

Chapter
IV-9

Dependencies

Dependency Formulas 196
Dependencies and the Object Status Dialog 197
Numeric and String Variable Dependencies 198
Wave Dependencies 199
Cascading Dependencies 199
Deleting a Dependency 201
Broken Dependent Objects 202
When Dependencies are Updated 202
Programming with Dependencies 202
Using Operations in Dependency Formulas 203
Dependency Caveats 203

Dependency Formulas

Igor Pro supports “dependent objects”. A dependent object can be a wave, a global numeric variable or a global string variable that has been linked to an expression. The expression to which an object is linked is called the object’s “dependency formula” or “formula” for short.

The value of a dependent object is updated whenever any other global object involved in the formula is modified (even if its value stays the same). We say that the dependent object *depends* on these other global objects *through* the formula.

You might expect that an assignment such as:

```
wave1 = sin(K0*x/16)
```

meant that wave1 would be updated whenever K0 changed. It doesn’t; values are computed for wave1 only once, and the relationship between wave1 and K0 is forgotten.

However, if the = in the above assignment is replaced with :=

```
wave1 := sin(K0*x/16)
```

then Igor *does* create just such a dependency. Now whenever K0 changes, the contents of wave1 will be updated. In this example, wave1 is a dependent object. It depends on K0, and $\sin(K0*x/16)$ is wave1’s dependency formula.

You can also establish a dependency using the SetFormula operation, like this:

```
SetFormula wave1, "sin(K0*x/16) "
```

Wave1 depends on K0 because K0 is a changeable variable. Wave1 *also* depends on the function x (x is not a variable) that returns the X scaling of the destination wave (wave1). When the X scaling of wave1 changes, the values that the x function returns change, so this dependency assignment is reevaluated. The remaining terms (sin and 16) are not changeable, so wave1 does not depend on them.

Like other assignment statements, the data folder context for the right-hand side is that of the destination object. Therefore, in the following example, wave2 and var1 must be in the data folder named foo, var2 must be in root and var3 must be in root:bar.

```
root:foo:wave1 := wave2*var1 + ::var2 + root:bar:var3
```

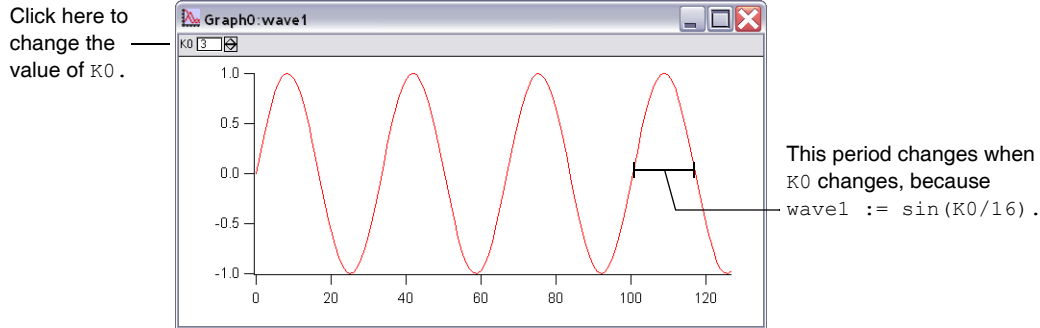
Data Folders are described in Chapter II-8, **Data Folders**.

A dependency assignment is often used in conjunction with SetVariable controls (see page III-358) and Value Display controls (see page III-359).

Here’s a simple example. Execute these commands on the command line:

```
K0=1
Make/O wave1:=sin(K0*x/16)
Display /W=(4,53,399,261) wave1
ControlBar 23
SetVariable setvar0, size={60,15}, value=K0
```

This results in the following graph:



Click the SetVariable control's up and down arrows to adjust K0 and observe that wave1 is automatically updated.

Dependencies and the Object Status Dialog

You can use the Object Status dialog in the Misc menu to check dependencies. On the Macintosh, all dependent objects are listed in the Current Object pop-up menu under Dependent Objects:

No objects depend on wave1. The "Current Object" is the wave named "wave1".

These objects depend on ... none_

the Current Object **wave1**

Wave
Status: Dependency is OK
Type: SP
Data Folder: root:

... which depends on these objects

v: K0

wave1 depends on the variable (v:) named "K0".

Broken Objects
All Objects
Dependent Objects w: wave1
Objects In Target
Waves
Variables
Strings
Annotations
Controls
User Functions
Background Task

Make Current Object Browse Waves... Misc

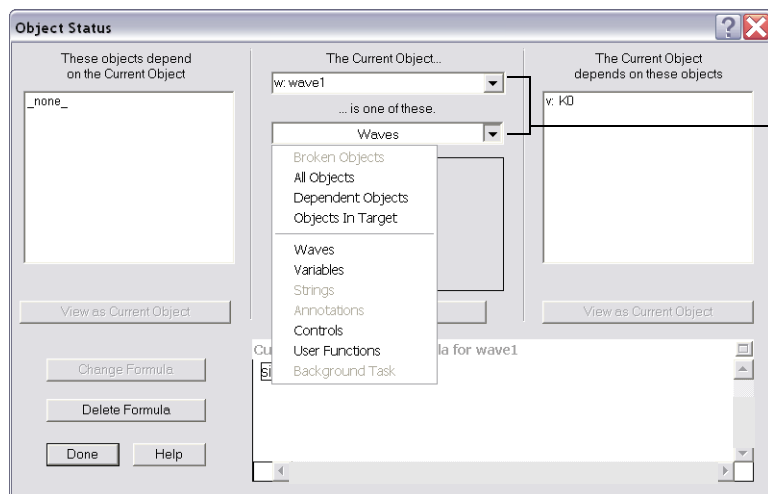
Change Formula Delete Formula Done Help

Current Dependency Formula for wave1: $\sin(K0*x/16)$

The "Current Object" pop-up menu.

$wave1 := \sin(K0*x/16)$

The Windows version of the Object Status dialog is essentially the same but for a slightly different arrangement of the pop-up menus:



Windows dialog pop-up menus for individually selecting the object and type of object.

The Status field in the box below the current object name indicates any dependency status:

- “No dependency” means that the current object does not depend on anything.
- “Dependency is OK” means that the dependency formula successfully updated the current object.
- “Update failed” means that the dependency formula used to compute the current object’s value failed, probably because there is a syntax error in the formula or one of the objects referenced in the formula does not exist or has been renamed. If an update fails, then the objects that depend on that update are broken and they appear in the Broken Objects submenu. See **Broken Dependent Objects** on page IV-202.

You can create a new dependency formula with the New Formula button, delete one with the Delete Formula button, change an existing one by typing in the Dependency Formula window and clicking the Change Formula button, and undo that change by clicking the Restore Formula button.

This is discussed further in **The Object Status Dialog** on page III-413.

You can also read the text of a dependency formula with the string function `GetFormula`, and set it with the operation `SetFormula`.

Numeric and String Variable Dependencies

Dependencies can also be created for global (but not local) user-defined numeric and string variables. Here is a user-defined function that creates a dependency (the global variable `recalculateThis` will depend on another global variable `dependsOnThis`):

```
Function TestRecalculation()
  Variable/G recalculateThis
  Variable/G dependsOnThis= 1

  // Create dependency on global variable
  SetFormula recalculateThis, "dependsOnThis"

  Print recalculateThis      // Prints original value
  dependsOnThis = 2         // Changes something recalculateThis
  DoUpdate                   // Make Igor recalculate formulae
  Print recalculateThis     // Prints updated value
End
```

Running this function prints the following to the history area:

```
•TestRecalculation()
  1
  2
```

The call to DoUpdate is needed because Igor recalculates dependency formulas only when no user-defined functions are running or when DoUpdate is called.

This function uses SetFormula to create the dependency because the := operator is not allowed in user-defined functions.

Note: You can not create a dependency *for* system numeric variables K0...K19 or veclen. You can create a dependency for something else *on* those variables, as in the first example of this chapter. In general, it is actually best if you do *not* use system variables in dependencies, since they are involved in Curve Fitting. You should define your own global variables for use in dependencies.

Wave Dependencies

The assignment statement:

```
dependentWaveName := formula
```

creates a dependency and links the dependency formula to the dependent wave. Whenever any change is made to any object in the formula, the contents of the dependent wave are updated.

The command

```
SetFormula dependentWaveName, "formula"
```

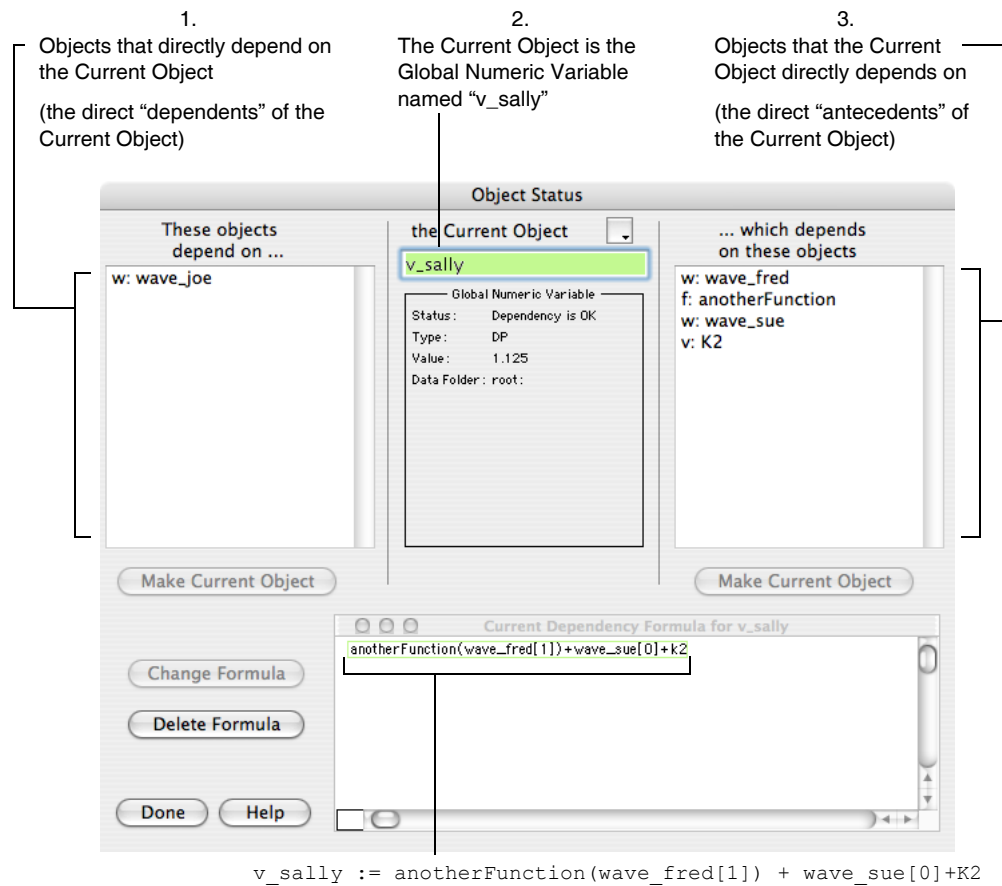
establishes the same dependency.

Cascading Dependencies

“Cascading dependencies” refers to the situation that arises when an object depends on a second object, which in turn depends on a third object, etc. When an object changes, all objects that directly depend on that object are updated, *and* objects that depend directly on those updated objects are updated until no more updates are needed.

The Object Status dialog shows three levels of dependency anchored on the Current Object:

Chapter IV-9 — Dependencies

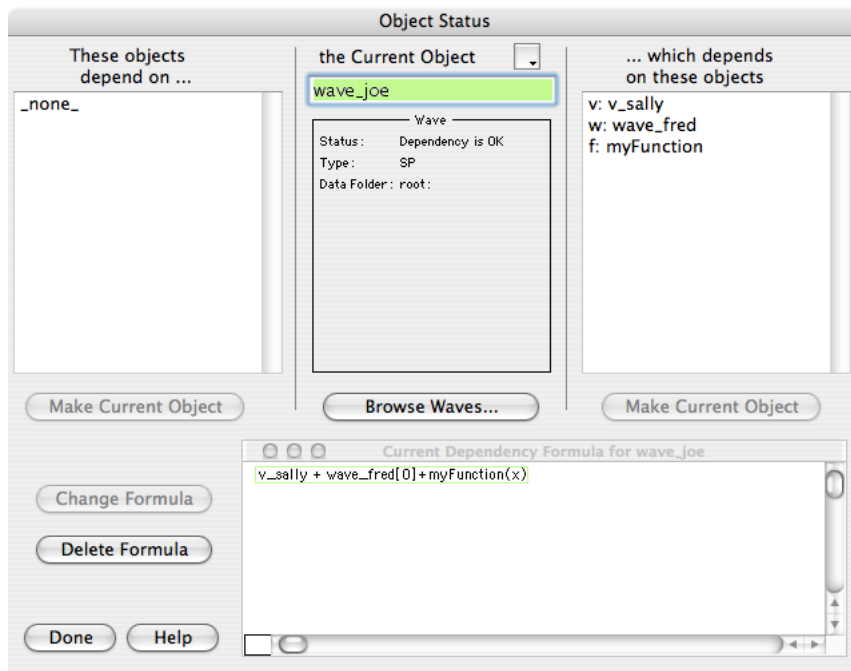


In this example, the current object is the global numeric variable `v_sally`. The only object directly dependent on `v_sally` is the wave `wave_joe`. `v_sally` directly depends on waves `wave_fred` and `wave_sue`, global numeric variable `K2`, and user-defined function `anotherFunction`. These dependencies exist because of the dependency assignment:

```
v_sally := anotherFunction(wave_fred[1]) + wave_sue[0] + K2
```

which can be changed or deleted with the dialog.

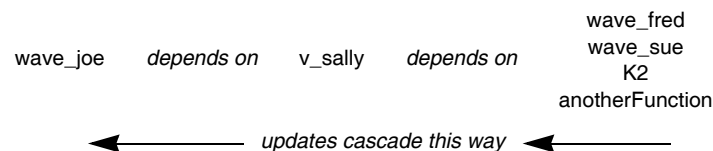
`Wave_joe` depends on `v_sally` for reasons that will become apparent only when `wave_joe` is made the current object:



The dependency was created by the dependency assignment:

```
wave_joe := v_sally + wave_fred[0] + myFunction(x)
```

Combining the dependency of wave_joe on v_sally with what v_sally depends on, you can see that wave_joe also *indirectly* depends on wave_sue, wave_fred, K2, and anotherFunction:



If you change K2, the dependencies will cascade so that v_sally and then wave_joe are updated. We call objects that wave_joe depends on directly or indirectly its “antecedents” (literally, “those that go before”).

Deleting a Dependency

A dependency is deleted when the dependent object is assigned a value using the = operator:

```
recalculateThis := dependsOnThis // creates dependency
recalculateThis = 0 // deletes the dependency
```

This method of deleting a dependency does not work in user-defined functions. You must use the SetFormula operation.

For example:

```
Execute "recalculateThis = 0"
```

will delete the dependency even in a user-defined function.

You can also delete this dependency using the SetFormula operation.

```
SetFormula recalculateThis, ""
```

Wave dependencies are also deleted by operations that overwrite the values of their wave parameters. Some of these operations are:

Chapter IV-9 — Dependencies

FFT Convolve Correlate Smooth GraphWaveEdit
Hanning Differentiate Integrate UnWrap

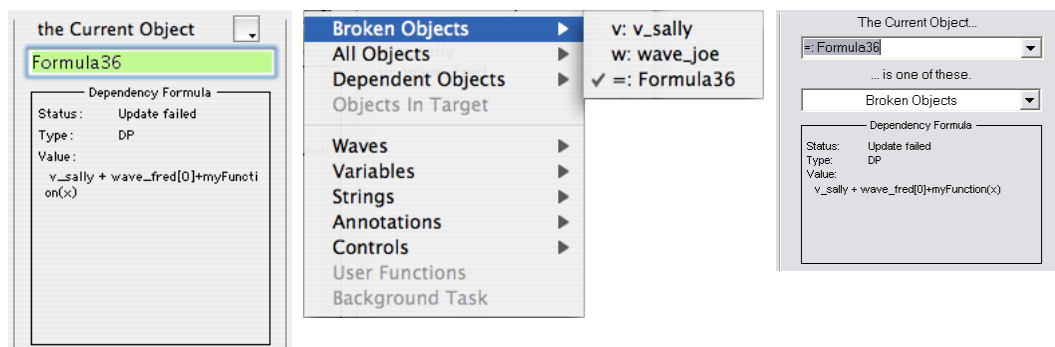
Dependencies can also be deleted via the Object Status dialog in the Misc menu.

Broken Dependent Objects

Igor compiles the text of a dependency formula to low-level code and stores both the original text and the low-level code with the dependent object. At various times, Igor may need to recompile the dependency formula text. At that time it is possible to get a compilation error if one of the objects in the formula has been renamed or deleted, or if the formula contains a syntax error.

When this happens, the dependent object will no longer update but will retain its last value. We call such an object “broken”. If you suspect this kind of problem has happened, invoke the Object Status dialog using the Misc menu.

Any such broken objects will show up in the Broken Objects submenu of the Current Object pop-up menu:



When Dependencies are Updated

Dependency updates take place at the same time that graphs are updated. This happens after each line in a macro is executed, or when DoUpdate is called from a macro or user function, or continuously if a macro or function is not running.

Dependency formulas used as input to the SetBackground and ValDisplay operations, and in some other contexts, can alternately be specified as a literal string of characters using the following syntax:

```
#"text_of_the_dependency_expression"
```

Note that what follows the # char must be a literal string — not a string expression.

This will set the dependency formula *without* compiling it or checking it for validity. It is mainly for use internally but if you find yourself in a situation where you need to set the dependency formula of an object to something that is not currently valid but will be in the future then feel free to use this alternate method.

Programming with Dependencies

You cannot use := to create dependencies in user-defined functions. Instead you must use the **SetFormula** operation (see page V-538).

```
Function TestFunc()  
    Variable/G varNum=-666  
    Make wavel  
    SetFormula wavel, "varNum"      // Equivalent to wavel := varNum  
End
```

Using Operations in Dependency Formulas

The dependency formula must be a single expression — and you can not use operations, such as FFT's, or other command type operations. However, you can invoke user-defined functions which *in turn* invoke operations:

```
Function MakeDependencyUsingOperation()
  Make/O/N=128 data = p          // A ramp from 0 to 127
  Variable/G power

  SetFormula power, "RMS(data)" // Dependency on function and wave
  Print power

  data = p * 2                  // Changes something power depends
  DoUpdate                      // Make Igor recalc formulae
  Print power
EndMacro

Function RMS(w)
  Wave w

  WaveStats/Q w                // An operation! One output is V_rms
  return V_rms
End
```

When `MakeDependencyUsingOperation` is executed, it prints the following in the history area:

```
•MakeDependencyUsingOperation()
  73.4677
  146.935
```

Dependency Caveats

The extensive use of dependencies can create a confusing tangle that can be difficult to manage. Although you can use the Object Status dialog to explore the dependency hierarchy, you can still become very confused very quickly, especially when the dependencies are highly cascaded. You should use dependencies only where they are needed. Use conventional assignments for the majority of your calculations.

There is no built-in limit to the depth of dependency cascading except that speed considerations will limit the depth to about 20. Similarly there will be a practical limit to the total number of objects with dependencies. The actual limit can not be specified in advance.

Dependency formulas are generally not recalculated when a user-defined function is running unless you explicitly call the `DoUpdate` operation. However, they can run at other hard to predict times (especially on Windows) and you should not make any assumptions as to the timing or the current data folder when they run.

The text of the dependency formula that is saved for a dependent object is the original literal text. The dependency formula needs to be recompiled from time to time, for example when procedures are compiled. Therefore, any objects used in the formula must persist until the formula is deleted.

We recommend that you never use \$ expressions in a dependency formula.

