first statement after **end select**. If every *booleanExpr* is zero ("false"), the `statementBlock` (if any) following the **case else** statement is executed; then execution continues at the first statement after **end select**.

Note 1:
Syntax 2 is a specialized form that works only for integer expressions and integer compile-time constants. It translates to a C `switch` statement, which is more readable than the C translation of syntax 1. Another advantage is performance: a `switch` statement can often be compiled to a jump table.

Note 2:
Remember that the syntaxes work differently. It's a common mistake to do something like this:
select case `myVar%`
 case `myVar% = 3`
 :

```
  case myVar% = 7
:
  case else
:
end select
```

Here the programmer probably intended to compare the value of `myVar%` to the values 3 and 7; but that's not what this select block does. Instead, it compares the value of `myVar%` first to the value of "`myVar%=3`" (which is either -1 or 0), and then to the value of "`myVar%=7`" (which is either -1 or 0). This block should have been written in one of these ways:

Using Syntax 1:
```
select case myVar%
  case 3
:
  case 7
:
  case else
:
end select
```

Using Syntax 3:
```
select case
  case myVar% = 3
:
  case myVar% = 7
:
  case else
:
end select
```

See Also:
long if

SendAppleEvent   **statement**

Syntax:
SendAppleEvent *eventtype& , eventClass& , dataAddress& , ¬*
dataSize& , processName$

Description:
This statement is used in conjunction with other Apple Event commands available starting with Release 5. It allows you to send information of any size to another process. The parameters include the standard event type and class. The *dataAddress&* specifies where the Apple Event Manager will find the data and the *dataSize&* variable tells the length of that data.
The *processName$* parameter may be a null string (to send the information to all running processes) or it may be a specific process name. (See GetProcessInfo for an example of how to build a list of running processes.)

See Also:
HandleEvents; GetProcessInfo

SetSelect **statement**

Syntax:
SetSelect *startSelect% , endSelect%*

Description:
If the current output window contains an active edit field, this statement selects (highlights) a range of text in the field, or sets the position of the blinking insertion point. It also scrolls the selected text into view, if it's not already in view.
The *startSelect%* and *endSelect%* parameters refer to positions between characters in the field. Position 0 is just to the left of the first character; position 1 is between the first and second characters; and so on. If you specify a position greater than or equal to the number of characters in the field, it indicates a position just to the right of the last character. If *startSelect%* equals *endSelect%*, then no text is highlighted, but a blinking insertion point is paced at the indicated position.

Example:
If you specify **SetSelect** `0,0`, a blinking insertion point is placed at the beginning of the field's text. If you specify **SetSelect** `0,32767`, all of the text in the field is selected. If you specify **SetSelect** `32767,32767`, a blinking insertion point is placed at the end of the field's text. The following inserts or replaces the selected range with the contents of `newString$`:
SetSelect `startSelect%, endSelect%`
`tekey$ = newString$`

Note:
Text selection and insertion-pont placement are normally handled automatically by the `HandleEvents` statement, in response to the user's mouse and keyboard actions. Use **SetSelect** for special situations.
Use the `window(_selStart)` and `window(_selEnd)` functions to determine the active field's current selection range.

See Also:
edit field; tekey$ statement; window function

sgn   **function**

Syntax:
signOfExpr = **sgn**(*expr*)

Description:
Use this function to determine the "sign" of *expr*. **sgn** returns:

- 1 if expr is positive;
- 0 if expr is zero;
- -1 if expr is negative.

Example:
This for loop counts up if first < last, and counts down if first > last:

```
for x = first to last STEP sgn(last-first)
  print x
next
```

See Also:
for; abs

shutdown   **statement**

Syntax:
shutdown [*msg$*]

Description:
When used without the *msg$* parameter, **shutdown** behaves identically as the end statement. If the *msg$* parameter is included, the string in *msg$* is displayed in an alert box before the program quits.

See Also:
end; system statement

sin   **function**

Syntax:
theSine# = **sin**(*expr*)

Description:
Returns the sine of *expr*, where *expr* is given in radians. The returned value will be in the range -1 to +1. **sin** always returns a double-precision result.

Note:
To find the sine of an angle `degAngle` which is given in degrees, use the following:
```
    theSine# = sin(degAngle * pi# / 180)
```
where `pi#` equals 3.14159265359

See Also:
asin; cos; tan

sinh   **function**

Syntax:

result# = **sinh** (*expr*)

Description:

Returns the hyperbolic sine of *expr*. **sinh** always returns a double-precision result.

Note:

sinh (*x*) is defined as: $\dfrac{e^{x} - e^{-x}}{2}$

See Also:

asinh; cosh; tanh; exp

sizeof   **function**

Syntax:

dataSize = **sizeof** ({ *var* | *typeName* | *ptrType^* | *hdlType^^* })

Description:

This function returns the number of bytes of memory allocated for a particular variable *var*, or the number of bytes allocated for each variable of a particular specified type.

If you specify *typeName*, it should either be the name of a type defined previously in your program (in a `begin record` statement or a `#define` statement), or the name of one of FutureBasic's built-in types (such as `int`, `long`, `Rect`, etc.). **sizeof** returns the size of a variable of that type.

If you specify *ptrType^*, then *ptrType* should be the name of a type which was previously declared to be a pointer to some other type (in a `#define` statement). In this case, **sizeof** returns the size of the type that *ptrType* points to. Note that if you omit the "`^`" symbol, **sizeof**(`ptrType`) just returns the size of a pointer variable (usually 4).

If you specify *hdlType^^*, then *hdlType* should be the name of a type which was previously declared to be a handle to some other type (in a `#define` statement). In this case, **sizeof** returns the size of the type referenced by *hdlType*. Note that if you omit the "`^^`" symbols, **sizeof**(`hdlType`) just returns the size of a handle variable (usually 4).

Note:

sizeof(`stringVar$`) returns the number of bytes reserved in memory for the string variable `stringVar$`. This is not the same thing as `len(stringVar$)`.

If a variable `handleVar` contains the handle to a relocatable block (of a possibly unknown type), you can use the Toolbox function `GetHandleSize` to determine the size of the block.

See Also:

typeof; len; begin record; #define

sound end   **statement**

Syntax:
sound end

Description:
This statement stops the sound that was initiated with the latest `sound <frequency>` or `sound <snd>` statement. If the sound is an "`snd `" resource that you played using the `sound resName$` syntax or the `sound %resID%` syntax, the **sound end** statement also releases the resource.

See Also:
sound <frequency>; sound <snd>; sound%

sound <frequency>   **statement**

Syntax:
sound *pitch* , *duration* `[ , [` *volume* `] [ ,` *async* `] ]`

Description:
This statement plays a tone of the given pitch, duration and volume.
The *pitch* is expressed as a frequency in cycles per second. The frequency that you specify is converted to the nearest "MIDI note value," which determines the actual note that's played. Middle "C" corresponds to a frequency of 261.625.
The *duration* is expressed in ticks, and can range from 0 to 32,767. However, the toolbox sound commands require FutureBasic to translate the ticks into an integer to represent half-milliseconds. This means you can play a note that is no longer than 32.8 seconds.
the *volume* can range from 0 through 127. Specifying 0 will result in silence, and 127 will play the sound at the maximum volume specified in the "Sound" control panel. If you omit this parameter, it is treated as 127.
If *async* is `_zTrue`, the sound will play asynchronously. If *async* is `_false`, or you omit the parameter, the sound plays synchronously. When you play asynchronously, your program starts executing the next statement immediately while the sound is playing in the background. When you play synchronously, the next statement in your program does not execute until the sound has finished playing.
If you are playing notes asynchronously, and your execute a second **sound** statement while another sound is still playing in the background, the new sound won't start playing until the first sound finishes. Note that on some machines, this technique can result in lost sounds. When playing asynchronously it's better to use the `sound%` function to determine when one sound has ended, before attempting to play the next sound.
One way to play sound frequencies is to use negative numbers (from -1 through -127) to represent the note that you wish to play. The table below shows how to use these values.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1				0	1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	60	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

Using the midi table for a guideline, we can create a version of "pop goes the weasel" as follows:
print `"Pop! "; : ` **sound** `-70, 45 ,,_false`
print `"Goes "; : ` **sound** `-64, 30 ,,_false`
print `"the "; : ` **sound** `-67, 15 ,,_false`
print `"wea"; : ` **sound** `-66, 40 ,,_false`
print `"sel "; : ` **sound** `-62, 45 ,,_false`

See Also:
sound%; sound end; sound <snd>

sound <snd>   **statement**

Syntax:
sound *soundIDstring$*
sound *%resIDNumber*
sound *&soundHandle&*

Description:
This statement plays a synthesized sound which is in the "snd" format. The *soundIDstring$* parameter identifies the sound you want to play; you can construct this string in any of the following ways:

- By resource name: Set *soundIDstring$* to the name of an "snd " resource in a currently open resource file. Don't use a resource name that begins with "%" or with "&", or it will be confused with the other forms discussed below.
- By resource ID number: If `%myResID` is the resource ID number of an "snd " resource in a currently open resource file, you can prefix the number with a percent sign:

      sound `%myResID`

- By "snd " handle: If `&mySoundHandle&` is a handle to a sound in "snd " format, you can construct the `soundIDstring$` parameter as follows:

      `sound &mySoundHandle&`

If FutureBasic needs to load a resource to play the sound, it does not automatically release the resource after the sound stops. In this case, you should either use purgeable "snd " resources, or you should execute the `sound end` statement to force the resource to be released. Sounds played using the **sound** `<snd>` statement are always played asynchronously; that is, your program continues to the next statement immediately, while the sound is playing in the background. However, if you execute a second **sound** `<snd>` statement while a previous sound is still playing in the background, the new sound won't start playing until the first sound finishes.

Example:
If you want your program to "wait" until a sound finishes before continuing, you can use
the `sound%` function:
sound `"reallyLongSound"`
while sound%
    `'(stays in this loop until sound finishes)`
wend

See Also:
sound%; sound <frequency>; sound end

sound%   **function**

Syntax:
soundIsPlaying = **sound%**

Description:
This function returns _zTrue if a sound is currently playing, or _false otherwise. This applies to sounds that you play using the sound <frequency> statement or the sound <snd> statement, but does not apply to Text-to-speech sounds.

Example:
The **sound%** function is useful for determining when an asynchronous sound has finished playing.
sound "Quack"
startTime& = **fn TickCount**
while sound%
wend
endTime& = **fn TickCount**
print "That sound lasted"; endTime& - startTime&; "ticks."

Note:
To determine whether a Text-to-speech sound is currently playing, use the Toolbox function fn SPEECHBUSY.

See Also:
sound <frequency>; sound <snd>; sound end

space$ **function**

Syntax:

*stringOfSpaces$* = **space$**(*numChars*)

Description:

Returns a string consisting of *numChars* space characters. *numChars* must be in the range 0 through 255. **space$**(0) returns an empty (zero-length) string.

See Also:

print; string$

spc **function**

Syntax:
spc (*numChars*)

Description:
When used with print or lprint, this outputs the number of spaces specified by *numChars*.

Example:
```
print "Hello" spc(10) "out there."
program output:
Hello out there.
```

See Also:
print; lprint; string$

sqr **function**

Syntax:
squareRoot# = **sqr** (*expr*)

Description:
Returns the square root of expr. **sqr** always returns a double-precision result.

stop   **statement**

Syntax:
stop [*string$*]

Description:
This statement interrupts program execution and displays a dialog box that shows the line number where the **stop** statement appeared, and gives the user the option of continuing or stopping the program. If the optional string parameter is used, it will be displayed as a message in the alert. If no string parameter is used, FutureBasic displays the name of the file in which the **stop** statement was used.
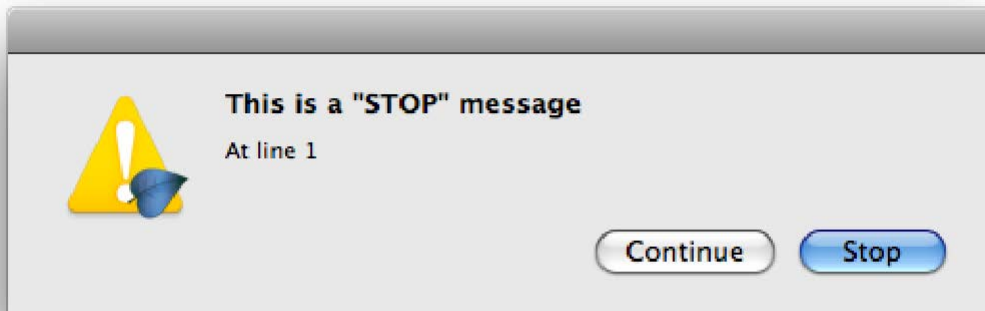
If the user clicks the "Stop" button, FutureBasic calls your stop-event handling routine (if you've designated one using the `on stop` statement), and then your program ends. If the user clicks the "Continue" button, execution continues at the next statement following the **stop** statement.
stop is mainly useful as a debugging tool. It's not recommended for use in the "production" version of your program.

See Also:
end

str#   **function**

Syntax:
stringFromList$ = **str#**(*resourceID%*, *index%*)

Description:
This function returns a string element from a resource of type "str#". The *resourceID%* should specify the resource ID number of an "str#" resource in a currently open resource file. *index%* indicates which string element to get; the first element is numbered 1. If the "str#" resource isn't found, or if *index%* is greater than the number of strings in the resource, the **str#** function returns an empty (zero-length) string.

"str#" resources hold lists of strings. Among other things, they're useful in helping you to localize your program so that it supports the native language of your user. You typically use a program like ResEdit to create an "str#" resource and its strings.

Note:
You can use the following function to determine how many strings are in an "str#" resource.

```
local fn GetSTRcount(resID%)
    resHndl& = fn GETRESOURCE(_"str#",resID%)
    long if resHndl&
resCount% = {[resHndl&]}'Get 1st word in block
  xelse
resCount% = 0 'Couldn't get the resource
    end if
end fn = resCount%
```

See Also:
compile _strResource; str&

str&   **function**

Syntax:

stringFromHandle$ = **str&** (*handle&, index%*)

Description:

This function returns a string element from a handle that is in the same format as a resource of type "str#". The *handle&* should specify the location of the memory block. *index%* indicates which string element to get; the first element is numbered 1. If the handle isn't found, or if *index%* is greater than the number of strings in the resource, the **str&** function returns an empty (zero-length) string.

Example:

This example creates and fills a handle in the form of a str# resource and displays the results..

```
dim sHndl as handle
dim x as word
// create an empty str# style handle
sHndl = fn NewHandleClear(2)
// Fill the handle with ASCII strings (1-10)
for x = 1 to 10
  def ApndStr("This is number"+str$(x), sHndl)
next
// Display the handle using str&
for x = 1 to 10
 print str&(sHndl,x)
next
// We made it. We must dispose of it.
DisposeH(sHndl)
```

See Also:

compile _strResource; str#

str$ **function**

Syntax:
*numberString$* = **str$**(*numericExpr*)

Description:
This function returns a string consisting of the characters in the decimal representation of *numericExpr*. If the number is non-negative, then *numberString$* will also have a leading space character (if the number is negative, the first character will be the minus sign). **str$** formats the number in a reasonable way; if you need the number to appear in a special format, use string functions such as using, hex$, etc. Generally speaking, **str$** is the inverse of the val function.

See Also:
val; print; hex$; using

string$ & string$$ **function**

Syntax:
*stringOfChars$* = **string$** (*numChars* , { *char$* | *asciiValue%* })
container$$* = **string$$** (*numChars* , { *char$ | *asciiValue%* })

Description:
This function returns a string or a container consisting of *numChars* repetitions of a single character. If you specify a string (*char$*) in the second parameter, the first character of *char$* is repeated. If you specify a number (*asciiValue%*) in the second parameter, **string$** repeats the character whose ASCII value is *asciiValue%*. *numChars* must be in the range 0 through 255; if *numChars* equals zero, **string$** returns an empty (zero-length) string.

Example:
print string$(12, "log")
print string$(9, 70)
program output:
LLLLLLLLLLLL
FFFFFFFFF

See Also:
space$; asc; chr$; BlockFill; Appendix F - The ASCII Character Codes

stringlist **statement**

Syntax:
stringlist on
stringlist off
stringlist

Description:
Controls the localizability of string literals (step 1 below). The default state is off. When source code is compiled, FutureBasic needs to know where to store quoted strings. Quoted strings are those that appear in the source code like `"Hello"`. The strings are placed in the appropriate *.lproj folder in the app's internal folder hierarchy. A string literal will be localized if:

[1] it is encountered while stringlist is on.
[2] a Localizable.strings file is present in the appropriate *.lproj folder of the app's internal folder hierarchy.
[3] the file contains a key that matches the original string literal.
[4] the value corresponding to the key has been edited to be localized.
otherwise the original unlocalized string is used.

A localizable string takes up more memory than a non-localizable one, and access to it is much slower. Recommended practice is to place each string (or group of strings) that you want localizable between 'stringlist on' and 'stringlist' directives:

```
stringlist on          // turn localizability on
print "Hello, world!"  // localizable
stringlist             // revert to previous state
plusSign$ = "+"        // not localizable
```

FBtoC's menu command 'File > Generate Localizable.strings' creates a file in the built_temp folder, with an entry for each localizable string. An entry consists of a comment (file: line) to assist the translator, followed by "key = "value";

```
/* HelloWorld.bas: 4 */
"Hello, world!" = "Hello, world!";
```

After editing the value for French.lproj:

```
"Hello, world!" = "Bonjour, le monde!";
```

Note:
stringlist statements are not nestable.
stringlist statements don't apply for string constants.

Reference:
Internationalization Programming Topics

swap   **statement**

Syntax:
swap *variable1 ,   variable2*

Description:
swap exchanges the contents of the two indicated variables. Both variables must be of the same type.

Example:
varOne% = 1200
varTwo% = 999
print varOne%, varTwo%

swap varOne%, varTwo%

print varOne%, varTwo%
program output:
1200 999
999 1200

See Also:
let; Appendix C -Data Types and Data Representation

system   **function**

Syntax:

systemInformation = **system** (*whichInfo*)

Description:

This function returns various kinds of information about the system and about the current application. Set *whichInfo* to one of the values shown here:

whichInfo	Value	System(whichInfo) returns:
_lastCurs	0	"CURS" resource ID number of the current cursor (if it is a resource).
_aplVol	1	A working directory reference number for the folder that contains the application file. This is useful in case you want to read or write files in the same folder as the application.
_sysVol	2	A working directory reference number for the System folder. (Note: to access special folders within the System folder, the preferred method is to use the Toolbox function FindFolder.)
_macPlus	3	System(_macPlus) always returns 0. This is retained for compatibility with older versions of FutureBasic.
_aplRes	4	Reference number for the Application file's resource fork (which is always open while your application is running).
_memAvail	5	Number of free bytes currently available in the application's heap. Returns the same value as Mem(_freeBytes).
_scrnWidth	6	The width of the main monitor, in pixels.
_scrnHeight	7	The height of the main monitor, in pixels.
_sysVers	8	An integer representing the current System version number. For example, 761 represents version 7.6.1.
_aplActive	9	Returns a positive value if the application is the currently active (foreground) process. Returns a negative value if the application is currently in the background. Note that FutureBasic3 also generates a dialog event every time your application is moved to the foreground or to the background (see the Dialog function).
_maxColors	10	Maximum color bit-depth available on the main monitor. Use the expression (2 ^ System(_maxColors)) to get the actual number of available colors.
_crntDepth	11	Current color bit depth on the main monitor. Use the expression (2 ^ System(_crntDepth)) to get the actual number of available colors.
_cpuType	12	A code indicating the machine's CPU type, for 68k CPU's. Return values: _env68000 (1); _env68010 (2); _env68020 (3); _env68030 (4); _env68040 (5). Note: this value is unreliable for PPC CPU's: to determine
_machType	13	A code indicating the machine type. Use machName$ = Str#(-16395, System(_machType)) to determine the machine's name.
_aplFlag	14	System(_aplFlag) returns _false if the project was run and _zTrue if the project was built.
_cpuSpeed (PPC only)	15	System(_cpuSpeed) returns the current clock speed (in megahertz) of the microprocessor.
_clockSpeed (PPC only)	16	System(_clockSpeed) returns the Gestalt clock speed (in megahertz) of the microprocessor.
_lastCursType (Appearance Manager)	17	This function may return 0 (plain), _themeCursorStatic or _themeCursorAnimate.
_aplvRefNum (Appearance Manager)	18	This is the volume reference number of the running application.
_aplparID (Appearance Manager)	19	This is the volume parent ID number of the running application.

Note:
The following argument constants are no longer supported:
_lastCurs [0]
_aplVol [1]
_sysVol [2]
_cpuType [12]
_machType [13]
_aplFlag [14]
_lastCursType [17]

Many other kinds of system-wide information can be obtained using the Toolbox function fn GESTALT.

system statement   statement

___

Syntax:
system *ExistingType nameOfConstVarOrMacro* `[ ,  nameOfConstVarOrMacro2 . . . ]`

Description:
In FutureBasic version 4 and prior, this was a synonym for the end statement; In FutureBasic version 5 onwards, it no longer serves that purpose. Now it is used, mainly in Headers files, to declare macros and extern constants in Carbon or other frameworks.

Example:

Taken from headers:
system CFDictionaryKeyCallBacks kCFTypeDictionaryKeyCallBacks
system CFBooleanRef kCFBooleanTrue
system CFBooleanRef kCFBooleanFalse

General:
system double INFINITY
print INFINITY
stop

tan   **function**

Syntax:
theTangent# = **tan**(*expr*)

Description:
Returns the tangent of *expr*, where *expr* is given in radians. **tan** always returns a double-precision result.

Note:
To find the tangent of an angle degAngle which is given in degrees, use the following:
    theTangent# = **tan**(degAngle * pi# / 180)
where pi# equals 3.14159265359.

See Also:
atn; sin; cos

tanh   **function**

Syntax:

result# = **tanh** (expr)

Description:

Returns the hyperbolic tangent of *expr* where *expr* is given in radians. **tanh** always returns a double-precision result.

Note:

tanh (*x*) is defined as: $$\dfrac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

See Also:

Asinh; Cosh; Sinh; Tan; Exp

tekey$ function

Syntax:
*char$* = **tekey$**

Description:
If there is an active edit field in the current window, **tekey$** returns a 1-character string indicating a key that the user pressed. You should check the value of **tekey$** within your edit-event handling routine, each time that routine is called.
When you use an edit-event handling routine, the user's keystrokes are not transmitted directly to the edit field. Based on the value returned by **tekey$**, your edit-event handling routine can decide what to do with the keypress; it can transmit it to the field unaltered, or transmit a different character, or perform some other action. Your routine can use the **tekey$** statement to transmit the keypress (or some other character) to the edit field, if desired. **tekey$** values that correspond to the backspace key, and the four arrow keys should generally be transmitted to the field unaltered.
Note that any keypresses that can't alter the contents of the field nor move the insertion point are generally not reported to your edit-event handling routine. Keypresses that will not be reported include the following:

- Arrow keys, if the insertion point is already as far as it can go in the indicated direction (use the `dialog` function (event types `_efLeftArrow`, `_efRightArrow`, `_efUpArrow` and `_efDownArrow`) to detect arrow keys in this situation);
- Command-key equivalents for menu items;
- Command-period.
- The "Tab" key (use the `dialog` function (event type `_efTab` or `_efShiftTab`));
- The "Clear" key (use `dialog` function (event type `_efClear`));
- The "Return" key, if the field is one of the "NoCR" types (use the `dialog` function (event type `_efReturn`));

See Also:
dialog function; tekey$ statement

tekey$ **statement**

Syntax:
**tekey$** = *string$*

Description:
If there is an active edit field in the current window, *string$* is inserted into the field at the current insertion point, or replaces the currently highlighted text in the field. The insertion point is then placed at the end of the newly-inserted text, and the text is scrolled into view if it is not already in view.

See Also:
tekey$ function; edit field

text   **statement**

Syntax:

text [*font%*][, [*size%*][, [*face%*][, *copyMode%*]]]

Description:

This statement sets the text characteristics for the current output window or printer. It affects subsequent `print` statements (in the current window or printer), subsequent `lprint` statements, and subsequently-created edit fields (in the current window). It does not change the appearance of any existing text, except in buttons created using the `_useWFont` type modifier (see the button statement).

Each of the parameters is optional. If you omit a parameter, the corresponding characteristic won't be changed. The parameters are interpreted as follows:

font%

A number which identifies the font family. Certain common fonts have standard numbers which are identified by these constants:

```
_newYork _venice _geneva
_monaco _times _helvetica
_courier _symbol
```

The following constants are also available:

`_sysFont` (System font; usually Chicago or Charcoal)

`_applFont` (Default application font; usually Geneva)

For other fonts, the best way to determine the font number is to pass the font's name to the `GETFNUM` Toolbox procedure, as here:

call GETFNUM(*fontName$*, *font%*)

When you do this, the font's number is returned in the `font%` variable, which you can then pass to the **text** statement.

size%

The font size, in points. This usually gives some indication of the height in pixels of the tallest character; however, you shouldn't rely on this.

* *face%*

The text style. To set the face to "plain," set this parameter to zero; otherwise, you can

set it to the sum of any of the following bitmask values:

```
_boldBit% _outlineBit%
_italicBit% _shadowBit%
_ulineBit% _condenseBit%
_extendBit%
```

Example:

text `_geneva, 12, _boldBit% + _italicBit%`

copyMode%

This determines how the text interacts with the existing background. The most commonly used values are `_srcCopy` (the entire character's rectangle replaces the background), and `_srcOr` (only the character's glyph replaces the backgound). Other transfer modes include:

```
_srcXor _addOver
_srcBic _subPin
_notSrcCopy _transparent
_notSrcOr _adMax
_notSrcXor _subOver
_notSrcBic _adMin
_blend _grayishTextOr
_addPin _ditherCopy (mask)
```

Note:

Each screen window maintains its own text characteristics separately from the others. The **text** statement affects only the characteristics in the current window.

To change the characteristics of text in existing edit fields, use the `edit text` statement.

To change the color of subsequently printed text, use the `color` or `long color` statement.

See Also:

edit text; route; print; button statement; color; long color

threadbegin   **statement**

Syntax:
threadbegin fn *name [, parameter& [, stackMin&]]*

Description:
What exactly is a thread? It is a function that performs a task. The difference between a threaded function and any other function is that the threaded version runs in the background. Your program (and other running processes) advance without interruption as the threaded task is performed.
There are many reasons for using threaded functions. One that is becoming popular relates to functions that access Internet connections. Since logging in and downloading often take a good bit of time, it makes sense to hand such a task to a threaded function and go on about your business. You might handle other tasks like complex calculations, lengthy sorts, or time-consuming file loads.
When a threaded function is called, it executes without stopping until it is complete -- just like any other function. But at some point inside of the function, you yield to other processes by calling `threadstatus`. The `threadstatus` statement allows other actions to take place and optionally sets a timed delay for the number of ticks that should elapse before your function regains control.
The *parameter&* passed to a thread may be any long integer required by your program. It is for your use. If a parameter is sent, you should accept it in the named local funtion. For instance, if a parameter were used in the example below, we would create the funtion with **local fn** `myThread(param` **as long** `)` instead of just plain **local fn** `myThread`.
If the *stackMin&* parameter is omitted, FutureBasic sets a minimum stack space of 131072 bytes (128K). You may experiment with larger or smaller values for your specific needs.

Example:
This simple example creates a thread that prints a string of text, piece by piece, in a `for`/`next` loop. The `threadstatus` call installs a callback time of 10 ticks which makes the text appear slowly. You can type into the edit field as the threaded text is placed on the screen.

```
local fn myThread
   dim t$,x, abort
   t$ = "Hello. I am a thread. "
   t$ += "I execute in the background."
   for X = 1 to len(t$)
print @(1,1) left$(t$,x);
abort = threadstatus(10)//call me in 10 ticks
if abort != 0 then exit fn
   next
end fn
window 1
text _sysfont,12,0,0
print@(1,3)"Type into the edit field"
edit field 1,"",(8,80)-(200,100)
threadbegin fn myThread
do
   HandleEvents
until 0
```

See Also:
threadstatus; timer; on timer

threadstatus **function**

Syntax:
abortBoolean = **threadstatus**[(*ticks&*)]

Description:
In a threaded function, it is necessary to tell the Thread Manager when you wish to yield to other running processes. A thread which yields very little time will run faster, but will cause all other operations on the computer to run at a slower rate.
The **threadstatus** function returns a Boolean result to indicate whether or not the thread should continue operation. One reason that a thread might be asked to cease operation would be if the computer was about to shut down. When you receive a non-zero result from the **threadstatus** statement, you should exit the threaded function immediately.

See Also:
threadbegin; timer; on timer

time$ **function**

Syntax:
time$ `[( formatString$ )]`

Description:
time$ returns a string based on the current time. It may be used both with and without a `formatString$`.

Using **time$** *without* a `formatString$` returns the current time as an 8-character string containing two digit numerals each for hour, minutes and seconds, separated by colon marks, specifically in "hh:mm:ss" 24-hour format.

Using **time$** *with* a `formatString$` returns a string based on the current time and formatted based on `formatString$`.

The `formatString$` must contain Unicode Date and Time symbols as shown below and in Appendix I - Data & Time Symbols.

Example:

print time$
print time$`("hh:mm:ss")`
print time$`("h:mm a")`
print time$`("h:mm a zzz")`

23:46:49
11:46:49
11:46 PM
11:46 PM GMT-07:00

More Date & Time symbols can be found in Appendix I - Data & Time Symbols.

See Also:
date$; Appendix I - Data & Time Symbols

timer   **statement**

Syntax:
timer = *interval*

Description:
If you have designated a timer-event handling routine (using the on timer statement), the **timer** statement alters the interval at which timer events occur. If timer events have not yet been initiated (because your on timer statement specified an interval of zero), the **timer** statement also initiates timer events.

If the *interval* parameter is greater than zero, it specifies the interval in seconds. If *interval* is less than zero, then abs(*interval*) specifies the interval in ticks (a tick is approximately 1/60 second). Setting *interval* to zero disables the timer. Fractional values of *interval* are acceptable. Timer firings are not queued; they are lost if your application does not handle events for times greater than the *interval*.

See Also:
on timer

tool_arg function   **function**

See the Appendix J - Command Line Tools page.

tool_argc function   **function**

See the Appendix J - Command Line Tools page.

tool_argv function   **function**

See the Appendix J - Command Line Tools page.

tool_getenv function   **function**

See the Appendix J - Command Line Tools page.

toolbox   **statement**

Syntax:
toolbox [**fn**] *functionName* [(*arg1* [, *arg2* . . .])] [= *returnedValueType*]

Description:
The toolbox statement declares a function to the FBtoC translator. It effectively provides a forward definition for accessing either Carbon frameworks calls (what we traditionally refer to as Apple toolbox calls - hence the name) or FutureBasic version 5 or later runtime functions. Toolbox statements are non-executable. They typically reside in Headers files whose names begin with "Tlbx". The location and naming convention is not an absolute requirement and toolbox statements may be placed anywhere before the calls that use them. Toolbox statements may declare either a function with a return value or a procedure with no return value. A function is defined with 'toolbox fn xxxxx' and a procedure omits the 'fn' to simply read as 'toolbox xxxxx'.

Function Name (or Procedure name if there is no return variable)
Unlike most FutureBasic coding, the name of a toolbox function is case sensitive and must match exactly the name in either the Carbon framework or FutureBasic version 5 runtime. Xcode documentation is the best source for finding the correct case-sensitive spelling of Carbon calls and the FutureBasic version 5 runtime source is included with the FutureBasic version 5 package.

typeof												**function**

Syntax:
dataType = **typeof**({ *variable* | *typeName* })

Description:
When FutureBasic compiles your program, it associates a unique integer with each data type that your program uses. The **typeof** function returns the integer that's associated with the specified type.
typeName should be the name of a type that was previously defined in a `begin record` statement or a `#define` statement; or the name of one of FutureBasic's built-in types (such as `int`, `long`, etc.).
variable can the name of any variable. In this case, **typeof** returns the type ID number associated with the variable's type. Note that if the variable was not previously declared in a `dim` statement, and has no type-identifier suffix, **typeof** will assume that the variable's type is the default type (which is `int` unless a `def<type>` statement applies).

Example:
This program uses **typeof** to determine what kind of data a pointer points to.
```
local fn DoSomething(@varAddr&, varType)
  print "The data you passed was: ";
  select varType
case  typeof(int)
print peek word(varAddr&)
case typeof(long)
print peek long(varAddr&)
case typeof(Str255)
print pstr$(varAddr&)
case else
print "Unknown"
    end select
end fn
myInt% = 1623
fn DoSomething(myInt%, typeof(myInt%))
myLong& = 426193
fn DoSomething(myLong&, typeof(myLong&))
myString$ = "Hello"
fn DoSomething(myString$, typeof(myString$))
end
```
program output:
```
The data you passed was: 1623
The data you passed was: 426193
The data you passed was: Hello
```

Note:
The integer values returned by **typeof** are determined dynamically at compile time. You should not count on **typeof**(someType) to return the same value every time your program is compiled.

See Also:
sizeof; Appendix C - Data Types and Data Representation

ucase$ ucase$$ **function**

Syntax:
*upperCaseString$* = **ucase$**(*string$*)
upperCaseContainer$$* = **ucase$$**(*container$$)

Description:
This function returns a copy of *string$* or *container$$* with all its lower-case letters converted to upper-case. This also applies to letters with diacritical marks; so for example the string "mágüey" will be converted to "MÁGÜEY".
ucase$ is useful when you want to compare two strings for equality without regard to their letter case. For example, suppose you want all of the following strings to be treated in the same way:
STAZ
Staz
staz
sTAz
Any pair from the above set will be considered to "match" if they're compared as follows:
if ucase$**(str1$) = **ucase$**(str2$) **then print "Names match."
The comparison of containers requires the use of an additional function. See fn FBcompareContainers for more information.

Note:
To test whether one string is greater than or less than another without regard to letter case, an alternative method is to use the string comparison operators ">>" and "<<". See Appendix D - Numeric Expressions, for more information.

See Also:
fn FBcompareContainers; Appendix F - ASCII Character Codes

uns$ **function**

Syntax:

digitString$ = **uns$**(*expr*)

Description:

This function interprets the internal bit pattern of *expr* as an unsigned integer, then returns a string of decimal digits representing that integer's value. The length of the returned string depends on which of defstr byte, defstr word or defstr long is currently in effect; the returned string may be padded on the left with one or more "0" characters, to make a string of the indicated length.

If *expr* is a positive integer, then the number represented in the returned string will be the same as the value of *expr* (provided that the current defstr mode allows **uns$** to return sufficient digits).

If *expr* is a negative integer, then its internal bit pattern is different from that of an unsigned integer. In this case, the number represented in the returned string will be:

- *expr* + 2^8, if defstr byte is in effect;
- *expr* + 2^{16}, if defstr word is in effect;
- *expr* + 2^{32}, if defstr long is in effect.

Note:

To convert a "signed integer" expression *sexpr* into an "unsigned integer" expression which has the same internal bit pattern, just assign *sexpr* to an unsigned integer variable. For example:

myUnsLong&` = mySignedLong&

See Also:

defstr byte/word/long; hex$; oct$; bin$; Appendix C - Data Types and Data Representation

See the do statement.

using   **function**

Syntax:
string$ = **using** *format$; expr*

Description:
This function returns a decimal string representation of the numeric *expr*, formatted according to specifications in *format$*. The characters in *format$* are interpreted as follows:

Specifier	Description
.	This represents a decimal point. In the returned string, the "." is replaced by the currently defined decimal point symbol (which is just a period, if Def Using has never been executed). This specifier also indicates where the integer part of expr should be separated from the fractional part in the returned string. If there is more than one "." character in format$, only the first one is interpreted as a decimal point specifier. If format$ does not contain a "." specifier, then only the integer part of expr will be represented in the returned string. Note that "." is interpreted somewhat differently if the "^^^^" specifier is also used.
#	Each "#" that appears to the left of the "." specifier is replaced by a digit from the integer part of expr, or (if there are excess "#" specifiers) by a blank space. Each "#" that appears to the right of the "." specifier is replaced by a digit from the fractional part of expr, or (if there are excess "#" specifiers) by a "0" character. There must be at least enough "#" specifiers to the left of the "." to represent the integer part of expr. If there are too few "#" specifiers to the right of the "." to represent the fractional part, the value of expr is rounded to the indicated number of digits. Note that "#" is interpreted somewhat differently if the "^^^^" specifier is also used.
*	This is treated the same as the "#" specifier, except that if there are excess "*" specifiers to the left of the ".", the excess specifiers are replaced by "*" characters rather than by blank spaces.
,	Each occurrence of "," which appears after the first "#" or "*" is interpreted as a thousands separator. If there are enough digits in expr to fill at least one of the "#" or "*" specifiers to the left of the ",", then the "," is replaced by the currently defined thousands-separator symbol (which is just a comma, if Def Using has never been executed); otherwise, the "," is replaced by a blank space.
$	If "$" appears to the left of the leftmost "#" or "*" specifier, it's replaced by the currently defined currency symbol (which is just a dollar sign, if Def Using has never been executed). However, if this would cause blank spaces to appear between the currency symbol and the number, then the currency symbol is moved to the right until there are no intervening blank spaces.
+	This is replaced by a "+" if expr is positive, or by a "-" if expr is negative. As with "$", the symbol may be moved to the right to eliminate intervening blank spaces.
-	This is replaced by a blank space if expr is positive, or by a "-" if expr is negative. As with "$", the symbol may be moved to the right to eliminate intervening blank spaces. This specifier is most useful if you want the "-" to appear in a non-standard location; if you use neither the "-" nor the "+" specifier, and expr is negative, and there is a sufficient number of "#" symbols used as placeholders, a "-" will always appear to the left of the number.
^^^^	This specifier causes expr to be represented in scientific notation. The "^^^^" characters will be replaced by an exponent expression, in the form "E+nn" or "Enn". When you use the "^^^^" specifier, the leftmost "#" specifier may not be used to specify digits to the left of the decimal in Format$ because "Standard" scientific notation places one nonzero digit before the decimal point
^^^^^	This is the same as the "^^^^" specifier, but it allows for up to 3 digits in the exponent. You should use this specifier whenever there's a chance that the exponent could exceed ±99.

If *format$* contains any characters other than the specifiers listed above, they are transferred unaltered to the returned string.

Example:
```
x! = 14.726
```
print using `"You owe me $#,###.##."; x!`
program output:
`You owe me $14.73.`

See Also:
str$; uns$

val function

Syntax:

numericValue = **val**(*string$*)

Description:

If *string$* contains the characters of a number in any of the standard FutureBasic formats (decimal, hex, octal or binary), **val** returns the number's value.

val ignores leading spaces in *string$*. When it finds a non-space character, it evaluates the remaining characters in *string$* until it encounters a character which is not part of the number. Thus, for example, the string "3245.6" would be evaluated as 3245.6, but the string "32W45.6" would be evaluated as 32. If the first non-space character in *string$* can't be recognized as part of a number, **val** returns zero. **val** performs the opposite of functions such as str$, hex$, oct$, bin$ and uns$.

Example:

```
data "-3.2", "1.4E2", "&4C1", "9+7"
for i = 1 to 4
  read s$
  print s$, val(s$)
next
```

program output:

```
-3.2 -3.2
1.4E2140
&4C1 1271
9+7 9
```

Note:

If *string$* represents an integer, consider using the val& function, which is faster.

See Also:

val&; mki$; cvi; str$; hex$; oct$; bin$; uns$; Appendix C - Data Types and Data Representation

val&   **function**

Syntax:
integerValue = **val&**(*string$*)

Description:
This function is similar to the `val` function, but it can only evaluate a string which represents an integer.
string$ can represent an integer in decimal, hex, octal or binary format. The absolute value of the represented integer must not exceed 4,294,967,295.
val& ignores leading spaces in *string$*, and it stops evaluating the string when it encounters a character that is not part of standard integer notation. Note that this means **val&** will stop evaluating the string when it encounters a decimal point or an "`E`" exponent indicator. That means that certain strings which represent integers will be evaluated differently by `val` than by **val&**. For example, the string "`24.61E2`" will be evaluated as 2461 by `val`, but as 24 by **val&**.

See Also:
val; mki$; cvi; str$; hex$; oct$; bin$; uns$; Appendix C - Data Types and Data Representation

varptr   **function**

Syntax:
address = **varptr**({ *var* | **fn** `userFunction`})
address = `@`*var*

Description:
varptr`(var)` returns the memory address where the first byte of the variable *var* is located. You can use this value as a "pointer" to *var*. If *var* is a local variable inside a local function, the value returned by **varptr**`( var)` may be different each time you execute the function, and is not valid after the function exits. The syntax `@var` is just a shorthand version of **varptr**`( var)`.
varptr(**fn** `userFunction`) is identical to the **@fn** `userFunction` function.

Note:
Because the "`@`" symbol has a special meaning when it appears after the `print` or `lprint` keyword, you cannot use the `@var` syntax as the first item in a list of print items.
print `@myVar# 'This does not work ("@" is misinterpreted)`
print `(@myVar#) 'This works.`
print **varptr**`(myVar#) 'So does this.`

See Also:
@fn; dim; print; lprint; peek; poke; BlockMove

while **statement**

Syntax:
while *expr*
 [statementBlock]
wend

Description:
The **while** statement marks the beginning of a "while-loop," which must end with a **wend** statement. statementBlock consists of zero or more executable statements, possibly including other while-loops. When a **while** statement is encountered, FutureBasic evaluates *expr*. If *expr* is nonzero, FutureBasic executes the statements in statementBlock; otherwise it jumps down to the first statement following **wend**. If the statementBlock statements are executed, the process is repeated; *expr* is evaluated again, and if it's still nonzero, the statementBlock statements are executed again. This loop continues until *expr* becomes zero, at which point the program exits the loop and jumps down to the first statement following **wend**.
Typically, *expr* is an expression involving logical operators, which is evalutated either as _zTrue (-1) or as _false (0). See the If statement for more information about *expr*.
Note that if *expr* is zero the first time it's evaluated, the statements in statementBlock are not executed at all. do...until is an alternative loop structure that executes statementBlock at least once.

See Also:
for...next; do...until; If

width statement

Syntax:
width [**lprint**][=]{_noTextWrap|_textWrap|numChars}

Description:
This statement affects how (and whether) text printed by subsequent print or lprint statements will "wrap."If you specify the lprint keyword, the **width** statement applies only to statements sent to the printer. If you omit the lprint keyword, the **width** statement applies only to subsequent print statementsdestined for the screen. **width** (without lprint) applies to all existing and subsequently-created windows.

while "wrapping" is enabled, any subsequently printed text whose location exceeds a certain limit on the current line will automatically "wrap around" and continue at the beginning of the next line. Wrapping does not necessarily occur on word boundaries.

If you specify _noTextWrap, wrapping is disabled. Text continues on the current line until the pen is explicitly moved to the next line (this usually happens automatically after the last item in the print or lprint statement has been printed). Note that if the window or the printer page is not wide enough to display all of the items in the print list, some of the items will be lost. The advantage of using _noTextWrap is that it greatly increases printing speed.

If you specify _textWrap, wrapping occurs at the right edge of the window or the printer page. This is the default condition in effect before the first execution of **width**.

If you specify numChars (which must be a number in the range 1 through 255), wrapping occurs either at the right edge of the window (or the printer page), or after numChars characters have been printed on the current line, whichever occurs first. Note that if you're using a proportional font, the horizontal pixel location where wrapping occurs may be different on different lines.

See Also:
lprint; print; route

window close **statement**

Syntax:
window close *windowID*

Description:
This statement closes the window whose ID number is *windowID*, removing it from the screen. It also closes all edit fields, picture fields, buttons and other controls that were in the window. If you re-use the same *windowID* value in a subsequent **window** statement, a new window is created.

If you're closing the active window, and your program has other visible windows open, one of the other windows becomes the active window, and becomes the current output window. If you're closing the current output window (but it's not the active window), you should explicitly designate a new destination for output (using the `window` statement or the `window output` statement) before executing any subsequent text or drawing commands.

See Also:
 window statement; appearance window statement

window **function**

Syntax:
WindowInformation = **window**(*expr*)

Description:
This function returns information related to a window (usually the current output window). The value you specify in *expr* determines what kind of information is returned, as described in the following paragraphs.

ID's of Active Window , Active Document Window, Active Palette Window and Output Window

- **window**(_activeWnd) returns the window ID number of the currently active window, or zero if no window is active.
- **window**(_activeDoc) returns the window ID number of the currently active document window or zero if no document window is active. In searching for the active document, this function bypasses all palettes in search of a window with the type attribute set to include _keepInBack.
- **window**(_activePlt) returns the window ID number of the frontmost palette. In order for there to be a palette, one or more document windows must be open with the type attribute set to include _keepInBack. At that point, all non-_keepInBack windows become palettes and float over the document windows.
- **window**(_outputWnd) returns the window ID number of the current output window, or zero if output is currently directed to somewhere besides a FutureBasic-created screen window (e.g., to the printer).

Window Size

- **window**(_width) returns the width (in pixels) of the content region of the current output window.
- **window**(_height) returns the height (in pixels) of the content region of the current output window. (Note: The content region does not include the window's frame.)

Window Position (Appearance manager)

- **window**(_kFBstructureTop) returns the distance from the top of the screen to the top of the structure region of the window.
- **window**(_kFBstructureLeft) returns the distance from the left of the screen to the left of the structure region of the window.
- **window**(_kFBstructureWidth) returns the width of the window's structure region.
- **window**(_kFBstructureHeight) returns the height of the window's structure region.
- **window**(_kFBcontentTop) returns the distance from the top of the screen to the top of the content region of the window.
- **window**(_kFBcontentLeft) returns the distance from the left of the screen to the left of the content region of the window.
- **window**(_kFBcontentWidth) returns the width of the window's content region. This is normally the same as **window**(_width).
- **window**(_kFBcontentHeight) returns the height of the window's content region. This is normally the same as **window**(_height).

Pen Position

- **window**(_penH) returns the horizontal position (in pixels) of the pen in the current output window.
- **window**(_penV) returns the vertical position (in pixels) of the pen in the current output window.

Window Record Pointer

- **window**(_wndPointer or _wndRef) returns a pointer to the Window Record of the current output window. For information about the contents of the Window Record, see the get window statement.
- **window**(_wndPort) return the current grafport being used for output.

Window Class

- **window**(_outputWClass) returns the "class number" assigned to the current output window.
- **window**(_activeWClass) returns the "class number" assigned to the currently active window.
- **window**(_outputWCategory) returns the "class number" assigned to the current output window for the Appearance Manager runtime.
- **window**(_activeWCategory) returns the "class number" assigned to the currently active window for the Appearance Manager runtime. (See the appearance window statement for more information about class numbers).

Other Window Info (Appearance Manager)

- **window**(`_kFBMacWClass`) returns the toolbox window class. Return values might include things like `_kDocumentWindowClass` or `_kMovableModalWindowClass`
- **window**(`_kFBMacWAttributes`) returns toolbox attributes about a window. Values might include `_kWindowResizableAttribute` or `_kWindowCloseBoxAttribute`
- **window**(`_kFBFloatingWndPtr`) returns the window pointer of the frontmost floating window.

Screen Borders in Local Coordinates

- **window**(`_toLeft`) returns the horizontal pixel position of the screen's left edge, expressed in the local coordinate system of the current output window (note this will be negative if the window lies entirely on the screen).
- **window**(`_toTop`) returns the vertical pixel position of the top of the screen, expressed in the local coordinate system of the current output window (note this will be negative if the window lies entirely on the screen).
- **window**(`_toRight`) returns the horizontal pixel position of the screen's right edge, expressed in the local coordinate system of the current output window.
- **window**(`_toBottom`) returns the vertical pixel position of the bottom of the screen, expressed in the local coordinate system of the current output window.

(Note that these numbers are meaningless if output is currently directed to some place other than a screen window.)

Checking Whether a Window Exists

If you specify a negative value in *expr*, **window**(*expr*) returns a nonzero value if there exists a window whose ID number is `abs(`*expr*`)`; it returns zero otherwise. The returned value does not depend on whether the window is currently visible or not; it only depends on whether the window has been created (using the `window` statement) and not yet closed (using the `window close` statement).

Edit Field and Picture Field Information

- **window**(`_efNum`) returns the ID number of the currently active edit field or picture field; or zero if there is no currently active edit field or picture field.
- **window**(`_selStart`) returns the character position of the beginning of the selected text or insertion point in the currently active edit field (if any).
- **window**(`_selEnd`) returns the character position of the end of the selected text or insertion point in the currently active edit field (if any).
- **window**(`_efClass`) returns the *efClass* parameter assigned to the currently active edit field (if any); or the *negative* of the *just* parameter assigned to the currently active picture field (if any).

Note:

If output is currently directed to a graphics port other than a screen window (e.g. to the printer, or to an offscreen GWorld), then references to the "current output window" apply to the current port, unless otherwise specified.

Some expressions not supported, including:
_textClip
_pictClip
_kFBwDescHandle
_kFBwClickThru
_efHandle
_lastEfNum
_efTextLen
_teBlock

See Also:

window statement; edit field; SetSelect; get window; system function; appearance window

window   **statement**

Syntax:
window *idExpr*`[ ,` `[` *titleString* `][ ,` *rect* `]]]`

Description:
A simplified form of the appearance window statement. Use the **window** statement to do any of the following:

- Create a new screen window, with a limited range of options;
- Activate (highlight and bring to the front) an existing window;
- Make an existing window visible or invisible;
- Alter the title or rectangle of an existing window.

The parameters should be specified as follows. They are interpreted slightly differently depending on whether you are creating a new window or altering an existing one.

- *idExpr* - a positive or negative integer whose absolute value is in the range 1 through 2147483647 (_maxLong).
- *titleString* - a string expression.
- *rect* - a rectangle in global screen coordinates. You can express it in either of two forms:
  `( x1 , y1 ) - ( x2 , y2 )` Two diagonally opposite corner points.
  rectAddr Pointer expression or variable that points to a `Rect` record.

To Create a New Screen Window

- A new document window is created if `abs( idExpr )` differs from the ID number of any existing window. The new window is assigned an ID number of `abs( idExpr )`. You can use the window's ID number later to identify the window in other FutureBasic statements and functions. If *idExpr* is negative, the window is created invisibly. When you create a new window, it becomes the current output window. If you create it visibly it also becomes the current active window.
- *title$* assigns a string to the window's title. If you omit this parameter, the window will be created "untitled".
- *rect* specifies the initial size and location of the window's content rectangle. Note that rect does not include the window's frame. This parameter is interpreted in a special way if you specify an upper-left coordinate of (0,0) in *rect*; in this case, the window is centered in the screen, and its width and height are determined by the right and bottom coordinates of *rect*. Note that this special interpretation applies only when you're creating a new window. If you omit this parameter, a window of default size and location is created.

To Activate an Existing Window

- Specify the (positive) ID number of an existing window in *idExpr*. You do not need to specify any of the other parameters, unless you also wish to to change some of the window's characteristics. The window also becomes the current output window. If the window was invisible, it becomes visible.

To Make an Existing Window Visible or Invisible

- To make a window visible, specify the (positive) ID number of an existing window in *idExpr*. The window also becomes the current active window, and it becomes the current output window.
- To make a window invisible, specify the negative of an existing window's ID number in *idExpr*. The window becomes the current output window. If it was the active window, it becomes inactive (possibly forcing another window to become active). You do not need to specify any of the other parameters, unless you also want to change some of the window's characteristics.

To Alter the Characteristics of an Existing Window

- Specify the ID number of an existing window (or its negative) in *idExpr*, and specify a new *title$* and/or *rect* parameter. If you omit any parameter, the corresponding characteristic won't change. Note that the *rect* parameter is interpreted slightly differently when you're altering an existing window, as opposed to creating a new window; in particular, specifying an upper-left coordinate of (0,0) will move an existing window so that it is partly under the menu bar. If you want to change an existing window's rectangle so that it's centered on the screen, use a *rect* parameter calculated as follows:
  dim as Rect `myRect`

```
        x = (system(_scrnWidth) - myWindowWidth) / 2
        y = (system(_scrnHeight) - myWindowHeight) / 2
        SetRect( myRect, x, y, x + myWindowWidth, y + myWindowHeight )
```

Note: If you specify the window's (positive) ID number when you alter a window's characteristics, the window also becomes the current active window. If you specify the negative of the window's ID number, the window becomes invisible.

Side Effects of Activating the Window

The **window** statement always makes the window active, unless you specify a negative *idExpr*. When you activate a window using the **window** statement, the following things also happen:

- The window also becomes the current output window. (See the `window output` statement to learn how to specify an output window that's different from the active window.)
- A dialog event of type `_wndActivate` is generated. (There are also other kinds of actions which generate `_wndActivate` events; see the `dialog` function for more information.)
- Any previously-active window becomes inactive (this also generates a separate `_wndActivate` dialog event).

Side Effects of Making a Window Invisible

If *idExpr* is negative, the window becomes invisible, and it becomes the current output window. If the window was previously active, it becomes inactive; if your program has other visible windows, one of them becomes the active window.

Customising the Console window

The automatically created default Console window can be customised using an *idExpr* of _FBConsoleWndNum. For example, to maximise the height, use:

include `"ConsoleWindow"`
window `_FBConsoleWndNum, "My Console Window", (5,50)-(500,` **system**`( _scrnHeight ) - 5 )`

See Also:

appearance window; window close; window output; window function; dialog function

window output statement

Syntax:
window output *idExpr*

Description:
This statement makes an existing window the current output window. The current output window is the destination for any subsequent text and drawing commands, and is the target window for such statements and functions as `button`, `edit field`, etc.
The *idExpr* can be either a positive or negative number; **window output** affects the window whose ID number is `abs(`*idExpr*`)`. If you specify a negative *idExpr*, the window becomes invisible. If you specify a positive *idExpr*, the window becomes visible.
window output does not activate the specified window (i.e., it does not highlight its contents nor bring it to the front), if there are other visible windows open. Use the `window` statement (or Appearance Window statement) when you want to explicitly activate a window. (Note: the `window` statement also makes the specified window the current output window.)
window output is useful in cases where you need to update the contents of a window which may be in the background, without bringing the window to the front.

See Also:
window statement; window function; appearance window statement

write dynamic   **statement**

Syntax:
write dynamic *deviceID* , *arrayName*

Description:
write dynamic sends the contents of a dynamic array to an open disk file. Data written to a file in this manner can be read back into memory with read dynamic.

Example:
```
dim as FSSpec fs
dim as long  j
dynamic myAry(_maxLong) as long
fn FSMakeFSSpec( system( _aplVRefNum ), system( _aplParID ), "Test", @fs )
for j = 0 to 9
 myAry(j) = j
next
open "O", 1, @fs
write dynamic 1, myAry
close 1
kill dynamic myAry
open "I", 1, @fs
read dynamic 1, myAry
close 1
for j = 0 to 9
 print myAry(j)
next
fn FSpDelete( fs )
```

See Also:
dynamic; read dynamic; write dynamic

write field   **statement**

Syntax:
write field[#] *deviceID, handle*

Description:
Use this statement to write the contents of the relocatable block specified by *handle* to the open file or serial port specified by *deviceID*, in a format suitable for input by the `read field` statement.

write field starts writing at the current "file mark" position. If first writes a 4-byte long-integer which indicates the size of the block; following this, it writes the contents of the block itself.

Note:
If you want to write only the block's contents to the file (without the 4-byte length indicator), use the `write file` statement instead:

write file# deviceID,[handle],**fn GetHandleSize**(handle)

See Also:
read field; write file; open

write file statement

Syntax:
write file[#] *deviceID, address, numberBytes*

Description:
This statement writes *numberBytes* of data to the open file or serial port specified by *deviceID*, starting at the current "file mark" position. The data is copied from the memory which starts at *address*. This is generally the fastest way to write a large amount of data to a file.

Example:

This program fragment saves the binary image of an array into an open file. You can later use the read file statement to load the array using the data in the file (see the example accompanying the read file statement).

```
_maxSubscript = 200
dim myArray(_maxSubscript) as SInt16
dim as UInt32 arrayBytes

arrayBytes = (_maxSubscript + 1) * SizeOf( SInt16 )
write file #1, @myArray(0), arrayBytes
```

See Also:
open; write; write field; read file

write **statement**

Syntax:
write [#] *deviceID* , { *var* | *stringVar* ; *len* } [, { *var* | *stringVar* ; *len* } . . .]

Description:
This statement writes the contents of the specified variables to the open file or serial port specified by *deviceID*, starting at the current "file mark" position. The variables in the list can be of any type (including record types). If you specify a string variable, it must be followed by a *len* parameter, which indicates the number of bytes to copy from the string. *len* can be any positive numeric expression whose value doesn't exceed the length of the string.

Unlike the print# statement, the **write** statement does not apply any formatting to the data before writing it. Instead, it simply writes an exact copy of the variables' internal bit patterns to the device. This makes the written data suitable for input by the read# statement. In general, the data written by **write** is not suitable for files which are to be interpreted as text.

See Also:
print#; read#; input#; open

xelse   **statement**

See the if or long if statement.

xor **operator**

Syntax:
result = exprA { **xor** | ^^} *exprB*

Description:
Expression *exprA* and expression *exprB* are each interpreted as 32-bit integer quantities. The **xor** operator performs a "bitwise comparison" of each bit in *exprA* with the bit in the corresponding posistion in *exprB*. The *result* is another 32-bit quantity; each bit in the result is determined as follows:

Bit value in *expr*	Bit value in *expr*	Bit value in *result*
0	0	0
1	0	1
0	1	1
1	1	0

A common use for **xor** is to toggle the state of individual bits in a bit pattern. For example:
```
pattern = pattern xor bit(7)
```

This flips bit 7 in `pattern` from 0 to 1 or from 1 to 0, and leaves all of `pattern's` other bits alone.

Example:
The example below shows how bits are manipulated with **xor**:
```
defstr long
print bin$(923)
print bin$(123)
print "--------------------------------"
print bin$(923 xor 123)
```
program output:
```
00000000000000000000001110011011
00000000000000000000000001111011
--------------------------------
00000000000000000000001111100000
```

See Also:
and; or; not

xref **statement**

Syntax:
xref *arrayName*`(`*maxSub1*`[`, *maxSub2* `...])` `[`**as** *dataType*`]`

Description:
The **xref** statement declares an array, and associates the array with the memory pointed to by a particular pointer variable, called the "link variable." You can use **xref** to cause any arbitrary block of memory to be treated as an array. This is especially useful when you need to dynamically create an array whose size can't be determined until runtime, or when you want to impose an array structure on data that was created outside of your FutureBasic program.
The link variable must be a simple (non-array, non-field) pointer variable which has the same name as the array (ignoring any type-identifier suffix). For example, if you specify the following:
xref `diameter(3,7) as long`
The compiler creates a long-integer variable called `diameter`. When you run the program, you should set `diameter` equal to some appropriate memory address (you do this after the **xref** statement); FutureBasic then assumes that the `diameter()` array begins at that address. When you examine elements in the array, they are retrieved from the memory pointed to by `diameter`. When you alter elements in the array, the memory pointed to by `diameter` is altered.

The first subscript is arbitrary
The *maxSub1*, *maxSub2* etc. values must be positive static integer expressions. However, since **xref** does not actually allocate any memory, the declared subscripts are used somewhat differently than in a `dim` statement. The second and subsequent subscripts (if any) determine the internal structure of the array, and they should exactly match the internal layout of the elements pointed to by the link variable. But the value of the first subscript (*maxSub1*) is basically ignored, and may be arbitrarily set to any value greater than zero. When you actually reference the array elements, you can use subscript values that are larger than *maxSub1*, as long as they reference valid elements within the block of memory pointed to by the link variable. However, whilst the first subscript (*maxSub1*) is generally ignored, the value of *maxSub1* gets used for bounds testing when the Preferences setting for "Check Array Bounds" is enabled.

Note:
xref is a non-executable statement, so you can't change its effect by putting it inside a conditional-execution structure such as `long if...end if`. However, you can conditionally include it or exclude it from the program by putting it inside a `#if...#endif` block. The **xref** statement should appear somewhere above the first line where the array is referenced.

See Also:
dim; xref@

xref@   **statement**

Syntax:
xref@ *arrayName*(*maxSub1*[*, maxSub2* ...]) [**as** *dataType]*

Description:
xref@ is identical to the xref statement, except that the link variable is interpreted as a handle, rather than as a pointer. You should use **xref@** when you want the contents of a relocatable block to be treated as an array.

Example:
The following declares an array called inclination, allocates a new block with enough room for numElements elements, and associates the inclination array with the contents of the block.

```
dim as long numElements  : numElements = 5  // 5 is arbitrary
xref@ inclination(1)
inclination = fn NewHandle( numElements * sizeof( long ) )
```

Note that, because the value of maxSub1 is ignored in the **xref@** statement, we can arbitrarily set it to 1. However, when we actually reference the elements of the inclination array, we can specify any subscript value in the range 0 through numElements - 1.

See Also:
dim; xref

A file object (file, folder, or volume) may be specified in one of four ways: FSRef, CFURLRef, FSSpec or POSIX path.

The preferred way of accessing file objects external to an application's bundle is to use FSRefs. CFURLRefs are favored for locating files and resources within an application bundle. FSSpecs are deprecated and should be avoided for general use in favor of the modern FSRefs. (An exception to the rule are a few QuickTime functions which still accept only FSSpecs.) POSIX paths are used in programs following UNIX conventions and are outside this discussion.

Working directories (obsolete)

Under MFS on the first Macs, files were identified by two parameters: name and volume reference number. When HFS superseded MFS, the directory structure required an additional parameter: the parID. The official way to identify a file then became the FSSpec, which contains all three parameters. To allow MFS code to work under the new file system, Apple devised a hack known as a working directory. An unfortunate consequence was that many programs, even newly written ones, continued to use the old MFS API instead of switching to the new-in-1985 FSSpecs. See Apple Documentation for more information.

If your code designed for versions of FutureBasic prior to version 4 attempts to identify a file by name and one number, that number is a working directory reference number.

Working directories were abandoned by Apple in Carbon. Working directory reference numbers, and related functions designed for versions of FutureBasic prior to version 4 such as system(_aplVol), system(_sysVol) and FOLDER, are not implemented, and will never be implemented, in FBtoC.

FSRef

Description

The MacOS X File Manager provides an abstraction layer that hides lower-level implementation details such as different file systems and volume formats. A key component of that abstraction layer is the FSRef.

An FSRef is an opaque reference stored in a record assigned by the File Manger to describe a file or folder.

The contents of an FSRefs are dynamic in nature. For instance, if your code utilizes an FSRef to reference a file or folder, when the Macintosh running your code is restarted, that FSRef structure is cleared. On restart, when your code creates an FSRef to the same file or folder previously referenced, the File Manager will create a new and unique FSRef to identify that file or folder, the content of which will differ from the former.

The declaration of FSRef in Files.h is:

struct FSRef {
 UInt8   hidden[80]; /* private to File Manager*/
};

Creating an FSRef with the files$ Function

FutureBasic (version 5 onwards) offers native creation of FSRefs with files$():

```
dim as FSRef    fref
dim as Str255   fStr

fStr = files$( _FSRefOpen, "TEXT", "Open text file", fref )
long if ( fStr[0] )
 // Do something with your text file FSRef
xelse
```

```
    // User canceled
end if
```

Additional examples of working with FSRefs can be found FutureBasic 5 Examples > Files.

FSRefs are also supported by the FutureBasic (version 5 onwards) NavDialog() and NavDialog_Xxxx() functions. See "NavDialog_Demos" in FutureBasic Examples > Files.)

CFURLRef

Description

CFURL provides facilities for creating, parsing, and dereferencing URL strings. The name is an abbreviation for "Core Foundation URL." Core Foundation constitutes the underpinnings of Apple's Carbon language on which FutureBasic is based.

Anyone familar with the internet is familiar with URLs (Uniform Resource Locators.) To understand how CFURLs work, let's examine a typical example of a URL used in a web browser:

   http://www.apple.com

The first part of that URL:

   http:

describes the file protocol (or scheme) that will be used to handle the ensuing address. HTTP is a abbreviation for Hypertext Transfer Protocol. Most modern web browsers will automatically recognize http, and will accept the shorthand: www.apple.com. Examples of other protocols include: ftp (File Transfer Protocol), https (Secure HTTP), mailto (Mail Protocol for sending email), gopher, wais - and there is even one for our own computer: file.

The second part of the URL:

   //www

points to the URL's host, in this case the "worldwide web."

The remainder of the URL:

   apple.com

identifies the address on the worldwide web we are looking for, the Apple Computer web site.

(It should be noted that protocols, hosts and addresses are case insensitive. Thus http://www.apple.com is synonymous with HTTP://WWW.APPLE.COM. And we would be remiss not to point out that the address, "www.apple.com" is actually an English language representation-formally known as a "domain name"-of a numerical IP (Internet Protocol) address. As of this writing, www.apple.com represents the IP address: 17.149.160.49. If you enter those numbers into your web browser, it will take you to the Apple website. The advantage of a domain name is that it can be modified to point to any IP address. Thus if in the future Apple changes it's internet provider and gets a new IP address, the www.apple.com domain name will be reassigned to point to the new IP address and users will never notice any change. Under the current addressing convention there are about 4.29 billion IP addresses, which means in the future they could well become exhausted. But that is beyond the scope of this discussion.)

The convenience of the URL structure is that it can pinpoint the address of any single file on any computer designed to implement its convention. This is done by appending the pathname of the file to be fetched to the URL.

But to find a file on our own computer, we first must know its host name, and that is universal for all MacOS X systems:

   localhost

Open your web browser and enter the following URL.

   file://localhost/Volumes/

Notice that this gives you an overview of the root level of your computer, in this case any hard drives installed on your computer. You may also notice that your browser converted the URL of the root level of your computer to:

file:///Volumes/

The third forward slash is simply an abbreviation for "localhost".

You can further define a URL on your personal computer by adding additional path elements. For instance, to examine the contents of the Applications folder on your hard drive, in your web browser enter this line substituting the name of your hard drive:

file:///Volumes/MyHardDrive/Applications/

WARNING: If this did not work, the name of your hard drive may contain a space or spaces it in, for instance "My Hard Drive." In that case you have to enter the URL coded in a way your browser will recognize this. We do this by "escaping" the space characters with the backlash character "\" like this:

file:///Volumes/My\ Hard\ Drive/Applications/

This demonstrates an important point: We need to understand how URLs are interpreted to work comfortably with CFURLs.

HINT: If you want to see the properly formed URL for any drive, folder or file on your computer, simply drag it onto a web browser's window. You can also open your Terminal and drag it onto the Terminal window to see the escaped path.

When FutureBasic creates an application, it actually creates a folder with several files and folders inside it. That folder is appended with the extension ".app" which your Macintosh operating system recognizes as a bundled application and treats that special folder as a single clickable file. However, as programmers we often need to refer to files inside that application folder. And this is one area where CFURLs excel.

Apple has provided an "opaque" object describer for a CFURL called a CFURLRef. (The internal composition of opaque types is not documented by Apple. This allows them the freedom of enhancing the internal construction of the describer without disturbing the way your code functions. This is not unlike the way the aforementioned domain name, which does not change, points an IP address that may vary.)

CRURLRefs point to a structure that contains all sorts of identifying characteristics of any given file. Apple has provided a host of Toolbox functions that we can use to extract the information we need from a CRURLRef. In addition, a CFURLRef is "toll-free bridged" with its Cocoa Foundation counterpart, NSURL. For FutureBasic programmers this is good news because of the many code snippets available online in Cocoa that are transplantable into our Carbon code.

In simple terms, rather than having to refer to our Application folder as:

file:///Volumes/My\ Hard\ Drive/Applications/

we can define it as a CFURLRef and use that variable with the bonus that the CFURLRef also holds a wealth of information about the folder other than just its path.

A CFURL object is composed of two components:

1. A base URL, which can be empty, 0 or NULL in C, and
2. A string that is resolved relative to the base URL.

A CFURL object whose string is fully resolved without a base URL is considered absolute; all others are considered relative.

As an example, assuming that your FutureBasic application is stored in your MacOS X Applications folder in a folder labeled "FutureBasic 5.4," an absolute CFURL path to its executable would be:

/Applications/FutureBasic/FutureBasic.app/Contents/MacOS/FutureBasic

This object is fully resolved since it is a complete path to the root level of the volume on which it is located.

On the other hand, looking inside an application bundle a relative path not fully resolved would be:

/FutureBasic.app/Contents/MacOS/FutureBasic

This relative path defines the location of the FutureBasic executable within the FutureBasic.app bundle, but does not indicate where the FutureBasic application resides.

The following example demonstrates a technique for obtaining a CFURLRef to a text file named "ReadMe.txt" located inside an application bundle, and passing the CFURL to LSOpenCFURLRef() which opens the file in the system's default text application.

```
include "Tlbx LSOpen.incl"
include "Tlbx CFBundle.incl"
include resources "ReadMe.txt" // file to be copied to <app>/Contents/Resources

local mode
local fn OpenReadMe( name as Str255 )
dim as CFBundleRef  bundle
dim as CFURLRef      url
dim as CFStringRef   cfName

bundle = fn CFBundleGetMainBundle()
long if ( bundle )
  cfName = fn CFStringCreateWithPascalString( 0, name, _kCFStringEncodingMacRoman )
  url = fn CFBundleCopyResourceURL( bundle, cfName, 0, 0 )
  CFRelease( cfName )
  long if ( url )
    fn LSOpenCFURLRef( url, #0 )
    CFRelease( url )
  end if
end if
end fn

fn OpenReadMe( "ReadMe.txt" )
do
 HandleEvents
until gFBQuit
```

Although FSRefs may be favored for locating files external to application bundles, some programmers prefer to use CFURLRefs instead. Many of Apple's Carbon code examples rely on them heavily. And it is not at all uncommon to see them inside Cocoa code examples. It would behoove an FutureBasic programmer to become familiar with both FSRefs and CFURLRefs and then choose the tool most appropriate for his program.

Like FSRefs, CFURLRefs are also supported by the FutureBasic (version 5 onwards) NavDialog() and NavDialog_Xxxx() functions. See "NavDialog_Demos" in FutureBasic 5 Examples > Files.)

FSSpec

Description:

A file spec record is defined in the Headers:

```
begin record FSSpec
 dim as short vRefNum
 dim as long parID
 dim as Str63 name
end record
```

Most of the relevant API, including the essential FSMakeFSSpec() has been deprecated by Apple since MacOS X 10.4 and could be removed in any future MacOS X release. FSSpecs still have some appeal because of their ease of use but have limitations and undesirable side effects that make them inappropriate for general public release:

[1] File names > 32 characters cannot be created

[2] File names using Unicode characters cannot be created

[3] Files can be opened and read whose name is problematic as in #1 or #2, but theSpec.name is not the original one. Apple deliberately encodes the nodeID in the altered name (for example "VeryVeryVeryLongLongLong#5DA4C0" or "???#5DF338.txt"). Saving such files works OK as long as the .name field is left untouched and the original file has not been moved or renamed. Attempts to save 'companion' files (by modifying .name) tend to give files on disk with bizarre names.

[4] There are mysterious bugs when name contains certain 'high-bit' MacRoman characters such as pi (option-p). Such a file can be opened, but theSpec.name contains garbled characters. Saving the file, even with no change to theSpec.name, gives a disk file whose name is garbled.

File spec records may be created as follows:

```
dim as FSSpec  fs
```

When the FSSpec is used as a parameter in files$(), or in the 'open' statement, the information is passed to file handling calls as a single record, but the individual fields may be extracted from the record as follows:

```
dim fs as FSSpec
fileName$ = fs.name
parentID = fs.parID
```

```
volRefNum = fs.vRefNum
```

Note of Caution: see undesirable side effects #3 and #4 above before using the name field.

Conversion between File Object Specifiers

The "Util_FileDirectory.incl" Headers file contains a suite of functions to assist a programmer working with files and folders.

For instance, at times a programmer has a need to convert between the formats. "Util_FileDirectory.incl" offers functions to expedite such conversions:

fn FD_FSRefCreateCFURL(FSRef *ref, CFURLRef *outUrl)
fn FD_FSRefGetFSSpec(FSRef *ref, FSSpec *outSpec)
fn FD_CFURLGetFSRef(CFURLRef url, FSRef *outRef)
fn FD_CFURLGetFSSpec(CFURLRef url, FSSpec *outSpec)
fn FD_FSSpecGetFSRef(FSSpec *spec, FSRef *outRef)
fn FD_FSSpecCreateCFURL(FSSpec *spec, CFURLRef *outUrl)

Some Carbon Toolbox functions for converting POSIX paths include:

CFURLCopyFileSystemPath(): Convert a CFURLRef to POSIX Path
CFURLCreateWithFileSystemPath(): Convert a POSIX path to CFURLRef

In FutureBasic, a variable can be thought of as a named container for data. The "container" has a specific size and (usually) a specific address in memory. Also, each variable has a specific type which determines how FutureBasic interprets its contents (See Appendix C - Data Types and Data Representation). You can copy data into a variable by putting the variable on the left side of the "=" symbol in a `let` statement; or by explicitly modifying the contents at the variable's address (through statements like `poke` and `BlockMove`). Certain other FutureBasic statements and functions (such as `swap` and `inc`) may also modify a variable when you include the variable as a parameter. In FutureBasic, a variable can have any of the following forms:

- `identifier[tiSuffix]`
  A simple string or numeric variable, such as: `myLong&`, or `theString$`. tiSuffix is the optional type-identifier suffix, such as "`$`", "`%`", "`&`", etc. See the `dim` statement, and Appendix C - Data Types and Data Representation, for a complete list of type-identifier suffixes. Examples:
  ```
  myIntVar
  xyz&`
  ```

- `stringVar$[offset]` (Note: the square brackets are part of the variable)
  This variable consists of the single byte which is located at offset bytes past the beginning of the string variable `stringVar$`. (The $ is required) This variable's type is `unsigned byte`. This kind of variable is normally used to quickly retrieve or alter a single character in a string. The statement, "`x=stringVar$[offset]`" is equivalent to: "`x=peek(@stringVar$+offset)`". The statement, "`stringVar$[offset]=x`" is equivalent to: "`poke@stringVar$+offset,x`". Examples:
  ```
  firstname$[3]
  ```

- `pointerVar`
  A pointer variable. This is an identifier declared as a `pointer` type; it can be declared either as a "generic" pointer, or a pointer to some other specific type. Examples:
  ```
  myPtr
  anotherPtr
  ```

- `handleVar`
  A handle variable. This is an identifier declared as a `Handle` type; it can be declared either as a "generic" handle, or a handle to some other specific type. Examples:
  ```
  myHandle
  thisHdl
  ```

- `recordName`
  The variable is an entire record declared using `dim recordName as RecordType`). Examples:
  ```
  myRec
  iopb
  ```

- `arrayName[tiSuffix](expr1 [,expr2...])`
  The variable is a specific element of an array. This can be an array of any type, but `tiSuffix` can only be used in numeric or string arrays. Note that an entire array is not considered to be a variable. Examples:
  ```
  firstName$(15)
  recArray(3, x%)
  ```

- `addressVar&.const1[.const2...]tiSuffix`
  The variable consists of the bytes located at a specific offset from the address given in `addressVar&`. The address of this variable is at (`const1` + `const2` + ...) bytes past the given address; the size and type of this variable are determined by `tiSuffix`. `addressVar&` must be a (signed or unsigned) long-integer variable, or a generic `pointer` variable. `addressVar&` must be a "simple" variable; it cannot be an array element nor a record field. Examples:
  ```
  recPtr&.myField%`
  genericPtr.rectangle.right%
  ```

- `handleVar&..const1[.const2...]tiSuffix`
  The variable consists of the bytes located at a specific offset from the beginning of the relocatable block referenced by `handleVar&`. The address of this variable is at (`const1` + `const2` + ...) bytes past the beginning of the block. The size and type of this variable are determined by `tiSuffix`. `handleVar&` must be a (signed or unsigned) long-integer variable, or a generic `Handle` variable. `handleVar&` must be a "simple" variable; it cannot be an array element nor a record field. Examples:
  ```
  recHdl&..thisField.thatField$
  genericHandle..someField`
  ```

Variables involving fields of "records"
The fields of a "record" are defined inside a `begin record`...`end record` block. A field's declared data type can be any valid type; if a

field is itself declared as another "record" type, then the field can have "subfields," which are just the fields of that secondary record.
A field can also be declared as an array (of any type). In this case, whenever the field's name is included as part of a variable specification, it must be followed by subscript(s) in parentheses. Thus, in each of the variable descriptions listed below, each `field` and `subfield` takes one of the following forms, depending on whether or not it's an array field:

For non-array fields:
`field/subfield ::= fieldName[tiSuffix]`
For array fields:
`field/subfield ::= fieldName[tiSuffix](sub1 [,sub2...])`
The type and size of each of the following variables is just the type and size of the last `field` or `subfield` specified.

- `recordName.field[.subfield ...]`
  The variable is the specified field or subfield of the specified "record." Examples:
  `myRec.myField%`
  `stats.game(7).teamName$(1)`

- `recordPtr.field[.subfield ...]`
  The variable is the specified field or subfield of the "record" pointed to by `recordPtr`. The `recordPtr` must be declared as a pointer to a specific type of record. Examples:
  `ptr1.myField`
  `ptr2.arrayField$(3)`

- `recordHdl..field[.subfield ...]`
  The variable is the specified field or subfield of the "record" referenced by recordHdl. The recordHdl must be declared as a handle to a specific type of record. Examples:
  `Hdl1..book(3).title$`
  `Hdl2..phoneNum`

- `arrayName(expr1[,expr2 ...]).field[.subfield ...]`
  This variable is the specified field or subfield of a specific element in an array of "records." Examples:
  `HouseArray(42,6).streetName$`
  `season(2).game(3).player(6)`

- `ptrArray(expr1[,expr2 ...]).field[.subfield ...]`
  This variable is the specified field or subfield in a "record" pointed to by an element in an array of pointers. The array must be declared as an array of pointers to a specific type of record. Examples:
  `myPtrArray(n).field3&`
  `ptrArray(6,2).miscInfo.chapter(7).title$`

- `handleArray(expr1[,expr2 ...])..field[.subfield ...]`
  This variable is the specified field or subfield in a "record" referenced by an element in an array of handles. The array must be declared as an array of handles to a specific type of record. Examples:
  `myHndlArray(7,j)..map`
  `myHndlArray(7,j)..map.quadrant(x,3).icon&`

Limitations

Arrays are limited to about 2 gigabytes (each).

The `.MAIN` file of a project often allocates variables outside of local functions that are not global. These are treated as variables for a local function.

I. Integers

Integers can be represented as literals, as symbolic constants, or as variables.

I.1 Integer Literals

- *Decimal:* a string of decimal digits, optionally preceded by "+" or "-".
  Examples: `7244 -328442`
- *Hexadecimal:* a string of up to 8 hexadecimal digits, preceded by "`&`" or "`&H`" or "`0x`" (that's a zero-x). Hexadecimal digits include the digits 0 through 9, and the letters `A` through `F`. Letters can be either in upper or lower case.
  Examples: `&H12a7 0x47BeeF &42AD9`
- *Octal:* a string of up to 10 octal digits, preceded by "`&O`" (that's the letter "`O`", not a zero). Octal digits include the digits 0 through 7.
  Examples: `&o70651 &o32277`
- *Binary:* a string of up to 32 binary digits, preceded by "`&x`". Binary digits include the digits 0and1.
  Examples: `&x0100011 &x10110000111011001`
- *Quoted:* a string of up to 4 characters, surrounded by double-quotes, with an underscore preceding the initial quote. Each character in the quoted string represents 8 bits in the internal bit pattern of the resulting integer, according to the character's ASCII code. Examples: `_"text" _"N*"`

Note: Hexadecimal, octal, binary and quoted literals reflect the actual bit patterns of the integers as they're stored in memory. These may be interpreted either as positive or negative quantities, depending on which types of variables they're assigned to. If they're not assigned to any variable, they're generally interpreted as positive quantities.

I.2 Symbolic Constants

A symbolic constant is an identifier preceded by an underscore character. There are many symbolic constants which have pre-defined values in FutureBasic. You can also define your own symbolic constants within your program, either by using a `begin enum`...`end enum` block; or a `dim record`...`end record` block; or by using a "constant declaration" statement. A constant declaration statement has this syntax:

`_constantName = staticExpression`

where `_constantName` is a symbolic constant which has not been previously defined, and `staticExpression` is a "static integer expression" (see [Appendix D - Numeric Expressions](#)). The value of `staticExpression` must be within the range -2,147,483,648 through +2,147,483,647. Once a symbolic constant has a value assigned to it, that value cannot be changed within your program. Like all constants, a symbolic constant has a global scope.

A constant declaration may also include pascal style strings using one of the following formats

```
_constantName$ = "I am a string constant"
_constantTab$ = 9 : rem chr$(9) = tab character
_constantCR$ = 13 : rem chr$(13) = carriage return
_twoByteKanjiChar = 10231: rem KCHR$(10231)
```

I.3 Integer Variables

There are six different types of integer variables in FutureBasic; they differ in the amount of storage space they occupy, and in the range of values they can represent. An integer variable's name may end with a type-identifier suffix which indicates its type; alternatively, you can declare an integer variable's type by using the `as` clause in a `dim` statement. If a variable has no type-identifier suffix, and wasn't declared with an `as` clause, then FutureBasic checks whether there are any `def<type>` statements which are applicable to the variable. Finally, if the variable can't be typed by any of the above means, FutureBasic assigns the type "signed short integer" to the variable. Arrays of integers, and integer record fields, are typed by similar means.

Type	Storage	Range	Type identification
signed byte	1 byte	-128..+127	x' Dim x As Byte Dim x As Char
unsigned byte	1 byte	0..255	x`` Dim x As Unsigned Byte Dim x As Unsigned Char
signed short integer	2 bytes	-32768..+32767	x% Dim x As Int

			Dim x As Word Dim x As Short
unsigned short integer	2 bytes	0..65535	x%` Dim x As Unsigned Int Dim x As Unsigned Word Dim x As Unsigned Short
long integer	4 bytes	-2147483648..+2147483647	x& Dim x As Long
Unsigned long integer	4 bytes	0..4294967295	x&` Dim x As Unsigned Long

II. Real Numbers
"Real numbers" are numbers which may have a fractional part. They can be represented as literals or as variables.

II.1 Real Number literals

- *Standard notation:* a string of decimal digits including a decimal point; optionally preceded by "+" or "-".
  Examples: `17.3 -62. 0.03`
- *Scientific notation:* a string of characters in this format:
  mantissa `{E|e}` *exponent*

mantissa is a string of decimal digits with an optional decimal point, optionally preceded by "+" or "-"; *exponent* is a string of decimal digits, optionally preceded by "+" or "-".
Examples: `3e-20 -6.7E4 0.05E+14`
The value of a number expressed in scientific notation is:
mantissa $10^{exponent}$

II.2 Real Number variables
There are three types of real number variables in FutureBasic; they differ in the amount of storage space they occupy, the range of values they can represent, and their precision (number of significant digits).

II.2.1 Fixed-point Reals
A fixed-point real number variable must be declared in a `dim` statement, using the `as Fixed` clause. It's accurate to about 5 places past the decimal point, and can handle numbers in the range of approximately -32767.99998 through +32767.99998. A fixed-point variable occupies 4 bytes of storage.

II.2.2 Floating-point Reals
FutureBasic supports two kinds of floating-point real number variables. A floating-point variable's name may end with a type-identifier suffix which indicates its type; alternatively, you can declare a floating-point variable's type by using the `as` clause in a `dim` statement. If a variable has no type-identifier suffix, and wasn't declared with an `as` clause, FutureBasic checks whether there are any `defsng <letterRange>` or `defdbl <letterRange>` statements which are applicable to the variable. Floating-point arrays, and floating-point record fields, are typed by similar means.
The methods used by FutureBasic when handling one of these variables can be modified by you. A set of constants is maintained in a file in the headers folder. (Path: FutureBasic Extensions/Compiler/Headers/UserFloatPrefs). If you want to change these parameters for all of your projects, copy the file named "UserFloatPrefs" into the User Libraries folder. The User Libraries folder is located at the same level as the editor.

```
//
// Required Floating Point Constants //
//
_NumberLeadingSpace = _True //FBII Default = _true
_RoundUpFloat2Long = _true // Un-remark to round up Float to Integer
```

Generally speaking, double-precision floating-point variables occupy more storage, represent a greater range of values, and have greater precision than single-precision floating-point variables. However, the exact storage space, ranges and precisions of these types depend on the target CPU of the compiled program (Note: the storage space for variables can vary between CPU devices. When in doubt, use the `sizeof` function to make a definite determination of the size of the variable.)

Type	Type Identification
single-precision	`x!` (4 bytes) `dim x as single`
double-precision	`x#` (8 bytes) `dim x as double`

III. Strings
A string is a list of up to 255 characters, which is usually interpreted as text. Strings can be represented as literals or as variables.

III.1 String Literals

A string literal is a group of characters surrounded by a pair of double-quotation marks (note: in certain contexts, such as in `data` statements, the quotation marks may be optional). If the string literal contains a pair of contiguous double-quotes, they are interpreted as a (single) embedded double-quote mark and treated as part of the string, rather than as a delimiter. Example:

print `"I said, ""Hello."""`

program output:

`I said, "Hello."`

III.2 String Variables

You can specify a string variable by appending the type-identifier suffix "`$`" to the variable's name; alternatively, you can declare a variable as a string by using the `as Str255` clause in a `dim` statement. If a variable has no type-identifier suffix, and wasn't declared with an `as` clause, then FutureBasic checks whether there are any `defstr <letterRange>` statements which are applicable to the variable. String arrays, and string record fields, are typed by similar means.

A string variable declared `as Str255` can hold up to 255 characters. The maximum number of characters that other string variables can represent is determined by the `maxLen` value specified in a `dim` statement, or by the value specified in the `DEFLEN` statement. If neither of these values was specified, then the string variable can hold a string of up to 255 characters.

Internally, strings are stored in "Pascal format." Pascal format begins with a "length byte" which is interpreted as a number in the range 0 through 255. The length byte's value indicates the number of characters currently in the string. The length byte is followed immediately by the string's characters, one byte per character. FutreBasic[3] always allocates an even number of bytes for a string variable in memory; this is enough to include the length byte, plus enough character bytes for the variable's maximum string length, plus an exta "pad" byte (if necessary) to make the total come out even. Use the `sizeof` function to determine the number of bytes allocated to a particular string variable.

IV Containers

Containers are compiler managed handles that hold up to 2 gigabytes of ASCII or numeric information. Containers may be identified by a double dollar sign (`dim myContainer$$`) or in a `dim as` statement (`dim as CONTAINER myContainer`).

Containers are always global. An attempt to dimension a container inside of a local function will result in an error message during compilation. When a container is first dimensioned, it is a long integer variable with a value of zero. Once data is placed in the container, a handle is allocated and the data is moved to that handle. To dispose of the allocated handle, set the container to a null string with `myContainer$$ = ""`.

Because a container may hold ASCII or numeric information, there are some trade-offs. The first is speed. Numeric values stored in containers are first converted to ASCII. When math operations are required, the data is reconverted before the calculation is performed. Another limitation relates to how containers are filled. Since FutureBasic has no idea what data may be in the container, it has to evaluate the information on the other side of the equal sign to see what it should be doing. If this data is a series of Pascal strings, then the container must be limited to 255 characters.

`myContainer$$ = a$ + b$ + c$`

If the information is to be a concatenated string and the right side of the equal sign contains only Pascal style strings, you must approach things from a different direction.

```
myContainer$$ = a$
myContainer$$+= b$
myContainer$$+= c$
```

In some cases, the compiler will not be able to determine what type of operation you had in mind. For instance...

`a$$ = b$$ + c$$`

The compiler has no clue as to whether it should concatenate strings or add numbers. You can force the correct operation by inserting an additional operator.

`a$$ = b$$ + c$$ + 0 :` **rem** `math`

`a$$ = b$$ + c$$ + "" :` **rem** `strings`

This is not a problem with other math operators like the minus sign or the multiplication (asterisk) symbol as these cannot pertain to strings. Containers may not be compared in the traditional sense. This is because a comparison by its very nature must return a numeric value. If you execute a statement like `print a$ = b$` the result will be zero (_false) or -1 (_zTrue). We have provided a substitute function that can handle the comparison for you.

`rslt& =` **fn FBcompareContainers**`(a$$,b$$)`

If a$$ is less than b$$ then the result will be negative and will represent the character position at which the difference was found. If rslt& is -3000 then a$$ and b$$ were identical for the first 2999 characters, at which time the next character in b$$ was found to be less than the one in a$$.

When `rslt&` is zero, the containers are equal.

When `rslt&` is positive, it points to the character position at which it was determined that a$$ is greater than b$$.

Containers are stored in the application heap as relocatable blocks. You can extract the handle to these blocks as follows:

`hndl& = [@myContainer$$]`

Be aware that the handle may be zero if the container has been cleared or if it was never initialized.

A syntax similar to that used for filling edit fields may be used to pass information to a container. The percent sign (%) indicates that the container is to be filled with the contents of a _"text" resource. An ampersand (&) tells FutureBasic to fill the container with information from a handle.

`a$$ = %resID% :` **rem** `fill container with text res ID resID%`

`a$$ = &hndl& :` **rem** `fill container with contents of hndl&`

Note:

You may not use complex expressions that include containers and/or Pascal strings on the right side of the equal sign. Instead of using:

```
c$$ = c$$ + left$$(a$$,10)
d$$ = c$$ + a$
```
Use:
```
c$$ += left$$(a$$,10)
d$$ = c$$
d$$ += a$
```

V. Pointers

A pointer variable is always declared in a `dim` statement. It can be declared using the `as pointer` (or `as PTR`) clause, or an `as ptrType` clause, where ptrType is a type which was previously identified as a `pointer` type (in a `#define` statement). If the `as pointer` clause included a `to` clause, then the variable is identified as "pointing to" a data structure of the indicated type; otherwise it's considered a "generic" pointer.

The value of a pointer is actually a long integer; it's the address of a data structure. In some cases a pointer's value may be `_nil` (zero), which indicates that the pointer currently isn't "pointing to" anything.

If you declare a pointer variable as pointing to a particular record type, you can use the pointer variable to refer to specific fields within a record (see Appendix B - Variables, for more information).

VI. Handles

A handle variable is always declared in a `dim` statement. It can be declared using the `as Handle` (or `as HNDL`) clause, or an `as hdlType` clause, where `hdlType` is a type which was previously identified as a `Handle` type (in a `#define` statement). If the `as Handle` clause included a `to` clause, then the variable is identified as a handle to a data structure of the indicated type; there are also a couple of pre-defined types (`RgnHandle` and `TEHANDLE`) which are recognized as handles to particular types of MacOS structures (specifically: to regions and TextEdit records). If the variable is declared simply "`as Handle`" (with no `to` clause), it's considered a "generic" handle.

The value of a handle is actually a long integer; it's the address of a "master pointer" which points to a relocatable block that contains a data structure. In some cases a handle's value may be `_nil` (zero), which indicates that it doesn't currently refer to any data structure.

If you declare a handle variable as referring to a particular record type, you can use the handle variable to refer to specific fields within a record (see Appendix B - Variables, for more information).

VII. Records

A record is a (usually small) collection of data items that are stored together in memory. You can access an entire record as a unit, or access its data elements individually. Unlike an array, whose elements are all of the same type, the elements of a record (also called its "fields") can be of differing types. A record variable must be declared in a dim statement, using the following syntax:

dim `recordName` **as** `recordType`

where recordType is previously-defined record type. You can define a record type and its fields by using a `begin record`...`end record` block. In addition, FutureBasic recognizes two built-in record types: `Rect` and `Point`. You use the `recordName.field` syntax to access the fields of a record variable (see Appendix B - Variables).

Compatibility of Types

You can assign values of one type to variables of another type, sometimes with certain restrictions. The following table shows which kinds of values can be assigned to which kinds of variables.

Values Variables	Sign. Byte	Uns. Byte	Sign. Byte	Uns. Byte	Sign. Byte	Uns. Byte	Fixed	Simple	Double	String	Pointer	Handle	Record
Sign. Byte	OK	2	2	2	2	2	2,3	2,3	2,3	2,3,8	NO	NO	NO
Uns. Byte	1	OK	1,2	2	1,2	2	1,2,3	1,2,3	1,2,3	1,2,3,8	NO	NO	NO
Sign. Word	OK	OK	OK	2	2	2	3	2,3	2,3	2,3,8	NO	NO	NO
Uns. Word	1	OK	1	NO	1,2	2	1,3	1,2,3	1,2,3	1,2,3,8	NO	NO	NO
Sign. Long	OK	OK	OK	OK	OK	2	3	2,3	2,3	2,3,8	10	10	NO
Uns. Long	1	OK	1	OK	1,2	OK	1,3	1,2,3	1,2,3	1,2,3,8	10	10	NO
Fixed	OK	OK	OK	2	2	2	OK	2,4	2,4	2,4,8	NO	NO	NO
Simple	OK	OK	OK	OK	4	4	4	OK	4	4,8	NO	NO	NO
Double	OK	OK	OK	OK	OK	OK	OK	OK	OK	8	NO	NO	NO
String	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5,8	5	5,8	5,8	NO
Pointer	OK	OK	OK	OK	OK	OK	2,3	2,3	2,3	2,3,8	6	NO	NO
Handle	OK	OK	OK	OK	OK	OK	2,3	2,3	2,3	2,3,8	OK	7	NO
Record	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	9

Notes:

1. Assigning a negative value to an unsigned integer type may produce unexpected results.
2. Assigning a number outside of a type's range may produce unexpected results.
3. Result will be rounded to the nearest integer.
4. Some digits of precision may be lost.
5. Make sure that the destination string variable is declared with sufficient storage.
6. Both must be pointers to the same type (or both "generic" pointers).
7. Both must be handles to the same type (or both "generic" handles).
8. Automatic string/number translation requires a special preference setting; otherwise, use the `val[&]` or `str$` functions.
9. Both must be the same record type.
10. Information about the type of thing referenced (by the handle or pointer) is lost when the handle or pointer value is assigned to a long integer variable. (This can sometimes be useful, if you want to "coerce" a pointer to point to a different type.)

A numeric expression is anything that can be evaluated as a number. A number can be expressed in the following ways:
I. Simple expressions

- A numeric literal, a symbolic constant, or a numeric variable. See Appendix C - *Data types and Data Representation*, for more information.
  Examples: `17.3 _true x& Z%(14)`
- A reference to any user-defined function or Toolbox function that returns a numeric value.
  Examples: **fn** `theSum#` **fn GETCATINFO**`(@pb)`
- A value returned by any built-in FutureBasic function whose name does not end with "`$`". (Note: the two exceptions to this are the `using` function and the `str#` function, both of which return a string value.)
  Examples: **len**`("Hello")` **dialog**`(0)`

II. Data comparison expressions
Data comparison expressions always return the value -1 or 0. In many contexts, these values are interpreted as meaning "true" or "false," respectively. Data comparison expressions have the following forms:
II.1 Equality comparisons
An equality comparison consists of two expressions of "compatible types" separated by the "`=`" operator or the "`<>`" operator (you can also use "`==`" as a synonym for "`=`", and "`!=`" as a synonym for "`<>`"). The two operands must fall into one of the following categories:

- A pair of numeric expressions;
- A pair of string expressions (see Appendix E);
- Any pair of variables of the same type.

An equality comparison using "`=`" (or "`==`") is evaluated as -1 if the two operands have equal values; otherwise it's evaluated as 0. An equality comparison using "`<>`" (or "`!=`") is evaluated as -1 if the two operands are not equal; otherwise it's evaluated as 0. Examples:
```
x& == len(acc$) * 3
"Bronson" <> theName$(7,4)
record1 = record2
```
II.2 Order comparisons
An order comparison can test the relative "order" of two numeric operands, or of two string operands; that is, it tests whether one operand is greater than or less than the other. In the case of strings, `string1$` is considered "less than" `string2$` if it precedes `string2$` alphabetically. More accurately, string comparison depends on the ASCII values of the characters in the strings. An order comparison takes the form `expr1 operator expr2`, where `expr1` and `expr2` are both numeric expressions or both string expressions, and `operator` is one of the operators in this table:

Operator	expr1 operater expr2 returns -1 if and only if:
>	expr1 is greater than expr2
>=, =>	expr1 is greater than or equal to expr2
<	expr1 is less than expr2
<=, =<	expr1 is less than or equal to expr2
>> (strings only)	expr1 is greater than expr2 without regard to letter case
<< (strings only)	expr1 is less than expr2 without regard to letter case

Examples:
```
blt& > ham% + rye%
"hello" << mid$(testString$,x,5)
```
III. Expressions with Unary Operators
A unary operator is an operator that takes only one operand. FutureBasic has three unary operators: "`+`"; "`-`"; "`not`". The unary operator always appears on the left side of the operand; the operand can be any numeric expression.

Operator	Operator expr returns:
+	expr

-	the negative (additive inverse) of expr
Not	the binary 1's complement of expr. See Not in the main part of the manual.

Examples:
```
+n!
-(x# + 12 / 7.3)
not found%
```
IV. Compound Expressions

A compound numeric expression is any list of numeric expressions separated by one or more of the operators in the table below. A compound expression has this form:
```
expr1 operator expr2 [operator expr3 ...]
```
Additionally, any expression which is surrounded by a pair of parentheses is also an expression. When you surround an expression with parentheses, the entire expression within parentheses is evaluated before any operator to the left or right of the parenthetical expression is applied. This is useful when you want to change the default order in which the operators are applied. For example:
```
3 * (7 + 1)
```
In the above expression, the "+" operator is applied before the "*" operator. 3 is multiplied by the sum of 7 and 1, giving a result of 24. But if the expression had been written like this:
```
3 * 7 + 1
```
then the "*" operator would have been applied first. In this case, 1 is added to the product of 3 and 7, giving a result of 22.

Operator	Description
+	Addition.
++	Increment a variable
+=	Add the expression from the right of the equal sign to the variable on the left.
-	Subtraction
--	Decrement a variable
-=	Subtract the expression from the right of the equal sign from the variable on the left.
*	Multiplication.
/	If both operands are integer expressions, this operator does integer division. If either operand is a real number, the operator does floating point division.
\	The operator always does floating point division. Integer operands are converted to floating point before the division.
\\	Identical to /
^	Exponentiation (raising to a power).
=; ==	Comparison.
<<	The first operand is shifted left by the number of bit positions specified by the second operand. Both operands must be integral values. A left shift by n is equivalent to multiplying by 2^n. The result is undefined if n is negative or greater than the width in bits of the first operand.
>>	The first operand is shifted right by the number of bit positions specified by the second operand. Both operands must be integral values. A right shift by n is equivalent to dividing by 2^n with rounding towards minus infinity. The result is undefined if n is negative or greater than the width in bits of the first operand.
And; &&	Bitwise And operator. See the description in the main part of the manual.
Nand; ^&	Bitwise Not And operator. See the description in the main part of the manual.
Or; ||	Bitwise Or operator. See the description in the main part of the manual.
Nor; ^|	Bitwise Not Or operator. See the description in the main part of the manual.
Xor; ^^	Bitwise Xor operator. See the description in the main part of the manual.
Mod	Modulus operator. See the description in the main part of the manual.

Examples:
```
7 + 3 + 6 * 18.7
x& and (not bit(7))
ZZ mod (x% + 8)
```

Operator Precedence

When an expression includes more than one operator, the order in which the operations are performed can affect the result. When an

operator appears to the left or right of a parenthetical expression, all of the operations within the parentheses are performed first. When several operators all appear within the same matching pair of parentheses (or outside of all parentheses), the order in which their operations are performed is determined by their order of precedence, with "higher precedence" operations being performed before "lower precedence" ones. For example, consider this expression:

```
4 + 7 * 5
```

The "`*`" operator has a higher precedence than the "`+`" operator (see the table below). So, when this expression is evaluated, first 7 is multiplied by 5 to get 35; then that result is added to 4 to get the final answer of 39.

The following table lists the operators in order of their precedence, from highest to lowest. When an expression contains several operators at the same level of precedence (and within the same depth of parentheses), their operations are always performed from left to right.

Precedence level	Operator
1	unary "+"; unary "-"; Not
2	^
3	*; /; \; \\; Mod
4	+ (addition); - (substraction)
5	<; <=; >; >=; =; ==; <>; != << (strings); >> (strings)
6	<< (shift left); >> (shift right)
7	And; Or; Xor; Nand; Nor

Example: Consider the following expression:

```
20 – 4 + 3 * (24 / (7 + 1) + 2)
```

The following shows the series of operations that FutureBasic performs to reduce this expression to its final value, 31.

Operation	Resulting expression
20-4 = 16	**16** + 3 * (24 / (7 + 1) + 2)
(7+1) = 8	16 + 3 * (24 / **8** + 2)
24/8 = 3	16 + 3 * (**3** + 2)
(3+2) = 5	16 + 3 * **5**
3*5 = 15	16 + **15**
16 + 15 = 31	**31**

Static Integer Expressions

Many FutureBasic statements require quantities that are expressed as static integer expressions. A static integer expression may be a simple or complex expression, but its operands are limited to the following:

- Integer literals;
- Symbolic constants;
- The `sizeof`, `offsetof` and `typeof` functions.

The following are examples of valid static integer expressions:

```
762
3 * _myConstant + sizeof(x&)
44 / 11
```

The following are not valid static integer expressions:

```
126 + x&
3.14159
sqr(49)
85 + fn Zilch(36)
```

Floating Point Number Display

Controlling the number of floating point digits displayed is performed by setting the global runtime variable "gFBFloatMaxDigits" to the desired value. It governs the number of significant digits kept during conversion of a floating point value to a string. This conversion is the basis of the **str$()** function. An identical conversion occurs when you **print** a floating point value.

The default value of gFBFloatMaxDigits (10) can be changed to suit:

```
gFBFloatMaxDigits = 3 // or 15 or whatever
```

Changing the value of gFBFloatMaxDigits has no effect on the precision of numerical calculations.

For more precise control of number display, see the using function.

String Expressions

A string expression is anything that can be evaluated as a string of 0 to 255 ASCII characters. A string can be expressed in any of the following ways:

I. Simple Expressions

- A string literal, or a string variable. See Appendix C - Data Types and Data Representation, for more information.
  Examples: `surname$(23) "Friday"`
- A reference to any user-defined function that returns a string value. *Examples:* `fn pathName$(v%,dirID&)`
- A value returned by any built-in FutureBasic function whose name ends with "`$`". *Examples:* `chr$(7) hex$(z&)`
- A value returned by the `using` function, or by the `str#` function. *Examples:* `using "##.#"; x! str#(130,5)`

II. Compound Expressions

A compound string expression is a list of simple string expressions separated by the concatenation operator, "+". The syntax of a compound string expression is:

`simpleExpr1 + simpleExpr2 [+ simpleExpr3 ...]`

The "+" operator builds a longer string by concatenating the operands. For example, consider this expression:

`"Ex" + "tra" + mid$("fiction",3)`

This expression has the value, `"Extraction"`.

Note: When two string expressions are separated by a data-comparison operator, such as: `=`; `<`; `>`, the result is a numeric expression. See Appendix D - Numeric Expressions, for more information.

Note: Because the dollar sign ($) is used to designate a full 255 byte pascal string, FutureBasic must determine what you really intended to use when you dimensioned a variable. The following examples demonstrate FutureBasic's evaluation methods:

dim as Str31 `z 'z is a 31 byte string`
dim `z$;32 'z is a 31 byte string`
dim `z$` **as Str31** `'z is a 31 byte string`
dim `z` **as Str31** `'z is a 31 byte string`
dim as Str31 `z$ "$" 'does not work. The $ overridces Str31.`

Note: String length errors are not reported. Instead, strings are silently truncated to the maximum length that will fit in the destination variable.

Alternative syntax for fn CFSTR()

There is a new syntax, borrowed from Objective-C, for obtaining a CF string from a string literal. In most cases the new form @"foo" is interchangeable with fn CFSTR("foo"), but there are differences. @"foo" uses Apple's official CFSTR macro, whereas fn CFSTR("foo") uses a CFSTR emulator in the runtime. The advantage of the '@' form is that it allows escaped characters (preceded by a backslash).

```
dim as CFStringRef cfstr
cfstr = @"printable ascii" // same as fn CFSTR( "printable ascii" )
cfstr = @"item 1\ritem 2" // embedded return char; same as fn CFSTR( "item 1" + chr$( 13 ) + "item 2" )
cfstr = @"\"" // double-quote char; same as fn CFSTR( """" )
cfstr = @"føµ" // non-ASCII chars; don't do that! Instead use fn CFSTR( "føµ" )
```

The '@' form requires a string literal (not a string expression).

Appendix F - ASCII Character Codes   **appendix**

Description:

Characters 0 to 127 of the MacRoman character set are identical to ASCII and are standardized and portable. Characters 128 to 255 are Mac-specific, and even on a Mac are applicable only to text (and text documents) whose encoding is kTextEncodingMacRoman.

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	∆	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	"	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	"	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	'	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	'	245	ı
22	<SYN>	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	^
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	~
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	/	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124			156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	fl	255	ˇ

Special Cases:

<NUL> = Null   <DC1> = Device Control 1

<SOH> = Start Of Heading   <DC2> = Device Control 2

<STX> = Start Of Text   <DC3> = Device Control 3

<ETX> = End Of Text   <DC4> = Device Control 4

<EOT> = End Of Transmission   <NAK> = Negative Acknowledge

<ENQ> = Enquiry   <SYN> = Synchronous Idle

<ACK> = Acknowledge   <ETB> = End Of Transmission Block

<BEL> = Bell   <CAN> = Cancel

<BS>	= Backspace	<EM>	= End Of Medium
<TAB>	= Tab	<SUB>	= Substitute
<LF>	= Line Feed	<ESC>	= Escape
<VT>	= Vertical Tab	<FS>	= File Separator
<FF>	= Form Feed	<GS>	= Group Separator
<CR>	= Carriage Return	<RS>	= Record Separator
<SO>	= Shift Out	<US>	= Unit Separator
<SI>	= Shift In	<SPC>	= Space
<DLE>	= Data Link Escape	<DEL>	= Delete

Description:
Symbol Table:

Symbol	Example	Description
`	` MOVEQ #0,D0	When used as the first character in a line, the grave (back apostrophe) tells the compiler that the code on that line should be handled by the PPC or 68K assembler.
byte`	x`= expr	Signed byte variable
byte``	x``= expr	Unsigned byte variable
word%	x% = expr	Signed integer
word%`	x%` = expr	Unsigned integer
long&	x& = expr	Signed long integer
long&`	x&` = expr	Unsigned long integer
single!	x! = expr	Single precision
double#	x# = expr	Double precision
"	"text"	Literal string
$	x$ = expr	Pascal String
$$	x$$ = expr	2 Gig container
;	Dim x;4	Force a specific size for a dimensioned variable
||	expr || expr	Or
&&	expr && expr	And
^&	expr ^& expr	Nand (Not And)
^|	expr ^| expr	Nor (Not Or)
^^	expr ^^ expr	Xor
!=	expr != expr	Not equal < >>
_	_constant = 4	Identifies a constant
_	_"PICT"	The text in quotes is taken as a 4 byte restype or OStype
||	|expr|	Peek (or Peek Byte)
{}	{expr}	Peek Word
[]	[expr]	Peek Long
|	| expr	Poke (or Poke Byte)
%	% expr	Poke Word
&	& expr	Poke Long
&	&hexExpr	Hexadecimal number
&H	&HhexExpr	Hexadecimal number

0x	0xhexExpr	Hexadecimal number
&O	&Oexpr	Octal literal
&X	&Xexpr	Binary literal
'	' remark	Indicates the beginning of a remark
//	// remark	Indicates the beginning of a remark
/* */	/* remark */	Marks the beginning and end of a multi-line block remark
#	#parameter	If a function or proceedure expects to receive a variable (to which it may point for an address) as a parameter, you cannot substitute a specific address. The pound (#) symbol overrides that feature and tells FutureBasic not to convert the parameter to an address.
@	@varName	Specifies that the operation should use the address of a variable rather than the contents of a variable. This does not work for register variables.

Description:
You may envision the printed page as something very similar to a window. In general, commands used to produce any type of display on the screen will produce a similar imprint on the page. The exception would be controls which cannot be sent to the printer port.
You instruct your program to switch to the printer using the `route` command.
route `_toPrinter`
rem `printing commands here`
route `_toScreen`
You may freely switch back and forth between the printed page and the screen by executing `route` commands. When it is time to eject a page or to terminate printing entirely, you can clear the page with `clear lprint` or close down the printer (which has the side effect of automatically clearing the page) with `close lprint`.

Page Size:
You can query the printer as to how large the page is by routine output to the printer, then executing **window()** functions.
route `_toPrinter`
`pageWidth =` **window**`(_width)`
`pageHeight =` **window**`(_height)`
route `_toScreen`

Print Dialogs:
Two dialogs are used before printing. The first is a style dialog that lets the user determine page orientation, scaling, and other items. This is usually brought up in response to selection of the Page Setup item under the File menu.
The second common dialog is a job dialog. It lets the user determine how many copies will be printed, which page numbers will be included, and other items that vary from one printer to the next. The job dialog is brought up with `def lprint` and is normally displayed before each print session. Note that the Print Manager actually handles the details of the job. If the user wants to print 2 copies of pages 7 through 10, your application may happily print a single copy of the entire document and the Print Manager correctly filter the output to adhere to the user's request.

Note:
Do not call `clear lprint` or `close lprint` when output is being routed to the printer. This may cause the system to crash. Instead, route output back to the screen, then clear or close.

Appearance Manager Printing:
Because buttons cannot be sent to the printed page, Appearance Manager edit fields cannot be printed. There is a simple work around. Create the edit fields in a window, then use the edit field statement (with only the field number as a parameter) and it will be copied to the printer. The following example show how this is done.
// Appearance Manager printing
window `1`
edit field `1,"This is a test",(10,10)-(120,32)`
`// Now print it`
route `_toPrinter`
edit field `1`
route `_toScreen`
In this example, we did not clear or close the printer (`clear lprint` or `close lprint`). This is because the operation is automatically performed when the program terminates.

Description:
Date & Time Symbols:

Field	Sym.	No.	Example	Description
era	G	1..3	AD	Era - Replaced with the Era string for the current date. One to three letters for the abbreviated form, four letters for the long form, five for the narrow form.
		4	Anno Domini	
		5	A	
year	y	1..n	1996	Year. Normally the length specifies the padding, but for two letters it also specifies the maximum length. Example: **Year** y yy yyy yyyy yyyyy AD 1 1 01 001 0001 00001 AD 12 12 12 012 0012 00012 AD 123 123 23 123 0123 00123 AD 1234 1234 34 1234 1234 01234 AD 12345 12345 45 12345 12345 12345
	Y	1..n	1997	Year (in "Week of Year" based calendars). This year designation is used in ISO year-week calendar as defined by ISO 8601, but can be used in non-Gregorian based calendar systems where week date processing is desired. May not always be the same value as calendar year.
	u	1..n	4601	Extended year. This is a single number designating the year of this calendar system, encompassing all supra-year fields. For example, for the Julian calendar system, year numbers are positive, with an era of BCE or CE. An extended year value for the Julian calendar system assigns positive values to CE years and negative values to BCE years, with 1 BCE being year 0.
quarter	Q	1..2	02	Quarter - Use one or two for the numerical quarter, three for the abbreviation, or four for the full name.
		3	Q2	
		4	2nd quarter	
	q	1..2	02	**Stand-Alone** Quarter - Use one or two for the numerical quarter, three for the abbreviation, or four for the full name.
		3	Q2	
		4	2nd quarter	
month	M	1..2	09	Month - Use one or two for the numerical month, three for the abbreviation, or four for the full name, or five for the narrow name.
		3	Sept	
		4	September	
		5	S	
	L	1..2	09	**Stand-Alone** Month - Use one or two for the numerical month, three for the abbreviation, or four for the full name, or 5 for the narrow name.
		3	Sept	
		4	September	
		5	S	
	l	1	*	Special symbol for Chinese leap month, used in combination with M. Only used with the Chinese calendar.
week	w	1..2	27	Week of Year.

	W	1	3	Week of Month
day	d	1..2	1	Date - Day of the month
	D	1..3	345	Day of year
	F	1	2	Day of Week in Month. The example is for the 2nd Wed in July
	g	1..n	2451334	Modified Julian day. This is different from the conventional Julian day number in two regards. First, it demarcates days at local zone midnight, rather than noon GMT. Second, it is a local number; that is, it depends on the local time zone. It can be thought of as a single number that encompasses all the date-related fields.
week day	E	1..3	Tues	Day of week - Use one through three letters for the short day, or four for the full name, or five for the narrow name.
		4	Tuesday	
		5	T	
	e	1..2	2	Local day of week. Same as E except adds a numeric value that will depend on the local starting day of the week, using one or two letters. For this example, Monday is the first day of the week.
		3	Tues	
		4	Tuesday	
		5	T	
	c	1	2	**Stand-Alone** local day of week - Use one letter for the local numeric value (same as 'e'), three for the short day, or four for the full name, or five for the narrow name.
		3	Tues	
		4	Tuesday	
		5	T	
period	a	1	AM	AM or PM
hour	h	1..2	11	Hour [1-12].
	H	1..2	13	Hour [0-23].
	K	1..2	0	Hour [0-11].
	k	1..2	24	Hour [1-24].
minute	m	1..2	59	Minute. Use one or two for zero padding.
second	s	1..2	12	Second. Use one or two for zero padding.
	S	1..n	3457	Fractional Second - rounds to the count of letters. (example is for 12.34567)
	A	1..n	69540000	Milliseconds in day. This field behaves *exactly* like a composite of all time-related fields, not including the zone fields. As such, it also reflects discontinuities of those fields on DST transition days. On a day of DST onset, it will jump forward. On a day of DST cessation, it will jump backward. This reflects the fact that is must be combined with the offset field to obtain a unique local time value.
zone	z	1..3	PDT	Time Zone - with the s*pecific non-location format*. Where that is unavailable, falls back to *localized GMT format*. Use one to three letters for the short format or four for the full format. In the short format, metazone names are not used unless the commonlyUsed flag is on in the locale.
		4	Pacific Daylight Time	
	Z	1..3	-0800	Time Zone - Use one to three letters for RFC 822 format, four letters for the localized GMT format.
		4	HPG+8:00	
	v	1	PT	Time Zone - with the *generic non-location format*. Use one letter for short format, four for long format.
		4	Pacific Time	
	V	1	PST	Time Zone - with the same format as z, except that metazone timezone abbreviations are to be displayed if available, regardless of the value of commonlyUsed.
		4	United States (Los Angeles) Time	Time Zone - with the *generic location format*. Where that is unavailable, falls back to the localized GMT format.

Source: http://unicode.org/reports/tr35/

Description:
FutureBasic normally builds standard Mac applications. However, FutureBasic can also be used to build console applications and command-line tools. The choice between an FutureBasic console application or a command-line tool is made at the beginning of the program. A simple compiler conditional determines which file is to be included:

```
_buildAsCommandLineTool = _true // _true or _false
#if _buildAsCommandLineTool
include "CommandLineTool"
#else
include "ConsoleWindow"
#endif
```

Most console apps can be built as tools with no other changes.

Retrieval of Command-Line Arguments:
The arguments that are passed on the command line can be inspected using:
tool_argc - the number of arguments
fn tool_arg(j) - Str255 containing up to 255 characters of the jth argument
tool_argv - a pointer giving access to the C strings in "argv[]". This can be used to parse arguments longer than 255 characters.

A command-line tool always has at least one argument: the path to the command. More arguments may follow if the tool was run via a terminal or pipe.

```
#if def _FBUnixTool // tool_argc etc available only for tool, not console
dim as long j
print "Number of arguments =" tool_argc
for j = 0 to tool_argc - 1
 print fn tool_arg( j ) // fn tool_arg() returns a Str255
next
print
#endif
```

```
$ ./test.command 123 nine "a quoted string"
Number of arguments = 4
/Users/username/Desktop/test.command
123
nine
a quoted string
```

The constant _FBUnixTool is defined by FBtoC when building a tool, and is undefined otherwise. It is used in the code snippet above to 'conditionalise out' the argument inspection code when building a console app. Only tools receive useful arguments.

Environment Variables:
The command line tool can read any named environment variable from the host environment list as illustrated below:

```
#if def _FBUnixTool // tool only, not console
print fn tool_getenv( "HOME" ) // fn tool_getenv() returns a Str255
#endif
```

Standard Input and Output:
In a tool, the input statement reads from stdin and the print statement writes to stdout.

```
dim as Str255 s
input s
print "The input was: " s
```

Note that only the first 255 characters will be read, or up to the newline character, whichever comes first.

File I/O:
A tool can perform any file-system operation except display a navigation dialog. Several functions in Util_FileDirectory.incl allow access to files in predictable locations. FD_PathGetFSRef() and FD_PathCreateCFURL() can be used to convert a path, passed as a tool argument, to a FSRef or CFURLRef.

64-bit Executables:
Command line tools (but not console apps) can be compiled in 64-bit mode. This is done by putting "-m64" into FBtoC's 'More compiler options'. That setting tells the compiler to produce a 64-bit executable instead of the older 32-bit kind.

```
#if def _LP64 // constant defined by FBtoC when building in 64-bit mode
print "Compiled in 64-bit mode"
#endif

def tab 10
// these values are 8 when compiled in 64-bit mode
print "sizeof( long )", sizeof( long ) " bytes"
print "sizeof( pointer )", sizeof( pointer ) " bytes"
```

How to make an Xcode project from a FutureBasic command-line tool:

[1] Put an Info.plist file in the same folder as the FutureBasic source. The xml content can be vacuous.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict/>
</plist>
```

[2] Set Optimization to None in your FutureBasic project window. (For a standalone source file, make the setting in FBtoC.app's preferences).

[3] Build and Run the FutureBasic source. You can either leave toolname.command running in Terminal.app or kill it.

[4] Bring FBtoC.app to the front, and choose File > Make Xcode Project.

[5] In Xcode, Build and Run the translated C source. To see the output, ensure that Xcode's Debugger Console is shown, perhaps by choosing the command Run > Console.

[6] By appropriate manipulations in the Finder you can drag the executable from "build_temp/XcodeProject/build/Debug/toolname.app/Contents/MacOS/toolname" to a more sensible location.

[7] A later release of FutureBasic (version 5 onwards) may provide a special Xcode project template to make step #1 unnecessary and step #6 simpler.

Build System Caching

During compilation, data is cached in the source directory's build_temp folder. Reusing this data in subsequent builds allows some steps of compilation to be skipped. Builds subsequent to the first one are therefore faster. The caches affect the compilation phase only, not translation.

The caches are maintained as files in build_temp. If build_temp or its files are missing, the files are regenerated. If certain critical compilation settings are changed, the cached information is invalidated, and automatically regenerated during the next build.

Two kinds of cached information are maintained, specifically precompiled FutureBasic (version 5 onwards) code and header information from the MacOS X frameworks.

Precompiled FutureBasic code

The compiled code is in a series of *.o files in build_temp. Each *.o file contains binary code derived from one or more FutureBasic source files. For example:

_0_TranslatedRuntime.o
...
_15_Loop_Statemen,Conditional.o
...
_19_Init,Main_,FBtoC,FBtoC.o

A *.o file is compiled from its matching C source (*.c and *.h files). If these are both unchanged since the last build, compilation can be skipped.

Special indicator files are used to vouch for the integrity of *.o files; their long names (like _0_TranslatedRuntime.c_9aee936b...1b85b8e50_10023_149.vch) encode the md5 checksum of the matching C source along with the current compiler settings.

Header information from the MacOS X frameworks

Also known as a precompiled header or pch, this cache is controlled by the settings checkbox 'Use precompiled header'.

The cache, in a format private to the compiler, is in the folder build_temp/FBtoCPrefix.h.gch/

Both forms of caching are most effective when repeated builds for a single architecture and with the same settings are built. This is typically the case during program development, for which fast compilation is especially desirable. The recommended development settings are:

Architecture = Current Mac
Use precompiled header = Yes
Optimization = None

If, on the other hand, certain settings are changed, the entire program (and pch if enabled) must be recompiled.

The relevant FBtoC settings are:

Architecture
Min OS deployment
Max OS features
Use DWARF debug format
Debug level
Optimization.

Changing any of those settings invalidates all cached information at the next build.

Running Under Xcode Debugger

Xcode preferences

Turn off 'Load symbols lazily', and set to show Debugger at startup:

Build settings

Build your app with FBtoC settings 'Debug level: fn names and line #'.

If possible, turn off the checkbox 'Allow dim a%, a&, a#, a$'. This makes variable names in C easier to read.

Debugging

After the build, choose File > Debug Last App.

FBtoC sets up an Xcode project to debug your app as a "custom executable" (see Debugging Arbitrary Applications With Xcode)

All going well, you get full source-code debugging, with symbolic break-points etc.:

Reference

Xcode User Guide: Debugging
Xcode User Guide: Attaching to a Running Process

Includes

*.icns file from 'icns' resource (to <app>/Contents/Resources)
If an 'icns' resource exists in a project resource file, the 'icns' resource is saved as a file in /Contents/Resources. This is automatic and does not require the use of the include resource statement. The file name is derived from the resID of the 'icns' resource, eg. 128.icns. This is the same method used by MakeBundle.

Info.plist (in <app>/Contents/)
FutureBasic automatically copies a generic Info.plist into the compiled bundle's Contents directory. That generic Info.plist template can be found in FutureBasic at: build_goodies/BuiltApplication.app/Contents/Info.plist.in

A consequence of using that generic template is that it indicates FBtoC's generic BlueLeafC.icns as the *.icns file. Hence, the compiled application will display FutureBasic/FBtoC's generic BlueLeaf icon.

The user may override the default behavior based on "Info.plist.in" by including a user-supplied Info.plist file in the FutureBasic source folder. FBtoC will automatically find the user-supplied Info.plist and use that, thus overriding its default behavior.

In that case, the user will also have to include a *.icns file corresponding to that pointed to by CFBundleIconFile in the custom Info.plist. The user's custom *.icns file can be copied to the compiled application bundle with 'include resources':

```
include resources "AppIcon.icns"
```

Requirements

FBtoC is the translation component of the FutureBasic package. Before using FutureBasic version 5 onwards along with FBtoC, you should have installed Apple's free Developer Tools (a.k.a. Xcode Tools), from your MacOS X install CDs or DVD.

The cross-development SDKs must be installed to create universal binaries. Also, only the native architecture of your Mac can be built unless the SDKs for prior (meaning releases prior to the one active where FutureBasic is installed) MacOS X releases are installed. The Finder view (below) of the Developer folder shows the optional MacOS X 10.4 and MacOS X 10.5 SDKs after installation on MacOS X 10.6. Cross-development is an optional part of the Xcode Tools install (from the MacOS X DVD or CDs):
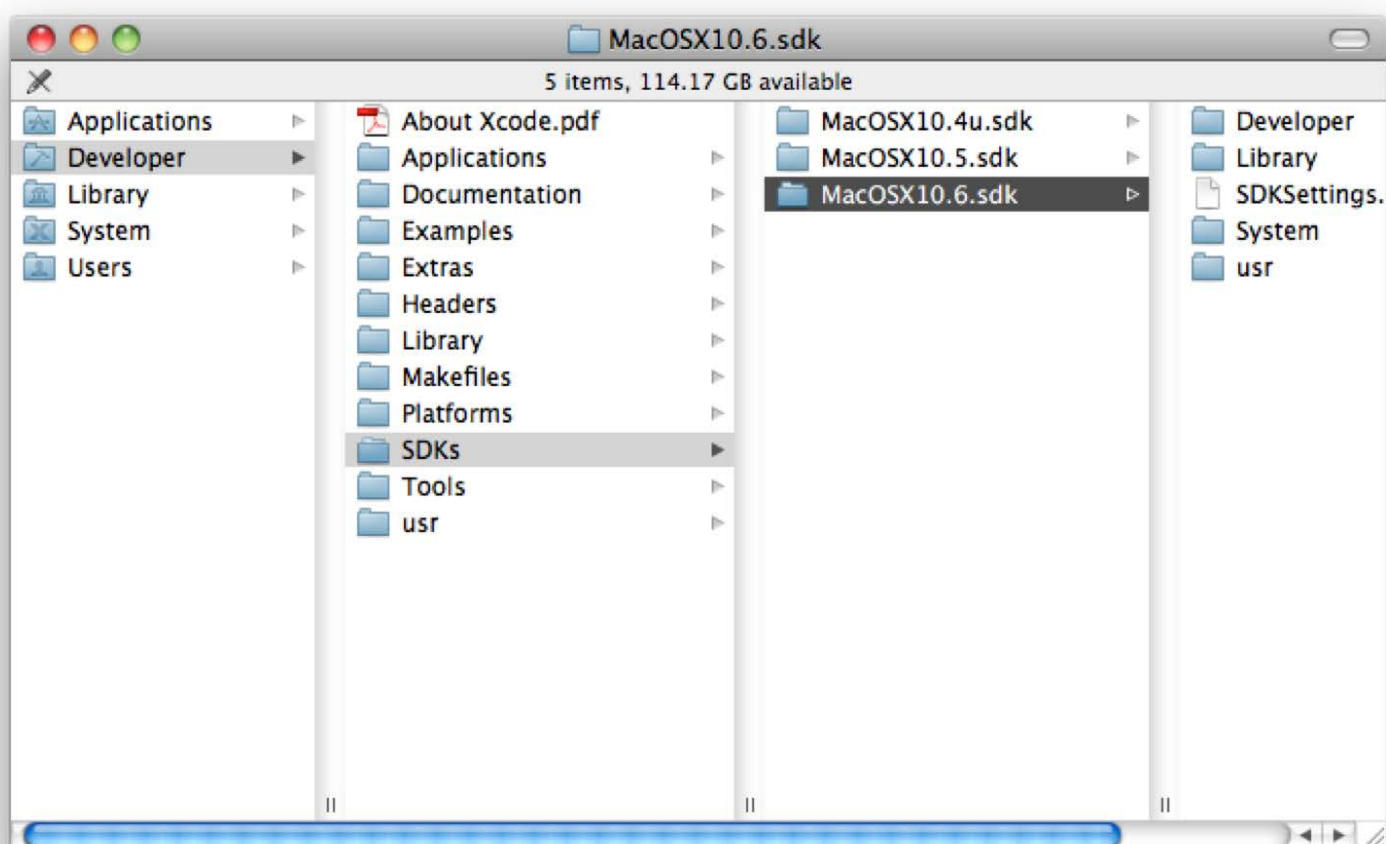
See Apple Documentation for more information about cross-development.

Settings

The FBtoC settings dialog is self-documenting by virtue of the titles, help tags, and grouping of the controls.

FAQ

- **What should I do with all those build_temp folders created by FBtoC?**

    Command-delete works well for me. But why delete them so soon? If you have the 'Use precompiled header' checkbox ON, build_temp

folders contain, er, the precompiled header. It's that huge file FBtoC.h.gch, used by the compiler as a cache.

- **Why do I get warnings like this from the compiler?**

  /Volumes/HD2/FBtoC/build_temp/Blah.c: In function 'SendHandleMessageToLog':
  /Volumes/HD2/FBtoC/build_temp/Blah.c:3027: warning: passing argument 1 of 'BlockMoveData' makes pointer from integer without a cast
  /Volumes/HD2/FBtoC/build_temp/Blah.c: In function 'FBToCResourcesCopy':
  /Volumes/HD2/FBtoC/build_temp/Blah.c:3955: warning: passing argument 4 of 'GetResInfo' from incompatible pointer type

  FutureBasic version 4 and its predecessor releases were weakly typed; and made little distinction between pointers and other 4-byte vars such as longs. C is more strongly typed, and the compiler complains when the wrong 4-byte type is used. These warnings are nearly always harmless and can nearly always be ignored. These warnings can be eliminated by typing variables correctly and not relying on FutureBasic's weak typing. Cleaning up the compiler warnings helps identify (often there is a long list of warnings) any warnings that DO need to be addressed, so cleaning up even harmless warnings can be beneficial.

- **I know my compilation failed because it says so at the end, but I can't find the compiler error among the horrible spew of warnings.**

  Suppress the spew by typing this into 'More compiler options' in FBtoC's settings dialog:
  -w

- **Sometimes I get two sets of identical warnings from the compiler. Why?**

  A Universal app is built by compiling the C source twice: once with -arch ppc and the other with -arch i386.

- **Namespace collision**

  Q: For some reason there is a problem with FN InstallControlEventHandler for the attached demo. The compiler keeps complaining that there are supposed to be six parameters and it only has one. I've looked at the C code and there are six parameters. I have no idea why it won't take it.

  A: InstallControlEventHandler is the name of macro in Carbon. You have a function in your code with the same name. You'll have to rename it, for instance to MyInstallControlEventHandler.

- **WTF does "void value not ignored as it ought to be" mean?**

  /Volumes/HD2/FBtoC/Test Files/build_temp/Demo.c: In function 'main':
  /Volumes/HD2/FBtoC/Test Files/build_temp/Demo.c:1680: error: void value not ignored as it ought to be
  You are trying to get a return value from a pure procedure. Here's an example from CoreGraphics.

  ```
  include "Tlbx CoreGraphics.incl"
  dim as CGImageRef myImage
  dim something

  // legal but pointless in FutureBasic; illegal in FBtoC
  something = fn CGImageRelease( myImage ) // <-- void value not ignored as it ought to be

  // this syntax avoids the error and is legal in both FutureBasic and FBtoC
  fn CGImageRelease( myImage )

  // illegal in FutureBasic; legal in FBtoC
  call CGImageRelease( myImage )
  CGImageRelease( myImage )
  ```

- **My program won't quit**

  ```
  Change your event loop.
  do
  HandleEvents
  until 0 // never quits


  do
  HandleEvents
  until ( gFBQuit ) // ah, that's better
  ```

Compiling C code Independent of FBtoC (not recommended but possible)

Is it possible to make minor changes to the C code in the "build_temp" folder using XCode and then have FBtoC run directly from the modified "build_temp" folder without going through the original FutureBasic code?

There's no official way, but this workaround isn't especially challenging.

[1] Turn on the 'Log UNIX commands' checkbox in FBtoC preferences settings.

[2] In FBtoC, open your project and wait until compilation finishes.

[3] Optionally edit the *.c files in the relevant build_temp folder.

[4] From your FBtoC Log window, copy the 3 consecutive blue lines beginning with cd, gcc, and touch. For example:

cd /Users/username/Desktop/build_temp;
gcc -I/FutureBasic/FBtoC/FBtoC_Preview1a117/build_goodies /Users/username/Desktop/build_temp/untitled_1.c -fpascal-strings -framework Carbon -framework QuickTime -framework IOKit -o /Users/username/Desktop/untitled\ 1.app/Contents/MacOS/untitled\ 1 -mdynamic-no-pic -trigraphs -Wall -Wno-trigraphs -Wno-sequence-point -Wno-multichar -Wno-deprecated-declarations -Wno-unused-label -Werror-implicit-function-declaration -O0 -pipe -gused -Wl,-dead_strip 2>&1
touch /Users/username/Desktop/untitled\ 1.app

[5] Paste these lines into Terminal.app to replicate the compilation step of the FBtoC build.

- **FlushWindowBuffer with FBtoC**

  Window flushing is very different in a Mach-O app (as produced by FBtoC) compared with a legacy CFM app (as produced by FBCompiler.app). A Mach-O app takes advantage of a Carbon optimization called "coalesced updates".
  The special first parameters for FlushWindowBuffer (_FBAutoFlushOff, _FBAutoFlushOn and _AutoFlushPrint) are therefore neither defined nor needed in FBtoC.

  Please change your code to:

  #if ndef _FBtoC
  FlushWindowBuffer _FBAutoFlushOff // FutureBasic sees this; FBtoC does not
  #endif

- **Why aren't all the error messages this good?**

  ... Not yet implemented by FBtoC in line 25 of Demo.bas: use #if/#else/#endif instead
  25: compile long if 1
  ^
  No answer has been received for this FAQ.

Endian Issues

The original FutureBasic Compiler, specifically the compiler prior to FBtoC, always produced traditional Mac big-endian PPC code. On an Intel Mac, such apps are run in Rosetta and thus keep their big-endian property. FBtoC can produce either big-endian or little-endian or both, depending on the Architecture setting (PPC, Intel, Universal).

Many FutureBasic programs, when run in little-endian mode for the first time, turn out to have bugs. Endian bugs affect multibyte numeric variables: short, long, pointer, single and double (along with synonyms: SInt16, SInt32, ptr...). Strings and 1-byte numeric vars (char, unsigned byte, UInt8...) are not affected directly by endianness.

Byte order in memory

As an illustrative example, consider how we might display the most- and least-significant byes of a short variable.

```
dim as short myShortVar
dim as byte lsByte, msByte
myShortVar = 1
print , "msByte", "lsByte"
```

... rest of program follows later...

Output (on Intel Mac):

```
                    msByte      lsByte
buggy method          1           0
fix 1                 0           1
fix 2                 0           1
fix 3                 0           1
fix 4                 0           1
```

Old code that assumes big-endian format :-(

```
print "buggy method",
msByte = peek( @myShortVar )      // endian bug on Intel
lsByte = peek( @myShortVar + 1 ) // endian bug on Intel
print msByte, lsByte
```

Patch a copy of the byte-peeking code to work little-endian, then set things up so the patch gets used on Intel (only).
The magic constant _LITTLEENDIAN can be used.
Con: conditional compilation makes code hard to read, understand and maintain.

```
print "fix 1",
#if def _LITTLEENDIAN // Intel
msByte = peek( @myShortVar + 1 ) // byte 1
lsByte = peek( @myShortVar ) // byte 0
#else // PPC
msByte = peek( @myShortVar ) // byte 0
lsByte = peek( @myShortVar + 1 ) // byte 1
#endif /* def _LITTLEENDIAN */
print msByte, lsByte
```

Byte-swap the data, do our calculation, then swap it back again.
CFSwapXxxxHostToBig() and CFSwapXxxxBigToHost() swap on Intel but not on PPC.
Pro: doesn't use conditional compilation
Con: remember to swap back again

```
print "fix 2",
include "Tlbx CFByteOrder.incl"
myShortVar = fn CFSwapInt16HostToBig( myShortVar )
// myShortVar is now certainly big-endian
// so we can safely use "buggy method"'s code
msByte = peek( @myShortVar ) // rescued from endian bug on Intel
lsByte = peek( @myShortVar + 1 ) // rescued from endian bug on Intel
myShortVar = fn CFSwapInt16BigToHost( myShortVar ) // restore previous byte order
print msByte, lsByte
```

Convert the data into a string, character sequence, text stream etc, which are inherently endian safe
Pro: doesn't use conditional compilation or byte-swapping
Con: slow, obfuscated

```
print "fix 3",
dim as Str255 tempString
defstr long
tempString = hex$( myShortVar )
msByte = val&( mid$( tempString, tempString[0] - 4, 2 ) )
lsByte = val&( right$( tempString, 2 ) )
print msByte, lsByte
```

Write endian-safe code in the first place.

Pro: shifting and masking are endian safe and very fast, so for this problem, fixes 1-3 above are redundant.

Con: not always possible

```
print "fix 4",
msByte = myShortVar >> 8
lsByte = myShortVar and 0x00FF
print msByte, lsByte
```

Byte order on disk: data

```
// string format on disk; endian safe
print #fileNum, anything
input #fileNum, anything

// written via the FBtoC runtime as big-endian on disk
// automatically swapped by the runtime to little-endian read on Intel
write #fileNum, shortVar, longVar, singleVar, doubleVar, int64Var // endian safe
read #fileNum, shortVar, longVar, singleVar, doubleVar, int64Var // endian safe

write #fileNum, anyRecordVar // on disk, has endianness of writer host; potential endian bug
read #fileNum, anyRecordVar // byte order on disk preserved in reader host memory; potential endian bug

write file #fileNum, address, numBytes // on disk, has endianness of writer host; potential endian bug
read file #fileNum, address, numBytes // byte order on disk preserved in reader host memory; potential
endian bug
```

Byte order on disk: resources

Standard system-defined resource types (e.g. STR#, moov, MENU, etc) are big-endian on disk. System-supplied 'resource flippers' automatically byte-swap on an Intel Mac, in both read and write directions, during any call to any call to the relevant Resource Manager functions. Standard resource types are thereby made to take on the endianness of the host, and all your resource management code should just work, unchanged, on Intel. (Inept coding on your part, though, could get the ResType wrong, so that you ask for a 'voom' resource).

Custom resources have a potential endian bug.

Swapping floating point values in an array

```
include "Subs FloatByteSwapping.incl"
dim as single value(9)
dim as long j, n

// read big-endian array in one chunk from disk
n = 10
read file #1, @value(0), n*sizeof( single )

// make it host-endian
for j = 0 to n - 1
value(j) = fn SwapSingleBigToHost!( value(j) )
next
// value array is now host-endian, ready for use
//...

// make it big-endian again
for j = 0 to n - 1
value(j) = fn SwapSingleHostToBig!( value(j) )
next

// write big-endian array in one chunk to disk
write file #1, @value(0), n*sizeof( single )
```

SwapDoubleBigToHost#() and SwapDoubleHostToBig#() are available for swapping doubles similarly. This code works in PPC and Intel, FutureBasic versions 4 or 5. If your FutureBasic project is FBtoC-only, you can remove the ugly #! suffices from the swapping functions.

Core Foundation/Foundation Framework

Use of Core Foundation both in memory and to read/write to files handles any endian issues automatically. The XML-style files (key/value coding, aka: KVC and commonly referred to as property list files) created are by definition endian safe because they are architecture-independent.

Reference

Xcode User Guide: Universal Binary Programming Guidelines
Xcode User Guide: Swapping Bytes
Xcode User Guide: Byte-Order Utilities Reference