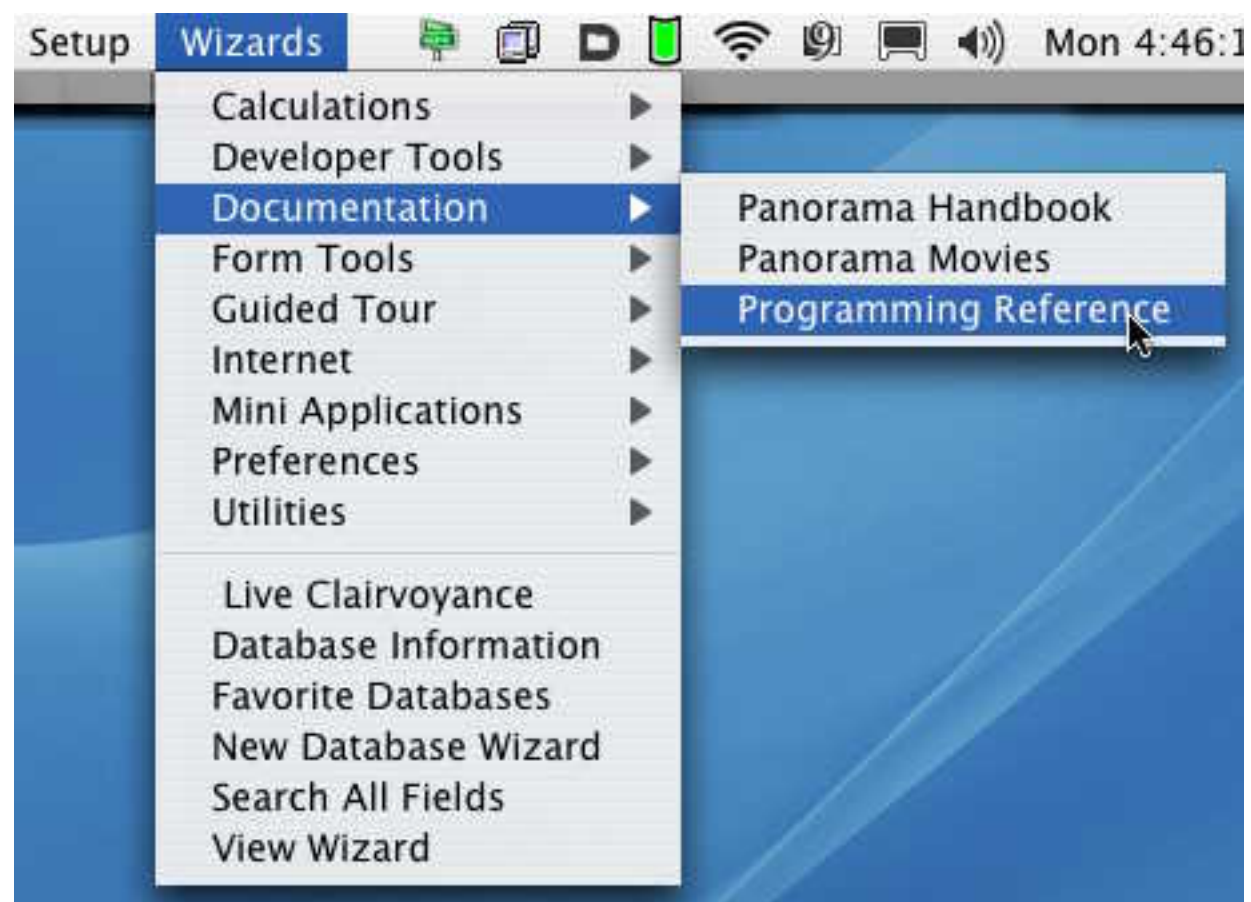


Panorama Reference

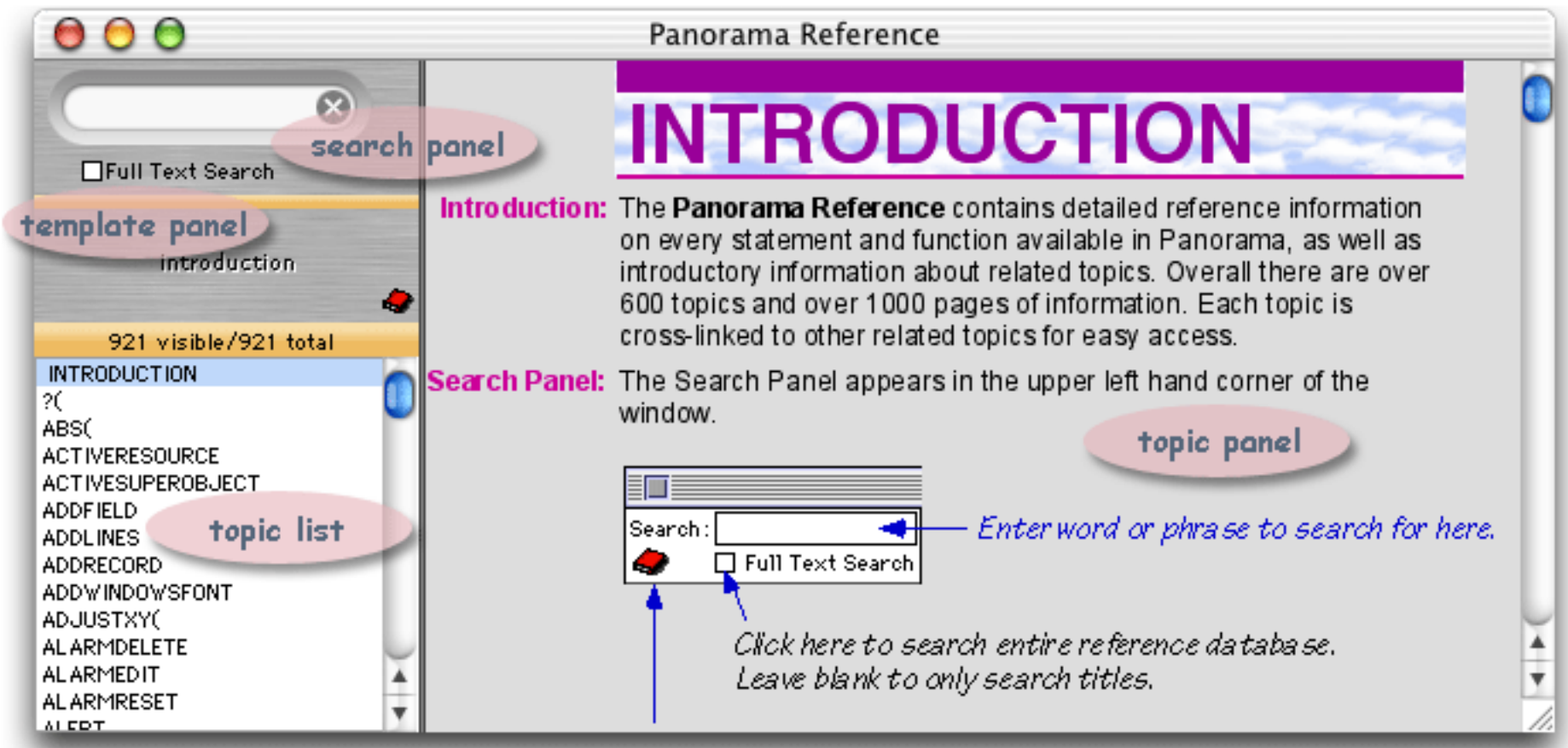


Introduction: This **Panorama Reference** supplement contains detailed reference information for every statement and function available in Panorama as of Panorama 4.0.2. It has not, however, been updated for Panorama V or later versions, and ProVUE no longer intends to keep this reference updated as new features are added. (It is still retained, however, because many sections of the Panorama Handbook link to topics in this reference.) For the most up-to-date and complete reference available we refer you to the **Programming Reference** wizard. This wizard is in the **Developer Tools** subfolder of the Wizard menu, and also can be accessed at any time by pressing **Control-R** on MacOS systems.

Online Reference: To open the online reference select **Programming Reference** from the Documentation submenu of the Wizard menu.

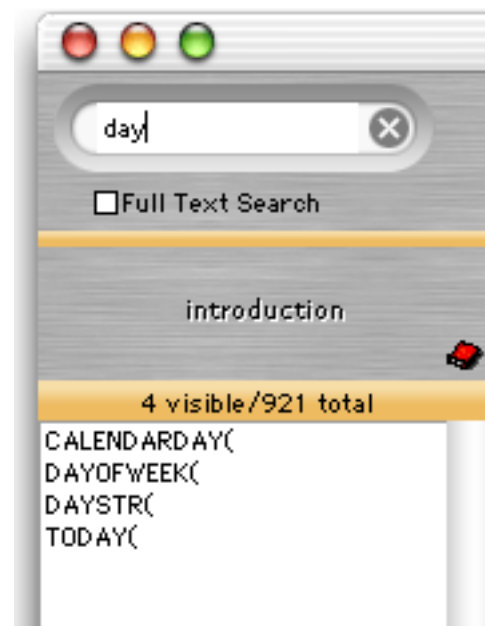


Once the wizard opens the reference window is divided into four sections: search panel, template panel, topic list and topic panel.

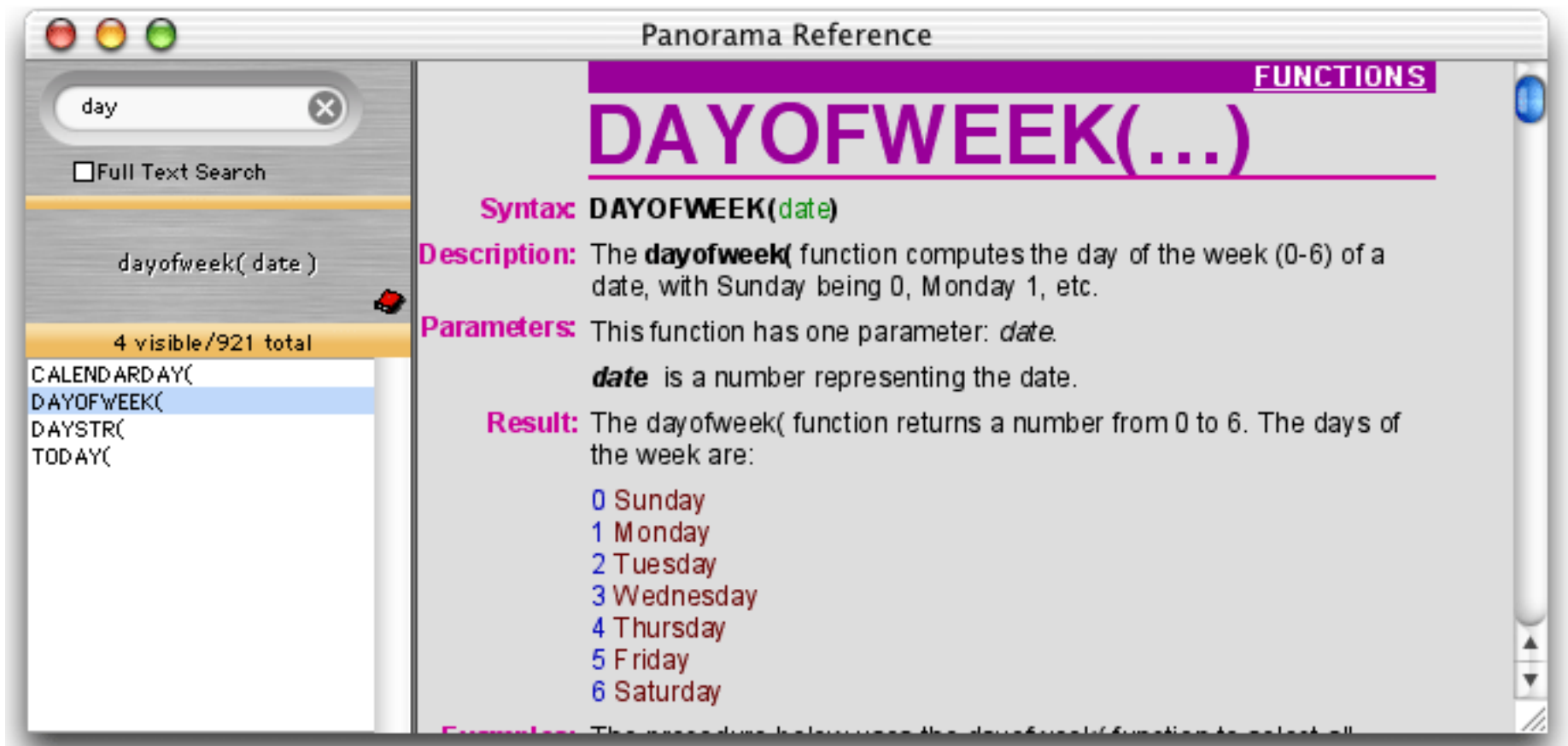


Navigation Using the Search Panel and Topic List

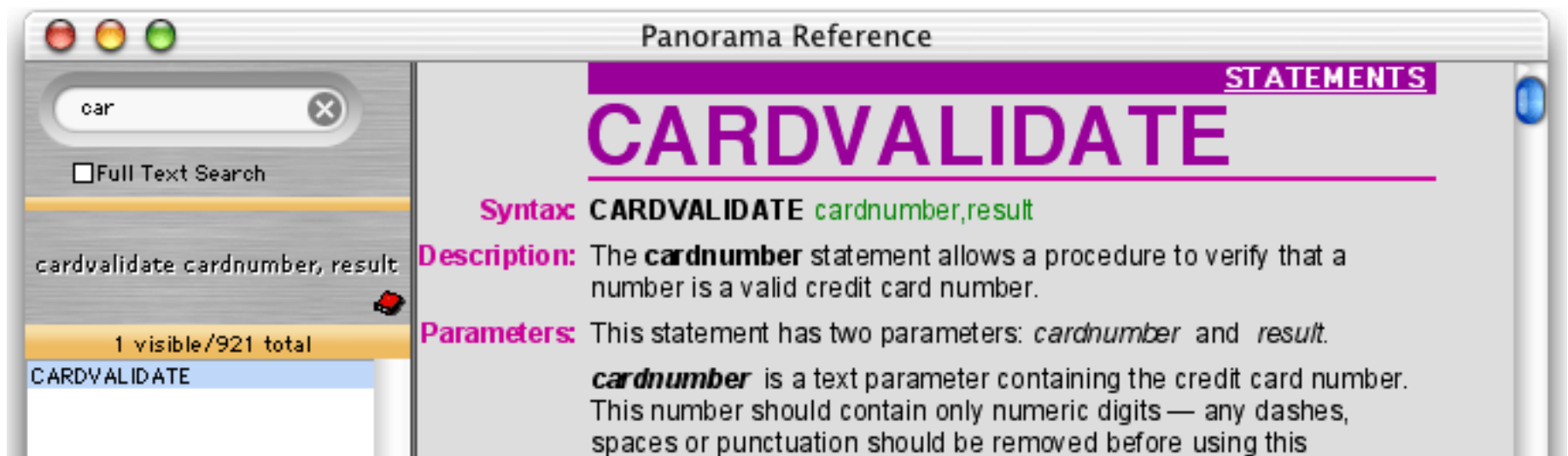
The search panel and topic list work together to help you locate a specific topic. As you type into the search panel, the topic list updates to show topics that match.



When you see the topic you want, click on it to display the topic in the topic panel.



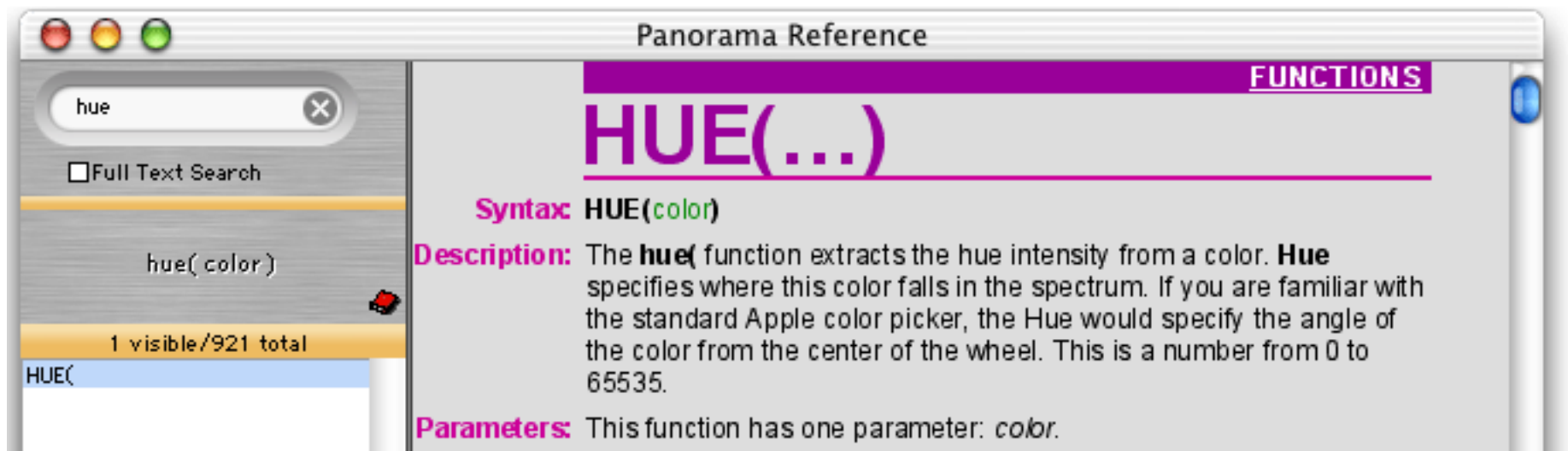
If there is only one topic in the topic list, the wizard will display the topic automatically (without having to click on the list).



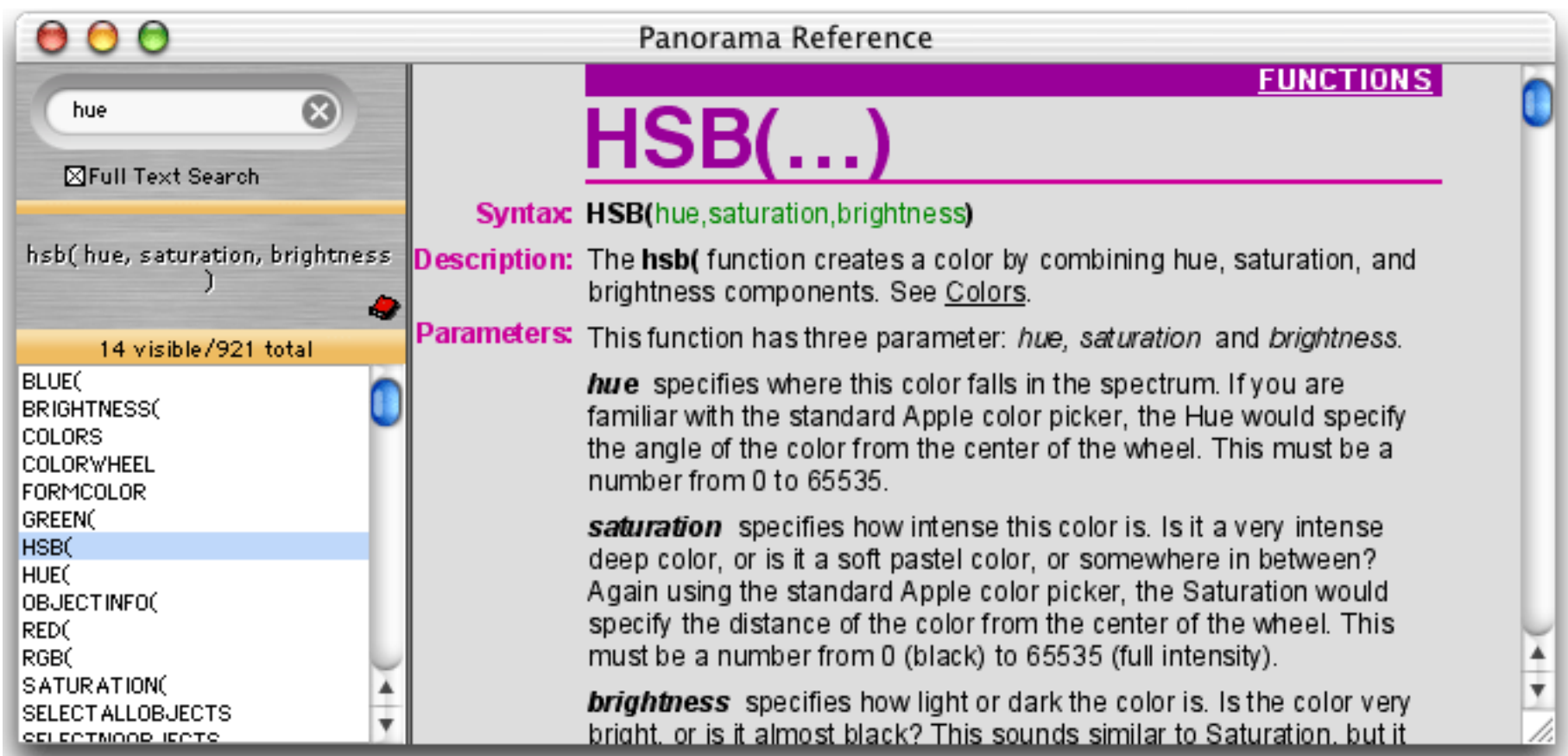
To quickly erase the query in the search panel, click on the  button.

The Full Text Search Option

The search panel normally searches only the name and category of each topic. When the **Full Text Search** option is checked the wizard will also search the complete text of each topic. This makes it possible to quickly find every topic that references a particular function or statement, as well as the function or statement itself. For example, a normal search for the word **hue** will turn up only one match.



Repeating the search with the **Full Text Search** option turned on yields 14 matches. You can click on the match you are interested in.



Navigation Using the Topic, Statement and Function Menus

To jump directly to any topic use the **Topic**, **Statement** or **Function** menus. The **Topic** menu divides topics into about two dozen submenus. (Some topics may be display under more than one submenu, and some topics may not be listed under any topics.)



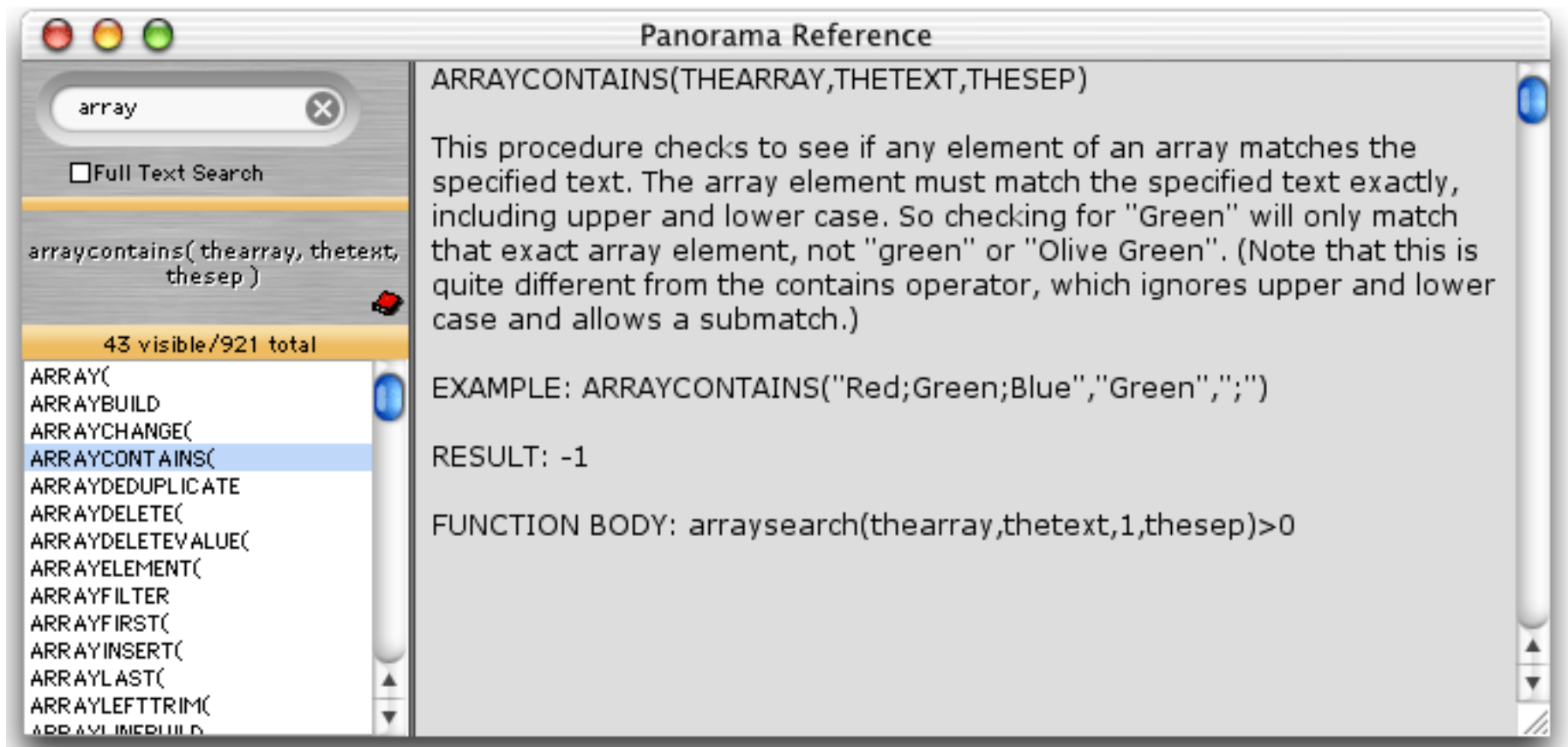
The **Statements** menu lists every statement in alphabetical order. The **Functions** menu lists every function in alphabetical order. Simply select a statement or function from one of these menus to jump to see the description of that topic in the topic panel.

Navigation Using HyperLinks

Like a web browser, the **Programming Reference** contains links from one page to other related topics. These links are underlined in the text. To jump any linked topic simply click on the underlined text.

Built In vs. Custom Statements and Functions

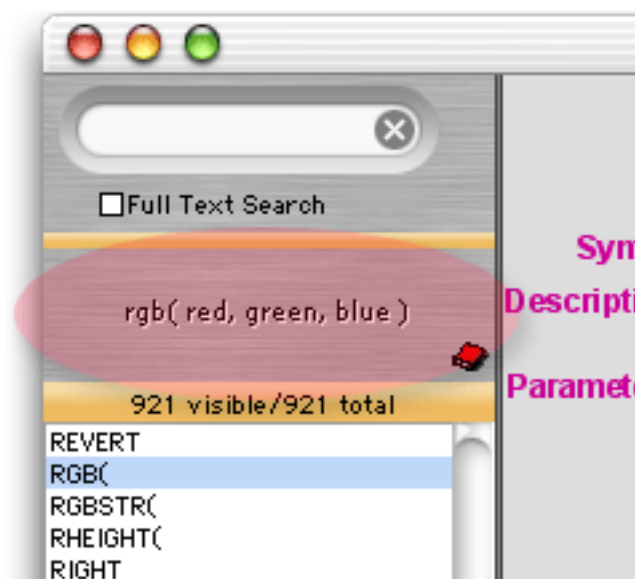
Panorama supports both built in and custom statements and functions. Several hundred custom statements and functions are included with Panorama, and these are also included as topics in the **Programming Reference** wizard. (You can also create your own custom statements and functions, but these are not included in the **Programming Reference** wizard.) For most custom statements and functions the topic panel uses a basic "plain text" format instead of the more graphical format used for built-in statements and functions.



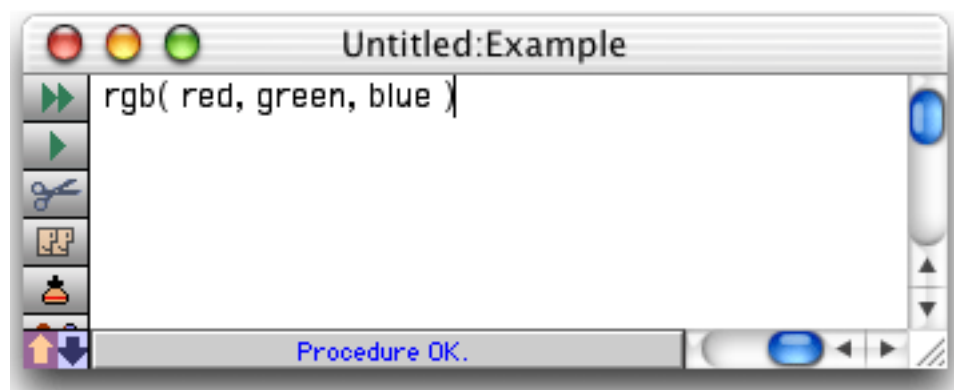
Don't adjust your set -- this plain text view is normal for custom statements and functions.

Using the Template Panel

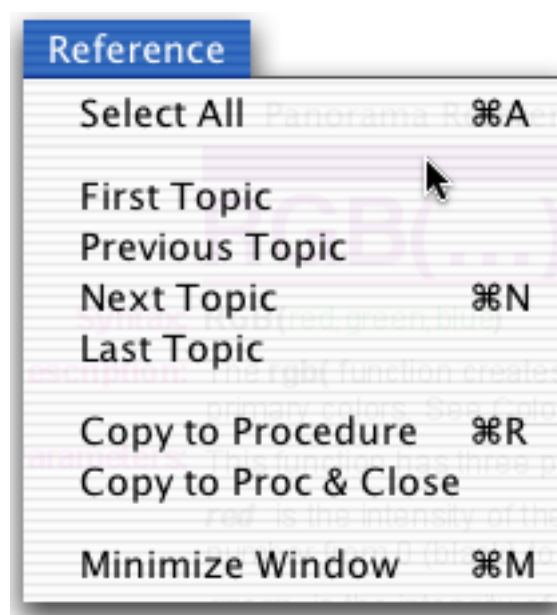
The template panel displays a sample that illustrates how this statement or function would be used in a formula or procedure. In this case the panel shows an example of the `rgb()` function, which has three parameters.



To copy the template into the topmost procedure window, hold down the **Control** key and click on the template panel. (If you are using a PC system you should right-click on the template panel.) The template will be pasted into the procedure at the current insertion point, and the procedure window will be brought to the front so that you can edit it further (for example, filling in the actual parameters).




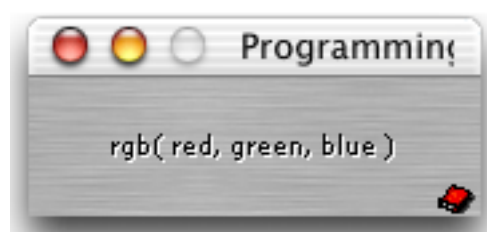
You can also copy the template into the procedure window using the **Reference** menu.




The **Copy to Procedure** command copies the template into the procedure and brings the procedure window forward (exactly like control-clicking on the template panel). The **Copy to Proc & Close** command does the same, but also closes the **Programming Reference** wizard.

Minimizing the Programming Reference Wizard

In addition to its normal "wide-screen" view, the wizard can also be used in a minimized view. To minimize the wizard, either choose **Minimize Window** from the Reference menu or click on the  button in the template panel. As shown here the minimized wizard hides the search panel, topic list, and topic display panel.



Although you can't search for topics when the window is minimized, you can still use the **Reference**, **Topics**, **Statements** and **Function** menus. When you want to maximize the window again choose either the **Maximize Window** from the Reference menu or click on the  button.

General Topics:

The following topics contain background information about various aspects of Panorama programming

[ascii](#)

[binary data](#)

[c/pascal structures](#)

[colors](#)

[date patterns](#)

[functions](#)

[graphic coordinates](#)

[non decimal numbers](#)

[numeric patterns](#)

[reminder data](#)

[statements](#)

[text arrays](#)

?(...)

Syntax: ?(truefalse,trueTormula,falseFormula)

Description: The ?(function allows a formula to make an either-or decision. (If X is true, then Y, else Z.)

Parameters: This function has three parameters: truefalse, trueFormula and falseFormula.

truefalse is a “mini-formula” that controls the decision of the ? function...will it be door number 1 or door number 2? This mini-formula must result in a true or false answer. Here are some typical “mini-formulas” that produce a true or false answer.

```
Age>18
Middle=" "
Country="USA"
Attendance
```

trueFormula is a “mini-formula” that will provide the result of the ?(function if the truefalse parameter turns out to be true.

falseFormula is a “mini-formula” that will provide the result of the ?(function if the truefalse parameter turns out to be false.

Result: This function returns the value of either the trueFormula or the falseFormula, depending on whether truefalse is true or false. The trueFormula and the falseFormula may calculate numbers or text, but usually both should calculate the same type of data.

Examples: The first example assumes that a database has three fields containing a person’s name: `First`, `Middle`, and `Last`. The formula below combines these three names into one, but if the middle name is empty the formula makes sure there is only one space between the first and last names.

```
Name=First+" "+?(Middle≠" ",Middle+" ",")+Last
```

The three parameters to the ? function are:

```
Middle ≠" "
Middle+" "
" "
```

The first parameter, `Middle ≠" "`, checks to see if this person has a middle name or not. If they do, the ?(function will use the trueFormula, `Middle+" "`, which includes the middle name followed by a space. If the person does not have a middle name the ?(function will use the falseFormula, which is simply an empty text item.

The next example is a formula that could be used in an auto-wrap text object or Text Display SuperObject™ to display real estate information. If a house has not sold yet, this formula will display something like this: 5492 Miramar (asking \$186,000). If the house has already sold it will display like this: 321 Olive (sold 3/14/95). The formula below uses the ?(function to pick which format use, depending on whether or not the SoldOn field is empty or not.

```
Address+" (" +  
?(sizeof(SoldOn)=0,  
"asking "+pattern(AskingPrice,"$#,")+")",  
"sold "+datepattern(SoldOn,"mm/dd/yy")+")")
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the truefalse parameter.

See Also: [if](#) statement

ABS(...)

Syntax: ABS(value)

Description: The abs(function returns the absolute (positive) value of the numeric parameter. In other words, negative numbers are converted to positive numbers, while positive numbers stay positive.

Parameters: This function has one parameters: value.

value is the value you want to convert to a positive number. You may use any numeric value, for example 1, 1000, 2.5, or -500.

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This simple example calculates the difference between two prices. The result will always be positive, even if Price2 is greater than Price1.

```
abs(Price1-Price2)
```

Price1 and Price2 must contain numeric values.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `abs("34")`. If you have a number in a text item you must convert the text to a numeric value before taking the absolute value, for example `abs(val("34"))`.

See Also: [val\(function](#)

ACTIVERESOURCE

Syntax: ACTIVERESOURCE file

Description: The `activeresource` statement specifies that a particular resource file should be the primary resource file for reading and writing. This gives you greater control when working with multiple resource files.

Parameters: This statement has one parameter: file.

file identifies the resource file you want to make the primary resource file. This file must have previously been opened with the `openresource` or `openresourcerw` statements. If the parameter is "" the original priority of the files is restored.

Action: This statement makes a resource file the primary resource file for reading and writing. It does not disable other resource files, however, Panorama will look in this primary file first when reading and will always write into the primary file.

Examples: For these examples we will assume that three resource files have been opened, like this.

```
openresourcerw "alpha"
openresourcerw "beta"
openresourcerw "gamma"
```

The program below will write a resource into the `beta` resource file instead of the `gamma` file.

```
activeresource "beta"
writeresource "DATA",2000,"This is not a test"
activeresource ""
```

The program below will only read the `DATA 2000` resource from the `beta` file, not `alpha` or `gamma`.

```
activeresource "beta"
temp=getresource("DATA",2000)
activeresource ""
```

Views: This statement may be used in any view

See Also: [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[writeresource](#) statement
[getresource\(\)](#) function
[getString\(\)](#) function
[getnstring\(\)](#) function
[getStringmatch\(\)](#) function
[resources\(\)](#) function
[resourcetypes\(\)](#) function

ACTIVESUPEROBJECT

SYNTAX: ACTIVESUPEROBJECT command,param1,param2,...paramN

Descriptive: The `activesuperobject` statement allows a procedure to communicate with a SuperObject on the current form. This statement is similar to the `superobject` statement, but instead of specifying a named object this statement sends a command to the SuperObject that is currently being edited (if any).

For information on using this command with specific types of SuperObjects, see:

[text editor programming](#)
[word processor programming](#)
[super flash art programming](#)
[list superobject programming](#)
[super matrix programming](#)
[scroll bar programming](#)

Parameters: This statement has a variable number of parameters, but always at least one: `command`.

`command` is an instruction that will be sent to a specific SuperObject. Different types of SuperObjects understand different types of commands. For example the Text Editor SuperObject understands commands like "InsertText" and "GetSelection", while the List SuperObject understands commands like "AddCell" and "FindCell". (Note: The Active-SuperObject statement can only be used with Text Editor and Word Processor types of objects, since these are the only types of SuperObjects that can become active. For other types of SuperObjects you must use the `superobject` statement.)

`param1...paramNN` are additional parameters used by individual commands, if required. For example the Text Editor's "InsertText" command requires one additional parameter which specifies the text to be inserted. The List SuperObject's "DeleteCell" command requires two additional parameters, the numbers of the first and last cells to be deleted. Parameters may also receive values from the SuperObject. For example the List SuperObject's "GetCount" command has one parameter into which the number of items in the list is stored. If a parameter is used to receive a value from the SuperObject, that parameter must be a single field or variable with no operators (`myValue`, not `myValue+yourValue` or `strip(myValue)`).

Action: This statement allows a procedure to communicate with a SuperObject on the current form. This statement is similar to the SuperObject statement, but instead of specifying a named object this statement sends a command to the SuperObject that is currently being edited (if any). Many SuperObjects have one or more commands that they understand. For example, the Text Editor SuperObject has commands for selecting text, locating text, modifying text, etc.

Examples: This example insert the current date and time into whatever SuperObject is currently being edited. This procedure will work with both Text Editor and Word Processor SuperObjects. The first line of the procedure checks to make sure that there actually is an active SuperObject (i.e. something is really being edited at this time).

```
if info("activesuperobject")≠"  
activesuperobject "InsertText",  
    datepattern( today(), "mm/dd/yy")+ "@"+  
    timepattern( now(), "hh:mm am/pm")  
endif
```

Views: This statement may be used in Form views.

See Also: [superobject](#) statement
[info\("activesuperobject"\)](#) statement
[text editor programming](#)
[word processor programming](#)
[super flash art programming](#)
[list superobject programming](#)
[super matrix programming](#)
[scroll bar programming](#)

ADDFIELD

Syntax: `ADDFIELD name`

Description: The `addfield` statement adds a new field to the current database.

Parameters: This statement has one parameter: `name`.

`name` is the name of the new field you want to create. If the parameter is the keyword `dialog` the procedure will stop and display the Field Properties dialog, allowing the user to set up the field name, type, and other properties of the new field.

Action: This statement adds a new field to the database at the end of all the current fields. (If you want to insert a new field into the middle of the current fields, use the [insertfield](#) statement. The new field is created as a text field. Use the [fieldtype](#) statement if you need to convert the new field to a numeric, date, choice or picture field. (Note: the new field does not become the current field. Use the [field](#) statement to make the new field current before changing the type or performing other operations on the field.)

Examples: This simple example adds a new field called Notes at the end of the current database.

```
addfield "Notes"
```

This example creates a new field called Ratio, converts it to floating point, and then calculates values for the new field.

```
addfield "Ratio"
field "Ratio"
fieldtype "float"
formulafill "Price/Cost"
```

The example below creates a new field, allowing the user to set up the field name and properties. After the new field is created the procedure determines the name of the new field and loads it into the variable `newFieldName`.

```
local newFieldName
addfield dialog
newFieldName=
array( dbinfo("fields",""),
array( dbinfo("fields",""),1),1)
```

Views: This statement may be used in the Data Sheet, Cross Tab view, and Form views **only**.

See Also: [dbinfo\(\)](#) function
[deletefield](#) statement
[field](#) statement
[fieldname](#) statement
[fieldstyle\(\)](#) function
[fieldtype](#) statement
[info\("datatype"\)](#) function

info("fieldname") function
insertfield statement
newgeneration statement

ADDLINES

Syntax: `ADDLINES field,start,end,bump,copies`

Description: The `addlines` statement allows Panorama to add a specified number of records to the end of a database. This statement will be faster than setting up a [loop](#) and using the `addrecord` statement.

Parameters: This statement has five required parameters: `field`, `start`, `end`, `bump`, and `copies`.

field is the name of the field you wish to increment as the new records are added to the database. This parameter must either be a 0-digit numeric or date type field or it may be a variable if you don't want to store the incrementing values in the database.

start is the beginning sequence number or date value. `start` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. `start` must be less than and not equal to `end`

end is the ending sequence number or date value. `end` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. `end` must be greater than and not equal to `start`.

bump is the increment value for your sequence. `bump` may be a number, a numeric variable, or a formula which results in a positive numeric integer. The `bump` value must be a positive integer for a numeric field. For a date field `bump` may also be one of the following:

"M" - for month (the M must be in quotes)

"Y" - for year (the Y must be in quotes)

1 - for day

7 - for week

copies is the number of records generated for each sequence number. `copies` may be a number, a numeric variable, or a formula which results in a positive numeric integer (in most cases this value will be 1).

Action: This statement will allow a procedure to add a predetermined number of records to the end of any database. The new records may be sequenced with incrementing integer or date values in a specified field.

Examples: This example shows how a procedure can add 100 records to a database and sequence those new records from 1200 to 1299 in a numeric, 0-digit field called `CheckNum`.

```
addlines CheckNum,1200,1299,1,1
```

This example shows how a procedure can add 31 records to a database and sequence those new records from October 1 to October 31 in a date field called `EntryDate` (Panorama will assume the year based on the computers system date). Also notice that a function was used to determine the values for the `start` and `end` parameters

```
addlines EntryDate,date("oct 1"),date("oct 31"),1,1
```

This procedure will add one record for each month to a database and sequence those new records from a start date to an end date that the user specifies. The date field sequenced is called Date. This time the bump value is "M", indicating that the value should be incremented by one month for each record.

```

local Start,End
gettext "Enter Start date, ex: 1/1/95",Start
Start = date(Start) /* converts to a date value */
gettext "Enter End date, ex: 12/1/95",End
End = date(End) /* convert to a date value */
addlines Date,Start,End,"M",1

```

This example adds 20 records to a database and sequences those new records from 1 to 10 in a field called In/Out, but makes two records for each sequence value by setting copies to 2. Notice also that the end value is a simple formula 1+9. The resulting sequence will be 11223344 ...

```

addlines «In/Out»,1,1+9,1,2

```

This example adds 10 records to a database, however no field will be sequenced because a variable is used as the first parameter.

```

local Useless
Useless = ""
addlines Useless,1,1+9,1,1

```

Views:

This statement may be used in a procedure which runs from the Data Sheet or Form views. This statement cannot be used to add records to the Design Sheet or a Cross Tab.

See Also:

[addrecord](#) statement
[cutrecord](#) statement
[deleterecord](#) statement
[printonemultiple](#) statement

ADDRECORD

Syntax: ADDRECORD

Description: The **addrecord** statement adds a new record at the end of the current database.

Parameters: This statement has no parameters.

Action: This statement adds a new record to the end of the database. It has the same effect as choosing the **Add New Record** command in the **Edit** menu.

Examples: This simple example adds twelve new records at the end of the current database.

```
loop  
    addrecord  
until 12
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views.

See Also: [insertrecord](#) statement
[insertbelow](#) statement
[returnkey](#) statement
[deleterecord](#) statement
[info\("records"\)](#) function

ADDWINDOWSFONT

Syntax: ADDWINDOWSFONT font

Description: The `addwindowsfont` statement notifies Windows that you have installed a new font. This is a very specialized statement used by the Panorama installer. It has no effect on MacOS computers..

Parameters: This statement has one parameter: font.
font is the name of the font you are installing.

Action: This statement will notify Windows that a new font has installed. Before you use this statement you should copy the font into the Fonts folder. After you use this statement you should set up the registry entries for the font.

Examples: This example installs the font true-type font San Diego. The example assumes that the file has already been copied into the Fonts folder (perhaps with the [filesave](#) statement).

```
local fontRegistryFolder
fontRegistryFolder=
    "HKLM\Software\Microsoft\Windows\CurrentVersion\Fonts:"
AddWindowsFont "San Diego.ttf"
RegistryWrite fontRegistryFolder+
    array(File,1,".")+" (TrueType)", "", "San Diego.ttf"
```

For installation on a Windows NT system the second line must be slightly modified.

```
fontRegistryFolder=
    "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Fonts:"
```

Views: This statement may be used in any view.

See Also: [registrywrite](#) statement

ADJUSTXY(...)

- Syntax:** ADJUSTXY(rectangle,boundary,deltav,deltaH)
- Description:** The `adjustxy()` function adjusts the four corners of a rectangle. However, only corners that are inside a boundary are adjusted. Corners outside the boundary are left alone.
- Parameters:** This function has four parameters: `rectangle`, `boundary`, `deltav` and `deltaH`.
- rectangle** is the rectangle that is being adjusted
- boundary** is a rectangle describing the area to be adjusted. Only points inside this rectangle will be adjusted.
- deltav** is the vertical distance each corner inside the boundary should be adjusted.
- deltaH** is the horizontal distance each corner inside the boundary should be adjusted.
- Result:** This function returns a rectangle. A rectangle is an 8 byte binary data item. Like all other binary data items, rectangles are actually stored as text (see [binary data](#))
- Examples:** The procedure below uses `adjustxy` to help move a slider on a form. This procedure is designed to be triggered by a button with the Click/Release option turned off.

```

local drag,dragstart,deltaV,deltaH,slider,slidebox
drag=info("buttonrectangle")
dragstart=drag
slider=xytoxy(drag,"s","f")
slider=rectangleadjust(slider,0,0,1,1)
selectobjects
intersectionrectangle(
    xytoxy(drag,"s","f"),
    objectinfo("rectangle") ≠ rectangle(0,0,0,0)
slidebox=xytoxy(objectinfo("boundary"),"f","s")
slidebox=rectangleadjust(insidewindow,0,16,0,-16)
draggraybox drag,slidebox,info("windowrectangle"),1
if drag="" stop endif
deltaV=rtop(drag)-rtop(dragstart)
deltaH=rleft(drag)-rleft(dragstart)
changeobjects "rectangle",
adjustxy(objectinfo("rectangle"),slider,deltaV,deltaH)

```

- Errors:**
- Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the rectangle or boundary.
- Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the delta.

- See Also:**
- [point\(\)](#) function
 - [rectangle\(\)](#) function
 - [rtop\(\)](#) function
 - [rbottom\(\)](#) function
 - [rleft\(\)](#) function
 - [rright\(\)](#) function
 - [rheight\(\)](#) function
 - [rwidth\(\)](#) function
 - [inrectangle\(\)](#) function
 - [unionrectangle\(\)](#) function

intersectionrectangle(function
rectangleadjust(function
rectanglecenter(function
info("screenrectangle") function
info("windowrectangle") function
info("buttonrectangle") function
info("cursorrectangle") function

ALARMDELETE

Syntax: `ALARMDELETE reminder`

Description: The `alarmdelete` statement tells the Team Alarm extension to delete an alarm.

Parameters: This statement has one parameter: `reminder`.

`reminder` is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. see [reminder data](#) for detailed information about reminders.

Action: If you have the optional Team Alarm extension installed, you can be notified of your reminders even when Panorama is not currently running. To do this, the Team Alarm extension keeps a separate private list of pending alarms. The `alarmdelete` statement tells the Team Alarm extension to delete a reminder from the this private list of alarms.

Examples: If a database contains alarms you should include the following statements in the `.DeleteRecord` procedure. This example assumes that the reminders are stored in a field called `Reminder`.

```
AlarmDelete Reminder
deleterecord
```

This example deletes all unselected records from a calendar database.

```
selectreverse
loop
alarmdelete reminder
downrecord
until info("stopped")
selectreverse
removeunselected
```

Here is another way to do the same job. If you are deleting a lot of records this method will probably be faster.

```
removeunselected
alarmreset Reminder,Message
```

Views: This statement may be used in any view.

See Also: [alarmedit](#) statement
[alarmreset](#) statement
[reminder data](#)

ALARMEDIT

- Syntax:** `ALARMEDIT reminder,message`
- Description:** The `alarmedit` statement tells the Team Alarm extension to change the message associated with an alarm.
- Parameters:** This statement has two parameters: `reminder` and `message`.
- reminder** is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. see [reminder data](#) for detailed information about reminders.
- message** is the message that goes with the alarm, for example Sue's flight arrives or Pick up kids.
- Action:** If you have the optional Team Alarm extension installed, you can be notified of your reminders even when Panorama is not currently running. To do this, the Team Alarm extension keeps a separate private list of pending alarms. The `alarmedit` statement changes the message associated with an alarm. Use this statement if the procedure has changed the message. (If the message has been changed by the user as a result of the [reminder](#) dialog, the `alarmedit` statement is not necessary.)
- Examples:** This example finds all reminders for Big Shot and adds the words **TOP PRIORITY:** to the start of the message. The example assumes that the reminder data is stored in a field called `Reminders`.
- ```

find Customer = "Big Shot"
loop
 stoploopif (not info("found"))
 Message="TOP PRIORITY: "+Message
 AlarmEdit Reminders,Message
next
while forever

```
- Here is another way to do the same job. If you are changing a lot of records this method will probably be faster.
- ```

field Message
formulafill
?(Customer="Big Shot","TOP PRIORITY: ","")+Message
alarmreset Reminders,Message

```
- Views:** This statement may be used in any view.
- See Also:** [alarmdelete](#) statement
[alarmreset](#) statement
[reminder data](#)

ALARMRESET

- Syntax:** `ALARMRESET reminder,message`
- Description:** The `alarmreset` statement tells the Team Alarm extension to rebuild its private list of alarms.
- Parameters:** This statement has two parameters: `reminder` and `message`.
- reminder** is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. See [reminder data](#) for detailed information about reminders. In this case you are not specifying the reminder itself, but the field containing the reminder data.
- message** is the message that goes with the alarm, for example Sue's flight arrives or Pick up kids. In this case you are not specifying the message itself, but the field containing the messages.
- Action:** If you have the optional Team Alarm extension installed, you can be notified of your reminders even when Panorama is not currently running. To do this, the Team Alarm extension keeps a separate private list of pending alarms. The `alarmreset` statement tells the extension to rebuild this list of alarms from the current database. Use this statement when you first create the reminder database, or if you think that Team Alarms's list may have gotten out of sync with the Panorama database. For example, this may happen if the user quits without saving changes (the changes have been made to Team Alarm's database), or when new data is imported into the database.
- Examples:** This example imports some new reminders into a calendar database, then makes sure that the Team Alarm list of alarms is kept up-to-date. The example assumes that the reminder data is stored in a field called [Reminders](#).
- ```

openfile "+Sandy's Appointments"
alarmreset Reminders,Message

```
- This example deletes all reminders more than 12 months old. The reminders are deleted from both the Panorama database and Team Alarm's private list of alarms. (Actually this is not technically necessary in this case, since the Team Alarm extension automatically deletes an alarm after its time has passed.)
- ```

selectreminder_data(Reminder)>monthmath( today(,-12)
removeunselected
alarmreset Reminders,Message

```
- This example changes the message for all of Big Shot's reminders.
- ```

field Message
formulafill
?(Customer="Big Shot","TOP PRIORITY: ","")+Message
alarmreset Reminders,Message

```
- Views:** This statement may be used in any view.

**See Also:**      [alarmdelete](#) statement  
                  [alarmedit](#) statement  
                  [reminder](#) data

# ALERT

**Syntax:** `ALERT resource#,message`

**Description:** The **alert** statement allows a Panorama procedure to open an alert dialog and display a specified message within that dialog.

**Parameters:** This statement has two required parameters: **resource#** and **message**.

**resource#** is the number that identifies an alert resource. The resource can either be constructed using a program like ResEdit or you can specify a resource number for an internal Panorama resource (resource numbers below 5000 are reserved for the Panorama program.)

This is a list of alert dialogs in Panorama that you may find useful in your procedures.

| Resource # | Buttons                         | Notes                 |
|------------|---------------------------------|-----------------------|
| 1000       | OK                              |                       |
| 1001       | OK, Cancel                      | 1st Button is default |
| 1002       | Cancel, OK                      |                       |
| 1003       | Save, Don't Save, Cancel        | message=filename      |
| 1005       | OK                              | Small version of 1000 |
| 1008       | Wait, Cancel                    |                       |
| 1009       | Cancel, Revert                  |                       |
| 1010       | Delete, Cancel                  |                       |
| 1012       | Re-Edit, Cancel                 |                       |
| 1013       | Yes, No                         |                       |
| 1014       | No, Yes                         |                       |
| 1015       | Cancel, Delete                  |                       |
| 1018       | Yes, No, Cancel                 |                       |
| 1101       | Cancel, OK, Select Problem Data |                       |

**Warning:** Specifying an undefined or unopened resource in an alert statement will cause Panorama to crash to the Finder.

**message** is the text string you wish to display within the alert dialog. This parameter can either be a quoted string of characters, a field, a variable, or a formula that results in a text string you wish to display. This parameter must be a text value.

**Action:** This statement will pause a procedure while Panorama displays a specified alert dialog and a message of your own choosing. This alert dialog is a modal dialog and must be responded to before you can continue the procedure. An alert dialog must have one or more buttons that allow the user to respond to the alert.

**Examples:** This sample procedure opens a yes-no dialog included in the Panorama program (resource number **1014**) and displays the message in the second parameter. If the user clicks on the yes button a save command will be executed.

```

alert 1014,"Do you want to save now?"
if info("dialogtrigger") contains "yes"
 save
endif

```

This is the same example except that the text string is stored in a local variable called `AlertMessage`.

```

local AlertMessage
AlertMessage = "Do you want to save now?"
alert 1014,AlertMessage
if info("dialogtrigger") contains "yes"
 save
endif

```

This is a similar example except that the resource (resource number **5001**) was created in a resource file called `Alerts`, which means it must be opened before it can be used. To do so you use the openresource statement.

```

openresource "Alerts"
alert 5001,"Do you want to save now?"
if info("dialogtrigger") contains "yes"
 save
endif

```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:**

- [cancelok](#) statement
- [customalert](#) statement
- [customdialog](#) statement
- [getscrap](#) statement
- [gettext](#) statement
- [info\("dialogtrigger"\)](#) function
- [message](#) statement
- [noyes](#) statement
- [okcancel](#) statement
- [openresource](#) statement
- [yesno](#) statement

# ALERTMODE

**Syntax:** ALERTMODE

**Description:** The **alertmode** statement allows you to suppress error and message alerts that Panorama would normally display. This is especially useful when using Panorama with a web server, since in that case there is no one to see the alert and you want to make sure the server keeps running no matter what.

**Parameters:** This statement has one parameter: **option**  
**option** controls whether Panorama displays alerts. If **option** is "yes", "true", or "on", alerts will be displayed. If **option** is "no", "false", or "off", alerts will not be displayed.

**Action:** Use the **alertmode** statement when you want to make sure that Panorama does not stop to display error message, no matter what happens. **Note: This option disables all Panorama alert dialogs until you either turn them back on or quit from Panorama - no matter what database is open.**

**Examples:** This example turns off all error message alerts (including alerts created by the [message](#) statement).

```
alertmode "off"
```

**Views:** This statement may be used in any view.

**See Also:** [onerror](#) statement  
[message](#) statement  
[alert](#) statement  
[okcancel](#) statement  
[cancelok](#) statement  
[yesno](#) statement  
[noyes](#) statement

# ALLINDEX

**Syntax:** ALLINDEX item,formula

**Description:** The **allindex** statement can be used to help locate the original data retrieved by the [lookupall\(\)](#) or [lookupcalendar\(\)](#) function.

**Parameters:** This statement has two required parameters: **item** and **formula**.

**item** tell what item retrieved by [lookupall\(\)](#) you want to locate. The item parameter must be a variable. Before the statement is used this variable should be set up with a number, 1, 2, 3, etc. This tells **allindex** whether you want to locate the first, second, third, etc. item of data retrieved by [lookupall\(\)](#). After the **allindex** statement the item variable will contain the line number in the target database that corresponds to the item, or zero if there is no such item.

**formula** must be a formula with a [lookupall\(\)](#) or [lookupcalendar\(\)](#) function as the primary function. You may use other functions or operators inside the parameters to this function, but the formula must not process the result of the lookup with any other function or operator.

**Action:** This statement calculates the line number of a record. The record is one of a set of records computed with the [lookupall\(\)](#) or [lookupcalendar\(\)](#) function.

**Examples:** The procedure example below will find out what item the user selected from a SuperObject list that has been filled with [lookupall\(\)](#), then use the AllIndex and find statements to locate the original record in the Reminders database for that item.

```

local AgendaItem AgendaItem=1
superobject "AgendaList","FindSelected",AgendaItem
if AgendaItem=0 stop endif ; nothing clicked!
allindex
AgendaItem,
lookupcalendar("Reminders",When,today(),Message,¶)
window "Reminders"
if info("selected")≠info("records") selectall endif
find seq()=AgendaItem
/* now we can modify the original data the mouse was pointing to */
reminder When,Message

```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [lookupall\(\)](#) function  
[lookupcalendar\(\)](#) function  
[seq\(\)](#) function

# ARCCOS(...)

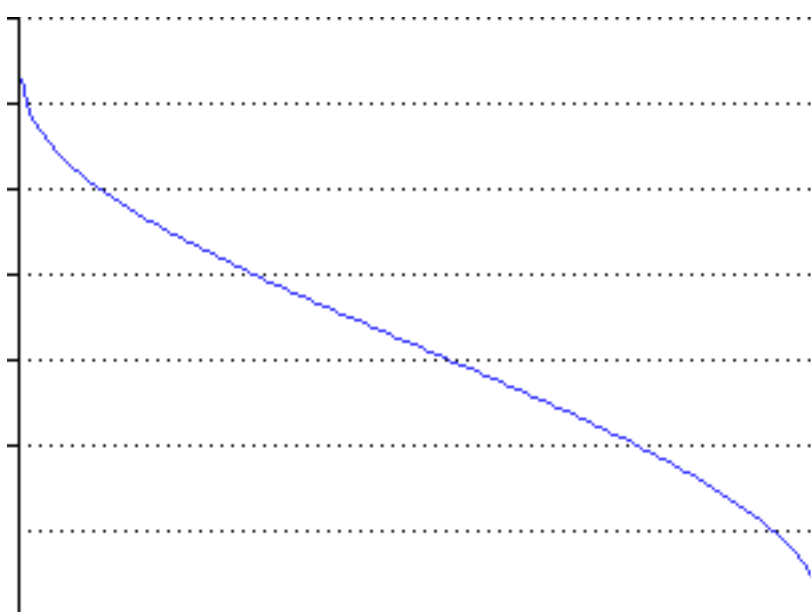
**Syntax:** ARCCOS(value)

**Description:** The `arccos()` function calculates the inverse cosine of a numeric value.

**Parameters:** This function has one parameter: value.  
**value** is a numeric value, which must be between -1 and +1.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse cosine function given input values from -1 to +1.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `arccos("0.5")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the inverse cosine, for example `arccos(val("0.665"))`.

**Floating point error.** This error occurs if you use an input value less than or equal to -1 or greater than or equal to +1. Mathematically, the inverse cosine function is only defined for values between -1 and +1.

**See Also:** [sin\(\)](#) function  
[cos\(\)](#) function  
[tan\(\)](#) function  
[arcsin\(\)](#) function  
[arctan\(\)](#) function  
[val\(\)](#) function

# ARCCOSH(...)

**Syntax:** ARCCOSH(value)

**Description:** The `arccosh()` function calculates the inverse hyperbolic cosine of a numeric value.

**Parameters:** This function has one parameter: value.

**value** is a numeric value, which must be between +1 and  $+\infty$  (positive infinity).

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse hyperbolic cosine function given input values from -100 to +100.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `arccosh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the inverse hyperbolic tangent, for example `arccosh(val("34"))`.

**Floating point error.** This error occurs if you use an input value less than 1. Mathematically, the inverse hyperbolic cosine function is only defined for input values 1 or greater.

**See Also:** [sinh\(\)](#) function  
[cosh\(\)](#) function  
[tanh\(\)](#) function  
[arcsinh\(\)](#) function  
[arctanh\(\)](#) function  
[val\(\)](#) function



# ARCSIN(...)

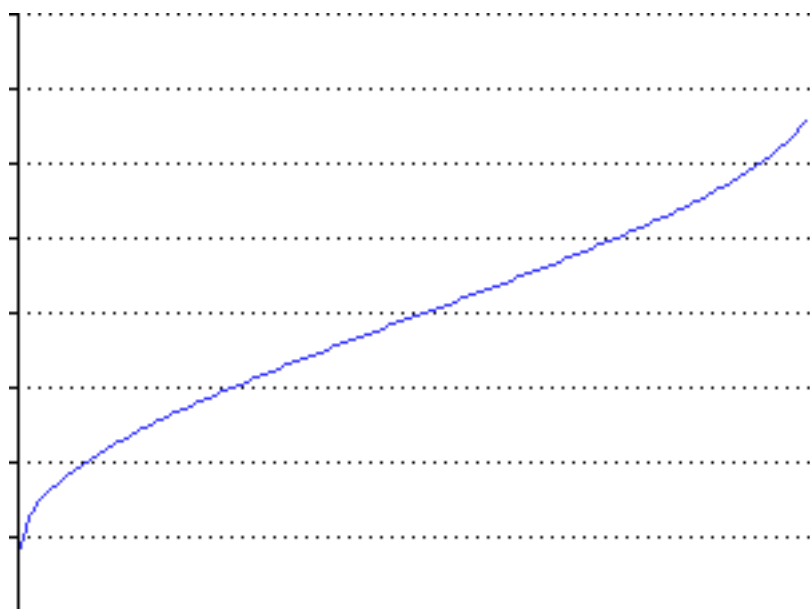
**Syntax:** ARCSIN(value)

**Description:** The arcsin( function calculates the inverse sine of a numeric value.

**Parameters:** This function has one parameter: value. value is a numeric value that must be between -1 and +1.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse cosine function given input values from -1 to +1.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `arcsin("0.5")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the inverse cosine, for example `arcsin(val("0.665"))`.

**Floating point error.** This error occurs if you use an input value less than or equal to -1 or greater than or equal to +1. Mathematically, the inverse sine function is only defined for values between -1 and +1.

**See Also:** [sin\( function](#)  
[cos\( function](#)  
[tan\( function](#)  
[arccos\( function](#)  
[arctan\( function](#)  
[val\( function](#)

# ARCSINH(...)

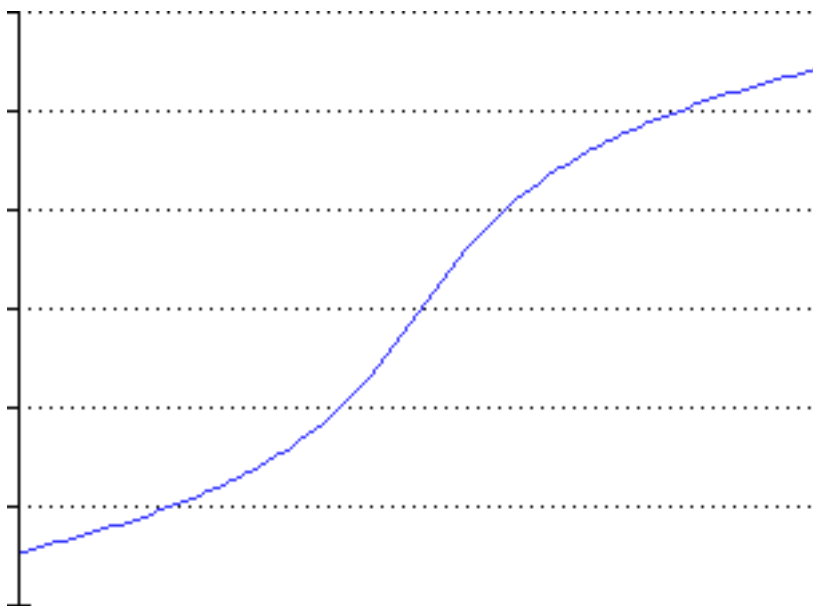
**Syntax:** ARCSINH(value)

**Description:** The `arcsinh()` function calculates the inverse hyperbolic sine of a numeric value.

**Parameters:** This function has one parameter: value.  
**value** is a numeric value.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse hyperbolic sine function given input values from -6 to +6.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `arcsinh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the hyperbolic sine, for example `arcsinh(val("34"))`.

**See Also:** [sinh\(\)](#) function  
[cosh\(\)](#) function  
[tanh\(\)](#) function  
[arccosh\(\)](#) function  
[arctanh\(\)](#) function  
[val\(\)](#) function

# ARCTAN(...)

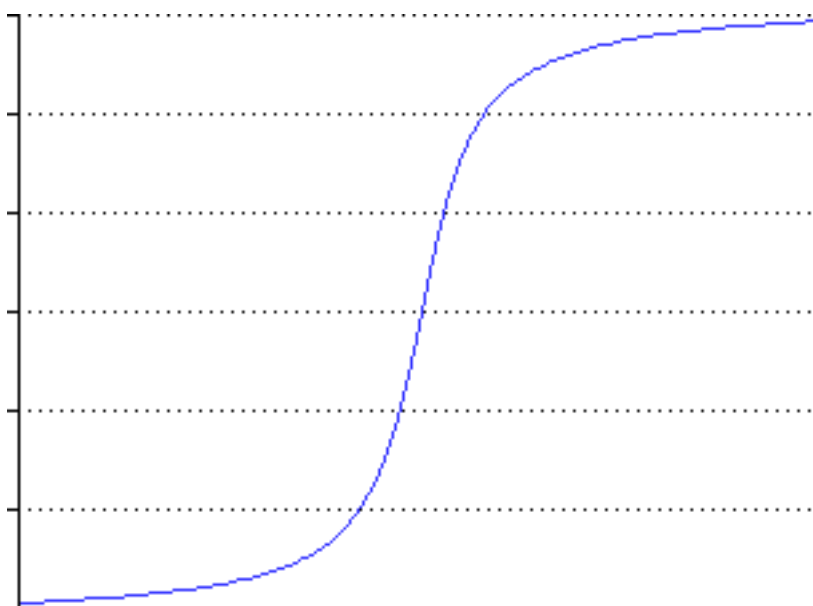
**Syntax:** ARCTAN(value)

**Description:** The `arctan()` function calculates the inverse tangent of a numeric value.

**Parameters:** This function has one parameter: value.  
**value** is a numeric value.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse hyperbolic sine function given input values from -10 to +10.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `arctan("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the inverse tangent, for example `arctan(val("34"))`.

**See Also:** [sin\(\)](#) function  
[cos\(\)](#) function  
[tan\(\)](#) function  
[arccos\(\)](#) function  
[arcsin\(\)](#) function  
[val\(\)](#) function

# ARCTANH(...)

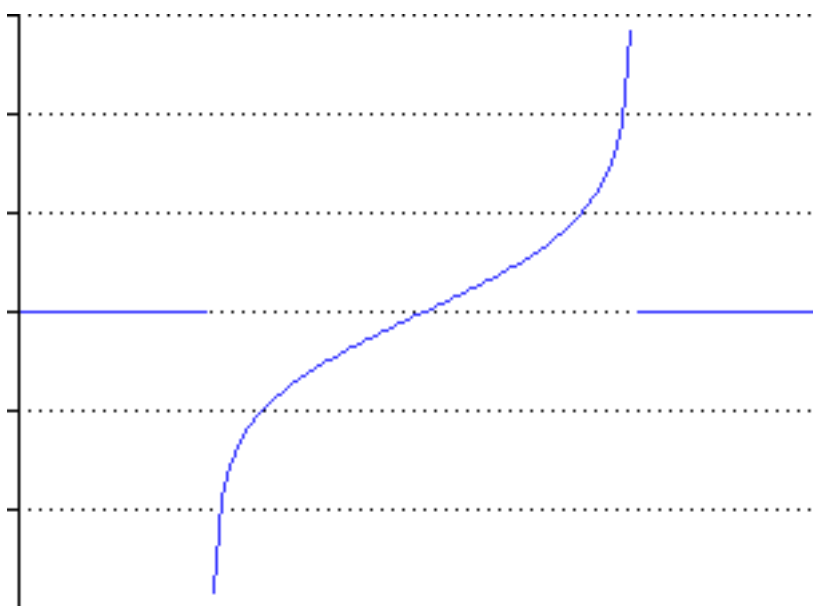
**Syntax:** ARCTANH(value)

**Description:** The `arctanh()` function calculates the inverse tangent of a numeric value.

**Parameters:** This function has one parameter: value.  
**value** is a numeric value, which must be between -1 and +1.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the inverse hyperbolic sine function given input values from -2 to +2.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `arctanh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the inverse hyperbolic tangent, for example `arctanh(val("34"))`.

**Floating point error.** This error occurs if you use an input value less than or equal to -1 or greater than or equal to +1. Mathematically, the inverse hyperbolic tangent function is only defined for values from -1 to +1.

**See Also:** [sinh\(\)](#) function  
[cosh\(\)](#) function  
[tanh\(\)](#) function  
[arcsinh\(\)](#) function  
[arccosh\(\)](#) function  
[val\(\)](#) function

# ARRAY(...)

**Syntax:** ARRAY(text,item,separator)

**Description:** The `array()` function extracts a single data item from a text array (see [text arrays](#)).

**Parameters:** This function has three parameters: text, item and separator.

**text** is the item of text that contains the data you want to extract.

**item** is the number of the data item you want to extract. The first item is item 1, the second is item 2, the third item is 3, etc.

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `\n` character (option-7). For tab delimited arrays use the `\t` character (option-L).

**Result:** This function returns an item of text from the array. Only the item itself is returned, the separator characters on each end are not included. If the item does not exist (for example if you ask for item 12 from a 7 item array) the function will return empty text ("").

**Examples:** There are 7 VHF television stations in Los Angeles. The procedure below will convert channel numbers into the names of the stations. For example, the procedure converts Channel 7 into KABC.

```
Stations=" ,KCBS , ,KNBC ,KTLA , ,KABC , ,KCAL , ,KTTV , ,KCOP "
<Channel Name>=array(Stations,7," , ")
```

The example uses an array called Stations. This array uses commas as a separator character.

The more complete example below displays the name of an element after the user enters the atomic number from 1 to 103. In this example the variable Elements contains an array of atomic element names, separated by semicolons (some of the assignment statement has been left out for clarity).

```
local Elements,AtomicNumber,AtomicName
Elements="Hydrogen;Helium;Lithium;Beryllium;Boron;"+
"Carbon;Nitrogen;Oxygen;Fluorine;Neon;"+
...
"Mendelevium;Nobelium;Lawrencium"
AtomicNumber="1"
gettext "Enter Atomic Number",AtomicNumber
AtomicNumber=val(AtomicNumber)
if error
 AtomicNumber=0
endif
AtomicName=array(Elements,AtomicNumber,";")
if AtomicName ≠ ""
 message "Atomic name is: "+AtomicName
else
 message "Atomic number must be an integer "+
"between 1 and 103."
endif
```

**Errors:**            **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the item parameter.

**See Also:**            [text arrays](#)  
[arrayrange\(\)](#) function  
[arraysearch\(\)](#) function  
[arrayelement\(\)](#) function  
[extract\(\)](#) function

# ARRAYBUILD

- Syntax:** ARRAYBUILD array,separator,database,formula
- Description:** The **arraybuild** statement builds an array by scanning a database and creating an array element for each record in the database (see “[TEXT ARRAYS](#)” on page 5844).
- Parameters:** This statement has four parameters: array, separator, database and formula.
- array** is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field.
- separator** is the separator character for the array, usually a carriage return (¶), tab (-) or comma.
- database** is the database that will be scanned. This database must be open. If this parameter is "" then the current database will be scanned. The **arraybuild** statement will scan **every** record in the database, including records that are not currently selected. If you want to build an array from only selected records, use the **arrayselctedbuild** statement.
- formula** is the formula that will be used to extract data from the database and build each array element. If the formula results in empty text ("") for a record then no element is added to the array for that record. The formula usually references fields in the database being scanned. It may also use the **seq()** function to find out the number of each record.
- Action:** This statement converts some of the data in a database into an array. Be sure to keep an eye on your scratch memory usage, since this statement can create a gigantic variable in no time flat! If a procedure needs to increase the scratch memory allocation it can use the **scratchmemory** statement.
- Examples:** The example below will display all the fish from the Fish Tank database with prices greater than \$30. Each fish name will be separated from the next by a comma.

```
local FishList
arraybuild FishList,"","Fish Tank",?(Price>30,Fish,"")
message FishList
```

Since the **arraybuild** statement can scan any open database, it can serve as a sophisticated lookup. This example looks up an address, given both the first and last names (the regular **lookup()** function can only search one field at a time). This example is hard coded for the name John Grant to make it clearer, but you can easily substitute fields or variables.

```
local CustAddress
arraybuild CustAddress,¶,"Customers",
?(FirstName="John" and LastName="Grant",
Address+--+City+--+State+--+Zip,"")
if arraysize(CustAddress,¶)>1
message "There is more than one John Grant!"
endif
Address=array(CustAddress,1)
City=arraybuild(CustAddress,2)
State=array(CustAddress,3)
ZipCode=array(CustAddress,4)
```

**Views:** This statement may be used in any view.

**See Also:** [arrayselectedbuild](#) statement  
[arraylinebuild](#) statement  
[seq\(\)](#) function  
[arrayscan\(\)](#) function



# ARRAYCHANGE(...)

**Syntax:** ARRAYCHANGE(text,newvalue,item,separator)

**Description:** The `arraychange(` function changes a single value inside a text array (see [text arrays](#)). Only the one item is changed, all the other items in the array remain the same.

This function has four parameters: text, newvalue, item and separator.

**Parameters:** **text** is the text array that contains the data you want to change.

**newvalue** is the new value of the data item.

**item** is the number of the data item you want to change. Items are numbered starting from 1 (1,2, 3,...). This item must already exist in the array. The `arraychange(` function will not add the item if it does not exist.

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the - character (option-L).

**Result:** This function returns a copy of the text array, with the data item changed. If you want to change the original array you should use an assignment statement (see below).

**Examples:** Suppose you have an array called `Colors`, and that this array uses a semicolon separator. The procedure below will change the 5th item of the array to `Navajo White`.

```
Colors=arraychange(Colors,"Navajo White",5,";")
```

The `arraychange(` function can only change an existing array element. If the array element does not already exist, it will not add it. In fact, it will do nothing. If it is possible that the array item does not exist you should check first with the [arraysize\(](#) function. The example below is a procedure that places a color name somewhere in the `Colors` array. If the specified array item does not exist it is created by replicating the separator character multiple times with the [rep\(](#) function.

```
loop ColorCount, ColorName, ColorNumber
ColorName=parameter(1)
ColorNumber=parameter(2)
ColorCount=arraysize(Colors,";")
if ColorNumber>ColorCount
Colors=Colors+rep(";",ColorNumber-ColorCount)
endif
Colors=arraychange(Colors,ColorName,ColorNumber,";")
```

Here's how another procedure might call this procedure to change a color item.

```
call .SetColor,"Boxcar Red",15
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array, newvalue or separator parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the item parameter.

**See Also:**

[text arrays](#)

[array\(\)](#) function

[arraysize\(\)](#) function

[arrayinsert\(\)](#) function

[arraydelete\(\)](#) function

[arrayfilter](#) statement

# ARRAYDEDUPLICATE

**Syntax:** ARRAYDEDUPLICATE oldarray,newarray,separator

**Description:** The `arraydeduplicate` statement removes duplicate elements from an array (see [text arrays](#)).

**Parameters:** This statement has three parameters: `oldarray`, `newarray` and `separator`.

**oldarray** is a formula that calculates the original array. Usually this is a text field or variable, but it is allowed to be any formula that produces a text result.

**newarray** is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field. If you want to change the array in place, use the same field or variable for both the `oldarray` and the `newarray`.

**separator** is the separator character for the array, usually a carriage return (¶), tab (→) or comma.

**Action:** This statement removes duplicate elements from an array. As a side effect it also sorts the array into alphabetical (A-Z) order.

**Examples:** The example builds an array containing a list of all the cities in Vermont where you have customers. Since you may have more than one customer in a given city, the `arraydeduplicate` statement removes the extras and makes sure each city is listed only once.

```
local Cities
select State="VT"
arrayselectedbuild Cities,¶,Customer,City
arraydeduplicate Cities,Cities,¶
```

**Views:** This statement may be used in any view.

**See Also:** [arraysort](#) statement  
[arrayfilter](#) statement

# ARRAYDELETE(...)

**Syntax:** ARRAYDELETE(text,item,count,separator)

**Description:** The `arraydelete`( function deletes one or more elements from the middle of a text array (see [text arrays](#)).

**Parameters:** This function has four parameters: `text`, `count`, `item` and `separator`.

**text** is the text array that you want to delete elements from.

**item** is the spot where you want the elements to be deleted.

**count** is the number of blank elements you want to delete from the array.

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ↵ character (option-L).

**Result:** This function returns a copy of the original text array, with the specified elements deleted from the middle.

**Examples:** The example below will delete the 3rd item from the SpeedDial array:

```
SpeedDial=arraydelete(SpeedDial,3,1,¶)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text` or `separator` parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the `item` or `count` parameters.

**See Also:** [text arrays](#)  
[array\( function](#)  
[arrayinsert\( function](#)  
[arrayfilter](#) statement

# ARRAYELEMENT(...)

**Syntax:** ARRAYELEMENT(text,position,separator)

**Description:** The `arrayelement()` function converts between character positions and array element numbers in a text array (see [text arrays](#)). Given a character position within the overall text, the `arrayelement()` function tells what array element the character is in. For example, in the array `red;blue;green` the 7th character (`u`) is in the 2nd array element.

**Parameters:** This function has three parameters: `text`, `position` and `separator`.

**text** is the text array that you are working with.

**position** is the position of the character within the overall text (starting with 1 for the first character).

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `¶` character (option-7). For tab delimited arrays use the `↵` character (option-L).

**Result:** This function returns a number. This is the number of the data element in the array corresponding to the character position parameter. If the position corresponds to a separator character, the function will return the element number of the data element to the right of the separator.

**Examples:** The procedure below adds a new color to the `RecentColors` array. It then arbitrarily cuts off the array so that it is less than 200 characters long. The `arrayelement()` function makes it possible to write this procedure so that the array can be cut off without cutting an array element in the middle.

```
local lastElement
RecentColors=parameter(1)+sandwich(¶,RecentColors,"")
lastElement=arrayelement(RecentColors,200,¶)
RecentColors=arrayrange(RecentColors,1,lastElement,¶)
```

This procedure could be useful for maintaining a pop-up menu of recently used colors. The procedure automatically keeps the menu to a reasonable size by lopping off old colors from the bottom if the array gets over 200 characters long.

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the position parameter.

**See Also:** [text arrays](#)  
[array\(\)](#) function  
[arraysearch\(\)](#) function

# ARRAYFILTER

**Syntax:** `ARRAYFILTER oldarray,newarray,separator,formula`

**Description:** The `arrayfilter` statement processes each element of an array with a formula (see [text arrays](#)).

**Parameters:** This statement has four parameters: `oldarray`, `newarray`, `separator` and `formula`.

**oldarray** is a formula that calculates the original array. Usually this is a text field or variable, but it is allowed to be any formula that produces a text result.

**newarray** is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field. If you want to change the array in place, use the same field or variable for both the `oldarray` and the `newarray`.

**separator** is the separator character for both arrays (they must use the same separator), usually a carriage return (`¶`), tab (`↵`) or comma.

**formula** is the formula for filtering. The `arrayfilter` statement operates by scanning `oldarray` element by element. For each element it processes the data with the formula you supply. The formula can use the `import()` function to access the actual value of the array element being processed. The `seq()` function can be used to access the array element number. Once the formula calculates a new value the `arrayfilter` statement takes that value and adds it to the end of the `newarray`.

**Action:** This statement filters the contents of an array. Rather than using a fixed processing method to filter the array, it allows the programmer to supply a formula that is used over and over again to filter each individual element of the array.

**Examples:** The example subroutine (called `.ArraySequence`) below adds a sequence number to the beginning of each element in the array that is passed to it (array in parameter 1, separator in parameter 2).

```
local tempArray
tempArray="Bob Johnson"+¶+"Sue Miller"+¶+"Joe Wills"
call .ArraySequence,tempArray,¶
```

For example, suppose you passed an array of names to this subroutine, like this.

```
local Places
arrayfilter parameter(1),Places,parameter(2),
"("+str(seq()+") "+import()
```

The result in the variable named `Places` would look like this.

```
(1) Bob Johnson
(2) Sue Miller
(3) Joe Wills
```

**Views:** This statement may be used in any view.

**See Also:** [arraysort](#) statement  
[arraydeduplicate](#) statement  
[import\(\)](#) function  
[seq\(\)](#) function  
[arraystrip\(\)](#) function

# ARRAYINSERT(...)

**Syntax:** ARRAYINSERT(text,item,count,separator)

**Description:** The `arrayinsert()` function inserts one or more elements into the middle of a text array (see [text arrays](#)).

**Parameters:** This function has four parameters: `text`, `count`, `item` and `separator`.

**text** is the text array that you want to insert elements into.

**item** is the spot where you want the new elements to be inserted.

**count** is the number of blank elements you want to insert into the array.

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `␣` character (option-7). For tab delimited arrays use the `↵` character (option-L).

**Result:** This function returns a copy of the original text array, with the new blank array elements inserted into the middle.

**Examples:** The example below will add 5 new array items to the SpeedDial array between the 2nd and 3rd array items:

```
SpeedDial=arrayinsert(SpeedDial,3,5,␣)
```

The new array items created by `arrayinsert()` are blank (empty). You can fill them in with the [arraychange\(\)](#) function.

The next example uses `arrayinsert()` and `arraychange()` to insert a new phone number at the beginning of the SpeedDial array, instead of at the end.

```
local NewPhone
NewPhone=" "
gettext "New Phone #",NewPhone
SpeedDial=arrayinsert(SpeedDial,1,1,␣)
SpeedDial=arraychange(SpeedDial,NewPhone,1,␣)
```

The example above separated the `arrayinsert()` and `arraychange()` functions into two separate assignments to make the procedure easier to understand. However, they can be combined into a single assignment like this:

```
local NewPhone
NewPhone=" "
gettext "New Phone #",NewPhone
SpeedDial=arraychange(arrayinsert(SpeedDial,1,1,␣),NewPhone,1,␣)
```

But just a minute sports fans...we're really not inserting in the middle of the array, but adding to the beginning. This can be done more simply with the [sandwich\(\)](#) function, like this.

```
local NewPhone
NewPhone=" "
gettext "New Phone #",NewPhone
SpeedDial=NewPhone+sandwich(␣,SpeedDial," ")
```



Of course the `arrayinsert()` and `arraychange()` example would still be the way to go if you need to insert the new item anywhere in the middle of the array, as opposed to the beginning or the end.

**Views:**

**Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text or separator parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the item or count parameters.

**See Also:**

[text arrays](#)

[array\(\)](#) function

[arraydelete\(\)](#) function

[sandwich\(\)](#) function

[arrayfilter](#) statement

# ARRAYLINEBUILD

**Syntax:** ARRAYLINEBUILD array,separator,database,formula

**Description:** The **arraylinebuild** statement builds a one-element array array by scanning a database and creating an array element for the current record in the database (see [text arrays](#)).

**Parameters:** This statement has four parameters: array, separator, database and formula.

**array** is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field.

**separator** is the separator character for the array, usually a carriage return (¶), tab (-) or comma.

**database** is the database that will be scanned. This database must be open. If this parameter is "" then the current database will be scanned. The **arraylinebuild** statement will “scan” only the current record in the database. The purpose of this is simply to build the array from any arrays in the record itself (see the formula section below). If you want to build an array from multiple records in the database use the [arraybuild](#) or [arrayselected-build](#) statement.

**formula** is the formula that will be used to extract data from the database and build each array element. The formula usually references fields in the database being scanned. It may also use the [arrayscan\(\)](#) function to extract array elements from fields in the record. (In fact, if your formula does not contain an [arrayscan\(\)](#) function there is no point in using the **arraylinebuild** statement.)

**Action:** This statement converts some of the data in a database into an array.

**Examples:** The example builds an array called PhoneList from the current record in the Contacts database. This example assumes that the Contacts database has a field called Phones which contains a carriage return separated list of phone numbers.

```
local PhoneList
arraylinebuild PhoneList,¶,"Contacts",
Name+": "+arrayscan(Phones,¶)
```

The final array will look something like this:

```
Susan Williams: Home 845-9564
Susan Williams: Work 631-4715
Susan Williams: Pager 482-5229
```

**Views:** This statement may be used in any view.

**See Also:** [arraybuild](#) statement  
[seq\(\)](#) function  
[arrayscan\(\)](#) function

# ARRAYRANGE(...)

- Syntax:** `ARRAYRANGE(text,firstitem,lastitem,separator)`
- Description:** The `arrayrange()` function extracts a series of data items from a text array (see [text arrays](#)).
- Parameters:** This function has four parameters: `text`, `firstitem`, `lastitem` and `separator`.
- text** is the item of text that contains the data you want to extract.
- firstitem** is the number of the first data item you want to extract. Items are numbered starting from 1 (1, 2, 3,...).
- lastitem** is the number of the last data item you want to extract. Items are numbered starting from 1 (1, 2, 3,...).
- separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ↵ character (option-L).
- Result:** This function returns a series of items from the array. It returns the first item, the last item, and everything in between (including any separators that are in between). If the last item does not exist (for example if you ask for item 12 from a 7 item array) the function will return up to the actual last item in the array. If both requested items do not exist, the function will return empty text ("").
- Examples:** This example will fill the variable `WeekDays` with the text `Mon,Tue,Wed,Thu,Fri`.
- ```
Days="Sun,Mon,Tue,Wed,Thu,Fri,Sat"
WeekDays=arrayrange(Days,2,6,"")
```
- The `arrayrange()` function is very handy for removing elements from the start or end of an array. The example below calculates what lessons a person still must complete to graduate, based on a numeric field or variable named `Completed`.
- ```
Lessons="Basic,Intermediate,Advanced"
ToDo=arrayrange(Lessons,Completed,10000,"")
```
- If the person has completed 1 lesson, `ToDo` will be `Intermediate,Advanced`. If the person has completed 2 lessons, `ToDo` will be `Advanced`.
- Errors:**
- Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.
- Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the `firstitem` or `lastitem` parameter.
- See Also:** [text arrays](#)  
[array\(\)](#) function  
[arraysearch\(\)](#) function  
[arraydelete\(\)](#) function  
[sandwich\(\)](#) function  
[arrayfilter](#) statement

# ARRAYREVERSE(...)

**Syntax:** ARRAYREVERSE(text,separator)

**Description:** The arrayreverse( function reverses the order of the elements in a text array (see Text Arrays). In other words, the first element becomes the last element, the second element becomes the second to last, etc.

**Parameters:** This function has two parameters: text and separator.

**text** is the text array that you want to modify.

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ␣ character (option-L).

**Result:** This function returns a copy of the original array with the elements reversed.

**Examples:** The arrayreverse( function reverses the order of the elements of an array. For example, the formula:

```
arrayreverse("1;2;3;4",";")
```

will produce the array

```
4;3;2;1.
```

The formula below could be used with an auto-wrap or text display object to display all the checks written to a company, starting with the most recent check (assuming the Checkbook database is sorted by date).

```
arrayreverse(lookupall("Checkbook",«Pay To»,Company,ChkNum,¶),¶)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text or separator parameters.

**See Also:** [text arrays](#)  
[array\( function](#)  
[arraysort statement](#)

# ARRAYSCAN(...)

**Syntax:** ARRAYSCAN(field,separator)

**Description:** The `arrayscan()` function allows the individual elements of a text array in a database field to be exported on separate lines (see [text arrays](#)).

**Parameters:** This function has two parameters: `field` and `separator`.

**field** is the name of the field that contains the array you want to export. (You can also use a variable, but this usually doesn't make sense).

**separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ␣ character (option-L).

**Result:** This function returns one element from the array. However, unlike the `array()` function, the `arrayscan()` modifies the way the [export](#) and [arraybuild](#) statements work. These statements will repeat the formula containing `arrayscan()` over and over again for each record. Each time, the function will return the next element in the array, until there are no more items.

**Examples:** Suppose your database has a `Phones` field which contains an array of one or more phone numbers, separated by a carriage return. Each array element contains the type of phone number, a comma, and the phone number itself, like this:

`home,(714) 555-1212 office,(714) 555-8932 fax,(714) 555-8938`

The procedure below will export the phone numbers with one record per phone number:

```
export "Phone List",Name+--+
array(arrayscan(Phones,¶),1,"")+--+
array(arrayscan(Phones,¶),2,"")+¶
```

This procedure will output a text file something like this:

```
Joan Selbyhome(714) 555-1212
Joan Selbyoffice(714) 555-8932
Joan Selbyfax(714) 555-8938
Sally Rogersoffice(508) 777-8922
Sally Rogersfax(508) 777-8910
Chris Robertsoffice(909) 874-1234
```

Notice that no blank lines are exported. Panorama counts the number of elements in the array, and outputs exactly that number of lines. If you use multiple `arrayscan()` functions in the formula Panorama will export enough lines to handle the largest array. The `arrayscan()` function can also be used in the formula for the [arraybuild](#), [arrayselectedbuild](#), or [arraylinebuild](#) statements. The `arrayscan()` function works exactly the same as it does with the [export](#) statement, but the final result is an array instead of a text file.

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text or separator parameters.

**Field or variable does not exist.** This error occurs if the field you specify is not in the current database. You probably misspelled the field name.

**See Also:**

- [text arrays](#)
- [array\(\) function](#)
- [export statement](#)
- [arraybuild statement](#)
- [arrayselectedbuild statement](#)
- [arraylinebuild statement](#)

# ARRAYSEARCH(...)

- Syntax:** ARRAYSEARCH(array,text,startitem,separator)
- Description:** The `arraysearch()` function searches a text array (see [text arrays](#)) to see it contains a specific value.
- Parameters:** This function has four parameters: `array`, `text`, `startitem`, and `separator`.
- array** is the text array that you want to search.
- text** is the text that you want to search for. This parameter may contain the wildcard characters `?` and `*`. For example, to search for array items that start with `John` use `John*`. To search for any array item containing `Pacific` use `*Pacific*`. The array item must match the text exactly, including upper/lower case. For more information on wildcard characters, see the `matchexact` operator.
- startitem** is the spot in the array where you want the search to begin from. If you want to search the entire array, this parameter should be one.
- separator** is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `␣` character (option-7). For tab delimited arrays use the `␣` character (option-L).
- Result:** If the `arraysearch()` function finds an array element that matches what you are searching for it returns the number of that array element (1, 2, 3, etc.). If there is no matching element, the function returns 0.
- Examples:** The example below uses the `arraysearch()` function to look up the atomic number of an element. If it finds the element in the array, it displays the number

```

local Elements,AtomicNumber,AtomicName
Elements="Hydrogen;Helium;Lithium;Beryllium;Boron;" +
 "Carbon;Nitrogen;Oxygen;Fluorine;Neon;" +
 ...
 "Mendelevium;Nobelium;Lawrencium"
AtomicName=""
gettext "Enter Name of Element",AtomicName
AtomicNumber=arraysearch(Elements,upperword(AtomicName),1,";")
ifAtomicNumber ≠ 0
 message "Atomic number of "+AtomicName+" is:"+str(AtomicNumber)
else
 message AtomicName+" is not an element."
endif

```

The `arraysearch()` function allows you to use wildcard searches to match a pattern, similar to the `matchexact` operator. The example below will search a carriage return delimited array named `Names`, and gives the user the chance to delete each occurrence of the name within the array. For example, if the user types in `john` the procedure will stop at each name that begins with `John`.

```

local findName, soFar
findName=""
gettext "Enter first few letters of name:",findName
findName=upperword(findName)+"*"
soFar=1
loop
 soFar=arraysearch(Names,findName,soFar,␣)

```

```
if soFar>0
 yesno "Delete "+array(Names,soFar,1)+"?"
 if clipboard() contains "yes"
 Names=arraydelete(Names,soFar,1,1)
 else
 soFar=soFar+1
 endif
endif
while soFar > 0
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array, text or separator parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the startitem parameter.

**See Also:** [text arrays](#)  
[array\(\)](#) function  
[search\(\)](#) function  
[arrayelement\(\)](#) function  
matchexact operator



# ARRAYSELECTEDBUILD

- Syntax:** `ARRAYSELECTEDBUILD array,separator,database,formula`
- Description:** The `arrayselectedbuild` statement builds an array by scanning a database and creating an array element for each record in the database (see [text arrays](#)).
- Parameters:** This statement has four parameters: `array`, `separator`, `database` and `formula`.
- array** is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field.
- separator** is the separator character for the array, usually a carriage return (¶), tab (-) or comma.
- database** is the database that will be scanned. This database must be open. If this parameter is "" then the current database will be scanned. The `arrayselectedbuild` statement will scan only currently selected records in the database. If you want to build an array from all the records in the database, including unselected records, use the [arraybuild](#) statement.
- formula** is the formula that will be used to extract data from the database and build each array element. If the formula results in empty text ("") for a record then no element is added to the array for that record. The formula usually references fields in the database being scanned. It may also use the `seq()` function to find out the number of each record.
- Action:** This statement converts some of the data in a database into an array. Be sure to keep an eye on your scratch memory usage, since this statement can create a gigantic variable in no time flat! If a procedure needs to increase the scratch memory allocation it can use the [scratchmemory](#) statement.
- Examples:** The example below will display the name and phone number of every selected person in the database.
- ```
local Prospects arrayselectedbuild Prospects,¶,"Customers",
Name+" "+Phone
message Prospects
```
- Views:** This statement may be used in any view.
- See Also:** [arraybuild](#) statement
[arraylinebuild](#) statement
[seq\(\)](#) function
[arrayscan\(\)](#) function

ARRAYSIZE(...)

Syntax: `ARRAYSIZE(text,separator)`

Description: The `arraysize()` function counts the number of items in a text array (see [text arrays](#)).

Parameters: This function has two parameters: `text` and `separator`. `text` is the text array that you want to count. `separator` is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `¶` character (option-7). For tab delimited arrays use the `↵` character (option-L).

Result: This function returns a number. This is the number of elements in the array. If there is no text in the array, the function will return one.

Examples: This example uses the `arraysize()` function to display the number of forms in the current database. (The `dbinfo("forms","")` function creates an array listing all the forms in the current database, separated by carriage returns.)

```
message "This database contains "+
str(arraysize(dbinfo("forms",""),¶))+" forms"
```

The `arraysize()` function can be used to check the size of an array before a procedure performs an operation on that array. For example, the `arraychange()` function can only change an existing array element. If the array element does not already exist, it will not add it. The example below uses the `arraysize()` function to make sure that an array element exists before the procedure attempts to change that element.

```
local LastColorNumber, ColorName, ColorNumber
ColorName=parameter(1)
ColorNumber=parameter(2)
LastColorNumber=arraysize(Colors,";")-1
if ColorNumber >= ColorCount
Colors=Colors+rep(";",ColorNumber-ColorCount)
endif
Colors=arraychange(Colors,ColorName,ColorNumber,";")
```

Here's how another procedure might call this procedure to change a color item.

```
call .SetColor,"Boxcar Red",15
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.

See Also: [text arrays](#)
[array\(\)](#) function
[extract\(\)](#) function

ARRAYSORT

Syntax: `ARRAYSORT oldarray,newarray,separator`

Description: The `arraysort` statement alphabetizes (A-Z) the elements in an array (see [text arrays](#)).

Parameters: This statement has three parameters: `oldarray`, `newarray` and `separator`. (Note: Your Panorama 3 Supplement may list these parameters in a different order, which is incorrect.)

oldarray is a formula that calculates the original array. Usually this is a text field or variable, but it is allowed to be any formula that produces a text result.

newarray is the variable or field that will contain the new array. If you use a field for this parameter it must be a text field. If you want to change the array in place, use the same field or variable for both the `oldarray` and the `newarray`.

separator is the separator character for the array, usually a carriage return (¶), tab (→) or comma.

Action: This statement sorts the array into alphabetical (A-Z) order.

Examples: The example below builds an alphabetized list of the forms in the current database. The list is separated by carriage returns, and could be used with a pop-up menu or List SuperObject.

```
global FormList
arraysort dbinfo("forms",""),FormList,¶
```

Views: This statement may be used in any view

See Also: [arraydeduplicate](#) statement
[arrayfilter](#) statement

ARRAYSTRIP(...)

Syntax: ARRAYSTRIP(text,separator)

Description: The `arraystrip()` function removes any blank elements from a text array (see [text arrays](#)).

Parameters: This function has two parameters: `text` and `separator`.

`text` is the text array that you want to strip the blank elements from.

`separator` is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `¶` character (option-7). For tab delimited arrays use the `↵` character (option-L).

Result: This function returns a copy of the original text array, with any blank array elements removed from the array.

Examples: The procedure below builds a list of all Arizona companies in the current database. Some of the customers in Arizona may be individuals with no company name. The procedure uses the `arraystrip()` function to remove these blank elements from the array produced by the `lookupall()` function.

```
local CompanyList
CompanyList=lookupall( info("databasename"), "State", "AZ", Company, ¶ )
CompanyList=arraystrip(CompanyList, ¶ )
```

A procedure can use the `arraystrip()` function in combination with the `arrayfilter()` statement to produce a subset of a database. This procedure takes the `CompanyList` array and removes all companies except for companies that begin with the letter B.

```
arrayfilter CompanyList,CompanyList,¶,
?( import() beginswith "B",import()," ")
CompanyList=arraystrip(CompanyList, ¶ )
```

The `arrayfilter` statement converts any array element that does not start with a B to an empty array element. Once this is done, the `arraystrip()` function strips out the empty array elements, leaving only array elements beginning with the letter B.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.

See Also: [text arrays](#)
[array\(\)](#) function
[arrayfilter](#) statement

ASC(...)

- Syntax:** ASC(text)
- Description:** The `asc()` function converts the first character of a text item into a number from 0 to 255 based on the [ascii](#) table.
- Parameters:** This function has one parameter: `text`.
text is the item of text that you want to convert to an ASCII value. Only the first character is converted, the rest of the text is ignored.
- Result:** The result of this function is always an integer from 0 to 255. See the table below for a list of ASCII values and characters.
- Examples:** This function allows you to perform math on characters. For example, suppose you want to know how many letters are between two letters (for example there are four letters between C and H). This procedure will calculate the number of letters between two letters.

```

local StartLetter, EndLetter, LetterCount
StartLetter="" EndLetter=""
gettext "Enter first letter", StartLetter
gettext "Enter second letter", EndLetter
LetterCount=asc(StartLetter)-asc(EndLetter)
message str(LetterCount-1)

```

The procedure uses the `asc()` function to convert the letters into numbers, then subtracts them.

Here is a procedure that generates and displays a sequenced list of characters. The user is allowed to enter the characters they want the list to start and stop with.

```

local Alphabet, Letter, StartLetter, EndLetter
Alphabet=""
StartLetter="" EndLetter=""
gettext "Enter first letter", StartLetter
gettext "Enter second letter", EndLetter
if StartLetter>EndLetter stop endif
Letter=asc(StartLetter)
loop
    Alphabet=Alphabet+chr(Letter)
    Letter=Letter+1
while Letter<=asc(EndLetter)
message Alphabet

```

Table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			enter						tab				trn		
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
											esc	←	→	↑	↓
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
Ä	Å	Ç	É	Ë	Ï	Ö	Ü	á	à	â	ã	ä	å	ç	é
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
†	°	£	€	§	•	¶	ß	©	®	™	'	"	≠	Æ	Ø
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
∞	±	≤	≥	¥	μ	ð	Σ	Π	π	∫	≠	°	Ω	æ	ø
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
¿	¡	¬	√	ƒ	≈	Δ	«	»	...		À	Á	Â	Ã	Ä
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
–	—	"	"	'	'	÷	ϕ	ÿ	ÿ	ı	œ	<	>	fi	fl
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
‡	·	,	„	%	Â	É	Á	È	È	Í	Î	Ï	Ï	Ó	Ô
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
•	ö	ú	û	ü	ı	ˆ	˜	˘	˙	˚	˛	˜	˜	˜	˘

Errors:

Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value with this function, for example `chr(34)`. If you have a number you must convert the number to text before using it with this function, for example `chr(str(34))`.

See Also:

[ascii](#)
[chr\(\)](#) function

ASCII

On most Macintosh systems there are 256 possible characters. (Japanese and Chinese allow thousands of characters). Each character has a number from 0 to 255. Of these 256 characters, about 200 are associated with symbols (letters, digits, punctuation, etc.). The table below shows each of the 256 characters and the number associated with each character.

0	1	2	3 enter	4	5	6	7	8	9 tab	10	11	12	13 return	14	15
16	17	18	19	20	21	22	23	24	25	26	27 esc	28 left	29 right	30 up	31 down
32	33 !	34 "	35 #	36 \$	37 %	38 &	39 '	40 (41)	42 *	43 +	44 ,	45 -	46 .	47 /
48	49 0	50 1	51 2	52 3	53 4	54 5	55 6	56 7	57 8	58 9	59 :	60 ;	61 <	62 =	63 > ?
64	65 @	66 A	67 B	68 C	69 D	70 E	71 F	72 G	73 H	74 I	75 J	76 K	77 L	78 M	79 N O
80	81 P	82 Q	83 R	84 S	85 T	86 U	87 V	88 W	89 X	90 Y	91 Z	92 [93]	94 ^	95 _
96	97 '	98 a	99 b	100 c	101 d	102 e	103 f	104 g	105 h	106 i	107 j	108 k	109 l	110 m	111 n o
112	113 p	114 q	115 r	116 s	117 t	118 u	119 v	120 w	121 x	122 y	123 z	124 {	125 	126 }	127 ~ □
128	129 À	130 Á	131 Â	132 Ã	133 Ä	134 Å	135 Ö	136 Û	137 à	138 á	139 â	140 ã	141 ä	142 å	143 ç é è
144	145 ê	146 ë	147 í	148 î	149 ï	150 ñ	151 ó	152 ô	153 õ	154 ö	155 ø	156 ú	157 ù	158 û	159 ü
160	161 †	162 °	163 £	164 §	165 •	166 ¶	167 ß	168 ©	169 ®	170 ™	171 '	172 "	173 ≠	174 Æ	175 Ø
176	177 ∞	178 ±	179 ≤	180 ≥	181 ¥	182 µ	183 ð	184 Σ	185 Π	186 π	187 ∫	188 ²	189 ³	190 Ω	191 æ ø
192	193 ¿	194 ¡	195 ¬	196 √	197 f	198 ≈	199 Δ	200 «	201 »	202 ...	203 À	204 Á	205 Â	206 Ã	207 Ä Å
208	209 –	210 —	211 " "	212 ' '	213 ' '	214 ÷	215 ϕ	216 ÿ	217 ÿ	218 ı	219 œ	220 <	221 >	222 fi	223 fl
224	225 ‡	226 ·	227 ,	228 „	229 ‰	230 Â	231 É	232 Á	233 È	234 È	235 Í	236 Î	237 Ï	238 Ó	239 Ô
240	241 •	242 Ö	243 Û	244 Ü	245 ı	246 ˆ	247 ˜	248 ˘	249 ˙	250 ˚	251 ˛	252 ˜	253 ˘	254 ˚	255 ˛

The numbers have not been assigned to symbols arbitrarily, but have been assigned using a system called ASCII. The number associated with a character is called the ASCII value of the character. (For you techno-weenies, ASCII stands for American Standard Computer Interchange Interface.) Looking at the table you'll notice that the characters with ASCII values from 0-31 have no symbols. These characters are used for special keys like return (13), tab (9), and enter (3). ASCII value 32 is the space character, then we have some punctuation. ASCII values 48 through 57 are the numeric digits 0 through 9, in order. ASCII values 65-90 are the upper case letters A through Z, in alphabetical order. ASCII values 97-122 are the lower case letters a through z, again in alphabetical order.

Panorama uses the ASCII values of characters when it compares two text items to see which is larger or smaller. Since the ASCII value of B (66) is greater than the ASCII value of A (65), the text item B is "larger" than A. However, the ASCII value of a (97) is greater than B (66), so the text item a is "larger" than B. You have to watch out for this problem whenever you compare text that is a mixture of upper and lower case.

The first 32 ASCII values do not display, but are reserved for control functions. Some of the special keys on the keyboard produce these values. Here is a set of descriptions of some of these values, and the special keys (if any) that go with them.

Value	Description
01	Home Key
03	Enter Key
08	Delete (also sometimes called backspace)
09	Tab (use <code>\t</code> in formula)
10	Line Feed (used only by PC's, not Macintosh)
11	Vertical Tab (used only for import/export)
12	Form Feed (used only by PC's, not Macintosh)
13	Carriage Return (use <code>\r</code> in formula)
27	Escape Key
28	Left Arrow
29	Right Arrow
30	Up Arrow
31	Down Arrow

These control values can also be produced by holding down the CONTROL key while you press a letter. For example, pressing CONTROL-A produces a character with value 1, CONTROL-B is 2, etc. This technique is commonly used on PC systems, but not by Macintosh software.

ASCII7TO8(...)

Syntax: ASCII7TO8(encodedtext)

Description: The `ascii7to8()` function takes 7 bit encoded text produced by the `ascii8to7()` function and converts it back into regular 8 bit text (which may contain any `ascii` character from 0 to 255).

Parameters: This function has one parameter: `encodedtext`.

encodedtext is the encoded 7-bit text you want to convert back into regular 8 bit text.

Result: This function returns regular `ascii` text. However, if the encoded 7 bit text is not the exact same text (same length and same contents) that was produced by the `ascii8to7()` function, the `ascii7to8()` function will return empty text (""). You can add headers and trailers to the encoded 7 bit text, but you must not modify the actual 7 bit text itself.

Examples: The example below takes the contents of the field `LetterBody`, packs it into 7 bit encoded format, adds a header and trailer, then copies it onto the clipboard.

```
clipboard="Letter from "+info("user")+ "transmitted "+
datepattern( today(), "DayOfWeek, Month ddnth, yyyy")+
"[[[[[[["+ascii8to7(LetterBody)+]]]]]]"+
```

You can copy this into an e-mail message, and send it to someone else. They can convert it back to regular text with this procedure, which uses the `ascii7to8()` function.

```
local bStart,bEnd,pLetter
pLetter=clipboard()
bStart=search(pLetter,"[[[[[[[")
bEnd=search(pLetter,"]]]]]]")
if bStart=0 or bEnd=0
message "No encoded letter on the clipboard"
stop
endif
LetterBody=ascii7to8(pLetter[(bStart+6),(bEnd-1)])
```

You can use this pair of functions to transmit complete Panorama records through an e-mail system. This procedure copies all the selected records into the clipboard in the encoded format.

```
local Batch
arraybuild Batch,info("databasename"),
replace( exportline(),chr(11))
clipboard="Database transmission: "+
pattern( info("selected"), " record~")+
" from "+info("databasename")+
"Transmitted on: "+
pattern( today(), "DayOfWeek, Month ddnth, yyyy")+
"[[[[[[["+ascii8to7(Batch)+]]]]]]"+
```

The following procedure takes this encoded data and converts it back to Panorama data, appending it to the end of the current database. (The databases at both ends of this transaction should have the exact same fields!)

```
local bStart,bEnd,Batch
Batch=clipboard()
bStart=search(Batch,"[[[[[[")
bEnd=search(Batch,"]]]]]")
if bStart=0 or bEnd=0
message "No encoded letter on the clipboard"
stop
endif
Batch=ascii7to8(Batch[(bStart+6),(bEnd-1)])
if Batch=""
message "This transmission has been corrupted!"
stop
endif
openfile "+@Batch"
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the encodedtext parameter.

See Also: [ascii8to7\(\)](#) function

ASCII8TO7(...)

Syntax: ASCII8TO7(text)

Description: The `ascii8to7` function takes regular text (which may contain any [ascii](#) character from 0 to 255) and processes it into a special encoded format that only uses ASCII characters 32 through 127. This format allows the text to be sent through any e-mail system (some e-mail systems do not allow characters 128 through 255). The encoded text can be turned back into regular text with the [ascii7to8](#) function. (These functions get their name because regular text uses 8 binary bits per character. The encoded text uses only 7 binary bits per character.)

Parameters: This function has one parameter: text.

text is the text you want to convert into 7 bit encoded format.

Result: This function returns the same text you passed to it, but converted into a special 7 bit encoded format.

Examples: The example below takes the contents of the field LetterBody, packs it into 7 bit encoded format, adds a header and trailer, then copies it onto the clipboard.

```
clipboard="Letter from "+info("user")+ " transmitted "+
datepattern( today(),"DayOfWeek, Month ddnth, yyyy")+¶+
"[[[[[[ "+¶+ascii8to7(LetterBody)+¶+" ]]]]]"+¶
```

You can copy this into an e-mail message, and send it to someone else. They can convert it back to regular text with this procedure, which uses the [ascii7to8](#) function.

```
local bStart,bEnd,pLetter
pLetter=clipboard()
bStart=search(pLetter,"[[[[[[[")
bEnd=search(pLetter," ]]]]]")
if bStart=0 or bEnd=0
    message "No encoded letter on the clipboard"
    stop
endif
LetterBody=ascii7to8(pLetter[(bStart+6),(bEnd-1)])
```

You can use this pair of functions to transmit complete Panorama records through an e-mail system. This procedure copies all the selected records into the clipboard in the encoded format.

```
local Batch
arraybuild Batch,¶,info("databasename"),
replace( exportline(),¶,chr(11))
clipboard="Database transmission: "+
pattern( info("selected") ," record~")+
" from "+info("databasename")+¶+
"Transmitted on: "+
pattern( today(),"DayOfWeek, Month ddnth, yyyy")+¶+
"[[[[[[ "+¶+ascii8to7(Batch)+¶+" ]]]]]"+¶
```

The following procedure takes this encoded data and converts it back to Panorama data, appending it to the end of the current database. (The databases at both ends of this transaction should have the exact same fields!)

```
local bStart,bEnd,Batch
Batch=clipboard()
bStart=search(Batch,"[[[[[[")
bEnd=search(Batch,"]]]]]")
if bStart=0 or bEnd=0
  message "No encoded letter on the clipboard"
  stop
endif
Batch=ascii7to8(Batch[(bStart+6),(bEnd-1)])
if Batch=""
  message "This transmission has been corrupted!"
  stop
endif
openfile "+@Batch"
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text parameter.

See Also: [ascii7to8\(function](#)

ATTACHSERVER

Syntax: ATTACHSERVER

Description: The **attachserver** statement takes a single user Panorama database and converts it into a Partner/Server database linked into an SQL server database.

Parameters: This statement has no parameters.

Action: This statement is very useful for distributing Partner/Server database templates. The database designer sets up all the server options in advance. The final end user simply presses a button that uses the **attachserver** statement to create the actual SQL database.

Before you use the **attachserver** statement you must have set up the Server Options for the database. To do this, go into the design sheet and open the Server Options dialog (in the Server Dialog). In this dialog you must set up the server field lengths and indexes.

Examples: This example converts the current database into a Partner/Server database. As part of the process an SQL database will be created on the server computer. The [save](#) statement saves a local copy of the database. This local copy contains the link to the server database.

```
attachserver  
save
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views.

See Also: [detachserver](#) statement
[serverfile](#) statement
[info\("serverstatus"\)](#) function
[info\("serverfile"\)](#) function

AVERAGE

Syntax: AVERAGE

Description: The **average** statement calculates averages and subaverages for the selected records in the current field.

Parameters: This statement has no parameters.

Action: This statement calculates averages for the current field. The current field must be numeric. If the database contains summary records, this statement will calculate subaverages for each summary record, along with an overall average for the whole database. If there are not any summary records in the database, one will be added at the end of the database and the overall average calculated and placed into the summary record. This statement has the same effect as choosing the **Average** command in the **Math** menu.

Examples: This simple example calculates the average of the Balance field.

```
field Balance  
average
```

This example calculates the average total for each state, along with the overall average.

```
field State  
group  
field Total  
average
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [total](#) statement
[count](#) statement
[group](#) statement

BACKSPACEKEY

Syntax: BACKSPACEKEY

Description: The `backspacekey` statement deletes the current record from the current database.

Parameters: This statement has no parameters.

Action: This statement deletes the current record from the database. The cursor will move up to the previous record (if any). The contents of the current line are copied into the clipboard before the line is deleted (use the [pasterecord](#) statement to insert the line back into the database.) The statement has the same effect as pressing the backspace key in the data sheet.

Examples: This example erases the current record if the Name field is empty.

```
if Name=""  
    backspacekey  
endif
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views (view-as-list forms only).

See Also: [deleterecord](#) statement
[cutrecord](#) statement
[copyrecord](#) statement
[deleteall](#) statement
[info\("records"\)](#) function

BAUD

Syntax: BAUD rate

Description: The **baud** statement specifies the baud rate for subsequent [dialmodem](#) and [dialprinter](#) statements.

Parameters: This statement has one parameter: rate.

rate is the baud rate that should be used the next time Panorama dials using a serial port (using the [dialmodem](#) or [dialprinter](#) statements). Choose the rate from the list below:

300
1200
2400
4800
7200
9600
14400
28800
38700

Action: This statement doesn't perform any visible action on its own. However, the next [dialmodem](#) or [dialprinter](#) statement will use the rate specified by this command.

Examples: This example dials toll free information using a 9600 baud Hayes compatible modem that is connected to the modem port.

```
baud 9600
dialmodem "ATDT8005551212"
```

Views: This statement may be used in any view.

See Also: [dialmodem](#) statement
[dialprinter](#) statement

BEEP

Syntax: BEEP

Description: The **beep** statement makes the Macintosh speaker beep once.

Parameters: This statement has no parameters.

Action: This statement makes the speaker “beep.” The actual sound may be a beep, or it may be whatever sound you have set up as the system beep. If you want to play another sound use the [sound](#) or [playsound](#) statements, which can play any sound in a resource file.

Examples: This example causes the Macintosh to beep if the Name field is empty.

```
if Name=""  
    beep  
endif
```

Views: This statement may be used in any views.

See Also: [sound](#) statement
[playsound](#) statement

BINARY DATA

Background: By now probably everyone who has ever used a computer for more than a week has heard that at their core, computers work with 1's and 0's, on and off, true and false. This is called binary data, because there are only two options. Fortunately, users don't ever have to deal with raw binary data. The programmers take the 1's and 0's and give them structure to create text, numbers, pictures, and other complex elements.

It's not much fun, and it's rarely necessary, but Panorama does allow a procedure programmer to work with raw unstructured, binary data 1's and 0's. When you work with raw binary data, it will always be in a text field or variable. Panorama normally interprets text as a series of characters, as described earlier in this chapter. The binary functions, however, do not interpret the binary data as characters. Instead, they allow you to directly access and manipulate the 1's and 0's. Panorama used the text data type to hold raw binary data because text may be of any length and places no restrictions on the binary information that is placed in it. (However, the text may look very strange if you display it in the data sheet or on a form, more on this later).

Bits: The fundamental unit of computer information is a bit. A bit contains a single 1 or 0. However, a bit is too small to be of much use by itself, so usually several bits are grouped together into a collection called a byte, word, or longword.

Bytes: A byte is a collection of 8 bits. There are 256 possible combinations of 1's and 0's within a byte (2 multiplied by itself 8 times, i.e. $2*2*2*2*2*2*2*2=256$). These 256 combinations could represent characters, they could represent numbers from 0 to 256 or anything else. The `byte()` function takes a number from 0 to 255 and converts it into the corresponding pattern of 8 bits.

Words: A word is a collection of 16 bits (or 2 bytes). There are 65,536 possible combinations of 1's and 0's within a word (2 multiplied by itself 16 times). These 65,536 possible combinations could represent numbers from 0-65,535 or they could represent 65,536 of anything else. the `word()` function takes a number from 0 to 65,536 and converts it into the corresponding pattern of 16 bits.

Longwords: A longword is a collection of 32 bits (or 4 bytes). There are over 4 billion possible combinations of 1's and 0's within a longword (2 multiplied by itself 32 times). The `longword()` function takes a number from 0 to 4,294,967,295 and converts it into the corresponding pattern of 32 bits.

Creating Binary Values: Binary values are created with the `byte()`, `word()` and `longword()` functions. The example below builds a text data value from a longword, a word, a word and a byte. The resulting text item has a length of 9 (4+2+2+1).

```
global rawData
rawData=
longword(96345)+
word(1249)+
word(9004)+
byte(80)
```

Accessing Binary Information:

Numeric values can be recovered from a text data item with the [binaryvalue\(\)](#) function. The text input into this function must have a length of 1, 2, or 4. You can use text funnels to control the position and length of the data being converted. The example below will extract four values from a text item that is at least 9 bytes long.

```
global rawData
local myLong,myFirstWord,mySecondWord,myByte
myLong=binaryvalue(rawData[1;4])
myFirstWord=binaryvalue(rawData[5;2])
mySecondWord=binaryvalue(rawData[7;2])
myByte=binaryvalue(rawData[9;1])
```

If `rawData` contains the information stored in it from the previous example, the `myLong` will be 96345, `myFirstWord` will be 1249, `mySecondWord` will be 9004, and `myByte` will be 80.

See Also:

[byte\(\)](#) function
[word\(\)](#) function
[longword\(\)](#) function
[binaryvalue\(\)](#) function
[string255\(\)](#) function
[text255\(\)](#) function
[radix\(\)](#) function
[radixstr\(\)](#) function
[textstuff\(\)](#) function
[c/pascal structures](#)

BINARYVALUE(...)

Syntax: BINARYVALUE(data)

Description: The `binaryvalue()` function converts binary data (a byte, word, or longword) into a number (see [binary data](#)).

Parameters: This function has one parameter: data.

data is the binary value that you want to convert into a number. This value must be a byte, a word (2 bytes) or a longword (4 bytes).

Result: This function converts the binary data into a number.

Examples: See [c/pascal structures](#) for examples of the `binaryvalue()` function.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the data parameter.

Illegal number. This error occurs if you attempt to convert a value that is not 1, 2, or 4 bytes.

See Also: [byte\(\)](#) function
[word\(\)](#) function
[longword\(\)](#) function
[radix\(\)](#) function
[radixstr\(\)](#) function

BLUE(...)

Syntax: BLUE(COLOR)

Description: The `blue()` function extracts the blue intensity from a color.

Parameters: This function has one parameter: `color`.

`color` is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

Result: This function extracts the intensity of the blue component of this color. This intensity is a number between 0 (black) and 65535 (full intensity).

Examples: The example below calculates the blue intensity of the color (in percent, from 0 to 100%).

```
Intensity=blue(HighlightColor)*100/65535
```

Errors: For more examples of color, see [colors](#).

Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value for the color parameter.

See Also: [rgb\(\)](#) function
[hsb\(\)](#) function
[green\(\)](#) function
[red\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement
[colors](#).

BRIGHTNESS(...)

Syntax: BRIGHTNESS(color)

Description: The `brightness()` function extracts the brightness of a color. **Brightness** specifies how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does. On the Apple color picker the Brightness is specified by the scroll bar on the right. This is a number from 0 to 65535.

Parameters: This function has one parameter: `color`.

color is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

Result: This function extracts the brightness of the color. This intensity is a number between 0 and 65535.

Examples: The example below calculates the brightness of the color (in percent, from 0 to 100%).

```
SquintFactor=saturation(HighlightColor)*100/65535
```

For more examples of color, see [colors](#).

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the color parameter.

See Also: [rgb\(\)](#) function
[hsb\(\)](#) function
[red\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement
[colors](#).

BUILDREMINDER

Syntax: BUILDREMINDER date,time,type,reminderfield

Description: The **buildreminder** statement builds a new reminder (see [reminder data](#)).

Parameters: This statement has four parameters: date, time, type and reminderfield.

date is the date for the new reminder. The **buildreminder** statement cannot create recurring reminders. Recurring reminders (2nd tuesday of the month, etc.) must be created manually using the [reminder](#) dialog.

time is the time for the new reminder.

type is the type of the new reminder: 0 for appointments or 1 for to-dos (see [reminder data](#)).

reminderfield is the field (or variable) where the new reminder data should be stored.

Action: This statement builds an appointment or to-do reminder at the specified time and date.

Examples: This example creates a new reminder for tomorrow at 9am. It then allows the user to modify the reminder with a dialog.

```
buildreminder today(+1,time("9am"),0,Reminder
Message=grabdata("Contacts","Name")
reminder Reminders,Message
```

Views: This statement may be used in a Data Sheet or Form view.

See Also: [reminder](#)(function
[reminderdate](#)(function
[remindertime](#)(function
[remindertype](#)(function
[reminder](#) statement
[reminder data](#)

BYTE(...)

Syntax: BYTE(number)

Description: The `byte()` function converts a number into a single byte of binary data (see [binary data](#)). (Note: the `byte()` function is basically the same as the `chr()` function.)

Parameters: This function has one parameter: `number`.
number is the value that you want to convert into a binary number. This value must be between 0 and 255.

Result: This function converts the number into a single byte of binary data (8 bits). This binary data should be handled as text data.

Examples: This example selects all records where the Text field contains tabs.

```
select Text contains byte(9)
```

See [c/pascal structures](#) for additional examples of the `byte()` function.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the number parameter. **Illegal number.** This error occurs if you attempt to convert a value less than 0 or greater than 255.

See Also: [word\(\)](#) function
[longword\(\)](#) function
[radix\(\)](#) function
[radixstr\(\)](#) function

C/Pascal Structures

Background: The binary data functions allow the procedure programmer to build, read, and modify C and Pascal structures. Why would you want to do that? Perhaps you want to pass a structure to an external procedure, or you might want to read or write a data file that contains such structures. In C on the Macintosh, a char is a byte, a short is a word, and a long is a longword. In Pascal, a byte is a byte, an integer is a word, and a longint is a longword. The example below shows a typical 70 byte C structure with three elements. This particular structure is used by the Macintosh toolbox to identify a file or folder.

```
typedef struct FSSpec {
short vRefNum; /* volume reference number */
long parID; /*directory ID of parent directory */
Str63 name; /* file name or directory name */
} FSSpec;
```

Here's a Panorama subroutine called `.BuildFileSpec` that can build such a structure. The subroutine has three parameters: the volume reference number (a number), the directory ID of the parent (a number), and the file or directory name (text). For this example it's not really important to understand what these numbers mean, but you do want to see how they are combined into the C structure. The structure is left in a global variable called `FileSpec` where it could be passed to an external procedure.

```
global FileSpec
FileSpec=word( parameter(1))+
longword( parameter(2))+
string255( parameter(3),64)
```

The next example extracts the three elements of the structure in `FileSpec` and places them into three global variables.

```
global FileSpec,vRefNum,parentID,fileName
vRefNum=binaryvalue(FileSpec[1;2])
parentID=binaryvalue(FileSpec[3;4])
fileName=text255(FileSpec[7;64])
```

Using the `textstuff()` function you can change individual elements in the C or Pascal structure. This example changes the file name:

```
FileSpec=textstuff(FileSpec,string255(NewName,64),6)
```

Remember, with the `textstuff()` function the offset starts at 0 for the first character, not 1 as it does for text funnels.

See Also:

- [byte\(\)](#) function
- [word\(\)](#) function
- [longword\(\)](#) function
- [binaryvalue\(\)](#) function
- [string255\(\)](#) function
- [text255\(\)](#) function
- [radix\(\)](#) function
- [radixstr\(\)](#) function
- [textstuff\(\)](#) function

CALCCROSSTAB

Syntax: CALCCROSSTAB

Description: The **calccrosstab** statement recalculates the currently active crosstab window.

Parameters: This statement has no parameters.

Action: This statement recalculates the information in the current cross tab window. It has the same effect as choosing the **Calculate Crosstab Tool** in the tool palette.

Examples: This example opens the Budget crosstab and re-calculates it using the latest information in the database.

```
opencrosstab "Budget "  
calccrosstab
```

Views: This statement may be used only in the Crosstab view.

See Also: [opencrosstab](#) statement
[gocrosstab](#) statement

CALENDARDATE(...)

Syntax: CALENDARDATE(date,boxnumber)

Description: The `calendardate()` function is designed to help in creating monthly calendars. A standard monthly calendar has 6 rows and 7 columns (Sunday through Saturday) for a total of 42 boxes. For any given month from 28 to 31 of these boxes will be valid dates. The `calendarday()` function calculates what date corresponds to one of these 42 boxes.

Parameters: This function has two parameters: `date` and `boxnumber`.

date is any date in the month being displayed.

boxnumber is the box within the monthly calendar being displayed. The boxes are numbered from 1 to 42, starting with the upper left hand corner. The table below shows the position of all 42 monthly calendar boxes.

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42

Result: This function returns a number corresponding to a date, or zero if the specified calendar box does not contain a day in this month.

Examples: The output of the `calendardate()` function is usually fed into a `lookupall()`, `lookupcalendar()`, or `lookupptime()` function. The last two functions can be used to lookup the events (appointments, to-do's, etc.) that occur on a particular day. You'll probably want to create your monthly calendar with a Matrix SuperObject™. The matrix should be 6 rows by 7 columns, with the cells numbered in horizontal order. To display the reminders that should appear in each of the calendar's 42 boxes use the formula below in an auto-wrap text object or a Text Display SuperObject™ inside the matrix frame. (This example assumes your reminders are stored in a database called `Reminders`. This database has at least two fields: `When`, which contains the Reminder data type (see [reminder data](#)), and `Message`, which contains text for each reminder.)

```
lookupcalendar(
  "Reminders",
  When,
  calendardate(Date,info("matrixcell")),
  Messages,¶)
```

This formula displays the times for each reminder.

```
lookuprtime(  
    "Reminders",  
    When,  
    calendardate(Date,info("matrixcell")),  
    "HH:MM AM/PM",#1)
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date or box parameter.

See Also: [calendarday\(function](#)
[info\("matrixcell"\) function](#)
[lookupcalendar\(function](#)
[lookuprtime\(function](#)
[reminder data](#)

CALENDARDAY(...)

Syntax: CALENDARDAY(date,boxnumber)

Description: The `calendarday()` function is designed to help in creating monthly calendars. A standard monthly calendar has 6 rows and 7 columns (Sunday through Saturday) for a total of 42 boxes. For any given month from 28 to 31 of these boxes will be valid dates. The `calendarday()` function calculates what number from 1 to 31 (if any) should be displayed in one of these 42 boxes.

Parameters: This function has two parameters: `date` and `boxnumber`.

date is any date in the month being displayed.

boxnumber is the box within the monthly calendar being displayed. The boxes are numbered from 1 to 42, starting with the upper left hand corner. The table below shows the position of all 42 monthly calendar boxes.

S	M	T	W	T	F	S
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	32	33	34	35
36	37	38	39	40	41	42

Result: This function returns a number from 1 to 31, or zero if the specified calendar box does not contain a day in this month.

Examples: The example below creates a basic calendar for a month. It builds a list of the days in a month in a field named Days.

```

local box,aday,DayList
box=1
DayList=""
loop
  aday=(" "+str(calendarday(Date,box)))[-2,-1]+" "
  if aday=" 0 "
    aday=" "
  endif
  DayList=DayList+aday
  if (box mod 7)=0
    DayList=DayList+¶
  endif
  box=box+1
until 42
Days=DayList

```

If you display the field Days using the Monaco font it will look something like this (this shows July 1995):

```

                1
            2  3  4  5  6  7  8
          9 10 11 12 13 14 15
        16 17 18 19 20 21 22
       23 24 25 26 27 28 29
      30 31
```

For more complicated calendars you'll probably want to use a Matrix SuperObject™. The matrix should be 6 rows by 7 columns, with the cells numbered in horizontal order. To display the date use the formula below in an auto-wrap text object or a Text Display SuperObject™ inside the matrix frame.

```
zeroblank(calendarday(Date,info("matrixcell")))
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date or box parameter.

See Also: [calendardate\(function](#)
[info\("matrixcell"\) function](#)

CALL

Syntax: CALL name[,parameter1,parameter2,...parameterN]

Description: The call statement executes another procedure, in the same database, as a subroutine.

Parameters: This statement has one required parameter: name.

There may also be additional parameters that you can define for passing data between the main program and the subroutine.

name is the name of the procedure you wish to execute as a subroutine. This must be the name of a procedure in the same database as the currently running procedure. (To call a procedure in another database, use the [farcall](#) statement. To execute a subroutine within the same procedure use the [shortcall](#) statement.) If the name contains blanks, symbol characters, or punctuation characters it must be surrounded by quotes (" ") or chevrons (« »).

parameter1, parameter 2, etc. are optional parameters that are defined by you, the subroutine programmer! Each parameter may be a field, variable, or a complete formula. (However, if you want to change the value of a parameter from inside the subroutine, the parameter must be a field or variable, not a formula.) The subroutine can find out the value of a parameter using the [parameter\(\)](#) function. To change the value of a parameter, the subroutine can use the [setparameter](#) statement.

Action: This statement allows a procedure programmer to execute a second procedure from a first as a subroutine to that procedure. Subroutines become important when you wish to use the same code at different times within the same procedure or within multiple procedures. This allows you to write the code once and use it again and again rather than duplicating the same code over and over again.

Subroutines normally finish when the end of the procedure is reached. To stop the subroutine before the end of the procedure, use the [rtn](#) statement. The [rtn](#) statement makes Panorama return control to the original procedure.

Examples: This simple example shows how to call a procedure called Summarize.

```
call Summarize
```

This example calls a subroutine named Calculate %. The subroutine's name is more than one word and contains symbol characters.

```
call "Calculate %"
```

The example below shows that a subroutine can be called from within the middle of a procedure. Once the subroutine finishes the parent procedure will continue executing.

```
global area,width,height
gettext "Enter Width in inches.",width
gettext "Enter Height in inches.",height
area = val(width) * val(height)
call Conversion
field Dimensions
...
...
...
```

Views: This statement may be used in any view.

See Also: [farcall](#) statement
[parameter\(\)](#) function
[rtn](#) statement
[setparameter](#) statement
[shortcall](#) statement

CANCELOK

- Syntax:** CANCELOK text
- Description:** The `cancelok` statement pauses a procedure and displays a modal dialog showing the text and two buttons **Cancel** (the default button) and **Ok**. The name of the button clicked on will be written to the clipboard.
- Parameters:** This statement has one parameter: text.
- `text` is the character string that will appear in a modal dialog displayed on screen. This text string must be surrounded by quote marks (" "). The string limit is dependent on the characters used and assumes the system font (Chicago, 12 point).
- Action:** This statement allows the procedure programmer to pause the procedure by presenting the user with a modal dialog that asks a question requiring one of two responses **Cancel** or **Ok**. Whichever response is selected will be written to the clipboard and it can be tested for later in the procedure.
- Examples:** This simple example will have Panorama display a modal dialog asking you if you wish to cancel or continue the procedure. If you click on the **Cancel** button the procedure will stop.

```
cancelok "Do you wish to continue this procedure?"
if clipboard() = "Cancel"
    stop
endif
...
...
```

This example first tests to see if a select command failed to select any records and if it fails asks you if you wish to cancel or try again.

```
local companyname
beginning:
gettext "Enter Company Name.",companyname
select «Company» contains companyname
if info("empty")
    cancelok "Cancel to stop, Ok to try again."
    if clipboard() contains "cancel"
        stop
    endif
    if clipboard() contains "ok"
        goto beginning
    endif
endif
```

The example is a procedure that runs again and again in a loop until you end the loop by clicking on the **Cancel** button.

```
local companyname
loop
    gettext "Enter Company Name.",companyname
    select «Company» contains companyname
    if info("empty")
        beep
        message "No records meeting select criteria."
```

```
endif  
cancelok "Cancel to stop, Ok to try again."  
until clipboard() contains "cancel"  
message "You are done."
```

Views: This statement may be used in a procedure run from any view, and also works when no windows are open at all.

See Also: [alert](#) statement
[clipboard\(\)](#) function
[customalert](#) statement
[customdialog](#) statement
[getscrap](#) statement
[gettext](#) statement
[info\("dialogtrigger"\)](#) function
[message](#) statement
[noyes](#) statement
[okcancel](#) statement
[openresource](#) statement
[yesno](#) statement
[alertmode](#) statement

CARDVALIDATE

Syntax: CARDVALIDATE cardnumber,result

Description: The `cardnumber` statement allows a procedure to verify that a number is a valid credit card number.

Parameters: This statement has two parameters: `cardnumber` and `result`.

cardnumber is a text parameter containing the credit card number. This number should contain only numeric digits — any dashes, spaces or punctuation should be removed before using this.

result will be either `Ok` or `Error`.

Action: Credit cards have an internal checksum that allows a number to be validated for simple data entry errors (for example missing or transposed digits). This statement checks to make sure that a number is a valid credit card number. Of course the statement cannot tell whether this card number has actually been issued, or what the credit limit is or any other financial information about the card. It simply provides a simple check for missing or transposed numbers. (Basically, if this statement says that the number is in error you know for sure that the number is wrong, but if this statement says the number is valid you would still need to check with the issuer to determine if this is a valid card.)

Examples: This example checks the card number in the field `CCNumber` to see if it is a valid credit card number.

```
local cctemp,ccvalid
cctemp=striptonum(CCNumber)
cardvalidate cctemp,ccvalid
if ccvalid<>"Ok"
    message "This credit card number is not valid!"
endif
```

Views: This statement may be used in any view.

See Also: [striptonum\(\)](#) function

CASE

Syntax: CASE true-false test

Description: The **case** statement is used in place of an **if endif** structure to execute a section of code when the case is true; false cases are bypassed. All cases after the first true case are ignored. Case statements are terminated by an **endcase** statement. An optional **defaultcase** statement is available and will execute when all cases evaluate as false. You may have up to 75 case statements in a row before you need an **endcase** statement. Case statements cannot be nested inside other case statements.

Parameters: This statement has one parameter: true-false test.

true-false test is an equation, formula, or function that resolves itself in a true or false result.

If the result is **false** Panorama goes on to the next case statement and evaluates the test. If the test is **true** Panorama will execute all commands after that **case** statement up to but not including the next **case**, **defaultcase**, or **endcase** statement. Afterwards Panorama executes all code after the **endcase** statement. If all cases evaluate false Panorama will either execute code between the **defaultcase** statement and **endcase** statement if the optional **defaultcase** is present or it will continue executing code after the **endcase** statement.

Action: These statements are used for evaluating a series of possible results and executing code based on the first true result or an optional default if all previous results were false.

Using the **Check Procedure** tool on a procedure that has a **case** statement with no corresponding **endcase** statement will result in an error alert. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts its operation.

Examples: This example has Panorama assigning a dollar amount to the field Rate based on the chosen shipper.

```

case Shipby = "UPS Ground"
    Rate = 5.00
case Shipby = "UPS Blue"
    Rate = 7.50
case Shipby = "UPS Red"
    Rate = 10.00
case Shipby = "Fed-ex standard"
    Rate = 12.00
case Shipby = "Fed-ex overnight"
    Rate = 18.00
case Shipby = "Fed-ex overnight Saturday"
    Rate = 25.00
defaultcase
    Rate = 4.50
endcase

```

This example has Panorama calculate pricing and determining inclusions and customer status based on the quantity ordered.

```

case QtyΩ = 1
    UnitCostΩ = 50.00
    PΩ = 50.00
case QtyΩ > 1 and QtyΩ ≤ 3

```

```
        UnitCostΩ = 47.50
        PΩ = QtyΩ * UnitCostΩ
    case QtyΩ > 3 and QtyΩ ≤ 7
        UnitCostΩ = 45.00
        PΩ = QtyΩ * UnitCostΩ
        «Inclusions» = "T-shirt"
        «Status» = "Bonus Customer"
    case QtyΩ > 7 and QtyΩ UnitCostΩ = 42.50
        PΩ = QtyΩ * UnitCostΩ
        «Inclusions» = "Sweatshirt & hat"
        «Status» = "Value Customer"
    case QtyΩ ≥ 10
        UnitCostΩ = 40.00
        PΩ = QtyΩ * UnitCostΩ
        «Inclusions» = "Coffee mug & T-shirt & hat"
        «Status» = "Deluxe Customer"
    endcase
```

Views: This statement may be used any view, and also works when no windows are open at all.

See Also: [defaultcase](#) statement
[else](#) statement
[endcase](#) statement
[endif](#) statement
[if](#) statement

CELL

Syntax: CELL value

Description: The **cell** statement enters the value into the currently active field (or cell).

Parameters: This statement has one parameter: value

value is the data that will be placed into the currently active field. value may be a literal value, a variable that contains a value or a formula which returns a value.

Value 's type can be text, numeric, date or picture but it must match that of the field it is going into, otherwise the procedure will return a **Type mismatch** error.

Action: This statement places value into the currently active field. If that field has any automatic equations assigned to it in the Design Sheet those equations will trigger after the cell statement does it's job. However, an automatic procedure assigned as an equation will not trigger. This is a safety feature to prevent that procedure from executing in an infinite loop.

Examples: This example has the cell statement change the Price field in the first selected record to 30% more than the current value in that numeric field.

```
firstrecord
field Price
cell Price* 1.30
```

The example above is almost equivalent to this example. The only difference is that the example below will not trigger any automatic equations associated with the Price field.

```
firstrecord
Price=Price*1.30
```

This simple example shows a procedure that replaces the value in the active field with the same value. This could be useful if the field has equations assigned to it in the Design Sheet so that those equations would trigger after the cell statement is finished. This same process could be accomplished by doing a copycell and then a pastecell, but is faster.

```
cell «» /* Note: «»represents the current field */
```

This procedure checks each selected record in the database to see if the customer has paid \$500 or more. If so it tags them as a Value Customer in the text field called Status.

If Status has the equation `«Commission» = «Total Paid» * .10` in the Design Sheet it will be triggered for each record tagged Value Customer.

```
hide
firstrecord
loop
  if «Total Paid» ≥ 500
    field «Status»
    cell "Value Customer"
  endif
```

```
    downrecord  
until info("stopped")  
show
```

This example shows a procedure that makes all choices type fields in the database center aligned.

```
opendesignsheet  
firstrecord  
field Align ;«Align» is a Design Sheet field  
loop  
    if «Type» = "Choices"  
        cell "Center"  
    endif  
    downrecord  
until info("stopped")  
newgeneration  
closewindow  
save
```

Views: This statement may be used for the Data Sheet, Design Sheet, or Form view.

See Also: [editcell](#) statement
[editselect](#) statement
[formulafill](#) statement
[pastecell](#) statement
[scrapcalc](#) statement

CHANGE

Syntax: CHANGE from to words caps

Description: The **change** statement allows Panorama to replace one string with another in the current field for all selected records. You may also control whether you replace full or partial strings and whether those string must match upper and lower case before the substitution. Panorama temporarily converts the field to text type before making the change and converts it back after the change so this statement will work on any field type except picture.

Parameters: This statement has two required parameters: from and to and two optional parameters: words and caps. Each parameter must be separated by at least one space.

from is the text string you want replaced in the current field. This text string can be a single character, a portion of a larger string, one word, several words, or the entire field. This parameter can either be a quoted text string, a variable which contains a text string, or a formula which results in a text string. This string cannot exceed 150 characters.

to is the text string you want to replace the from value with in the current text field. This text string can be a single character, a portion of a larger string, one word, several words, or the entire field. This parameter can either be a quoted text string, a variable which contains a text string, or a formula which results in a text string. This string cannot exceed 150 characters.

words (optional) tells the change statement to make the change only if the text string being replaced is not part of a larger text string.

caps (optional) tells the change statement to make the change only if the text string being replaced has the same upper and lower case characters in the same order as the from parameter.

Action: This statement will work on any fields. It will allow you to make a global replace in one field only (the active field) for all selected records.

This statement has the same effect as choosing the **Change** command from the **Search** menu without necessarily bring up the Change dialog.

Warning: The total number of characters being changed in the column of the database for all selected records cannot exceed 10,000 characters before or after the change.

Examples: This example would replace the abbreviation Corp. with the full word Corporation in the current field. However, if Corp does not end in a period (.) it will not be replaced.

```
change "Corp." "Corporation"
```

This example could be use to police typos. It will change all double dash characters (--) to a single dash character (-) in the text field called Home Phone.

```
field «Home Phone»
change "--" "-"
```

This example would simply open the change dialog and allow you to fill it in yourself.

```
change dialog
```

This example will change the word new to old, but WILL NOT change newton to oldton.


```
change "new" "old" words
```

This procedure will ask the user to input the from and to values then change the database accordingly. However, the capitalization in the database must exactly match how the user entered it. For example, if the user asks to change lead to gold, this procedure **WILL NOT** change Lead to Gold.

```
local From,To  
From = "" ;initializing the variable  
To = "" ;initializing the variable  
gettext "Enter text to change.",From  
gettext "Enter replacement text",To  
change From To caps
```

This example will change any width in the Design Sheet that contains a 10. For example 10 will change to 6, and 104 will change to 64.

```
godesignsheet  
field Width  
change "10" "6"  
newgeneration  
gosheet
```

Views: This statement may be used in any view.

See Also: [replace\(\)](#) function

CHANGEOBJECTS

Syntax: CHANGEOBJECTSattribute,formula

Description: The `changeobjects` statement changes an attribute (size, font, color, etc.) of selected graphic objects in the current form. You can only change one attribute at a time. This statement is usually used with the `selectobjects` statement to define which objects are to be modified.

Parameters: This statement has two parameters: attribute and formula. attribute controls what attribute of the selected object is changed. There are twelve different attributes that may be modified (see Action section below). formula calculates the new value of the attribute. This value may need to be text, a number, a rectangle, or a color depending on what attribute you are changing.

Action: This statement changes an attribute (size, font, color, etc.) of selected graphic objects in the current form. You may choose from the twelve different attributes shown in this table:

ATTRIBUTE	EXAMPLE
rectangle (size/location)	<code>changeobjects "rectangle",rectangle(20,5,45,80)</code>
field name ¹	<code>changeobjects "fieldname","Price5"</code>
font ²	<code>changeobjects "font","Palatino"</code>
textsize ²	<code>changeobjects "textsize",24</code>
text style ^{2,3}	<code>changeobjects "textstyle",4</code>
color	<code>changeobjects "color",rgb(20000,0,65535)</code>
fill pattern ⁴	<code>changeobjects "fillpattern",radix(16,"FF888888FF888888")</code>
line pattern ⁴	<code>changeobjects "linepattern",radix(16,"FFFFFFFFFFFFFFFF")</code>
line width ⁵	<code>changeobjects "linewidth",4</code>
expandable ⁶	<code>changeobjects "expand",-1</code>
expandshrink ⁶	<code>changeobjects "expandhrink",-1</code>
lock (in graphic editor) ⁶	<code>changeobjects "lock",0</code>

1. Applies only to data cells
2. Applies only to objects that contain text
3. 0=plain, 1=bold, 2=italic, 4=underline, 8=outline, 16=shadow
4. Use "" for none (empty pattern)
5. Specify line width in pixels
6. 0=false, -1=true.

Examples: The example below changes all objects in the body of a report to Palatino 9 point, and all objects in the header of a report to Eurostile 14 point. (Note: For this example to work the objects in the form must be set up so that the names of objects in the body contain "Body" and the names of objects in the header contains "Header".)

```
selectobjects objectinfo("name") contains "Body"
changeobjects "font","Palatino"
changeobjects "textsize",9
selectobjects objectinfo("name") contains "Header"
changeobjects "font","Eurostyle"
changeobjects "textsize",14
selectnoobjects
```

Views: This statement may be used in the Form view.

See Also: [selectobjects](#) statement
[selectallobjects](#) statement
[selectnoobjects](#) statement
[objectinfo\(\)](#) function

CHR(...)

Syntax: CHR(number)

Description: The `chr()` function converts a number from 0 to 255 into a single character of text based on the [ascii](#) table.

Parameters: This function has one parameter: `number`.

number is the value of the ASCII character you want to generate. See the table below for a list of values and characters.

Result: The result of this function is always a single character text item.

Examples: This function can be used to generate any character, including characters that cannot normally be entered from the keyboard. This function also makes it easy to generate sequences of characters, for example `ABC` or `123`. Here is a procedure that generates and displays the complete alphabet from A to Z. (The ASCII value for A is 65 and the value for Z is 90.)

```

local Alphabet,Letter
Alphabet=""
Letter=65
loop
    Alphabet=Alphabet+chr(Letter)
    Letter=Letter+1
while Letter<=90
message Alphabet

```

The `chr()` function can also be used to generate special control characters, including

```

chr(13) carriage return
chr(9)  tab

```

Here is a formula that uses the `chr()` function to create a three line address.

```
Name+chr(13)+Address+chr(13)+City+", "+State+Zip
```

(Formulas like this are so common that Panorama has special symbols for carriage return and tab. The symbol for carriage return is ¶ (option-7), the symbol for tab is ⇥ (option-L). Here is the same formula re-written using the ¶ symbol instead of `chr(13)`.)

```
Name+¶+Address+¶+City+", "+State+Zip
```

ASCII Table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			enter						tab				trn		
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
											esc	←	→	↑	↓
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
À	Á	Â	Ã	Ä	Å	Ö	Û	à	á	â	ã	ä	å	ç	è
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
ê	ë	í	ì	î	ï	ñ	ó	ô	õ	ö	ø	ú	ù	û	ü
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
†	°	£	€	§	•	¶	ß	©	®	™	'	"	≠	Æ	Ø
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
∞	±	≤	≥	¥	μ	ð	Σ	Π	π	∫	≠	°	Ω	æ	ø
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
¿	¡	¬	√	ƒ	≈	Δ	«	»	...		À	Á	Â	Ë	œ
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
–	—	"	"	'	'	÷	ϕ	ÿ	ÿ	ı	ı	<	>	fi	fl
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
‡	·	,	„	%	Â	É	Á	È	È	Í	Î	İ	İ	Ó	Ô
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255
•	Ö	Ü	Ü	Ü	ı	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ	ˆ

Errors:

Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a text value with this function, for example `chr("34")`. If you have text item you must convert the text to a number before using it with this function, for example `chr(val("34"))`.

See Also:

[ascii](#)
[asc\(function](#)

CITY(...)

Syntax: CITY(zip)

Description: The `city()` function uses Panorama's optional zip code dictionary to look up the name of a city associated with a zip code.

Parameters: This function has one parameter: `zip`.

zip is a US 5 digit zip code. You can specify the zip code either as a number or as text.

Result: The function returns the name of the city for the zip code. If there is more than one possible name, the function returns the primary zip code name as defined by the US Post Office. If the optional zip code dictionary is not installed, the function will return -- instead of a city name.

Examples: One primary use for the `city()` function is to enter the city automatically when the zip code is entered, saving keystrokes and reducing the probability of data entry errors. This example assumes that the database has a `ZipCode` field that contains text. The example uses a text funnel to strip off any extra characters in the zip code (for example 9 digit zip codes).

```
City=city(ZipCode[1,5])  
State=state(ZipCode[1,5])
```

Another use for the `city()` function is to double-check data entry. This example locates all records where the zip code does not match the city name.

```
select City#city(ZipCode[1,5])
```

Errors: This function does not generate any error message. However, if the zip code dictionary is not installed the function will always return -- instead of a city name.

See Also: [state\(\)](#) function
[county\(\)](#) function

CLEAR

- Syntax:** CLEAR
- Description:** The **clear** statement will delete the contents of the current field and not place the deleted data or picture on the clipboard.
- Parameters:** This statement has no parameters.
- Action:** This statement removes the data or picture from any type of field and, unlike the [cutcell](#) statement, will not place the deleted data on the clipboard. This statement will only affect the currently active field (or cell).
- The clear statement does the same job as the [clearcell](#) statement, but will only work in the Data Sheet and Design Sheet.
- This statement has the same effect as choosing the **Clear** command from the **Edit** menu.
- Examples:** This simple example will remove the contents of the active field leaving the clipboard as is.

```
clear
```

This example checks all selected records one by one and clears the Name field only if the Status field contains the word old for that record.

```
firstrecord
loop
  field Name
  if Status contains "old"
    clear
  endif
downrecord
until info("stopped")
```

Views: This statement may be used in the Data Sheet and Design Sheet views **only**.

See Also: [clearcell](#) statement
[clearrecord](#) statement
[copy](#) statement
[copycell](#) statement
[copyrecord](#) statement
[cutstatement](#)
[cutcell](#) statement
[cutrecord](#) statement
[deleteabove](#) statement
[deleteall](#) statement
[deleterecord](#) statement
[paste](#) statement
[paste cell](#) statement
[pasterecord](#) statement

CLEARCELL

Syntax: CLEARCELL

Description: The `clearcell` statement will delete the contents of the current field and not place the deleted data or picture on the clipboard.

Parameters: This statement has no parameters

Action: This statement removes the data or picture from any type of field and, unlike the [cutcell](#) statement, will not place the deleted data or picture on the clipboard. This statement will only affect the currently active field (or cell).

This statement does the same job as the [clear](#) statement, but will work in all views.

This statement has the same effect as choosing the **Clear** command from the **Edit** menu.

Examples: This simple example will remove the contents of the active field, leaving the clipboard as is.

```
clearcell
```

This example checks all selected records one by one and clears the Name field only if the Status field contains the word old for that record.

```
firstrecord
loop
  field Name
  if Status contains "old"
    clearcell
  endif
downrecord
until info("stopped")
```

Views: This statement may be used in any view.

See Also: [clear](#) statement
[clearrecord](#) statement
[copy](#) statement
[copycell](#) statement
[copyrecord](#) statement
[cut](#) statement
[cutcell](#) statement
[cutrecord](#) statement
[deleteabove](#) statement
[deleteall](#) statement
[deleterecord](#) statement
[paste](#) statement
[pasteccell](#) statement
[pasterecord](#) statement

CLEARMENU MARKS

Syntax: CLEARMENU MARKS menu

Description: The `clearmenu marks` statement clears all the marks in an entire menu. You could do this by using `setmenumark` over and over again, but `clearmenu marks` is much faster.

Parameters: This statement has one parameter: menu.

menu is the name or ID number of the menu you want to clear. The menu ID is assigned in ResEdit.

Action: This statement clears all the marks (checkmarks, diamonds, etc.) from an entire menu. (Note: **Only custom menus can be cleared.** You cannot clear Panorama's standard menus or the Action menu.)

Examples: This example changes the baud rate, and marks the new rate in the **Baud** menu with a diamond.

```
Rate=9600
clearmenu marks "Baud"
setmenumark "Baud", str(Rate),chr(19)
```

Views: This statement may be used in any view that has custom menus installed.

See Also: [setmenumark](#) statement
[getmenumark](#) statement
[setmenutext](#) statement
[getmenutext](#) statement
[menudisable](#) statement
[menuenable](#) statement
[menubuild](#) statement
[getmenus](#) statement
[setmenus](#) statement

CLEARRECORD

Syntax: CLEARRECORD

Description: The `clearrecord` statement will **delete** the currently selected record and not place the deleted record on the clipboard.

Parameters: This statement has no parameters.

Action: This statement removes the entire active record from the database and, unlike the [deleterecord](#) statement or [cutrecord](#) statement, will not place the deleted record on the clipboard. After deleting the record all other records move up one row in the window and the cursor will reside on the next visible record immediately following the deleted record. On a single record view form the display will change to the next visible record after the deleted record.

This statement will delete the currently active record unless there is only one record selected, then it will do nothing except cause Panorama to beep once. The beep is a warning because you can never delete the last visible record from any Panorama database.

Examples: This simple example will remove the active record from the database leaving the clipboard as is.

```
clearrecord
```

This example checks all selected records one by one and deletes the record when the Status field contains the word old for that record. If only one record remains and it is an old record the procedure will warn you that you cannot delete it.

```
firstrecord
loop
  if Status contains "old"
    if info("selected") = 1
      beep
      message "You cannot delete the last record."
      stop
    else
      clearrecord
    endif
  endif
  downrecord
until info("stopped")
```

Views: This statement may be used in any view

See Also: [clear](#) statement
[clearcell](#) statement
[copy](#) statement
[copycell](#) statement
[copyrecord](#) statement
[cut](#) statement
[cutcell](#) statement
[cutrecord](#) statement

deleteabove statement
deleteall statement
deleterecord statement
paste statement
pastecell statement
pasterecord statement

CLIPBOARD(...)

Syntax: CLIPBOARD()

Description: The `clipboard()` function returns whatever text is currently on the Macintosh clipboard.

Parameters: This function has no parameters.

Result: This function returns whatever text is currently on the Macintosh clipboard.

Examples: Here's an example that grabs a name from the clipboard and selects all the records containing that name:

```
local findThis
findThis=clipboard()
select Name contains findThis
```

This example copies the contents of the clipboard into a variable before using it in the `select` statement. This is not absolutely necessary—in fact this procedure could have been written in a single line like this:

```
select Name contains clipboard()
```

However, the original procedure will be much faster. Because of the overhead involved in accessing the operating system, accessing the clipboard is much slower than accessing a variable. If you're only going to be accessing the clipboard a few times, by all means use it directly. But if you are going to access the clipboard over and over again (as the `select` statement does) it's much better to copy the value into a variable first.

Errors: This function does not produce any errors.

See Also: [scrapcalc](#) statement
[cut](#) statement
[copy](#) statement
[cutcell](#) statement
[copycell](#) statement

CLIPTOPICTFILE

- Syntax:** `CLIPTOPICTFILE filename`
- Description:** The `cliptopictfile` statement converts a PICT image on the clipboard into a PICT file on disk.
- Parameters:** This statement has one parameter: `filename`
- filename** is a text string that will be used as the file name for the PICT file when it is created on disk. This parameter can either be quoted text, a variable or field that contains the name of the file you wish to create, or a formula that results in the PICT file name.
- This string may also be a full path name if you intend to write the PICT file to a different folder or disk than the location of the Panorama database the image came from. It may also be a variable or field name containing the path name or a formula which results in the path name.
- Action:** This statement will cause a generic file icon to appear on the disk you are writing to which contains the PICT image from the clipboard in Panorama. The created file will have a Type = **PICT** and a Creator = **KAS1**. This means the file is a PICT image file created by the Panorama application, but it is not a Panorama database.) This statement will return the alert message **"The clipboard doesn't contain a picture."** if the clipboard is empty.
- Examples:** This example creates a PICT file called Logo from the PICT image on the clipboard (if any). Panorama will put it in the same folder as the database that controls this procedure.
- ```
cliptopictfile "Logo"
```
- This example will copy a picture from a picture type field called Photo and create a PICT file with a name that is the contents of the PhotoName field. It will create the PICT file in a folder called pictures on the disk called **mac 660**. (Path name strings are not case sensitive.)
- ```
field «Photo»
copycell
cliptopictfile "mac 660:pictures:"+«PhotoName»
```
- Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.
- See Also:** [clipboard\(\)](#) function
[copycell](#) statement
[pastecell](#) statement

CLOSEFILE

Syntax: CLOSEFILE

Description: The `closefile` statement will close the currently active database.

Parameters: This statement has no parameters

Action: This statement closes the entire database associated with the active window when the statement is triggered. As a result all windows for this database will also close.

Note: Closing the database does not stop the procedure. If the procedure has additional steps it will continue running. For example a procedure can close the current database and then open another one.

Warning: This statement will **not** automatically save the database it is closing unless two conditions are met: 1. The file being closed has been changed **and** 2. `closefile` is the last procedure statement. Note: If the database being closed is in a Design Sheet window which has changed Panorama **will always** ask for a New Generation prior to closing the file.

Examples: This example closes the database connected to the currently active window and will prompt you to save if the database has changed since its last save.

```
closefile
```

This nearly identical example closes the database connected to the currently active window and will never save the database being closed.

```
closefile
nop
```

This example attempts to make the Data Sheet for the database Archives active, verifies that Archives is active, asks if you would like to save the database, and then closes that database. If Archives is not found it will do nothing.

```
window "Archives"
if info("databasename") = "Archives"
  yesno "Save this file?"
  if clipboard() contains "yes"
    save
  endif
  closefile
endif
```

Views: This statement may be used in any view even if there are no visible windows open.

See Also: [closewindow](#) statement
[info\("changes"\)](#) function
[info\("databasename"\)](#) function
[info\("formname"\)](#) function
[info\("windowname"\)](#) function
[gocrosstab](#) statement

godesignsheet statement
goform statement
gosheet statement
opencrosstab statement
opendesignsheet statement
opendialog statement
openfile statement
openform statement
opensheet statement
window statement

CLOSERESOURCE

- Syntax:** CLOSERESOURCE file
- Description:** The `closeresource` statement closes a resource file that has been opened with the [openresource](#) statement.
- Parameters:** This statement has one parameter: `file`.
`file` identifies the resource file you want to close.
- Action:** This statement closes a resource file that has been opened with the [openresource](#) statement. (If this file has not been previously opened with the [openresource](#) statement, the `closeresource` statement will have no effect.)

If a resource file is open, it cannot be edited with a program resource editor program like ResEdit. Closing the resource file allows the resources within the file to be modified with such a program.
- Examples:** The example procedure below closes the resource file My Menus. (**Warning:** Once the resource file is closed, any custom menus based on the resource will no longer function.)


```
closeresource "My Menus"
```
- Views:** This statement may be used in any view.
- See Also:** [openresource](#) statement
[getresource\(\)](#) function
[getString\(\)](#) function
[getnstring\(\)](#) function
[getStringmatch\(\)](#) function
[resources\(\)](#) function
[resourcetypes\(\)](#) function

CLOSESOUND

Syntax: CLOSESOUND

Description: The `close` statement will close the currently open sound resource if one is open.

Parameters: This statement has no parameters.

Action: This statement closes the sound resource currently opened by an `opensound` statement. When the sound resource is closed it is also removed from Scratch Memory.

A sound resource contains one or more SND resource for each sound and each SND resource is given a name. Sound resources may be created with a sound program like Farallon's MacRecorder or shareware programs like Sound->snd or SoundMover. System 7 users may even create sound resources with the Sound control panel and a Macintosh which supports a microphone.

Examples: This example opens a sound resource called Bird Calls, plays the sound titled Robin and then closes Bird Calls.

```
opensound "Bird Calls"
playsound "Robin"
closesound
```

This example asks the user a question to answer, opens a sound resource called Remarks, and depending on the answer plays the sound called **Correct** or **Wrong**. It then closes Remarks.

```
local Answer,Reply
message "The Alamo is in:"+¶+"1. New Mexico"+
  ¶+"2. Texas"+¶+"3. California"
gettext "Is the answer 1, 2, or 3",Answer
opensound "Remarks"
case Answer = "1"
  Reply = "Wrong"
case Answer = "2"
  Reply = "Correct"
case Answer = "3"
  Reply = "Wrong"
endcase
playsound Reply
closesound
```

Views: This statement may be used in any view even if there are no visible windows open.

See Also: [sound](#) statement
[playsound](#) statement

CLOSEWINDOW

Syntax: CLOSEWINDOW

Description: The `closewindow` statement will close the currently active window, even if the window is not visible.

Parameters: This statement has no parameters.

Action: This statement closes the currently active window for any database. The window affected may be from any area in the database or even a form opened as a dialog.

Warning: If only one window is open for the database, the database will close as soon as the window closes. When this happens Panorama will ask you to save changes if:

1. The file being closed has been changed **and**
2. `closewindow` is the last procedure statement.

This statement has the same effect as choosing the **Close Window** command from the **File** menu or clicking on the closebox for any window.

Examples: This example assumes the window entitled `Budget:Monthly Budget` is open. It makes the window active and closes that window. If `Budget:Monthly Budget` is the only window open for the Budget database and that database has changed Panorama will ask if you wish to save the file.

```
window "Budget:Monthly Budget"
closewindow
```

This nearly identical example closes the same window as the previous example. However the `nop` statement will suppress the Save dialog if `Budget:Monthly Budget` is the only window open for the Budget database and that database has changed. Budget will close unchanged.

```
window "Budget:Monthly Budget"
closewindow
nop
```

This example attempts to go to the Chart form window in the database called Taxes, checks to see if it is active, and if so, closes Chart only.

```
window "Taxes:Chart"
if info("windowname") contains "chart"
    closewindow
endif
```

Views: This statement may be used in any view even if there are no visible windows open.

See Also: [closefile](#) statement
[info\("changes"\)](#) function
[info\("databasename"\)](#) function
[info\("formname"\)](#) function
[info\("windowname"\)](#) function

gocrosstab statement
godesignsheet statement
goform statement
gosheet statement
opencrosstab statement
opendesignsheet statement
opendialog statement
openfile statement
openform statement
opensheet statement
window statement

COLLAPSE

Syntax: COLLAPSE

Description: The **collapse** statement hides the detail records associated with the currently active summary record.

Parameters: This statement has no parameters.

Action: This statement makes all records associated with the currently active summary record that have a level lower than that summary record to become invisible, however they still remain in your database.

If the active record is not a summary record this statement is ignored by Panorama.

The visible record count will accurately change after a proper collapse is performed.

This statement has the same effect as clicking on the **Collapse** tool on a tool palette (when available).

Examples: This example totals a numeric field called Expenses and writes the answer to a summary record created by the total statement, then goes to that summary record, and collapses the database to see just that summary record.

```
field Expenses
total
lastrecord
collapse
```

This example groups a database by the GL Category field, totals the Balance field and then checks record by record, collapsing the records when it finds a level 1 summary record only.

```
field «GL Category»
groupup
field Balance
total
firstrecord
loop
  if info("summary") =1
    collapse
  endif
  downrecord
until info("stopped")
```

Note: The previous example could be simplified and made to run faster by using the outlinelevel statement.

```
field «GL Category»
groupup
field Balance
total
outlinelevel 1
```

This example uses the **collapse**, **expand**, and **expandall** statements in a procedure to calculate percentages of Balance values for each data record associated with a specific summary 1 record for that group.

```

local BalTotal
field «GL Category»
groupup
field Balance
total
outlinelevel 1
firstrecord
field Percentage
loop
  if info("summary") =1
    BalTotal = Balance
    expand
    formulafill (Balance/BalTotal)*100
    collapse
  endif
  downrecord
until info("eof") or info("summary") >1
lastrecord
fill Percentage
fill "100"
Percentage = zeroblank(0)
expandall

```

Views: This statement may be used in the Data Sheet or Form views only











See Also:

- [average](#) statement
- [count](#) statement
- [expand](#) statement
- [expandall](#) statement
- [group](#) statement
- [groupdown](#) statement
- [groupup](#) statement
- [info\("records"\)](#) function
- [info\("selected"\)](#) function
- [info\("summary"\)](#) function
- [maximum](#) statement
- [minimum](#) statement
- [outlinelevel](#) statement
- [removesummaries](#) statement
- [selectsummaries](#) statement
- [total](#) statement

COLORS

Background: We think of colors as the spectrum of the rainbow, but the computer builds up all colors from just three: red, green, and blue. By varying the relative intensity of these three colors the computer can generate all the colors of the rainbow. A Panorama color data item combines red, green, and blue intensity values into a single raw binary data item (like other binary data, colors are stored as text items).

RGB Color: The Macintosh measures color intensity on a scale from 0 (completely dark) to 65,535 (full brightness). Values in between denote intermediate intensity. By combining the three primary colors with different intensities you can create virtually any color. The table below shows a small sample of the colors that are possible.

RED	GREEN	BLUE	COLOR	SAMPLE
0	0	0	Black	
65535	65535	65535	White	
15000	15000	15000	Dark Gray	
45000	45000	45000	Light Gray	
65535	0	0	Pure Red	
0	65535	0	Pure Green	
0	0	65535	Pure Blue	
65535	0	65535	Purple	
65535	65535	0	Yellow	
0	65535	65535	Cyan	
3441	4276	32336	Dark Blue	
39235	30211	30211	Brown	
24367	23356	31931	Light Green	
65535	23356	2936	Orange	

Using Color: A color in a field or variable is just a piece of data that describes a color...you can't actually see the color. However, some SuperObjects allow you to control their color using a color data item, and you can look at or modify the color of any graphic object in a form. See [changeobjects](#) and [objectinfo\(\)](#) for more information.

HSB Color: Another way to specify a color is the HSB, or Hue, Saturation, Brightness system. Like the RGB system, the HSB system uses three numbers from 0 to 65,535 to describe a color. However, the three components have different meanings in this system.

The **Hue** component specifies where this color falls in the spectrum. If you are familiar with the standard Apple color picker, the Hue would specify the angle of the color from the center of the wheel.

The **Saturation** component refers to how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? Again using the standard Apple color picker, the Saturation would specify the distance of the color from the center of the wheel.

The **Brightness** component refers to how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does. On the Apple color picker the Brightness is specified by the scroll bar on the right.

See Also: [rgb\(\)](#) function
[hsb\(\)](#) function
[red\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement

COLORWHEEL

- Syntax:** `COLORWHEEL prompt,color`
- Description:** The `colorwheel` statement allows the user to pick a color using the operating system's standard color picker dialog (see also [colors](#)).
- Parameters:** This statement has two required parameters: `prompt` and `color`.
- `prompt`** is a message that will be displayed in the color picker dialog. The prompt should be a text item like "Background Color" or "Negative Highlight Color".
- `color`** must be a variable or text field. The color the user picks (a six byte [binary data](#) value) will be placed into the field or variable.
- Action:** This statement displays the color picker dialog. The user can pick a color, then press the **Ok** or **Cancel** buttons (the procedure can tell what button was pressed by using the [info\("dialogtrigger"\)](#) function).
- Examples:** The procedure below lets the user pick a color, then changes the color of foreground objects to that color. (Note: For this example, "foreground" objects are defined as objects that contain the word foreground in the object name.)
- ```

local foreColor
colorwheel "Foreground Color",foreColor
if info("dialogtrigger") contains "Cancel"
 stop
endif
selectobjects objectinfo("name") contains "Foreground"
changeobjects "color",foreColor
selectnoobjects

```
- Views:** This statement may be used in any view.
- See Also:** [colors](#)  
[rgb\(\)](#) function  
[hsb\(\)](#) function  
[red\(\)](#) function  
[green\(\)](#) function  
[blue\(\)](#) function  
[hue\(\)](#) function  
[saturation\(\)](#) function  
[brightness\(\)](#) function  
[objectinfo\(\)](#) function  
[changeobjects](#) function



# CONVERTIMAGE

**Syntax:** CONVERTIMAGE input,output,type,height,width

**Description:** The `convertimage` statement can convert an image file from one format to another (for example from PICT to JPEG or JPEG into PICT) and can also adjust the size of an image. This statement requires the optional Enhanced Image Pack, which must be purchased separately. It also requires the Apple Quicktime Version 4 (or later) be installed. You can download QuickTime from [www.apple.com](http://www.apple.com), and it is also on the Panorama CD.

**Parameters:** This statement has five parameter: `input`, `output`, `type`, `height` and `width`.

**input** specifies the original image file. If the image file is in the same folder as the currently active database then only the file name is required (for example `Cool Sunset.jpg`). If the image file is in a different folder then both the folder and file name must be included (for example `D:\Photography\Cool Sunset.jpg`). (Note: The original image file may be a GIF file, but the output file cannot be.)

**output** specifies the new, converted image file. If you want to put this new image file in the same folder as the current database then only the file name is required, if you want to put it in a different folder then both the folder and file name must be included. If a file with this name already exists in this location it will be erased.

**type** specifies the type of image that will be created. If the output file has an extension (for example `.jpg`, `.pct`, `.tif`) you should leave this parameter blank ("" ) and let Panorama automatically figure out the type. If the output file does not have an extension you must specify the type from the list below.

| Image Type | PC Extensions | Notes                                |
|------------|---------------|--------------------------------------|
| PICT       | .pct          | Apple PICT bitmap                    |
| BMP        | .bmp          | Windows and OS/2 bitmap              |
| JPEG       | .jpg .jpeg    | JPEG compressed image                |
| PNG        | .png          | Portable Network Graphics bitmap     |
| TIFF       | .tif .tiff    | Does not support LZW compressed TIFF |
| TARGA      | .targa        |                                      |
| PHOTOSHOP  | .psg          | Adobe Photoshop                      |
| FLASHPIX   | .fpx          | FlashPix bitmap                      |

**height** specifies the new height of the image. If this is zero the original height will be used.

**width** specifies the new width of the image. If this is zero the original width will be used.

**Action:** This statement allows a procedure to extract and look at the text (source code) of other procedures. This statement is disabled if the user is not authorized to see the contents of the procedure.

**Examples:** This example that converts a BMP image into a TIFF image. Since the output file has an extension (`.tif`) the output image type does not need to be specified. The TIFF image will have the same dimensions as the original PICT image.

```
convertimage "my picture.bmp","my picture.tif","",0,0
```

This example converts an image into a 32 by 32 pixel icon. Since the files do not have any extensions this example can only work on a Macintosh, not on a Windows PC.

```
convertimage "my picture", "my icon", "PICT", 32, 32
```

Here is a similar example that can work on a Windows PC (it can work on a Macintosh also, if the file names have extensions).

```
convertimage "my picture.jpg", "my icon.jpg", "", 32, 32
```

**Views:** This statement may be used in any view.

**See Also:** [imagequality](#) statement

# COPY

- Syntax:** COPY
- Description:** The `copy` statement will duplicate the contents of the current field onto the clipboard, replacing anything previously on the clipboard.
- Parameters:** This statement has no parameters.
- Action:** This statement duplicates the data or picture from any type of field to the clipboard. This statement will only affect the currently active field (or cell).
- Note: When you **copy** from a date type field the date will transcribe to the clipboard in the format of `MM/DD/YYYY` regardless of the date field's established output pattern.
- The `copy` statement does the same job as the `copycell` statement, but will **not** work correctly in a cross tab window.
- This statement has the same effect as choosing the **Copy** command from the **Edit** menu.
- Examples:** This simple example will duplicate the contents of the active field into the clipboard.

```
copy
```

This example copies the value in the Qty field into the clipboard in order to use that value in a cost calculation for that record; turning the clipboard into a variable. The answer is stored in the field Cost.

```
field Qty
copy
Cost = «Price» * val(clipboard())
```

This example checks all selected records one by one and copies and clears the Name field into the clipboard only if the Status field contains the word old for that record. The copied name is pasted into Old Name.

```
firstrecord
loop
 field Name
 if Status contains "old"
 copy
 clear
 field «Old Name»
 paste
 endif
downrecord
until info("stopped")
```

- Views:** This statement may be used in the Data Sheet, Design Sheet and form views **only**.

- See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copycell](#) statement

copyrecord statement  
cut statement  
cutcell statement  
cutrecord statement  
deleteabove statement  
deleteall statement  
deleterecord statement  
paste statement  
pastecell statement  
pasterecord statement

# COPYCELL

**Syntax:** COPYCELL

**Description:** The `copycell` statement will duplicate the contents of the current field onto the clipboard, replacing anything previously on the clipboard.

**Parameters:** This statement has no parameters.

**Action:** This statement duplicates the data or picture from any type of field to the clipboard. This statement will only affect the currently active field (or cell).

**Note:** When you copy from a date type field the date will transcribe to the clipboard in the format of `MM/DD/YYYY` regardless of the date field's established output pattern.

The `copycell` statement does the same job as the `copy` statement, but will work correctly in all views.

This statement has the same effect as choosing the **Copy** command from the **Edit** menu.

**Examples:** This simple example will duplicate the contents of the active field into the clipboard.

```
copycell
```

This example copies the value in the Qty field into the clipboard in order to use that value in a cost calculation for that record; turning the clipboard into a variable. The answer is stored in the field Cost.

```
field Qty
copycell
Cost = «Price» * val(clipboard())
```

This example checks all selected records one by one and copies and clears the Name field into the clipboard only if the Status field contains the word old for that record. The copied name is pasted into Old Name.

```
firstrecord
loop
 field Name
 if Status contains "old"
 copycell
 clear
 field «Old Name»
 paste
 endif
downrecord
until info("stopped")
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copyrecord](#) statement

cut statement  
cutcell statement  
cutrecord statement  
deleteabove statement  
deleteall statement  
deleterecord statement  
paste statement  
pastecell statement  
pasterecord statement

# COPYFORM

**Syntax:** COPYFORM

**Description:** The `copyform` statement copies the current form onto the clipboard.

**Parameters:** This statement has no parameters

**Action:** This statement copies the entire current form onto the clipboard. This includes the graphic objects on the form, the page setup, the custom menu setup, and form and report preferences. Using the `copyform` and `pasteform` statements creates an exact duplicate of a form.

This statement has the same effect as choosing the **Copy Form** command from the **Edit** menu (graphics mode).

**Examples:** This example will make a copy of the form **Report** Template. The `pasteform` statement will stop and ask the user what to call the new form.

```
openform "Report Template"
copyform
pasteform
```

**Views:** This statement may be used in a Form view.

**See Also:** [pasteform](#) statement  
[openform](#) statement  
[copy](#) statement  
[cut](#) statement  
[paste](#) statement

# COPYRECORD

**Syntax:** COPYRECORD

**Description:** The `copyrecord` statement will duplicate the contents of the current record onto the clipboard, replacing anything previously on the clipboard. The record on the clipboard will be tab delimited fields.

**Parameters:** This statement has no parameters.

**Action:** This statement duplicates the data from all fields except picture type fields onto the clipboard for the current record only. Panorama will leave a blank space for each picture field, so field order will not be disturbed.

**Note:** Data in date type fields will transcribe to the clipboard in the format of `MM/DD/YY` regardless of the date field's established output pattern. Numeric field's output patterns are ignored.

This statement has the same effect as clicking on the **Copy Record** tool on a tool palette (when available).

**Examples:** This simple example will duplicate the contents of the active record onto the clipboard separating each field's value with a tab character.

```
copyrecord
```

This example copies the current record from the Current Records database, goes to the Old Records database, pastes the copy into that file, and returns to Current Records.

```
copyrecord
window "Old Records"
pasterecord
window "Current Records"
```

**Views:** This statement may be used in any view

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[cutcell](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement



# COS(...)

**Syntax:** COS(angle)

**Description:** The `cos()` function calculates the cosine of an angle.

**Parameters:** This function has one parameter: `angle`.

**angle** is a numeric value, an angle. The angle is usually specified in a mathematical unit of measurement called radians, however, within a procedure you can temporarily force Panorama to use degrees (see below). One radian is equal to approximately 57.2958 degrees (the exact value is  $180/\pi$ ).

**Result:** The result of this function is a numeric floating point value between -1 and 1.

**Examples:** The graph below shows the result of the cosine function given input values from 0 to +20 radians.



This formula calculates the cosine of an angle in degrees.

```
cos(Angle*180/π)
```

In this example the angle is in a field or variable named `Angle`, however, you may use any formula that produces a numeric result in this location. (On the Macintosh the pi symbol ( $\pi$ ) is produced by pressing **Option-P**. On Windows systems the pi symbol is created by pressing **Alt-0254**.)

Here is another way to calculate angles in degrees in a procedure.

```
degree
NewAngle=cos(Angle)
```

The `degree` statement tells Panorama to use degrees instead of radians in all trigonometry calculations. Panorama will continue to use degrees until the end of the procedure, or until a `radian` statement is encountered.

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `cos("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating the cosine, for example `cos(val("34"))`.

**See Also:** [sin\( function](#)  
[tan\( function](#)  
[degree statement](#)  
[radian statement](#)  
[arcsin\( function](#)  
[arccos\( function](#)  
[arctan\( function](#)  
[val\( function](#)

# COSH(...)

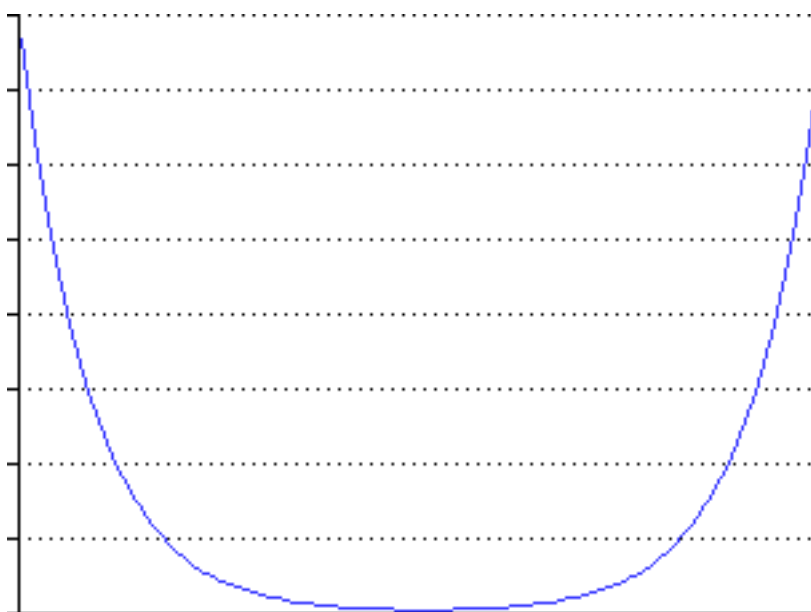
**Syntax:** COSH(value)

**Description:** The `cosh()` function calculates the hyperbolic cosine of a numeric value.

**Parameters:** This function has one parameter: value.  
**value** is a numeric value.

**Result:** The result of this function is a numeric floating point value.

**Examples:** The graph below shows the result of the hyperbolic cosine function given input values from -6 to +6.



**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `cosh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the hyperbolic cosine, for example `cosh(val("34"))`.

**See Also:** [sinh\(\)](#) function  
[tanh\(\)](#) function  
[arcsinh\(\)](#) function  
[arccosh\(\)](#) function  
[arctanh\(\)](#) function  
[val\(\)](#) function

# COUNT

**Syntax:** COUNT

**Description:** The count statement counts all non-empty data cells for the current field.

**Parameters:** This statement has no parameters.

**Action:** This statement counts the non-empty entries for the current field. The current field may be numeric, text, or a choices type field, however you **cannot** count a date or picture type field because the count cannot be formatted for those field types.

If the database contains summary records, this statement will perform counts for each group of records placing the count in the summary record for that group. It will also place an overall count for the whole database in the grand total summary.

If there are not any summary records in the database, one will be added at the end of the database and the overall count is placed into that summary record. This statement has the same effect as choosing the **Count** command from the **Math** menu.

**Examples:** This simple example counts the entries in the Name field.

```
field Name
count
```

This example groups up on the State field, copies the State field into a newly created field called Temp and counts the State field and collapses the database to show you only the summary records.

```
field State
groupup
insertfield "Temp"
field Temp
formulafill State
count
outlinelevel 1
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [group](#) statement  
[groupdown](#) statement  
[groupup](#) statement  
[maximum](#) statement  
[minimum](#) statement  
[total](#) statement

# COUNTY(...)

**Syntax:** COUNTY(zip)

**Description:** The `county()` function uses Panorama's optional zip code dictionary to look up the name of a county associated with a zip code.

**Parameters:** This function has one parameter: `zip`.

**zip** is a US 5 digit zip code. You can specify the zip code either as a number or as text.

**Result:** The function returns the name of the county for the zip code. If the optional zip code dictionary is not installed, the function will return -- instead of a county name.

**Examples:** One primary use for the `county()` function is to enter the state automatically when the zip code is entered, saving keystrokes and reducing the probability of data entry errors. For example, you might need to know the county to calculate sales tax. This example assumes that the database has a `ZipCode` field that contains text. The example uses a text funnel to strip off any extra characters in the zip code (for example 9 digit zip codes).

```
Country=county(ZipCode[1,5])
```

Another use for the `county()` function is to double-check data entry. This example locates all records where the zip code does not match the county name.

```
select County≠county(ZipCode[1,5])
```

**Errors:** This function does not generate any error message. However, if the zip code dictionary is not installed the function will always return -- instead of a county name.

**See Also:** [city\(\)](#) function  
[state\(\)](#) function

# CUSTOMALERT

**Syntax:** CUSTOMALERT resource#

**Description:** The `customalert` statement allows a Panorama procedure to open an preexisting Panorama alert dialog in a format you have customized using a resource editing program like ResEdit or Resourcerer™.

**Parameters:** This statement has one parameter: resource#

**resource#** is the number that identifies an alert resource. The resource for the alert must be created using a resource editing program. The alert resource must have the same items in the same order as the preexisting alert you wish to emulate, but the sizes, locations and text may be different. You may also add additional static text and/or picture items to the alert as long as they **are at the end** of the item list.

**Warning:** Specifying a resource# that is undefined or contained in an unopened resource will cause Panorama to crash to the Finder.

**Action:** You must use the `customalert` statement immediately before one of the following statements:

[cancelok](#)

[noyes](#)

[okcancel](#)

[yesno](#)

The `customalert` statement allows you to change the appearance of the dialog that opens for the alert statement that follows it in the procedure.

The custom resource must have the same items as the original, but the position of the buttons and the length of text strings can be different.

Alert statements like [yesno](#) and [okcancel](#) that return button names to the clipboard will return the name of the actual button pressed in the custom alert.

**Examples:** In this example a alert resource (resource number 5001) was created in a resource file called Alerts. This resource must have the same items as the `yesno` alert dialog it emulates. The resource file `Alerts` must be opened before it can be used. To open the file use the [openresource](#) statement.

```
openresource "Alerts"
customalert 5001
yesno "Do you want to save?"
if info("dialogtrigger") contains "yes"
 save
endif
```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:** [cancelok](#) statement  
[clipboard\(\)](#) function  
[customdialog](#) statement  
[getscrap](#) statement  
[gettext](#) statement  
[info\("dialogtrigger"\)](#) function  
[message](#) statement  
[noyes](#) statement  
[okcancel](#) statement  
[openresource](#) statement  
[yesno](#) statement

# CUSTOMDIALOG

**Syntax:** CUSTOMDIALOG resource#

**Description:** The **customdialog** statement allows a Panorama procedure to open an preexisting Panorama dialog in a format you have customized using a resource editing program like ResEdit or Resourcer.

**Parameters:** This statement has one parameter: resource#

**resource#** is the number that identifies a dialog resource. The resource for the dialog must be created using a resource editing program. The dialog resource must have the same items in the same order as the preexisting dialog you wish to emulate, but the sizes, locations and text may be different. You may also add additional static text and/or picture items to the dialog as long as they **are at the end** of the item list.

**Warning:** Specifying a resource# that is undefined or contained in an unopened resource will cause Panorama to crash.

**Action:** You must use the customdialog statement immediately before one of the following statements:

```

getscrap
gettext
message

```

The customdialog statement allows you to change the appearance of the dialog that opens for the dialog statement that follows it in the procedure.

The custom resource must have the same items as the original, but the position of the buttons and the length of text strings can be different.

**Examples:** In this example a dialog resource (resource number **6001**) was created in a resource file called Dialogs. This resource must have the same items as the [getscrap](#) dialog it emulates. The resource file Dialogs must be opened before it can be used. To open the file use the [openresource](#) statement.

```

openresource "Dialogs"
customalert 6001
Step1:
getscrap "Enter the start date. ex: 1/12/95"
if date(clipboard()) < today()
 message "Your date must be after todays date."
 goto Step1
endif

```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:** [cancelok](#) statement  
[clipboard\(\)](#) function  
[customalert](#) statement  
[getscrap](#) statement



gettext statement  
info("dialogtrigger") function  
message statement  
noyes statement  
okcancel statement  
openresource statement  
yesno statement

# CUT

- Syntax:** CUT
- Description:** The `cut` statement will delete the contents of the current field and will place the deleted data or picture on the clipboard.
- Parameters:** This statement has no parameters.
- Action:** This statement removes the data or picture from any type of field and places the deleted data on the clipboard. This statement will only affect the currently active field (or cell).  
The `cut` statement does the same job as the `cutcell` statement, but will **not** work correctly in a cross tab window  
This statement has the same effect as choosing the **Cut** command from the **Edit** menu.
- Examples:** This simple example will remove the contents of the active field and puts it on the clipboard.
- ```
cut
```
- This example checks all selected records one by one and cuts the data from the Name field only if the Status field contains the word old for that record.
- ```
firstrecord
loop
 field Name
 if Status contains "old"
 cut
 endif
downrecord
until info("stopped")
```
- Views:** This statement may be used in the Data Sheet, Design Sheet, and Form views **only**.
- See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement

# CUTCELL

- Syntax:** CUTCELL
- Description:** The `cutcell` statement will delete the contents of the current field and places the deleted data or picture on the clipboard.
- Parameters:** This statement has no parameters.
- Action:** This statement removes the data or picture from any type of field and places the deleted data or picture on the clipboard. This statement will only affect the currently active field (or cell).  
 This statement does the same job as the [cut](#) statement, but will work in all views.  
 This statement has the same effect as choosing the **Cut** command from the **Edit** menu.
- Examples:** This simple example will remove the contents of the active field and place it on the clipboard.

```
cutcell
```

This example checks all selected records one by one and cuts the data from the Name field only if the Status field contains the word old for that record.

```
firstrecord
loop
 field Name
 if Status contains "old"
 cutcell
 endif
downrecord
until info("stopped")
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement

# CUTRECORD

**Syntax:** CUTRECORD

**Description:** The `cutrecord` statement will first copy to the clipboard and then delete the currently selected record. There is no difference between this statement and the `deleterecord` statement.

**Parameters:** This statement has no parameters.

**Action:** This statement removes the entire active record from the database after first copying the record to the clipboard. After deleting the record all other records move up one row in the window and the cursor will reside on the next visible record immediately following the deleted record. On a single record view form the display will change to the next visible record after the deleted record.

This statement will delete the currently active record unless there is only one record selected, then it will only copy the record to the clipboard and beep once. The beep is a warning because you can **never** delete the last visible record from any Panorama database.

Note: If this statement is the last procedure statement it will present you with a **Delete/Cancel** alert dialog saying `About to delete current record.`, otherwise this alert is suppressed.

This statement has the same effect as clicking on the **Cut Record** tool on a tool palette (when available). Holding down the **Option** key while clicking on this tool will suppress the alert.

**Examples:** This simple example will copy to the clipboard and remove the active record from the database after alerting you to the action.

```
cutrecord
```

This nearly identical example will copy to the clipboard and remove the active record from the database, but will suppress the alert.

```
cutrecord
nop
```

This example checks all selected records in the Stock database one by one and deletes a record when the Status field contains the word old for that record. It then pastes the deleted record into the Old Stock database. If only one record remains in Stock and it is an old record the procedure will warn you that you cannot delete it.

```
firstrecord
loop
 if Status contains "old"
 if info("selected") = 1
 beep
 message "You cannot delete the last record."
 stop
 else
 cutrecord
 window "Old Stock"
 pasterecord
```

```
 window "Stock"
 endif
endif
 downrecord
until info("stopped")
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement

# DATAFORK

**Syntax:** DATAFORK

**Description:** The **datafork** statement cancels the effect of the [resourcefork](#) statement.

**Parameters:** This statement has no parameters.

**Action:** This statement has no direct action of its own. However, this statement modifies any file access functions that are used later in the procedure. Instead of accessing the resource fork, Panorama will access the data fork.

**Examples:** This example copies the resource fork of MyResources into the data fork of MyResources.RSR. This .RSR file can be transferred to a PC for use with the PC version of Panorama.

```
local rezdata
resourcefork
rezdata=fileload("", "MyResources")
datafork
filesave " ", "My Resources.RSR", "", rezdata
```

**Views:** This statement may be used in any view.

**See Also:** [resourcefork](#) statement  
[filesave](#) statement  
[fileappend](#) statement  
[fileload\(\)](#) function  
[filesize\(\)](#) function  
[openresource](#) statement  
[openresourcerw](#) statement  
[resources\(\)](#) function  
[resourcetypes\(\)](#) function  
[getresource\(\)](#) function  
[getstring\(\)](#) function  
[getnstring\(\)](#) function

# DATATYPE(...)

**Syntax:** DATATYPE(fieldvariablename)

**Description:** The `datatype()` function determines what kind of data is in a field or variable: text, number, etc.

**Parameters:** This function has one parameter: `fieldvariablename`.

**fieldvariablename** is the name of the field or variable that you want to get information about. If this is the name of a variable it must be surrounded with quotes (").

**Result:** The function returns the type of data from the list below:

```
Text
Compressed
Picture
Date
Floating Point
Integer
Fixed 1 Digit (.#)
Fixed 2 Digits (###)
Fixed 3 Digits (####)
Fixed 4 Digits (#####)
```

Note: The Compressed, Picture, and Date types can only occur if the `datatype()` function is used with a field as the parameter. Variables cannot contain data of these types (for a date, the data type is Integer).

**Examples:** Here's an example which checks the `AreaCode` variable. If it is text, it converts it into a number.

```
if datatype("AreaCode")="Text "
 AreaCode=val(AreaCode)
endif
```

**Errors:** **Field or variable does not exist.** This error occurs if there is no variable or field in the current database with the name you have specified. You probably misspelled the field or variable name or forgot to put the variable name in quotes.

**See Also:** [sizeof\(\)](#) function  
[info\("datatype"\)](#) function

# DATE PATTERNS

**Background:** In Panorama dates are represented as a number which is the number of days since January 1, 4713 B.C. For example, the date August 7, 1991 corresponds to the number 2,448,476. Most people aren't too interested in knowing that a certain day is 2 million plus days from some pre-historic date. Date patterns allow you to control how dates are formatted when they are converted to text. Date patterns can be used in the Design Sheet and by the [datepattern\(\)](#) function.

**Common Patterns:** Panorama date patterns supports a wide variety of date formats. The table below lists several common date patterns.

| Date      | Pattern                      | Display                   |
|-----------|------------------------------|---------------------------|
| 3/9/2002  | mm/dd/yy                     | 3/9/02                    |
| 3/9/2002  | MM/DD/YY                     | 03/09/02                  |
| 3/9/2002  | mm-dd-yyyy                   | 3-9-2002                  |
| 3/9/2002  | dd-MON-yy                    | 9-MAR-02                  |
| 3/9/2002  | dd-Month-yy                  | 9-March-02                |
| 3/9/2002  | Month dd, yyyy               | March 9, 2002             |
| 3/9/2002  | Month ddnth, yyyy            | March 9th, 2002           |
| 3/9/2002  | DayOfWeek, Month ddnth, yyyy | Saturday, March 9th, 2002 |
| 3/9/2002  | qqqyy                        | 1q02                      |
| 3/9/2002  | Week ww of yyyy              | Week 11 of 2002           |
| 5/23/2002 | Quarter "Quarter" yyyy       | Second Quarter 2002       |
| 7/11/2004 | Qtr "Qtr" yyyy               | 3rd Qtr 2004              |
| 3/9/2002  | "Day" dd, "Month" mm         | Day 9, Month 3            |
| 3/1/2002  | ddnth "day of" Month, yyyy   | 1st day of March, 2002    |
| 3/2/2002  | ddnth "day of" Month, yyyy   | 2nd day of March, 2002    |
| 3/9/2002  | ddnth "day of" Month, yyyy   | 9th day of March, 2002    |
| 3/1/1867  | mmnth "month of" yyyy        | 3rd month of 1867         |
| 3/9/1978  | wwnth week of yyyy           | 3rd week of 1978          |



**Components:** There are 15 different basic components that can be used as part of a date pattern. A date pattern is built up by combining these components together with punctuation to build a complete pattern.

| Component | Example   | Description                 |
|-----------|-----------|-----------------------------|
| yy        | 02        | Year (within century)       |
| yyyy      | 2002      | Year (including century)    |
| qq        | 2         | Quarter (numeric)           |
| qtr       | 2nd       | Quarter (abbreviated)       |
| quarter   | second    | Quarter (spelled out)       |
| mm        | 9         | Month (numeric)             |
| MM        | 09        | Month (with leading zero)   |
| mon       | sep       | Month (abbreviated)         |
| month     | september | Month (spelled out)         |
| ww        | 46        | Week (within year)          |
| dd        | 5         | Day (numeric)               |
| DD        | 05        | Day (with leading zero)     |
| day       | tue       | Day Of Week (abbreviated)   |
| dayofweek | tuesday   | Day Of Week (spelled out)   |
| dow       | 3         | Day Of Week (0[sun]-6[sat]) |

**Upper/Lower Case:**

Some of these components (`qtr`, `quarter`, `mon`, `month`, `day`, and `dayofweek`) can be either upper or lower case. The table below shows how a component can be displayed in all lower case, initial caps, or all upper case.

| Pattern   | Display   |
|-----------|-----------|
| month     | september |
| Month     | September |
| MONTH     | SEPTEMBER |
| dayofweek | friday    |
| DayOfWeek | Friday    |
| DAYOFWEEK | FRIDAY    |

**Quoted Literals:**

If you need to include the words `qtr`, `quarter`, `mon`, `month`, or `day` in your date, you must quote them so that they are not treated as components. (Note: the quote key (**Shift-**) is just to the left of the **Return** key. Be sure to use regular quotes, not smart quotes (" not “ ”)). Since the text itself contains quotes, you will need to surround the pattern with another quote character. In the following examples we have used `{ }` as the quote characters surrounding each pattern.

| Pattern                  | Converted Text     |
|--------------------------|--------------------|
| {Quarter "Quarter" YYYY} | First Quarter 1992 |
| {"Day" dd, "Month" mm}   | Day 9, Month 3     |

You can add the suffix `nth` to the `mm`, `ww`, or `dd` components, as shown below.

| Date   | Pattern                 | Converted Text    |
|--------|-------------------------|-------------------|
| 3/1/89 | {ddnth "day of" Month}  | 1st day of March  |
| 3/2/89 | {ddnth "day of" Month}  | 2nd day of March  |
| 3/9/89 | {ddnth "day of" Month}  | 9th day of March  |
| 3/9/89 | {Month ddnth, yyyy}     | March 9th, 1989   |
| 3/9/89 | {mmnth "month of" yyyy} | 3rd month of 1989 |
| 3/9/89 | {wwnth week of yyyy}    | 11th week of 1989 |

Panorama automatically adds the correct suffix.

**See Also:**

[datepattern\(\)](#) function

[date\(\)](#) function

[numeric patterns](#)

# DATE(...)

**Syntax:** DATE(text)

**Description:** The `date()` function converts text into a number representing a date.

**Parameters:** This function has one parameter: text.

**text** is the text that you want to convert to a number representing a date. The text must contain a valid date. Here are some examples of valid dates:

```
5/7/96
August 8, 1957
may 12
today
yesterday
tomorrow
tuesday
last fri
next monday
```

**Result:** This function returns a number representing the date. The number is the number of days since January 1, 4713 B.C. For example, if the date is August 7, 1991 this function will return the number 2,448,476.

**Examples:** The example below asks the user to enter a date, then selects all the records printed on that date (assuming the database has a date field called `PrintDate`). The text entered by the user is converted to a number by the `date()` function before it is compared with the `PrintDate` field.

```
local xDate
xDate=""
gettext "Select invoices printed on what date?",xDate
select PrintDate=date(xDate)
```

If the `date()` function is handed text it cannot interpret as a date, the procedure will stop and an error message is displayed. You can trap this error with the `if error` statement and handle it yourself within the procedure, as you can see in this example.

```
local xDate
xDate=""
gettext "Select invoices printed on what date?",xDate
xDate=date(xDate)
if error
 message "Sorry, invalid date. Please try again."
 stop
endif
select PrintDate=xDate
```

This example has another advantage: the `select` statement will run slightly faster. This is because it does not have to convert the text to date over and over again for each record.

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to convert a numeric value.

**Illegal date.** This error occurs if the text does not contain a valid date that Panorama understands.

**See Also:** [datepattern\(\)](#) function  
[today\(\)](#) function

# DATEPATTERN(...)

**Syntax:** DATEPATTERN(number,pattern)

**Description:** The `datepattern()` function converts a number representing a date into text. The function uses a pattern to control how the date is formatted (see [date patterns](#)).

**Parameters:** This function has two parameters: `number` and `pattern`.

**number** is the number that you want to convert to text. This number is treated as the number of days since January 1, 4713 B.C. (the `date()` function can convert text into such a number).

**pattern** is text that contains a pattern for formatting the date (see [date patterns](#)).

**Result:** This function returns an item of text containing the formatted date.

**Examples:** The `datepattern()` function is useful for displaying dates via a formula in an auto-wrap text object or Text Display SuperObject™. Here is a pattern that will output dates in european format (for example 9-AUG-98)

```
datepattern(OrderDate,"dd-MON-yy")
```

For more information on the variety of patterns that are possible, see [date patterns](#)

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the pattern parameter.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the number parameter.

**See Also:** [date patterns](#)  
[date\(\)](#) function  
[numeric patterns](#)  
[pattern\(\)](#) function

# DAYOFWEEK(...)

**Syntax:** DAYOFWEEK(date)

**Description:** The `dayofweek()` function computes the day of the week (0-6) of a date, with Sunday being 0, Monday 1, etc.

**Parameters:** This function has one parameter: `date`.  
**date** is a number representing the date.

**Result:** The `dayofweek()` function returns a number from 0 to 6. The days of the week are:

```
0 Sunday
1 Monday
2 Tuesday
3 Wednesday
4 Thursday
5 Friday
6 Saturday
```

**Examples:** The procedure below uses the `dayofweek()` function to select all weekday records (Monday through Friday).

```
select
 dayofweek(Date) ≥ 1 and dayofweek(Date) ≤ 5
```

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

**See Also:** [date\(\)](#) function  
[datepattern\(\)](#) function  
[week1st\(\)](#) function

## DBINFO(...)

**Syntax:** DBINFO(option,database)

**Description:** The `dbinfo()` function gets information about a database: what forms it contains, what fields, what flash art pictures, etc.

**Parameters:** This function has two parameters: `option` and `database`.

**option** controls what kind of information this function will retrieve. There are eight possible options:

"fields"

"forms"

"procedures"

"crosstabs"

"flash art"

"permanent"

"folder"

"level"

"autosave"

The "**fields**" option produces a text array (with carriage return separators) containing a list of the fields in the database. (If a field name contains carriage returns, they are converted to spaces before being placed into the array.)

The "**forms**" option produces a text array (with carriage return separators) containing a list of the forms in the database.

The "**procedures**" option produces a text array (with carriage return separators) containing a list of the procedures in the database.

The "**crosstabs**" option produces a text array (with carriage return separators) containing a list of the crosstabs in the database.

The "**flash art**" option produces a text array (with carriage return separators) containing a list of the flash art in the database's Flash Art™ gallery.

The "**permanent**" option produces a text array (with carriage return separators) containing a list of the permanent variables associated with this database. However, the permanent variables will not be listed if the database is in user or custom mode (the database must be in author mode).

The "**folder**" option produces a binary data item that describes the exact location of the folder containing the database. (See the [folder\(\)](#) and [folderpath\(\)](#) functions.)

The "**level**" option returns a number that indicates the privilege level for this database: **0** = author mode, **1** = user mode, or **2** = custom mode.

The "**autosave**" option returns the number the number of minutes between automatic saves, or zero if the auto-save option is turned off. (See also [setautosave](#).)

**database** is the name of the database you want to get information about. This must be a database that is currently open. If you want to get information about the current database you can use the [info\("databasename"\)](#) function or simply use empty text ("").

**Result:** This function returns a text array, number or binary data, depending on the option selected. See the descriptions for each option listed above.

**Examples:** This example displays the number of forms in the current database.

```
message "This database contains "+
 str(arraysize(dbinfo("forms",""),1))+" forms."
```

This example displays the folder the current database is in.

```
message "This database is in the "+
 folderpath(dbinfo("folder",""))[1,-2][":-:",-1][2,-1]+" folder."
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric item for the option or database parameters.

**Illegal info() argument.** This error occurs if the option parameter is not one of the six options listed above.

**See Also:** [folder\(\)](#) function  
[folderpath\(\)](#) function  
[info\("panoramafolder"\)](#) function  
[info\("systemfolder"\)](#) function  
[openform](#) statement  
[goform](#) statement  
[openprocedure](#) statement  
[goprocedure](#) statement  
[opencrosstab](#) statement  
[gocrosstab](#) statement  
[permanent](#) statement



# DEBUG

**Syntax:** DEBUG

**Description:** The **debug** statement stops the current program, allowing you to examine variables or single step.

**Parameters:** This statement has no parameters.

**Action:** This statement stops the current program, allowing you to examine variables or single step. (Note: If the procedure window is not open, it will not stop.)

After the procedure is stopped you may proceed one step at a time by using the Single Step tool (or menu item), or you can continue at full speed using the Proceed tool (or menu item).

You may use as many debug statements as you like to stop the program wherever you want to see what is going on inside your program. If you want the procedure to stop only if a certain condition is met, put the debug statement inside a pair of **if** **endif** statements.

**Examples:** This procedure will stop just after the start of the loop (if the window is open).

```

local X,theChar,aPhone
X=1
aPhone=""
loop
 debug
 theChar=Memo[X;1]
 X=X+1
 stoploopif theChar=""
 repeatloopif theChar≠ "(" and aPhone=""
 aPhone=aPhone+theChar
until aPhone match "(???) ???-????"
message aPhone

```

**Views:** This statement may be used in any view.

**See Also:** [if](#) statement  
[endif](#) statement

# DEFAULTCASE

**Syntax:** DEFAULTCASE

**Description:** The **defaultcase** statement is an optional statement used in a [case](#) construct to define a default that will trigger when all previous cases test false.

**Parameters:** This statement has no parameters.

**Action:** This optional statement may only be used in a [case](#) construct and **must** come after the last [case](#) statement, but before the [endcase](#) statement. If no default is required this statement may be omitted from the case construct.

When a defaultcase is triggered Panorama will execute all statements between the defaultcase and the [endcase](#) statement and then continue on with the procedure.

Using the Check Procedure tool on a procedure that has a defaultcase statement with no corresponding [case](#) statement will result in an **error alert**. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts it's operation.

**Examples:** In this example the defaultcase will ask the user to enter a Rate if the other cases test false, meaning the Shipby method is not one listed in the procedure.

```

case Shipby = "UPS Ground"
 Rate = 5.00
case Shipby = "UPS Blue"
 Rate = 7.50
case Shipby = "UPS Red"
 Rate = 10.00
case Shipby = "Fed-ex standard"
 Rate = 12.00
case Shipby = "Fed-ex overnight"
 Rate = 18.00
case Shipby = "Fed-ex overnight Saturday"
 Rate = 25.00
defaultcase
 getscrap "Enter shipping rate."
 Rate = val(clipboard())
endcase

```

**Views:** This statement may be used in any view.

**See Also:** [case](#) statement  
[else](#) statement  
[endcase](#) statement  
[endif](#) statement  
[if](#) statement

# DEFINE

**Syntax:** DEFINE variable,formula

**Description:** The **define** statement performs an assignment, much like an equals sign or the set statement. However, the **define** statement only performs the assignment if the variable is currently undefined. If the variable already has a value, the define statement leaves it alone. The **define** statement is especially useful for initializing [permanent](#) variables.

**Parameters:** This statement has two parameters: **variable** and **formula**.

**variable** is the name of the field or variable that you want to modify.

**formula** calculates the value that will be placed into the destination.

**Action:** This statement defines a value for a variable, unless that variable already has a value. In other words, this statement will initialize the variable if the variable's value has not been defined yet, but if the variable already has a value it will not touch the value.

**Examples:** The procedure below will initialize the variables DefaultAreaCode and TaxRate unless they have already been initialized.

```
permanent DefaultAreaCode,TaxRate
define DefaultAreaCode,"714"
define TaxRate,4.25
```

In other words, if some other procedure has already assigned the DefaultAreaCode as **323** and the TaxRate as **8.25**, this procedure will not change those values. But if the DefaultAreaCode and TaxRate have never been set, they will be initialized to **714** and **4.25**.

**Views:** This statement may be used in any view.

**See Also:** [set statement](#)  
[formulacalc statement](#)  
[info\("globalvariables"\) function](#)

# DEGREE

**Syntax:** DEGREE

**Description:** The **degree** statement tells Panorama that all values in trigonometric functions should be treated as degree's rather than radian's (the program's default).

**Parameters:** This statement has no parameters.

**Action:** This statement doesn't perform any visible action on its own. However, all attributes for any trig functions in the procedure will be treated as degree values. Panorama will revert back to radian after the procedure is finished; radian is the default setting

**Examples:** This example tells Panorama to calculate the tan of 30 degrees, not 30 radians, and put the answer in Height.

```
degree
Height = tan(30)
```

**Views:** This statement may be used in any view.

**See Also:** [arccos\(\)](#) function  
[arccosh\(\)](#) function  
[arcsin\(\)](#) function  
[arcsinh\(\)](#) function  
[arctan\(\)](#) function  
[arctanh\(\)](#) function  
[cos\(\)](#) function  
[cosh\(\)](#) function  
[radian](#) statement  
[sin\(\)](#) function  
[sinh\(\)](#) function  
[tan\(\)](#) function  
[tanh\(\)](#) function

# DELETEABOVE

**Syntax:** DELETEABOVE

**Description:** The `deleteabove` statement will first copy to the clipboard, then delete the current record, and then move the cursor to the record above the deleted record was

**Parameters:** This statement has no parameters

**Action:** This statement removes the entire active record from the database after first copying the record to the clipboard and then moves the cursor to the record directly above where the deleted record used to be.

This statement will delete the currently active record unless there is only one record selected, then it will only copy the record to the clipboard and beep once. The beep is a warning because you can **never** delete the last visible record from any Panorama database.

**Note:** This statement will not generate a warning alert even if it is the last statement in a procedure.

This statement has the same effect as hitting the **Backspace** or **Delete** key on the keyboard.

**Examples:** In this example assume we have a database of three records and we are on record 2 before we run the following procedure statement. Afterwards, record 2 is copied to the clipboard and then deleted and the cursor ends up on record 1.

```
deleteabove
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[pasteall](#) statement  
[pasterecord](#) statement

# DELETEALL

**Syntax:** DELETEALL

**Description:** The `deleteall` statement will remove all of the database's records and replace them with one blank record.

**Parameters:** This statement has no parameters.

**Action:** This statement removes the active database's entire collection of record, whether they are visible or invisible and replaces them with one blank record. This statement **must** leave one blank record because you must always have at least one record in any Panorama database at all times.

**Warning:** This statement can be extremely **dangerous** if used incorrectly; saving after running this statement will eliminate all of your database records permanently, you cannot undo this command. However, `deleteall` is the quickest way to clear a database of all records. It can be very useful when making a copy of a databases and clearing the copy to begin new record keeping (see example below).

**Note:** If **Auto Save** is turned on and `deleteall` is the last procedure statement Panorama will ask if you wish to suspend **Auto Save** before deleting all records. This will allow you to do a Revert To Saved (**File** menu) later on to restore your records.

**Warning:** This statement will **not** bring up an alert dialog even if it is the last statement in a procedure.

**Note:** In a cross tab window `deleteall` will only delete the cross tab records for that cross tab. This statement has the same effect as choosing the **DeleteALL** command from the **Edit** menu, except no alert precedes the deletion.

**Examples:** This simple example will delete all record from the active database and leave one blank record in their place.

```
deleteall
```

This example makes a copy of the current database and then deletes all the records from the copy called My Second File.

```
saveas "My Second File"
deleteall
save
```

This example shows how the procedure can ask you if you wish to do a `deleteall` and only do so if you click Ok.

```
cancelok "Do you really want to delete ALL records?"
if clipboard() contains "ok"
 deleteall
 save
endif
```

**Views:** This statement may be used in the Data Sheet, Form view, or Cross Tab view **only**.

**See Also:**

[clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement

# DELETEFIELD

**Syntax:** DELETEFIELD

**Description:** The **deletefield** statement will remove the current field from the database.

**Parameters:** This statement has no parameters.

**Action:** This statement removes the currently active field from the database and all data contained in that field without any prior warning. This statement will delete any type of field, however Panorama **will not** allow you to delete the last field in the database.

**Warning:** This statement can be extremely **dangerous** if used incorrectly; saving after running this statement will eliminate the field and all of the data contained in the field permanently, you cannot **undo** this command. You may wish to turn off **Auto Save** before using this statement. If you do not save you may reclaim the deleted field(s) by using the revert statement or by using the **Revert to Saved** command in the **File** menu.

**Warning:** This statement will **not** bring up an alert dialog even if it is the last statement in a procedure.

This statement has the same effect as choosing the **Delete Field** command from the **Setup** menu, except no alert precedes the deletion of the field.

**Examples:** This simple example will delete the field called Temp.

```
field Temp
deletefield
```

This example ask you if you wish to delete the current field and only does so if you click **Yes**.

```
noyes "Do you really want to delete The Field: "
 +info("fieldname")+ "?"
if clipboard() contains "yes"
 deletefield
 save
endif
```

This example copies the current database to a database called **New File** and then deletes all the fields that are Picture type fields in New File.

```
saveas "New File"
field «Field 1» ;first field in database.
loop
 if info("datatype")= 1 ;for Picture type fields.
 deletefield
 left
 endif
 right
until info("stopped")
```

**Views:** This statement may be used in the Data Sheet view **only**.



**See Also:** [addfield](#) statement  
[dbinfo\(\)](#) function  
[field](#) statement  
[fieldname](#) statement  
[fieldstyle\(\)](#) function  
[fieldtype](#) statement  
[info\("datatype"\)](#) function  
[info\("fieldname"\)](#) function  
[insertfield](#) statement  
[newgeneration](#) statement

# DELETERECORD

**Syntax:** DELETERECORD

**Description:** The `deleterecord` statement will first copy to the clipboard and then delete the currently selected record. There is no difference between this statement and the `cutrecord` statement.

**Parameters:** This statement has no parameters.

**Action:** This statement removes the entire active record from the database after first copying the record to the clipboard. After deleting the record all other records move up one row in the window and the cursor will reside on the next visible record immediately following the deleted record. On a single record view form the display will change to the next visible record after the deleted record.

This statement will delete the currently active record unless there is only one record selected, then it will only copy the record to the clipboard and beep once. The beep is a warning because you can **never** delete the last visible record from any Panorama database.

**Note:** If this statement is the last procedure statement it will present you with a **Delete/Cancel** alert dialog saying "About to delete current record.", otherwise this alert is suppressed.

This statement has the same effect as clicking on the **Cut Record** tool on a tool palette (when available). Holding down the **Option** key (Macintosh only) while clicking on this tool will suppress the alert.

**Examples:** This simple example will copy to the clipboard and remove the active record from the database after alerting you to the action.

```
deleterecord
```

This nearly identical example will copy to the clipboard and remove the active record from the database, but will suppress the alert.

```
deleterecord
nop
```

This example checks all selected records in the Stock database one by one and deletes a record when the Status field contains the word `old` for that record. It then pastes the deleted record into the Old Stock database. If only one record remains in Stock and it is an old record the procedure will warn you that you cannot delete it.

```
firstrecord
loop
 if Status contains "old"
 if info("selected") = 1
 beep
 message "You cannot delete the last record."
 stop
 else
 deleterecord
 window "Old Stock"
 pasterecord
```

```
 window "Stock"
 endif
endif
 downrecord
until info("stopped")
```

**Views:** This statement may be used in any view

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[paste](#) statement  
[paste](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement

# DELETERESOURCE

**Syntax:** `DELETERESOURCE type,id`

**Description:** The `deleteresource` statement deletes a resource item. The resource file must be opened with the `openresourcerw` statement.

**Parameters:** This statement has two parameters: `type` and `id`.

**type** is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).

**id** is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

**Action:** This statement deletes a resource item.

**Examples:** The procedure below deletes `STR` resource number `2000`.

```
deleteresource "STR ",2000
```

**Views:** This statement may be used in any view.

**See Also:** [openresource](#) statement  
[openresourcerw](#) statement  
[closeresource](#) statement  
[writeresource](#) statement  
[renameresource](#) statement  
[activeresource](#) statement

# DELETEWINDOW

**Syntax:** DELETEWINDOW

**Description:** The `deletewindow` statement allows you to delete the active form from its corresponding database.

**Parameters:** This statement has no parameters

**Action:** This statement allows you to delete the active form from the database it is stored in. This statement will first warn you that you are about to delete the active window, if you click the **Ok** button the form will be deleted and its window will be removed from the screen, if you click on the **Cancel** button the operation will be dismissed and the procedure will continue on. The alert cannot be suppressed for this statement.

**Note:** if the only window open for the database is a form window and you use this statement Panorama will delete the form and replace the form window on screen with the Data Sheet window for that database.

**Examples:** This example will delete the active window provided it is a form window and that you respond by hitting the **Ok** button.

```
deletewindow
```

This example allows you to choose the form you wish to delete, provided it exists.

```
getscrap "Enter form name."
openform clipboard()
if error
 message "Form: "+clipboard()+" not found."
 stop
endif
deletewindow
```

This example allows you to select a form from the list provided by the `formselect` statement and, if the **Delete** button is selected, will open the form window and then delete the form from the database

```
local B,F
formselect 2086,0,B,F
if B contains "delete" and F ≠ ""
 windowbox "99 77 244 435"
 openform F
 deletewindow
endif
```

**Views:** This statement may be used in a Form view **only**.

**See Also:**

[goform](#) statement  
[graphicsmode](#) statement  
[openform](#) statement  
[renamewindow](#) statement  
[window](#) statement

# DETACHSERVER

**Syntax:** DETACHSERVER

**Description:** The `detachserver` statement takes a Partner/Server database and removes the connection to the SQL server database, leaving a standard Panorama database.

**Parameters:** This statement has no parameters.

**Action:** This statement takes a multi-user Partner/Server Panorama database and converts it into a single user database. After this statement is used the database is no longer linked to an SQL database, and will no longer be able to access data from the SQL database.

Before using this statement you may wish to use the [subsetselectall](#) statement to load all the server data into the local Panorama database. You may use this statement in the data sheet, design sheet or any form. **Note:** The Panorama database is not permanently detached from the SQL database until you save it.

**Examples:** This example first makes sure that the entire SQL server database is loaded into the local Panorama database, then converts the database to single user, removing all links to the server database.

```
subsetselectall
detachserver
```

**Views:** This statement may be used in the Data Sheet, Design Sheet, and Form views.

**See Also:** [attachserver](#) statement  
[serverfile](#) statement  
[info\("serverstatus"\)](#) function  
[info\("serverfile"\)](#) function

# DIAL

**Syntax:** DIAL dialstring

**Description:** The **dial** statement generates dialing touch-tones for number characters and plays them through the Macintosh's speaker. (On Windows systems this statement is ignored.)

**Parameters:** This statement has one parameter: dialstring

**dialstring** must be in a text format. It can be a quoted string, text formula, field, or variable that contains the numeric characters you wish to dial. Non-numeric characters in dialstring will cause Panorama to pause momentarily, but are ignored. The Mac's speaker volume may be adjusted using the [dialvolume](#) statement.

**Action:** This statement takes the numeric characters in dialstring, converts them to dialing touch-tones, and plays them through the computer's speaker. Theoretically, you could hold the handset to your telephone up to the speaker and have the computer dial the telephone, however most Mac speakers are not powerful enough to generate the proper signal strength for the telephone to respond to.

It may be possible to hook up auxiliary speakers that are stronger or you can purchase hardware like Sophisticated Circuits': **Desktop Dialer** which will allow you to dial the telephone through your computer. Panorama has a [dialdesktop](#) statement for use with this hardware.

**Hint:** You may use the dial statement to test a dialstring prior to using it with a [dialmodem](#) or [dialprinter](#) statement.

**Examples:** This simple example plays the tones for 555-1212 through the speaker, pausing briefly for the dash (-) character.

```
dial "555-1212"
```

This example dials the numeric characters in the text field called Phone ignoring non-numeric characters.

```
dial Phone
```

If Phone were a numeric field you could convert the characters to text for dial with this example.

```
dial str(Phone)
```

The example shows how a phone number can be typed in and read from the clipboard by dial.

```
getscrap "Enter the phone number."
dial clipboard()
```

The example stores the area code in a variable called **AreaCode** and stores the phone number in a variable called **Phone#** and uses them in a formula as the dialstring.



```
local AreaCode,Phone#
AreaCode = "714"
«Phone#» = "8417779"
dial AreaCode+«Phone#»
```

**Views:** This statement may be used in any view.

**See Also:** [baud](#) statement  
[dialdesktop](#) statement  
[dialmodem](#) statement  
[dialprinter](#) statement  
[dialvolume](#) statement  
[str\(\)](#) function  
[pattern\(\)](#) function

# DIALDESKTOP

- Syntax:** DIALDESKTOP dialstring
- Description:** The `dialdesktop` statement sends numeric characters to Sophisticated Circuits': **Desktop Dialer** hardware, which in turn sends them to a telephone connect to it.
- Parameters:** This statement has one parameter: dialstring  
**dialstring** must be in a text format. It can be a quoted string, text formula, field, or variable that contains the numeric characters you wish to dial.
- Action:** This statement will only be useful if you are using Sophisticated Circuits': **Desktop Dialer** hardware. The Desktop Dialer allows you to connect your telephone to your Macintosh computer through the ADB port. This statement will allow Panorama to dial the telephone connected to the device.  
 This statement takes the numeric characters in dialstring and sends them to the Desktop Dialer which in turn converts them to dial signals and sends them to the telephone, thereby allowing the computer to dial the phone.  
 For more information on the **Desktop Dialer** you can reach Sophisticated Circuits', located in Bothel Washington, at (425) 458-7979. Or you can go to their web site, <http://www.sophisticated.com>.
- Examples:** This simple example sends the characters 555-1212 to the Desktop Dialer.
- ```
dialdesktop "555-1212"
```
- This example dials the numeric characters in the text field called Phone.
- ```
dialdesktop Phone
```
- If Phone were a numeric field you could convert the characters to text for dialdesktop with this example.
- ```
dialdesktop str(Phone)
```
- The example shows how a phone number can be typed in by the user and then read from the clipboard by dialdesktop.
- ```
getscrap "Enter the phone number."
dialdesktop clipboard()
```
- The example stores the area code in a variable called AreaCode and stores the phone number in a variable called Phone# and uses them in a formula as the dialstring.
- ```
local AreaCode, Phone#
AreaCode = "714"
«Phone#» = "8417779"
dialdesktop AreaCode+«Phone#»
```
- Views:** This statement may be used in any view.

See Also: [baud](#) statement
 [dial](#) statement
 [dialmodem](#) statement
 [dialprinter](#) statement
 [dialvolume](#) statement
 [str\(\)](#) function
 [pattern\(\)](#) function

DIALMODEM

- Syntax:** DIALMODEM dialstring
- Description:** The `dialmodem` statement sends any characters in dialstring out through your Mac's modem serial port. This statement is similar to the [dialprinter](#) statement.
- Parameters:** This statement has one parameter: dialstring
- dialstring** must be in a text format. It can be a quoted string, text formula, field, or variable that contains the characters you wish to send to your modem port. dialstring may contain numeric and non-numeric characters. For example, a modem connected to your modem port that is Hayes compatible will accept commands beginning with the letters **AT**. These commands give the modem instructions; for example, **ATH0** will hang up the modem. See your modem guide for information on acceptable modem commands.
- Action:** This statement takes the characters in dialstring and sends them out through the computer's modem serial port. Normally, you will have a modem connected to this port which will allow Panorama to send characters to it. However, any device with a Macintosh serial connector attached to the modem port should be able to accept characters sent by Panorama.
- Examples:** This simple example sends the characters ATDT 555-1212 through the modem port. The letters ATDT instruct Hayes compatible modems to dial tone characters. The dash character (-) is optional.

```
dialmodem "ATDT555-1212"
```

This example sends the characters in the text field called ModemPhone to the modem port.

```
dialmodem ModemPhone
```

This example could be used when you have your telephone connected to a modem which is connected to the modem port. Once the number in the field Phone is dialed by the modem it is then hung up using the command **ATH0**; this should allow you to talk on the telephone. You must have the telephone off hook prior to running this procedure.

```
dialmodem "ATDT"+Phone
dialmodem "ATH0"
```

The example shows how a phone number can be typed in by the user and then read from the clipboard by dialmodem.

```
getscrap "Enter the phone number."
dialmodem "ATDT"+clipboard()
```

The example shows a formula as the dialstring that tells Panorama to dial 6 digits if the phone number has a 714 area code, otherwise dial a 1 followed by the 10 digit number. This example assumes the field Phone is a text field containing phone numbers formatted with parenthesis () and dashes -; ex.: **(714) 841-7779**.

```
    baud "9600"  
    dialmodem "ATDT"+  
        ?(Phone[2,4]≠"714", "1"+Phone, Phone[6,-1])
```

Views: This statement may be used in any view.

See Also: [baud](#) statement
[dial](#) statement
[dialdesktop](#) statement
[dialprinter](#) statement
[dialvolume](#) statement
[str\(\)](#) function
[pattern\(\)](#) function

DIALPRINTER

Syntax: DIALPRINTER dialstring

Description: The `dialprinter` statement sends any characters in dialstring out through your Mac's modem serial port. This statement is similar to the [dialmodem](#) statement.

Parameters: This statement has one parameter: dialstring

dialstring must be in a text format. It can be a quoted string, text formula, field, or variable that contains the characters you wish to send to your printer port. dialstring may contain numeric and non-numeric characters. For example, a modem connected to your modem port that is Hayes compatible will accept commands beginning with the letters **AT**. These commands give the modem instructions; for example, **ATH0** will hang up the modem. See your modem guide for information on acceptable modem commands.

Action: This statement takes the characters in dialstring and sends them out through the computer's printer serial port. Normally, you will have a modem connected to this port which will allow Panorama to send characters to it. However, any device with a Macintosh serial connector attached to the modem port should be able to accept characters sent by Panorama.

Examples: This simple example sends the characters ATDT 555-1212 through the printer port. The letters ATDT instruct Hayes compatible modems to dial tone characters. The dash character (-) is optional.

```
dialprinter "ATDT555-1212"
```

This example sends the characters in the text field called ModemPhone to the printer port.

```
dialprinter ModemPhone
```

This example could be used when you have your telephone connected to a modem which is connected to the printer port. Once the number in the field **Phone** is dialed by the modem it is then hung up using the command **ATH0**; this should allow you to talk on the telephone. You must have the telephone off hook prior to running this procedure.

```
dialprinter "ATDT"+Phone
dialprinter "ATH0"
```

The example shows how a phone number can be typed in by the user and then read from the clipboard by dialmodem.

```
getscrap "Enter the phone number."
dialprinter "ATDT"+clipboard()
```

The example shows a formula as the dialstring that tells Panorama to dial 6 digits if the phone number has a 714 area code, otherwise dial a 1 followed by the 10 digit number. This example assumes the field Phone is a text field containing phone numbers formatted with parenthesis () and dashes -; ex.: **(714) 841-7779**.

```
    baud "9600"  
    dialprinter "ATDT"+  
        ?(Phone[2,4]≠"714", "1"+Phone,Phone[6,-1])
```

Views: This statement may be used in any view.

See Also: [baud](#) statement
[dial](#) statement
[dialdesktop](#) statement
[dialmodem](#) statement
[dialvolume](#) statement
[str\(\)](#) function
[pattern\(\)](#) function

DIALVOLUME

Syntax: DIALVOLUME level

Description: The **dialvolume** statement controls the volume of the Macintosh's speaker.

Parameters: This statement has one parameter: level

level must be a number between 1 (soft) and 7 (loud), however level can be in a text or numeric format. level can be a literal numeric value, a formula, a field, or a variable that contains the numeric character you wish the volume to be.

Action: This statement takes the numeric character in level and uses it to set the volume of the Mac's speaker. Using this command prior to a dial statement will control the volume as the characters are played through the speaker.

This statement has the same effect as adjusting the volume in the **Sound** control panel.

Examples: This simple example plays the tones for 555-1212. through the speaker as loudly as possible.

```
dialvolume 7
dial "555-1212"
```

This example allows you to set the volume prior to dialing the numeric characters in the text field called Phone.

```
getscrap "Enter the volume number."
dialvolume clipboard()
dial Phone
```

This almost identical example also works, but notice that the clipboard value is converted to a numeric format by the val() function.

```
getscrap "Enter the volume number."
dialvolume val( clipboard())
dial Phone
```

The example stores the volume in a variable called Volume, sets the volume to 5., and then dials the number in the field called Phone through the speaker.

```
local Volume
Volume = 5
dialvolume Volume
dial Phone
```

Views: This statement may be used in any view.

See Also: [baud](#) statement
[dial](#) statement
[dialdesktop](#) statement
[dialmodem](#) statement

dialprinter statement
str() function
pattern() function

DISABLEABORT

Syntax: DISABLEABORT

Description: The `disableabort` statement tells Panorama not to stop the current procedure if the Command-Period keys are pressed. Pressing Command-Period normally stops any procedure right in it's tracks, no matter what the procedure is doing. This is normally an important safety valve, but you may have a procedure that should not be stopped in the middle with a job halfway done. Command-Period may be re-enabled with the [enableabort](#) statement.

Parameters: This statement has no parameters.

Examples: The example below shows how `disableabort` and `enableabort` can be used to make sure that a sequence of steps is always completed -- even if the user presses **Command-Period** (Macintosh) or **Control-Period** (Windows). In this case there is no way that the new record can be added without being filled in by the lookup formulas. You either get all or nothing, but not a halfway done job.

```

disableabort
addrecord
Name=dialogName
Address=lookup("Contacts", "Name", Name, Address, "", 0)
City=lookup("Contacts", "Name", Name, City, "", 0)
State=lookup("Contacts", "Name", Name, State, "", 0)
Zip=lookup("Contacts", "Name", Name, Zip, "", 0)
enableabort

```

When using the `disableabort` statement you must be careful, especially when using loops. The procedure below will hang Panorama because `tag=""` will never be true. The only way to stop the loop is to reboot the computer or do a force quit on Panorama (**Command-Shift-Option-Escape** on Macintosh, **Control-Alt-Delete** on Windows).

```

disableabort
local i,tag
i=1
loop
    tag="<"+array(Text,i,¶)+">"
    stoploopif tag<>""
    i=i+1
while forever
enableabort

```

While this example may look silly, it is easy to create an endless loop without realizing it. It is also possible to check for **Command/Control-Period** yourself. To learn how to do this, see [info\("abort"\)](#).

Views: This statement may be used in any view, and also works when no windows are open at all.

See Also: [enableabort](#) statement
[info\("abort"\)](#) function

DIVZERO(...)

Syntax: DIVZERO(numerator,denominator)

Description: The **divzero**(function divides two numbers. However, unlike the / operator, the divzero function does not care if you attempt to divide by zero. If you attempt to divide by zero, this function simply returns zero.

Parameters: This function has two parameters: **numerator** and **denominator**.
numerator is the number you want to divide.
denominator is the number you want to divide by.

Result: This function returns the numerator divided by the denominator (numerator/denominator) unless the denominator is zero. If the denominator is zero, the function returns zero.

Examples: This example calculates the unit price of an item. If the Quantity is zero (or is not filled in yet) the UnitPrice will also be zero.

```
UnitPrice=divzero(Price,Quantity)
```

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text item for the numerator or denominator parameter.

DOWNRECORD

Syntax: DOWNRECORD

Description: The **downrecord** statement moves the cursor down one visible record in the active window. This is the opposite of the [uprecord](#) statement.

Parameters: This statement has no parameters

Action: This statement moves the cursor down one visible record in the Data Sheet, Design Sheet, Cross Tab view, or View-as-list Form view. In a Individual Record Form view the view will change to the next record down in the database. If the cursor is already on the last visible record this statement will do nothing.

You can use this statement in conjunction with the [info\("eof"\)](#) or [info\("stopped"\)](#) functions to test to see if you are on the last visible record in the window.

This statement has the same effect as clicking on the **Down Record** tool on a tool palette (when available).

Examples: This simple example could be used in either the Data Sheet, Form view or Cross Tab view to move the cursor to the next visible record below the current record, making this next record the current record.

```
downrecord
```

This example moves the cursor down record by record until it finds the last visible record in the window. You could replace `info("eof")` with `info("stopped")`.

```
loop
  downrecord
until info("eof")
```

This example moves the cursor to the third record in the Design Sheet window, which is the third field in the database, and changes it's width to 6.

```
opendesignsheet
loop
  downrecord
until 3
Width = 6
newgeneration
closewindow
```

Views: This statement may be used in any view.

See Also: [firstrecord](#) statement
[info\("eof"\)](#) function
[info\("stopped"\)](#) function
[lastrecord](#) statement
[uprecord](#) statement

DRAGGRAYBOX

Syntax: DRAGGRAYBOX dragrectangle,limits,slop,axis

Description: The **draggraybox** statement allows the user to drag a gray box around on the screen. It can be used for drag-and-drop or for manipulating graphic objects.

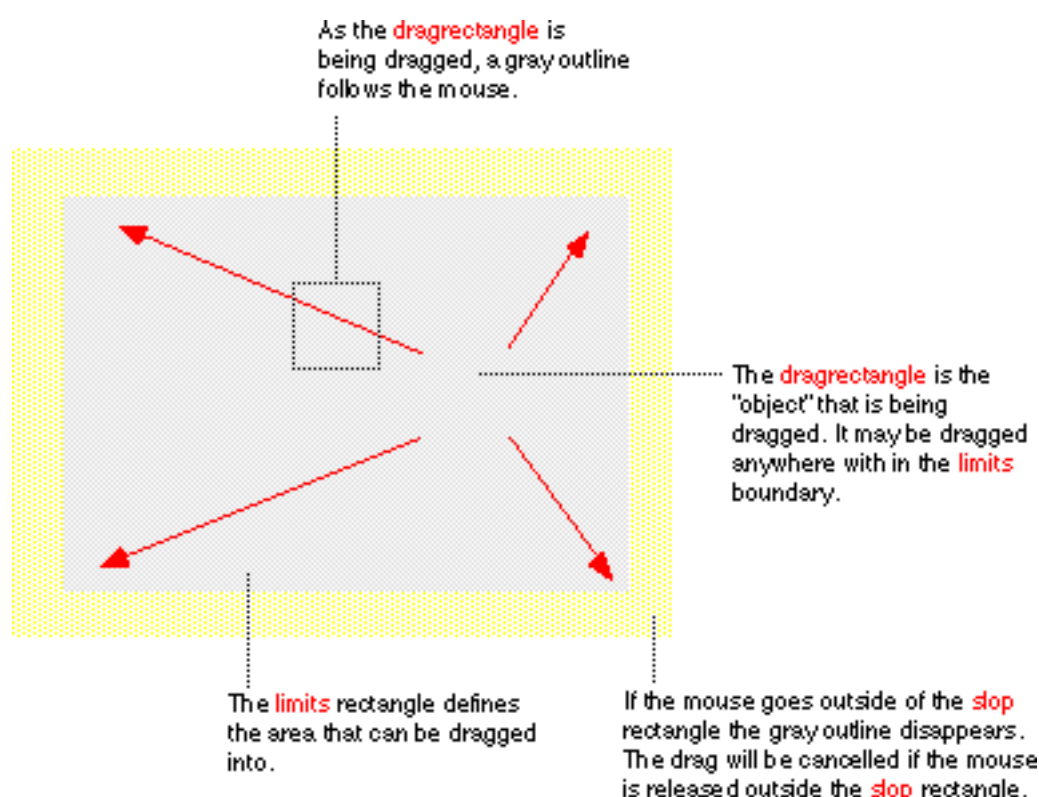
Parameters: This statement has four required parameters: **dragrectangle**, **limits**, **slop** and **axis**.

dragrectangle is the original co-ordinates of the rectangle the user will drag around. Often these co-ordinates are the same as the co-ordinates for the button the user pressed on. (Note: the co-ordinates for this rectangle, along with the next two, are relative to the upper left hand corner of the screen.) This parameter should be a field or variable (not a more complex formula) because after the user has released the mouse Panorama will copy the final co-ordinates into this parameter.

limits is the co-ordinates of a boundary rectangle that defines how far the dragrectangle can be dragged in each direction. For example if you don't want the user to be able to drag the box outside of the current window you should supplied the co-ordinates of the current window for limits. If the limits parameter is empty ("") there will be no limit on how far the rectangle can be dragged.

slop is the co-ordinates of a boundary rectangle past the limits boundary. If the user drags the mouse beyond the slop rectangle the gray rectangle will disappear completely (until the user drags back inside the slop rectangle). If the slop parameter is empty ("") it will be the same as the limits boundary rectangle.

axis allows the procedure to restrict the direction the rectangle can be dragged to either horizontal or vertical. If the axis parameter is 0 the rectangle can be dragged in any direction. If the axis is 1 the rectangle can only be dragged horizontally. If the axis is 2 the rectangle can only be dragged vertically.



Action: This statement allows the user to drag a gray box around on the screen. It is designed to be used in a procedure that is triggered by a transparent button with the click/release option turned off. When the user presses on the button, the procedure is triggered immediately. The procedure calculates size and location of the original rectangle to drag

around, as well as the limits to where this rectangle can be dragged. Then the **DragGrayBox** statement takes over. As long as the user continues to hold down the mouse a gray box will follow the mouse around. When the user lets up on the mouse button the **DragGrayBox** statement tells the procedure the final position of the box. The procedure can then take whatever action is appropriate (moving a graphic object, copying data, etc.)

Examples:

The procedure below allows the user to drag a button around the window. When the user releases the mouse, the procedure moves the button to the new location.

```

local drag,insidewindow
drag=info("buttonrectangle")
selectobjects xytoxy(drag,"s","f")=objectinfo("rectangle")
insidewindow=rectangle(
rtop( info("windowrectangle")+20,
rleft( info("windowrectangle")+26,
rbottom( info("windowrectangle")-16,
rright( info("windowrectangle")-16)
draggraybox drag,insidewindow,info("windowrectangle"),0
if drag="" /* was mouse released outside slop area? */
stop /* yes, so stop */
endif
drag=xytoxy(drag,"s","f")
changeobjects "rectangle",drag

```

The procedure below implements drag and drop. When the button is dragged onto the object named Landing Zone, an item is added to the order.

```

local drag,insidewindow
drag=info("buttonrectangle")
insidewindow=rectangle(
rtop( info("windowrectangle")+20,
rleft( info("windowrectangle")+26,
rbottom( info("windowrectangle")-16,
rright( info("windowrectangle")-16)
draggraybox drag,insidewindow,info("windowrectangle"),0
object "Landing Zone"
if inrectangle( xytoxy( info("mouse"),"s","f"),
objectinfo("rectangle"))
/* dragged into the landing zone, so copy data */
call .AddToOrder
endif

```

Views: This statement may be used in a Form view.

See Also: [graphic coordinates](#)

DRAWMENUS

Syntax: DRAWMENUS

Description: The `drawmenus` statement redraws the menus in the menu bar at the top of the screen for the current window.

Parameters: This statement has no parameters.

Action: This statement is primarily used to draw custom menus on the screen after they have been set up for your database's window(s). Setting up custom menus requires creating a menu resource using a program like **ResEdit** or **Resorcerer™**. Once created you must indicate which windows in your database are to use these custom menus. When you first open the database you will need to include the `drawmenus` statement in an `.Initialize` procedure so that the proper menu comes up with the window when the database first opens.

Examples: This example opens a menu resource called `Custommenus` and then draws the menus in the menu bar. These two statements should be included in an `.Initialize` procedure for any database using custom menus.

```
openresource "Custommenus"  
drawmenus
```

Views: This statement may be used in the Data Sheet, Form views and Cross Tab views **only**.

See Also: [clearmenumarks](#) statement
[getmenus](#) statement
[getmenumark](#) statement
[getmenutext](#) statement
[menubuild](#) statement
[menudisable](#) statement
[menuenable](#) statement
[openresource](#) statement
[setmenus](#) statement
[setmenumark](#) statement
[setmenutext](#) statement

DRAWOBJECTS

Syntax: DRAWOBJECTS

Description: The `drawobjects` statement redraws one or more objects on the current form.

Parameters: This statement has no parameters.

Action: This statement forces one or more objects in the current form to be redrawn. To specify which object or objects should be redrawn you must use either the [object](#), [objectid](#) or [selectobjects](#) statements immediately before the `drawobjects` statement.

Examples: This example redraws the object called `Swiss Cheese`.

```
object "Swiss Cheese"  
drawobjects
```

This example redraws all of the objects in the current form that are displayed in the font Courier.

```
selectobjects objectinfo("font")="Courier"  
drawobjects
```

Views: This statement may be used in a Form view **only**.

See Also: [object](#) statement
[objectid](#) statement
[selectobjects](#) statement
[noshow](#) statement
[show](#) statement

EDITCELL

Syntax: EDITCELL

Description: The `editcell` statement opens the edit window for the currently active field (or cell), highlighting the data, and allows you to edit that field.

Parameters: This statement has no parameters.

Action: This statement opens the edit window for the current field allowing you to make changes to the field manually. After this statement opens the edit window the procedure will suspend operations temporarily until you manually close the edit window, then the procedure will continue after the `editcell` statement.

Warning: opening a field containing data will cause that data to be highlighted (selected) which will cause the data to be replaced with the keystrokes that follow. Remember to insert the cursor into the field before editing begins. If you prefer you may use the `editselect` statement in the same procedure to preset the insertion point for the editing cursor (see examples below).

Note: This statement will work on **any** type of field with the exception of a **picture** type field.

This statement has the same effect as **Double-clicking** on the current field (or cell).

Examples: This simple example will open the edit window for the active field to allow you to edit that cell.

```
editcell
```

This example combines the `editcell` statement with the `editselect` statement to insert the edit cursor at the end of the Notes field.

```
field Notes
editselect 32768,32768
editcell
```

This example is a trick you can use if you wish to open and close an edit window for a text field without actually editing the field. This trick is useful for clearing the Find/Select dialog. This trick will only work if Name is a text type field.

```
local A
A = 0
field Name ; This can actually be any text field.
editcell
A=A+1
```

Views: This statement may be used in a Form view

See Also: [editselect](#) statement
[editcellstop](#) statement
[field](#) statement
[floatingedit](#) statement

EDITCELLSTOP

Syntax: EDITCELLSTOP

Description: The `editcellstop` statement opens the edit window for the currently active field (or cell), highlighting the data, and allows you to edit that field. The procedure then stops immediately. This is not quite the same as using the regular `editcell` statement followed by a `stop` statement. In that case the procedure doesn't stop immediately, but waits until the user finishes editing. Use `editcellstop` if you want automatic procedures to trigger when the user finishes editing.

Parameters: This statement has no parameters.

Action: This statement opens the edit window for the current field allowing you to make changes to the field manually. After this statement opens the edit window the procedure will stop.

Warning: opening a field containing data will cause that data to be highlighted (selected) which will cause the data to be replaced with the keystrokes that follow. Remember to insert the cursor into the field before editing begins. If you prefer you may use the `editcell` statement in the same procedure to preset the insertion point for the editing cursor (see examples below).

Note: This statement will work on **any** type of field with the exception of a **picture** type field.

This statement has the same effect as **Double-clicking** on the current field (or cell).

Examples: This example combines the `editcellstop` statement with the `editselect` statement to insert the edit cursor at the end of the Notes field.

```
if Notes=""
  field Notes
  editselect 32768,32768
  editcellstop
else
  Notes=datepattern( today(),"Month ddnth, yyyy")
endif
```

If the Notes field has an automatic procedure defined in the design sheet, it will be triggered when the user has finished editing. The example below, which uses the regular `editcell` statement, **will not** trigger the automatic procedure.

```
if Notes=""
  field Notes
  editselect 32768,32768
  editcell
  stop
else
  Notes=datepattern( today(),"Month ddnth, yyyy")
endif
```

Views: This statement may be used in a Form view.

See Also: [editcell](#) statement
 [editselect](#) statement
 [stop](#) statement
 [field](#) statement
 [floatingedit](#) statement

EDITSELECT

Syntax: EDITSELECT start,end

Description: The `editselect` statement allows you to highlight a specified section of data or establish an insertion point inside an edit window of a text, numeric, or date field you wish to edit.

Parameters: This statement has two parameters: `start` and `end`.

start is a number between **0** and **32768** which indicates the beginning of the highlighted characters by number of characters from the beginning of the field. `start` may be a number, a field or variable containing a number, or a formula that results in an integer value between 0 and 32768.

end is a number between **0** and **32768** which indicates the ending of the highlighted characters by number of characters from the beginning of the field. `end` may be a number, a field or variable containing a number, or a formula that results in an integer value between 0 and 32768. Note: Specifying an ending value greater than the length of the field will place the highlight at the end of the field's data.

Note: If `start` and `end` are the same value Panorama will simply insert the edit cursor at that point in the field's edit window.

Action: This statement allows you to specify which characters, if any, are to be highlighted after an edit window is opened for a field. Once the text is highlighted it can be copied, deleted, or replaced manually. If the values for `start` and `end` values are equal then Panorama will **insert** the edit cursor (blinking cursor) at that point in the field. This statement **must** be combined with the `editcell` or `floatingedit` statement in order to be used in a procedure (see examples below).

Warning: Any data selected by the `editselect` statement is subject to being replaced when you edit the field. Caution should be used when working with this statement to insure no loss of data due to accident. Note: This statement will work for **text**, **numeric**, and **date** type fields. It **will not** apply to **choice** or **picture** type fields.

This statement has the same effect as **Double-clicking** on the current field (or cell) and manually inserting a cursor or highlighting some or all of the text in the edit window.

Examples: This simple example will open the edit window for the active field and insert the edit cursor at the beginning of the field to begin typing. This procedure will work provided that the field being edited is a text, numeric, or date type field.

```
editselect 0,0
editcell
```

This example combines the `editcell` statement with the `editselect` statement to insert the edit cursor at the end of the text field called Notes.

```
field Notes
editselect 32768,32768
editcell
```

This example opens an edit window and highlights the text string **-name-**, only if it is present in the text field Comments, allowing you to change it.

```

local start,end
field Comments
start = search(«Comments», "-name-")
end = search(«Comments», "-name-") + 5
if start ≠ 0
    editselect start-1,end
    editcell
endif

```

This nearly identical example shows how you can combine **editselect** with the **floatingedit** statement.

```

local start,end
openform "Edit Sheet"
field Comments
start = search(«Comments», "-name-")
end = search(«Comments», "-name-") + 5
if start ≠ 0
    editselect start-1,end
    floatingedit "Comments",20,20,90,90,"Times",12,0
endif

```

Views: This statement may be used in any view.

See Also: [editcell](#) statement
[field](#) statement
[floatingedit](#) statement
[length\(\)](#) function
[search\(\)](#) function

ELSE

Syntax: ELSE

Description: The **else** statement is an optional statement used to separate procedure code for a true **if** test from a false **if** test in an if construct. See **if** and **endif** for more information.

Parameters: This statement has no parameters.

Action: This statement is used to mark the division between procedure code you wish to run when an **if** test is true vs. when the test is false. This statement must be placed after an **if** statement and before its corresponding **endif** statement and cannot be used as a stand-alone statement.

In the event of a **true if** test: all procedure code between the **if** and the **else** will be executed and then the procedure will continue with everything after the **endif** statement. In the event of a **false if** test: all procedure code between the **else** and the **endif** will be executed and then the procedure will continue with everything after the **endif** statement.

All procedure statements after an **endif** statement will execute, regardless of which portion of the **if** construct executes.

Using the Check Procedure tool on a procedure that has an **else** statement with no corresponding **if** and **endif** statements will result in an **error alert**. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts its operation.

Examples: In this example the **else** statement shows where the separation is between what is to happen if the sex field equals **Male** or **Female**. The prompt will differ based on the sex.

```

if sex = "Male"
  getscrap "Please enter his name."
else
  getscrap "Please enter her name."
endif
...
...
...

```

In this example the **info("empty")** is the true/false test; if it is true the message is displayed before going to the Sales Rep field, if it is false everything between the **else** and the **endif** statement is executed before going to Sales Rep.

```

select «#Sold» ≥ 0 and «Sale Date» ≥ month1st( today())
if info("empty")
  message "No sales records for this month."
else
  field «#Sold»
  total
  field Bonus
  formulafill ?(«#Sold»> 500.00,«#Sold»*.05,0)
endif
field «Sales Rep»
...
...
...

```

Views: This statement may be used in any view.

See Also: [?\(function](#)
[case statement](#)
[defaultcase statement](#)
[endcase statement](#)
[endif statement](#)
[if statement](#)

EMPTYFIELD

Syntax: `EMPTYFIELD fieldname`

Description: The `emptyfield` statement tells Panorama to move to the next available (or empty) field in a series of line item fields.

Parameters: This statement has one parameter: `fieldname`

fieldname can be a quoted string, field or variable, or formula that returns the name of a line item field. This field name should include the symbol (typed Option-Z) in place of the number portion of the line item field name.

Action: This statement moves Panorama's cursor to the next line item field that is blank typically so you may begin editing that field or to place a value into that field. This command would be useful if you wished to add something to an invoice record that had been previously created, allowing you to go to the next available line item field to add the additionally ordered items.

Note: If there are no available line item fields to go to this statement will be ignored.

Examples: This simple example tells Panorama to go to the next line item PriceΩ field that is blank.

```
emptyfield "PriceΩ"
```

This example looks for the next available line item field similar in name to the current field name.

```
local theFieldName
theFieldName = info("fieldname")["A-z","-A-z"]
emptyfield theFieldName+"Ω"
```

This example shows a procedure which looks for a previously made invoice record and then goes to the next available ItemΩ field to begin editing that field if it is blank. It will warn you when you've run out of fields, too.

```
local InvNo
goform "Order Entry"
gettext "Enter Invoice number.",InvNo
find «Invoice#» = InvNo
if (not info("found"))
    message "No record found for Invoice "+InvNo
    stop
endif
emptyfield "ItemΩ"
if «» = ""
    editcell
else
    message "This are no empty ItemΩ fields available."
endif
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [field](#) statement

EMPTYFILL

Syntax: EMPTYFILL value

Description: The **emptyfill** statement fills every empty visible cell in the active field with the specified value.

Parameters: This statement has one parameter: value

value can be a literal, a field or a variable, or a formula that returns the value you wish to put into the empty cells of the current field. The resulting value must match the type and format (if any) of field you are placing it in or an error will result. If value is the word **dialog** Panorama will present you with an input dialog so you may enter your own value.

Action: This statement evaluates value and if it is compatible with the current field type writes it to every empty cell in that field for all selected records only. When value is the word **dialog** Panorama will pause the procedure and present the user with the **Empty Fill... dialog** allowing the user to enter their own value. Clicking on the **Ok** button will allow the procedure to continue. This statement will work for all field types except **picture** type fields.

This statement has the same effect as using the **Empty Fill** command from the **Math** menu.

Examples: This simple example tells Panorama to fill the empty cells with the text string **n/a** in the active field. Therefore, the active field should be a text or choice type field.

```
emptyfill "n/a"
```

This example will have Panorama open the Empty Fill... dialog which allows you to enter a value to be filled into the empty cells in the current field.

```
emptyfill dialog
```

This example uses a formula to fill the empty cells in the field Date to Renew with a date that is one year beyond the current date. The final date will be the 1st of the month.

```
field «Date to Renew»
emptyfill month1st( today() + 365)
```

Views: This statement may be used in any view.

See Also: [fill](#) statement
[formulafill](#) statement

ENABLEABORT

Syntax: ENABLEABORT

Description: The `enableabort` statement tells Panorama to stop the current procedure if the Command-Period keys are pressed. Pressing Command-Period normally stops any procedure right in it's tracks, no matter what the procedure is doing. However, this feature may be turned off with the [disableabort](#) statement.

Parameters: This statement has no parameters.

Examples: The example below shows how [disableabort](#) and `enableabort` can be used to make sure that a sequence of steps is always completed -- even if the user presses **Command-Period** (Macintosh) or **Control-Period** (Windows). In this case there is no way that the new record can be added without being filled in by the lookup formulas. You either get all or nothing, but not a halfway done job.

```

disableabort
addrecord
Name=dialogName
Address=lookup("Contacts", "Name", Name, Address, "", 0)
City=lookup("Contacts", "Name", Name, City, "", 0)
State=lookup("Contacts", "Name", Name, State, "", 0)
Zip=lookup("Contacts", "Name", Name, Zip, "", 0)
enableabort

```

When using the [disableabort](#) statement you must be careful, especially when using loops. The procedure below will hang Panorama because `tag=""` will never be true. The only way to stop the loop is to reboot the computer or do a force quit on Panorama (**Command-Shift-Option-Escape** on Macintosh, **Control-Alt-Delete** on Windows).

```

disableabort
local i,tag
i=1
loop
    tag="<"+array(Text,i,¶)+">"
    stoploopif tag=""
    i=i+1
while forever
enableabort

```

While this example may look silly, it is easy to create an endless loop without realizing it.

It is also possible to check for **Command/Control-Period** yourself. To learn how to do this, see [info\("abort"\)](#)

Views: This statement may be used in any view, and also works when no windows are open at all.

See Also: [disableabort](#) statement
[info\("abort"\)](#) function

ENDCASE

Syntax: ENDCASE

Description: The **endcase** statement is used to terminate a case construct.

Parameters: This statement has no parameters.

Action: This statement is used to mark the end of a case construct and must come after the last procedure statement that is part of the final case statement or an optional defaultcase statement.

In the event of a **true** case: after the last statement for that case executes the procedure will perform the next statement after the **endcase** statement. In the event of all cases testing **false**: the procedure will execute a defaultcase or continue executing after the endcase statement.

All procedure statements will execute after an endcase statement even if all cases test false and no defaultcase is present.

Using the Check Procedure tool on a procedure that has a endcase statement with no corresponding case statement will result in an **error alert**. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts it's operation.

Examples: In this example, the **endcase** statement shows where the case construct ends and the rest of the procedure, starting with the message statement, begins.

```

case medal = "Gold"
    Score = Score + 10
case medal = "Silver"
    Score = Score + 5
case medal = "Bronze"
    Score = Score + 2
endcase
message "The Score is: "+str(Score)
...
...

```

In this example, the message statement, after the **endcase** statement, will always execute regardless of which case comes up true.

```

case Shipby = "UPS Ground"
    Rate = 5.00
case Shipby = "UPS Blue"
    Rate = 7.50
case Shipby = "UPS Red"
    Rate = 10.00
case Shipby = "Fed-ex standard"
    Rate = 12.00
case Shipby = "Fed-ex overnight"
    Rate = 18.00
case Shipby = "Fed-ex overnight Saturday"
    Rate = 25.00
defaultcase

```

```
    getscrap "Enter shipping rate."  
    Rate = val( clipboard() )  
endcase  
message "The rate is: "+str(Rate)
```

Views: This statement may be used in any view.

See Also: [case statement](#)
[defaultcase statement](#)
[else statement](#)
[endif statement](#)
[if statement](#)

ENDIF

- Syntax:** ENDIF
- Description:** The **endif** statement is needed to mark the end of an if construct.
- Parameters:** This statement has no parameters.
- Action:** This statement is used to mark the end of an **if** construct and separate the PanTalk code to be executed inside the **if** construct from the code to be executed after the **if** construct is completed. All **if** statements **must** have a corresponding **endif** in the same procedure regardless of whether the **if** constructs are nested or not. Therefore, a procedure with, say, five **if** statements must have five **endif** statements.
- See the help files for **if** and **else** for more information on **if** constructs.
- All procedure statements after an **endif** statement will execute, regardless of which portion, if any, of the **if** construct executes.
- Using the **Check Procedure** tool on a procedure that has an **endif** statement with no corresponding **if** statements will result in an **error alert**. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts its operation.
- Examples:** In this example, the PanTalk code after the **endif** statement will only be executed if the variable Password is equal to the words: **Open Sesame**, otherwise the procedure displays a message regarding an incorrect password and stops.

```

local Password
gettext "Enter Password:", Password
if Password ≠ "Open Sesame"
    message "Your password is incorrect."
    stop
endif
goform "Secret Stuff"
beep
message "You're in."

```

In this example, the **info("empty")** is the true/false test; if it is true the message is displayed before going to the Sales Rep field, if it is false everything between the **else** and the **endif** statements is executed before going to Sales Rep.

```

select «#Sold» ≥ 0 and «Sale Date» ≥ month1st( today())
if info("empty")
    message "No sales records for this month."x
else
    field «#Sold»
    total
    field Bonus
    formulafill ?( «#Sold» > 500.00, «#Sold»*.05, 0)
endif
field «Sales Rep»
...
...
...

```

In this example, there are two nested `if` constructs both which have a corresponding `endif` statement. Only if both `if` statements test true will the record be marked **Closed**. Notice the indenting of code within the `if` constructs, this is not required, but is helpful when trying to follow the PanTalk code logically through.

```
selectall
firstrecord
field Status
loop
  if Status = "New"
    if «Final Payment» > 0
      «Date Closed» = today()
      «Status» = "Closed"
    endif
  endif
downrecord
until info("eof")
```

Views: This statement may be used in any view.

See Also: [?\(function](#)
[case statement](#)
[defaultcase statement](#)
[else statement](#)
[endcase statement](#)
[if statement](#)

ENDNOSHOW

Syntax: ENDNOSHOW

Description: The `endnoshow` statement resumes the output of text and graphics after it has been disabled with the `noshow` statement. You should use this command in a procedure when you want to disable the display of intermediate steps.

Parameters: This statement has no parameters.

Action: This statement tells Panorama to resume all output of text and graphics. Although output will resume after this statement, the `endnoshow` statement does not update the display. To show any changes that have been made since the `noshow` statement you should use `showpage`, `showline`, `showfields`, `showvariables`, `showcolumns` or `showrecordcounter`.

Examples: Here is an example that performs several operations on the current database, but only updates the display once.

```
noshow
field Date
groupup by month
field Category
groupup
field Amount
total
outlinelevel 2
showpage
endnoshow
```

Views: This statement should only be used when a form or data sheet is active.

See Also: [noshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[nundo](#) statement
[nowatchcursor](#) statement
[watchcursor](#) statement

ENDSIMULATE

Syntax: ENDSIMULATE

Description: The **endsimulate** statement restores full Panorama operation after it has been temporarily downgraded by the [simulatedirect](#) or [simulateengine](#) statements.

Parameters: This statement has no parameters.

Examples: This procedure tells Panorama to simulate Panorama Engine. If you are already using a copy of Panorama Direct or Panorama Engine this statement will be ignored.

```
simulateengine
```

This procedure restores full Panorama operation (assuming that you started with a full copy of Panorama).

```
endsimulate
```

Views: This statement may be used in any view.

See Also: [simulatedirect](#) statement
[simulateengine](#) statement

EXECUTE

- Syntax:** EXECUTE program.
- Description:** The `execute` statement allows a procedure to build a “mini-procedure” on the fly, and then run that procedure.
- Parameters:** This statement has one parameter: `program`.
program is the text (source code) of the program you want to run.
- Action:** This statement allows a program to build a “mini-procedure” on the fly, and then run that procedure. The text of the “mini-procedure” can be in a field, a variable, or constructed using a formula. You can use the `if` error statement to check for syntax errors in the mini-program. Run time errors (missing variable, type mismatch, etc.) are handled just as they would be for any other program. You cannot use the debugger (single step, etc.) on a mini-procedure. After the mini-procedure has finished the main program continues from the next statement, just like a subroutine (unless the mini-procedure contains a stop statement).
- If there is a syntax error in the mini-program (misspelled command, unexpected formula operator, etc.) two special local variables will be created: `ExecuteErrorStart` and `ExecuteErrorEnd`. These two variables contain the numbers for the starting and ending position of the error (the number of characters from the start of the mini-program.) The procedure can also find out the exact error with the `info("error")` function.
- Examples:** The example below illustrates how the `execute` statement works, but is rather silly since the mini-program could simply have been included in the main program. The real power of this statement is the ability to generate the mini-program on the fly, in response to changing circumstances. You could even keep mini-programs in a database field, giving each record its own program!

```
local myProgram
myProgram=
  {if clipboard() beginswith "Error"}+
  {message clipboard()}+
  {endif}
execute myProgram
```

If you want to check the mini-program for syntax errors without running it, put a `rtn` at the beginning of the mini-program. The `rtn` statement will prevent the mini-program itself from executing. This example assumes that the mini-program is in a field called `Source`. If there is an error it displays the error and selects the offending section of the program (this assumes that `Source` is being edited in a SuperObject Text Editor.) The `-4` compensates for the `rtn` statement and carriage return.

```
execute "rtn"+¶+Source
if error
  message "Mini-program contains an error: "+
    info("error")
  activesuperobject "SetSelection",
    ExecuteErrorStart-4,ExecuteErrorEnd-4
else
  message "Mini-program is A-OK"
endif
```

Views: This statement may be used in any view.

See Also: [call](#) statement
[rtn](#) statement

EXP(...)

Syntax: EXP(value)

Description: The `exp()` function raises e to a power specified by value.

Parameters: This function has one parameter: value.
value is a numeric value.

Result: The result of this function is a numeric floating point value.

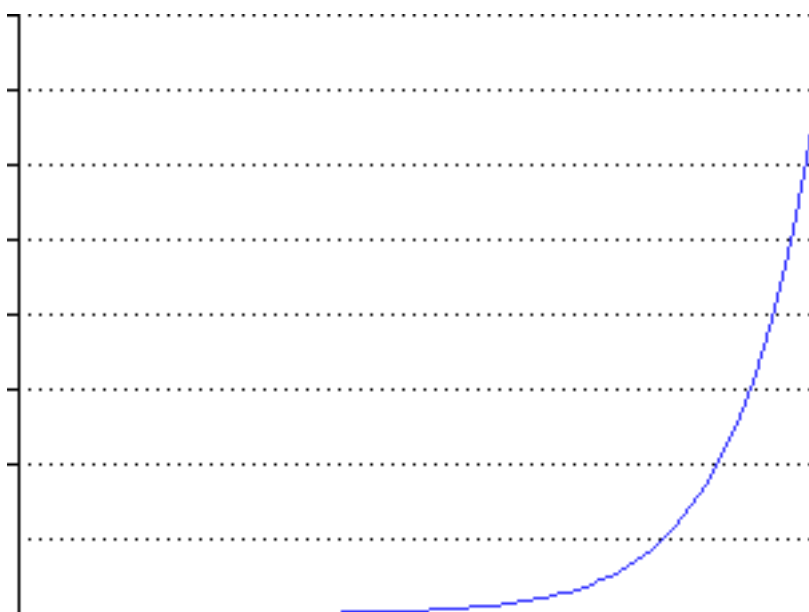
Examples: E is a mathematical constant that is approximately 2.71828. This function raises e to a power. For example, the formula

`exp(10.2)`

Is equivalent to the mathematical formula:

$e^{10.2}$

The graph below shows the result of the `exp` function given input values from -5 to +5.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `exp("23")`. If you have a numeric value in a text item you must convert the text to the number data type before raising e to a power, for example `exp(val("34"))`.

See Also: [log\(\)](#) function
[val\(\)](#) function

EXPAND

Syntax: EXPAND

Description: The **expand** statement makes visible the next lowest level of records associated with the currently active summary record.

Parameters: This statement has no parameters.

Action: This statement makes all records associated with the currently active summary record that are one level lower than that summary record to become visible.

If the active record is not a summary record (data record) this statement is ignored by Panorama.

The visible record count will accurately change after a proper expand is performed.

This statement has the same effect as clicking on the **Expand** tool on a tool palette (when available).

Examples: This example goes to the last record in the database and, if it is a summary record, expands it to the next level below that summary record's level.

```
lastrecord
expand
```

This example uses the collapse, expand, and expandall statements in a procedure to calculate percentages of Balance values for each data record associated with a specific summary 1 record for that group.

```
local BalTotal
field «GL Category»
groupup
field Balance
total
outlinelevel 1
firstrecord
field Percentage
loop
  if info("summary") =1
    BalTotal =Balance
    expand
    formulafill (Balance/BalTotal)*100
    collapse
  endif
  downrecord
until info("eof") or info("summary") >1
lastrecord
field Percentage
fill "100"
Percentage = zeroblank(0)
expandall
```

Views: This statement may be used in the Data Sheet or Form views only.

See Also:

[average](#) statement
[count](#) statement
[collapse](#) statement
[expandall](#) statement
[group](#) statement
[groupdown](#) statement
[groupup](#) statement
[info\("records"\)](#) function
[info\("selected"\)](#) function
[info\("summary"\)](#) function
[maximum](#) statement
[minimum](#) statement
[outlinelevel](#) statement
[removesummaries](#) statement
[selectsummaries](#) statement
[total](#) statement

EXPANDALL

Syntax: EXPANDALL

Description: The **expandall** statement makes visible all lower level of records associated with the currently active summary record.

Parameters: This statement has no parameters.

Action: This statement makes all records associated with the currently active summary record that are any level lower than that summary record to become visible. For example, if you are on the highest level summary record when you use the **expandall** statement it will make all selected records visible.

If the active record is not a summary record (data record) this statement is ignored by Panorama.

The visible record count will accurately change after a proper expand is performed.

This statement has the same effect as clicking on the **Expand All** tool on a tool palette (when available).

Examples: This example goes to the last record in the database and, if it is a summary record, expands all levels below that summary record's level.

```
lastrecord
expandall
```

This example uses the [collapse](#), [expand](#), and **expandall** statements in a procedure to calculate percentages of Balance values for each data record associated with a specific summary 1 record for that group.

```
local BalTotal
field «GL Category»
groupup
field Balance
total
outlinelevel 1
firstrecord
field Percentage
loop
  if info("summary") =1
    BalTotal = Balance
    expand
    formulafill (Balance/BalTotal)*100
    collapse
  endif
  downrecord
until info("eof") or info("summary") >1
lastrecord
field Percentage
fill "100"
Percentage = zeroblank(0)
expandall
```

Views: This statement may be used in the Data Sheet or Form views only.

See Also:

[average](#) statement
[count](#) statement
[collapse](#) statement
[expand](#) statement
[group](#) statement
[groupdown](#) statement
[groupup](#) statement
[info\("records"\)](#) function
[info\("selected"\)](#) function
[info\("summary"\)](#) function
[maximum](#) statement
[minimum](#) statement
[outlinelevel](#) statement
[removesummaries](#) statement
[selectsummaries](#) statement
[total](#) statement

EXPORT

Syntax: EXPORT file,formula

Description: The **export** statement exports the current database into a text file. If the text file does not already exist it will be created. **If it does already exist its contents will be replaced!**

Parameters: This statement has two parameters: file and formula.

file is the name of the file you wish to save. If you are using a Macintosh, the file name may be up to 31 characters long, and may not contain / characters. If you are using windows, the file name may be up to 255 characters long and may not contain the \ or : characters.

If the file should be saved in a different folder than the current database the file name must contain a complete path description. A path description is a list of the folders the file is in, with each folder separated by a colon (for example [Disk:Joe:January](#) or [C:\MyStuff\January.txt](#)). You can create a path from a folder id with the [folderpath\(\)](#) function. The path and file name may be up to 255 characters long.

formula is the formula that will be used to extract data from the database and build each line in the text file. If the formula results in empty text ("") for a record then no line is added to the text file for that record. The formula usually references fields in the database being exported. It may also use the [seq\(\)](#) function to find out the number of each record.

To export line items add a Ω (**Option-Z** or **ALT-0166**) after the line item field names in the formula. Each line item will be output on a separate line. Warning: All the line item fields must have the same number of fields, for example Qty1...Qty6, Price1...Price6, etc. If some line items have more or less fields, the export will not work correctly. Remember, any field name that ends with a number will be treated as a line item, so if your invoice database contains fields named Address1 and Address2 you will not be able to export the line items using this method.

Action: This statement exports data by scanning the current database and processing each record with a formula (similar to [arraybuild](#), but builds a text file instead of an array). The result of the formula is placed into a text file. By changing the formula you can control what data is exported, and the arrangement of the data.

Examples: This example exports all fields from all selected records in the current database. The procedure asks the user to specify the name and folder for the new file.

```
local file, folder
savefiledialog folder, file, "Export file name"
if file="" stop endif
export folderpath(folder)+file, exportline()+¶
```

The procedure below creates a text file named List.TXT in that contains the Name, Address, City, State and Zip code (separated by tabs) of every record in Iowa.

```
select State="IA"
export "List.TXT", Name+--+Address+--+
City+--+State+--+Zip+¶
```


The procedure below outputs line items on separate lines in the text file. Each line will contain the date and the quantity, description and price for a specific line item, for example `3/21/97,4,Widgets,3.39`. The `Ω` symbols cause the export statement to output a separate line for each line item.

```
export "products",  
datepattern(Date, "mm/dd/yy" )+" , "+QtyΩ+" , "+DescriptionΩ+" , "+PriceΩ+¶
```

Views: This statement may be used in a Data Sheet or Form view.

See Also:

- [arraybuild](#) statement
- [filesave](#) statement
- [filetrash](#) statement
- [openfiledialog](#) statement
- [savefiledialog](#) statement
- [saveastext](#) statement
- [exportline\(\)](#) function
- [folder\(\)](#) function
- [folderpath\(\)](#) function
- [fileload\(\)](#) function
- [filesize\(\)](#) function

EXPORTCELL(...)

- Syntax:** EXPORTCELL(field)
- Description:** The `exportcell()` function takes any database field and converts it to text, using the appropriate pattern if one has been defined in the design sheet.
- Parameters:** This function has one parameter: `field`
field is the name of the field to be converted to text.
- Result:** The `export()` function always returns a text type data item. The power of the `exportcell()` function is that it does not require you to know what type of data you are exporting. It simply takes whatever kind of data is in the field (text, number, date, whatever) and converts it into text.
- Examples:** This example takes any database and outputs it in a comma separated format with quotes around each field.

```
global LineFormula
LineFormula=dbinfo("fields","")
arrayfilter LineFormula,LineFormula,¶,
  {" "+exportcell(«»)+import()+{»}+" " }
LineFormula=replace(LineFormula,¶,{+", "+"}+¶)
execute {export "Comma.txt",}+LineFormula
```

For example, the output of this procedure might look something like this if used with a checkbook database:

```
"145","12/5/97","Acme Widgets","356.78"
"146","12/6/97","Wilson Publishing","2,994.12"
"147","12/9/97","Yellow Freight","390.12"
```

If you've never used the `execute` statement before, this procedure may be a bit unclear. The first five lines of the procedure build a formula. The final formula will be something like this

```
" "+exportcell(«CheckNo»)+" "+", "+
" "+exportcell(«Date»)+" "+", "+
" "+exportcell(«Pay To»)+" "+", "+
" "+exportcell(«Amount»)+" "
```

The final line in the original procedure uses this formula to export the data from the database.

- Errors:** This function does not produce any errors.
- See Also:** [import\(\)](#) function
[exportline\(\)](#) function
[str\(\)](#) function
[pattern\(\)](#) function
[datepattern\(\)](#) function

EXPORTLINE(...)

Syntax: EXPORTLINE()

Description: The `exportline()` function generates a tab delimited line of data containing all the fields in the current record. This function is designed to be used with the [export](#) and [arraybuild](#) statements.

Parameters: This function has no parameters.

Result: The `exportline()` function returns a a tab delimited line of data containing all the fields in the current record. Any non-text fields (numeric, date) will be converted to text as they are placed into the tab delimited line. The tab delimited line does NOT include a carriage return on the end.

Examples: This function make it easy to export the entire database with a procedure.

```
local file,folder
savefiledialog folder,file,"Export file name"
if file="" stop endif
export folderpath(folder)+file,exportline()+¶
```

Here is a modified version of this procedure that converts any carriage returns inside a cell into vertical tabs as the data is being exported.

```
local file,folder
savefiledialog folder,file,"Export file name"
if file="" stop endif
export folderpath(folder)+file,
replace(exportline(),¶,chr(11))+¶
```

Errors: This function does not produce any errors.

See Also: [export](#) statement
[savefiledialog](#) statement
[replace\(\)](#) function

EXTRACT(...)

Syntax: `EXTRACT(text,separator,item)`

Description: The `extract()` function extracts a single data item from a text array (see [text arrays](#)). This function is almost identical to the `array()` function. The `extract()` function is excellent for extracting a word, line or phrase from a larger text item. It can also be used to count the number of items in the array.

Parameters: This function has three parameters: `text`, `separator`, and `item`.

text is the item of text that contains the data you want to extract.

separator is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the `\n` character (option-7). For tab delimited arrays use the `\t` character (option-L).

item is the number of the data item you want to extract. The first item is item 1, the second is item 2, etc.

Using an item number of -1 tells the `extract()` function to count the number of data items in the array. This is similar to the `arraysize()` function. In this case the `extract()` function will return a number, not text.

Result: If the item parameter is 1 or greater, this function returns an item of text from the array. Only the item itself is returned, the separator characters on each end are not included. If the item does not exist (for example if you ask for item 12 from a 7 item array) the function will return empty text (`""`). If the item parameter is -1, the `extract()` function returns a number—the number of data items in the array.

Examples: The example below displays the name of an element after the user enters the atomic number from 1 to 103. In this example the variable `Elements` contains an array of atomic element names, separated by semicolons (some of the assignment statement has been left out for clarity).

```

local Elements,AtomicNumber,AtomicName
Elements="Hydrogen;Helium;Lithium;Beryllium;Boron;" +
        "Carbon;Nitrogen;Oxygen;Fluorine;Neon;" +
        ...
        "Mendelevium;Nobelium;Lawrencium"
AtomicNumber="1"
gettext "Enter Atomic Number",AtomicNumber
AtomicNumber=val(AtomicNumber)
if error
    message "Atomic number must be an integer "+
           "between 1 and 103."
endif
AtomicName=extract(Elements,";",(AtomicNumber))
if AtomicName=""
    message "Atomic name is: "+AtomicName
else
    message "Atomic number must be an integer "+ "between 1 and 103."
endif

```

This example shows how the `extract()` function can be used to count the number of items in an array. This simple procedure displays the number of database files that are currently open.

```
message extract(info("files"), 1, -1)
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the array or separator parameters.

Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value for the item parameter.

Illegal function argument. This error occurs if you attempt to use zero as the item parameter. The item parameter must be 1 or greater to extract individual data items, or -1 to count data items.

See Also: [text arrays](#)
[array\(function](#)
[arraysize\(function](#)

EXTRAPAGES(...)

Syntax: EXTRAPAGES(pagelist)

Description: The `extrapages()` function is used to control the printing of extra pages. This function must be used in an auto-wrap text object, it has no effect in any other situation.

Parameters: This function has one parameter: `pagelist`.

pagelist is a text item listing the extra pages that should be printed. For example, if you want to print data tiles 3 and 5 the page list should be "35".

Result: This function returns an empty text item ("").

Examples: This function must be used in an auto-wrap text object on a form to be printed. The form must have extra data tiles (Data Tile 2, Data Tile 3, etc.) Panorama will always print the main data tile, but the `extrapages()` function can control what other data tiles are printed on a record by record basis.

For example, suppose that your form contains a statement (main data tile) and a reminder letter (data tile 2). The reminder letter should only be printed if the account is more than 45 days overdue. To do this put the formula below in an auto-wrap text object somewhere on the main data tile.

```
{extrapages(? ( today()-45>ShipDate,"2",""))}
```

Errors: This function does not generate any error messages.

See Also: [print](#) statement

FACT(...)

Syntax: FACT(value)

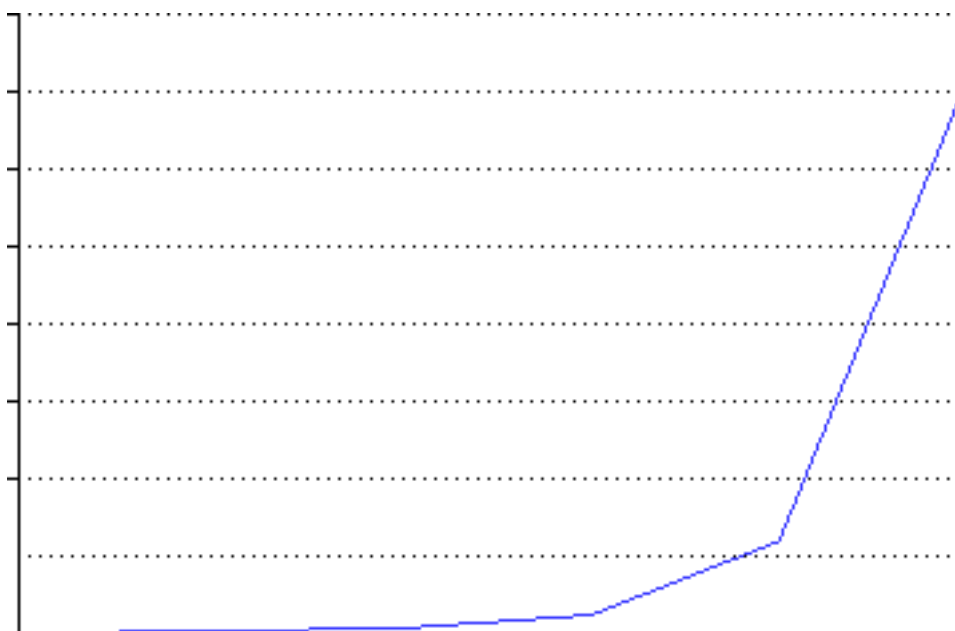
Description: The `fact` function computes the factorial of a value. The value must be an integer. A factorial is a value multiplied by the next lower value and the next lower etc. For example 4 factorial (written 4! by mathematicians) is the same as $4*3*2*1$. Factorials get big in a hurry as the value increases. Panorama cannot calculate a factorial for a value larger than 170. Larger values cause a floating point overflow because the answer is too large!

Parameters: This function has one parameter: value.

value is a numeric value. The value must be an integer between 1 and 170 (1, 2, 3, ... 169, 170).

Result: The result of this function is a numeric floating point value.

Examples: The graph below shows the result of the factorial function given input values from 1 to 6.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `fact("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating a factorial, for example `fact(val("34"))`.

Illegal function argument. This error occurs if you attempt to calculate the factorial of a negative value, or of a value greater than 170.

See Also: [val\(\)](#) function

FARCALL

Syntax: FARCALL file,procedure[,param1,param2,...paramN]

Description: The **farcall** statement executes another procedure, in a separate database open in memory, as a subroutine.

Parameters: This statement has two required parameter: **file** and **procedure**. There may also be additional parameters that you can define for passing data between the main program and the subroutine.

file is the name of the database that contains the procedure you wish to execute as a subroutine. This must be the name of a database currently open in memory. If file contains blanks, symbol characters, or punctuation characters it must be surrounded by quotes (" ") or chevrons (« »).

procedure is the name of the procedure you wish to execute as a subroutine. **Warning:** This must be the name of a procedure in the database specified by file and that database must be open in memory. If procedure contains blanks, symbol characters, or punctuation characters it must be surrounded by quotes (" ") or chevrons (« »).

parameter1, parameter 2, etc. are optional parameters that are defined by you, the subroutine programmer! Each parameter may be a field, variable, or a complete formula. (However, if you want to change the value of a parameter from inside the subroutine, the parameter must be a field or variable, not a formula.) The subroutine can find out the value of a parameter using the [parameter\(\)](#) function. To change the value of a parameter, the subroutine can use the [setparameter](#) statement.

Action: This statement allows a procedure programmer to execute a second procedure from a first as a subroutine to that procedure, however, it has the added advantage that the called procedure does not need to part of the active database. Subroutines become important when you wish to use the same code at different times within the same procedure or within multiple procedures. This allows you to write the code once and use it again and again rather than duplicating the same code over and over again. Farcall allows you use that same procedure for any Panorama database and only have it stored in one database.

Subroutines normally finish when the end of the procedure is reached. To stop the subroutine before the end of the procedure, use the [rtn](#) statement. The [rtn](#) statement makes Panorama return control to the original procedure.

Warning: Even though you have called a procedure in another database, the active database may not have the same field or form names as the one associated with the called procedure. This means that all references to field names or form names must match the names in the active database when the procedure is running.

Note: To call a procedure in the same database, use the [call](#) statement. To execute a subroutine within the same procedure use the [shortcall](#) statement.

Examples: This simple example shows how to call a procedure called Summarize from a database called Tax File.

```
farcall "Tax File",Summarize
```

This example calls a subroutine named Calculate % from the database called Calculator. The subroutine's name is more than one word and contains symbol characters.


```
farcall "Calculator","Calculate %"
```

The example below shows that a subroutine, named Conversion, can be called from within the middle of a procedure. Once the subroutine finishes the parent procedure will continue executing. **Note:** The parameter area (a global variable) is handed off to the subroutine.

```
global area,width,height  
gettext "Enter Width in inches.",width  
gettext "Enter Height in inches.",height  
area = val(width) * val(height)  
openfile "Calc"  
farcall "Calc",Conversion,area  
window "Calc"  
closefile  
«Sq. Ft.» = area  
field Dimensions  
...  
...
```

Views: This statement may be used in any view

See Also: [call](#) statement
[parameter\(\)](#) function
[rtn](#) statement
[setparameter](#) statement
[shortcall](#) statement

FIELD

Syntax: FIELD fieldname

Description: The **field** statement tells Panorama to move to the specified field for the current record.

Parameters: This statement has one parameter: fieldname

fieldname can be a literal field name, a quoted string, field, variable, or formula that returns the name of a valid field for the active database. This field name is case sensitive and must match the field names as they were established in the database's Design Sheet. If the field name is two or more words it must be enclosed in chevron characters « » to be recognized as a valid field name.

Note: If fieldname is a field or variable which contains the name of the field you wish to move to you must enclose it inside parenthesis () so that Panorama treats fieldname as a formula and not a literal value. See second example below.

Action: This statement moves Panorama's cursor to the field indicated by fieldname for the currently active record.

Note: If fieldname does not match a valid field for the active database Panorama will return an error alert.

This statement has the same effect as choosing a field name from the **Fields** menu in the Data Sheet.

Examples: This simple example tells Panorama to go to the Address field.

```
field Address
```

This example assumes a database that has a field for each day of the week Sunday, Monday, etc. The following procedure will place the cursor on the field matching the current day. To distinguish fieldName as a formula it must be enclosed inside parenthesis () characters.

```
local theFieldName
theFieldName = datepattern( today(), "Day" )
field (theFieldName)
```

This example goes from line item field Price 1 to Price 10 blanking out the fields as it goes along.

```
local TheCount
TheCount = 0
loop
  TheCount = TheCount + 1
  field "Price "+str(TheCount)
  clearcell
until TheCount = 10
```

This example shows a procedure that makes all choices type fields in the database center aligned.

```
opendesignsheet  
firstrecord  
field Align ;«Align» is a Design Sheet field  
loop  
  if «Type» = "Choices" ;«Type» is a Design Sheet field  
    cell "Center"  
  endif  
  downrecord  
until info("stopped")  
newgeneration  
closewindow  
save
```

Views: This statement may be used in any view.

See Also: [emptyfield](#) statement

FIELDMAX(...)

Syntax: FIELDMAX(fieldname)

Description: The `fieldmax()` function returns the maximum number of characters that can be stored in a field.

Parameters: This function has one parameter: `fieldname`.

fieldname is the name of the field that you want to calculate the maximum size of.

Result: The function returns the maximum number of bytes that can be stored in this field. If this is not an SQL client database, this number is always 65535. If this is an SQL client, this function returns the length of the corresponding SQL field in the server database.

Examples: Use the `fieldmax()` function when you are programming a SQL database. The function below looks up the city name and if it will fit it stores it in the `City` field.

```
local theCity
theCity=city(Zip)
if length(theCity)≤fieldmax(City)
    City=theCity
else
    message "The city name is too long."
endif
```

Errors: **Field or variable does not exist.** This error occurs if there is no field in the current database with the name you have specified. You probably misspelled the field name.

See Also: [length\(\)](#) function
[sizeof\(\)](#) function

FIELDNAME

Syntax: FIELDNAME fieldname

Description: The **fieldname** statement changes the current field's name to that specified by fieldname.

Parameters: This statement has one parameter: fieldname
fieldname can be in the form of a quoted string, field, variable, or formula that returns a text value.

Action: This statement changes the name of the currently active field to the value in fieldname.

Examples: This example changes the active field's name to Notes.

```
fieldname "Notes"
```

This example changes the current field's name to Ratio, converts it to a floating point numeric field, and then calculates values for Ratio.

```
fieldname "Ratio"  
fieldtype "float"  
formulafill "Price/Cost"
```

Views: This statement may be used in the Data Sheet and Form views **only**.

See Also: [addfield](#) statement
[dbinfo\(\)](#) function
[deletefield](#) statement
[fieldstyle\(\)](#) function
[fieldtype](#) statement
[info\("datatype"\)](#) function
[info\("fieldname"\)](#) function
[insertfield](#) statement
[newgeneration](#) statement

FIELDSTYLE(...)

Syntax: FIELDSTYLE(fieldname)

Description: The `fieldstyle()` function determines the style and color of a field (see also the [style](#) statement, which can set the style and color of a field).

Parameters: This function has one parameter: `fieldname`.

fieldname is the name of the field that you want to determine the style of.

Result: The function returns a text data item listing all the styles that apply to this field in the current record. The possible styles are:

```
bold
italic
underline
shadow
black
red
green
blue
cyan
magenta
yellow
```

Examples: The example below selects all the records where the name is bold.

```
select fieldstyle(Name)="bold"
```

If there is more than one style for a cell, this function will list each one. The example below will select all records where the name is italic, even if other styles also apply (for example bold italic or underline italic).

```
select fieldstyle(Name) contains "italic"
```

This final example selects all the records where the name is plain (no styles at all).

```
select fieldstyle(Name)=""
```

Errors: **Field or variable does not exist.** This error occurs if there is no variable or field in the current database with the name you have specified. You probably misspelled the field or variable name.

See Also: [style](#) statement

FIELDTYPE

Syntax: FIELDTYPE datatype

Description: The **fieldtype** statement changes the current field to the specified datatype.

Parameters: This statement has one parameter: datatype

datatype must be one of the listed values below. It can be in the form of a quoted string, field, variable, or formula that returns one of the text values listed below.

```
text
0 digits
1 digit
2 digits
3 digits
4 digits
float
date
choices
picture
```

Action: This statement changes the type of the currently active field to that specified in datatype. If datatype contains the word digit or float Panorama will convert both the field's type to Numeric and set the number of digits to match datatype.

Note: If the contents of the currently active field will not be compatible with the new fieldtype Panorama will warn you with an alert dialog. You may hit Cancel, Ok, or Select Problem Data from this dialog. This warning dialog cannot be suppressed.

Examples: This simple example changes the current field to the a 4-digit numeric field.

```
fieldtype "4 digits"
```

This example creates a new field called Ratio, converts it to floating point, and then calculates values for the new field.

```
addfield "Ratio"
field "Ratio"
fieldtype "float"
formulafill "Price/Cost"
```

Views: This statement may be used in the Data Sheet and Form views only.

See Also: [addfield](#) statement
[dbinfo\(\)](#) function
[deletefield](#) statement
[field](#) statement
[fieldname](#) statement
[fieldstyle\(\)](#) function

info("datatype") function
info("fieldname") function
insertfield statement
newgeneration statement

FILEAPPEND

Syntax: `FILEAPPEND folder,filename,typecreator,data`

Description: The `fileappend` statement saves data directly into a file. If the file does not already exist it will be created. If the file does already exist the new data will be appended to the existing data.

Parameters: This statement has four parameters: `folder`, `filename`, `typecreator` and `data`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

filename is the name of the file you wish to save. The file name may be up to 31 characters long, and may not contain / characters.

typecreator is the type and creator of the new file. This parameter only applies when using Panorama on a Macintosh system - the typecreator parameter is ignored when using Panorama on a Windows system. The type and creator identify the type of file and the application that should be launched when the file is double clicked, and each is four characters long. If the typecreator is "" then the type and creator default to `TEXTtxt`, the type and creator for SimpleText text files.

The 4 character TYPE code identifies the type of data in the file. Here are some 4 character types codes:

Type Code	Description
TEXT	Text File
PICT	Picture File
ESPF	Encapsulated Postscript
APPL	Application (Program)
ZEPD	Panorama Database
KSET	Panorama File Set
GIFf	GIF Image File
JPEG	JPEG Image File
MOOV	QuickTime Movie
PDF	Adobe Acrobat PDF Format

The 4 character CREATOR code identifies what application created the file (and will be launched when the file is double clicked.) Here are some 4 character creators:

Creator Code	Description
KASX	Panorama 3.5 or later
KAS1	Panorama 3.1 or earlier
CARO	Adobe Acrobat
ToyS	AppleScript Editor
WILD	HyperCard
XCEL	Microsoft Excel
WBDN	Microsoft Word
8BIM	Adobe Photoshop
ttxt	SimpleText
R*ch	BBEdit
TVOD	MoviePlayer
MPS	Macintosh Programmers Workshop (MPW)

data is the actual data that will be placed into the file. This may be in a field, variable, or constructed with a formula.

Action: This statement saves data directly into a disk file. Using this statement you can create any kind of disk file you want.

Examples: The procedure below adds to a text file named TIME STAMP in the system folder that contains the current date and time.

```
fileappend info("systemfolder"), "TIME STAMP", "",
datepattern( today(), "mm/dd/yy ")+"@"+
timepattern( now(), "hh:mm:ss am/pm")+¶
```

Each time this procedure runs, another line will be added to the TIME STAMP file. The result will be something like this:

```
3/17/98 @3:42:03 PM
3/17/98 @4:18:28 PM
3/18/98 @10:21:51 PM
3/18/98 @2:07:41 PM
3/19/98 @8:47:09 PM
```

Views: This statement may be used in any view.

See Also: [filesave](#) statement
[filerename](#) statement
[filetrash](#) statement
[datafork](#) statement
[resourcefork](#) statement
[openfiledialog](#) statement
[savefiledialog](#) statement

[folder\(\)](#) function
[folderpath\(\)](#) function
[fileload\(\)](#) function
[filesize\(\)](#) function

FILEGLOBAL

Syntax: FILEGLOBAL variables

Description: The **fileglobal** statement creates one or more global variables that are specific to the current database. File global variables may be used by any procedure as long as the same database is open, and remain active until you close the database.

Parameters: This statement has one parameter: **variables**.

variables is a list of variables to be created. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: This statement creates one or more file global variables. Variables can be used to hold pieces of information (numbers or text). Each variable has a name.

The **fileglobal** statement reduces that chance for conflict between databases. If two different database define a global variable with the same name, they can conflict with each other. (On the other hand, the two database can also use the global variable to communicate with each other.)

Fileglobal variables are not shared between databases. If two different databases have **fileglobal** variables with the same name, they will not conflict with each other. In fact, Panorama will keep two separate variables...one for each database. When you are using the **fileglobal** statement you don't have to worry about whether your variable names conflict with variables in any other database. (You do still need to worry about conflicts with field names, however. You should not define a fileglobal variable with the same name as one of the fields in your database.)

Globals vs. File Globals

Since using **fileglobal** eliminates conflicts, why would you ever use the global statement to define variables? Most of the time you would not, and you may want to review your existing databases and change most or all of the global statements to fileglobals. However, there are two cases where you may want to continue to use global variables.

If you need to share data between multiple databases, a global variable may be the way to go. (However, it is also possible to access fileglobal variables from another file using the [grabfilevariable\(\)](#) function.)

If you need the variable to remain even after the database is closed, use a global variable. Global variables remain active until you quit from Panorama. Variables created with **fileglobal** disappear when the file is closed.

Examples: The example creates two fileglobal variables, Channel and Home Timeouts.

```
fileglobal Channel, «Home Timeouts»
```

You may change the value of a variable with an assignment, like this:

```
«Home Timeouts»=«Home Timeouts»-1
```

However, if you switch to another database, the fileglobal variables can no longer be accessed.

```
window "New Zealand" /* switch to another db */
Channel=7/* this will produce a runtime error! */
```

Note: If you need to get the value of a fileglobal variable in another database you can use the [grabfilevariable\(\)](#) function.

Views: This statement may be used in any view.

See Also: [global](#) statement
[local](#) statement
[windowglobal](#) statement
[permanent](#) statement
[globalize](#) statement
[grabfilevariable\(\)](#) function
[undefine](#) statement

FILEINFO(...)

Syntax: FILEINFO(folder,filename)

Description: The `fileinfo()` function gets information about a file (or folder) on the disk, including the size, creation and modification date and time, type, creator and lock status.

Parameters: This function has two parameters: `folder` and `filename`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder containing the file. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `pathid()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements.

filename is the name of the file (or folder) you want to get information about.

Result: This function returns a text array with 8 elements separated by carriage returns. (However, if the specified file does not exist it returns empty text ("")). The eight elements are:

Type of item. This is either "File" or "Folder".

Type (4 bytes) and **Creator** (4 bytes). This identifies the file's type and creator codes (Macintosh only). Here are some typical Type/Creator values:

Type/Creator	Description
ZEPDKASX	Panorama Database (3.5 or later)
KSETKASX	Panorama File Set (3.5 or later)
ZEPDKAS1	Panorama Database (3.1 or earlier)
ZEPDKAS1	Panorama File Set (3.1 or earlier)
JPEG8BIM	JPEG Image (Photoshop)
TEXTttxt	Text File (SimpleText)
MooVTVOD	QuickTime Movie (MoviePlayer)

This is only a small sample of the types and creators you will find on your hard disk. (On Windows computers this field will be ????????)

Creation Date in internal Panorama format. Although this is a number, it has been converted to text. If you convert the number back to text you can format the date with `datepattern()`.

Creation Time in seconds since midnight. Although this is a number, it has been converted to text. If you convert the number back to text you can format the time with `timepattern()`.

Modification Date in internal Panorama format. Although this is a number, it has been converted to text. If you convert the number back to text you can format the date with `datepattern()`.

Modification Time in seconds since midnight. Although this is a number, it has been converted to text. If you convert the number back to text you can format the time with `timepattern()`.

File size in bytes. (Or if the specified file is actually a directory, this is the number of files in directory)

File status: This is either "Locked" or "Unlocked" .

Examples: Here is what the result of FileInfo might look like for a typical Panorama database:

```
File
ZEPDKAS1type (ZEPD) and creator (KAS1)
2449476creation date (5/3/94)
43423 creation time (12:03:43 PM)
2449834modification date (4/26/95)
85804modification time (11:50:04 PM)
45769file size
Unlocked
```

This example displays the creation date and size of the currently running copy of Panorama.

```
local PanoramaInfo
PanoramaInfo=fileinfo(info("panoramafolder"), "Panorama")
message "This copy of Panorama was created on "+
datepattern( val( array(PanoramaInfo,3,¶)), "mm/dd/yy")+
" and is "+
array(PanoramaInfo,7,¶)+" bytes long."
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric item for the folder or filename parameters

See Also: [folder](#)(function
[folderpath](#)(function
[listfiles](#)(function
[info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#) function

FILELOAD(...)

Syntax: `fileload(folder,file)`

Description: The `fileload(` function reads the entire contents of any file on disk. It is especially useful for reading text files.

Parameters: This function has two parameters: `folder` and `file`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder(`, `dbinfo(` and some `info(` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

file is the name of the file that is to be read.

Result: This function returns the entire contents of the file as an item of text. (Technical note: Macintosh files may be split up into two components, called the "data fork" and the "resource fork." The `fileload(` function normally reads the data fork, but not the resource fork. If you wish to read the resource fork you must use the `getresource(` and related functions or use the `resourcefork` statement.)

Examples: The example below reads the contents of the Macintosh notebook file.

```
message fileload(info("systemfolder"),"Note Pad File")
```

The example below creates a list of all text files in the current folder that contain the word "Internet."

```
local fileList,fileFolder
fileFolder=dbinfo("folder","")
fileList=listfiles(fileFolder,"TEXT????")
arrayfilter fileList,fileList,¶,
    ?(fileload(fileFolder,import()) contains "Internet",
    import(),"")
fileList = arraystrip(fileList,¶)
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder or file parameters.

File not found. The specified file does not exist.

File is too large, cannot load. The specified file is larger than the available memory and cannot be read. If you trap this with if error you can try reading part of the file with the `fileloadpartial(` function.

See Also: `fileloadpartial(` function
`filesize(` function
`filesave` statement
`datafork` statement
`resourcefork` statement

[openfiledialog](#) statement
[folder\(\)](#) function
[folderpath\(\)](#) function

FILELOADPARTIAL(...)

Syntax: FILELOADPARTIAL(folder,file,start,length)

Description: The `fileloadpartial`(function reads a portion of the contents of any file on disk. It is especially useful for reading text files.

Parameters: This function has four parameters: `folder`, `file`, `start` and `length`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder`(, `dbinfo`(and some `info`(functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

file is the name of the file that is to be read.

start is the first byte (character) of the file that should be read. This function assumes bytes in the file are numbered starting from 0.

length is the number of bytes that should be read.

Result: This function returns a portion of the contents of the file as an item of text. (Technical note: Macintosh files may be split up into two components, called the "data fork" and the "resource fork." The `fileloadpartial`(function normally reads the data fork, but not the resource fork. If you wish to read the resource fork you must use the `getresource`(and related functions or use the `resourcefork` statement.)

Examples: The example below displays the contents of the first line of the Macintosh notebook file. A maximum of 512 bytes is used, so this procedure will work even if the note pad contains megabytes of information.

```
message
array(
  fileloadpartial(
    info("systemfolder"),
    "Note Pad File",0,512),1,¶)
```

The example below searches the text file "System Log" for a word. The search is conducted a small section at a time (2000 bytes) so that it will work properly (although somewhat slowly) even if the System Log file is very large.

```
local searchWord,fileSpot,fileChunk
searchWord=""
gettext searchWord,"Search for:"
fileSpot=0
loop
  fileChunk=fileloadpartial("", "System Log",fileSpot,2000)
  stoploopif fileChunk="" or fileChunk contains searchWord
  fileSpot=fileSpot+2000-length(searchWord)
while forever
  if fileChunk contains searchWord
    message "Found "+searchWord+" in System Log"
  endif
```

- Errors:**
- Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a number for the folder or file parameters.
 - Type mismatch: text argument used when number was expected. This error occurs if you attempt to use text for the start or length parameters.
 - File not found. The specified file does not exist.
 - File is too large, cannot load. The specified portion of the file is larger than the available memory and cannot be read. If you trap this with if error you can try reading a smaller portion.

- See Also:**
- [fileload\(\)](#) function
 - [filesize\(\)](#) function
 - [filesave](#) statement
 - [openfiledialog](#) statement
 - [folder\(\)](#) function
 - [folderpath\(\)](#) function

FILERENAME

Syntax: `FILERENAME folder,filename,newfilename`

Description: The `filerename` statement renames a disk file.

Parameters: This statement has three parameters: `folder`, `filename` and `newfilename`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder that contains the file. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the [folder\(\)](#), [dbinfo\(\)](#) and some [info\(\)](#) functions, and the [openfiledialog](#) and [savefiledialog](#) statements. If this parameter is empty text ("") the folder containing the current database is assumed.

filename is the current name of the file.

newfilename is the new name of the file. The new file name may be up to 31 characters long. On MacOS systems the filename may not contain the `:` character. On Windows systems the filename may not contain the `\`, `*` or `?` characters. There must not already be a file with the same name in the folder.

Action: This statement changes the name of a file on the disk. This is the same as if you clicked on the file's name in the finder, then typed in a new name.

Examples: The procedure below renames a file named Sales. This file is in the same folder as the currently open database. The new name for this file is based on the date, it will be something like `Sales (10/97)`.

```
filerename "", "Sales",
"Sales"+" (" +datepattern( today()-30, "mm/yy")+ ")"
```

Views: This statement may be used in any view.

See Also: [filesave](#) statement
[filetypecreator](#) statement
[filetrash](#) statement
[openfiledialog](#) statement
[folder\(\)](#) function
[folderpath\(\)](#) function
[fileload\(\)](#) function
[filesize\(\)](#) function

FILESAVE

Syntax: FILESAVE folder,filename,typecreator,data

Description: The `filesave` statement saves data directly into a file. If the file does not already exist it will be created. **If it does already exist its contents will be replaced!** If you want to add to the contents of the file, use the [fileappend](#) statement.

Parameters: This statement has four parameters: folder, filename, typecreator and data.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the [folder\(\)](#), [dbinfo\(\)](#) and some [info\(\)](#) functions, and the [openfiledialog](#) and [savefiledialog](#) statements. If this parameter is empty text ("") the folder containing the current database is assumed.

filename is the name of the file you wish to save. The file name may be up to 31 characters long. On MacOS systems the filename may not contain the `:` character. On Windows systems the filename may not contain the `\`, `*` or `?` characters.

typecreator is the type and creator of the new file. The type and creator identify the type of file and the application that should be launched when the file is double clicked, and each is four characters long. If the typecreator is "" then the type and creator default to TEXTtxt, the type and creator for SimpleText text files.

The 4 character TYPE code identifies the type of data in the file. Here are some 4 character types codes:

Type Code	Description
TEXT	Text File
PICT	Picture File
ESPF	Encapsulated Postscript
APPL	Application (Program)
ZEPD	Panorama Database
KSET	Panorama File Set
GIF8	GIF Image File
JPEG	JPEG Image File
MOOV	QuickTime Movie
PDF	Adobe Acrobat PDF Format

The 4 character CREATOR code identifies what application created the file (and will be launched when the file is double clicked.) Here are some 4 character creators:

Creator Code	Description
KASX	Panorama 3.5 or later
KAS1	Panorama 3.1 or earlier
CARO	Adobe Acrobat
ToyS	AppleScript Editor
WILD	HyperCard
XCEL	Microsoft Excel
WBDN	Microsoft Word
8BIM	Adobe Photoshop
ttxt	SimpleText
R*ch	BBEdit
TVOD	MoviePlayer
MPS	Macintosh Programmers Workshop (MPW)

data is the actual data that will be placed into the file. This may be in a field, variable, or constructed with a formula.

Action: This statement saves data directly into a disk file. Using this statement you can create any kind of disk file you want.

Examples: The procedure below creates a text file named **TIME STAMP** in the system folder that contains the current date and time.

```
filesave info("systemfolder"),"TIME STAMP","",
datepattern( today(),"mm/dd/yy ")+"@"+
timepattern( now(),"hh:mm:ss am/pm")+¶
```

Views: This statement may be used in any view.

See Also: [fileappend](#) statement
[filerename](#) statement
[filetypecreator](#) statement
[filetrash](#) statement
[datafork](#) statement
[resourcefork](#) statement
[openfiledialog](#) statement
[savefiledialog](#) statement
[folder](#) function
[folderpath](#) function
[fileload](#) function
[filesize](#) function

FILESIZE(...)

Syntax: `filesize(folder,file)`

Description: The `filesize()` function determines the size of any file on disk.

Parameters: This function has two parameters: `folder` and `file`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

file is the name of the file.

Result: This function returns a number—the size of the entire contents of the file. (Technical note: Macintosh files may be split up into two components, called the “data fork” and the “resource fork.” The `filesize()` function reads the size of the data fork, but not the resource fork.)

Examples: The example below displays the size of the Macintosh notebook file.

```
message filesize(info("systemfolder"),"Note Pad File")
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder or file parameters.

File not found. The specified file does not exist.

See Also: [fileload\(\)](#) function
[fileloadpartial\(\)](#) function
[filesave](#) statement
[openfiledialog](#) statement
[folder\(\)](#) function
[folderpath\(\)](#) function

FILETRASH

Syntax: FILETRASH folder,filename

Description: The `filetrash` statement erases a disk file. This is the same as dragging the file into the trash can and choosing Empty Trash from the Special menu.

Parameters: This statement has two parameters: `folder` and `filename`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder that contains the file you want to erase. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

filename is the name of the file to be erased.

Action: This statement erases a file on the disk.
There is no way to get the file back, so be careful!

Examples: The procedure below erases a file named Invoice Worksheet. This file is in the same folder as the currently open database.

```
filetrash "", "Invoice Worksheet"
```

Views: This statement may be used in any view.

See Also: [filesave](#) statement
[filerename](#) statement
[openfiledialog](#) statement
[folder\(\)](#) function
[folderpath\(\)](#) function
[fileload\(\)](#) function
[filesize\(\)](#) function

FILETYPECREATOR

Syntax: FILETYPECREATOR folder,filename,typecreator

Description: The `filetypecreator` statement allows a procedure to change the type and creator of a file.

Parameters: This statement has three parameters: folder, filename, and typecreator.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.

filename is the name of the file you wish to save. The new file name may be up to 31 characters long. On MacOS systems the filename may not contain the `:` character. On Windows systems the filename may not contain the `\`, `*` or `?` characters.

typecreator is the type and creator of the new file. The type and creator identify the type of file and the application that should be launched when the file is double clicked, and each is four characters long. If the typecreator is "" then the type and creator default to TEXTttxt, the type and creator for SimpleText text files. Here are some other 4 character types:

The 4 character TYPE code identifies the type of data in the file. Here are some 4 character types codes:

Type Code	Description
TEXT	Text File
PICT	Picture File
ESPF	Encapsulated Postscript
APPL	Application (Program)
ZEPD	Panorama Database
KSET	Panorama File Set
GIFf	GIF Image File
JPEG	JPEG Image File
MOOV	QuickTime Movie
PDF	Adobe Acrobat PDF Format

The 4 character CREATOR code identifies what application created the file (and will be launched when the file is double clicked.) Here are some 4 character creators:

Creator Code	Description
KASX	Panorama 3.5 or later
KAS1	Panorama 3.1 or earlier
CARO	Adobe Acrobat
ToyS	AppleScript Editor
WILD	HyperCard
XCEL	Microsoft Excel
WBDN	Microsoft Word
8BIM	Adobe Photoshop
ttxt	SimpleText
R*ch	BBEdit
TVOD	MoviePlayer
MPS	Macintosh Programmers Workshop (MPW)

Action: This statement changes a disk file's type and creator codes (Macintosh only). Using this statement you can control what icon appears for this file and what application is launched when the file is double clicked.

Examples: The procedure below examines a file that has been transferred from a PC computer. Depending on the three character "extension" at the end of the filename, it converts the file into a text file, a Panorama file, or a Photoshop picture file.

```

case myfile endswith ".txt"
  filetypecreator myfolder,myfile,"TEXTttxt"
case myfile endswith ".pan"
  filetypecreator myfolder,myfile,"ZEPDKAS1"
case myfile endswith ".pct"
  filetypecreator myfolder,myfile,"PICT8BIM"
endcase

```

Views: This statement may be used in any view.

See Also: [fileappend](#) statement
[filerename](#) statement
[filesave](#) statement
[filetrash](#) statement
[datafork](#) statement
[resourcefork](#) statement
[openfiledialog](#) statement
[savefiledialog](#) statement
[folder\(\)](#) function

[folderpath\(\)](#) function
[fileload\(\)](#) function
[filesize\(\)](#) function

FILL

Syntax: FILL value

Description: The **fill** statement fills every visible cell in the active field with the specified value.

Parameters: This statement has one parameter: value

value can be a literal, a field or a variable, or a formula that returns the value you wish to put into the cells of the current field. The resulting value must match the type and format (if any) of the field you are placing it in or an error will result. If value is the word dialog Panorama will present you with an input dialog so you may enter your own value.

Action: This statement evaluates value and if it is compatible with the current field type writes it to every cell in that field for all selected records only.

Warning: If a cell already has a value in it fill will replace that value with the new one.

When value is the word dialog Panorama will pause the procedure and present the user with the **Fill... dialog** allowing the user to enter their own value. Clicking on the Ok button will allow the procedure to continue. This statement will work for all field types except picture type fields.

This statement has the same effect as using the **Fill** command from the **Math** menu.

Examples: This simple example tells Panorama to clear all the selected cells of the active field. Therefore, the active field should be a text or choice type field.

```
fill ""
```

This similar example works for numeric or date type fields.

```
fill zeroblank(0)
```

This example tells Panorama to fill all the selected cells of the active field with the text string n/a. Therefore, the active field should be a text or choice type field.

```
fill "n/a"
```

This example fills the numeric field Starting Value with the number 1200.

```
field "Starting Value"
fill 1200
```

This example will have Panorama open the **Fill... dialog** which allows you to enter a value to be filled into the visible cells in the current field.

```
fill dialog
```

This example uses a formula to fill the visible cells in the field Date to Renew with a date that is one year beyond the computer's current date. The final date will be the 1st of the month.

```
field «Date to Renew»
fill datepattern( month1st( today() + 365 ), "mm/dd/yy" )
```

This example changes all the field types in the database to text type fields.

[opendesignsheet](#)
[field](#) Type
[fill "Text"](#)
[newgeneration](#)
[closewindow](#)

Views: This statement may be used in any view.

See Also: [emptyfill](#) statement
[formulafill](#) statement

FIND

Syntax: FIND true-false test

Description: The FIND statement locates the first visible record, if any, which matches the true-false test for the active database.

Parameters: This statement has one parameter: true-false test.

true-false test may be one or more functions or equations which result in a true or a false condition. Multiple true-false tests must be separated by an **and** or an **or** operator. Grouping true-false tests inside parenthesis () will give those tests priority in the processing order when Panorama evaluates them.

Action: This statement is used to locate and make active the first record, starting from the beginning of the database, that matches the true portion of the true-false test. If a subset of records are selected from the database find will only examine that sub-set for a match. You do not need to have the cursor on the field you are performing the find on prior to executing the find.

Compound true-false tests connected by an **or** operator(s) requires only one of the tests to be true to make the test true. Compound true-false tests connected by an **and** operator(s) requires all tests to evaluate true to make the test true.

If no records match a true test then Panorama will leave the cursor where it was before the find statement executed. You may test to see if any records were found by using the [info\("found"\)](#) function as the parameter of an [if](#) statement immediately after the find statement (see examples below.)

This statement will work on all fields except **Picture** type fields.

This statement has the same effect as clicking on the **Find** button in the Find/Select dialog, **Search** menu.

Examples: This example makes the first visible record for John Smith the active record.

```
find Customer = "John Smith"
```

This example makes the first visible record over 850 pounds the active record.

```
find Weight > 850
```

This example makes the first visible record for Jan. 1995 the active record.

```
find «Date Opened» >= date("1/1/95")
```

This example shows how you can test to see if a record was found matching the input Work Order No. using the [info\("found"\)](#) function.

```
getscrap "Enter Find value."
find «Work Order No.» = val( clipboard() )
if info("found")
    «Mark Record» = "X"
else
    message "There is no record for Work Order: "+
    clipboard()
```

```

    stop
endif
...

```

In this example the [info\("found"\)](#) function is used to test to see if no record was found for the date entered. Since the **not** operator is used, the entire formula must be enclosed inside parenthesis **()**.

```

local ADate
gettext "Enter date ex. 12/31/95",ADate
find «Process Date» = date(ADate)
if (not info("found"))
    beep
    message "No records match this date: "+ADate
    stop
endif
field «Sales Rep»
...

```

This example uses a compound true-false test that must match a name to either Customer or Company and who has made a Purchase Price more than **\$500**.

```

local Name
gettext "Enter name:",Name
find (Customer contains Name or Company contains
Name) and «Purchase Price» > 500
if (not info("found"))
    beep
    message "No records match found."
endif

```

This example uses a formula to test multiple text fields to find the record where the variable's contents matches any one of the fields.

```

local TheText
gettext "Enter text to find:",TheText
find " "+Customer+" "+Company+" "+Address+" "+
Comments contains TheText
if (not info("found"))
    beep
    message "No matching records found."
endif

```

Views: This statement may be used in any view

See Also: [findselect](#) statement
[formulafindselect](#) statement
[info\("empty"\)](#) function
[info\("found"\)](#) function
[info\("records"\)](#) function
[info\("selected"\)](#) function
[next](#) statement
[select](#) statement
[selectadditional](#) statement
[selectall](#) statement

selectreverse statement
selectsummaries statement
selectwithin statement

FINDSELECT

Syntax: FINDSELECT

Description: The `findselect` statement pauses a procedure while opening the Find/Select dialog allowing the user to manually perform one of the find or select options.

Parameters: This statement has no parameters.

Action: This statement pause the procedure while displaying the standard Find/Select dialog. The user can then enter their search criteria and choose one of the four search options: **Find**, **Select**, **Select Additional**, or **Select Within** or they can hit the **Cancel** button. Hitting the **Cancel** button will cancel the search operation, but it will not terminate the procedure. The procedure will continue running after the `findselect` statement.

Compound searches may be performed by expanding the Find/Select dialog by clicking on the double arrow icon pointing downward (at the left of the dialog.)

This statement **cannot** be used in combination with the `info("found")` or `info("empty")` functions as the parameter of an if statement immediately after the `findselect` statement to test for a null `find` or `select`.

All search options will work on all fields except **Picture** type fields.

This statement has the same effect as choosing the **Find/Select** command from the **Search** menu.

Examples: This example opens the Find/Select dialog.

```
findselect
```

This example allows you to use the Find/Select dialog and then it sorts the selected records and brings you to the first one.

```
message "Use the upcoming dialog to select the records you wish to work
with."
findselect
field «Company Name»
sortup
firstrecord
```

Views: This statement may be used in the Data Sheet, Form views and Cross Tab views **only**.

See Also: [find](#) statement
[formulafindselect](#) statement
[info\("empty"\)](#) function
[info\("found"\)](#) function
[info\("records"\)](#) function
[info\("selected"\)](#) function
[next](#) statement
[select](#) statement
[selectadditional](#) statement
[selectall](#) statement

selectreverse statement
selectsummaries statement
selectwithin statement

FINDWINDOW(...)

Syntax: FINDWINDOW(point)

Description: The `findwindow()` function checks to see if a point (in screen relative co-ordinates) is inside any Panorama window. If it is inside a window, this function returns the name of the window.

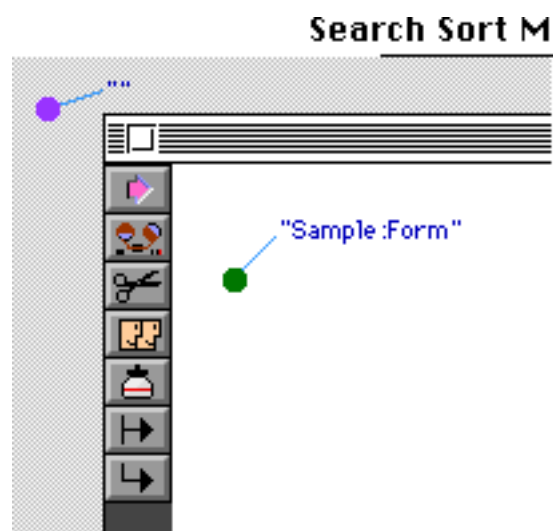
Parameters: This function has one parameter: `point`.

point is a point. This point must be in screen relative co-ordinates. All measurements are in pixels (1 pixel = 1/72 inch).

Result: This function returns a text item. If the point is inside a Panorama window, the function returns the name of the window. You can use the window statement to bring this window to the top. If the point is not inside any Panorama window the function returns empty text ("").

Examples: This illustration shows a window and two points. The green point is inside the window, so the `findwindow()` function will return the window name, in this case `Sample:Form`. The purple point is not inside the window, so the `findwindow()` function will return "".

Here's an example that uses `findwindow()` as part of a drag and drop procedure. The procedure lets the user drag from a button. When the user releases the mouse the procedure checks to see if the mouse is over a Panorama window, and if so, attempts to drag the name and address to the database for this window. (This procedure is designed to be triggered by a pushbutton with the click/release option turned off.)



```

local drag,landingWindow,landingDatabase,landingFields
local dragname,dragAddress,dragCity,dragState,dragZip
dragName=Name
dragAddress=Address
dragCity=City
dragState=State
dragZip=Zip
drag=info("buttonrectangle")
draggraybox drag,"",",",0
if drag="" stop endif
landingWindow=findwindow(info("mouse"))
if landingWindow=""
    stop
endif
landingDatabase=stripchar(landingWindow[1,":"],"!9;ÿ")

```

```
if landingDatabase=info("databasename")
  stop
endif
landingFields=dbinfo("fields",landingDatabase)
if (not (landingFields contains "Name" and
  landingFields contains "Address" and
  landingFields contains "City" and
  landingFields contains "State" and
  landingFields contains "Zip"))
  message "Cannot drag name/address to this database"
  stop
endif
window landingWindow
Name=dragName
Address=dragAddress
City=dragCity State=dragState Zip=dragZip
```

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the point parameter.

See Also: [point](#)(function
[window](#) statement
[info](#)("windowname") function
[rectangle](#)(function
[rectanglesize](#)(function
[unionrectangle](#)(function
[intersectionrectangle](#)(function
[info](#)("screenrectangle") function
[info](#)("windowrectangle") function
[info](#)("buttonrectangle") function
[info](#)("cursorrectangle") function

FIRSTRECORD

Syntax: FIRSTRECORD

Description: The **firstrecord** statement moves the cursor to the first visible record in the active window. This is the opposite of the [lastrecord](#) statement.

Parameters: This statement has no parameters.

Action: This statement moves the cursor directly to the first visible record in the Data Sheet, Design Sheet, Cross Tab view, or View-as-list Form view. In a Individual Record Form view the view will change to the first record in the database. If the cursor is already on the first visible record this statement will do nothing.

This statement has the same effect as clicking on the **First Record** tool on a tool palette (when available).

Examples: This simple example could be used in either the Data Sheet, Form view or Cross Tab view to move the cursor to the first visible record in the window, making this record the current record.

```
firstrecord
```

This example adds a record to the beginning of the database, finds a record where the field City contains **Huntington Beach**, copies the zip code, returns to the new record, and pastes in the copied zip code as well as updating the City and State fields.

```
firstrecord
insertrecord
find City contains "Huntington Beach"
field "Zip Code"
copycell
firstrecord
pasteccell
City = "Huntington Beach"
State = "CA"
```

This example changes the first and last field's widths to 6.

```
opendesignsheet
lastrecord
Width = 6
firstrecord
Width = 6
newgeneration
closewindow
```

Views: This statement may be used in any view

See Also: [downrecord](#) statement
[lastrecord](#) statement
[uprecord](#) statement

FITWINDOW

Syntax: FITWINDOW

Description: The `fitwindow` statement makes certain the window you are zooming will not exceed your current monitor's screen size.

Parameters: This statement has no parameters

Action: This statement, if needed, must be used immediately before a [setwindow](#) or [zoomwindow](#) statement and will insure that the window in question will not exceed the size of the current screen it is displayed on, whether that screen is a 9" screen, a 20" screen, or larger.

By using a `fitwindow` statement into your procedure you can set the co-ordinates for the [setwindow](#) or [zoomwindow](#) statement to accommodate the largest possible monitor size and the procedure will scale down that size if the computer's screen is smaller.

Examples: This example will scale down the size of the form World Map if you open it on a screen smaller than 680 x 680 pixels.

```
fitwindow
setwindow 20,20,700,700,""
openform "World Map"
```

This example zooms the cross tab window Yearly Sales to a larger size, unless the screen cannot accommodate that larger size.

```
window "Yearly Sales"
fitwindow
zoomwindow 20,20,1200,1200,"nopalette"
```

Views: This statement may be used in Cross Tab views or Form views **only**.

See Also: [getwindow](#) statement
[getmaxwindow](#) statement
[setwindow](#) statement
[window](#) statement
[windowbox](#) statement
[zoomwindow](#) statement

FIX(...)

Syntax: FIX(value)

Description: The `fix()` function truncates a number to an integer. It always truncates towards zero. (If you want to truncate to $-\infty$ use the `int()` function.)

Don't confuse the `fix()` function with the `fixed()` function, which converts floating point numbers to fixed point.

Parameters: This function has one parameter: `value`.

`value` is the value you want to convert to an integer. You may use any numeric value, for example 1, 563.14, -2.5, or even π .

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This simple example calculates the temperature in whole degrees.

```
fix(Temperature)
```

Temperature must contain a numeric value.

The table below shows how the `fix()` and `int()` functions work with some typical values.

Value	fix()	int()
98.700	98	98
4.5640	4	4
-3.14000	-3	-4

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `abs("34")`. If you have a number in a text item you must convert the text to a numeric value before taking the absolute value, for example `abs(val("34"))`.

See Also: [int\(\)](#) function
[fixed\(\)](#) function

FIXED(...)

Syntax: FIXED(value)

Description: The **fixed()** function converts a floating point number to a fixed point number. Since formulas usually perform this conversion automatically when necessary, you'll probably never need this function.

Don't confuse the **fixed()** function with the **fix()** function, which truncates a number to an integer.

Parameters: This function has one parameter: **value**.

value is the value you want to convert to a fixed point number. You may use any numeric value, for example 1, 563.14, -2.5, or even π .

Result: The result of this function is always a fixed point numeric value, using the least number of digits possible.

Examples: This simple example converts the temperature into a fixed point number.

```
fixed(Temperature)
```

Temperature must contain a numeric value.

Warning: The number may lose some precision when it is processed with the **fixed()** function. For example, if the number has more than 4 places after the decimal point, the extra precision will be truncated.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `fixed("34")`. If you have a number in a text item you must convert the text to a numeric value before using it with this function, for example `abs(fixed("34"))`.

Fixed point overflow. This error occurs if you convert a floating point value that is too large to fit into the fixed point number format. If the value is an integer it must not be larger than $\pm 2,100,000,000$. If the value has four (or more) places after the decimal point it must not be larger than $\pm 210,000$.

See Also: [float\(\)](#) function
[int\(\)](#) function
[fix\(\)](#) function

FLOAT(...)

Syntax: `FLOAT(value)`

Description: The `float()` function converts a fixed point number to a floating point number. You can use this function to avoid the overflow, underflow and accuracy problems that can occur when using fixed point arithmetic.

Note: If the destination of a formula is a floating point value (or is text) Panorama performs an implicit `float()` function for every numeric value.

Parameters: This function has one parameter: `value`.

value is the value you want to convert to a floating point number. You may use any numeric value, for example 1, 563.14, -2.5, or even pi (π).

Result: The result of this function is always a floating point numeric value.

Examples: This simple example converts the temperature into a floating point number.

```
float(Temperature)
```

The `float()` function can solve underflow, overflow and accuracy problems when performing calculations with fixed point numbers. Suppose you have three fields: Tax, SubTotal and TaxRate. Each of these fields is set up as 2 digit fixed point numbers. To calculate the tax you use this formula.

```
Tax=SubTotal*(TaxRate/100)
```

This formula will work fine as long as the tax rate is an integer value like 4 or 7. But what if it is a non-integer value, say 8.25? In this case the intermediate calculation `TaxRate/100` will produce a value of .0825. This value will be cut off to .08 to fit in the 2 digit fixed point value, and the calculation will be incorrect. One solution is to convert all values to floating point. This forces Panorama to perform intermediate calculations using floating point, eliminating the accuracy problem.

```
Tax=float(SubTotal)*(float(TaxRate)/float(100))
```

Note: In this particular example, there is another way to solve the problem. By changing the location of the parentheses we can re-arrange the calculation to eliminate the precision problem.

```
Tax=(SubTotal*TaxRate)/100
```

This calculation is slightly faster than the formula using the `float()` function (Panorama performs fixed point math faster than floating point math). However, if you cannot re-arrange your formula to eliminate fixed point math problems, just use the `float()` function.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `float("34")`. If you have a number in a text item you must convert the text to a numeric value before using it with this function, for example `abs(float("34"))`.

See Also: [fixed\(\)](#) function

FLOATINGEDIT

Syntax: `FLOATINGEDIT field,x,y,hi,wide,font,size,just`

Description: The `floatingedit` statement allows you to open an edit window for a database field anywhere over a form window, even if there is no data cell for that field on the form.

Parameters: This statement has eight required parameters: `field`, `x`, `y`, `hi`, `wide`, `font`, `size`, and `just`.

field is the name of the field in the active database you wish to edit. This field can be of any type.

x and **y** are the screen's pixel co-ordinates of the position of the upper left hand corner of the edit window you wish to open. These co-ordinate are internal form co-ordinates which are measured from the top left edge of the form window. **Warning:** if the form window is not visible on the screen unwanted results may ensue with respect to the edit window's position.

hi and **wide** specify the size of the edit window, in pixels, you wish opened for the field.

font is the name of the font your wish the edit window to display the characters in. The font may be any font currently installed in your computer's System. You may view the available System fonts from the **Font** submenu on Panorama's **Text** menu.

size is the point size of the font you wish used inside the edit window. System font point sizes will display in an outline style on the **Size** submenu on Panorama's **Text** menu.

just sets the justification for the edit window based on the three choices below:

0	Left Justification
1	Right Justification
-1	Center Justification

The parameters listed above may be specified as literal values, values in a field or variables, or a formula which results in the proper value for that parameter. You may mix them any way you like. All literal text parameters must be in quotes (" ").

Action: This statement will pause a procedure and allow you to open and control the attributes of a floating edit window for the specified database field anywhere over an active form window. The field you wish to edit must be part of the database associated with the active form window.

Warning: If the active window is **not** a form window the results may be a variety of errors from alert dialogs to unwanted font changes.

Examples: This example opens an edit window for the field `CheckNum`, assigning the edit window a 10 point Helvetica font that is center justified in the edit window.

```
floatingedit "CheckNum",20,20,20,50,"helvetica",10,-1
```

This procedure will loop through all the fields in the database between the field `First` and `Last` and present you with an edit window for each field.

```
local thefieldname field "First"
openform "Edit Fields"
loop
  thefieldname = info("fieldname")
```

```
floatingedit thefieldname,20,20,20,50,"Geneva",10,0
right
until thefieldname = "Last"
left ; so the cursor remains on the field "Last"
```

This procedure, assigned to a button on a form, will determine the co-ordinates of the button and place the edit window over it.

```
local x,y,hi,wide
getlocalbutton x,y,hi,wide
floatingedit "Start Date",x,y,hi,wide,"Geneva",10,0
```

Views: This statement may be used in a Form view only.

See Also: [editcell](#) statement
[editselect](#) statement
[info\("fieldname"\)](#) function

FOLDER(...)

Syntax: FOLDER(folder)

Description: The folder(function creates a binary data item that unambiguously describes the location of a folder on the hard disk. This pathid can be used in other functions and statements.

Parameters: This function has one parameter: folder.

folder is a complete description of the path to this folder, for example [HD:System Folder:Extensions:](#) or [C:\Business\Documents\](#).

Result: This function returns a 6 byte binary data item that unambiguously describes the location of the folder. However, if the folder does not exist the function returns an empty binary data item ("").

Examples: This example checks to see if a folder exists.

```
if ""=folder("HD:Panorama Accounting:Order Entry:")  
    message "You do not have the Order Entry option."  
    stop  
endif
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder parameter.

See Also: [folderpath\(](#) function
[listfiles\(](#) function
[dbinfo\(](#) function
[info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#) function
[openfiledialog](#) statement
[savefiledialog](#) statement
[makefolder](#) statement

FOLDERPATH(...)

Syntax: FOLDERPATH(folder)

Description: The `folderpath()` function takes a path id and converts it to a description of the path to that folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements.

Parameters: This function has one parameter: `folder`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder.

Result: This function returns complete description of the path to this folder, for example `HD:System Folder:Extensions:` or `C:\Personal\Shopping\`.

Examples: This example displays the folder the currently running copy of Panorama is located in.

```
message "This database is in the "+  
folderpath( info("panoramafolder"))+" folder."
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder parameter. This error can also occur if the folder parameter is more or less than 6 bytes long.

See Also: [folder\(\)](#) function
[listfiles\(\)](#) function
[dbinfo\(\)](#) function
[info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#) function
[openfiledialog](#) statement
[savefiledialog](#) statement
[makefolder](#) statement

FONT

- Syntax:** FONT font
- Description:** The **font** statement specifies the font for the current Data Sheet, Design Sheet, or Cross Tab window.
- Parameters:** This statement has one parameter: font
- font** is the name of the font you wish assigned to the active Data Sheet, Design Sheet, or Cross Tab window. This parameter may be a quoted string, field, variable, or formula which results in a name of a font currently loaded into your computer's system. All available system fonts will be listed on Panorama's **Font** submenu (**Text** menu).
- Action:** This statement will immediately set the font for the active window to the one specified provided the window is a Data Sheet, Design Sheet, or Cross Tab window. All other views will ignore the font statement.
- You may also change the point size for the window using the **size** statement or change the style using the **style** statement.
- This statement has the same effect as using the **Font** submenu from the **Text** menu.
- Examples:** This example changes the font for the Data Sheet to Helvetica.
- ```

opensheet
font "Helvetica"

```
- This example opens a cross tab window called Budget, changes the font to palatino, changes the point size to 14, prints the window and then returns to the original form.
- ```

local form
form = info("formname")
gocrosstab "Budget"
calccrosstab
font "palatino"
size 14
print ""
goform form

```
- Views:** This statement may be used in the Data Sheet, Design Sheet, or Cross Tab views only.
- See Also:** [fieldstyle\(\)](#) function
[size](#) statement
[style](#) statement

FORMCOLOR

Syntax: FORMCOLOR color

Description: The `formcolor` statement changes the background color of the current form (see [colors](#)).

Parameters: This statement has one parameter: color.

color is the new background color for the form. A color can be created with the [rgb\(\)](#) or [hsb\(\)](#) functions, or with the [colorwheel](#) statement.

Action: This statement changes the background color of the current form. The default background color is white, but you can change it to any color with this statement. The background color can be changed manually by going into Graphics Mode, selecting **Form Preferences** from the Setup Menu, then choosing the color from the pop-up menu. A procedure can find out what the current background form color is with the [info\("formcolor"\)](#) function.

Examples: The example sets the background color to light gray.

```
formcolor rgb(56000,56000,56000)
```

Views: This statement may be used in a form view.

See Also: [colors](#)
[info\("formcolor"\)](#) function
[rgb\(\)](#) function
[hsb\(\)](#) function
[red\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function

FORMCOMMENT(...)

Syntax: FORMCOMMENT(database,form)

Description: The `formcomment()` function returns the form comment for any form in any open database. The form comment is a description of the form that you can set up using the Form Comment dialog in the Setup menu (graphics mode).

Parameters: This function has two parameters: database and form.

database is the name of the database that contains the form. The database must be currently open. If this parameter is empty text ("") the current database is assumed.

form is the name of the form.

Result: This function returns a textual description of the specified form. This description must be set up in advance with the Form Comment dialog in the Setup menu (graphics mode)

Examples: The example below builds a list of all forms in the current database that have the word "label" in the form comment.

```
local theDatabase,allForms,labelForms
theDatabase="" /* could be set to another db */
allForms=dbinfo("forms",theDatabase)
arrayfilter allForms,labelForms,¶,
    ?(formcomment(theDatabase,import()) contains "label",import(),"")
labelForms = arraystrip(labelForms,¶)
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the database or form parameters.

See Also: [info\("formcomment"\)](#) function

FORMCOMMENTS

Syntax: FORMCOMMENTS

Description: The **Formcomments** statement allows you to pause a procedure to open and edit the Form Comments dialog.

Parameters: This statement has no parameters.

Action: This statement is used to open the Form Comments dialog so that you may edit the comments associated with the active form window. If the active window is not a form window this statement will result in an alert dialog warning you that you cannot do that in this window.

The Form Comments dialog has three sub-divisions:

Primary Purpose of Form... which allows you to set an arbitrary "class" for that form. This setting does not effect how Panorama deals with the form, it is simply used by the designer as a means to categorize what the form is used for.

Notes... allows you to assign a description of the form's purpose. You may also use this section to give the form's dimensions, orientation, or number of pages it prints, etc.

Preview... allows you to paste in a PICT image of what the form will look like. This may be useful say, if someone is looking for a specific print format before choosing a print form.

This statement has the same effect as choosing the **Form Comments...** command from the **Setup** menu (Graphics Design mode of any form window.)

Examples: Running this example from a form window will open the Form Comments dialog for the form and allow you to make changes to it.

`formcomments`

This example takes you to a form called date entry and allows you to edit the form comments for it.

```
goform "data entry"  
formcomments
```

Views: This statement may be used in a Form view **only**.

See Also: [formcolor](#) statement
[formselect](#) statement
[formcomment\(\)](#) function
[formtype\(\)](#) function
[goform](#) statement
[info\("formcolor"\)](#) function
[info\("formname"\)](#) function
[info\("typeofwindow"\)](#) function
[info\("windowname"\)](#) function
[openform](#) statement
[window](#) statement

FORMCOMMENTTYPE

Syntax: FORMCOMMENTTYPE type,subtype

Description: The `formcommenttype` statement allows a procedure to change the “type” of a form. The form “type” is a pair of numbers that can be used to identify the purpose of the form (display, printing, dialog, etc.). Panorama does not enforce the form type, so you can designate a form as a report and then use it as a dialog if you wish. However, you can use the form type to help you keep track of what each form is used for. See the [formselect](#) statement.

Parameters: This statement has two parameters: `type` and `subtype`.

type is a numeric integer from 0 to 255. New forms default to zero. 1 is reserved for data entry forms, 2 for reports, and 3 for dialogs. You may use 4 through 255 for your own special designations.

subtype is a numeric integer from 0 to 255. If the main type is 2, the subtype is suggested to be either 0 (full page form) or 1 (multiple line report). Otherwise you can use this value any way you like.

Action: This statement changes the “form type,” which is usually done manually with the **Form Comments** dialog. If the active window is **not** a form window this statement will result in an alert dialog warning you that you cannot do that in this window.

Examples: This example creates a new form and designates it as a report.

```
newform
formcommenttype 2,1
```

Views: This statement may be used in a Form view **only**.

See Also: [formcolor](#) statement
[formselect](#) statement
[formcomment\(\)](#) function
[formtype\(\)](#) function
[goform](#) statement
[info\("formcolor"\)](#) function
[info\("formname"\)](#) function
[info\("typeofwindow"\)](#) function
[info\("windowname"\)](#) function
[openform](#) statement
[window](#) statement

FORMSELECT

Syntax: FORMSELECT dialog#,filter,button,form

Description: The `formselect` statement pauses a procedure and displays a dialog through which the user can choose a form from the active database. The dialog may also show the Form Comments information (see [formcomments](#).)

Parameters: This statement has four required parameters: `dialog#`, `filter`, `button`, and `form`.

dialog# is the resource number that identifies the dialog you wish to display. If you do not wish to create your own dialog, with ResEdit for example, you may use Panorama's built in dialog# **2086**. This parameter may be a literal value, a field or variable containing a numeric value or a formula which results in the proper numeric value.

filter is a numeric value used to determine which type of forms will be displayed in the dialog. `filter` can be set for each form using the [formcomments](#), or [formcommenttype](#) statements or by using the **Form Comments...** command from the **Setup** menu (Graphics Design mode of a form window.) This value is set in the Primary Purpose of Form... section of the Form Comments dialog. This parameter may be a literal value, a field or variable containing a numeric value or a formula which results in the proper numeric value. The following list shows the possible `filter` values.

0	All Forms
1	Data entry forms
2	Printing forms
3	Dialog & related forms
4 or greater	Custom Forms

button is the name of a variable that will contain the name of the button that was pressed inside the Form Select dialog. Clicking on any button in the dialog closes the dialog and allows the procedure to continue.

form is the name of a variable that will contain the name of the form selected in the dialog. If the variable is pre-set to the form name before the `formselect` statement is reached this form will be selected when the dialog opens. If no form is select this variable will equal "".

Action: This statement will pause a procedure while opening a Form Selection dialog allowing the user to select a form and choose an operation from a series of buttons in the dialog before continuing on with the procedure.

Examples: This example opens the built-in Panorama Form Selection dialog, displaying all forms. It will store the button selection and form selection in the global variables defined.

```
global buttonname,formname
formselect 2086,0,buttonname,formname
```

This procedure opens a custom Form Selection dialog (# **3000**) displaying Print forms only and pre-selects the form called Sheet. The procedure makes a decision based on one of three buttons pressed: **Cancel**, **Print.**, or **Edit**.

```

local PrintButton, PrintForm PrintForm = "Sheet"
openresource "Dialogs"
formselect 3000,2,PrintButton,PrintForm
if PrintButton = "Cancel"
    stop
endif
if PrintButton = "Print"
    openform PrintForm
    print dialog
    closewindow
endif
if PrintButton = "Edit"
    openform PrintForm
    graphicsmode
    stop
endif
endif

```

This procedure asks the user to choose a form type, opens the standard Panorama Form Select dialog, allowing them to select a form, and then hands off to another procedure, called FormHandeling, to process the form.

```

local FormType
global TheButton,TheForm
gettext "Enter Form Type 0-4",FormType
formselect 2086,FormType,PrintButton,PrintForm
call FormHandeling,TheButton,TheForm

```

Views: This statement may be used in the Data Sheet of Forms view **only**.

See Also: [formcomments](#) statement
[goform](#) statement
[info\("formname"\)](#) function
[openform](#) statement

FORMSERVERLOOKUP

- Syntax:** `FORMSERVERLOOKUP status`
- Description:** The `formserverlookup` statement allows a procedure to turn the User Server for Lookup option on or off. (This option can also be turned on or off in the Form Preferences Dialog.)
- Parameters:** This statement has one parameter: `status`.
- status** should be either 0 or 1. If the value is 0 then lookups in this form will come from the local database. If the value is 1 then lookups will be made directly from the SQL Server database. Note: In addition to 0 and 1, you may also use "off" or "on", "no" or "yes", or "false" or "true".
- Action:** This statement allows the database designer to trade off speed vs. up-to-the-minute accuracy in lookups on a form. This option only affects Partner/Server databases that are linked to an SQL server database. For up-to-the minute accuracy lookups should be made directly from the server. However lookups from the server are substantially slower than lookups from the local database. This option does not affect lookups made in procedures, only lookups in auto-wrap text objects and Text Display SuperObjects.
- Examples:** The example forces lookups in the form Patient Status to be made from the local database.
- ```
openform "Patient Status"
formserverlookup 0
```
- Views:** This statement may be used in Form views.
- See Also:** [lookup\(\)](#) function

# FORMTYPE(...)

**Syntax:** FORMTYPE(database,form)

**Description:** The `formtype()` function returns the form type (a number) for any form in any open database. The form type is a number that you can set up using the **Form Comment** dialog in the Setup menu (graphics mode).

**Parameters:** This function has two parameters: `database` and `form`.

**database** is the name of the database that contains the form. The database must be currently open. If this parameter is empty text ("") the current database is assumed.

**form** is the name of the form.

**Result:** This function returns a number (integer) from 0 to 255. The value of this number depends on the Primary Purpose of Form area of the Form Comment dialog (in the graphics mode Setup Menu.) There are predefined radio buttons for 1) Data Entry, 2) Printing, and 3) Dialog and other. Or you may enter any value from 0 to 255 in the Custom area.

**Examples:** The example below builds a list of all forms in the current database that are designated for printing (type = 2).

```
local theDatabase,allForms,labelForms
theDatabase="" /* could be set to another db */
allForms=dbinfo("forms",theDatabase)
arrayfilter allForms,labelForms,␣,
 ?(formtype(theDatabase,import())=2,import(),"")
labelForms = arraystrip(labelForms,␣)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the database or form parameters.

**See Also:** [formcomments](#) function  
[info\("formcomment"\)](#) function

# FORMULABUFFER

**Syntax:** FORMULABUFFER size

**Description:** The **formulabuffer** statement allows you to increase the size of the buffer Panorama uses for evaluating formulas. The default value is 2000 bytes. If your formulas are too complex you will get the error message **Expression too complicated**. This problem can be cured with the **formulabuffer** statement.

**Parameters:** This statement has one parameter: size.

**size** is the new size of the formula buffer. If you specify a size of zero, Panorama will use the default 2000 byte buffer, freeing any extra memory you have allocated for a larger buffer.

**Action:** This statement forces Panorama to expand the formula evaluation buffer. The expanded formula buffer is not created until the procedure is run. That means that if the complex formula is in the same procedure as the **formulabuffer** statement, you won't be able to compile the procedure because the buffer hasn't been expanded yet. The best way to eliminate this problem is to put the formulabuffer statement into your **.Initialize** procedure.

The **formulabuffer** statement is semi-permanent: it applies to all formulas in all databases until you quit Panorama or change the setting again. If you want to cancel expanded buffer and go back to the internal buffer, use formulabuffer statement with a size of 0.

How large should you make the formula buffer? Most users have never encountered the **Expression too complicated** error message and have no need to expand the buffer. If you do encounter this error, you should probably start by modestly expanding the buffer, perhaps to 3000 to 4000 bytes. If you still have a problem you can expand it further until the problem disappears. However, if your database allows users to enter formulas of any length, you may wish to expand the buffer in advance to a very large size, perhaps 32000 bytes.

**Examples:** This example expands the formula buffer to a very large size. Keep in mind that this buffer is taken from your scratch memory space, which you may want to adjust accordingly.

```
formulabuffer 3200
```

**Views:** This statement may be used in any view

**See Also:** none



# FORMULACALC

- Syntax:** FORMULACALC result,formula
- Description:** The `formulacalc` statement allows you to evaluate a formula that you are not able to code into the procedure when it was being written.
- Parameters:** This statement has two parameters: `result` and `formula`.  
**result** must be the name of a field or a variable where you wish the result of the formula to be placed. The formula's resulting value must match the type and format (if any) of `result` or an error alert dialog will be displayed.  
**formula** can be a literal value, a field or a variable, or an expression that returns the value you wish to put into `result`. The resulting value must match the type and format (if any) of `result` or an error alert dialog will be displayed.
- Action:** This statement evaluates `formula` and, if it is compatible with `result`'s type, writes the answer to `result`.  
**Warning:** If `result` already has a value in it `formulacalc` will replace that value with the new one.
- Examples:** This example evaluates the formula in the field `Expression` and writes the answer to field `A`.

```
formulacalc «A»,«Expression»
```

This example allows you to enter the formula you wish evaluated into the clipboard and then placed the answer in a global variable called `Answer`.

```
global Answer
getscrap "Enter formula: "
formulacalc Answer,clipboard()
```

This example is basically the same but stored the formula into a local variable called `formula` and writes the answer to a global variable called `Result` which you could later display on a form.

```
global Result
local formula
gettext "Enter formula: ",formula
formulacalc Result,formula
```

This example increases the width for the third field in the database to the amount determined by the formula entered.

```
opendesignsheet
local expression
downrecord
downrecord
gettext "Enter formula: ",expression
formulacalc Width,expression
newgeneration
closewindow
```

**Views:** This statement may be used in any view

**See Also:** [emptyfill](#) statement  
[fill](#) statement  
[formulafill](#) statement  
[formulasum](#) statement  
[scrapcalc](#) statement  
[set](#) statement

# FORMULAFILL

**Syntax:** FORMULAFILL formula

**Description:** The **formulafill** statement fills every visible cell in the active field with the result of the specified formula.

**Parameters:** This statement has one parameter: formula

**formula** can be a literal value, a field or a variable, or an expression that returns the value you wish to put into the cells of the current field. The resulting value must match the type and format (if any) of the field you are placing it in or an error will result. If formula is the word **dialog** Panorama will present you with an input dialog so you may enter your own formula.

**Action:** This statement evaluates formula and if it is compatible with the current field type writes it's result to every cell in that field for all selected records only.

**Warning:** If a cell already has a value in it formulafill will replace that value with the new one.

When formula is the word **dialog** Panorama will pause the procedure and present the user with the **Formula Fill** dialog allowing the user to enter their own formula. Clicking on the **Ok** button will allow the procedure to continue. This statement will work for all field types except **picture** type fields.

This statement has the same effect as using the **Formula Fill** command from the **Math** menu.

**Examples:** This simple example tells Panorama to clear all the selected cells of the active field. Therefore, the active field should be a text or choice type field.

```
formulafill ""
```

This similar example works for numeric or date type fields.

```
formulafill zeroblank(0)
```

This example tells Panorama to look at all the selected cells of the field Options and only fill the empty ones with the text string n/a.

```
field Options
formulafill ?(Options = "", "n/a", Options)
```

This example alters the area code for the selected records in the numeric field Phone Number from 213 to 310.

```
field "Phone Number"
select strip(«Phone Number») beginswith "213"
formulafill val("310"+str(«Phone Number»)[4,-1])
```

This example will have Panorama open the Formula Fill... dialog which allows you to enter a formula, the result of which will be filled into the visible cells in the current field.

```
formulafill dialog
```

This example uses a formula to fill the visible cells in the field Date to Renew with a date that is one year beyond the computer's current date. The final date will be the 1st of the month.

```
field «Date to Renew»
formulafill datepattern(month1st(today() +
365), "mm/dd/yy")
```

This example increases all the field widths by 10%.

```
opendesignsheet
field Width
formulafill Width * 1.10
newgeneration
closewindow
```

**Views:** This statement may be used in any view.

**See Also:** [emptyfill](#) statement  
[fill](#) statement

# FORMULAFINDSELECT

**Syntax:** FORMULAFINDSELECT

**Description:** The `formulafindselect` statement pauses a procedure while opening the Formula Find/Select dialog allowing the user to manually enter their search formula and perform one of the find or select options.

**Parameters:** This statement has no parameters.

**Action:** This statement pause the procedure while displaying the standard Formula Find/Select dialog. The user can then enter their search formula and choose one of the four search options: **Find**, **Select**, **Select Additional**, or **Select Within** or they can hit the **Cancel** button. Hitting the Cancel button will cancel the search operation, but it will not terminate the procedure. The procedure will continue running after the `formulafindselect` statement.

Compound searches may be performed by writing your search formula correctly.

This statement **cannot** be used in combination with the `info("found")` or `info("empty")` functions as the parameter of an if statement immediately after the `formulafindselect` statement to test for a null `find` or `select`.

All search options will work on all fields except **Picture** type fields.

This statement has the same effect as choosing the **Formula Find/Select** command from the Search menu.

**Examples:** This example opens the **Formula Find/Select** dialog.

```
formulafindselect
```

This example allows you to use the Formula Find/Select dialog and then it sorts the selected records and brings you to the first one.

```
message "Use the upcoming dialog to select the records you wish to work
with."
formulafindselect
field «Company Name»
sortup
firstrecord
```

**Views:** This statement may be used in the Data Sheet, Form views and Cross Tab views only.

**See Also:** [find](#) statement  
[findselect](#) statement  
[info\("empty"\)](#) function  
[info\("found"\)](#) function  
[info\("records"\)](#) function  
[info\("selected"\)](#) function  
[next](#) statement  
[select](#) statement  
[selectadditional](#) statement

selectall statement  
selectreverse statement  
selectsummaries statement  
selectwithin statement

# FORMULASUM

**Syntax:** `FORMULASUM result,formula`

**Description:** The `formulasum` statement allows you to evaluate a formula over and over again for each selected record and return the accumulated sum to result.

**Parameters:** This statement has two parameters: `result` and `formula`.

**result** must be the name of a field or a variable where you wish the result of the summed formula to be placed. The resulting summed value must match the type and format (if any) of `result` or an error alert dialog will be displayed. If `result` is a variable you can pre-set the type by assigning it an arbitrary value of the same type prior to using the `formulasum` statement. If `result` is a field Panorama will write the sum to this field for the active record only.

**formula** can be a literal value, a field or a variable, or an expression that returns the value you wish to put into `result`. The formula's resulting value must match the type and format (if any) of `result` or an error alert dialog will be displayed.

**Action:** This statement evaluates `formula` for each selected record accumulating a total on a record by record basis and, if it is compatible with `result`'s type, writes the sum to `result`.

**Warning:** If `result` already has a value in it `formulasum` will replace that value with the new one.

**Examples:** This example sums the numeric field `Amount` and writes the answer to `Total`.

```
formulasum Total,Amount
```

This example counts the records where `Amount` is over a certain `Threshold` value and writes the answer to the global variable `Over`.

```
local Threshold
global Over
Over = 0
gettext "Enter Threshold value: ",Threshold
formulasum Over,?(Amount > val(Threshold),1,0)
```

This example allows you to enter the formula you wish evaluated over and over again for each record into the clipboard. The accumulated sum for this formula is then placed in the global variable called `Answer`.

```
global Answer
Answer = 0 ; This initializes the variable as numeric
getscrap "Enter formula: "
formulasum Answer,clipboard()
```

This example is basically the same, but stored the formula into a local variable called `formula` and writes the answer to a global variable called `Result` which you could later display on a form.

```
global Result
local formula
gettext "Enter formula: ",formula
formulasum Result,formula
```

**Views:** This statement may be used in any view.

**See Also:** [emptyfill](#) statement  
[fill](#) statement  
[formulacalc](#) statement  
[formulafill](#) statement  
[scrapcalc](#) statement



# FORMXY

**Syntax:** FORMXY vertical,horizontal

**Description:** The `formxy` statement allows you to scroll a form in any direction to a new position within the form's window on screen.

**Parameters:** This statement has two required parameters: `vertical` and `horizontal`.

`vertical` and `horizontal` are the co-ordinates (in pixels) you wish to move the respective scroll bars to reposition the form in it's window (72 pixels = 1 inch). This statement will basically change the upper left hand corner of the form in the screen's window. These co-ordinates are measured from the upper left most portion of the form. These parameters can be literal values, fields, variables, or formulas which result in a positive integer number. Note: Both parameters will be rounded to the nearest multiple of 8 to insure that patterns are displayed correctly.

**Action:** This statement allows you to perform the same task as manually adjusting the vertical or horizontal scroll bars of any active form window; even if the scroll bars are not available for that form's window.

**Note:** Using this statement in a non-form window will result in an error alert dialog.

**Examples:** This example opens a form called List View 1 and scrolls the window down four inches from the top of the form.

```
openform "List View 1"
formxy 4*72,0
```

This example first scrolls the form Enter Data down one inch to edit the field Job Number, then scroll down another inch to edit the field Customer and finally scrolls over three inches to edit the field Description.

```
openform "Enter Data"
formxy 1*72,0
field «Job Number»
editcell
formxy 2*72,0
field «Customer»
editcell
formxy 2*72,3*72
field «Description»
editcell
```

This example allows you to select a city name and scroll the US map form, open on screen, to the form co-ordinates for that city stored in the fields X and Y.

```
getscrap "Enter city name"
find City contains clipboard() ;find the city's record
formxy X,Y
```

**Views:** This statement may be used in a Form view **only**.

**See Also:**

[magnification](#) statement  
[goform](#) statement  
[info\("formname"\)](#) function  
[info\("typeofwindow"\)](#) function  
[info\("windowname"\)](#) function  
[openform](#) statement  
[window](#) statement

# FUNCTIONS

**Introduction:** Functions are used within formulas to calculate or retrieve information. Most functions have one or more parameters which are used as input to the calculation.

Functions are easy to spot because they always are followed by parenthesis: `sin(angle)`, `int(number)`, `today()`, `replace(Company,"Inc","Incorporated")`. If the function has any parameters, they must appear within the parenthesis. You must supply the exact parameters required by the function—no more and no less, and they must be in the correct order.

Functions may be used in any formula—in a procedure, in an auto-wrap text object, and in many SuperObjects.

**See Also:** [statements](#)

# FV(...)

**Syntax:** `FV(rate,periods,payment,pv,begin)`

**Description:** The `fv` function (short for future value) calculates the future value of an investment. For example, if you invest \$500 a month at 10% annual interest, how much money will you have at the end of ten years? This function will calculate that value for you.

**Parameters:** This function has five parameters: `rate`, `periods`, `payment`, `pv` and `begin`.

**rate** is the interest rate of the investment (per period). For example, if there is one payment per year, this is the annual percentage rate. If there is one payment per month, then this is the monthly percentage rate.

**periods** is the number of payments that will be made during the life of the investment. For example, if this is a 36 month investment with one payment per month, this value is 36. If this is a 30 year investment with one payment per month this value is 1080.

**payment** is the amount being invested each period. This should actually be the negative of the amount invested. For example, if you invest \$500 per month, the payment amount should be -500.

**pv** (present value) is the initial value of the investment at the start of the period (actually the negative of the initial value). For example if your savings account starts with \$2,000, this value should be -2000.

**begin** specifies whether payments are made at the beginning (1) or end of each period (0).

**Result:** The result of this function is always a numeric floating point value.

**Examples:** If the payment period is annually the calculation is simple. Suppose you start with \$3,000 and invest it in a savings account at 7% annual interest. Then you take \$5,000 at the end of each year for the next ten years and invest that also. This formula will calculate the final balance in your account at the end of 10 years.

```
fv(0.07,10,-5000,-3000,0)
```

Most investments are paid more frequently than once a year...usually once a month. To calculate the future value for such an investment you must convert the annual percentage rate into a monthly percentage rate by dividing by 12. Suppose you invest \$200 a month at the beginning of each month for 5 years into an investment that returns 13.5% annual interest. The final value of this investment after 10 years can be calculated with this formula.

```
fv(0.135/12,5*12,-200,0,1)
```

You can also use this function to calculate the appreciation of a fixed up-front investment, with no payments. Suppose you invest \$50,000 for 5 years at 14% annual interest. This formula will calculate the value of this money after 5 years:

```
fv(0.14,5,0,-50000,0)
```

Of course the `fv` function only works with fixed interest rates.

**Notes:** Here is the formula that Panorama uses to calculate future value.

$$\text{future value} = - \frac{\text{payment} * ((1 + \text{rate} * \text{begin}) * (1 + \text{rate})^{\text{periods}} - 1)}{\text{rate}} + \text{pv} * (1 + \text{rate})$$

**Errors:** Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value with this function, for example `fv("12%",...)`. If you have a number in a text item you must convert the text to a numeric value before calculating the future value, for example `fv(val("12%"[1,-2]),...)`.

**See Also:** [pmt\(\)](#) function  
[pv\(\)](#) function

# GENERATEVALUES

**Syntax:** GENERATEVALUES threshold

**Description:** The `generatevalues` statement automatically generates a list of choices for a field. The list appears in the Choices attribute in the design sheet.

**Parameters:** There is one parameter: `threshold`

**threshold** is the number of times a field's content must appear in other records before that content is added to a choices list.

**Action:** The `generatevalues` statement scans the database and builds a list of the values in the field. This list is placed into the Choices column in the design sheet.

In order to understand the purpose of this statement, you need to understand Panorama's various "Choices" options. First consider the Choices attribute. After a list of elements is entered in a field's Choices attribute, data editing in that field (back in the datasheet or form view) is changed from "anything goes" to a radio button choice of only the elements entered in that field's designsheet Choices attribute.

Creating a Choices list is also the first step in creating a popup button choice for the field.

Also, once a database contains thousands of records it might be necessary to reduce it's size. This can be done by creating a Choices attribute of the most common content of a field, then changing the field Type to Choices. Panorama would then build its own dictionary of the names in the Choices list, and substitutes shorter codes in all the records. This is mostly a benefit if the field's content is large and mostly repeating.

**Note** that it is not necessary to create a Choices Type, it's only an option after the choices list has been created. Though you don't see the dictionary substitution, it still adds a little bit of overhead. Choices Type is only used when file size becomes an issue.

Also, when you sort a field that has Choices attributes, it sorts according to the Order of the elements in the Choices list. Any records in the "Other" category are sorted after.

So how would you decide what content to put in your Choices attribute? Panorama automatically does that for you if you select **Automatic Choices** under the Special menu in the Design Sheet, and `generatevalues` will do that for you in a procedure.

If all the records in the database match your threshold criteria, just a list of the fields contents is created. If there are some records whose content did not make it to the list because that content didn't appear enough to meet the threshold criteria, an "Other" element is added to the end of the list. The width of the "Other" element is equal to the length of the largest content of that field (from those who didn't make it to the list) in the database.

**Examples:** If you had six records in the database and your field «mycity» contained the following data and you ran the sample procedure shown below

```
Portland
Huntington Beach
Portland
Seal Beach
Huntington Beach
Huntington Beach
```

```
opendesignsheet
field «mycity»
generatevalues 2
newgeneration
closewindow
```

The Choices attribute for «myfield» would contain

```
Portland Huntington_Beach _____
```

Note the \_\_\_\_\_ "Other" is added because at least one record, **Seal Beach** did not satisfy the criteria of occurring two or more times.

**Views:** This statement may be used only in the design sheet window.

**See Also:** [opendesignsheet](#) statement  
[field](#) statement  
[newgeneration](#) statement  
[closewindow](#) statement

# GESTALT(...)

GESTALT

**Syntax:** GESTALT(option)

**Description:** The `gestalt()` function makes a wide variety of system configuration information available to the Panorama programmer. Since this function is based on the MacOS Gestalt system function most of the information options are only available on MacOS based systems, but some information is available even on Windows PC systems.

**Parameters:** This function has one parameter: `option`.

**option** is a four letter code that specifies what information you want to access. There are literally hundreds of valid codes available on MacOS based systems, and new codes are added all the time (by third parties as well as by Apple). Only a few of these options will be discussed here, for a more information you will need to consult Apple programming documentation as well as programming documentation provided by third party vendors.

**Result:** The result of this function depends on the option selected. In most cases it returns a number but some options return a text value.

**Examples:** This example displays the current system version (for example 8.6.1 or 9.0.4).

```
message radixstr(16,gestalt("sysv"))[6;1]+". "+
 radixstr(16,gestalt("sysv"))[7;1]+". "+
 radixstr(16,gestalt("sysv"))[8;1]
```

This example displays the amount of physical RAM memory installed in the machine.

```
message pattern(gestalt("ram "),"#, bytes")
```

This example displays the amount of logical RAM memory (including virtual memory) available in the machine.

```
message pattern(gestalt("lram "),"#, bytes")
```

This example displays whether or not AppleScript is available on this machine.

```
message "AppleScript: "+?(1 and gestalt("ascr"),"YES","NO")
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the option parameter.

**See Also:** [radixstr\(\)](#) function  
[pattern\(\)](#) function  
[info\("panoramafolder"\)](#) function  
[info\("systemfolder"\)](#) function



# GETADDRESS

- Syntax:** `GETADDRESS raw,streetaddress,city,state,zip,country`
- Description:** The `getaddress` statement divides an address into its individual components: street address, city, state, zip code and country.
- Parameters:** This statement has six parameters: `raw`, `streetaddress`, `city`, `state`, `zip`, and `country`
- `raw` may be in a field or variable. The `raw` address must be either 2 or 3 lines high. It contains the street address in one or two lines and the city, state, and zip and country information in the last line.
- `streetaddress` may be in a field or variable. It holds the street address returned by `getaddress`.
- `city` may be in a field or variable. It holds the city returned by `getaddress`.
- `state` may be in a field or variable. It holds the state returned by `getaddress`.
- `zip` may be in a field or variable. It holds the zip code returned by `getaddress`.
- `country` may be in a field or variable. It holds country returned by `getaddress`.
- If the city and state are left off the `raw` address, `getaddress` will look them up using the zip code (if it is a US address and the optional Zip Magic file is installed).
- Action:** The `getaddress` statement investigates the content of the `raw` parameter starting from the end (last character). That is, if the last field in the address is not a number, it's considered the country and capitalized.
- If the next field (or last field) is a number, it is considered the zip code. If the zip code has letters in it, it is considered a Canadian zip code and CANADA is assigned to the `country` parameter (note capitalization). If the zip code is the last element on that line, and it's a USA zip code, the `city` and `state` are looked up (if you have the Zip Magic file) with the two letter state abbreviation capitalized and the first letter of the state capitalized. Alert, the city must be separated from the state by a comma in order for the parser to properly identify the fields.
- The content of the field after the first carriage return - remember, we are looking from the end of the field to the front - is considered the `streetaddress`. If another carriage return is found, the content prior to it is the first line of the `streetaddress`.
- Only first letter of the names in the address are capitalized. So 1015 s river road would become 1015 S. River Road. But if there is a two letter direction like nw (north west), you'd have to manually capitalize it at entry time, like 1015 NW river road
- Examples:** Given an address in the following format:
- ```
Student Building
1234 West Street
Huntington Beach, CA 92648 USA
```
- in a field called `«myaddress»`
- ```
local Madr, Mcty, Mste, Mzip, Mcntry
getaddress «myaddress»,Madr,Mcty,Mste,Mzip,Mcntry
```
- would return

Student Building  
1234 West Street in Madr  
Huntington Beach in Mcty  
CA in Mste  
92648 in Mzip  
USA in Mcntry

Because this function cannot anticipate every variation of address format, it best to try it on your anticipated data. For example, if you also had a Name above Student Building in the «myaddress» field, the **getaddress** function would incorrectly parse the rest of the **address**.

**Views:** This statement may be used in any view

**See Also:** [getcitystatezip](#) statement

# GETAUTONUMBER

**Syntax:** GETAUTONUMBER *variable*

**Description:** The `getautonumber` statement finds out the automatically generated number for the next record that will be added to database.

**Parameters:** This statement has one parameter: *variable*.

*variable* will be filled with the automatically generated number for the next record. **Note:** If this is an SQL Partner/Server database the number will change if other users add records to the database.

**Action:** Panorama can automatically number new records as they are added to the database. This statement allows a procedure to find out what the next number will be. The next number can be changed with the [setautonumber](#) statement.

**Examples:** The example skips autonumbering ahead by 100.

```
local xNumber
getautonumber xNumber
setautonumber xNumber+100
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [setautonumber](#) statement  
[addrecord](#) statement  
[insertrecord](#) statement

# GETBUTTON

- Syntax:** `GETBUTTON top,left,height,width`
- Description:** The `getbutton` statement retrieves the global screen co-ordinates of the last button that was pressed. (Global screen co-ordinates are the co-ordinates from the top left hand corner of the menu bar. The co-ordinates are measured in pixels.)
- Parameters:** This statement has four parameters: `top`, `left`, `height` and `width`.  
The four parameters should be variables. **Note:** Panorama 3 has a function that returns the same information: [info\("buttonrectangle"\)](#).  
**top** is a field or variable. The statement will fill this variable with the distance from the top of the screen to the top of the button.  
**left** is a field or variable. The statement will fill this variable with the distance from the left side of the screen to the left side of the button.
- Action:** **height** is a field or variable. The statement will fill this variable the height of the button.  
**width.** is a field or variable. The statement will fill this variable the width of the button.
- Examples:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field. It is used in a procedure that can be triggered by a button push.
- ```

local btop, bleft, bheight, bwidth
getbutton btop, bleft, bheight, bwidth

```
- Views:** This statement may be used in a Form view
- See Also:** [getlocalbutton](#) statement
[getinternalbutton](#) function
[info\("trigger"\)](#) statement
[popup](#) statement
[setwindow](#) statement
[floatingedit](#) statement
[info\("buttonrectangle"\)](#)

GETCITYSTATEZIP

- Syntax:** `GETCITYSTATEZIP raw,city,state,zip,country`
- Description:** The `getcystatezip` statement divides the last line of an address into separate components.
- Parameters:** This statement has five parameters: `raw`, `city`, `state`, `zip` and `country`
- `raw` is a text item containing the combined **city**, **state**, and **zip** and **country** information in one line.
- `city` may be in a field or variable. It holds the **city** returned by `getcystatezip`.
- `state` may be in a field or variable. It holds the **state** returned by `getcystatezip`.
- `zip` may be in a field or variable. It holds the **zip code** returned by `getcystatezip`.
- `country` may be in a field or variable. It holds **country** returned by `getcystatezip`.
- If the **city** and **state** are left off the `raw` address, `getcystatezip` will look them up using the **zip code** (if it is a US address and the Zip Magic file is installed).
- Action:** The `getcystatezip` statement investigates the content of the `raw` parameter starting from the end (last character). If the last field in the address is not a number, it's considered the **country** and capitalized.
- If the next field (or last field) is a number, it is considered the **zip code**. If the **zip code** has letters in it, it is considered a Canadian **zip code** and CANADA is assigned to the `country` parameter (note capitalization).
- If the **zip code** is the last element on that line, and it's a USA **zip code**, the **city** and **state** are looked up (if you have the Zip Magic file) with the two letter state abbreviation capitalized and the first letter of the **state** capitalized. **Alert**, the **city** must be separated from the **state** by a comma in order for the parser to properly identify the fields.
- Only first letter of the names in the **address** are capitalized. So 1015 s river road would become 1015 S. River Road. But if there is a two letter direction like nw (north west), you'd have to manually capitalize it at entry time, like 1015 NW river road
- Examples:** Given an address in the following format:
- Huntington Beach, CA 92648 USA**
- in a field called `«mycsz»`
- ```
local Mcty, Mste, Mzip, Mcntry
getcystatezip «mycsz»,Mcty,Mste,Mzip,Mcntry
```
- would return
- Huntington Beach in `Mcty`  
 CA in `Mste`  
 92648 in `Mzip`  
 USA in `Mcntry`
- Because this function cannot anticipate every variation of address format, it best to try it on your anticipated data.

**Views:** This statement may be used in a procedure.

**See Also:** [getaddress](#) statement

# GETCLICK

**Syntax:** GETCLICK vertical,horizontal

**Description:** The `getclick` statement gets the location of the mouse in global coordinates (co-ordinates from the top left corner of the menu bar). Note: Panorama 3 has a function that returns the same information: [info\("click"\)](#)

**Parameters:** There are two parameters: `vertical` and `horizontal`  
**vertical** holds the pixel distance from the top of the screen.  
**horizontal** holds the pixel distance from the left edge of the screen.

**Action:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field.

**Examples:** This example displays the distance of the click from the upper left hand corner of the screen (for example 4.5 inches).

```

local mouseV,mouseH
getclick mouseV,mouseH
message str(
 sqr(mouseV*mouseV+mouseH*mouseH)/72)+" inches "

```

**Views:** This statement may be used in Form views.

**See Also:** [getlocalclick](#) statement  
[info\("trigger"\)](#) function  
[popup](#) statement  
[setwindow](#) statement  
[floatingedit](#) statement  
[info\("click"\)](#) function  
[xytoxy\(\)](#) function

# GETCURSOR

**Syntax:** GETCURSOR top,left,height,width

**Description:** The `getcursion` statement retrieves the global screen co-ordinates of the current active field. (Global screen co-ordinates are the co-ordinates from the top left hand corner of the menu bar. The co-ordinates are measured in pixels.)

**Parameters:** This statement has four parameters: `top`, `left`, `height` and `width`.

The four parameters should be variables. Note: Panorama 3 has a function that returns the same information: [info\("cursorrectangle"\)](#).

**top** holds the pixel distance from the top of the screen.

**left** holds the pixel distance from the left edge of the screen.

**height** holds the pixel height of the button.

**width.** holds the pixel width of the button.

**Action:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field.

**Examples:** This example opens the form window Detail over the currently active field.

```
local btop, bleft, bheight, bwidth
getcursion btop,bleft,bheight,bwidth
setwindow statement
floatingedit btop,bleft,100,250,""
openform "Detail"
```

**Views:** This statement may be used in a Form view.

**See Also:** [popup](#) statement  
[setwindow](#) statement  
[floatingedit](#) statement  
[info\("cursorrectangle"\)](#)



# GETFILEFINDERINFO

- Syntax:** `GETFILEFINDERINFO folder,filename,typecreator,position, flags, creationdate, modificationdate`
- Description:** The `getfilefinderinfo` statement retrieves a collection of information about a file, including when it was created and last modified and its position within the window.
- Parameters:** This statement has seven parameters: `folder`, `filename`, `typecreator`, `position`, `flags`, `creationdate`, and `modificationdate`.
- folder** is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()` and `dbinfo()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.
- filename** is the name of the file you wish to save. The file name may be up to 31 characters long, and may not contain / characters.
- typecreator** may be a field or variable. Panorama will place the 4 character type code and 4 character creator code into this field or variable, for example `ZEPDKASX` for a Panorama database file.
- position** is the visual x-y position of this file within the folder (see [graphic coordinates](#)). If you don't want to change the position use 0.
- flags** is a number that specifies operating system specific options for this file. If bit 14 of this value is set then the file is invisible. If you don't want to change the flags use 0.
- creationdate** contains the time and date the file was created, in SuperDate format (see `superdate()`). If you don't want to change this date use 0.
- modificationdate** contains the time and date the file was last modified, in SuperDate format (see `superdate()`). If you don't want to change this date use 0.
- Action:** The `getfilefinderinfo` statement allows a procedure to determine the visible properties of a disk file. It can be used with the `setfilefinderinfo` statement to examine and modify those properties.
- Examples:** This program examines the file `Sunset.jpg` and checks to see if it has ever been modified.
- ```

local fileCodes,fileSpot,fileOptions,fileCreated,fileModified
getfilefinderinfo "", "Sunset.jpg",fileCodes,fileSpot,
    fileOptions,fileCreated,fileModified
if fileCreated=fileModified
    message "This file has not been modified since it was created."
endif

```
- Views:** This statement may be used in any view.

See Also: [setfilefinderinfo](#) statement
[filerename](#) statement
[filetypecreator](#) statement
[filesave](#) statement
[filetrash](#) statement
[folder\(\)](#) function

GETINTERNALBUTTON

- Syntax:** `GETINTERNALBUTTON top,left,height,width`
- Description:** The `getinternalbutton` statement retrieves the form relative co-ordinates of the last button that was pressed. (Form-relative are the same co-ordinates that you would get if you used the Dimension command to find the co-ordinates of the button (see the [xytoxy\(\)](#) function. The co-ordinates are measured in pixels.)
- Parameters:** This statement has four parameters: `top`, `left`, `height` and `width`.
The four parameters should be variables.
top holds the pixel distance from the top of the screen.
left holds the pixel distance from the left edge of the screen.
height holds the pixel height of the button.
width. holds the pixel width of the button.
- Action:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field. It is used in a procedure that can be triggered by a button push.
- Examples:** This example scrolls the current form so that the top left edge of the button is at the top left edge of the window.
- ```

local btop, bleft, bheight, bwidth
getinternalbutton btop, bleft,bheight,bwidth
formxy btop,bleft

```
- Views:** This statement is used in a form view when a button triggers a procedure.
- See Also:** [getbutton](#) statement  
[getlocalbutton](#) function  
[info\("trigger"\)](#) statement  
[popup](#) statement  
[setwindow](#) statement  
[floatingedit](#) statement

# GETLOCALBUTTON

**Syntax:** GETLOCALBUTTON

**Description:** The `getlocalbutton` statement retrieves the local screen co-ordinates of the last button that was pressed. (Local screen co-ordinates are the co-ordinates from the top left hand corner of the current window. The co-ordinates are measured in pixels.)

**Parameters:** This statement has four parameters: `top`, `left`, `height` and `width`.

The four parameters should be variables.

`top` holds the pixel distance from the top of the screen.

`left` holds the pixel distance from the left edge of the screen.

`height` holds the pixel height of the button.

`width` holds the pixel width of the button.

**Action:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field. It is used in a procedure that can be triggered by a button push.

**Examples:** This example allows the user to choose a color using a button that triggers a popup menu.

```

local btop,bleft,bheight,bwidth,NewColor
getlocalbutton btop,bleft,bheight,bwidth
NewColor=" "
popup "Red"+¶+"Green"+¶+"Blue"+¶+" (-"+¶+"Black" ,
 btop,bleft,"Blue",NewColor

```

**Views:** This statement is may be used in a Form view.

**See Also:** [getbutton](#) statement  
[getinternalbutton](#) function  
[info\("trigger"\)](#) statement  
[popup](#) statement  
[setwindow](#) statement  
[floatingedit](#) statement

# GETLOCALCLICK

**Syntax:** GETLOCALCLICK vertical,horizontal

**Description:** The `getlocalclick` statement gets the location of the mouse in local coordinates (coordinates from the top left corner of the current window).

**Parameters:** This statement has two parameters: `vertical` and `horizontal`  
**vertical** holds the pixel distance from the top of the screen.  
**horizontal** holds the pixel distance from the left edge of the screen.

**Action:** This statement is used to capture coordinates which can be used to position a PopUp menu, window or floating edit field.

**Examples:** This example checks to see if the mouse was clicked more than 5 pixels away from the previous mouse click.

```

fileglobal mouseV,mouseH
local newV,newH
define mouseV=0 /* in case this is the first time! */
define mouseH=0
getlocalclick newV,newH
if abs(newV-mouseV)>5 or abs(newH-mouseH)-5
 message "Clicked far away"
endif
mouseV=newV /* the new click becomes the old click for next time */
mouseH=newH

```

**Views:** This procedure statement may used in a Form view.

**See Also:** [getclick](#) statement  
[info\("trigger"\)](#) function  
[info\("click"\)](#) function  
[popup](#) statement  
[setwindow](#) statement  
[floatingedit](#) statement  
[xytoxy\(\)](#) function

# GETMAXWINDOW

**Syntax:** GETMAXWINDOW top,left,height,width

**Description:** The `GetMaxWindow` statement returns the location and size of the largest possible window on the main screen.

**Parameters:** This statement has four parameters, the top edge, the left edge, the height, and the width (all dimensions in pixels).

**Action:** The numeric values of the location and size of the largest window on the main screen are returned by the `GetMaxWindow` statement. This is the size a window would have when it is zoomed out.

**Examples:** The example below shows a typical application of this statement.

```
local ZTop, ZLeft, ZHght, ZWidth
GetMaxWindow ZTop, ZLeft, ZHght, ZWidth
...
...
```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:** [getwindow](#) statement  
[setwindow](#) statement  
[zoomwindow](#) statement

# GETMENUMARK

**Syntax:** GETMENUMARK menu,item

**Description:** The `getmenumark` statement gets the mark attached to a menu item (if any), and places it into the clipboard. The mark is usually a checkmark (for example **F**ast), but may be any character.

**Parameters:** This statement has two parameters: menu and item.

**menu** is the name or ID number of the menu that contains the item that may be marked. The menu ID is assigned in ResEdit.

**item** is the name of the menu item, or the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu is **Cleared**. This menu item may be specified as either "Cleared" or 3.

**Action:** This statement finds out if a mark is attached to a menu item, and if so, what that mark is. The mark character is placed in the clipboard.

**Examples:** Suppose a database has a **Rush** menu item in the **Order** custom menu. The example below could be part of the .CustomMenu procedure, and handles adding and removing a checkmark from the **Rush** menu item (**Rush/Rush**).

```
if info("trigger") = "Menu.Order.Rush"
getmenumark "Order","Rush"
if clipboard() = ""
setmenumark "Order","Rush",chr(18)
else
setmenumark "Order","Rush",""
endif
```

**Views:** This statement may be used in any view that has custom menus installed.

**See Also:** [setmenumark](#) statement  
[clearmenumarks](#) statement  
[getmenutext](#) statement  
[setmenutext](#) statement  
[menudisable](#) statement  
[menuenable](#) statement  
[menubuild](#) statement  
[getmenus](#) statement  
[setmenus](#) statement

# GETMENUS

**Syntax:** GETMENUS

**Description:** The `getmenus` statement gets the custom menu bar configuration and places it into the clipboard. By using this statement in combination with the `setmenus` statement a procedure can change the menu bar and then later change it back.

**Parameters:** This statement has no parameters.

**Action:** This statement creates a list of the menus in the menu bar, and places that list into the clipboard. The menu numbers in the list will be separated by spaces. Menu numbers below 128 are standard Panorama menus (see the list of standard menus below). Menus above 128 are custom menus which you create with a resource editing program like ResEdit.

| Menu   | Number | Notes                              |
|--------|--------|------------------------------------|
| Apple  | 1      | Apple Menu                         |
| File   | 7      | Use in Data Sheet window           |
| File   | 27     | Use in Form windows                |
| Window | S18    | Arrange (submenu of File menu)     |
| Edit   | E19    | Use in Data Sheet window           |
| Edit   | E37    | Use in Form windows                |
| Fields | 28     | Normally used in Data Sheet window |
| Text   | 73     | Normally used in Data Sheet window |
| Font   | S3     | Submenu of Text menu               |
| Size   | S4     | Submenu of Text Menu               |
| Search | 8      |                                    |
| Sort   | 9      |                                    |
| Math   | 10     |                                    |
| Setup  | 68     | Normally used in Data Sheet window |
| Setup  | 70     | Normally used in Form window       |

**Examples:** The example below checks to see if the current menu configuration contains a **Text** menu (menu number 73, see table above). If it does contain a **Text** menu the procedure adds custom menu 3006 at the end of the menu bar.

```

local theMenus
getmenus
theMenus=clipboard()
if theMenus contains "73"
setmenus theMenus+" 3006"
endif

```



**Views:** This statement may be used in Data Sheet and Form views.

**See Also:** [menubuild](#) statement  
[setmenus](#) statement  
[setmenutext](#) statement  
[getmenutext](#) statement  
[setmenumark](#) statement  
[getmenumark](#) statement  
[clearmenumarks](#) statement  
[menudisable](#) statement  
[menuenable](#) statement

# GETMENUTEXT

**Syntax:** GETMENUTEXT menu,item

**Description:** The `getmenutext` statement finds out the text of a menu item and places that text on the clipboard.

**Parameters:** This statement has two parameters: `menu` and `item`.

**menu** is the name or ID number of the menu that contains the item the procedure is interested in. The menu ID is assigned in ResEdit.

**item** is the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu may be **Post Now** or **Post Later**, and you want to find out which. The item parameter should be 3.

**Action:** This statement finds out the name of a single item within a menu.

**Examples:** This example toggles the fifth menu item in the Preferences custom menu between **Fast** and **Slow**.

```
getmenutext "Invoice",5
if clipboard() contains "Fast"
 setmenutext "Preferences",5,"Slow"
else
 setmenutext "Preferences",5,"Fast"
endif
```

**Views:** This statement may be used in any view that has custom menus installed

**See Also:** [setmenutext](#) statement  
[setmenumark](#) statement  
[getmenumark](#) statement  
[clearmenumarks](#) statement  
[menudisable](#) statement  
[menuenable](#) statement  
[menubuild](#) statement  
[getmenus](#) statement  
[setmenus](#) statement

# GETNAME

**Syntax:** GETNAME Raw,Prefix,First,Middle,Last,Suffix

**Description:** The `GetName` statement parses a person's name in natural format into separate fields.

**Parameters:** There are six parameters to the `GetName` statement.

The first parameter is the raw name, which should be a single line of text in a field or variable. The next five parameters are variables or fields for storing the Prefix (Mr., Mrs., Ms., Dr., etc.), First, Middle, Last, and Suffix(Jr., III, etc.).

**Action:** The `GetName` parses the person's name into its component parts.

**Examples:** The example below shows a typical application of this statement.

```
local name
global prfx, first, mi, lstnme, sufx
name="Mr. William J. Edwards, Jr."
GetName name, prfx, first, mi, lstnme, sufx
```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

# GETNSTRING(...)

- Syntax:** `GETNSTRING(type,id,number)`
- Description:** The `getnstring()` function gets a text resource from an open resource file and copies it into a variable. The string is extracted from a `STR#` resource, which holds a collection of multiple strings in each resource.
- Parameters:** This function has three parameters: `type`, `id` and `number`.
- type** is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR#" (multiple Pascal Strings). (If you specify "" for the type, Panorama will assume "STR#".)
- id** is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). `number` is the number of the string item within the collection. For example, if the collection contains 6 strings they will be numbered 0, 1, 2, 3, 4, and 5.
- Result:** This function returns whatever text is in the specified item within the specified resource collection.
- Examples:** This example displays the contents of STR# resource #693 item 12.
- ```
openresource "Accounting Messages"
message getnstring("",1296,11)
```
- All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example displays the 12th item in the Errors collection.
- ```
openresource "Accounting Messages"
message getnstring("","Errors",11)
```
- Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the type parameter. The type must be a four letter text item.
- Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a number for the number parameter.
- See Also:** [openresource](#) statement  
[openresourcerw](#) statement  
[closeresource](#) statement  
[getresource\(\)](#) function  
[getstring\(\)](#) function  
[getstringmatch\(\)](#) function  
[resourcetypes\(\)](#) function  
[resources\(\)](#) function

# GETPHONE

- Syntax:** GETPHONE raw,localarea,country,phfld1,phfldmax,phtyp
- Description:** The `getphone` statement breaks out one or more phone numbers from the raw field. It splits them into various components and formats the number into the phone number format (aaa) ppp-ssss.
- Parameters:** This statement has six parameters: raw, localarea, country, phfld1,phfldmax, phtype
- Be sure to read all requirements before using. They vary depending upon the nature of phfld1.
- raw** may be in a field, variable. raw contains one or more phone numbers, separated by spaces or carriage returns.
- localarea** may be a field or variable but is usually a "constant". It specifies the **area code** used by `getphone` if the local **area code** is missing from the first number. **Alert:** if subsequent numbers in raw don't have and **area code**, they use the previous number's area code, not the localarea number.
- country** may be in a field or variable or formula but is usually a "constant". If it's something other than blank [""], or USA or CANADA (note: capitalization), the phone number won't be formatted.
- phfld1** may be in a field or array. It holds the **phone type**, like "voice" or "fax" that's extracted by `getphone`. If fields are used, they must appear as a pair with the phone field in the design sheet. For example, phfld1, phone1, phfld2, phone2, phfld3, phone3. Note that the field names don't matter. They must just end with a sequential number and appear in phonetype,phonenumber order.
- phfldmax** may be in a field or variable but is usually a "constant". It specifies the number of pairs of phone type and phone number field pairs set up to hold data. **Alert:** if you are using an array for phfld1, then phfldmax must be zero. **phtype** may be in a field or variable but is usually a "constant". It specifies the phone type used by `getphone` if one isn't found for the number. Usually "Voice" is the default value.
- Action:** The parsing rules are thus: The phone type description in raw must start with a letter. Everything up to the first number is taken as the type's description, except for a trailing space or spaces between the last letter and the first number. Next follows the unformatted phone number if the phone number starts with a 1, and the country parameter calls for formatting, the 1 will be removed. Extension information can follow the phone number, usually separated by a space. But the extension information must begin with x, ext, extension, rm, room, op, or operator, otherwise it could be misinterpreted.
- Note:** The default type only applies to the first phone number. if subsequent phone numbers are separated by spaces rather than carriage returns, they must have a type. If subsequent phone numbers are separated by carriage returns, the default type is applied to the number.
- Because there are many ways to interpret the phone data, it's recommended that you try this with several examples of your expected data to find the appropriate use of type descriptions, spaces, and carriage returns.

**Examples:**

In a field called «rawphones» you have the following content: 4085552345 office 5553434 fax 8971234. The Design Sheet has fields aaa, bbb, ccc, phntyp1, here1, phntyp2, here2, phntyp3, here3, xxx, yyy, zzz

```
getphone «rawphones», "714", "USA", phntyp1, 3, "Voice",
```

would return

Voice in phntyp1  
 (408) 555-2345 in here1  
 Office in phntyp2  
 (408) 555-3434 in here2  
 Fax in phntyp3  
 (408) 897-1234 in here3

**Note** that because the first number already had an area code, 714 wasn't used. Also, the first letter of the phone types was capitalized.

If instead of splitting the phone numbers in to separate variables, you wanted to store them in a field or variable called HoldMe, that statement would look like:

```
getphone «rawphones», "714", "USA", «HoldMe», 0, "Voice",
```

This would result in one phone type and number stored per line. it would look like:

Voice (408) 555-2345  
 Office (408) 555-3434  
 Fax (408) 897-1234

in «HoldMe»

extract() and array() are functions that could be used to pull specific types and/or numbers from «HoldMe»

**Views:**

This statement can be used in any view.

**See Also:**

extract(text,separator,element) function  
array(text, item, separator) function

# GETPROCEDURETEXT

- Syntax:** `GETPROCEDURETEXT database,procedure, variable`
- Description:** The `getproceduretext` statement gets the contents (source) of a procedure and places it in a variable.
- Parameters:** This statement has three parameter: database, procedure and variable.  
**database** is the name of the database that contains the procedure.  
**procedure** is the name of the procedure.  
**variable** is the name of the variable you wish to place the procedure text into.
- Action:** This statement allows a procedure to extract and look at the text (source code) of other procedures. This statement is disabled if the user is not authorized to see the contents of the procedure.
- Examples:** This example scans through all of the procedures in the database named `Contacts` (which must be already open) and builds a list of procedures that contain the word `Name`.
- ```

local tempProcedureList,tempProcedureText,
    n,procedureName,activeDatabase,procedureSearchText
activeDatabase="Contacts"
procedureSearchText="Name"
tempProcedureList=dbinfo("procedures",activeDatabase)
n=1
tempProcedureText=""
viewList
loop
    procedureName=array(tempProcedureList,n,¶)
    stoploopif procedureName=""
    getproceduretext activeDatabase,procedureName,tempProcedureText
    if tempProcedureText contains procedureSearchText
        viewList=sandwich(" ",viewList,¶)+procedureName
    endif
    n=n+1
while forever

```
- Views:** This statement may be used in any view.
- See Also:** [openprocedure](#) statement

GETRESOURCE(...)

Syntax: GETRESOURCE(type,id)

Description: The `getresource()` function gets a resource from an open resource file and copies it into a variable.

Parameters: This function has two parameters: `type` and `id`.

type is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).

id is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

Result: This function returns whatever binary data is in the specified resource (see [binary data](#)).

Examples: This example loads the contents of TEXT resource #415 into the field LetterBody.

```
openresource "Letter Templates"
LetterBody=getresource("TEXT",415)
```

All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example loads the contents of the TEXT resource named Thank You #2 into the field LetterBody.

```
openresource "Letter Templates"
LetterBody=getresource("TEXT","Thank You #2")
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the type parameter. The type must be a four letter text item.

See Also: [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[getstring\(\)](#) function
[getnstring\(\)](#) function
[getstringmatch\(\)](#) function
[resourcetypes\(\)](#) function
[resources\(\)](#) function

GETSCRAP

- Syntax:** GETSCRAP prompt
- Description:** The `getscrap` statement displays a dialog. This dialog asks the user to enter an item of text.
- Parameters:** This statement has one parameter: `prompt`.
- prompt** is a message that will be displayed in the dialog. This message should explain what the user needs to enter.
- This statement displays a dialog with a single area for text entry. The user may enter something and press **Ok**, or they may press the **Stop** button. If they press **Ok**, the text they entered will be placed on the clipboard. If they press the **Stop** button the procedure will stop.
- Action:** The dialog is usually just large enough to enter one line containing about 25 characters of text. To make this dialog larger use the [customdialog](#) statement.
- Examples:** This example asks the user to enter an area code, then selects all phone numbers in that area code. The default area code is **909**.
- ```
getscrap "Area code:"
select Phone match "(" + clipboard() + ") *"
```
- Views:** This statement may be used in any view
- See Also:** [customdialog](#) statement  
[getscrapok](#) statement  
[gettext](#) statement

# GETSCRAPOK

- Syntax:** GETSCRAPOK prompt
- Description:** The `getscrapok` statement displays a dialog. This dialog asks the user to enter an item of text.
- Parameters:** This statement has one parameter: `prompt`.  
**prompt** is a message that will be displayed in the dialog. This message should explain what the user needs to enter.
- Action:** This statement displays a dialog with a single area for text entry. The user may enter something and press **Ok**. The text they entered will be placed on the clipboard. (Note: This dialog has no **Stop** button. If you need a **Stop** button use the `getscrap` or `gettext` statements.) The dialog is usually just large enough to enter one line containing about 25 characters of text. To make this dialog larger use the `customdialog` statement.
- Examples:** This example asks the user to enter an area code, then selects all phone numbers in that area code. The default area code is 909.
- ```
getscrapok "Area code:"  
select Phone match "(" + clipboard() + ")*"
```
- Views:** his statement may be used in any view.
- See Also:** [customdialog](#) statement
[getscrap](#) statement
[gettext](#) statement

GETSTRING(...)

Syntax: GETSTRING(type,id)

Description: The `getstring()` function gets a text resource from an open resource file and copies it into a variable.

Parameters: This function has two parameters: `type` and `id`.

`type` is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR " (Pascal String). (If you specify "" for the type, Panorama will assume "STR ".)

`id` is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

Result: This function returns whatever text is in the specified resource.

Examples: This example displays the contents of STR resource #1296.

```
openresource "Accounting Messages"
message getstring("",1296)
```

All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example displays the text in the Overflow Error resource.

```
openresource "Accounting Messages"
message getstring("", "Overflow Error")
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the type parameter. The type must be a four letter text item.

See Also: [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function
[getnstring\(\)](#) function
[getstringmatch\(\)](#) function
[resourcetypes\(\)](#) function
[resources\(\)](#) function

GETSTRINGMATCH(...)

Syntax: GETSTRINGMATCH(type,id,text)

Description: The `getstringmatch()` function searches through a collection of multiple strings in a STR# resource. If it finds a match with the text you supply, it returns the number of the text item within the collection.

Parameters: This function has three parameters: `type`, `id` and `text`.

type is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR#" (multiple Pascal Strings). (If you specify "" for the type, Panorama will assume "STR#".)

id is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

text is the text you want to search for. For a match, this text must be exactly the same as one of the text items in the STR# collection.

Result: This function returns a number. If the text does not match any of the text items in the STR# collection, the function will return 0. If there is a match, the function will return the number of the item that matched, starting with 1 for the first item. (Notice that this numbering system is different than the `getstring()` function, which starts with 0 for the first item.)

Examples: One application for this function is looking up commands or keywords. Suppose you have a STR# resource #320 that contains the following text items.

```
DIAL
APPOINTMENT
TODO
LETTER
CHECK
```

Now suppose the database has a global variable called `CommandLine`. The user types a command into this variable with a Text Editor SuperObject™. Here is part of a procedure that can process these commands using the STR# 320 resource.

```
local commandWord, commandNumber, commandExtras
openresource "Accounting Extras"
commandWord=upper(strip(CommandLine[1," "]))
commandExtras=strip(CommandLine["",-1])
commandNumber=getstringmatch("",320,commandWord)
if commandNumber=0 stop endif
if commandNumber=1
dial commandExtras
endif
if commandNumber=2
...
endif
if commandNumber=3
...

```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the type parameter. The type must be a four letter text item.

Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a number for the number parameter.

See Also: [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function
[getnstring\(\)](#) function
[getstringmatch\(\)](#) function
[resourcetypes\(\)](#) function
[resources\(\)](#) function

GETTEXT

Syntax: GETTEXT prompt,input

Description: The `gettext` statement displays a dialog. This dialog asks the user to enter an item of text

Parameters: This statement has two parameters: prompt and input.

prompt is a message that will be displayed in the dialog. This message should explain what the user needs to enter.

input is a field or variable where the user's input will be placed. If this is a field, it should be a text field.

The input field or variable should be assigned a value before the `gettext` statement is used. This value will be displayed as the default value in the dialog.

Action: This statement displays a dialog with a single area for text entry. The user may enter something and press Ok, or they may press the Stop button. If they press the Stop button the procedure will stop. The dialog is usually just large enough to enter one line containing about 25 characters of text. To make this dialog larger use the `customdialog` statement.

Examples: This example asks the user to enter an area code, then selects all phone numbers in that area code. The default area code is 909.

```
local whatArea
whatArea="909"
gettext "Area code:",whatArea
select Phone match "("+whatArea+")*"
```

Views: This statement may be used in any view.

See Also: [customdialog](#) statement
[getscrap](#) statement
[getscrapok](#) statement

GETWINDOW

Syntax: GETWINDOW Top,Left,Height,Width

Description: The `GetWindow` retrieves the current position and size of the current window.

Parameters: The `GetWindow` statement has four parameters, the top edge, the left edge, the height, and the width (all dimensions in pixels).

Action: The `GetWindow` retrieves the current position and size of the current window. These values can be used for the [setwindow](#) or [zoomwindow](#) statements.

Examples: The example below shows a typical application of this statement.

```
local WTop, WLeft, WHght, WWidth  
GetWindow WTop, WLeft, WHght, WWidth  
...
```

Views: This statement may be used in a procedure run from any view, and also works when no windows are open at all.

See Also: [setwindow](#) statement
[zoomwindow](#) statement

GLOBAL

Syntax: GLOBAL variables

Description: The **global** statement creates one or more global variables. Global variables may be used by any procedure, and remain active until you quit from Panorama.

Parameters: This statement has one parameter: **variables**.

variables is a list of variables to be created. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: This statement creates one or more global variables. Global variables can be used to hold pieces of information (numbers or text). Each variable has a name.

Panorama keeps all global variables together in a common “pool” that is used by all procedures and SuperObjects. If procedure A creates a global variable named myValue, that local variable may be used by all other procedures, in any open database. In fact you must be careful to avoid conflicts when creating global variables. If the name of a global variable in one procedure is the same as the name of a global variable in another procedure (even in a database) that is used for a different purpose, a conflict will result. Global variable names can also conflict with field names. One solution to variable conflicts between databases is to use the fileglobal statement instead of the global statement.

Examples: The example creates two global variables, [Channel](#) and [Home Timeouts](#).

```
global Channel, «Home Timeouts»
```

You may change the value of a variable with an assignment, like this:

```
«Home Timeouts»=«Home Timeouts»-1
```

Views: This statement may be used in any view.

See Also: [fileglobal](#) statement
[local](#) statement
[windowglobal](#) statement
[permanent](#) statement
[globalize](#) statement
[undefine](#) statement

GLOBALIZE

Syntax: GLOBALIZE variables

Description: The `globalize` statement converts one or more [fileglobal](#) variables into [global](#) variables.

Parameters: This statement has one parameter: variables.

variables is a list of variables to be converted. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: This statement converts one or more file global, window, or local variables into global variables. The variables keep their values. This statement is especially convenient when you need to use a [permanent](#) variable as a global variable, instead of as a [fileglobal](#) variable.

Examples: The example creates a permanent variable named pAreaCode, then makes it global so any database can access it.

```
permanent pAreaCode
pAreaCode="714"
globalize pAreaCode
```

An important point to keep in mind is that if you convert the file global, window or local variable again you will now have two variables. If you want to have just one, don't create the variable twice. The example below shows how to do this with a [permanent](#) variable.

```
if info("globalvariables") notcontains pAreaCode
    /* we have not defined this variable yet */
    permanent pAreaCode /* create the variable */
    globalize pAreaCode /* convert it to a global */
endif
```

Views: This statement may be used in any view.

See Also: [fileglobal](#) statement
[local](#) statement
[windowglobal](#) statement
[permanent](#) statement
[info\("globalvariables"\)](#) function
[info\("filevariables"\)](#) function
[info\("windowvariables"\)](#) function
[dbinfo\(\)](#) function

GOCROSSTAB

- Syntax:** `GOCROSSTAB crosstab`
- Description:** The `gocrosstab` statement opens a crosstab from the current database in the current window.
- Parameters:** This statement has one parameter: `crosstab`.
`crosstab` is the name of the crosstab to open.
- Action:** This statement opens a crosstab in the current window. The effect is similar to selecting the crosstab from the View menu (the pop-up menu in the window title). If the crosstab is already open in another window, that window is simply brought to the front.
- Examples:** The procedure below opens the crosstab Budget. If the current window is a crosstab then Budget will be opened in the current window. If the current window is not a crosstab then Budget will be opened in a new window that is offset 16 pixels from the current window.

```

local newWindowRect
if info("typeofwindow")contains "cross"
    gocrosstab "Budget"
else
    newWindowRect=rectangleadjust(
        info("windowrectangle"),16,16,16,16)
    setwindowrectangle newWindowRect,
        "noHorzScroll noVertScroll noPalette"
    opencrosstab "Budget"
endif

```

Views: This statement may be used in any view.

See Also: [opencrosstab](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[openprocedure](#) statement
[openform](#) statement
[gosheet](#) statement
[godesignsheet](#) statement
[goform](#) statement
[goprocedure](#) statement
[info\("windows"\)](#) function
[listwindows\(\)](#) function

GODESIGN SHEET

Syntax: ODESIGN SHEET

Description: The `godesignsheet` statement opens the design sheet window for the current database in the current window.

Parameters: This statement has no parameters

Action: This statement opens the design sheet in the current window. The effect is similar to selecting Design Sheet from the View menu (the pop-up menu in the window title. If the design sheet is already open in another window, that window is simply brought to the front.

Examples: The procedure below opens the design sheet in the current window.

```
godesignsheet
```

Views: This statement may be used in any view.

See Also: [newgeneration](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[opencrosstab](#) statement
[gosheet](#) statement
[goform](#) statement
[goprocedure](#) statement
[gocrosstab](#) statement
[setwindow](#) statement
[setwindowrectangle](#) statement
[windowbox](#) statement
[info\("windows"\)](#) function
[listwindows\(\)](#) function

GOFORM

Syntax: GOFORM form

Description: The `goform` statement opens a form from the current database in the current window.

Parameters: This statement has one parameter: form.

form is the name of the form to open.

Action: This statement opens a form in the current window. The effect is similar to selecting the form from the View menu (the pop-up menu in the window title). If the form is already open in another window, that window is simply brought to the front.

Examples: The procedure below opens the form Utilities. If the current window is a form then Utilities will be opened in the current window. If the current window is not a form then Utilities will be opened in a new window that is offset 16 pixels from the current window.

```

local newWindowRect
if info("typeofwindow")contains "form"
    goform "Utilities"
else
    newWindowRect=rectangleadjust(
        info("windowrectangle"),16,16,16,16)
        setwindowrectangle newWindowRect,
            "noHorzScroll noVertScroll noPalette"
        openform "Utilities"
endif

```

Views: This statement may be used in any view.

See Also: [openform](#) statement
[formxy](#) statement
[opendialog](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[openprocedure](#) statement
[opencrosstab](#) statement
[gosheet](#) statement
[godesignsheet](#) statement
[goprocedure](#) statement
[gocrosstab](#) statement
[info\("windows"\)](#) function
[listwindows\(\)](#) function

GOPROCEDURE

- Syntax:** GOPROCEDURE procedure
- Description:** The `goprocedure` statement opens a procedure from the current database in the current window.
- Parameters:** This statement has one parameter: `procedure`.
`procedure` is the name of the procedure to open.
- Action:** This statement opens a procedure in the current window. The effect is similar to selecting the procedure from the View menu (the pop-up menu in the window title). If the procedure is already open in another window, that window is simply brought to the front.
- Examples:** The procedure below opens the procedure `.CustomMenu`. If the current window is a procedure then `.CustomMenu` will be opened in the current window. If the current window is not a procedure then `.CustomMenu` will be opened in a new window that is offset 16 pixels from the current window.

```

local newWindowRect
if info("typeofwindow")contains "procedure"
    goprocedure ".CustomMenu"
else
    newWindowRect=rectangleadjust(
        info("windowrectangle"),16,16,16,16)
    setwindowrectangle newWindowRect,
        "noHorzScroll noVertScroll noPalette"
    openprocedure ".CustomMenu"
endif

```

Views: This statement may be used in any view.

See Also: [openprocedure](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[opencrosstab](#) statement
[openform](#) statement
[gosheet](#) statement
[godesignsheet](#) statement
[goform](#) statement
[gocrosstab](#) statement
[info\("windows"\)](#) function
[listwindows\(\)](#) function

GOSHEET

Syntax: GOSHEET

Description: The `gosheet` statement opens the data sheet for the current database in the current window.

Parameters: This statement has no parameters.

Action: This statement opens the data sheet. The effect is similar to selecting Data Sheet from the View menu (the pop-up menu in the window title). If the data sheet is already open in another window, that window is simply brought to the front.

Examples: The procedure below opens the data sheet in the current window. `gosheet`

Views: This statement may be used in any view.

See Also: [opensheet](#) statement
[opendesignsheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[opencrosstab](#) statement
[godesignsheet](#) statement
[goform](#) statement
[goprocedure](#) statement
[gocrosstab](#) statement
[info\("windows"\)](#) function
[listwindows\(\)](#) function

GOTO

Syntax: GOTO label

Description: The **goto** statement allows a procedure to arbitrarily jump from one spot to another within the procedure.

Parameters: This statement has one parameter: label.

label is the spot the procedure is supposed to jump to. A label is a unique series of letters and numbers that identifies a location within the procedure. The label may not contain any spaces or punctuation except for . and %, and must always end with a colon. The colon is not actually part of the label, it simply identifies the series of letters and numbers as a label instead of a field or variable.

Action: This statement jumps directly to another spot in the current procedure. We recommend that you avoid the **goto** statement if possible. The **goto** statement tends to make it difficult to understand the logic of a procedure.

Examples: The example jumps back to the beginning of the procedure if there is an error. The procedure has one label, which is on the first line. We have highlighted the label in bold to make it more clear.

```
tryOpen:  
openfile dialog  
if error  
    goto tryOpen  
endif
```

Here is a procedure that does the same thing, but without the **goto**. If you use the **goto** statement too much, your programs will start looking like “spaghetti.”

```
loop  
    openfile dialog  
    if error  
        repeatloopif 1=1  
    endif  
while 1=0
```

Views: This statement may be used in any view.

See Also: [if statement](#)
[loop statement](#)
[case statement](#)

GRABDATA(...)

Syntax: GRABDATA(file,field)

Description: The `grabdata()` function grabs the contents of a field in the current record of a database. You can grab data from the current database, or from another database.

Parameters: This function has two parameters: `file` and `field`.

file is the name of the database that you want to grab data from. The database must be open. If you specify "" for the file name, `grabdata()` will grab data from the current file.

field is the name of the field that you want to grab data from. The field must be in the database specified by the first parameter.

Result: The function returns the contents of the specified field in the specified database. Unlike `lookup()` functions, the `grabdata()` function does not search for the data, it simply uses whatever data is in the currently active record for that database.

Examples: The `grabdata()` function identifies data by position. This function retrieves data from the current record in the specified database. For example, if the user has selected [7404 hex inverter](#) in the [Price List](#) database by clicking on it (or searching for it with the **Find/Select** command), this procedure will copy the information into the current line item in the Invoice database.

```
ItemΩ=grabdata("Price List",Part)
PriceΩ=grabdata("Price List",Price)
```

The drawback of the `grabdata()` function is that the user must manually locate the data, but sometimes this is exactly what you want. You can also use the `grabdata()` function to grab data the current record, but where you need to calculate the field name on the fly. Suppose you have a database that has four fields for phone numbers: [Phone1](#), [Phone2](#), [Phone3](#), [Phone4](#). Another field, [PrimaryPhone](#), contains a number (1-4) telling which of these phone numbers is the primary phone number. The procedure below will dial the primary phone number thru the speaker.

```
dial grabdata("","Phone"+str(PrimaryPhone))
```

If the formula calculating the field name is a single field or variable, you must surround it with (and). For example, `grabdata("",(Primary))` tells Panorama to grab the contents of the field whose name is in `Primary`. The formula `grabdata(" ",Primary)`, however, tells panorama to grab the contents of the field `Primary` itself.

Errors: **Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also: [lookup\(\)](#) function

GRABFILEVARIABLE(...)

- Syntax:** GRABFILEVARIABLE(file,variable)
- Description:** The `grabfilevariable()` function makes it possible to access a [fileglobal](#) or [permanent](#) variable from other databases. (Usually these variables can only be accessed from the database in which they were created.)
- Parameters:** This function has two parameters: `file` and `variable`.
- file** is the name of the database that contains the [fileglobal](#) or [permanent](#) variable. Note: This database must currently be open!
- variable** is the name of the variable you want to access. In general, this variable name must be enclosed in quotes (unless you are using a formula to calculate the name).
- Result:** The result of this function is whatever value that is contained in the specified variable. This may be text or numeric.
- Examples:** This example dials the phone using the default area code stored as a [permanent](#) variable in the Prefs database
- ```
dial grabfilevariable("Prefs","DefaultAreaCode")+Phone
```
- Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for either the file or variable name.
- See Also:** [fileglobal](#) statement  
[permanent](#) statement  
[info\("filevariables"\)](#) function

# GRABWINDOWVARIABLE(...)

**Syntax:** GRABWINDOWVARIABLE(window,variable)

**Description:** The `grabwindowvariable()` function makes it possible to access a [windowglobal](#) variable in a different window from the one in which it was created.

**Parameters:** This function has two parameters: `window` and `variable`.

**window** is the name of the window in which the variable was created.

**variable** is the name of the variable you want to access. In general, this variable name must be enclosed in quotes (unless you are using a formula to calculate the name).

**Result:** The result of this function is whatever value that is contained in the specified variable. This may be text or numeric.

**Examples:** This example below assumes that there is a series of windows with names that end with (1), (2), (3) etc. It scans through the window until it finds a window where the Company variable is Apple. If it finds such a window it makes it the top window.

```

local wX, wName
wX=1
loop
 wName=info("databasename")+":"+iinfo("formname")+ ("+"str(wX)+")"
 stoploopif info("windows") notcontains wName
 if grabwindowvariable(wName,"Company")="Apple"
 window wName
 rtn
 endif
 wX=wX+1
while forever

```

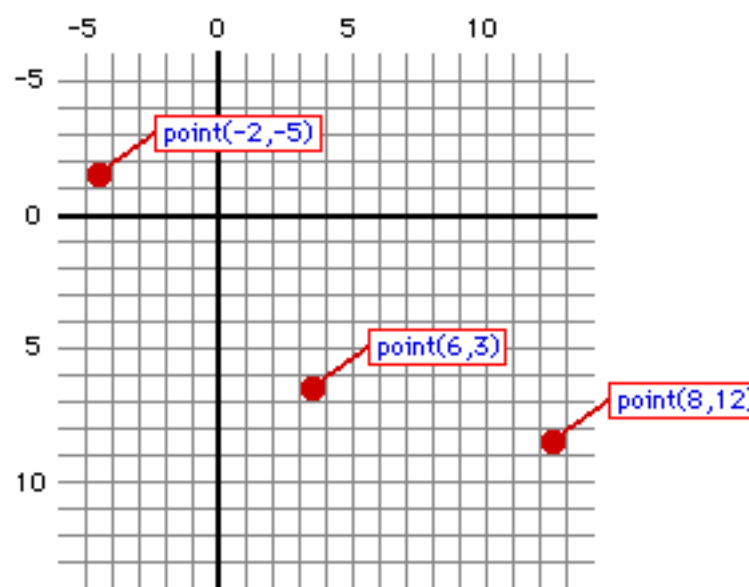
**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for either the window or variable name.

**See Also:** [windowglobal](#) statement  
[info\("windowvariables"\)](#) function

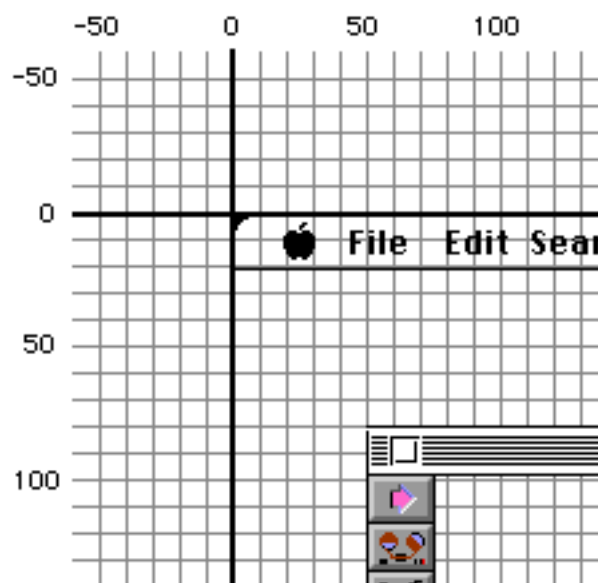
# Graphic Coordinates

**Background:** Many Panorama operations refer to locations on the screen, within a window, or within a form. In fact, Panorama has two secondary data types that refer to graphic locations: point and rectangles.

**X-Y Grid:** The Macintosh uses an X-Y coordinate system to define locations. This X-Y system divides any area into an invisible grid of criss-crossing lines. There are 72 lines per inch. Each point where the lines intersect is identified by two numbers, the vertical and the horizontal position. The numbers increase as you move down and to the right. The illustration below shows a greatly expanded view of the X-Y coordinate system with several sample points.

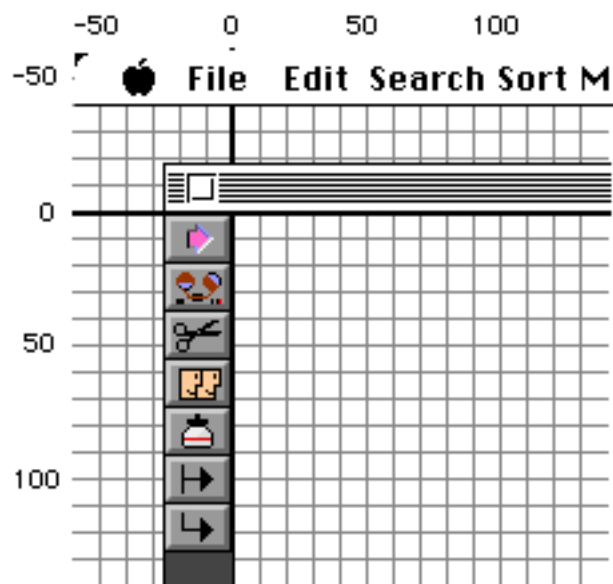


**Screen** Panorama actually has three different X-Y co-ordinate systems. The difference between these systems is the location of zero-zero. Screen relative co-ordinates (sometimes called global co-ordinates) measure all locations from the upper left hand corner of menu bar. This actual size illustration shows screen relative co-ordinates.



## Window

Window relative co-ordinates (sometimes called local co-ordinates) measure all locations from the upper left hand corner of the current window. This actual size illustration shows window relative co-ordinates.



## Form Relative:

Form relative co-ordinates measure all locations from the upper left hand corner of the current form. If the form has not been scrolled, this is exactly the same as window relative co-ordinates. However, if the form has been scrolled, the zero-zero point is not visible, but is somewhere above and/or to the left of the upper left hand corner of the window.

The `xytoxy()` function can convert a point or rectangle from any of the three co-ordinate systems to any of the other three co-ordinate systems.

## See Also:

[xytoxy\(\)](#) function  
[point\(\)](#) function  
[v\(\)](#) function  
[h\(\)](#) function  
[info\("click"\)](#) function  
[info\("mouse"\)](#) function  
[rectangle\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[tright\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function

# GRAPHICSMODE

**Syntax:** GRAPHICSMODE

**Description:** The `graphicsmode` statement switches a form into graphics mode. This is the same as pressing the **Switch to Graphic Design Mode** tool in the tool palette.

**Parameters:** This statement has no parameters.

**Examples:** This example opens a form and sets it up for graphic editing.

```
openform "Letterhead"
graphicsmode
```

**Views:** This statement may be used in a Form view.

**See Also:** [openform](#) statement  
[goform](#) statement

# GREEN(...)

**Syntax:** GREEN(color)

**Description:** The `green()` function extracts the green intensity from a color.

**Parameters:** This function has one parameter: color.

**color** is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

**Result:** This function extracts the intensity of the green component of this color. This intensity is a number between 0 (black) and 65535 (full intensity).

**Examples:** The example below calculates the green intensity of the color (in percent, from 0 to 100%).

```
Intensity=green(HighlightColor)*100/65535
```

**Errors:** For more examples of color, see [colors](#).

**Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the color parameter.

**See Also:** [rgb\(\)](#) function  
[hsb\(\)](#) function  
[red\(\)](#) function  
[blue\(\)](#) function  
[hue\(\)](#) function  
[saturation\(\)](#) function  
[brightness\(\)](#) function  
[objectinfo\(\)](#) function  
[changeobjects](#) function  
[colorwheel](#) statement  
[colors](#)

# GROUP

**Syntax:** `GROUP period`

**Description:** The `group` statement divides the database into groups. Each group has a summary record at the end of the group. The groups are arranged in ascending alphabetical order (A to Z).

**Parameters:** This statement has one parameter: `period`. `period` identifies the span of each group. This parameter only applies if the current field is a date field. The options for this parameter are listed below. Use the options exactly as shown, with no quotes.

`by Day`  
`by Week`  
`by Month`  
`by Quarter`  
`by Year`

If the current field is a text, numeric or choice field this parameter should be omitted.

**Action:** This statement divides the database into groups according to the current field. For example if the current field contains company names there will be one group per company, if the current field contains zip codes there will be one group per zip code.

**Examples:** This example calculates summaries for cities and states, then displays the summary information. The original data is hidden.

```
field State
group
field City
group
field Amount
total
outlinelevel "1"
```

This example calculates totals for each month.

```
field Date
group by month
field Debit
total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:**

- [`groupup` statement](#)
- [`groupdown` statement](#)
- [`groupbycolor` statement](#)
- [`outlinelevel` statement](#)
- [`removesummaries` statement](#)
- [`removedetail` statement](#)
- [`sortup` statement](#)
- [`sortdown` statement](#)

# GROUPBYCOLOR

**Syntax:** GROUPBYCOLOR

**Description:** The `groupbycolor` statement divides the database into groups of like colors. Each group has a summary record at the end of the group.

**Parameters:** This statement has no parameters.

**Action:** This statement divides the database into groups according to the colors of cells in the current field. The colors are grouped in this order: black, red, green, blue, cyan, magenta, yellow.

**Examples:** This example calculates totals for each color of name.

```
field Name
groupbycolor
field Debit
total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [fieldstyle\(\)](#) function  
[group](#) statement  
[groupup](#) statement  
[groupdown](#) statement  
[outlinelevel](#) statement  
[removesummaries](#) statement  
[removedetail](#) statement  
[sortup](#) statement  
[sortdown](#) statement



# GROUPDOWN

**Syntax:** `GROUPDOWN period`

**Description:** The `groupdown` statement divides the database into groups. Each group has a summary record at the end of the group. The groups are arranged in descending alphabetical or numeric order (Z to A).

**Parameters:** This statement has one parameter: `period`. `period` identifies the span of each group. This parameter only applies if the current field is a date field. The options for this parameter are listed below. Use the options exactly as shown, with no quotes.

by Day  
by Week  
by Month  
by Quarter  
by Year

If the current field is a text, numeric or choice field this parameter should be omitted.

**Action:** This statement divides the database into groups according to the current field. For example if the current field contains company names there will be one group per company, if the current field contains zip codes there will be one group per zip code.

**Examples:** This example calculates totals for each month, with the most recent months toward the top of the database.

```
field Date
groupdown by month
field Debit
total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [group](#) statement  
[groupup](#) statement  
[groupbycolor](#) statement  
[outlinelevel](#) statement  
[removesummaries](#) statement  
[removedetail](#) statement  
[sortup](#) statement  
[sortdown](#) statement

# GROUPUP

**Syntax:** GROUPUP period

**Description:** The **groupup** statement divides the database into groups. Each group has a summary record at the end of the group. The groups are arranged in ascending alphabetical order (A to Z).

**Parameters:** This statement has one parameter: **period**. **period** identifies the span of each group. This parameter only applies if the current field is a date field. The options for this parameter are listed below. Use the options exactly as shown, with no quotes.

by Day  
by Week  
by Month  
by Quarter  
by Year

If the current field is a text, numeric or choice field this parameter should be omitted.

**Action:** This statement divides the database into groups according to the current field. For example if the current field contains company names there will be one group per company, if the current field contains zip codes there will be one group per zip code.

**Examples:** This example calculates summaries for cities and states, then displays the summary information. The original data is hidden.

```
field State
groupup
field City
groupup
field Amount
total
outlinelevel "1"
```

This example calculates totals for each month.

```
field Date
groupup by month
field Debit
total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:**

- [group](#) statement
- [groupdown](#) statement
- [groupbycolor](#) statement
- [outlinelevel](#) statement
- [removesummaries](#) statement
- [removedetail](#) statement
- [sortup](#) statement
- [sortdown](#) statement

## H(...)

- Syntax:** H(point)
- Description:** The **h**( function extracts the horizontal position from a point (see [point\(, graphic coordinates\)](#)).
- Parameters:** This function has one parameter: **point**
- point** is a number that includes both the vertical and horizontal components of a position. This number is usually created with the [point\(](#), [info\("click"\)](#), or [info\("mouse"\)](#) functions.
- Result:** This function returns a number (an integer) that describes the horizontal position of the point. This number will be between -32,768 and +32,767. (Like standard cartesian coordinates, positive is right and negative is left.)
- Examples:** The procedure below displays a message if you click on a spot less than 75 pixels from the left edge of the screen.
- ```
if h( info("click")) ≤ 75
message "You're near the left edge!"
endif
```
- Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the point parameter.
- See Also:** [xytoxy\(function](#)
[point\(function](#)
[v\(function](#)
[info\("click"\) function](#)
[info\("mouse"\) function](#)
[rectangle\(function](#)
[rtop\(function](#)
[rbottom\(function](#)
[rleft\(function](#)
[rright\(function](#)
[info\("screenrectangle"\) function](#)
[info\("windowrectangle"\) function](#)
[info\("buttonrectangle"\) function](#)
[info\("cursorrectangle"\) function](#)

HIDE

Syntax: HIDE

Description: The **hide** statement turns off most of the normal screen drawing Panorama does after each statement. This makes the procedure run faster and look cleaner.

Note: Panorama 3.1 introduced the [noshow](#) and [endnoshow](#) commands, which we recommend you use instead of [hide](#) and [show](#).

Parameters: This statement has no parameters.

Action: To eliminate unnecessary erasing and display of screens, you should use the **hide** and [show](#) statements. The **hide** statement tells Panorama not to redraw windows after each procedure statement or command. Redrawing remain off until the end of the procedure, or until the [show](#) or [showfields](#) statement turns them back on again. The [show](#) statement redraws the window immediately and restores normal operation. **Warning:** Never switch windows or open new windows while redrawing is turned off! This may cause crashes, or in rare cases, data corruption.

Examples: The example below would normally erase and re-display the window four times. Adding the **hide** and [show](#) statements suppresses the extra drawing. The window is only erased and re-displayed once at the end of the procedure.

```
Hide
  field State
  groupup
  field City
  groupup
  field Fees
  total
  outlinelevel "1"
  show
```

Views: This statement may be used in the Data Sheet, Design Sheet, Crosstab or Form Views.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[show](#) statement
[noundo](#) statement

HSB(...)

Syntax: HSB(hue,saturation,brightness)

Description: The `hsb()` function creates a color by combining hue, saturation, and brightness components. See [colors](#).

Parameters: This function has three parameter: hue, saturation and brightness.

hue specifies where this color falls in the spectrum. If you are familiar with the standard Apple color picker, the Hue would specify the angle of the color from the center of the wheel. This must be a number from 0 to 65535.

saturation specifies how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? Again using the standard Apple color picker, the Saturation would specify the distance of the color from the center of the wheel. This must be a number from 0 (black) to 65535 (full intensity).

brightness specifies how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does. On the Apple color picker the Brightness is specified by the scroll bar on the right. This must be a number from 0 (black) to 65535 (full intensity).

Result: This function returns 6 bytes of raw binary data (see [binary data](#)).

Examples: The example below changes the color of any object named Border to orange.

```
local Orange
Orange=rgb( 3563,62600,65535)
selectobjects objectinfo("name") = "Border"
changeobjects "color",Orange
```

For more examples of colors see [colors](#).

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for any parameter.

See Also: [rgb\(\)](#) function
[hsb\(\)](#) function
[red\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement
[colors](#)

HTML TABLES

Introduction: Starting with Panorama 3.1, Panorama's text import capability has been enhanced to allow the import of HTML tables. Panorama automatically checks any text file you import for a table tag. If the text file contains a `-table-` tag Panorama will parse the HTML and input the data in the table. All other text will be ignored. If there is more than one table, only the first will be imported. If the table data contains HTML tags (``, ``, etc.) those tags are retained and become part of the Panorama database.

If the import is creating a new database (instead of appending or replacing the data in an existing database), Panorama automatically creates the columns for you. If the HTML table has `<TH>` tags (table header) Panorama will automatically set up the field names from the table header.

Any Panorama import may use the HTML import feature, including imports set up by procedures. You can even import an HTML table contained in a variable using the `@` notation. This could be useful if you want to pre-parse the HTML before importing the tables.

Examples: This example imports the first web page in the file `SomeTables.html`. The imported data is brought into a new, untitled data sheet.

```
openfile "SomeTables.html"
```

This more extensive data imports ALL of the tables found in `SomeTables.html`. If there are 10 tables, all ten will be imported.

```
local MyWebPage,myTable,n
MyWebPage=fileload("","SomeTables.html")
n=1
loop
  myTable=tagdata(MyWebPage,"","",n
  stoploopif myTable=""
  myTable=""+myTable+"
  openfile "@MyTable"
  n=n+1
until forever
```

See Also: [openfile](#) statement
[tagdata\(\)](#) function

HTML TAG PARSING

Background: HTML (Hyper Text Markup Language) is a specification that allows commands and text to be mixed together to create a document. Commands are embedded into the text in the form of "tags." Some common HTML tags include:

Tag	Description
	Start bold text
	End bold text
<I>	Start italic text
</I>	End italic text
<CENTER>	Start centered text
</CENTER>	End centered text
 	Line Break
<P>	Paragraph Break

Panorama has special functions that allow you to extract the contents of one or more tags from a body of text. While you could do this with regular Panorama functions, the tag parsing functions are much faster.

As you can see, each tag begins with a < and ends with a >. This is called the **tag header** and the **tag trailer**. Panorama's tag parsing functions allow you to specify what character is used for the header or trailer, so you could use these functions with other languages besides HTML.

The tag parsing functions also allow you to use multiple characters as a tag header and tag trailer. For example, using the tag header Š and tag trailer you can easily locate all bold text within an HTML document.

See Also:

- [tagarray\(](#) function
- [tagcount\(](#) function
- [tagdata\(](#) function
- [tagstart\(](#) function
- [tagend\(](#) function
- [tagnumber\(](#) function
- [tagparameter\(](#) function
- [tagparameterarray\(](#) function

HTMLDECODE(...)

Syntax: HTMLDECODE()

Description: The `htmlencode()` function takes HTML text and converts any special characters in the text into standard ASCII. For example `©` is converted to © and `&` is converted to &.

Parameters: This function has one parameter: `text`.
`text` is the ASCII text you want to convert from HTML format.

Result: This function returns regular ASCII text. Any special characters in the text are converted from their HTML equivalents into regular ASCII characters. A complete list of HTML equivalents is listed below.

Character	HTML Equivalent
&	&
¶	¶
ï	&luml;
	è
	
•	·
	Ñ
	é
`	¡
	¸
æ	Ò
	ê
ç	¢
°	º
	Ó
	ë
£	£
»	»
	Ô
	ì
	¤
	¿
¨	Õ
	í

Character	HTML Equivalent
•	¥
◌	À
	Ö
	î
	¦
	Á
	Ù
	ï
/	§
	Â
	Ú
	ñ
<	¨
	Ã
	Û
	ò
'	©
	Ä
	Ü
	ó
»	ª
	Å
	&zslig;
	ô
«	«
fi	Æ
	à
	õ
¬	¬
	Ç
	á
	ö
—	­
Ø	È
	â
÷	÷

Character	HTML Equivalent
◊	®
	É
	ã
	ù
±	¯
	Ê
	ä
	ú
ß	°
Ł	Ë
	å
	û
±	±
	Ì
	æ
	ü
ˆ	´
Ɔ	Í
	ç
	ÿ
	µ
◊	Î

Examples:

The example below imports the first table in the file Table.html. It then loops through each field, converting any special characters that were in the table into regular ASCII format.

```

openfile "Table.html"
loop
formulafill htmldecode(«»)
right
until stopped

```

Errors:

Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value for the text parameter.

See Also:

[htmlencode\(\)](#) function

HTMLENCODE(...)

Syntax: HTMLENCODE(text)

Description: The `htmlencode()` function takes standard [ascii](#) text and converts any special characters in the text into the HTML equivalents. For example © is converted to `©` and `&` is converted to `&`. Special characters that do not have HTML equivalents are removed. (However, the smart quote characters, “,”, ‘ and ’ are converted to regular quote characters " and '.)

Parameters: This function has one parameter: `text`.

`text` is the [ascii](#) text you want to convert to HTML format.

Result: This function returns regular ASCII text. Any special characters in the text are converted to their HTML equivalents. A complete list of HTML equivalents is listed in the [htmldecode\(\)](#) section.

Examples: The example below takes a price list and converts it into an HTML table ready to be pasted into a web authoring application. The `htmlencode` function is used to make sure that no illegal characters get into the web page.

```
local tablebody
arraybuild tablebody,¶,"",
  "<tr><td>"+Product+"</td><td align=right>"+
  str(Price)+"</td></tr>"
tablebody="<table>"+htmlencode(tablebody)+"</table>"
clipboard=tablebody
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text parameter.

See Also: [htmldecode\(\)](#) function

HUE(...)

Syntax: HUE(color)

Description: The `hue()` function extracts the hue intensity from a color. **Hue** specifies where this color falls in the spectrum. If you are familiar with the standard Apple color picker, the Hue would specify the angle of the color from the center of the wheel. This is a number from 0 to 65535.

Parameters: This function has one parameter: color.

color is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

Result: This function extracts the intensity of the hue of this color. This intensity is a number between 0 and 65535.

Examples: The example below calculates the hue intensity of the color (in percent, from 0 to 100%).

```
Color=hue(HighlightColor)*100/65535
```

For more examples of color, see [colors](#).

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the color parameter.

See Also: [rgb\(\)](#) function
[hsb\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[red\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement
[colors](#)

IF

Syntax: IF true-false test

Description: The if statement is needed to mark the beginning of an if construct.

Parameters: This statement has one parameter: true-false test.

true-false test may be one or more functions or equations which result in a true or a false condition. Multiple true-false tests must be separated by an **and** or an **or** operator. Grouping true-false tests inside parenthesis () will give those tests priority in the processing order when Panorama evaluates them.

Action: This statement is used to mark the beginning of an if construct. It evaluates true-false test and if it is **true** executes the PanTalk code between the if and the endif statements and then continues on after the endif statement. If the true-false test evaluates **false** Panorama skips the if construct and executes all code after the endif statement.

When the optional else statement is used and true-false test evaluates true all PanTalk code between the if and the else is executed and then control resumes after the endif statement. When true-false test evaluates **false** all PanTalk code between the else and endif is executed and then control resumes after the endif statement.

See the help files for and, else and endif for more information on if constructs.

All if statements **must** have a corresponding endif statement in the same procedure regardless of whether the if constructs are nested or not. Therefore, a procedure with, say, five if statements must have five endif statements.

All procedure statements after an endif statement will execute, regardless of which portion, if any, of the if construct executes.

Using the Check Procedure tool on a procedure that has an if statement with no corresponding endif statements will result in an **error alert**. Attempting to run a similar procedure will result in a general warning regarding the procedure which aborts its operation.

Examples: This section of PanTalk code calculates the field #Available and if it is less than the field Minimum the if construct will have Panorama beep and display a message before going back to the window Invoices>Data Entry, otherwise the procedure just goes to Invoices>Data Entry after the calculation.

```

...
...
window "Inventory"
<#Available> = <#Available> - NoOrdered
if <#Available> > <Minimum>
beep
  message "Your low on "+<Item>
endif
window "Invoice>Data Entry"
...
...

```

In this example the PanTalk code after the endif statement will only be executed if the variable Password is equal to the words: **Open Sesame**, otherwise the procedure displays a message regarding an incorrect password and stops.

```

local Password
gettext "Enter Password:",Password
if Password ≠ "Open Sesame"
    message "Your password is incorrect."
    stop
endif
goform "Secret Stuff"
beep
message "You're in."

```

In this example the `info("empty")` is the true/false test; if it is true the message is displayed before going to the Sales Rep field, if it is false everything between the `else` and the `endif` statements is executed before going to Sales Rep.

```

select «#Sold» ≥ 0 and «Sale Date» ≥ month1st( today())
if info("empty")
    message "No sales records for this month."
else
    field «#Sold»
    total
    field Bonus
    formulafill ?( «$Sold» > 500.00,«$Sold»*.05,0)
endif
field «Sales Rep»
...
...
...

```

In this example there are two nested if constructs both which have a corresponding `endif` statement. Only if both if statements test true will the record be marked **Closed**. Notice the indenting of code within the if constructs, this is not required, but is helpful when trying to follow the PanTalk code logically through.

```

selectall
firstrecord
field Status
loop
    if Status = "New"
        if «Final Payment» > 0
            «Date Closed» = today()
            «Status» = "Closed"
        endif
    endif
downrecord
until info("eof")

```

This example demonstrates how to use the error flag in an if statement to perform error checking in a procedure. Note: error checking is not available for all procedure statements.

```

openfile "ListFile"
if error
    openfile dialog
endif

```

Views: This statement may be used in any view.

See Also: [?\(function](#)
[case statement](#)
[defaultcase statement](#)
[else statement](#)
[endcase statement](#)
[endif statement](#)

IMAGEQUALITY

Syntax: `IMAGEQUALITY quality`

Description: The `imagequality` statement specifies the image quality (and compression ratio) to be used when generating JPEG images with the `convertimage` statement. This statement requires the optional Enhanced Image Pack, which must be purchased separately.

Parameters: This statement has one parameter: `quality`.

quality is the image quality factor, a number from 0 (very low quality, high compression) to 100 (high quality, least compression).

Action: This statement must be placed just before the `convertimage` statement. When the `convertimage` statement is used to convert an image to JPEG format the `imagequality` statement controls the quality and level of compression.

Examples: The example procedure below creates two JPEG images from a TIFF original, one low quality and one high quality. Each is placed in a different subfolder of the current database folder.

```
imagequality 80
convertimage "Sunset.tif",":Hi Quality:Sunset.jpg",",",0,0
imagequality 20
convertimage "Sunset.tif",":Lo Quality:Sunset.jpg",",",0,0
```

Views: This statement may be used in any view.

See Also: [convertimage](#) function

IMPORT(...)

Syntax: IMPORT()

Description: The `import()` function returns a line of imported data. This function only works in conjunction with the [importusing](#) (see also [importcell\(\)](#) and [arrayfilter](#) statements).

Parameters: This function has no parameters.

Result: The `import()` function always returns a text type data item. When it is used with the [importusing](#) statement, the `import()` function returns the contents of the line that is currently being imported. Using this function you can process and re-arrange the data as it is being imported.

When it is used with the [arrayfilter](#) statement, the `import()` function returns the individual array element currently being processed. Using this function you can process the data in each array element.

When it is used at any other time, the `import()` function returns empty text

Examples: Suppose you have a text file named `MainframeDownload` that you want to load into a Panorama database. This text file contains four fields, but these are fixed length fields padded by spaces instead of being separated by tabs or commas. The first field is 20 characters long, the 2nd field is 5 characters long, the third is 2 characters long, and the fourth 25 characters long. The procedure below will import this data into the first 4 fields of the current database.

```
importusing
strip(import())[1,20])+--+
strip(import())[21,25])+--+
strip(import())[25,26])+--+
strip(import())[27,51])
openfile "&MainframeDownload"
```

Here is a procedure that uses `import()` with the [arrayfilter](#) statement. This procedure will build a text array of elements with parenthesis around each element: (Hydrogen);(Helium);(Lithium); etc.

```
local Elements
Elements="Hydrogen;Helium;Lithium;Beryllium;Boron;"+
"Carbon;Nitrogen;Oxygen;Fluorine;Neon;"+
...
"Mendelevium;Nobelium;Lawrencium"
arrayfilter Elements,Elements,";", "("+import()+")"
```

Errors: This function does not produce any errors.

See Also: [importusing](#) statement
[arrayfilter](#) statement
[importcell\(\)](#) function

IMPORTCELL(...)

Syntax: IMPORTCELL(columnNumber)

Description: The `importcell()` function returns one cell of imported data. This function only works in conjunction with the `importusing` statement (see also `import()` function, which returns an entire line of data).

Parameters: This function has one parameter: `columnNumber`.

Result: `columnNumber` is the column of data from the imported text that you want to return. The text being imported is separated into columns by either tabs or commas. The first column is column 0, the next is column 1, etc.

The `import()` function always returns a text type data item. When it is used with the `importusing` statement, the `importcell()` function returns the contents of the specified column from the line that is currently being imported. If the text being imported is comma delimited, the `importcell()` function will strip off any quotes around the data before returning it. Using this function you can process and re-arrange the data as it is being imported.

When it is used at any other time, the `importcell()` function returns empty text. It will also return empty text if you specify a column number that does not exist in the text being imported.

Examples: Suppose you have a text file named Sam's Contacts that contains data like this:

```
Smith,John,World Widgets,124 W. Olive St,San Jose,CA,95134
Lee,Susan,Industrial Metals,2347 N. Riverside,Cambridge,MA,02139
Marklee,Lance,Zipper Technologies,687 E. Dorothy Lane,Bothell,WA,98011
Anders,Fred,Acme Fireworks,5672 Lakewood Drive,Salinas,CA,93908
```

You want to import this data into a database that contains these fields:

```
Company, First Name, Last Name, Title, Address, City, State, Zip
```

Here's a procedure that will append the data in Sam's Contacts into the current database. The tabs (`\t`) in the formula divide the output into separate columns again so it can be imported.

```
importusing
importcell(2)+\t+
importcell(1)+\t+
importcell(0)+\t+
\t+
importcell(3)+\t+
importcell(4)+\t+
importcell(5)+\t+
importcell(6)
openfile "+Sam's Contacts"
```

The formula re-arranges the incoming data so that third column in the input text goes into the first field, the 2nd column goes into the 2nd field, the first column goes into the 3rd field, the 4th field is empty, the 4th column goes into the 5th field, the 5th column goes into the 6th field, the 6th goes into the 7th field and the 7th column goes into the 8th field.

In this example each column in the input corresponds with one field in the final database. However, you could split up a column into multiple fields, or combine multiple columns in the input text into a single field in the final database. For example, here is a procedure that imports Sam's Contacts into a database with the fields [Name](#), [Address](#), [City](#), [State](#) and [Zip](#).

```
importusing  
  importcell(1)+" "+importcell(0)+--+  
  importcell(3)+--+  
  importcell(4)+--+  
  importcell(5)+--+  
  importcell(6)  
openfile "+Sam's Contacts"
```

This formula simply concatenates the first and last names with a space, but you can use any function you want, including the [?\(\)](#), [sandwich\(\)](#), [upper\(\)](#), [lower\(\)](#), even [lookup\(\)](#) functions.

Errors: This function does not produce any errors.

See Also: [importusing](#) statement
[arrayfilter](#) statement
[import\(\)](#) function

IMPORTUSING

Syntax: IMPORTUSING formula

Description: The **importusing** statement processes and re-arranges data as it is imported from a text file. This statement is designed to be used together with the [openfile](#) statement.

Parameters: This statement has one parameter: formula.

formula is the formula used to process the import data. The **importusing** statement works by “inserting” this formula into the middle of the import process. The formula must be designed to take in a line of text and transform it into a different line of text. Panorama uses this formula to transform each raw line of the import data into a new, manipulated line. Panorama then imports this new manipulated line into the database instead of the original line.

Panorama has a two functions that allows the import translation formula to access the line that has been read from the disk: [import\(\)](#) and [importcell\(\)](#). The [import\(\)](#) function returns the entire line that has been imported. The [importcell\(\)](#) function has one parameter that specifies the number of the cell you want, for example [importcell\(1\)](#) or [importcell\(14\)](#).

Action: This statement is designed to be used in combination with the [openfile](#) statement to manipulate and preprocess data as it is imported. Because a formula is used to perform the manipulation this process is very flexible: you can re-arrange fields, convert lower to upper case (or the reverse), extract variable length data from fixed length fields, etc. The **importusing** statement should be placed directly before the [openfile](#) statement. (Note: The **importusing** statement only works when importing TEXT files. It cannot be used when importing from OverVUE or other Panorama files.)

Examples: Suppose you have a text file named Sam’s Contacts that contains data like this:

```
Smith,John,World Widgets,124 W. Olive St,San Jose,CA,95134
Lee,Susan,Industrial Metals,2347 N. Riverside,Cambridge,MA,02139
Marklee,Lance,Zipper Technologies,687 E. Dorothy Lane,Bothell,WA,98011
Anders,Fred,Acme Fireworks,5672 Lakewood Drive,Salinas,CA,93908
```

You want to import this data into a database that contains these fields:

```
Company, First Name, Last Name, Title, Address, City, State, Zip
```

Here’s a procedure that will append the data in Sam’s Contacts into the current database. The tabs (↵) in the formula divide the output into separate columns again so it can be imported.

```
importusing
  importcell(2)+↵+
  importcell(1)+↵+
  importcell(0)+↵+
  ↵+
  importcell(3)+↵+
  importcell(4)+↵+
  importcell(5)+↵+
  importcell(6)
openfile "+Sam’s Contacts"
```

The formula re-arranges the incoming data so that third column in the input text goes into the first field, the 2nd column goes into the 2nd field, the first column goes into the 3rd field, the 4th field is empty, the 4th column goes into the 5th field, the 5th column goes into the 6th field, the 6th goes into the 7th field and the 7th column goes into the 8th field.

In this example each column in the input corresponds with one field in the final database. However, you could split up a column into multiple fields, or combine multiple columns in the input text into a single field in the final database. For example, here is a procedure that imports Sam's Contacts into a database with the fields Name, Address, City, State and Zip.

```
importusing
  importcell(1)+" "+importcell(0)+--+
  importcell(3)+--+
  importcell(4)+--+
  importcell(5)+--+
  importcell(6)
openfile "+Sam's Contacts"
```

This formula simply concatenates the first and last names with a space, but you can use any function you want, including the [?](#), [sandwich\(\)](#), [upper\(\)](#), [lower\(\)](#), even [lookup\(\)](#) functions.

Suppose you have a text file named [MainframeDownload](#) that you want to load into a Panorama database. This text file contains four fields, but these are fixed length fields padded by spaces instead of being separated by tabs or commas. The first field is 20 characters long, the 2nd field is 5 characters long, the third is 2 characters long, and the fourth 25 characters long. The procedure below will import this data into the first 4 fields of the current database.

```
importusing
  strip( import()[1,20])+--+
  strip( import()[21,25])+--+
  strip( import()[25,26])+--+
  strip( import()[27,51])
openfile "&MainfameDownload"
```

Views: This statement may be used in the Data Sheet and Form views only.

See Also: [openfile](#) statement
[import\(\)](#) function
[importcell\(\)](#) function

info("abort")

Description: The `info("abort")` function returns true if the user has pressed **Command-Period** (Macintosh) or **Control-Period** (Windows). You should use this procedure if you have used the `disableabort` statement and want to check for Command/Control-Period yourself.

Examples: This example shows how you can manage the handling of Command/Control-Period. If the user presses Command/Control-Period, this procedure will stop and ask the user if he or she wants to abort. If they press Yes, the loop will stop. If they press No, the loop will continue.

```
disableabort
loop
  if info("abort")
    alert 1014,"Abort?"
    if info("dialogtrigger")contains "yes"
      stoploopif 1=1
    endif
  endif
  call ProcessFile
while currentFile enableabort
```

The `info("abort")` function will only return true ONCE for each time Command/Control-Period is pressed. In the example above, this allows the loop to continue if the user press the No button, and then stop again if they press Command/Control-Period a second time. If you need to test `info("abort")` more than once for a single Command/Control-Period, you should copy `info("abort")` into a variable and then test the variable.

See Also: [disableabort](#) statement
[enableabort](#) statement

info("activesuperobject")

Description: The `info("activesuperobject")` function returns the name of the currently active text editor or word processor SuperObject, if any. If no such object is currently being edited, the function returns empty text ("").

Examples: This example insert the current date and time into whatever SuperObject is currently being edited. This procedure will work with both Text Editor and Word Processor SuperObjects. The first line of the procedure checks to make sure that there actually is an active SuperObject (i.e. something is really being edited at this time).

```
if info("activesuperobject")≠"  
activesuperobject "InsertText",  
    datepattern( today(), "mm/dd/yy")+"@"+  
    timepattern( now(), "hh:mm am/pm")  
endif
```

See Also: [objectinfo\(function](#)
[activesuperobject statement](#)
[superobject statement](#)

info("applemenufolder")

Description: The `info("applemenufolder")` function returns a binary data item that unambiguously describes the location of the Apple Menu folder. This function is only valid when used on MacOS computers. This folder id can be used in other functions and statements.

Examples: This procedure saves a copy of the current database in the Apple Menu folder so that it will appear in the Apple .

```
saveacopyas folderpath(info("applemenufolder")+info("databasename"))
```

See Also: [info\("systemfolder"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("bof")

Description: The `info("bof")` function returns true if the database is currently on the first visible record. (Note: "bof" stands for "beginning of file".)

Examples: This example loops until the database is at the very first record.

```
loop  
  stoploopif info("bof")  
  uprecord  
while forever
```

The example above illustrates `info("bof")`, but there is a much quicker way to get to the top of the database.

```
firstrecord
```

See Also: [info\("eof"\)](#) function
[uprecord](#) statement

info("buttonrectangle")

Description: The `info("buttonrectangle")` function returns a rectangle defining the edges of the button that was clicked on (needless to say, this function should be used in a procedure that is triggered by a button). The rectangle is in screen relative coordinates (use the [xytoxy\(\)](#) function to convert to window or form relative co-ordinates).

Examples: Here's an example that times how long it takes the user to move the mouse away from a button.

```
local startTime
startTime=now()
loop
  nop
while inrectangle( info("mouse"),info("buttonrectangle"))
message pattern( now()-startTime,"# second~")
```

As long as the mouse is over the button, the procedure will whirl around and around doing nothing (remember, the `nop` statement does no operation). When the mouse moves away from the button, the `inrectangle()` function goes false and the loop is broken.

See Also: [info\("mousedown"\)](#)function
[info\("click"\)](#) function
[info\("mouse"\)](#)function

info("change count")

Description: The `info("change count")` function returns the number of changes that were made the last time the `change` statement was used..

Examples: This procedure changes all Saturday's to Friday's. It then uses the `info("change count")` function to see if anything actually changed.

```
field Day
change "Saturday","Friday"
if info("change count")=0
    message "There were no appointments on Saturday."
endif
```

See Also: [change statement](#)

info("changes")

Description: The `info("changes")` function returns the number of changes that have been made to the current database since the last time it was saved.

Examples: This procedure checks to see if there have been any changes to the current database. If there have been, it saves the database, then continues with its regular job.

```
if info("changes")>0  
  save  
endif  
field State  
sortup  
field City  
sortupwithin
```

See Also: [save statement](#)

info("click")

Description: The `info("click")` function returns the location of the last mouse click in screen relative co-ordinates.

Examples: The example below selects the data cell(s) the user clicked on. The procedure uses the `inrectangle()` function to determine which object (if any) was clicked on. (Note: Presumably this procedure would be triggered by a push button which covers the data cell objects.)

```

local hitPt, hitField
hitPt=xytoxy(info("click"), "Screen", "Form")
selectobjects
inrectangle(hitPt, objectinfo("rectangle")) and
objectinfo("type") beginswith "Data Cell:"
objectnumber 1
hitField=objectinfo("type")[":", -1][-2, -1]
if hitField="" stop endif
field hitField
editcell

```

If the user did click on a data cell, the procedure activates the cell.

See Also: [info\("mouse"\)function](#)
[xytoxy\(\)](#) function

info("cursorrectangle")

Description: The `info("cursorrectangle")` function returns a rectangle defining the edges of the current data cell (if any). The rectangle is in screen relative coordinates (use the `xytoxy()` function to convert to window or form relative co-ordinates).

Examples: This example opens the window PopCell directly over the current data cell.

```
local wTop,wLeft
wTop=rtop(info("cursorrectangle"))
wLeft=rleft(info("cursorrectangle"))
setwindowrectanglee rectangleize(wTop,wLeft,200,350)
openform "PopCell"
```

See Also: [info\("buttonrectangle"\)](#) function
[xytoxy\(\)](#) function
[field](#) statement

info("databasefilename")

Description: The `info("databasefilename")` function returns the name of the file containing the current database. This function returns the complete file name, including the `.pan` extension (if any).

Examples: This procedure looks up checks to see if this database is ready to be copied to a Windows PC computer.

```
if info("databasefilename") endswith ".pan"  
    message "This database is ready for the PC!"  
endif
```

See Also: [info\("databasename"\)](#) function

info("databasename")

Description: The `info("databasename")` function returns the name of the current database.

Examples: This procedure looks up checks in the current database. The procedure will continue to work even if you rename the database or make a copy of it.

```
local CheckTo
CheckTo=""
gettext "List checks written to:",CheckTo
Checks=lookupall(
info("databasename"),
    Payee,
    CheckTo,
    «Check#»,",")
if Checks=""
    message "No checks written to "+CheckTo
else
    message "Checks written to "+CheckTo+": "+Checks
endif
```

See Also: [info\("windowname"\)](#) function

info("datatype")

Description: The `info("datatype")` function returns the data type of the current field. The function returns a number from 0 to 10:

Number	Type
0	Text
1	Compressed (Choice)
2	Compressed (Choice)
3	Picture
4	Date
5	Floating Point
6	Integer
7	Fixed 1 Digit (##)
8	Fixed 2 Digits (###)
9	Fixed 3 Digits (####)
10	Fixed 4 Digits (#####)

Examples: This procedure below displays a dialog to allow the user to enter a new value into the current field. The procedure adjusts automatically for numbers, dates and text fields.

```

local newValue
newValue=""
case info("datatype")≥5
  gettext "Enter a number",newValue
  newValue=val(newValue)
case info("datatype")=4
  gettext "Enter the date",newValue
  newValue=date(newValue)
defaultcase
  gettext "Enter new data",newValue
endcase
clipboard=newValue
pastecell

```

See Also: [datatype\(function](#)

info("desktopfolder")

Description: The `info("desktopfolder")` function returns a binary data item that unambiguously describes the location of the desktop folder. Any file saved in this folder will appear on the desktop. This function is only valid when used on MacOS computers. This folder id can be used in other functions and statements.

Examples: This procedure saves a copy of the current database in the desktop folder so that it will appear on the desktop.

```
saveacopyas folderpath(info("desktopfolder"))+info("databasename")
```

See Also: [info\("systemfolder"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("dialogtrigger")

Description: The `info("dialogtrigger")` function returns the name of the last button pressed in a dialog. (Note: This function does not work with standard system dialogs like the **Open** and **Save As** dialogs.)

Examples: This procedure lets the user select data with the find/select dialog. If the user presses Cancel the procedure stops, otherwise it calculates the total of the Amount field.

```
findselect  
if info("dialogtrigger") contains "Cancel"  
    stop  
endif  
field Amount  
total
```

See Also: [info\("trigger"\)](#) function

info("empty")

Description: The `info("empty")` function returns true or false depending on the result of the last select operation. If no records were selected the function will return true, otherwise it will return false.

Examples: The procedure below selects all records that are "Ready", whatever that means. If there are any ready records, the procedure prints them.

```
select Status="Ready"  
if info("empty")  
  message "Nothing ready today!"  
  stop  
endif  
print dialog  
field Status  
formulafill "Printed"
```

See Also: [info\("found"\)](#) function
[select](#) statement

info("eof")

Description: The `info("eof")` function returns true if the database is currently on the last visible record. (Note: "eof" stands for "end of file".)

Examples: This example loops until the database is at the very last record.

```
loop  
  stoploopif info("eof")  
  downrecord  
while forever
```

The example above illustrates `info("eof")`, but there is a much quicker way to get to the bottom of the database.

```
lastrecord
```

See Also: [info\("bof"\) function](#)
[downrecord statement](#)

info("error")

Description: The `info("error")` function can be used after an `if error` statement. It returns the error message that would have been displayed if the error had not been trapped by `if error`.

Examples: This example asks the user to enter a temperature. When the text the user enters is converted to a number with the `val()` function, there could be an error (for example if there are no digits in the number!). If this happens, the `if error` statement traps the error, then the message statement uses `info("error")` to display the error message. After the error message is displayed the program goes back and asks the user for a temperature again.

```
local sometext, Temperature
readTemp:
sometext=""
gettext "Temperature",sometext
Temperature=val(sometext)
if error
    message info("error")
    goto readTemp
endif
if Temperature>TodaysHigh
    TodaysHigh=Temperature
endif
if Temperature<TodaysHigh
    TodaysLow=Temperature
endif
```

See Also: [if statement](#)

info("expandable")

Description: The `info("expandable")` function checks to see if the current record is a collapsed summary record.

Examples: This example could be used as a formula for a Flash Art or Super Flash Art object. This object will display an arrow that shows whether this record is expandable or not, much like the outline view in the Finder.

```
?(info("expandable"),"right arrow","up arrow")
```

See Also: [info\("summary"\)](#) function
[expand](#) statement
[collapse](#) statement

info("fieldname")

Description: The `info("fieldname")` function returns the name of the current field.

Examples: The procedure below clear the whole column of the current field.

```
noyes "Clear out all data in the "+  
      info("fieldname")+ " column?"  
if clipboard() contains "yes"  
  formulafill ""  
endif
```

See Also: [info\("databasename"\)](#) function
[info\("windowname"\)](#) function

info("files")

Description: The `info("files")` function builds a carriage return separated text array containing a list of all the currently open database files.

Examples: The example below uses `info("files")` to check to see if the Price List database is open. If not, the procedure opens the database.

```
if 0 = arraysearch(info("files"),"Price List",1,1)
  openfile "Price List"
endif
```

See Also: [listwindows](#)(function
[info\("windows"\)](#) function
[info\("volumes"\)](#) function
[dbinfo](#)(function
[openfile](#) statement

info("filevariables")

Description: The `info("filevariables")` function builds a carriage return separated text array containing a list of the currently allocated fileglobal variables in the current database.

Examples: The example below uses `info("filevariables")` to check to see if a fileglobal variable `SalesTax` exists. If it does not, then it displays a message that database doesn't support sales tax - if it does, it sets the `SalesTax` to 7.75.

```
if 0 = arraysearch(info("filevariables"),"SalesTax",1,1)
  beep
  message "This database does not support Sales Tax"
  stop
endif
SalesTax=7.75
```

See Also: [info\("globalvariables"\)](#)
[info\("localvariables"\)](#)
[info\("windowvariables"\)](#)
[fileglobal](#) statement

info("formcolor")

Description: The `info("formcolor")` function returns the background color of the current form. If the current window does not contain a form, the function will return empty text ("").

Examples: The procedure below checks to see if the current form has a white background.

```
if info("formcolor")=rgb(65535,65535,65535)  
    message "This form has a white background"  
endif
```

See Also: [formcolor](#) statement
[rgb\(\)](#) function
[hsb\(\)](#) function

info("formcomment")

Description: The `info("formcomment")` function returns the form comment that has been set up for the currently open form (if any).

Examples: This example assumes that the formcomment has been set up with instructions for sorting when this form is printed. For example, the form comment might look something like this:

```
Sort:Zip Code;
```

This procedure will sort the database as instructed, then print.

```
local sortThis,x
x=search(info("formcomment"),"Sort:")
if x>0
  sortThis=info("formcomment")[x+4,-1]
  sortThis=sortThis[1,";"][1,-2]
  field (sortThis)
  sortup
endif
print dialog
```

See Also: [print statement](#)
[formcomment\(function](#)

info("formname")

Description: The `info("formname")` function returns the name of the current form. If the current window does not contain a form, the function will return empty text ("")

Examples: The procedure below checks to see if the List form is the current window. If it is, the Detail form is opened.

```
if info("formname")="List"  
    openform "Detail"  
endif
```

See Also: [info\("windowtype"\)](#) function
[info\("windowname"\)](#) function

info("found")

Description: The `info("found")` function returns true or false depending on the result of the last find or next statement

Examples: The procedure below looks for Acme Widgets. If it finds this company, it increases the quantity in this record by one.

```
find Company="Acme Widgets"  
if info("found")  
    Quantity=Quantity+1  
endif
```

See Also: [info\("empty"\)](#) function
[find](#) statement
[next](#) statement

info("freememory")

Description: The `info("freememory")` function calculates how much free database memory is available (this does not include scratch memory).

Examples: This procedure checks in advance to see if a database may be loaded.

```
if info("freememory") < filesize("", "Big File")
  message "Sorry, not enough memory to load database!"
else
  openfile "Big File"
endif
```

See Also: [info\("scratchmemory"\)](#) function

info("globalvariables")

Description: The `info("globalvariables")` function builds a carriage return separated text array containing a list of the currently allocated global [and permanent variables if you are in author mode] variables.

Examples: The example below uses `info("globalvariables")` to check to see if the global variable «gSalesDbName» exists. If it does not, then it issues a message to open a database that creates the variable; for example in an .Initialize procedure.

```
if 0 = arraysearch(info("globalvariables"), "gSalesDbName", 1, ¶)
  beep
  message "Sales Database is not open"
  stop
endif
```

See Also: [info\("localvariables"\)](#)
[info\("filevariables"\)](#)
[info\("localvariables"\)](#)
[info\("windowvariables"\)](#)

info("imagepack")

Description: The `info("imagepack")` function allows a formula to make choices based on whether or not the Enhanced Image Pack is installed.

Examples: The example below builds a list of image files in the current folder. If the image pack is not installed only PICT files are included. If the image pack is installed then JPEG and TIFF files will also be listed.

```
local imageFiles
imageFiles=listfiles( dbinfo("folder",""),
  "PICT????"+?(info("imagepack"), "JPEG????TIFF????", ""))
```

See Also: [info\("serialnumber"\)](#) function

info("keyboard")

Description: The `info("keyboard")` function returns the last key that was pressed.

Examples: The example below checks to see if the last key pressed was a tab:

```
if info("keyboard") = chr(9)
```

This example checks to see if the last key pressed was an upper case A:

```
if info("keyboard") = "A"
```

See Also: [info\("keycode"\)](#) function
[info\("trigger"\)](#) function

info("keycode")

Description: The `info("keycode")` function returns a special numeric code that represents last key that was pressed. This code is unique for every key in the keyboard. For example, the `info("keycode")` will return a different value for the 1 key on the numeric keypad and the 1 key above the Q key. The following tables list the keycode values for each key (in hexadecimal).

Main Key-board:

A - 00
 B - 0B
 C - 08
 D - 02
 E - 0E
 F - 03
 G - 05
 H - 04
 I - 22
 J - 26
 K - 28
 L - 25
 M - 2E
 N - 2D
 O - 1F
 P - 23
 Q - 0C
 R - 0F
 S - 01
 T - 11
 U - 20
 V - 09
 W - 0D
 X - 07
 Y - 10
 Z - 06
 [- 21
] - 1E
 ; - 29
 ' - 27
 , - 2B
 . - 2F
 / - 2C
 SHIFT - 38
 CONTROL - 3B
 OPTION - 3A
 COMMAND - 37
 SPACE - 31
 \ - 32
 1 - 12
 2 - 13
 3 - 14
 4 - 15
 5 - 17

6 - 16
7 - 1A
8 - 1C
9 - 19
0 - 1D
- - 1B
= 18
DELETE - 33
` - 2A
ENTER - 24

Arrow Keys: UP ARROW - 7E
DOWN ARROW - 7D

LEFT ARROW - 7B
RIGHT ARROW - 7C

**Numeric Key-
pad:** 0 - 52
1 - 53
2 - 54
3 - 55
4 - 56
5 - 57
6 - 58
7 - 59
8 - 5B
9 - 5C
ENTER - 4C
CLEAR - 47
= - 51
/ - 4B
* - 43
+ - 4E
- - 45

**Function
Keys:** ESC - 35
F1 - 7A
F2 - 7B
F3 - 63
F4 - 76
F5 - 60
F6 - 61
F7 - 62
F8 - 64
F9 - 65
F10 - 6D
F11 - 67
F12 - 6F
F13 - 69
F14 - 6B

F15 - 71
HELP - 72
HOME - 73
PAGE UP - 74
PAGE DN - 79
DEL - 75
END - 77

Examples:

The example below checks to see if the last key pressed was the 0 on the numeric keypad:

```
if info("keycode") = radix(16,"52")
```

This example checks to see if the last key pressed was the F1 key:

```
if info("keycode") = radix(16,"7A")
```

See Also:

[info\("keyboard"\)](#) function
[info\("trigger"\)](#) function

info("localvariables")

Description: The `info("localvariables")` function builds a carriage return separated text array containing a list of the currently allocated local variables.

Examples: The example below uses `info("localvariables")` to check to see if the local variable «SalesDbName» exists. If it does not, then it issues a message to open a database.

```
if 0 = arraysearch(info("localvariables"), "SalesDbName", 1, ¶)
    beep
    message "Sales Database is not open"
    stop
endif
```

See Also: [info\("globalvariables"\)](#)
[info\("filevariables"\)](#)
[info\("windowvariables"\)](#)

info("magicwindow")

Description: The `info("magicwindow")` function returns the name of the currently designated "magic window," if any.

Examples: This procedure re-fills the "People List" list in the List form in the [Contacts](#) database. The form must be open (this is checked by the `info("magicwindow")` function).

```
magicwindow "Contacts:List"  
if info("magicwindow")<>" "  
    superobject "People List","Fill List"  
    magicwindow " "  
endif
```

See Also: [magicwindow](#) statement
[magicformwindow](#) statement

info("matrixcell")

Description: The `info("matrixcell")` function is designed to be used with matrix SuperObjects™. The function returns the cell number within the matrix, starting with 1 in the upper left hand corner. If the matrix order is horizontal, then the cell numbers will be consecutively numbered from left to right in each row. If the matrix order is vertical, then the cell numbers will be consecutively numbered from top to bottom in each column.

Examples: The illustration shows two matrixes with the cell number displayed in the upper right hand corner. The matrix on the left has vertical cell order, the matrix on the right has horizontal cell order

vertical cell order			horizontal cell order		
1	6	11	1	2	3
2	7	12	4	5	6
3	8	13	7	8	9
4	9	14	10	11	12
5	10	15	13	14	15

The cell numbers were displayed using a Text Display SuperObject™.

See Also: [info\("matrixrow"\)](#) function
[info\("matrixcolumn"\)](#) function

info("matrixcolumn")

Description: The `info("matrixcolumn")` function is designed to be used with matrix SuperObjects™. The function returns the column number, starting with 1 for the left hand column and increasing by one for each column to the right.

Examples: The illustration shows the columns and rows for a matrix SuperObject.

		--- columns ---		
		1	2	3
--- rows ---	1			
	2			
	3			
	4			
	5			

See Also: [info\("matrixrow"\)](#) function
[info\("matrixcell"\)](#) function

info("matrixname")

Description: The `info("matrixname")` function is designed to be used with matrix SuperObjects™. The function returns the name of the matrix object, if any. This function allows a single matrix frame to be used with two or more matrixes. The display for each matrix can be adjusted based on the matrix name.

Examples: The formula below can be used to display items from a catalog in several matrixes. Each matrix will display a different category, depending on what the name of the matrix is.

```
array(  
  lookupall(  
    "Catalog",  
    "Category",  
    info("matrixname"),  
    "Item",  
    ¶),  
  info("matrixcell"),  
  ¶  
)
```

See Also: [super matrix programming](#)

info("matrixrow")

Description: The `info("matrixrow")` function is designed to be used with matrix SuperObjects™. The function returns the row number, starting with 1 for the top and increasing by one for each row as you go down.

Examples: The illustration shows the columns and rows for a matrix SuperObject.

		--- columns ---		
		1	2	3
--- rows ---	1			
	2			
	3			
	4			
	5			

See Also: [info\("matrixcolumn"\)](#) function
[info\("matrixcell"\)](#) function

info("maximumwindow")

Description: The `info("maximumwindow")` function returns the largest possible rectangle for this window. This function is designed for use with an Auto Grow SuperObject for this window and will return the maximum size allowed by this object (see "[Maximum Window Size](#)" on page 1047).

Examples: This procedure enlarges the current window to it's largest possible size. If possible, the window will not be moved as it is enlarged.

```
local newWinBox,oldWinBox
newWinBox=info("maximumwindow")
oldWinBox=info("windowrectangle")
fitwindow
zoomwindow
  rtop(oldWinBox),
  rleft(oldWinBox),
  rtop(oldWinBox)+rheight(newWinBox),
  rleft(oldWinBox)+rwidth(newWinBox), " "
```

See Also: [info\("minimumwindow"\)](#) function
[setwindowrectangle](#) statement
[zoomwindow](#) statement

info("minimumwindow")

Description: The `info("minimumwindow")` function returns the smallest possible rectangle for this window. This function is designed for use with an Auto Grow SuperObject for this window and will return the maximum size allowed by this object (see "[Maximum Window Size](#)" on page 1047).

Examples: This procedure reduces the current window to its smallest possible size, without moving the window.

```
local newWinBox,oldWinBox
newWinBox=info("minimumwindow")
oldWinBox=info("windowrectangle")
zoomwindow
  rtop(oldWinBox),
  rleft(oldWinBox),
  rtop(oldWinBox)+rheight(newWinBox),
  rleft(oldWinBox)+rwidth(newWinBox), ""
```

See Also: [info\("maximumwindow"\)](#) function
[setwindowrectangle](#) statement
[zoomwindow](#) statement

info("modifiers")

Description: The `info("modifiers")` function returns the status of the modifier keys.

```
"shift"  
"capslock"  
"option"    (returned when Alt key pressed on PC)  
"command"   (returned when Control key pressed on PC)  
"control"   (returned when right mouse button pressed on PC)
```

The `info("modifiers")` function also returns the status of the last mouse click. If the last mouse click was a double click, `info("modifiers")` will return `"doubleclick"`. If the last mouse click was a `triple click`, `info("modifiers")` will return `"tripleclick"`.

If more than one modifier key is active the function will return all of them strung together like this:

```
"shift option"
```

You should check for a specific modifier with the `contains` operator.

Examples: This example opens the **Status** form if the user double clicks on a button.

```
if info("modifiers") contains "double"  
    openform "Status"  
endif
```

See Also: [info\("click"\)](#) function
[info\("mouse"\)](#) function
[info\("mousedown"\)](#) function
[info\("mousetilldown"\)](#) function
[key](#) statement

info("mouse")

Description: The `info("mouse")` function returns the current location of the mouse in screen relative co-ordinates.

Examples: The example below calculates the distance (in inches) and angle between the current mouse position and the point where the mouse was last clicked.

```
local dragDistance, dragAngle, mousePoint, deltaV, deltaH
mousePoint=info("mouse")
deltaV=v(mousePoint-v( info("click")))
deltaH=h(mousePoint-h( info("click")))
dragDistance=sqr(deltaV^2+deltaH^2)/72
dragAngle=tanh(deltaV/deltaH)*180/π
```

Notes: The second line copies the current mouse position into the variable `mousePoint`. This is necessary in case the mouse moves between the third and fourth lines of the procedure. The division by 72 at the end of the `dragDistance` calculation converts the distance to inches. The multiplication by $180/\pi$ in the angle calculation converts the angle from radians to degrees.

See Also: [info\("click"\)](#) function
[xytoxy\(\)](#) function

info("mousedown")

Description: The `info("mousedown")` function returns true or false depending on whether or not the mouse is currently down.

Examples: The example below calculates how long the mouse is held down after the user presses a button.

```
local downTime
downTime=now()
loop
  nop
while info("mousedown")
  downTime=now()-downTime
```

See Also: [info\("mousetiltdown"\)](#) function
[info\("click"\)](#) function
[info\("mouse"\)](#) function

info("mousetiltdown")

Description: The `info("mousetiltdown")` function returns true or false depending on whether or not the mouse is currently down and has not been let up since the button was pressed.

Examples: The example below calculates how long the mouse is held down after the user presses a button.

```
local downTime
downTime=now()
loop
  nop
while info("mousetiltdown")
downTime=now()-downTime
```

See Also: [info\("mousedown"\)](#) function
[info\("click"\)](#) function
[info\("mouse"\)](#) function

info("multiuser")

Description: The `info("multiuser")` function returns true if the database is currently in multi-user mode. (Note: This function reflects the status of the obsolete pre-Panorama 3.0 multi-user system, not the Panorama 3.0 client/server system. See the [info\("serverstatus"\)](#) function.)

Examples: This example loops until the database is at the very first record.

```
if info("multiuser")
  message "Please switch to single user mode."
else
  field Date
  groupup by month
  field Amount
  total
endif
```

See Also: [info\("serverstatus"\)](#) function

info("openresourcefiles")

Description: The `info("openresourcefiles")` function returns a carriage return delimited list of all open resource files (if any).

Examples: The procedure below lists all open resource files.

```
message info("openresourcefiles")
```

See Also: [openresource](#) statement
[openresourcew](#) statement
[closeresource](#) statement

info("pagenumber")

Description: The `info("pagenumber")` function returns the current report page number. This function is designed to be used as part of an auto-wrap text object or Text Display SuperObject™ in a report form.

Examples: This example formula could be used to print the page number with an auto-wrap text object or Text Display SuperObject™. This formula causes the printed page numbers to begin with page 25 instead of page 1.

```
24+info("pagenumber")
```

See Also: [print](#) statement

info("panoramabuild")

Description: The `info("panoramabuild")` function returns a SuperDate. This value contains the date and time when the currently running copy of Panorama was compiled. You can use this function to differentiate between two slightly different versions of Panorama even if they have the same version number.

Examples: The procedure below checks to make sure that it will run on the current version of Panorama.

```
if regulardate(info("panoramabuild"))<date("1/1/2001")
  message "This procedure requires Panorama 3"
endif
...
  (rest of procedure)
...
```

See Also: [info\("version"\)](#) function
[regulardate\(\)](#) function
[regulartime\(\)](#) function

info("panoramafolder")

Description: The `info("panoramafolder")` function returns a binary data item that unambiguously describes the location of the folder containing the currently running copy of Panorama. This folder id can be used in other functions and statements.

Examples: This procedure below checks to see if the optional zip code file is installed.

```
local inPanoramaFolder
inPanoramaFolder=listfiles(info("panoramafolder"), "")
if 0 = arraysearch(inPanoramaFolder, "Zip Magic! Data", 1, ¶)
    message "Sorry, Zip Code database is not installed."
endif
```

See Also: [info\("panoramatoolsfolder"\)](#) function
[info\("panoramaname"\)](#) function
[info\("version"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function
[listfiles\(\)](#) function

info("panoramaname")

Description: The `info("panoramaname")` function returns the name of the currently running copy of Panorama. In other words, if you have renamed your copy of Panorama this function will tell what the name is.

Examples: This procedure displays the name, version and folder of the currently running copy of Panorama. This can be useful if you have multiple copies of Panorama on your system, and want to know which one you are using.

```
message"You are using "+info("panoramaname")+  
"version "+info("version")+  
"in the "+pathstr(info("panoramafolder"))+" folder."
```

See Also: [info\("panoramafolder"\)](#) function
[info\("panoramatoolsfolder"\)](#) function
[info\("version"\)](#)function

info("panoramatoolsfolder")

Description: The `info("panoramtoolsfolder")` function returns the folder id for the Panorama Tools folder in the Prefs folder. If this folder is not found, it returns the folder id of the folder containing Panorama itself. Panorama uses this folder for extra files it needs to use (scripts, dictionaries, etc.)

Examples: This procedure saves a prefs file into the Panorama tools folder.

```
filesave info("panoramtoolsfolder"), "myPrefs", "", thePrefs
```

See Also: [info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#)function
[info\("panoramaname"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("plugandrun")

Description: The `info("plugandrun")` function tells how Panorama will resolve conflicts between the client and server when the client database is reconnected to the server after being used off line. There are four possible modes

off This is the value that will be returned if this is a single user Panorama database that is not linked to an SQL server database.

client This means that if a record has been modified by both the client and the server, the clients changes will be kept and the servers changes will be discarded.

server This means that if a record has been modified by both the client and the server, the servers changes will be kept and the clients changes will be discarded.

manual This means that if a record has been modified by both the client and the server, the user will be presented with a list of the changed record and allowed to "cherry pick" which records to keep.

Examples: This procedure below warns the user if the database is set up so that the server may override client changes.

```
if info("plugandrun") contains "server"  
    message "Warning: Your changes may be lost when you reconnect with the  
server!"  
endif
```

See Also: [setplugandrun](#) statement

info("preferencesfolder")

Description: The `info("preferencesfolder")` function returns a binary data item that unambiguously describes the location of the Preferences folder. This function is only valid when used on MacOS computers. This folder id can be used in other functions and statements.

Examples: This procedure loads preference data from a file named `MyAppPref.dat` in the Preferences folder.

```
local myAppPreferences  
myAppPreferences=fileload(info("preferencesfolder"), "MyAppPref.dat")
```

See Also: [info\("panoramatoolsfolder"\)](#) function
[info\("systemfolder"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("records")

Description: The `info("records")` function returns the total number of records in the current database. To find out the number of selected records, use [info\("selected"\)](#).

Examples: This example checks to see if all records are selected. If some records are not selected, the procedure does a [selectall](#) statement.

```
if info("selected")<info("records")  
  selectall  
endif
```

See Also: [info\("selected"\)](#) function
[select](#) statement

info("reportcolumns")

Description: The `info("reportcolumns")` function returns the number and direction of report columns that have been set up for the currently open form (if any). The function returns a text string that contains the number of columns followed by the direction, for example "3 Down" or "2 Across".

Examples: This procedure displays the number of columns in the current report.

```
message "This report has "+  
array(info("reportcolumns"),1,"")+ "columns."
```

See Also: [setreportcolumns](#) statement

info("rulers")

Description: The `info("rulers")` function returns the current measurement units for the rulers in this form. The function may return five possible values: Inches, Centimeters, Pixels, Deca-Pica, or Deca-Elite.

Examples: This procedure displays the dimensions of the logo in the current form. Depending on the ruler setting, the dimensions will be displayed in inches, centimeters or pixels.

```

local logoHeight,logoWidth,Units
object "Logo"
logoHeight=rheight( objectinfo("rectangle"))
logoWidth=rwidth( objectinfo("rectangle"))
Units=info("rulers")
if Units beginswith "D"
    Units="Pixels"
endif
if Units="Inches"
    logoHeight=logoHeight/72
    logoWidth=logoWidth/72
endif
if Units="Centimeters"
    logoHeight=logoHeight/28.346
    logoWidth=logoWidth/28.346
endif
message "The logo is "+
str(logoHeight)+" by "+
str(logoWidth)+" "+Units

```

See Also: [setrulers](#) statement

info("scratchmemory")

Description: The `info("scratchmemory")` function returns the amount of memory allocated for scratch memory.

Examples: The procedure below checks the to make sure that at least 350K of scratch memory is allocated.

```
if info("scratchmemory")<350000  
    scratchmemory 350000  
endif
```

See Also: [scratchmemory](#) statement
[info\("freememory"\)](#) function

info("screenrectangle")

Description: The `info("screenrectangle")` function returns a rectangle defining the edges of the main screen (the screen that contains the menu bar). The rectangle is in screen relative coordinates.

Examples: Here's a simple procedure that displays the size of your main screen:

```
local sHeight,sWidth
sHeight=rheight(info("screenrectangle"))
sWidth=rwidth(info("screenrectangle"))
message "Screen dimensions are: "+
str(sWidth)+" by "+str(sHeight)+ "pixels."
```

See Also: [info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function

info("selected")

Description: The `info("selected")` function returns the number of selected records in the current database. To find out the total number of records, use `info("records")`.

Examples: This example checks to see if all records are selected. If some records are not selected, the procedure does a `selectall` statement.

```
if info("selected")<info("records")  
    selectall  
endif
```

See Also: [info\("records"\) function](#)
[select statement](#)

info("serialname")

Description: The `info("serialname")` function returns the name of the person this copy of Panorama is registered to.

Examples: This procedure below displays the registration information for this copy of Panorama. This is the same information that you typed in when you first installed your copy of Panorama.

```
message
  "Panorama Serial# "+info("serialnumber")["-", "--"][2, -2]+
  " is registered to "+info("serialname")+
  ", "+info("serialorganization")+", "+info("serialphone")+"."
```

See Also: [info\("serialorganization"\)](#) function
[info\("serialnumber"\)](#) function
[info\("serialphone"\)](#) function
[info\("user"\)](#) function

info("serialnumber")

Description: The `info("serialnumber")` function returns the serial number of this copy of Panorama is registered to. The serial number is returned as a text item in this format:

10000.ABC

Examples: This procedure below displays the serial number.

```
message  
"Panorama Serial# "+info("serialnumber")
```

See Also: [info\("serialname"\)](#) function
[info\("serialorganization"\)](#) function
[info\("user"\)](#) function

info("serialorganization")

Description: The `info("serialorganization")` function returns the organization this copy of Panorama is registered to.

Examples: This procedure below displays the registration information for this copy of Panorama. This is the same information that you typed in when you first installed your copy of Panorama.

```
message
  "Panorama Serial# "+info("serialnumber")["-", "--"][2, -2]+
  " is registered to "+info("serialname")+
  ", "+info("serialorganization")+", "+info("serialphone")+"."
```

See Also: [info\("serialname"\)](#) function
[info\("serialnumber"\)](#) function
[info\("serialphone"\)](#)function
[info\("user"\)](#) function

info("serialphone")

Description: The info("serialphone") function returns the phone number of the person or organization this copy of Panorama is registered to.

Examples: This procedure below displays the registration information for this copy of Panorama. This is the same information that you typed in when you first installed your copy of Panorama.

```
message
  "Panorama Serial# "+info("serialnumber")["-", "--"][2, -2]+
  " is registered to "+info("serialname")+
  ", "+info("serialorganization")+", "+info("serialphone")+"."
```

See Also: [info\("serialname"\)](#) function
[info\("serialnumber"\)](#) function
[info\("serialorganization"\)](#) function
[info\("user"\)](#) function

info("serverfile")

Description: The `info("serverfile")` function returns the name of the SQL database linked to this Panorama database, if any.

Examples: This procedure below checks two Panorama databases to see if they are linked to the same server database.

```
local sqlfile1,sqlfile2
sqlfile1=info("serverfile")
openfile "Another Database"
sqlfile2=info("serverfile")
if sqlfile1=sqlfile2
    message "These two Panorama databases "+
        "are linked to the same server database."
endif
```

See Also: [serverfile](#) statement

info("serverrecordid")

Description: The `info("serverrecordid")` function returns the internal serial number on the server of the current Panorama record. This number is guaranteed to be unique in this database if this is a SQL connected database. If this is a standalone database, this function will return zero for all records. This function will also return zero for new records created while disconnected to the server. These new records are not assigned an internal serial number until the next time the database is synchronized with the server.

Examples: This function is intended for debugging purposes only. It is included here for completeness.

See Also: [info\("serverrecords"\)](#) function

info("serverrecordts")

Description: The `info("serverrecordts")` function returns the internal "time stamp" number Panorama uses to determine which records need to be synchronized. By itself, this number is basically meaningless. However, if the time stamp for record A is higher than record B, then record A was edited later than record B. If this is a standalone database, this function will return zero for all records. This function will also return zero for new records created while disconnected to the server. These new records are not assigned an internal time stamp until the next time the database is synchronized with the server.

Examples: This function is intended for debugging purposes only. It is included here for completeness. However, it could possibly have useful non-debugging purposes. Here is an example that selects all records that have been modified after the current record. Remember, this will only work with an SQL database.

```
local ts
ts=info("serverrecordts")
select info("serverrecordts") > ts
```

See Also: [info\("serverrecordid"\)](#) function

info("serverstatus")

Description: The `info("serverstatus")` function returns the connection status of the SQL database linked to this Panorama database, if any. There are four possible values:

Standalone (Read/Write) This is the value that will be returned if this is a single user Panorama database that is not linked to an SQL server database.

Connected (Read/Write) This is the value that will be returned if this is a multi user Panorama database that is linked to an SQL server database, and the link is currently open with full record locking.

No Connection (Read/Only) This is the value that will be returned if this is a multi user Panorama database but there is currently no network connection to the SQL server database. For example the user might be using this database on a laptop computer with no connection to the server. This database does not allow off-line editing, so the database cannot be edited.

Standalone (Read/Write) This is the value that will be returned if this is a multi user Panorama database but there is currently no network connection to the SQL server database. For example the user might be using this database on a laptop computer with no connection to the server. This database does allow off-line editing, so the database may be edited. The changes made off-line will be saved and synchronized the next time the server is available.

Examples: This procedure below checks to make sure that the database is connected to the server, and then loads a subset of the server database into the local Panorama database.

```
if info("serverstatus") contains "Connected"  
    subsetselect Region="Western"  
else  
    message "Sorry, cannot update region now."  
endif
```

See Also: [attachserver](#) statement
[detachserver](#) statement

info("servertimeout")

Description: The `info("servertimeout")` function returns the maximum time Panorama will keep a record locked with no keyboard or mouse activity. This timeout can help prevent a user from starting to edit a record and then walking away from the computer and leaving the record locked and unavailable to other users indefinitely. The time interval is specified in seconds. A timeout value of zero indicates no timeout (infinite time).

Examples: This procedure below checks to see if the user has set a timeout value. If there is no timeout value, a default value of 3 minutes is set (180 seconds).

```
if info("servertimeout")=0
  servertimeout 180
endif
```

See Also: [servertimeout](#) statement
[lockrecord](#) statement
[lockorstop](#) statement
[unlockrecord](#) statement

info("startupfolder")

Description: The `info("startupfolder")` function returns a binary data item that unambiguously describes the location of the startup folder. Any file saved in this folder will be opened automatically when the system boots up. This function is only valid when used on MacOS computers. This folder id can be used in other functions and statements.

Examples: This procedure saves a copy of the current database in the startup folder so that it will open automatically when the computer is turned on.

```
savecopyas folderpath(info("startupfolder"))+info("databasename")
```

See Also: [info\("systemfolder"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("stopped")

Description: The `info("stopped")` function returns true or false depending on the result of the last [uprecord](#), [downrecord](#), [left](#) or [right](#) statement. If the statement could not move the active cell because the active cell was already as far as it could go, the function will return true. Otherwise it will return false.

Examples: The procedure below renames all the fields based on the data in the current record. The procedure starts at the leftmost column and works its way to the right until it reaches the last column, when `info("stopped")` will return true and stop the loop.

```
field (array( dbinfo("fields",""),0,1))  
loop  
  fieldname «»  
  right  
until info("stopped")
```

If there are no other elements to the formula you can simply use the word `stopped` with the `if` and `until` statements, for example `if stopped` or `until stopped`.

See Also: [info\("found"\)](#) function
[info\("bof"\)](#) function
[info\("eof"\)](#) function
[uprecord](#) statement
[downrecord](#) statement
[left](#) statement
[right](#) statement

info("subsetformula")

Description: The `info("subsetformula")` function returns the formula used to extract the current local subset from the server database. If the local database contains a copy of the entire server database (select all) the result will be an empty string ("").

Examples: This procedure below displays the region loaded as the current local subset. It assumes that the selection formula is something like `Region="Western"`, in which case it will display Western.

```
if info("subsetselect") beginswith "Region"
  message info("subsetselect")["},{-"}][2,-2]
endif
```

One common use for this function is to save the current selection formula, make another selection, and then go back to the original selection, like procedure which counts the number of records after 1/1/97 and then goes back to the original selection.

```
local wasSubset,count97
wasSubset=info("subsetselect")
subsetselect Date>date("Jan 1, 1997")
count97=info("records")
clipboard=wasSubset
subsetselect clipboard
```

Note: You must be very careful in this example to use the keyword `clipboard`, not `clipboard()`.

See Also: [subsetselect](#) statement
[subsetselectall](#) statement
[subsetselectdialog](#) statement
[subsetformulaselect](#) statement

info("summary")

Description: The `info("summary")` function returns the summary level of the current record, from 0 (data record) to 7.

Examples: The procedure below finds the first summary record in the database.

```
find info("summary")>0
```

See Also: [group](#) statement
[selectsummaries](#) statement
[summarylevel](#) statement

info("systemfolder")

Description: The `info("systemfolder")` function returns a binary data item that unambiguously describes the location of the system folder. This folder id can be used in other functions and statements.

Examples: This procedure below lists all the extensions in the current system folder.

```
local extensionFiles,extensionsFolder  
extensionsFolder=folder(  
folderpath(info("systemfolder"))+"Extensions")  
extensionFiles=listfiles(extensionsFolder,"")
```

See Also: [folder\(\)](#) function
[folderpath\(\)](#) function
[listfiles\(\)](#) function
[info\("panoramafolder"\)](#) function

info("tabdown")

Description: The `info("tabdown")` function returns true if the tab down option is on, false if the tab down option is off.

Examples: The procedure below turns off the tab down option.

```
if info("tabdown")  
    tabdown  
endif
```

See Also: [tabdown](#) statement

info("tempfolder")

Description: The `info("tempfolder")` function returns a binary data item that unambiguously describes the location of the system temporary file folder. This folder is a handy location for files that you want to create temporarily and then delete. This function is only valid when used on MacOS computers. This folder id can be used in other functions and statements.

Examples: This procedure exports a copy of the current database in the temporary folder.

```
local extensionFiles,extensionsFolder
extensionsFolder=folder(
folderpath(info("systemfolder"))+"Extensions")
extensionFiles=listfiles(extensionsFolder,"")
```

See Also: [info\("systemfolder"\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function

info("tickcount")

Description: The `info("tickcount")` function returns the number of ticks (1/60th second) since the system started up. This function can be used to time intervals smaller than 1 second.

Examples: This example calculates the time interval between now and the last time this procedure was triggered. The result is stored in a floating point field named `Interval`, and is accurate to 1/60th of a second.

```
fileglobal lastTicks
lastTicks=lastTicks
if error
  lastTicks=0
  Interval=0
  rtn
endif
Interval=(info("tickcount")-lastTicks)/60
lastTicks=info("tickcount")
```

See Also: [now\(\)](#) function

info("trigger")

Description: The `info("trigger")` function returns information about how the current procedure was triggered.

If the procedure was triggered by data entry this function will return the word `Key` followed by a period and then the key that actually triggered the procedure:

```
Key.Return
Key.Enter
Key.Tab
```

If the procedure was triggered by a button, the function will return the word `Button` followed by a period and then the name of the button, for example:

```
Button.Save
Button.Calculate Tax
Button.Show Chart
```

If the procedure was triggered by a custom menu, the function will return the word `Menu` followed by a period, the name of the menu, another period, and then the menu item. Here are some examples:

```
Menu.Accounting.Aging
Menu.Letter.New
```

Examples: The example below shows a `.CustomMenu` procedure written for two custom menus with two menu items apiece.

```
if info("trigger") = "Menu.Organize.SortByName"
  field FirstName sortupwithin
  stop
endif
if info("trigger") = "Menu.Organize.SortByZip"
  field Zip sortup
  stop
endif
if info("trigger") = "Menu.Transaction.Add"
  addrecord
  stop
endif
if info("trigger") = "Menu.Transaction.Void"
  Description="Void"
  Amount=0
  stop
endif
```

This example assumes that the database contains a field called `Carrier`, and a custom menu called `Airlines` that contains menu items listing airlines: `American`, `Delta`, `Southwest`, etc. When the user selects an airline the name of the airline is copied into the `Carrier` field.

```
local MenuName,MenuItemName
MenuName=info("trigger")[6,"-."][1,-2]
MenuItemName=info("trigger")["-.",-1][2,-1]
```

```
if MenuName="Airlines"  
    Carrier=MenuItemName  
    stop  
endif  
; other menu processing continues below
```

See Also:

[info\("keyboard"\)function](#)
[info\("keycode"\) function](#)
[info\("click"\) function](#)
[info\("mouse"\) function](#)
[info\("dialogtrigger"\) function](#)

info("typeofwindow")

Description: The `info("typeofwindow")` function determines what type of window the current window is. The window may be one of the types listed below:

```
Data Sheet  
Form (Data Mode)  
Draw (Graphics Mode)  
View As List  
Design Sheet  
Cross Tab Sheet  
Desk Accessory  
Floating Input Window  
Procedure  
Flash Art Gallery  
Clipboard  
Memory Usage  
Print Preview
```

Examples: This procedure checks to see if a form is currently on top. If so, the procedure opens the data sheet (or brings it to the front if it is already open).

```
if info("typeofwindow")="Form"  
    opensheet  
endif
```

See Also: [info\("windowname"\)](#) function
[info\("windowtype"\)](#) function
[window](#) statement
[opensheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[gosheet](#) statement
[goform](#) statement
[goprocedure](#) statement

info("user")

Description: The `info("user")` function returns the name of the user of this computer. If you are using System 6 you can set the name of the user with the Chooser. If you are using System 7 you can set the name of the user with the Sharing Setup control panel.

Examples: The procedure below will not work for anyone except Meg Reeves.

```
if info("user")≠"Meg Reeves"  
    stop  
endif  
...  
    (rest of procedure)  
...
```

See Also: [info\("username"\) function](#)
[info\("userid"\) function](#)
[info\("userlevel"\) function](#)
[info\("serialname"\) function](#)
[info\("serialorganization"\)function](#)

info("userid")

Description: The `info("userid")` function works with the Panorama security system. It returns the id (usually initials or the first name) the user has logged in under. If the user has not logged in, this function will return empty text ("").

Examples: The formula below could be used in a report header (in an auto-wrap text object or Text Display SuperObject™) to show who printed the report.

```
"Printed by: "+info("userid")+  
" @"+timepattern\( now\(\), "hh:mm am/pm" \)
```

See Also: [info\("username"\)](#) function
[info\("userlevel"\)](#) function

info("userlevel")

Description: The `info("userlevel")` function works with the Panorama security system. It returns the current user level for this user, a number from 0 to 255. If the user has not logged in, this function will return 0.

Examples: The procedure below only allows users with access levels of 25 or higher to use the rest of the procedure.

```
if info("userlevel")<25
    message "Sorry, your access level does not allow this operation"
    stop
endif
...
(rest of procedure)
...
```

See Also: [info\("username"\)](#) function
[info\("userid"\)](#) function

info("username")

Description: The `info("username")` function works with the Panorama security system. It returns the name the user has logged in under. If the user has not logged in, this function will return empty text ("").

Examples: The formula below could be used in a report header (in an auto-wrap text object or Text Display SuperObject™) to show who printed the report.

```
"Printed by: "+info("username")+  
" @"+timepattern( now(), "hh:mm am/pm")
```

See Also: [info\("userid"\)](#) function
[info\("userlevel"\)](#) function

info("version")

Description: The `info("version")` function returns the version of the currently running copy of Panorama.

Examples: The procedure below checks to make sure that it will run on the current version of Panorama.

```
if info("version")[1,1]<"3"  
    message "This procedure requires Panorama 3"  
endif  
...  
    (rest of procedure)  
...
```

See Also: [info\("panoramaname"\)](#) function
[info\("panoramafolder"\)](#) function
[info\("panoramabuild"\)](#) function

info("volumes")

Description: The `info("volumes")` function returns a carriage return separated array listing all of the currently mounted volumes (disks) on the computer. On MacOS computers the startup volume is always listed as the first element of this array.

Examples: This example checks to see if the Panorama Reference CD is currently mounted.

```
local mydisks
mydisks=info("volumes")
if mydisks notcontains "Panorama Reference"
    message "The Panorama Reference CD is not mounted!"
endif
```

See Also: [listfiles](#)(function
[folder](#)(function
[folderpath](#)(function
[dbinfo](#)(function
[info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#) function

info("windowbox")

Description: The `info("windowbox")` function returns text containing four numbers which define the edges of the current window. The rectangle is in screen relative coordinates.

Examples: This is a two part example. The first part is a `.CloseWindow` procedure that saves the dimensions of the Main form window when it is closed.

```
global MainWindowBox  
if info("formname")="Main"  
    MainWindowBox=info("windowbox")  
endif  
closewindow
```

The second half of the example is a procedure that can re-open the main window with the same position and size it had before it was closed.

```
windowbox MainWindowBox  
openform "Main"
```

With the introduction of Panorama 3 this function is somewhat obsolete—new applications will probably want to use the `info\("windowrectangle"\)` function instead.

See Also: [windowbox](#) statement
[info\("windowrectangle"\)](#) function
[info\("screenrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function

info("windowdepth")

Description: The `info("windowdepth")` function returns the pixel depth of the current window. This table shows the possible values:

Depth	Description
1	Black and White
2	4 colors
4	16 colors
8	256 colors
16	Thousands of Colors
32	Millions of Colors

If the window crosses over two monitors with different pixel depths, the `info("windowdepth")` function will return the lower value.

Examples: The formula below could be used in a Flash Art object. If this is a black and white monitor it displays the picture `bwSky`, otherwise it displays the picture `Sky`.

```
?(info("windowdepth")=1,"bwSky","Sky")
```

See Also: [info\("windowname"\)](#) function

info("windowoptions")

Description: The `info("windowoptions")` function returns the names of any currently enabled window options, if any. The available options are:

```
nopalette  
novertscroll  
nohorzscroll  
nodragbar
```

Examples: This procedure checks to see if the current window has a tool palette. If so, it simply deletes the current record. If there is no tool palette it asks for confirmation before deleting the record.

```
if info("windowoptions") notcontains "palette"  
  alert 1013,"Are you sure you want to delete?"  
  if info("dialogtrigger") contains "no" rtn endif  
endif  
deleterecord
```

See Also: [setwindow](#) statement
[setwindowrectangle](#) statement
[windowbox](#) statement
[zoomwindow](#) statement

info("windowname")

Description: The `info("windowname")` function returns the name of the current window.

Examples: This procedure switches to the Appointments database and selects today's appointments. When this is finished it switches back to the original window, right back where you started.

```
local wasWindow  
wasWindow=info("windowname")  
openfile "Appointments"  
select When=today()  
window wasWindow
```

See Also: [info\("windowtype"\)](#) function
[window](#) statement
[openfile](#) statement

info("windowrectangle")

Description: The `info("windowrectangle")` function returns a rectangle defining the edges of the current window. The rectangle is in screen relative coordinates

Examples: Here's an example that times how long it takes the user to move the mouse away from the current window.

```
local startTime
startTime=now()
loop
  nop
while inrectangle( info("mouse"),info("windowrectangle"))
message pattern( now()-startTime,"# second~")
```

As long as the mouse is over the window, the procedure will whirl around and around doing nothing (remember, the `nop` statement does no operation). When the mouse moves away from the window, the `inrectangle()` function goes false and the loop is broken.

See Also: [info\("screenrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function

info("windows")

Description: The `info("windows")` function builds a carriage return separated text array containing a list of all the currently open windows. The windows are listed in order from front to back.

Examples: The example below uses `info("windows")` to check to see if the Invoice:Status window is open. If not, the procedure opens the form.

```
if 0 = arraysearch(info("windows"),"Invoice:Status",1,1)
    setwindowrectangle rectangle(20,5,120,255)," "
    openform "Status"
endif
```

See Also: [listwindows\(function](#)
[dbinfo\(function](#)

info("windowtype")

Description: The `info("windowtype")` function determines what number type the current window is. The window number may be one of the listed below:

```
2 = Data Sheet
5 = Form (Data Mode)
6 = Draw (Graphics Mode)
15 = View As List
7 = Design Sheet
10 = Cross Tab Sheet
1 = Desk Accessory
3 = Floating Input Window
8 = Procedure
11 = Flash Art Gallery
13 = Clipboard
12 = Memory Usage
14 = Print Preview
```

Examples: This procedure checks to see if a form is currently on top. If so, the procedure opens the data sheet (or brings it to the front if it is already open).

```
if info("windowtype")= 5
    opensheet
endif
```

See Also: [info\("windowname"\)](#) function
[info\("typeofwindow"\)](#) function
[window](#) function
[opensheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[gosheet](#) statement
[goform](#) statement
[goprocedure](#) statement

info("windowvariables")

Description: The `info("windowvariables")` function builds a carriage return separated text array containing a list of the currently available window global variables (i.e. the [windowglobal](#) variables defined for the active window).

Examples: The example below uses `info("windowvariables")` to reset all of the window variables in the current window to empty.

```
local wVars,n,oneVar
wVars=info("windowvariables")
n=1
loop
  oneVar=array(wVars,n,1)
  stoploopif oneVar=""
  execute oneVar+"={ } showvariables "+oneVar
  n=n+1
while forever
```

See Also: [info\("globalvariables"\)](#)
[info\("localvariables"\)](#)
[info\("filevariables"\)](#)
[windowglobal](#) statement

info("windowview")

Description: The `info("windowview")` function determines what type of window the current window is. The window may be one of the types listed below:

```
"Data Sheet "  
"Floating Input Window"  
"Floating Input Window"  
"Form"  
"Draw"  
"Design Sheet "  
"Procedure"  
"Reserved"  
"Cross Tab Sheet "  
"Flash Art Gallery"  
"Memory Usage "  
"Clipboard"  
"Print Preview"  
"View As List Form"
```

Examples: This procedure checks to see if a form is currently on top. If so, the procedure switches to graphics mode so that the form can be edited.

```
if info("windowview") contains "Form"  
    graphicsmode  
endif
```

See Also: [info\("windowname"\)](#) function
[info\("windowtype"\)](#) function
[info\("typeofwindow"\)](#) function
[window](#) statement
[opensheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[gosheet](#) statement
[goform](#) statement
[goprocedure](#) statement

INFO(...)

Syntax: INFO(option)

Description: The `info()` function returns miscellaneous bits of information about the system, the current database, the current window, etc.

Parameters: This function has one parameter: `option`.
option is the type of information that you want to retrieve.

See Also:

- [info\("abort"\)](#) function
- [info\("activesuperobject"\)](#) function
- [info\("applemenufolder"\)](#) function
- [info\("bof"\)](#) function
- [info\("buttonrectangle"\)](#) function
- [info\("changecount"\)](#) function
- [info\("changes"\)](#) function
- [info\("click"\)](#) function
- [info\("cursorrectangle"\)](#) function
- [info\("databasefilename"\)](#) function
- [info\("databasename"\)](#) function
- [info\("datatype"\)](#) function
- [info\("desktopfolder"\)](#) function
- [info\("dialogtrigger"\)](#) function
- [info\("empty"\)](#) function
- [info\("eof"\)](#) function
- [info\("error"\)](#) function
- [info\("expandable"\)](#) function
- [info\("fieldname"\)](#) function
- [info\("files"\)](#) function
- [info\("filevariables"\)](#) function
- [info\("formcolor"\)](#) function
- [info\("formcomment"\)](#) function
- [info\("formname"\)](#) function
- [info\("found"\)](#) function
- [info\("freememory"\)](#) function
- [info\("globalvariables"\)](#) function
- [info\("keyboard"\)](#) function
- [info\("keycode"\)](#) function
- [info\("localvariables"\)](#) function
- [info\("magicwindow"\)](#) function
- [info\("matrixcell"\)](#) function
- [info\("matrixcolumn"\)](#) function
- [info\("matrixname"\)](#) function
- [info\("matrixrow"\)](#) function
- [info\("maximumwindow"\)](#) function
- [info\("minimumwindow"\)](#) function
- [info\("modifiers"\)](#) function
- [info\("mouse"\)](#) function
- [info\("mousedown"\)](#) function
- [info\("mousetilldown"\)](#) function
- [info\("multiuser"\)](#) function

[info\("openresourcefiles"\)](#) function
[info\("pagenumber"\)](#) function
[info\("panoramabuild"\)](#) function
[info\("panoramafolder"\)](#) function
[info\("panoramaname"\)](#) function
[info\("panoramatoolsfolder"\)](#) function
[info\("plugandrun"\)](#) function
[info\("preferencesfolder"\)](#) function
[info\("records"\)](#) function
[info\("reportcolumns"\)](#) function
[info\("rulers"\)](#) function
[info\("scratchmemory"\)](#) function
[info\("screenrectangle"\)](#) function
[info\("selected"\)](#) function
[info\("serialname"\)](#) function
[info\("serialnumber"\)](#) function
[info\("serialorganization"\)](#) function
[info\("serialphone"\)](#) function
[info\("serverfile"\)](#) function
[info\("serverrecordid"\)](#) function
[info\("serverrecordts"\)](#) function
[info\("serverstatus"\)](#) function
[info\("startupfolder"\)](#) function
[info\("stopped"\)](#) function
[info\("subsetformula"\)](#) function
[info\("summary"\)](#) function
[info\("systemfolder"\)](#) function
[info\("tabdown"\)](#) function
[info\("tempfolder"\)](#) function
[info\("trigger"\)](#) function
[info\("typeofwindow"\)](#) function
[info\("user"\)](#) function
[info\("userid"\)](#) function
[info\("userlevel"\)](#) function
[info\("username"\)](#) function
[info\("version"\)](#) function
[info\("volumes"\)](#) function
[info\("windowbox"\)](#) function
[info\("windowdepth"\)](#) function
[info\("windowname"\)](#) function
[info\("windowoptions"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("windows"\)](#) function
[info\("windowview"\)](#) function

INRECTANGLE(...)

Syntax: INRECTANGLE(point,rectangle)

Description: The `inrectangle()` function checks to see if a point is inside a rectangle (see [point\(\)](#), [rectangle\(\)](#)).

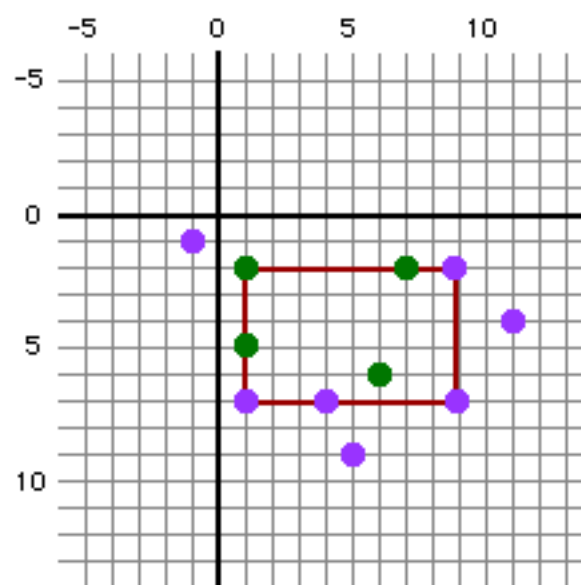
Parameters: This function has two parameters: `point` and `rectangle`. These parameters may be in screen, window, or form relative co-ordinates as long as you make sure both use the same co-ordinate system. All measurements are in pixels (1 pixel = 1/72 inch)

point is a point.

rectangle is a rectangle.

Result: This function returns a true or false result. If the point is inside the rectangle, the function returns true (-1). If the point is not inside the rectangle, the function returns false (0). You can use this function with the `if` statement and the `?(` function.

Examples: The illustration below shows a rectangle and several points. Green points are inside the rectangle, purple points are not. Notice that points on the top and left edge of the rectangle are considered inside. Points on the bottom and right edge are considered outside.



Here's an example that times how long it takes the user to move the mouse away from a button.

```

local startTime
startTime=now()
loop
  nop
while inRectangle(info("mouse"),info("buttonrectangle"))
message pattern( now()-startTime,"# second~")

```

As long as the mouse is over the button, the procedure will whirl around and around doing nothing (remember, the `nop` statement does no operation). When the mouse moves away from the button, the `inRectangle()` function goes false and the loop is broken.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the point parameter.

See Also:

[point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function
[rright\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

INSERTBELOW

Syntax: INSERTBELOW

Description: The `insertbelow` statement inserts a new record below the current record.

Parameters: This statement has no parameters.

Action: This statement inserts a new record just below the current record. It has the same effect as pressing the RETURN key in the data sheet.

Examples: This simple example inserts twelve new records into the middle of the current database.

```
loop  
    insertbelow  
until 12
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views.

See Also: [addrecord](#) statement
[insertrecord](#) statement
[returnkey](#) statement
[deleterecord](#) statement
[info\("records"\)](#) function

INSERTFIELD

Syntax: `INSERTFIELD name`

Description: The `insertfield` statement adds a new field to the current database.

Parameters: This statement has one parameter: name.

name is the name of the new field you want to create. If the parameter is the keyword `dialog` the procedure will stop and display the Field Properties dialog, allowing the user to set up the field name, type, and other properties of the new field.

Action: This statement inserts a new field into the database in front of the current field. (If you want to add a new field at the end of the database, use the [addfield](#) statement.) The new field is created as a text field. Use the `FieldType` statement if you need to convert the new field to a numeric, date, choice or picture field. (Note: the new field does not become the current field. Use the `field` statement to make the new field current before changing the type or performing other operations on the field.)

Examples: This simple example adds a new field called Organization in front of the Address field.

```
field Address
insertfield "Organization"
```

This example creates a new field called Ratio in front of the Balance field. The procedure converts the new field to floating point, and then calculates values for the new field.

```
field "Balance"
insertfield "Ratio"
field "Ratio"
fieldtype "float"
formulafill "Price/Cost"
```

Views: This statement may be used in the Data Sheet and Form views only.

See Also: [addfield](#) statement
[dbinfo\(\)](#) function
[deletefield](#) statement
[field](#) statement
[fieldname](#) statement
[fieldstyle\(\)](#) function
[fieldtype](#) statement
[info\("datatype"\)](#) function
[info\("fieldname"\)](#) function
[newgeneration](#) statement

INSERTRECORD

Syntax: INSERTRECORD

Description: The `insertrecord` statement inserts a new record above the current record.

Parameters: This statement has no parameters.

Action: This statement adds a new record above the current record. It has the same effect as choosing the **Insert New Record** tool from the tool palette.

Examples: This simple example inserts twelve new records into the middle of the current database.

```
loop  
    insertrecord  
until 12
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views.

See Also: [addrecord](#) statement
[insertbelow](#) statement
[returnkey](#) statement
[deleterecord](#) statement
[info\("records"\)](#) function

INT(...)

Syntax: INT(value)

Description: The `int()` function truncates a number to an integer. It always truncates towards $-\infty$. (If you want to truncate to 0 use the `fix()` function.)

Parameters: This function has one parameter: `value`.

value is the value you want to convert to an integer. You may use any numeric value, for example 1, 563.14, -2.5, or even π .

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This simple example calculates the temperature in whole degrees.

```
int(Temperature)
```

Temperature must contain a numeric value. The table below shows how the `fix()` and `int()` functions work with some typical values.

Value	<code>fix()</code>	<code>int()</code>
98.700	98	98
4.5640	4	4
-3.1400	-3	-4

Errors: Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value with this function, for example `int("34")`. If you have a number in a text item you must convert the text to a numeric value before taking the absolute value, for example `int(val("34"))`.

See Also: [fix\(\)](#) function

INTERSECTIONRECTANGLE(...)

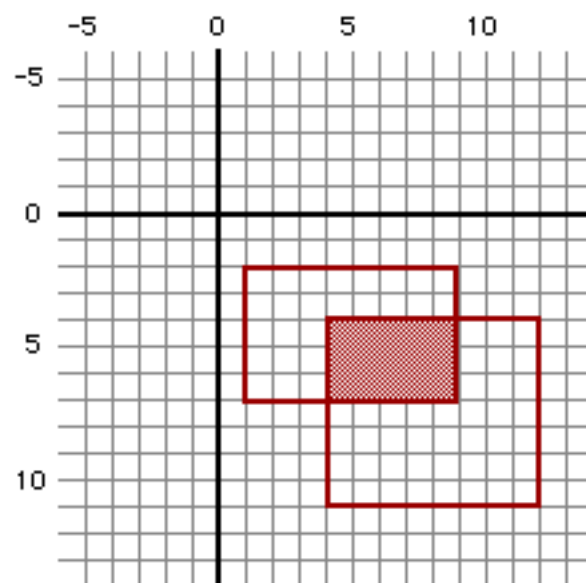
Syntax: INTERSECTIONRECTANGLE((rectangle1,rectangle2)

Description: The `intersectionrectangle()` function creates a rectangle by combining two rectangles. The new rectangle is the area where the two rectangles overlap (if any). A rectangle is 8 bytes or raw binary data (see [binary data](#), [graphic coordinates](#)).

Parameters: This function has two parameters: `rectangle1` and `rectangle2`.
rectangle1 is the first rectangle.
rectangle2 is the second rectangle.

Result: This function returns a rectangle with the area where the two rectangles overlap. If these two rectangles do not touch each other the function will return an empty rectangle (same as `rectangle(0,0,0,0)`).

Examples: The illustration below shows how this function combines two rectangles, creating a third rectangle where the original two rectangles overlap:



The `intersectionrectangle()` function can be used to check if two rectangles overlap each other. The procedure below checks to see if the current window is on the main screen, or if it is on another screen.

```
if intersectionrectangle(
    info("windowrectangle"),
    info("screenrectangle")) = rectangle(0,0,0,0)
    message "This window is completely off the main screen"
else
    message "This window is partially on the main screen"
endif
```

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for any of the four parameters.

See Also:

[point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function
[rright\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[unionrectangle\(\)](#) function
[inrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function










KEY

Syntax: KEY modifiers,keys

Description: The **key** statement allows a procedure to simulate one or more keystrokes. The keystrokes are not simulated immediately but are buffered until the end of the procedure. If you need the keystrokes to be processed immediately use the [key](#) statement.

Parameters: This statement has two parameters: modifiers and keys.

modifiers is a text item containing a list of the modifier keys (if any) for the simulated keystrokes. The modifier keys may be specified by any combination of the following words (separated by spaces, for example "shift option"):

Text	Equivalent Key (Macintosh)	Equivalent Key (Windows)
"shift"		
"capslock"		
"option"		
"command"		
"control"		none

keys is one or more characters corresponding to the keystrokes to be simulated. For example if you want to simulate the S key this parameter should be "S".

Action: This statement allows a procedure to simulate a keystroke.

Examples: This example simulates the key sequence Command-S Command-Q.

```
Key "command", "SQ"
```

The key statement is designed to be used with the **.KeyDown** procedure. If the **.KeyDown** procedure doesn't want to process a keystroke itself, it can pass the keystroke to the **key** statement for normal processing. This **.KeyDown** procedure performs special actions when the enter or \$ keys are pressed, but handles all other keys normally.

```
local KeyStroke
KeyStroke=info("trigger")[5,-1]
case KeyStroke=chr(3) /* enter key */
  closewindow
case KeyStroke="$"
  field Amount
  editcellstop
defaultcase
  key info("modifiers"),KeyStroke
endcase
```

Views: This statement may be used in any view.

See Also: [keynow](#) function
[info\("modifiers"\)](#) function
[info\("keyboard"\)](#) function
[info\("keycode"\)](#) function










KEYNOW

Syntax: KEYNOW modifiers,keys

Description: The **keynow** statement allows a procedure to simulate keystrokes. The keystrokes are processed immediately (not at the end of the procedure like the **key** statement).

Parameters: This statement has two parameters: modifiers and keys.

modifiers is a text item containing a list of the modifier keys (if any) for the simulated keystrokes. The modifier keys may be specified by any combination of the following words (separated by spaces, for example "shift option"):

Text	Equivalent Key (Macintosh)	Equivalent Key (Windows)
"shift"		
"capslock"		
"option"		
"command"		
"control"		none

keys is one or more characters corresponding to the keystrokes to be simulated. For example if you want to simulate the S key this parameter should be "S".

Action: This statement allows a procedure to simulate a keystroke.

Examples: This example simulates the key sequence Command-S Command-Q.

```
Key "command", "SQ"
```

This example uses types the phrase **type this now!** at 3 characters per second (1 tick equals 1/60th second).

```
local keyStrokes,n,c
keyStrokes="type this now!"
n=1
loop
  c=keyStrokes[n;1]
  stoploopif c=""
  keynow "",c
  t=info("tickcount")
  loop
    nop
  while t+20 > info("tickcount")
while forever
```

Don't use this statement with the **.KeyDown** procedure — doing so will disrupt Panorama's handling of menu keyboard equivalents (shortcuts).

Views: This statement may be used in any view.

See Also:

[key](#) statement
[info\("modifiers"\)](#) function
[info\("keyboard"\)](#) function
[info\("keycode"\)](#) function

LASTRECORD

Syntax: LASTRECORD

Description: The `lastrecord` statement moves the cursor to the last visible record in the active window. This is the opposite of the [firstrecord](#) statement.

Parameters: This statement has no parameters.

Action: This statement moves the cursor directly to the last visible record in the Data Sheet, Design Sheet, Cross Tab view, or View-as-list Form view. In a Individual Record Form view the view will change to the last record in the database. If the cursor is already on the last visible record this statement will do nothing.

This statement has the same effect as clicking on the **Last Record** tool on a tool palette (when available).

Examples: This simple example could be used in either the Data Sheet, Form view or Cross Tab view to move the cursor to the last visible record in the window, making this record the current record.

```
lastrecord
```

This example groups the database by the fields City and State, moves to the last visible record (the level 3 summary record) in the window, and deletes it.

```
field State
groupup
field City
groupup
lastrecord
cutrecord
```

This example moves the cursor to the last record in the Design Sheet window, which is the last field in the database, and changes it's width to 6.

```
opendesignsheet
lastrecord
Width = 6
newgeneration
closewindow
```

Views: This statement may be used in any view.

See Also: [firstrecord](#) statement
[downrecord](#) statement
[uprecord](#) statement

LEFT

Syntax: LEFT

Description: The **left** statement moves the cursor to the previous field in the active window. To decide what the previous field is, Panorama uses the data sheet order of the fields. This is the opposite of the [right](#) statement.

Parameters: This statement has no parameters.

Action: This statement moves the cursor to the previous field in the active window. If the current window is the data sheet the cursor will appear to move to the left one column. If the cursor is already on the first visible column this statement will do nothing.

You can use this statement in conjunction with the [info\("fieldname"\)](#) or [info\("stopped"\)](#) functions to test to see if you are on the last visible record in the window.

Examples: This example copies the data from the previous cell into the current cell.

```
left
if info\("stopped"\)
stop
endif
copycell
right
pastecell
```

Views: This statement may be used in any view.

See Also: [info\("fieldname"\)](#) function
[info\("stopped"\)](#) function
[dbinfo\(\)](#) function
[field](#) statement
[right](#) statement

LENGTH(...)

Syntax: LENGTH(text)

Description: The `length()` function calculates the length (number of characters) of an item of text.

Parameters: This function has one parameter: text.
text is the item of text that you want to know the length of.

Result: The result of this function is always a positive integer numeric value, for example 0, 1, 2, 3, etc.

Examples: This simple example calculates the length of the city name.

```
length(City)
```

City must contain a text item. The sample procedure below uses the `length()` function to check for possibly incorrect addresses.

```
select length(City)30
```

Very few valid city names contain less than 3 or more than 30 characters.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `length(34)`. If you have a number you must convert the number to text before using it with this function, for example `length(str(34))`.

See Also: [sizeof\(\)](#) function

LINEITEMARRAY(...)

Syntax: LINEITEMARRAY(field,separator)

Description: The `lineitemarray()` function converts the data in a set of line item fields into a text array (see [text arrays](#)).

Parameters: This function has two parameters: `field` and `separator`.

field is the line item field that contains the data. You should put the line item field name in quotes, and it should end with the Ω symbol (option-Z).

separator is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the `↵` character (option-L).

Result: This function returns a copy of the line item data packed into an array. If the line items contain numbers or dates they are converted to text before being added to the array.

Examples: This example will copy the data in the Price line item fields (`Price1`, `Price2`, `Price3`, etc.) into a text item named `PriceArray`:

```
PriceArray=lineitemarray("PriceΩ",";")
```

In this case the prices will be converted to text before they are added to the array. This will produce an array something like this:

```
43.25;167.12;22.95;4.25
```

The `lineitemarray()` function is also useful for searching a database. This procedure will find all line items that contain the word `red`, no matter what line item the item is in:

```
select lineitemarray("DescriptionΩ",";") contains "red"
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text or separator parameters.

Field or variable does not exist. This error occurs if the line item field you specify is not in the current database. You probably misspelled the field name.

See Also: [text arrays](#)
[array\(\)](#) function
[arraybuild](#) statement

List SuperObject Programming

Background: The [superobject](#) and [activesuperobject](#) statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

Quick

```
"GetList" , <List>
"GetSelected" , <List>
"GetCount" , <Number>
"GetCell" , <CellNumber> , <Result>
"FindCell" , <CellNumber> , <Search>
"SelectCell" , <CellNumber>
"UnselectCell" , <CellNumber>
"SetCell" , <CellNumber> , <Text>
"AddCell" , <Text>
"InsertCell" , <CellNumber> , <Text>
"DeleteCell" , <StartCell> , <EndCell>
"AutoScroll"
"FindSelected" , <CellNumber>
"FillList" , <Formula> , <Database>
"CellRectangle" , <Item> , <Rectangle>
"PointToCell" , <Point> , <Cell>
```

GetList

,<List>

This command produces a list of all the items in the list, with each item separated from the next by a carriage return. The list is placed into the field or variable specified by <List>.

GetSelected

,<List>

This command produces a list of all the selected items in the list, with each item separated from the next by a carriage return. The list is placed into the field or variable specified by <List>. (Note: This command is redundant if the list is already associated with a field or variable.)

GetCount

,<Number>

This command returns a count of the total number of items currently in the list into the field or variable specified by <Number>.

GetCell

,<CellNumber>,<Result>

This command extracts the contents of a particular item in the list. This command assumes the items are numbered, starting from 1 at the top of the list. This example will copy the first item in the list PartsList into the variable NextPart.

```
local NextPart
superobject "PartsList" , "GetCell" , 1 , NextPart
```

This example will copy the last item in the list PartsList into the variable NextPart.

```

local NextPart,ListCount
superobject "PartsList","GetCount",ListCount
superobject "PartsList","GetCell",ListCount,NextPart

```

FindCell ,<CellNumber>,<SearchText>

This command searches the list to find a specified value. The list item must match exactly, or the search will be unsuccessful. The search starts with the item specified by <CellNumber>. If successful, the number of the item containing the searched for value will be placed in <CellNumber>, otherwise <CellNumber> will be set to zero. The example below will locate Garlic in the list of pizza toppings and select it (tasty!).

```

local ListCell ListCell=1
superobject "Toppings","FindCell",ListCell,"Garlic"
if ListCell≠0
    superobject "Toppings","SelectCell",ListCell
endif

```

Keep in mind that the word or phrase must match exactly. In this case only Garlic will be located; garlic or Roasted Garlic will not.

SelectCell ,<CellNumber>

This command selects a specified item in the list. The item is specified by <CellNumber>, which should be a number from 1 to the maximum number of items in the list. See the previous command for an example of this command.

UnSelectCell ,<CellNumber>

This command unselects a specified item in the list. The item is specified by <CellNumber>, which should be a number from 1 to the maximum number of items in the list. The example below makes sure that there are no anchovies on the pizza!

```

local ListCell
ListCell=1
superobject "Toppings","FindCell",ListCell,"Anchovies"
if ListCell≠0
    superobject "Toppings","UnSelectCell",ListCell
endif

```

SetCell ,<CellNumber>,<NewText>

This command changes the contents of a specified item in the list. The item is specified by <CellNumber>, and should be from 1 to the maximum number of items in the list. The example below changes the Cheese item to Extra Cheese.

```

local ListCell
ListCell=1
superobject "Toppings","FindCell",ListCell,"Cheese"
if ListCell≠0
    superobject "Toppings","SetCell",ListCell,"Extra Cheese"
endif

```

AddCell ,<NewText>

This command adds a new item to the end of the list. This example adds the item Sun Dried Tomatoes to the end of the list of pizza toppings.

```
superobject "Toppings", "AddCell", "Sun Dried Tomatoes"
```

InsertCell ,<CellNumber>,<NewText>

This command inserts a new item into the middle of the list. The <CellNumber> parameter specifies where the new item should be inserted. The new item will go above the item specified. For example, you could insert the item Extra Cheese at the very top of the pizza topping list:

```
superobject "Toppings", "InsertCell", 1, "Extra Cheese"
```

This more complex example inserts Grilled Onions after Onions.

```
local ListCell
ListCell=1
superobject "Toppings", "FindCell", ListCell, "Onions"
if ListCell#0
    ListCell=ListCell+1
    superobject "Toppings", "InsertCell", ListCell, "Grilled Onions"
endif
```

Notice that the example adds one to ListCell before inserting the new item. This is so the new item (Grilled Onions) will be inserted after the original item (Onions) instead of before it.

DeleteCell ,<StartingCell>,<EndingCell>

This command deletes one or more items from the list. If you just want to delete a single cell, then only one number is needed. This example deletes the first item in the list.

```
superobject "Toppings", "DeleteCell", 1
```

To delete a bunch of cells at once, specify two numbers. This example deletes the first 5 items in the list.

```
superobject "Toppings", "DeleteCell", 1, 5
```

This example will delete the entire list in a big hurry!

```
superobject "Toppings", "DeleteCell", 1, 10000
```

AutoScroll

This command scrolls the list so that the first selected item is visible. For example, this procedure selects **Pineapple** and scrolls the list to make sure that the **Pineapple** item is visible.

```
local ListCell
ListCell=1
superobject "Toppings", "FindCell", ListCell, "Pineapple"
if ListCell#0
    superobject "Toppings", "SelectCell", ListCell
    superobject "Toppings", "AutoScroll"
endif
```


FindSelected ,<CellNumber>

This command finds the next selected cell, starting with <CellNumber>. The result is placed in <CellNumber>, or zero if there are no selected cells below the starting spot. The example below deletes all the selected items from the list.

```

local Spot
Spot=1
loop
  superobject "Toppings", "FindSelected", Spot
  if Spot=0
    stop
  endif
  superobject "Toppings", DeleteCell, Spot
next

```

FillList ,<Formula>,<Database>

This command re-fills the specified list. You can use this command to update the list, or to fill it with completely new information.

To update the list, leave off the <Formula> and <Database>. The FillList command will update the list using the formula and database specified in the List dialog. For example, if the pizza toppings list was derived from a pizza toppings database, you would want to use this procedure when the pizza toppings database had changed:

```

superobject "Toppings", "FillList"

```

You can also use the FillList command to fill the list with entirely new information, completely ignoring the formula and database originally specified in the List dialog. The same list can be filled and refilled again and again with different items as conditions change. Below are three samples that could be used to fill a list from the Pizza Toppings database. The first sample lists all toppings, the next veggie only, the final meat only.

```

superobject "Toppings", "FillList",
  {Topping}, "Pizza Toppings"

superobject "Toppings", "FillList",
  {?(Category="Veggie", Topping, " ")}, "Pizza Toppings"

superobject "Toppings", "FillList",
  {?(Category="Meat", Topping, " ")}, "Pizza Toppings"

```

Of course you can also use the FillList command to directly specify the contents of the list. In this case the database should be set to "".

```

superobject "Toppings", "FillList",
  {"Pepperoni"+¶+"Sausage"+¶+"Meatballs"+¶+
  "Mushrooms"+¶+"Olives"+¶+"Onions"}, ""

```

The topping list can also be created with variables. Here's an example:

```

local MeatToppings, VeggieToppings, SpecialtyToppings
MeatToppings="Pepperoni"+¶+"Sausage"+¶+"Meatballs"
VeggieToppings="Mushrooms"+¶+"Olives"+¶+"Onions"
SpecialtyToppings="Anchovies"+¶+"Garlic"
superobject "Toppings", "FillList", {VeggieToppings}, ""

```

CellRectangle ,<Item>,<Rectangle>

This command allows a procedure to determine the physical location and size (i.e. rectangle) of any item in the list. This command has two parameters as shown below: the item number (from 1 to the maximum number of items in the list) and the rectangle. The Rectangle should be a variable that will contain the final result.

Here is an example that fills in the variable dragRectangle with the dimensions of the third item in the list.

```
superobject "My List", "CellRectangle", 3, dragRectangle
```

Note: The rectangle that is returned by this command is in window relative co-ordinates. You can change this to screen or form relative co-ordinates using the xytoxy(function.

PointToCell ,<Item>,<Rectangle>

This command allows a procedure to determine what list item (if any) corresponds to any point on the screen. For example, if someone drags something onto the list, this command allows the procedure to determine where in the list the item should be placed. This command has two parameters as shown below: the Point and the Cell. The Cell parameter should be a variable that will contain the final result.

```
superobject "object name", "PointToCell", Point, Cell
```

An Example of Dragging from a List

This example shows how items from a list can be dragged and dropped to another location on the screen. This example assumes that there is a List SuperObject on the current form called Alphabet List, and that this object has the Click/Release option turned off.

```
local cell, cellbox
cell=1
superobject "Alphabet List", "findselected", cell
superobject "Alphabet List", "cellrectangle", cell, cellbox
cellbox=xytoxy(cellbox, "w", "s")
draggraybox cellbox, "", "", 0
/* normal post-drag code follows */
```

The third line of this procedure finds out what item in the list is currently selected (item 1, 2, 3, etc.) The fourth line of this procedure uses this item number to find out the rectangle of the selected item. The fifth line converts this rectangle to screen relative co-ordinates instead of window relative co-ordinates. Finally, the draggraybox statement allows the user to drag the item around on the screen. The Real World Programmer's Guide shows how to finish this procedure by allowing the item to be dragged onto another form item or window.

Dragging to Change the Order of a List

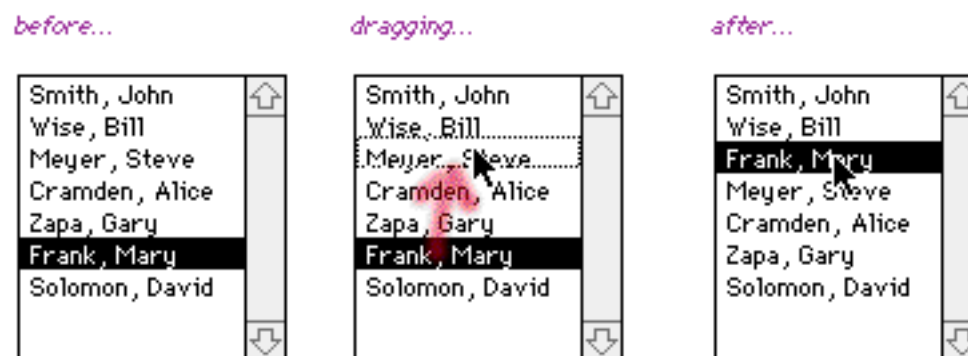
This example shows how to set up a procedure that allows the user to drag items up or down in a list to change the order of a list. This example assumes that there is a list of names in a field called **Names**, with each name separated from the next by a carriage return. The current form must contain a List SuperObject named **Names List** (use the **Object Name** command in the Edit menu to change object names). The **Names List** object displays the Names field, it is also linked to a global variable named theName.

```

global theName
local cell,cellbox,listbox,mouse,newcell,newNames
cell=1
superobject "Names List","findselected",cell
superobject "Names List","cellrectangle",cell,cellbox
cellbox=xytoxy(cellbox,"w","s")
object "Names List"
listbox=xytoxy( objectinfo("rectangle"),"f","s")
draggraybox cellbox,listbox,listbox,0
if cellbox="" stop endif /* user dragged out of the list */
mouse=xytoxy( info("mouse"),"s","w")
/* where is the new location for this item? */
superobject "Names List","pointtocell",mouse,newcell
if cell=newcell stop endif /* item did not move */
if newcell>cell
newcell=newcell-1 /* adjust for deleting item in old spot */
endif
/* delete item from old spot */
newNames=arraydelete(Names,cell,1,1)
if newcell>0
/* insert item in new spot */
newNames=arrayinsert(newNames,newcell,1,1)
newNames=arraychange(newNames,theName,newcell,1)
else
/* add item to end of list */
newNames=newNames+1+theName
endif
Names=newNames /* update original field */
superobject "Names List","filllist" /* re-display list */
showvariables theName/* and select correct item */

```

The illustration below shows this procedure in action.



LISTCHOICES(...)

Syntax: LISTCHOICES(field,separator)

Description: The `listchoices()` function builds a text array containing a list of all the values stored in a specified field. (Note: this function is not related to the `choices` data type.)

Parameters: This function has two parameters: `field` and `separator`.

field is the name of the field that contains the values you want build a list of.

separator is the separator character for the text array you are building (see [text arrays](#)).

Result: The function scans the specified field and builds a list of all the values stored in that field. The list is returned in the format of a text array (see [text arrays](#)).

Examples: The `listchoices()` function is often used to build lists for Pop-Up Menu or List SuperObjects™. Here is a formula that builds a list of the states in the current database.

```
listchoices(State,¶)
```

Since this text array uses carriage returns for separators, it can be used directly in a Pop-Up Menu or List SuperObject™. A pop-up menu from this formula might look like this:

AZ
CA
OR
NU

By adding to this option we can add an **Other...** option to the pop-up menu.

```
sandwich(" ",listchoices(State,¶),"(-"+¶+"Other...")
```

AZ
CA
OR
NU
Other...

Errors: **Field or variable does not exist.** This error occurs if there is no field in the current database with the name you have specified. You probably misspelled the field name.

See Also: [lookupall\(\)](#) function
[text arrays](#)

LISTFILES(...)

Syntax: LISTFILES(folder,filter)

Description: The `listfiles()` function builds a text array listing the files in a folder.

Parameters: This function has two parameters: `folder` and `filter`.

folder is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()` and `dbinfo()` functions, and the `openfiledialog` and `savefiledialog` statements.

filter is a text item that specifies what type (or types) of files (and folders) to list. If this is an empty text item ("") all files will be listed. Otherwise the type parameter should be a series of one or more 8 character sections. The first four characters are the file type, the second four are the file creator. You can also use the ? character if you do not care what a character is. Here are some useful file types:

`TEXT????` list all text files

`APPL????` list all applications

`????KASX` list all Panorama database files (Panorama 3.5 or higher)

You can combine more than one specification into a filter, for example `TEXT????????KASX` to list all text files and Panorama database files.

The `listfiles()` function normally does not list folders. However, if you precede the filter specification with the `f` (option-F) character the function will list folders as well as files. For example:

`fTEXT????` list all text files and folders

`f????KASXTEXT????` list databases, text files, and folders

If the filter is empty ("") then ALL files and folders will be included.

Result: This function returns a carriage return separated text array. Each item contains a single file name.

Examples: This example counts and displays the number of database files in the folder that contains the currently running copy of Panorama.

```
message "There are "+
str( arraysize(listfiles(info("panoramafolder"), "????KAS1"),
    ¶))+" database files in the Panorama folder."
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder or filter parameters. This error can also occur if the folder parameter is more or less than 6 bytes long.

See Also: [folder\(\)](#) function
[folderpath\(\)](#) function
[dbinfo\(\)](#) function
[info\("panoramafolder"\)](#) function
[info\("systemfolder"\)](#) function

info("volumes") function
openfiledialog statement
savefiledialog statement

LISTWINDOWS(...)

Syntax: LISTWINDOWS(file)

Description: The `listwindows(` function builds a text array containing a list of all the open windows associated with a particular file.

Parameters: This function has one parameter: file.

file is the name of the database file that you want to list the windows of. This should be the name of an open database. If the file parameter is empty ("") the `listwindows(` function will list all open windows, no matter what database they are in.

Result: The function scans the windows and builds a text array using carriage returns (¶) as separators (see [text arrays](#)). The windows are listed in order from front to back.

Examples: The example below uses `listwindows(` to check to see if the Invoice:Status window is open. If not, the procedure opens the form.

```
if 0=arraysearch(listwindows("Invoice"),"Invoice:Status",1,¶)
  setwindowrectangle rectangle(20,5,120,255),"
  openform "Status"
endif
```

The `listwindows(` function could be used to build a Pop-Up Menu SuperObject™.

```
listwindows("")
```

This formula will list all of the current Panorama windows. A pop-up menu based on this formula will look something like this:

<p>Invoice:MAIN Invoice:Status Price List Customer List:Form</p>

Errors: This function does not generate any error messages.

See Also: [info\("windows"\)](#) function
[info\("files"\)](#) function
[dbinfo\(](#) function
[text arrays](#)

LOADFILEVARIABLES

- Syntax:** `LOADFILEVARIABLES variablelist,variablevalues,separator`
- Description:** The `loadfilevariables` statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them as [fileglobal](#) variables. (Note: You can easily create an array of values using the [savevariables](#) statement.)
- Parameters:** This statement has three parameters: `variablelist`, `variablevalues` and `separator`.
- `variablelist`** is an array containing the names of the variables to be "loaded" with the values from the `variablevalues` parameter. Each item in the array must be separated from the next by the separator character.
- `variablevalues`** is the list of values to be loaded into the variables. The values may be in a field, a variable, or may be generated with a formula. Each value must be separated from the next with the separator character.
- `separator`** is the character that will be used to separate the values in the combined array. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example `chr(1)` or `chr(255)`. For more information on separators see [text arrays](#)
- Action:** This statement takes an array of values and loads each value into a separate array. You could perform the same operation with a loop, but the `loadfilevariables` statement is much faster. One use for the `loadfilevariables` statement is importing a group of variables from a field or file (perhaps a field or file generated with the [savevariables](#) statement.)
- Examples:** Here is a very simple example that loads three variables from an array.

```
loadfilevariables
  "Gold,Silver,Bronze" ,
  "Johnson,Smith,Fetzl" ,
  " , "
```

This example is exactly the same as:

```
fileglobal Gold,Silver,Bronze
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

Actually it is not quite exactly the same. If the variables `Gold`, `Silver` and `Bronze` have already been defined as [local](#), [global](#), or [windowglobal](#) variables, the `loadfilevariables` statement will not re-define them. If a variable already contains a numeric value, the `loadfilevariables` statement keep it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
fileglobal Red,Green,Blue
Red=0 Green=0 Blue=0
loadfilevariables "Red,Green,Blue" ,"24,58,199" , " , "
```


This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```
fileglobal Red,Green,Blue
Red=24
Green=58
Blue=199
```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called ContactInfo that contained name/value pairs like this:

```
Name=Johnson
Email=ajohnson@worldwide.com
url=www.worldwide.com
```

The example below can take this array and separate it into three variables called [ContactName](#), [ContactEmail](#), and [Contacturl](#).

```
global ContactInfo
fileglobal contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,¶,"Contact"+array(
import(,1,"=")
arrayfilter ContactInfo,contactValues,¶,array( import(,2,"=")
loadfilevariables contactVariables,contactValues,¶
```

The procedure starts by splitting the ContactInfo array into two separate arrays for variable names and values, then creates and loads the variables with the values. For an example of how to generate a file like this, see the [savevariables](#) statement.

Views: This statement may be used in any view.

See Also: [savevariables](#) statement
[loadglobalvariables](#) statement
[loadvariables](#) statement
[loadlocalvariables](#) statement
[loadwindowvariables](#) statement

LOADGLOBALVARIABLES

Syntax: `LOADGLOBALVARIABLES variablelist,variablevalues,separator`

Description: The `loadglobalvariables` statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them as [global](#) variables. (Note: You can easily create an array of values using the [savevariables](#) statement.)

Parameters: This statement has three parameters: `variablelist`, `variablevalues` and `separator`.

variablelist is an array containing the names of the variables to be "loaded" with the values from the `variablevalues` parameter. Each item in the array must be separated from the next by the `separator` character.

variablevalues is the list of values to be loaded into the variables. The values may be in a field, a variable, or may be generated with a formula. Each value must be separated from the next with the `separator` character.

separator is the character that will be used to separate the values in the combined array. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example `chr(1)` or `chr(255)`. For more information on separators see [text arrays](#)

Action: This statement takes an array of values and loads each value into a separate array. You could perform the same operation with a loop, but the `loadglobalvariables` statement is much faster. One use for the `loadglobalvariables` statement is importing a group of variables from a field or file (perhaps a field or file generated with the `savevariables` statement.)

Examples: Here is a very simple example that loads three variables from an array.

```
loadglobalvariables
  "Gold,Silver,Bronze" ,
  "Johnson,Smith,Fetzl" ,
  " , "
```

This example is exactly the same as:

```
global Gold,Silver,Bronze
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

Actually it is not quite exactly the same. If the variables `Gold`, `Silver` and `Bronze` have already been defined as [local](#), [fileglobal](#), or [windowglobal](#) variables, the `loadglobalvariables` statement will not re-define them. If a variable already contains a numeric value, the `loadglobalvariables` statement keep it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
global Red,Green,Blue
Red=0 Green=0 Blue=0
loadglobalvariables "Red,Green,Blue" ,"24,58,199" , " , "
```

This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```
global Red,Green,Blue
Red=24
Green=58
Blue=199
```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called `ContactInfo` that contained name/value pairs like this:

```
Name=Johnson
Email=ajohnson@worldwide.com
url=www.worldwide.com
```

The example below can take this array and separate it into three variables called `ContactName`, `ContactEmail`, and `Contacturl`.

```
global ContactInfo
global contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,["Contact"+array(
import(),1,"=")
arrayfilter ContactInfo,contactValues,["array( import(),2,"=")
loadglobalvariables contactVariables,contactValues,["
```

The procedure starts by splitting the `ContactInfo` array into two separate arrays for variable names and values, then creates and loads the variables with the values. For an example of how to generate a file like this, see the [savevariables](#) statement.

Views: This statement may be used in any view.

See Also: [savevariables](#) statement
[loadfilevariables](#) statement
[loadvariables](#) statement
[loadlocalvariables](#) statement
[loadwindowvariables](#) statement

LOADLOCALVARIABLES

Syntax: `LOADLOCALVARIABLES variablelist,variablevalues,separator`

Description: The `loadlocalvariables` statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them as [local](#) variables. (Note: You can easily create an array of values using the [savevariables](#) statement.)

Parameters: This statement has three parameters: `variablelist`, `variablevalues` and `separator`.

variablelist is an array containing the names of the variables to be "loaded" with the values from the `variablevalues` parameter. Each item in the array must be separated from the next by the separator character.

variablevalues is the list of values to be loaded into the variables. The values may be in a field, a variable, or may be generated with a formula. Each value must be separated from the next with the separator character.

separator is the character that will be used to separate the values in the combined array. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example `chr(1)` or `chr(255)`. For more information on separators see [text arrays](#)

Action: This statement takes an array of values and loads each value into a separate array. You could perform the same operation with a loop, but the `loadlocalvariables` statement is much faster. One use for the `loadlocalvariables` statement is importing a group of variables from a field or file (perhaps a field or file generated with the `savevariables` statement.)

Examples: Here is a very simple example that loads three variables from an array.

```
loadlocalvariables
  "Gold,Silver,Bronze",
  "Johnson,Smith,Fetzl",
  ", "
```

This example is exactly the same as:

```
local Gold,Silver,Bronze
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

Actually it is not quite exactly the same. If the variables `Gold`, `Silver` and `Bronze` have already been defined as [local](#), [global](#), or [fileglobal](#) variables, the `loadlocalvariables` statement will not re-define them. If a variable already contains a numeric value, the `loadlocalvariables` statement keep it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
local Red,Green,Blue
Red=0 Green=0 Blue=0
loadlocalvariables "Red,Green,Blue", "24,58,199", " , "
```

This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```

local Red,Green,Blue
Red=24
Green=58
Blue=199

```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called `ContactInfo` that contained name/value pairs like this:

```

Name=Johnson
Email=ajohnson@worldwide.com
url=www.worldwide.com

```

The example below can take this array and separate it into three variables called `ContactName`, `ContactEmail`, and `Contacturl`.

```

global ContactInfo
local contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,["Contact"+array(
import(),1,"=")
arrayfilter ContactInfo,contactValues,["array( import(),2,"=")
loadlocalvariables contactVariables,contactValues,["

```

The procedure starts by splitting the `ContactInfo` array into two separate arrays for variable names and values, then creates and loads the variables with the values. For an example of how to generate a file like this, see the [savevariables](#) statement.

Views: This statement may be used in any view.

See Also:

- [savevariables](#) statement
- [loadfilevariables](#) statement
- [loadvariables](#) statement
- [loadglobalvariables](#) statement
- [loadwindowvariables](#) statement

LOADVARIABLES

Syntax: `LOADVARIABLES variablelist,variablevalues,separator`

Description: The `loadvariables` statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them as [fileglobal](#) variables. (Note: You can easily create an array of values using the [savevariables](#) statement.)

Parameters: This statement has three parameters: `variablelist`, `variablevalues` and `separator`.

variablelist is an array containing the names of the variables to be "loaded" with the values from the `variablevalues` parameter. Each item in the array must be separated from the next by the separator character.

variablevalues is the list of values to be loaded into the variables. The values may be in field, a variable, or may be generated with a formula. Each value must be separated from the next with the separator character.

separator is the character that will be used to separate the values in the combined array. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example `chr(1)` or `chr(255)`. For more information on separators see [text arrays](#)

Action: This statement takes an array of values and loads each value into a separate array. You could perform the same operation with a loop, but the `loadvariables` statement is much faster. One use for the `loadvariables` statement is importing a group of variables from a field or file (perhaps a field or file generated with the [savevariables](#) statement.)

Examples: Here is a very simple example that loads three variables from an array.

```
loadvariables
  "Gold,Silver,Bronze",
  "Johnson,Smith,Fetzl",
  ", "
```

This example is exactly the same as:

```
fileglobal Gold,Silver,Bronze
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

Actually it is not quite exactly the same. If the variables `Gold`, `Silver` and `Bronze` have already been defined as [local](#), [global](#), or [windowglobal](#) variables, the `loadfilevariables` statement will not re-define them. If a variable already contains a numeric value, the `loadvariables` statement keep it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
fileglobal Red,Green,Blue
Red=0 Green=0 Blue=0
loadfilevariables "Red,Green,Blue", "24,58,199", ", "
```

This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```
fileglobal Red,Green,Blue
Red=24
Green=58
Blue=199
```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called `ContactInfo` that contained name/value pairs like this:

```
Name=Johnson
Email=ajohnson@worldwide.com
url=www.worldwide.com
```

The example below can take this array and separate it into three variables called `ContactName`, `ContactEmail`, and `Contacturl`.

```
global ContactInfo
fileglobal contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,¶,"Contact"+array(
import(),1,"=")
arrayfilter ContactInfo,contactValues,¶,array( import(),2,"=")
loadvariables contactVariables,contactValues,¶
```

The procedure starts by splitting the `ContactInfo` array into two separate arrays for variable names and values, then creates and loads the variables with the values. For an example of how to generate a file like this, see the [savevariables](#) statement.

Views: This statement may be used in any view.

See Also: [savevariables](#) statement
[loadglobalvariables](#) statement
[loadlocalvariables](#) statement
[loadwindowvariables](#) statement

LOADWINDOWVARIABLES

Syntax: LOADWINDOWVARIABLES variablelist,variablevalues,separator

Description: The **loadwindowvariables** statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them as [windowglobal](#) variables. (Note: You can easily create an array of values using the [savevariables](#) statement.)

Parameters: This statement has three parameters: **variablelist**, **variablevalues** and **separator**.

variablelist is an array containing the names of the variables to be "loaded" with the values from the **variablevalues** parameter. Each item in the array must be separated from the next by the separator character.

variablevalues is the list of values to be loaded into the variables. The values may be in a field, a variable, or may be generated with a formula. Each value must be separated from the next with the separator character.

separator is the character that will be used to separate the values in the combined array. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example [chr\(1\)](#) or [chr\(255\)](#). For more information on separators see [text arrays](#)

Action: This statement takes an array of values and loads each value into a separate array. You could perform the same operation with a loop, but the **loadwindowvariables** statement is much faster. One use for the **loadwindowvariables** statement is importing a group of variables from a field or file (perhaps a field or file generated with the [savevariables](#) statement.)

Examples: Here is a very simple example that loads three variables from an array.

```
loadwindowvariables
  "Gold,Silver,Bronze" ,
  "Johnson,Smith,Fetzl" ,
  ", "
```

This example is exactly the same as:

```
windowglobal Gold,Silver,Bronze
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

Actually it is not quite exactly the same. If the variables Gold, Silver and Bronze have already been defined as [local](#), [global](#), or [fileglobal](#) variables, the **loadwindowvariables** statement will not re-define them. If a variable already contains a numeric value, the **loadwindowvariables** statement keep it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
windowglobal Red,Green,Blue
Red=0 Green=0 Blue=0
loadfilevariables "Red,Green,Blue" ,"24,58,199" , " , "
```


This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```
windowglobal Red,Green,Blue
Red=24
Green=58
Blue=199
```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called `ContactInfo` that contained name/value pairs like this:

```
Name=Johnson
Email=ajohnson@worldwide.com
url=www.worldwide.com
```

The example below can take this array and separate it into three variables called `ContactName`, `ContactEmail`, and `Contacturl`.

```
global ContactInfo
windowglobal contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,¶,"Contact"+array(
import(,1,"=")
arrayfilter ContactInfo,contactValues,¶,array( import(,2,"=")
loadwindowvariables contactVariables,contactValues,¶
```

The procedure starts by splitting the `ContactInfo` array into two separate arrays for variable names and values, then creates and loads the variables with the values. For an example of how to generate a file like this, see the [savevariables](#) statement.

Views: This statement may be used in any view.

See Also: [savevariables](#) statement
[loadglobalvariables](#) statement
[loadvariables](#) statement
[loadlocalvariables](#) statement
[loadfilevariables](#) statement

LOCAL

Syntax: LOCAL variables

Description: The **local** statement creates one or more local variables. Local variables are temporary, and vanish when the procedure is finished.

Parameters: This statement has one parameter: **variables**.

variables is a list of variables to be created. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: This statement creates one or more local variables. Local variables can be used to temporarily hold pieces of information (numbers or text). Each variable has a name.

Panorama keeps the local variables for each procedure completely separate. If procedure A creates a local variable named myValue, that local variable cannot be used by procedure B. In fact, procedure B can create its own variable called myValue, and the two will be kept completely separate, with different values.

Examples: The example creates two local variables, **Counter** and **Operating Ratio**.

```
local Counter, «Operating Ratio»
```

You may change the value of a variable with an assignment, like this:

```
«Operating Ratio»=Revenues/Costs
```

Views: This statement may be used in any view.

See Also: [global](#) statement
[fileglobal](#) statement
[globalize](#) statement
[windowglobal](#) statement
[permanent](#) statement
[undefine](#) statement

LOCKORSTOP

Syntax: LOCKORSTOP

Description: The `lockorstop` statement attempts to lock the currently active record. This statement only applies to Partner/Server databases. It is ignored when used with a standard Panorama database.

Parameters: This statement has no parameters.

Action: This statement attempts to lock the current record. If the current record is not already locked by another user on the network, the record is locked and the procedure continues normally. If the current record is already locked by another user on the network, the procedure will display a message and stop immediately. Unlike the `lockrecord` statement, this statement does not pause to wait for the record to become available—it gives up right away. If you don't want the procedure to stop at this point you should use the `if error` statement to trap the error. (Note: Panorama automatically locks the current record when a procedure modifies any field in the record. However when using automatic locking the programmer cannot control the response if the record is already locked.)

Examples: This example makes 10 attempts to lock a record. If it does not succeed after ten tries, it gives up.

```

local tryCount
tryCount=1
loop
  lockorstop
  if error
    tryCount=tryCount+1
    if tryCount>10
      rtn /* stop trying */
    endif
  else
    stoploopif 0=0 /* sweet success! */
  endif
while forever
  Qty=Qty-SalesQty
unlockrecord

```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [lockrecord](#) statement
[unlockrecord](#) statement
[info\("serverstatus"\)](#) function
[info\("servertimeout"\)](#) function

LOCKRECORD

Syntax: LOCKRECORD

Description: The `lockrecord` statement attempts to lock the currently active record. This statement only applies to Partner/Server databases. It is ignored when used with a standard Panorama database.

Parameters: This statement has no parameters.

Action: This statement attempts to lock the current record. If the current record is not already locked by another user on the network, the record is locked and the procedure continues normally. If the current record is already locked by another user on the network, the procedure will pause and wait until the record becomes available. (This is the difference between **LockRecord** and the [lockorstop](#) statement, which simply stops immediately without pausing.) While it is waiting, the procedure will display a dialog explaining the situation. If the user does not want to wait, they can press Command-Period to stop waiting and abort the procedure. If you don't want the procedure to stop at this point you should use the [if](#) error statement to trap the error. (Note: Panorama automatically locks the current record when a procedure modifies any field in the record. However when using automatic locking the programmer cannot control the response if the record is already locked.)

Examples: This example attempts to lock a record in an invoice database. If the invoice is not already being edited by another user, the procedure opens a form to allow the user to edit the record.

```
lockrecord
if error
    message "Sorry, someone else is using this invoice now!"
    stop
else
    openform "Invoice"
endif
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [lockorstop](#) statement
[unlockrecord](#) statement
[info\("serverstatus"\)](#) function
[info\("servertimeout"\)](#) function

LOG(...)

Syntax: LOG(value)

Description: The `log()` function computes the natural logarithm (base e) of a value.

Parameters: This function has one parameter: value.
value is a numeric value.

Result: The result of this function is a numeric floating point value.

Examples: e is a mathematical constant that is approximately 2.71828. This function calculates a logarithm using e as the base. Mathematicians call this a natural logarithm. The graph below shows the result of the natural logarithm function given input values from -10 to +10.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `log("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating a natural logarithm, for example `log(val("34"))`.

Floating point error. The natural logarithm of values 0 or less (negative numbers) is undefined. If you attempt to calculate the natural logarithm of such a number, a floating point error will occur.

See Also: [exp\(\)](#) function
[val\(\)](#) function

LOG10(...)

Syntax: LOG10(value)

Description: The `log10()` function computes the common logarithm (base10) of a value.

Parameters: This function has one parameter: value.
value is a numeric value.

Result: The result of this function is a numeric floating point value.

Examples: This function calculates a logarithm using 10 as the base. Mathematicians call this a common logarithm. The graph below shows the result of the common logarithm function given input values from -10 to +10.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `log10("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating a natural logarithm, for example `log10(val("34"))`.

Floating point error. The common logarithm of values 0 or less (negative numbers) is undefined. If you attempt to calculate the common logarithm of such a number, a floating point error will occur.

See Also: [log\(\) function](#)
[val\(\) function](#)

LOGMESSAGE

Syntax: LOGMESSAGE message

Description: The **logmessage** statement writes a message to the current debug log file (if any).

Parameters: This statement has one parameter message.
message is the text that is to be written to the log file. You may use any formula to create the text.

Action: Use the **logmessage** statement to include custom information in the log file. To open a log file use the **Debug Log Wizard**.

Examples: This example uses the logmessage statement to record the progress of a loop.

```

local theWord,n,words,sentences
n=1 words=0 sentences=0
loop
  theWord=array( replace( replace(Joke,¶," "), " ", " "),n," ")
  stoploopif theWord=""
  theWord=strip(theWord)
  words=words+1
  logmessage "("+str(words)+") "+theWord
  if theWord contains "." or theWord contains "?" or theWord contains "!"
    sentences=sentences+1
    logmessage " ("+str(sentences)+")"
  endif
  n=n+1
while forever
  logmessage "Complete ****"

```

Here is an example of the log created by this procedure.

```

(1) Please
(2) distribute
(3) this
(4) to
(5) everyone
(6) you
(7) know
(8) on
(9) earth.
(1)
(10) When
(11) John
(12) Glenn
(13) returns
(14) from
(15) space,
(16) everybody
(17) dress
(18) in
(19) Ape
(20) Suits.
(2)
(21) We
(22) have
(23) 6
(24) days
(25) in

```

```
(26) which
(27) to
(28) bury
(29) the
(30) Statue
(31) of
(32) Liberty
(33) up
(34) to
(35) her
(36) head.
(3)
(37) EVERYBODY
(38) HURRY
(39) !
(4)
(40) !
(5)
(41) !
(6)
Complete ****
```

Views: This statement may be used in any view.

See Also: [debug](#) statement
[message](#) statement
[statusmessage](#) statement

LOGNORMAL

Syntax: LOGNORMAL

Description: The **lognormal** statement changes the current security level back to the user's security level after it has been bumped up by the [logprogrammer](#) statement.

Parameters: This statement has no parameters.

Action: If a database uses Panorama's security system, each user has a security level from 0 (low) to 255 (high). A procedure normally cannot perform any action that the user running the procedure is not allowed to perform. If the procedure programmer has a higher security level than the final end user, he or she can temporarily give that higher level to the procedure with the [logprogrammer](#) statement. To set the security level back to the user's real level use the lognormal statement. (Panorama will automatically set the security level back to the user's real level at the end of the procedure, even if you forget.)

Examples: Suppose the user doesn't have the authority to change the Salary field, but the person writing the procedure does have a high enough level. The example below shows how the procedure can be written to change the salary.

```
logprogrammer  
Salary=Salary*1.1  
lognormal
```

Views: This statement may be used in any view

See Also: [logprogrammerr](#) statement
[logon](#) statement
[info\("userlevel"\)](#) function

LOGON

Syntax: LOGON

Description: The **logon** statement opens the dialog allowing the user to log on or re-log into Panorama's security system. This statement is usually unnecessary because Panorama automatically opens the logon dialog when you open a database that requires security

Parameters: This statement has no parameters.

Examples: This procedure asks a user to re-log on if their security level is less than 100.

```
if info("userlevel")<100
  logon
endif
```

Views: This statement may be used in any view.

See Also: [logprogrammer](#) statement
[lognormal](#) statement
[info\("userlevel"\)](#) function

LOGPROGRAMMER

Syntax: LOGPROGRAMMER

Description: The **logprogrammer** statement changes the current security level to the level of the last person who modified the procedure. If the person who programmed the procedure has a high security level and the person running the procedure has a low security level, this statement allows the procedure to temporarily “bump up” to the higher level under program control. Panorama will automatically return to the lower security level when the [lognormal](#) statement is used, or when the procedure ends.

Parameters: This statement has no parameters.

Action: If a database uses Panorama’s security system, each user has a security level from 0 (low) to 255 (high). A procedure normally cannot perform any action that the user running the procedure is not allowed to perform. If the procedure programmer has a higher security level than the final end user, he or she can temporarily give that higher level to the procedure with the logprogrammer statement. To set the security level back to the user’s real level use the [lognormal](#) statement. (Panorama will automatically set the security level back to the user’s real level at the end of the procedure, even if you forget.)

Examples: Suppose the user doesn’t have the authority to change the Salary field, but the person writing the procedure does have a high enough level. The example below shows how the procedure can be written to change the salary.

```
logprogrammer  
Salary=Salary*1.1  
lognormal
```

Views: This statement may be used in any view.

See Also: [lognormalstatement](#)
[logon](#) statement
[info\("userlevel"\)](#) function

LONGWORD(...)

Syntax: LONGWORD(number)

Description: The longword(function converts a number into a single longword (4 bytes) of binary data (see [binary data](#)).

Parameters: This function has one parameter: **number**.

number is the value that you want to convert into a binary number. This value must be an integer.

Result: This function converts the number into a single word of binary data (32 bits). This binary data should be handled as text data.

Examples: This example converts the number 285486 into a binary longword, then copies that binary data into the variable X.

```
local X  
X=longword(285486)
```

If you check the size of X with the sizeof(function, you'll find that it is 4 bytes long.

See [c/pascal structures](#) for additional examples of the **longword(** function.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the number parameter.

Illegal number. This error occurs if you attempt to convert a value less than 0 or greater than 65,535.

See Also: [byte\(](#) function
[word\(](#) function
[radix\(](#) function
[radixstr\(](#) function

LOOKUP(...)

Syntax: LOOKUPfile,keyField,keyValue,dataField,default,level)

Description: The `lookup()` function searches a database for a value, then returns other information from the same record. For example, the `lookup()` function can look up a phone number given a customer name, or look up a price given a part number.

Parameters: This function has six parameters: `file`, `keyField`, `keyValue`, `dataField`, `default` and `level`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("database")`.

keyField is the name of the field that you want to search in. For example if you want to look up a customer by name, this should be the field that contains customer names. The field must be in the database specified by the first parameter.

keyData is the actual data that you want to search for. For example if you want to look up a customer by name, this should be the actual name of the customer. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve a phone number, this should be the name of the field that contains phone numbers. This must be a field in the database specified by the first parameter.

default is the value you want this function to return if it is unable to find the information specified by the `keyField` and `keyData` parameters. The data type of the default value should match the data type of the `dataField`. If the `dataField` is numeric, the default should usually be zero. If the `dataField` is text, the default should usually be "".

level is the minimum summary level to be searched. Usually this parameter is zero so that the entire database will be searched. If the level is set to 1 through 7, only summary records will be searched.

Result: If the function is able to locate the information specified by the `keyField` and `keyData` parameters it returns the contents of the specified field in the specified database. (If there is more than one match, only the first will be returned.) If it cannot locate the information it returns the default value.

Examples: Suppose you have two databases: Invoices and Customers. These two databases have 5 identical fields: `Company`, `Address`, `City`, `State` and `Zip`. When the user enters the company name, we want to write a procedure that will automatically move the address, city, state and zip from the customer file into the new invoice.

```
gettext "What company name?",Company
Address=lookup("Customers",Company,Company,Address,"",0)
City=lookup("Customers",Company,Company,City,"",0)
State=lookup("Customers",Company,Company,State,"",0)
Zip=lookup("Customers",Company,Company,Zip,"",0)
```

Since each `lookup()` statement can only transfer one value, four lookups are required. However, Panorama doesn't actually scan the Customer file four times. When a procedure performs multiple lookups with the same target database, key field, and key value Panorama realizes that it doesn't have to re-scan the database—it already knows where

the data is, and it just goes and gets it. (If your database is set up for it, you can use the [speedcopy](#) statement to move this data even faster.) This example looks up a price from a Price List database. The [ItemΩ](#) line item field in the invoice database contains the catalog item number, which should match a value in the [Catalog#](#) field in the Price List.

```
PriceΩ=lookup("Price List",«Catalog#»,ItemΩ,Price,0,0)
```

Errors:

Database does not exist. This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookuplast\(\)](#) function
[lookupselected\(\)](#) function
[lookuplastselected\(\)](#) function
[table\(\)](#) function
[grabdata\(\)](#) function
[lookupall\(\)](#) function
[serverlookup](#) statement
[formserverlookup](#) statement

LOOKUPALL(...)

Syntax: LOOKUPALL(file,keyField,keyValue,dataField,separator)

Description: The `lookupall()` function builds a text array containing one item for every record in the target database where the data in the `keyField` matches the `keyValue`. Each item in the text array contains the value extracted from the `dataField` for that record. If the data field is a numeric or date field, it is converted to text using the default patterns for that field.

Parameters: This function has five parameters: `file`, `keyField`, `keyValue`, `dataField` and `separator`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use [info\("database"\)](#).

keyField is the name of the field that you want to search in. For example if you want to look up all checks written to a certain vendor, this should be the field that contains vendor names. The field must be in the database specified by the first parameter.

keyData is the actual data that you want to search for. For example if you want to look up all checks written to a certain vendor, this should be the actual vendor name. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve check numbers, this should be the name of the field that contains check numbers. This must be a field in the database specified by the first parameter.

separator is the separator character for the text array you are building (see [text arrays](#)).

Result: The function returns a text array from all the records where the `keyField` and `keyData` match.

Examples: This example looks up all the checks written to a certain person or company. The checks are displayed with a comma in between each check number.

```

local CheckTo
CheckTo=""
gettext "List checks written to:",CheckTo
Checks=lookupall("Checkbook",Payee,CheckTo,«Check#»,",")
if Checks=""
    message "No checks written to "+CheckTo
else
    message "Checks written to "+CheckTo+": "+Checks
endif

```

The `lookupall()` function will often return a lot of duplicate data. Since the result is an array, you can use the [arraydeduplicate](#) statement to sort and eliminate the duplicates. This example produces a sorted list of customers in Arizona.

```

global theCustomers
theCustomers=lookupall("Invoices",State,"AZ",Company,¶)
arraydeduplicate theCustomers,theCustomers,¶

```

The `lookupall()` function is especially useful for displaying or printing lists of items, and for other user interface elements like lists or pop-up menus. The `lookupall()` function can be used directly in auto-wrap text or a Text Display SuperObject to display a list.

Errors: **Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also: [lookup\(\)](#) function
[lookuplast\(\)](#) function
[lookupselected\(\)](#) function
[table\(\)](#) function
[grabdata\(\)](#) function
[lookupcalendar\(\)](#) function
[lookupptime\(\)](#) function
[ascii](#)

LOOKUPCALENDAR(...)

Syntax: LOOKUPCALENDAR(file,reminderField,date,dataField,separator)

Description: The `lookupcalendar()` function builds a text array containing one item for every record in the target database where the date in the `reminderField` matches the date. Each item in the text array contains the value extracted from the `dataField` for that record.

Parameters: This function has five parameters: `file`, `reminderField`, `date`, `dataField` and `separator`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use [info\("database"\)](#).

reminderField is the name of the field that you want to search in. This field must contain valid reminders (see [reminder data](#)). The field must be in the database specified by the first parameter.

date is the actual date that you want to match. For example if you want to look up all appointments on July 23rd, this should be `date("July 23")`. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve appointment information, this should be the name of the field that contains that information. This must be a field in the database specified by the first parameter.

separator is the separator character for the text array you are building (see [text arrays](#)).

Result: The function returns a text array from all the records where the `reminderField` and `date` match.

Examples: This example builds a list of today's reminders.

```
todayMessages=lookupcalendar("Reminders",When,today(),Message,¶)
```

Errors: **Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also: [lookupall\(\)](#) function
[lookupptime\(\)](#) function
[lookupprtypes\(\)](#) function
[reminder data](#)
[reminder\(\)](#) function
[reminderdate\(\)](#) function
[remindercompare\(\)](#) function

LOOKUPLAST(...)

Syntax: LOOKUPLAST(file,keyField,keyValue,dataField,default,level)

Description: The `lookuplast()` function searches a database for a value, then returns other information from the same record. For example, the `lookuplast()` function can look up a phone number given a customer name, or look up a price given a part number. Unlike the `lookup()` function which searches from the top of the database, the `lookuplast()` function searches backwards from the bottom.

Parameters: This function has six parameters: `file`, `keyField`, `keyValue`, `dataField`, `default` and `level`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("databaseName")`.

keyField is the name of the field that you want to search in. For example if you want to look up a customer by name, this should be the field that contains customer names. The field must be in the database specified by the first parameter.

keyData is the actual data that you want to search for. For example if you want to look up a customer by name, this should be the actual name of the customer. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve a phone number, this should be the name of the field that contains phone numbers. This must be a field in the database specified by the first parameter.

default is the value you want this function to return if it is unable to find the information specified by the `keyField` and `keyData` parameters. The data type of the default value should match the data type of the `dataField`. If the `dataField` is numeric, the default should usually be zero. If the `dataField` is text, the default should usually be "".

level is the minimum summary level to be searched. Usually this parameter is zero so that the entire database will be searched. If the level is set to 1 through 7, only summary records will be searched.

Result: If the function is able to locate the information specified by the `keyField` and `keyData` parameters it returns the contents of the specified field in the specified database. (If there is more than one match, only the last one will be returned. However, if you are searching through the current database, the `lookuplast()` function will skip the current record, even if it is the last matching record in the database.) If it cannot locate the information it returns the default value.

Examples: This `lookuplast()` example finds the most recent order for a given customer, and the amount of that order.

```
local theCustomer,lastOrderDate,lastOrderAmount
theCustomer=""
gettext "Customer name:",theCustomer
lastOrderDate=
lookuplast("Invoice",Company,theCustomer,Date,0,0)
if lastOrderDate=0
    message "No previous invoices for this customer."
    stop
endif
lastOrderAmount=
```

```
lookuplast("Invoice",Company,theCustomer>Total,0,0)
message theCompany+"s most recent order was for "+
pattern(lastOrderAmount,"$#,.#")+" on "+
datepattern(lastOrderDate,"Month ddnth, yyyy")+". "
```

Here's another example that combines [lookup\(\)](#) and [lookuplast\(\)](#). This example first tries to look up a customer in the Customers database. If they are not found there, it checks to see if there is a previous invoice for this customer.

```
Address=
lookup("Customers",Company,Company,Address,"",0)
if Address=""
    City=lookup("Customers",Company,Company,City,"",0)
    State=lookup("Customers",Company,Company,State,"",0)
    Zip=lookup("Customers",Company,Company,Zip,"",0) else
    Address=lookuplast(info("databasename"),Company,Company,Address,"",0)
    City=lookuplast(info("databasename"),Company,Company,City,"",0)
    State=lookuplast(info("databasename"),Company,Company,State,"",0)
    Zip=lookuplast(info("databasename"),Company,Company,Zip,"",0)
endif
```

Remember, the [lookuplast\(\)](#) function locates the matching information that is physically closest to the bottom of the database. What this proximity to the bottom means depends on how the database is sorted.

Errors:

Database does not exist. This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookup\(\)](#) function
[lookupselected\(\)](#) function
[lookuplastselected\(\)](#) function
[table\(\)](#) function
[grabdata\(\)](#) function
[lookupall\(\)](#) function

LOOKUPLASTSELECTED(...)

Syntax: LOOKUPLASTSELECTED(file,keyField,keyValue,dataField,default,level)

Description: The `lookuplastselected()` function searches a database for a value, then returns other information from the same record. For example, the `lookuplast()` function can look up a phone number given a customer name, or look up a price given a part number. Unlike the `lookup()` function which searches from the top of the database, the `lookuplastselected()` function searches backwards from the bottom. The `lookuplastselected()` function only searches selected records, it ignores unselected records.

Parameters: This function has six parameters: `file`, `keyField`, `keyValue`, `dataField`, `default` and `level`.
file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("database")`.

keyField is the name of the field that you want to search in. For example if you want to look up a customer by name, this should be the field that contains customer names. The field must be in the database specified by the first parameter.

keyData is the actual data that you want to search for. For example if you want to look up a customer by name, this should be the actual name of the customer. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve a phone number, this should be the name of the field that contains phone numbers. This must be a field in the database specified by the first parameter.

default is the value you want this function to return if it is unable to find the information specified by the `keyField` and `keyData` parameters. The data type of the default value should match the data type of the `dataField`. If the `dataField` is numeric, the default should usually be zero. If the `dataField` is text, the default should usually be "".

level is the minimum summary level to be searched. Usually this parameter is zero so that the entire database will be searched. If the level is set to 1 through 7, only summary records will be searched.

Result: If the function is able to locate the information specified by the `keyField` and `keyData` parameters it returns the contents of the specified field in the specified database. (If there is more than one match, only the last one will be returned. However, if you are searching through the current database, the `lookuplastselected()` function will skip the current record, even if it is the last matching record in the database.) If it cannot locate the information it returns the default value.

Examples: This `lookuplastselected()` example finds the most recent paid order for a given customer, and the amount of that order.

```

local theCustomer,lastOrderDate,lastOrderAmount
local wasWindow
wasWindow=info("windowname")
openfile "Invoice"
select Paid="Yes"
window wasWindow
theCustomer=""
gettext "Customer name:",theCustomer
lastOrderDate=

```

```
lookuplast("Invoice",Company,theCustomer,Date,0,0)
if lastOrderDate=0
    message "No previous invoices for this customer."
    stop
endif
lastOrderAmount=
lookuplastselected(
    "Invoice",
    Company,
    theCustomer,
    Total,0,0)
message theCompany+"'s most recent order was for "+
pattern(lastOrderAmount,"$#,##")+ " on "+
datepattern(lastOrderDate,"Month ddnth, yyyy")+ "."
```

Errors:

Database does not exist. This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookup\(](#) function
[lookuplast\(](#) function
[lookupselected\(](#) function
[table\(](#) function
[grabdata\(](#) function
[lookupall\(](#) function

LOOKUPRTIME(...)

Syntax: LOOKUPRTIME(file,reminderField,date,pattern,separator)

Description: The `lookuprtime()` function builds a text array containing one item for every record in the target database where the date in the `reminderField` matches the date. Each item in the text array contains the time of the corresponding reminder.

Parameters: This function has five parameters: `file`, `reminderField`, `date`, `pattern` and `separator`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("database")`.

reminderField is the name of the field that you want to search in. This field must contain valid reminders (see [reminder data](#)). The field must be in the database specified by the first parameter.

date is the actual date that you want to match. For example if you want to look up the times for all appointments on July 23rd, this should be `date("July 23")`. This parameter is often a field in the current database.

pattern is the pattern you want to use to format the time. See the `timepattern()` function.

separator is the separator character for the text array you are building (see [text arrays](#)).

Result: The function returns a text array from all the records where the `reminderField` and `date` match. The text array contains the times for the reminders that match.

Examples: The example below will fill the global variable `Agenda` with the items on today's schedule.

```
global Agenda
local todayTimes,todayMessages
todayTimes=
lookuprtime("Reminders",When,today(),"hh:mm am/pm",¶)
todayMessages=
lookupcalendar("Reminders",When,today(),Message,¶)
arrayfilter Agenda,todayTimes,¶,
import()+" - "+array(todayMessages,seq()-1,¶)
```

The list of items in `Agenda` will be formatted something like this:

```
7:00 am - Breakfast meeting with Bob
9:25 am - Make sure Williams got our quote
1:30 pm - Late lunch with Jennings group
3:45 pm - get prepped for staff meeting
4:00 pm - Weekly staff meeting
5:30 pm - Don't forget flowers for Pat
```

This list can easily be displayed with an auto-wrap text object, a `Text Display SuperObject`, or a `List SuperObject`.

Errors: **Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookupall\(\)](#) function
[lookupcalendar\(\)](#) function
[lookuprtypes\(\)](#) function
[reminder data](#)
[reminder\(\)](#) function
[reminderdate\(\)](#) function
[remindercompare\(\)](#) function

LOOKUPRTYPES(...)

Syntax: LOOKUPRTYPES(file,reminderField,date,pattern,separator)

Description: The `lookuprtypes()` function builds a text array containing one item for every record in the target database where the date in the `reminderField` matches the date. Each item in the text array contains the type of the corresponding reminder, either "a" (appointment) or "t" (to-do).

Parameters: This function has five parameters: `file`, `reminderField`, `date`, `pattern` and `separator`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use [info\("database"\)](#).

reminderField is the name of the field that you want to search in. This field must contain valid reminders (see [reminder data](#)). The field must be in the database specified by the first parameter.

date is the actual date that you want to match. For example if you want to look up the times for all appointments on July 23rd, this should be `date("July 23")`. This parameter is often a field in the current database.

pattern is the pattern you want to use to format the time. This parameter is not used, and should be "".

separator is the separator character for the text array you are building (see [text arrays](#)). If this parameter is "" there is no separator, and the result of this function will be something like "aatatta".

Result: The function returns a text array from all the records where the `reminderField` and `date` match. Each element of the text array contains either "a" or "t" for the reminders that match.

Examples: The example below will fill the global variable `Agenda` with the items on today's schedule.

```
global Agenda
local todayTimes, todayMessages, todayTypes
todayTimes=
lookuprtime("Reminders",When,today(),"hh:mm am/pm",¶)
todayTypes=
lookuprtypes("Reminders",When,today(),"",¶)
todayMessages=
lookupcalendar("Reminders",When,today(),Message,¶)
arrayfilter todayTypes,todayTypes,¶
replace( replace( import(),"t","Todo:"),"a","Appt:")
arrayfilter Agenda,todayTimes,¶,
array(todayTypes,seq()-1,¶)+
import()+" - "+array(todayMessages,seq()-1,¶)
```

The list of items in `Agenda` will be formatted something like this:


```
Appt: 7:00 am - Breakfast meeting with Bob
Todo: 9:25 am - Make sure Williams got our quote
Appt: 1:30 pm - Late lunch with Jennings group
Appt: 3:45 pm - get prepped for staff meeting
Appt: 4:00 pm - Weekly staff meeting
Todo: 5:30 pm - Don't forget flowers for Pat
```

This list can easily be displayed with an auto-wrap text object, a Text Display SuperObject, or a List SuperObject.

Errors:

Database does not exist. This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookupall\(\)](#) function
[lookupcalendar\(\)](#) function
[lookupptime\(\)](#) function
[reminder data](#)
[reminder\(\)](#) function
[reminderdate\(\)](#) function
[remindercompare\(\)](#) function

LOOKUPSELECTED(...)

Syntax: LOOKUPSELECTED(file,keyField,keyValue,dataField,default,level)

Description: The `lookupselected()` function searches a database for a value, then returns other information from the same record. For example, the `lookupselected()` function can look up a phone number given a customer name, or look up a price given a part number. Unlike the `lookup()` function which searches all records in the database, the `lookupselected()` function only searches through selected records.

Parameters: This function has six parameters: `file`, `keyField`, `keyValue`, `dataField`, `default` and `level`.

file is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("database")`.

keyField is the name of the field that you want to search in. For example if you want to look up a customer by name, this should be the field that contains customer names. The field must be in the database specified by the first parameter.

keyData is the actual data that you want to search for. For example if you want to look up a customer by name, this should be the actual name of the customer. This parameter is often a field in the current database.

dataField is the name of the field that you want to retrieve data from. For example if you want to retrieve a phone number, this should be the name of the field that contains phone numbers. This must be a field in the database specified by the first parameter.

default is the value you want this function to return if it is unable to find the information specified by the `keyField` and `keyData` parameters. The data type of the default value should match the data type of the `dataField`. If the `dataField` is numeric, the default should usually be zero. If the `dataField` is text, the default should usually be "".

level is the minimum summary level to be searched. Usually this parameter is zero so that the entire database will be searched. If the level is set to 1 through 7, only summary records will be searched.

Result: If the function is able to locate the information specified by the `keyField` and `keyData` parameters in a selected record it returns the contents of the specified field in the specified database. (If there is more than one match, only the last one will be returned.) If it cannot locate the information it returns the default value.

Examples: This example looks up a price from a Price List database. This price list database has multiple discount levels. The procedure first selects distributor pricing, then looks up the price. The `ItemΩ` line item field in the invoice database contains the catalog item number, which should match a value in the `Catalog#` field in the Price List.

```

window "Price List"
select «Discount Level»="Distributor"
window "Invoice"
PriceΩ=
lookupselected("Price List",«Catalog#»,ItemΩ,Price,0,0)

```

Errors: **Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

Field or variable does not exist. This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

See Also:

[lookup\(\)](#) function

[lookuplast\(\)](#) function

[lookuplastselected\(\)](#) function

[table\(\)](#) function

[grabdata\(\)](#) function

[lookupall\(\)](#) function

LOOP

- Syntax:** LOOP
- Description:** The **loop** statement is used at the beginning of a loop. A loop is a sequence of statements that are executed over and over again. The end of the loop is always an [until](#) or [while](#) statement.
- Parameters:** This statement has no parameters.
- Action:** Loops are one of the fundamental building blocks of programming. In Panorama all loops begin with a loop statement and end with either an [until](#) or [while](#) statement. The statements in between the top and bottom of the loop are said to be “inside the loop.” These are the statements that will be repeated over and over again. Although it is not required, your procedures will usually be easier to read and understand if the statements inside the loop are indented. It is possible to put one loop inside of another. This is called a “nested loop.”
- Examples:** This simple example adds 10 new records to the current database.

```
loop
  addrecord
until 10
```

This example prints all unprinted records using the appropriate form. In this case the loop can only stop at the top (because of the `stoploopif`), because this **while** is forever!

```
find PrintedStatus=""
loop
  stoploopif (not info("found"))
  openform PrintForm
  print ""
  PrintedStatus="Complete"
  closewindow
  next
while forever
```

- Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.
- See Also:** [while](#) statement
[until](#) statement
[stoploopif](#) statement
[repeatloopif](#) statement

LOWER(...)

Syntax: LOWER(text)

Description: The lower(function converts text to all lower case (no capitalized letters).

Parameters: This function has one parameter: text.
text is the item of text that you want to force to all lower case.

Result: The result of this function is always a text item.

Examples: This function can be used to modify fields or variables, or to display data. This example makes sure that every payment method was lower case, i.e. visa not VISA or Visa.

```
field «Payment Method»
formulafill lower(«Payment Method»)
```

For example, you might use this procedure after you imported data that was not properly capitalized.

Another handy use for this function is to make comparisons when you don't know how the data is capitalized. The procedure below will select all data where the terms are NET 30, Net 30, or net 30.

```
select lower(Terms)="net 30"
```

The table below shows how the lower(function affects various items of text:

Formula	Result
lower("John Smith")	john smith
lower("NET 30")	net 30
lower("NEW York")	new york

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `lower(34)`. If you have a number you must convert the number to text before using it with this function, for example `lower(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

See Also: [upper\(function](#)
[upperword\(function](#)

MAGICFORMWINDOW

Syntax: MAGICFORMWINDOW database,form

Description: The **magicformwindow** statement designates an open window as the temporary active window for the purposes of info(functions and graphic statements. It does not change the actual active (frontmost) window, and does not affect database operations like data entry, sorting and searching.

Parameters: This statement has two parameters: database and form.

database is the database that contains the window you want to designate as the "magic window." This database must be open (in memory).

form is the form corresponding to the window you want to designate as the "magic window." This statement allows the window name to be different than the form name (for example if you have used the [windowname](#) statement).

Action: This statement designates a window as the temporary active window.

Examples: This procedure re-fills the [People List](#) list in the List form in the [Contacts](#) database. The form must be open (this is checked by the [info\("magicwindow"\)](#) function).

```
magicformwindow "Contacts", "List"
if info("magicwindow")<>" "
    superobject "People List", "Fill List"
    magicwindow " "
endif
```

This example finds out the current dimensions (location and size) of the [Plain](#) form in the [Checkbook](#) database. This example assumes that the form is already open.

```
magicformwindow "Checkbook", "Plain"
windowDimensions=info("windowrectangle")
magicwindow " "
```

Views: This statement may be used in a procedure run from any view.

See Also: [magicwindow](#) statement
[info\("magicwindow"\)](#) function

MAGICWINDOW

Syntax: MAGICWINDOW window

Description: The `magicwindow` statement designates an open window as the temporary active window for the purposes of `info()` functions and graphic statements. It does not change the actual active (frontmost) window, and does not affect database operations like data entry, sorting and searching.

Parameters: This statement has one parameter: `window`.

window is the window that you want to designate as the "magic" active window. This must be a window that is currently open. If `window` is "" then the "magic" window designation is cancelled.

Action: This statement designates a window as the temporary active window.

Examples: This procedure re-fills the **People List** list in the **List** form in the **Contacts** database. The form must be open (this is checked by the `info("magicwindow")` function).

```
magicwindow "Contacts:List"
if info("magicwindow")<>" "
  superobject "People List","Fill List"
  magicwindow " "
endif
```

This example finds out the current dimensions (location and size) of the **Plain** form in the **Checkbook** database. This example assumes that the form is already open.

```
magicwindow "Checkbook:Plain"
windowDimensions=info("windowrectangle")
magicwindow " "
```

Views: This statement may be used in a procedure run from any view.

See Also: [magicformwindow](#) statement
[info\("magicwindow"\)](#) function

MAGNIFICATION

Syntax: MAGNIFICATION scale

Description: The **magnification** statement changes the magnification of the current form, zooming in or out.

Parameters: This statement has one parameter: scale

scale is a text item that tells the magnification statement what enlargement factor to use for the form. The options are:

```
25%  
50%  
100%  
2X  
4X  
8X
```

(Note: The **X** in the last three options must be upper case.)

Action: This statement allows a procedure to zoom in and out in a form. For example, a procedure can get a crude preview by zooming out to 25%. To scroll the form to different locations use the [formxy](#) statement.

Examples: This example opens the **Letter** form and previews it at 25%. The preview may not be exact because text objects often do not scale completely accurately.

```
openform "Letter"  
magnification "25%"
```

Views: This statement may be used in the Form view.

See Also: [formxy](#) statement

MAKEFOLDER

Syntax: MAKEFOLDER folderpath

Description: The **makefolder** statement creates a new folder.

Parameters: This statement has one parameter: folderpath

folderpath is the complete path of the folder you want to create. For example, suppose your hard disk is called **Disk**, and you want to create a new folder named **Obsolete** in the System folder. In that case, the folderpath would be **Disk:System:Obsolete**.

Action: Use the **makefolder** statement when you need to create a new folder. Important note: The **makefolder** statement can only create one folder at a time. In other words, if you ask it to create the **Disk:Plans:ProjectX** folder, the **Plans** folder must already exist.

Examples: This example is a general purpose subroutine that can be used to create new folders. This subroutine will create any folders needed. If you gave this procedure the name **.MakeFolder** you could call it like this

```
call .MakeFolder,"Disk:Plans:1999:ProjectZ"
```

If the **Plans** or **1999** folders don't exist, the subroutine will create them, along with the **ProjectZ** folder. Here is the full source of this useful subroutine.

```
/*
* make a folder, and all nested folders
*
*parameter(1)=folderpath
*for example "Alaska:Alpha Folder:Gamma Folder:Zed Quadrant:Foo"
*if error set parameter(1) to ""
*/
local newfolder,targetfolder,tempfolder,tempfile,tftype,depth,vSep
if folderpath( info("panoramafolder") ) [2,3]=":\\"
    vSep="\\"
else
    vSep=":"
endif
newfolder=parameter(1)
targetfolder=newfolder
shortcut testtarget
if tftype contains "folder" rtn endif /* folder already exists */
if tftype contains "file"
    setparameter 1,"" /* can't create folder with same name as a file */
    rtn
endif

depth=1
loop
    targetfolder=arrayrange(newfolder,1,depth+1,vSep)
    shortcut testtarget
    if tftype=""
        makefolder targetfolder
    endif
    depth=depth+1
while newfolder#targetfolder
rtn

testtarget:
tempfolder=
```

```
    arrayrange(targetfolder,1,-1+arraysize(targetfolder,vSep),vSep)
tempfile=array(targetfolder,arraysize(targetfolder,vSep),vSep)
tftype=array( fileinfo( folder(tempfolder),tempfile),1,¶)
if tftype contains "file"
    setparameter 1,""
    rtn
endif
rtn
```

Views: This statement may be used in any view.

See Also: [folder](#)(function
[folderpath](#)(function

MAKESECRET

Syntax: MAKESECRET

Description: The `makesecret` statement makes the current database disappear. All of the windows for this database will be closed. However, the database is still in memory, so the database can still be used for lookups, etc.

Parameters: This statement has no parameters.

Examples: This procedure makes the price list database invisible. (You can also make a database invisible by checking the **No Windows** option in the **Save As** dialog.)

```
window "Price List"
makesecret
```

Even though the price list database is invisible, you can still access information in the file.

```
myPrice=lookup("Price List",Description,Item,Price,0,0)
```

To modify information in a hidden database you must temporarily create a secret window. This can be done with the `window` statement. This example adjusts the quantity of an item in an invisible inventory file.

```
local wasWindow
wasWindow=info("windowname")
window "Inventory:Secret"
find Description=grabdata("Invoice",InvoiceItem)
OnHandQty=OnHandQty-grabdata("Invoice",InvoiceQt)
window wasWindow
```

To make the Price list database visible again use this procedure. (This assumes that the **No Windows** option is not checked in the **Save As** dialog.)

```
openfile "Price List"
```

To open a specific window in an invisible database you must first create a temporary secret window, then open the window you really want.

```
window "Price List:Secret"
setwindowrectangle rectanglesize(20,20,300,180)
openform "Distributor"
```

Views: This statement may be used in any view.

See Also: [window](#) statement
[opensecret](#) statement
[info\("files"\)](#) function

MAX(...)

Syntax: MAX(value1,value2)

Description: The `max()` function compares two values and returns the larger value.

Parameters: This function has two parameters: value1 and value2.

value1 is the first value you want to compare. This must be a number, not text.

value2 is the second value you want to compare. This must be a number, not text.

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This example calculates the hottest city, LA or NY.

```
max(LAtemp, NYtemp)
```

If you need to calculate the maximum of three or more values you can nest multiple `max()` functions together like this.

```
max(LAtemp, max(ChicagoTemp, NYtemp))
```

Temperature must contain a numeric value. The table below shows how the `max()` function works with some typical values. The first column shows the result.

max(value 1	value 2
98	98	37
604	154	604
-3	-3	-4
1	1	-2264

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `max("Sue", "Bob")`. If you have a number in a text item you must convert the text to a numeric value before taking the absolute value, for example `max(val("34"), val("876"))`.

See Also: [val\(\)](#) function

MAXIMUM

Syntax: MAXIMUM

Description: The maximum statement calculates the maximum and submaximums for the current field.

Parameters: This statement has no parameters.

Action: This statement calculates the maximum for the current field. The current field may be numeric, text, or a date field. If the database contains summary records, this statement will calculate submaximum for each summary record, along with an overall maximum for the whole database. If there are not any summary records in the database, one will be added at the end of the database and the overall maximum calculated and placed into the summary record. This statement has the same effect as choosing the **Maximum** command in the Math menu.

Examples: This simple example calculates the largest check in the database.

```
field Debit  
maximum
```

This example calculates the largest sale for each state, along with the overall largest sale.

```
field State  
group  
field Sales  
total
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [total](#) statement
[sum\(\)](#) function
[count](#) statement
[average](#) statement
[minimum](#) statement
[group](#) statement
[outlinelevel](#) statement

MENUBUILD

Syntax: MENUBUILD menu,list

Description: The **menubuild** statement rebuilds a custom menu on the fly. The menu must already exist in an open resource file, because this command cannot create a menu completely from scratch.

Parameters: This statement has two parameters: **menu** and **list**.

menu is the name or ID number of the menu that is to be rebuilt. The menu ID is assigned in ResEdit.

list is a text array that contains a semicolon separated list of the menu items. For example, the list below will build a menu of car rental companies:

```
"Alamo;Avis;Budget;Dollar;Hertz;National"
```

Menu items are normally displayed in plain Chicago 12 point text. By adding a special suffix to menu item names in the menu item list you can change menu items to different styles: bold, italic, etc. The table below lists the different styles and corresponding suffixes.

Style	Suffix	Example
Bold	<B	Monthly Report<B
Italic	<I	New Invoice<I
Underline	<U	Initialize Payroll<U
Outline	<O	Back Order<O
Shadow	<S	Erase All Statements<S

It is also possible to assign a command key equivalent by adding a suffix. The suffix consists of a / character followed by the character you want to assign as a command key equivalent. The list below defines five menu items with command key equivalents from 1 thru 5.

```
"UPS/1;US Mail/2;FedEx/3;DHL/4;Airborne/5"
```

You can build separator lines in the custom menu by defining a menu item as (-. The list below would build a menu with two separator lines, one between White and Red, and another between Blue and Orange.

```
"Black;White;(-;Red;Green;Blue;(-;Orange;Yellow;Brown"
```

Action: This statement allows a procedure to completely change the contents of an entire menu. The old menu is completely replaced by the new menu items you define.

Examples: This example builds a custom menu named **Customer**. (Remember, this custom menu must be already created in the resource file.) The procedure builds the menu with a list of all the customers in the database.

```
local customerArray  
arraybuild customerArray,";",",",Customer  
arraydeduplicate Customer,Customer,";"  
menubuild "Customer",customerArray
```

Views: This statement may be used in any view that has custom menus installed.

See Also: [menudisable](#) statement
[menuenable](#) statement
[getmenumark](#) statement
[setmenumark](#) statement
[clearmenumarks](#) statement
[getmenutext](#) statement
[setmenutext](#) statement
[getmenus](#) statement
[setmenus](#) statement

MENUDISABLE

Syntax: MENUDISABLE menu,item

Description: The **menudisable** statement disables a custom menu item. The menu item will turn gray and can no longer be selected.

Parameters: This statement has two parameters: menu and item.

menu is the name or ID number of the menu that contains the item to be disabled. The menu ID is assigned in ResEdit.

item is the name of the menu item, or the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu is **Cleared**. This menu item may be specified as either "Cleared" or 3.

Action: A menu item may be disabled when it is not appropriate. For example, in an accounting database the **Void Transaction** menu item might be disabled after an invoice is posted. To show that the item is disabled, the text of the item turns gray, for example Void Transaction. The **menudisable** statement disables a menu item. The [menuenable](#) statement turns the menu item on again. (Note: **Only custom menus can be enabled and disabled.** A procedure cannot enable or disable one of Panorama's standard menus or an item in the **Action** menu.)

Examples: Suppose a database has a **Rush** menu item in the **Order** custom menu. This menu item is not valid if the shipping method is US Mail. The example below could be part of the **.CurrentRecord** procedure, and handles enabling/disabling the **Rush** menu item (**Rush/Rush**).

```
if ShipMethod="US Mail"
    menudisable "Order", "Rush"
else
    menuenable "Order", "Rush"
endif
```

Views: This statement may be used in any view that has custom menus installed.

See Also: [menuenable](#) statement
[getmenumark](#) statement
[setmenumark](#) statement
[clearmenumarks](#) statement
[getmenutext](#) statement
[setmenutext](#) statement
[menubuild](#) statement
[getmenus](#) statement
[setmenus](#) statement

MENUENABLE

Syntax: MENUENABLE menu,item

Description: The **menuenable** statement enables a custom menu item. The menu item will turn black and can be selected. (See also [menudisable](#).)

Parameters: This statement has two parameters: menu and item.

menu is the name or ID number of the menu that contains the item to be enabled. The menu ID is assigned in ResEdit.

item is the name of the menu item, or the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu is **Cleared**. This menu item may be specified as either "Cleared" or 3.

Action: A menu item may be disabled when it is not appropriate. For example, in an accounting database the **Void Transaction** menu item might be disabled after an invoice is posted. To show that the item is disabled, the text of the item turns gray, for example Void Transaction

The [menudisable](#) statement disables a menu item. The **menuenable** statement turns the menu item on again. (Note: **Only custom menus can be enabled and disabled**. A procedure cannot enable or disable one of Panorama's standard menus or an item in the **Action** menu.)

Examples: Suppose a database has a Rush menu item in the Order custom menu. This menu item is not valid if the shipping method is US Mail. The example below could be part of the .CurrentRecord procedure, and handles enabling/disabling the Rush menu item (Rush/Rush).

```
if ShipMethod="US Mail"
  menudisable "Order", "Rush"
else
  menuenable "Order", "Rush"
endif
```

Views: This statement may be used in any view that has custom menus installed.

See Also: [menudisable](#) statement
[getmenumark](#) statement
[setmenumark](#) statement
[clearmenumarks](#) statement
[getmenutext](#) statement
[setmenutext](#) statement
[menubuild](#) statement
[getmenus](#) statement
[setmenus](#) statement

MESSAGE

Syntax: MESSAGE message

Description: The **message** statement displays a message. The message is displayed in an alert with one button: Ok. The message stays on the screen until the user presses the Ok button, then the procedure continues with the next statement.

Parameters: This statement has one parameter message.

message is the text that is to be displayed. You may use any formula to create the text, but the text may not be more than 255 characters long. (The standard message alert, however, is only big enough to show about 160 characters. You can enlarge the alert with the **customalert** statement.)

Action: Use the **message** statement when you need to temporarily display an important message. Don't overdo it, because the message alert can be very annoying if used too often. The **message** statement doesn't give the user any choices. If you need to display a message and give the user a choice use the [alert](#), [okcancel](#), [cancelok](#), [yesno](#) or [noyes](#) statements.

Examples: This example uses the message statement to inform the user that the requested operation cannot be performed.

```
if Transaction="Deposit"
    message "Sorry, deposits cannot be deleted."
else
    deleterecord
endif
```

This example uses a formula to build a complex message. The message will be something like The database contains 12 deposits for a total of \$3,932.67 dollars.

```
local quickTotal,quickCount
formulasum quickCount,?(Transaction="Deposit",1,0)
formulasum quickTotal,?(Transaction="Deposit",Amount,0)
message "The database contains "+
    pattern(quickCount,"# deposit~")+
    " for a total of "+pattern(quickTotal,"$#,## dollar~")
```

Views: This statement may be used in any view.

See Also: [customalert](#) statement
[yesno](#) statement
[noyes](#) statement
[okcancel](#) statement
[cancelok](#) statement
[gettext](#) statement
[getscrap](#) statement
[getscrapok](#) statement
[alertmode](#) statement

MIN(...)

Syntax: MIN(value1,value2)

Description: The `min()` function compares two values and returns the smaller value.

Parameters: This function has two parameters: value1 and value2.

value1 is the first value you want to compare. This must be a number, not text.

value2 is the second value you want to compare. This must be a number, not text.

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This example calculates the coldest city, LA or NY.

```
min(LAtemp,NYtemp)
```

If you need to calculate the minimum of three or more values you can nest multiple `min()` functions together like this.

```
min(LAtemp,min(ChicagoTemp,NYtemp))
```

Temperature must contain a numeric value. The table below shows how the `min()` function works with some typical values. The first column shows the result.

min(value 1	value 2
37	98	37
154	604	154
-4	-3	-4
-2264	1	-2264

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text values with this function, for example `min("Bob","Sue")`. If you have a number in a text item you must convert the text to a numeric value before taking the minimum value, for example `min(val("34"),val("576"))`.

See Also: [max\(\)](#) function

MINIMUM

Syntax: MINIMUM

Description: The **minimum** statement calculates the minimum and subminimums for the current field.

Parameters: This statement has no parameters.

Action: This statement calculates the minimum for the current field. The current field may be numeric, text, or a date field. If the database contains summary records, this statement will calculate subminimum for each summary record, along with an overall minimum for the whole database. If there are not any summary records in the database, one will be added at the end of the database and the overall minimum calculated and placed into the summary record. This statement has the same effect as choosing the **Minimum** command in the **Math** menu.

This statement ignores empty cells. For example if a numeric cell is empty, it cannot be the minimum value. However, if the cell contains zero, it can be the minimum value (unless there are negative values).

Examples: This simple example calculates the smallest check in the database.

```
field Debit
minimum
```

This example fills the summary record with the name of the first company in each state.

```
field State
group
field Company
minimum
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [total](#) statement
[sum\(\)](#) function
[count](#) statement
[average](#) statement
[maximum](#) statement
[group](#) statement
[outlinelevel](#) statement

MONTH1ST(...)

Syntax: MONTH1ST(date)

Description: The `month1st` function computes the first day of a month.

Parameters: This function has one parameter: `date`.
`date` is a number representing the date.

Result: This function calculates the first day of the month. For example, if the date passed to this function is **October 18, 1997**, this function will return the date **October 1, 1997**. The date is returned as a number.

Examples: The example below selects the orders placed this month, then displays the count.

```
select
OrderDate>month1st(today()) and
OrderDate<month1st(today()+monthlength( today()))
message str( info("records") )+" orders this month"
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

See Also: [monthlength](#)(function
[monthmath](#)(function
[week1st](#)(function
[year1st](#)(function
[date](#)(function
[datepattern](#)(function

MONTHLENGTH(...)

Syntax: MONTHLENGTH(date)

Description: The `monthlength()` function computes the length (number of days) of a month

Parameters: This function has one parameter: `date`.
`date` is a number representing the date.

Result: The `dayofweek()` function calculates the number of days in any month. For example, if the date passed to this function is **October 18, 1997**, this function will return the **31**, the number of days in October. This function knows about leap years and adjusts the length of February accordingly.

Examples: The example below selects the orders placed this month, then displays the count.

```
select
  OrderDate > month1st( today() ) and
  OrderDate < month1st( today() ) + monthlength( today() )
message str( info("records") ) + " orders this month"
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

See Also: [month1st\(\)](#) function
[monthmath\(\)](#) function
[week1st\(\)](#) function
[year1st\(\)](#) function
[date\(\)](#) function
[datepattern\(\)](#) function

MONTHMATH(...)

Syntax: MONTHMATH(date,offset)

Description: The **monthmath**(function takes a date and computes another date that is one or more months before or after the original date.

Parameters: This function has two parameters: **date** and **offset**.

date is a number representing the original date.

offset is the number of months that you want to add or subtract to the original date.

Result: The **monthmath**(function offsets a date by a number of months. For example, if you offset the date **May 12, 1997** by two months the result is **July 12, 1997**. If you offset the same original date by minus two months the result is **March 12, 1997**.

If the new date does not exist because a month does not have enough days in it, the **monthmath**(function will pick the last day of the month. For example, if you offset **March 31** by 1 month the result is **April 30**. If the new month lands in February the function knows about leap years and adjusts accordingly.

Examples: This example calculates a renewal date exactly one year from today.

```
RenewalDate=monthmath(today(),12)
```

Errors: Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value for the date or offset parameters.

See Also: [month1st](#)(function
[monthlength](#)(function
[week1st](#)(function
[year1st](#)(function
[date](#)(function
[datepattern](#)(function

NEWDATABASE

Syntax: NEWDATABASE

Description: The `newdatabase` statement creates a new empty database. The new database is called “Untitled”, and contains one field, which is also called “Untitled.”

Parameters: This statement has no parameters.

Examples: This procedure makes a new empty database with five fields: Name, Address, City, State and Zip

```
newdatabase
  fieldname "Name"
  addfield "Address"
  addfield "City"
  addfield "State"
  addfield "Zip"
```

Views: This statement may be used in any view.

See Also: [openfile](#) statement
[fieldname](#) statement
[addfield](#) statement
[deletefield](#) statement
[fieldtype](#) statement

NEWFORM

Syntax: NEWFORM

Description: The **newform** statement creates a new form. The statement displays the standard new form dialog, which allows the user to type in the name for the new form. This statement is the same as choosing **New Form** from the View menu (the pop-up menu in the window title).

Parameters: This statement has no parameters.

Examples: This example creates a **New Form** command in your own custom **Form** menu. Here are the statements to use in your **.CustomMenu** procedure.

```
if info("trigger") beginswith "Menu.Form.New Form"  
    newform  
    stop  
endif
```

Views: This statement may be used in a Form view.

See Also: [openform](#) statement
[goform](#) statement
[dbinfo](#)("forms",...) function

NEWGENERATION

Syntax: NEWGENERATION

Description: The **newgeneration** statement updates the database configuration to match the design sheet. This statement is the same as choosing **New Generation** from the tool palette.

Parameters: This statement has no parameters

Examples: This example adds five new fields to the database.

```
opendesignsheet  
addrecord  
«Field Name»="Address"  
addrecord  
«Field Name»="City"  
addrecord  
«Field Name»="State"  
addrecord  
«Field Name»="Zip"  
newgeneration  
closewindow
```

Views: This statement may be used in the Design Sheet view.

See Also: [opendesignsheet](#) statement
[godesignsheet](#) statement

NEXT

- Syntax:** NEXT
- Description:** The NEXT statement locates the next visible record, if any, which matches the most recent [find](#) statement.
- Parameters:** This statement has no parameters.
- Action:** This statement is used to locate and make active the next record, starting from the currently active record, that complies with the last find statement. If a subset of records are selected from the database next will only examine that sub-set for a match. You do not need to have the cursor on the field you are performing the next on prior to executing the next.
- If no records meet the [find](#) criteria Panorama will leave the cursor where it was before the next statement executed.
- This statement has the same effect as selecting the **Find Next** option from the **Search** menu.
- Examples:** This example finds the first record for John Smith and then makes the next visible record, if any, for John Smith the active record.
- ```
find Customer = "John Smith"
next
```
- This example uses a formula to test multiple text fields to find the record where the variable's contents matches any one of the fields. It then asks if you wish to keep looking for additional records, one at a time, and makes each the active record if it exist.
- ```
local TheText
gettext "Enter text to find:",TheText
find " "+Customer+" "+Company+" "+Address+" "+
      Comments contains TheText
if (not info("found"))
  beep
  message "No matching records found."
  stop
endif
loop
  yesno "Keep Looking?"
  if clipboard() contains "yes"
    next
  endif
until clipboard() contains "no"
```
- Views:** This statement may be used in any view.
- See Also:** [find](#) statement
[formselect](#) statement
[formulafindselect](#) statement
[info\("empty"\)](#) function
[info\("found"\)](#) function
[info\("records"\)](#) function
[info\("selected"\)](#) function

select statement
selectadditional statement
selectall statement
selectreverse statement
selectsummaries statement
selectwithin statement

NODEFAULTEXTENSION

Syntax: NODEFAULTEXTENSION

Description: The `nodefaultextension` statement works with the [openfile](#) statement. On Macintosh systems the behavior of the [openfile](#) statement changes slightly if the current database name ends with `.pan`. In that case the [openfile](#) statement will automatically add `.pan` to any file name that doesn't already have an extension. This makes it easier to set up a set of database files that can work on both the Macintosh and the PC. However, this also means that you will not be able to open a database that doesn't have any extension at all with this statement (of course you can always open such a file manually using the **Open File** dialog in the File menu). By placing the `nodefaultextension` statement just before the [openfile](#) statement you can cancel this automatic action and allow any file to be opened. (This statement is ignored on Windows PC system, where database file always must end with `.pan`).

Parameters: This statement has no parameters.

Examples: This example opens the database `Contacts`, even if the current file name ends with `.pan`. Without the `nodefaultextension` statement the [openfile](#) would try to open `Contacts.pan` if the current file name ended with `.pan`.

```
nodefaultextension  
openfile "Contacts"
```

Views: This statement may be used in any view.

See Also: [openfile](#) statement

NOEDITSCROLL

Syntax: NOEDITSCROLL

Description: The `noeditscroll` statement disables the scroll bar in the pop-up editing window the next time that window is opened. It can be used before the [editcell](#) or [floatingedit](#) statements. (Note: This statement only works with standard data cells. It does not work with Super-Objects™.)

Parameters: This statement has no parameters.

Examples: This example allows the user to edit the Address data cell. Even if the Address data cell is more than 1" high, no scroll bar will appear.

```
field Address  
noeditscroll  
editcell
```

Views: This statement may be used in a Form view.

See Also: [editcell](#) statement
[floatingedit](#) statement

NOEVENT

Syntax: NOEVENT

Description: The `noevent` statement disables Panorama's event processing. It can be re-enabled with the `yesevent` statement.

Parameters: This statement has no parameters.

Action: This statement turns off Panorama's event processing. You should only use this statement in one place—at the beginning of the `.CustomMenu` procedure as shown in the example below. In this application the `noevent` statement allows AppleEvents to open a database properly via your `.CustomMenu` procedure. This is important because the Finder uses AppleEvents to open a file when you double click on its icon. **Using `noevent`** in any other way will probably cause Panorama to crash! No kidding.

Examples: If your database uses a custom menu for the File menu your `.CustomMenu` procedure should contain the following statements at the very beginning of the procedure. The `noevent` statement must be the very first statement in the procedure. This will allow the Finder to open files with AppleEvents.

```
noevent
if info("trigger") beginswith "Menu.File.Open"
  openfile dialog
  stop
endif
yesevent
```

Views: This statement may be used in any view.

See Also: [yesevent](#) statement
[openfile](#) statement

Non Decimal Numbers

Background: For raw data, programmers use numbers that are base 2 (binary), base 8 (octal), or base 16 (hexadecimal or just “hex”). All these number systems are convenient for working with raw data because they are powers of two. When using base 2, there are 2 digits (0 and 1) and each digit represents a power of 2. When using base 8, there are 8 digits (0, 1, 2, 3, 4, 5, 6, 7) and each digit represents a power of 8. When using base 16 there are 16 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) and each digit represents a power of 16. Hex numbers are most programmers favorite because each digit corresponds to four bits:

Digit	Bits	Digit	Bits
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

In hexadecimal, a byte is always 2 digits, a word is 4 digits, and a longword is 8 digits.

Radix: The number of digits a numeric system uses is called its **radix**. The radix of a number is often displayed as a subscript at the end of the number. For example 101_2 (binary) is the same value as 5_{10} (decimal). If you’ve never been exposed to non-decimal radix numbers before, this whole concept probably seems quite strange. However, for raw binary information non-decimal numbers (especially hex) are much easier to work with, and they are universally used by programmers from the largest mainframe to the smallest micro-computer.

Conversion: The table below shows the equivalent decimal, hex, octal and binary numbers from 0 to 32 (decimal).

Decimal	Hex	Octal	Binary
00	00	000	000000
01	01	001	000001
02	02	002	000010
03	03	003	000011
04	04	004	000100
05	05	005	000101
06	06	006	000110
07	07	007	000111
08	08	010	001000
09	09	011	001001
10	0A	012	001010
11	0B	013	001011
12	0C	014	001100
13	0D	015	001101
14	0E	016	001110
15	0F	017	001111
16	10	020	010000
17	11	021	010001
18	12	022	010010
19	13	023	010011
20	14	024	010100
21	15	025	010101
22	16	026	010110
23	17	027	010111
24	18	030	011000
25	19	031	011001
26	1A	032	011010
27	1B	033	011011
28	1C	034	011100
29	1D	035	011101
30	1E	036	011110
31	1F	037	011111
32	20	040	100000

Fortunately, Panorama has some built-in functions for converting numbers between different radix systems. The [radixstr\(\)](#) function (short for radix to string) converts a number (or some raw binary data) into binary, octal, or hexadecimal digits. The [radix\(\)](#) function converts text that spells out a number in binary, octal, or hex into a standard Panorama number.

See Also: [radix\(\)](#) function
[radixstr\(\)](#) function
[byte\(\)](#) function
[word\(\)](#) function
[longword\(\)](#) function
[binaryvalue\(\)](#) function

NOP

Syntax: NOP

Description: The **nop** statement does nothing (**no operation**). A procedure can use the **nop** statement as a placeholder or to delay for a short time.

Parameters: This statement has no parameters

Examples: This procedure will delay exactly 10 seconds.

```
local startTime  
startTime=now()  
loop  
    nop  
until now()>startTime+10
```

Another use for the **nop** statement is to fool Panorama into not displaying a warning dialog. When used as the last statement in a procedure, or just before a stop statement, statements like **quit** and **closefile** will ask the user if they want to save changes. By adding a **nop** statement you can prevent this alert from appearing.

```
if info("trigger") contains "Close w/o Save"  
    closefile  
    nop  
    stop  
endif
```

Views: This statement may be used in any view.

See Also: [closefile](#) statement
[quit](#) statement

NOSHOW

Syntax: NOSHOW

Description: The **noshow** statement temporarily disables the output of text and graphics. You should use this command in a procedure when you want to disable the display of intermediate steps.

Parameters: This statement has no parameters.

Action: This statement tells Panorama to suppress all output of text and graphics. Only output that is directly generated by Panorama will be suppressed. For example, the output generated by the Sort or [formulafill](#) commands will be suppressed. However, output that is generated by the operating system will not be suppressed. Primarily this means that when switching from one window to another the display will not be suppressed. Output will be disabled until the procedure is finished, or until an [endnoshow](#) statement is encountered.

Examples: Here is an example that performs several operations on the current database, but only updates the display once.

```
noshow
  field Date
  groupup by month
  field Category
  groupup
  field Amount
  total
  outlinelevel 2
  showpage
  endnoshow
```

Views: This statement should only be used when a form or data sheet is active.

See Also: [endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[nundo](#) statement
[nowatchcursor](#) statement
[watchcursor](#) statement

NOUNDO

Syntax: NOUNDO

Description: The **noundo** statement disables the undo statement until the end of the procedure. Some statements (particular statements from the **Fill** menu) run slightly faster when undo is disabled.

Parameters: This statement has no parameters.

Examples: This example calculates the running bank balance in a checkbook. The procedure will run slightly faster because of the **noundo** statement.

```
noundo
field Balance
formulafill Credit-Debit
runningtotal
```

Views: This statement may be used in any view.

See Also: [change](#) statement
[fill](#) statement
[formulafill](#) statement
[group](#) statement
[groupbycolor](#) statement
[groupup](#) statement
[groupdown](#) statement
[hide](#) statement
[propagate](#) statement
[propagateup](#) statement
[runningdifference](#) statement
[runningtotal](#) statement
[select](#) statement
[selectadditional](#) statement
[selectall](#) statement
[selectwithin](#) statement
[undo](#) statement
[unpropagate](#) statement
[unpropagateup](#) statement

NOW(...)

Syntax: NOW()

Description: The `now()` function returns the current time (number of seconds since midnight).

Parameters: This function has no parameters.

Result: This function calculates the current time. The time is a number (the number of seconds since midnight).

Examples: This example selects flights that will be arriving in the next 20 minutes.

```
select ArrivalTime>now() and ArrivalTime<=now()+20*60
```

This formula could be used in an auto-wrap text object or Text Display SuperObject™ to display the current time.

```
timepattern(today(),"HH:MM:SS AM/PM")
```

Errors: This function does not produce any errors.

See Also: [seconds\(\)](#) function
[time\(\)](#) function
[timepattern\(\)](#) function
[today\(\)](#) function
[info\("tickcount"\)](#) function

NOWATCHCURSOR

Syntax: NOWATCHCURSOR

Description: The `nowatchcursor` statement temporarily disables the watch cursor.

Parameters: This statement has no parameters.

Action: This statement tells Panorama to suppress the watch and pie cursors that Panorama normally uses when performing a potentially slow operation. Use this when you want to leave the arrow cursor while the procedure runs.

The watch and pie cursors will be disabled until the procedure is finished, or until a `watchcursor` hyperlink statement is encountered.

Examples: Here is an example that performs several operations on the current database, but only updates the display once. While these operations are performed the cursor stays as an arrow or cross instead of flipping into a watch.

```
noshow
nowatchcursor
field Date
groupup by month
field Category
groupup
field Amount
total
outlinelevel 2
showpage
endnoshow
```

Views: This statement may be used in any view.

See Also: [watchcursor](#) statement
[noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[noundo](#) statement

NOWINDOWSAVE

Syntax: NOWINDOWSAVE

Description: The `nowindowsave` statement tells Panorama NOT to save the new window positions the next time the database is saved. This statement is designed to be used just before the `save` or `saveall` statements. This is useful for templates that have specific window positions that you don't want to be disturbed. (Of course if your template runs in User or Custom mode the NoWindowSave command isn't really necessary, because window positions aren't saved anyway.)

Parameters: This statement has no parameters.

Examples: This example simulates the Save command in your own custom File menu. However, unlike the regular Save command, this one does not save the window positions, unless you hold down the OPTION key. Here are the statements to use in your `.CustomMenu` procedure.

```
if info("trigger") beginswith "Menu.File.Save"  
  if info("trigger") notcontains "option"  
    nowindowsave  
  endif  
  save  
  stop  
endif
```

Views: This statement may be used in any view

See Also: [save](#) statement
[saveall](#) statement

NOYES

- Syntax:** NOYES text
- Description:** The `noyes` statement displays an alert with a message and two buttons: **Yes** and **No**. The default is **No**.
- Parameters:** This statement has one parameter: text.
- `text` is the message that will appear in the dialog when it is displayed. You may use any formula to create this text, but usually a text constant is used (text surrounded by double quote marks (Example: "Do you want to continue?").
- Action:** This statement allows the procedure to pause and asks a question requiring one of two responses: **Yes** or **No**. The response will be written to the clipboard so that it can be tested for later in the procedure.
- Examples:** This example asks the user if they want to remove old records. If they press the Yes button the procedure will remove all records that are more than 1 year old, otherwise the procedure will do nothing.
- ```

noyes "Remove old records?"
if clipboard() contains "Yes"
 select Date>today()-365
 removeunselected
endif

```
- Views:** This statement may be used in any view, and also works when no windows are open at all.
- See Also:** [yesno](#) statement  
[alert](#) statement  
[cancelok](#) statement  
[customalert](#) statement  
[customdialog](#) statement  
[getscrap](#) statement  
[gettext](#) statement  
[message](#) statement  
[okcancel](#) statement  
[alertmode](#) statement  
[clipboard\(\)](#) function  
[info\("dialogtrigger"\)](#) function



# NUMERIC PATTERNS

**Background:** Numeric patterns allow you to control how a number is displayed or converted to text. All of the numbers listed below have the value 2654, but have been converted to text using different patterns:

2654, 2,654, \$2,654.00, 002654, 2.654e+3, 26-54

**Basics:** The basic building block of any numeric pattern is the # symbol. This symbol represents one or more digits of the number being converted. By arranging one or more # symbols with other characters and punctuation you can control the format of the number being displayed. The example below shows a very basic format for converting numbers to text. This format formats the number with two digits after the decimal point—no matter how many digits actually exist in the original number.

```
Text=pattern(Amount, "#.##")
```

**Rounding:** If the original number has too many digits after the decimal point (as does the number on the last line of the table above) the number will be rounded (not truncated) to fit into the formatted text (as the example shows).

**Leading Zeros:** If the pattern has extra # symbols in front of the decimal point the pattern( function will add leading zeros in front of the number.

```
Text=pattern(Amount, "#####.#")
```

As the last number shows, the pattern( function will never chop off digits in front of the decimal point. Even though 8911272 has 7 digits and the pattern only has 5 # symbols in front of the decimal point, all 7 digits are included in the converted value. The # symbols in front of the decimal point define a lower limit on the number of digits in front of the decimal point, not an upper limit. (If you want to establish an upper limit you could use a text funnel to strip off the extra characters.)

**Comma Separators:** Fixed point numbers often have a comma every third digit to make the number easier to read. Place a comma anywhere within or adjacent to the stream of # symbols if you want the number to be formatted with a comma. This example shows numbers with a comma and with no decimal point or digits after the decimal point.

```
Text=pattern(Amount, "#,")
```

**Negative Numbers:** Negative numbers are usually converted with a minus sign in front of the number. If you want a minus sign at the end of the number put a minus sign after the last # in the pattern.

```
Text=pattern(Amount, "#, .##-")
```

Negative numbers can also be displayed with parentheses around them. Simply put ( and ) characters around the # symbols. In this case the pattern( function will put an extra space after the number if the number is positive. The extra space helps make positive numbers line up with negative numbers if they are displayed in a list.

```
Text=pattern(Amount, "(#, .##)")
```

**Scientific Notation:**

To output a number using scientific notation add an E (or e) after the last # symbol.

```
Text=pattern(Amount, "#.####e")
```

Note: Scientific notation does not support more than one # in front of the decimal point, and it does not support commas every three digits.)

**Prefixes, Suffixes:**

The previous section described basic numeric patterns. You can embellish on these patterns by adding a prefix and or suffix. You may add any characters you want in the prefix or suffix, and they will be added “as-is” to the final converted number. The most common prefix is \$ for monetary values.

```
Text=pattern(Amount, "$#, .##")
```

Here is another example that adds a suffix for percentages.

```
Text=pattern(Amount, "#.##%")
```

**Plurals:**

If a suffix contains a measurement unit you may want to properly pluralize the units depending on the value being displayed. The pattern can use the ~ symbol to include an optional s in the suffix. The s is included if the value is not 1, for example 4 miles vs. 1 mile. Here is an example that converts text as kilograms.

```
Text=pattern(Amount, "#, kilogram~")
```

The ~ symbol can be used with any word that is plural with an s: mile~, ounce~, meter~, dollar~, cent~, hour~, day~, month~, year~, ohm~, volt~ etc. It does not work with words that change spelling when plural: foot (feet), inch (inches), etc.

**Spelling Numbers as Words:**

Numbers are normally converted to text as a sequence of digits. Through the use of a special symbol (§), a number can be spelled out as words (for example one hundred twenty three). There should only be one § symbol in the pattern. On the Macintosh, press **option-6** to create the § symbol. On Windows, press **Alt-0167**.

Only the integer part of the number is converted by the § symbol. If you are converting money you'll probably want to convert the fractional part (cents) also. You can convert the fractional part with the ¢ symbol. Use one ¢ symbol for each digit you want to display (usually 2). On the Macintosh, press **option-4** to create the ¢ symbol. On Windows, press **Alt-0162**.

The formula below shows a typical pattern for spelling out dollar values:

```
Text=pattern(Amount, "§ dollar~ and ¢¢ cent~")
```

If you want to spell out the cents also you must use two pattern functions like this:

```
Text=pattern(Amount, "§ dollar~ and ") +
lower(pattern(Amount*100, "§ cent~"))
```

The § symbol normally converts the number with the first letter capitalized and the rest lower case. To change this you can use the [lower\(\)](#), [upper\(\)](#), or [upperword\(\)](#) functions.

**Multiple Component Numbers:**

Numbers are normally converted to text as a continuous sequence of digits. You can also convert a number with the digits split up by punctuation or other characters. To do this, create a pattern with # symbols broken up by other characters. For example, here is a pattern that converts a number into standard social security number format (000-00-0000).

```
Text=pattern(SSNumber, "###-##-####")
```

As the last example shows, only the integer part is converted when the pattern has multiple components. Here's another example that converts a number into the standard format for a combination lock.

```
Text=pattern(Combination, "Right ## Left ## Right ##")
```

An integer can contain up to 9 digits. A floating point number can contain up to 15 digits.

**See Also:**

[pattern\(\)](#) function  
[str\(\)](#) function  
[date patterns](#)  
[datepattern\(\)](#) function

# OBJECT

**Syntax:** OBJECT name

**Description:** The **object** statement will select one graphic object in the current form based on the object's name. (This is the same concept as going into graphics mode and clicking on the object.) After the object is selected you can get information about it or change the attributes of the object.

**Parameters:** This statement has one parameter: name.

**name** is the name of the graphic object you want to select. If a graphic object does not have a name, you cannot select it with the **object** statement.

There are two ways to give a name to an object. Both start by selecting the object (in Graphics Mode). Once the object is selected, you can use the **Object Name** command (in the Edit menu) to assign the name. Or you can click on the object name portion of the Graphic Control Strip along the bottom of the window. (If the object name is not visible, click the triangle on the right side of the strip until you see the object name.

It is possible to have two or more graphic objects on a form with the same name, but this is usually not a good idea. If the **object** statement finds more than one object with the same name it only select the first one it finds.

**Action:** This statement allows a procedure to select a graphic object within a form. Once an objects is selected it can be examined or changed. This gives Panorama a limited capability to actually change forms on the fly.

**Examples:** This example will check to see if the mouse click was in an object called **Blue Square**. (This doesn't have to be a rectangle object, it could be any object with the name **Blue Square**.) If the click was in **Blue Square** the procedure displays the message **Hit the Bullseye!**

```

local blueRectangle
object "Blue Square"
blueRectangle=objectinfo("rectangle")
blueRectangle=xytoxy(blueRectangle,"form","screen")
if inrectangle(info("click"),blueRectangle)
 message "Hit the Bullseye!"
endif

```

**Views:** This statement may be used in Form views.

**See Also:** [selectobjects](#) statement  
[selectallobjects](#) statement  
[selectnoobjects](#) statement  
[objectid](#) statement  
[objectnumber](#) statement  
[changeobjects](#) statement  
[objectinfo\(\)](#) function

# OBJECTID

- Syntax:** `OBJECTID number`
- Description:** The `objectid` statement will select one graphic object in the current form based on an ID number.
- Parameters:** This statement has one parameter: `number`.  
**number** is a special ID number that identifies a graphic object. Each graphic object in a form has a unique ID number. You can use the `objectinfo("id")` function to find out the ID number for an object. Warning: Editing the form in graphics mode may change all the ID value for objects in the form.)
- Action:** This statement allows a procedure to select a graphic object within a form based on an earlier selection. A procedure usually starts by getting the ID number of an object using the `objectinfo("id")` function. The ID number is stored in a variable. Later, the ID number is recalled and used to select the graphic object.
- Examples:** This example will find and store the ID of the object the user clicks on. (Note: This procedure is assumed to be triggered by a button which overlays several other graphic objects.)

```
global hitObject
local hitPt
hitPt=xytoxy(info("click"), "Screen", "Form")
selectobjects inrectangle((hitPt, objectinfo("rectangle"))
 and objectinfo("type")≠"Button"
objectnumber objectinfo("count")
hitObject=objectinfo("ID")
```

Later another procedure can re-select this object easily.

```
global hitObject
objectid hitObject
```

**Views:** This statement may be used in Form views.

**See Also:** [selectobjects](#) statement  
[selectallobjects](#) statement  
[selectnoobjects](#) statement  
[object](#) statement  
[objectnumber](#) statement  
[changeobjects](#) statement  
[objectinfo\(\)](#) function

# OBJECTINFO(...)

**Syntax:** OBJECTINFO(OPTION)

**Description:** The `objectinfo()` function returns information about a graphic object: its location, size, color, font, etc. This function must be used in combination with either the [object](#), [selectobjects](#) or [changeobjects](#) statement.

**Parameters:** This function has one parameter: `option`.

**option** is the type of information you want to retrieve about an object. You must pick the option from the list below:

```
objectinfo("rectangle")
objectinfo("ID")
objectinfo("name")
objectinfo("fieldname")
objectinfo("type")
objectinfo("custom")
objectinfo("font")
objectinfo("textsize")
objectinfo("textstyle")
objectinfo("alignment")
objectinfo("color")
objectinfo("selected")
objectinfo("locked")
objectinfo("expandable")
objectinfo("expandshrink")
objectinfo("tile")
objectinfo("text")
objectinfo("fillpattern")
objectinfo("linepattern")
objectinfo("linewidth")
objectinfo("count")
objectinfo("boundary")
```

**Result:** The function returns different types of data depending on the option selected. See the examples section for details.

**Examples:** There are about a dozen types of information the `objectinfo()` function can extract from an object.

**objectinfo("rectangle")** - This option returns the dimensions (location and size) of the object. The dimensions are returned using the rectangle data type (see the [rectangle\(\)](#) function) and in form relative co-ordinates (See "[XYTOXY\(\)](#)" on page 5910).

The example below selects the data cell(s) the user clicked on. The procedure uses the `inrectangle()` function to determine which object (if any) was clicked on. (Note: Presumably this procedure would be triggered by a push button which covers the data cell objects.)

```
local hitPt, hitField
hitPt=xytoxy(info("click"), "Screen", "Form")
selectobjects inrectangle(hitPt,objectinfo("rectangle"))
and objectinfo("type") beginswith "Data Cell:"
objectnumber 1
hitField=objectinfo("type")[":",-1][-2,-1]
```

```

if hitField="" stop endif
field hitField
editcell

```

If the user did click on a data cell, the procedure activates the cell.

**objectinfo("name")** - This option returns the name of the object. This is the name that is assigned by the Object Name dialog (in the Edit menu, or Graphic Control Strip). The two lines shown below are basically equivalent.

```

object "Swiss Cheese" selectobjects objectinfo("name")="Swiss Cheese"

```

These two statements are not completely equivalent. If there is more than one object named Swiss Cheese the selectobjects statement will select all of them. The object statement will select only the one closest to the back.

**objectinfo("type")** - This option returns the type of the object. The object types are:

```

Rectangle
Rounded Rectangle
Oval
Line
Picture
Auto-Wrap Text
Click Text
Data Cell:<field>
Button
Chart
Flash Art
Flash Sound
Balloon Help
SuperObject:<type of SuperObject>
Tile:<type of tile>
Group

```

**objectinfo("font")** - This option returns the font for this object. If the option does not have a font (an oval, for example) this option will return empty text. This example converts all Courier text to American Typewriter.

```

selectobjects objectinfo("font")="Courier"
changeobjects "font","American Typewriter"

```

**objectinfo("textsize")** - This option returns the size of text displayed by this object. If the object does not have a text size (an oval, for example) this option will return zero.

**objectinfo("textstyle")** - This option returns the text style of text displayed by the object. The text style is a number that is created by adding up the numbers for each individual style from the table below. For example, for bold italic text the style will be 3.

```

0 Plain
1 Bold
2 Italic
4 Underline
8 Outline
16 Shadow

```

The example below selects all italic objects and then changes the color of the selected objects to blue.

```

selectobjects objectinfo("textstyle") and 2
changeobjects "color",rgb(0,0,65535)

```

**objectinfo("alignment")** - This option returns the alignment of text displayed by this object, either "Left", "Center", or "Right". If the object does not have a text size (an oval, for example) this option will return "". **objectinfo("color")** - This option returns the color of the object. For example, this procedure selects all objects with brightness below 50%, then changes it to a minimum brightness of 50%.

```
selectobjects brightness(objectinfo("color"))<32768
changeobjects "color",
hsb(
hue(objectinfo("color")),
saturation(objectinfo("color")),32768)
```

**objectinfo("selected")** - This option returns true or false depending on whether or not the object is already selected (by a previous selectobjects statement).

**objectinfo("locked")** - This option returns true or false depending on whether or not the object is locked. (A locked object cannot be modified when in graphic editing mode.) The example below selects all rectangles that are not locked.

```
selectobjects objectinfo("type")="Rectangle"
and not objectinfo("locked")
```

**objectinfo("expandable")** - This option returns true or false depending on whether or not the object is expandable depending on the amount of data to be printed.

**objectinfo("expandshrink")** - This option returns true or false depending on whether or not the object can expand or shrink depending on the amount of data to be printed.

**objectinfo("text")** - This option returns the text in auto-wrap text objects or click text objects. When used with any other type of object it returns empty text. This example changes all text objects that contain the word Phone to italic.

```
selectobjects objectinfo("text") contains "Phone"
changeobjects "textstyle",objectinfo("textstyle") or 2
```

**objectinfo("fillpattern")** - This option returns the fill pattern of the object (if any). Patterns are 8 bytes of raw data (see Binary Data). Here are some formulas for typical patterns.

| Formula                                   | Pattern            |
|-------------------------------------------|--------------------|
| <code>radix(16,"FFFFFFFFFFFFFFFF")</code> | black              |
| <code>radix(16,"0000000000000000")</code> | white              |
| " "                                       | none (transparent) |
| <code>radix(16,"AA55AA55AA55AA55")</code> | 50% gray pattern   |
| <code>radix(16,"8822882288228822")</code> | light gray         |
| <code>radix(16,"DD77DD77DD77DD77")</code> | dark gray          |
| <code>radix(16,"8888888888888888")</code> | vertical lines     |
| <code>radix(16,"FF000000FF000000")</code> | horizontal lines   |
| <code>radix(16,"FF888888FF888888")</code> | cross hatch        |

This list shows only a few of the possible patterns—there are literally millions of patterns that can be created.



**objectinfo("linepattern")** - This option returns the line pattern of the object (if any). Patterns are 8 bytes of raw data (see [binary data](#)). See the table above for examples of typical patterns.

**objectinfo("linewidth")** - This option returns the line width of the object (if any). The line width is a number from 1 to 8, or zero if this object does not support a line width.

**objectinfo("ID")** - This option returns a unique number that can be used to identify this object later. The number is valid as long as the form is not edited in graphics mode. The `objectid` statement can use this unique ID number to re-locate this object later.

**objectinfo("count")** - This option applies not to a specific object, but to the entire form. It counts the number of currently selected objects. For example, this example displays the number of rectangles in the current form.

```
selectobjects objectinfo("type") contains "rectangle"
message "This form contains "+str(objectinfo("count"))+" rectangles."
```

**objectinfo("boundary")** - This option applies not to a specific object, but to the entire form. It calculates the minimum rectangle that encloses all of the selected objects.

#### Errors:

**Illegal info argument.** This error occurs if the option is not one of the legal options listed above.

#### See Also:

[object](#) statement  
[selectobjects](#) statement  
[changeobjects](#) statement

# OBJECTNUMBER

**Syntax:** OBJECTNUMBER number

**Description:** The **objectnumber** statement identifies one graphic object from a set of selected graphic objects in the current form.

**Parameters:** This statement has one parameter: **number**.

**number** identifies the graphic object within the set of selected graphic objects. The first graphic object is 1, the second is 2, etc. If you specify a number that is larger than the number of selected object [info\("found"\)](#) will be false. (Note: Graphic objects are numbered in back to front order.)

**Action:** This statement locates the nth selected object so that you can retrieve information about the object. The first selected object is 1, the second is 2, etc. After this statement you can use the [objectinfo\(\)](#) function to get information about the object. This statement allows you to create a loop to accumulate information about multiple graphic objects within the form.

**Examples:** The example procedure below builds a list of the names of all SuperObjects in the current form.

```

local objectNames,X
X=1
selectobjects objectinfo\("type"\) beginswith "SuperObject"
loop
 objectnumber X
 stoploopif (not info\("found"\))
 objectNames=
 sandwich(" ",objectNames,¶)+objectinfo\("name"\)
 X=X+1
while forever

```

This statement can also be used to identify the object closest to the front or the back in the selected set. To identify the object closest to the back, use 1.

```

selectobjects objectinfo\("font"\)="Helvetica"
objectnumber 1

```

To identify the object closest to the front, use [objectinfo\("count"\)](#).

```

selectobjects objectinfo\("font"\)="Helvetica"
objectnumber objectinfo\("count"\)

```

**Views:** This statement may be used in Form views.

**See Also:** [selectobjects](#) statement  
[selectallobjects](#) statement  
[selectnoobjects](#) statement  
[object](#) statement

[objectid](#) statement  
[changeobjects](#) statement  
[objectinfo\(\)](#) function

# OKCANCEL

- Syntax:** OKCANCEL text
- Description:** The `okcancel` statement pauses a procedure and displays a modal dialog showing the text and two buttons **Ok** (the default button) and **Cancel**. The name of the button clicked on will be written to the clipboard.
- Parameters:** This statement has one parameter: `text`.
- `text` is the character string that will appear in a modal dialog displayed on screen. This text string must be surrounded by quote marks ( " " ). The string limit is dependent on the characters used and assumes the system font (Chicago, 12 point).
- Action:** This statement allows the procedure programmer to pause the procedure by presenting the user with a modal dialog that asks a question requiring one of two responses **Cancel** or **Ok**. Whichever response is selected will be written to the clipboard and it can be tested for later in the procedure.
- Examples:** This simple example will have Panorama display a modal dialog asking you if you wish to cancel or continue the procedure. If you click on the **Cancel** button the procedure will stop.

```
okcancel "Do you wish to continue this procedure?"
if clipboard() = "Cancel"
 stop
endif
...
...
...
```

This example first tests to see if a select command failed to select any records and if it fails asks you if you wish to cancel or try again.

```
local companyname
beginning:
gettext "Enter Company Name.",companyname
select «Company» contains companyname
if info("empty")
 okcancel "Cancel to stop, Ok to try again."
 if clipboard() contains "cancel"
 stop
 endif
 if clipboard() contains "ok"
 goto beginning
 endif
endif
```

- Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:**

[alert](#) statement  
[cancelok](#) statement  
[clipboard\(\)](#) function  
[customalert](#) statement  
[customdialog](#) statement  
[getscrap](#) statement  
[gettext](#) statement  
[info\("dialogtrigger"\)](#) function  
[message](#) statement  
[noyes](#) statement  
[openresource](#) statement  
[yesno](#) statement  
[alertmode](#) statement

# ONERROR

**Syntax:** ONERROR statements

**Description:** The `onerror` statement can be used to catch all errors that are not trapped by `if error` statements. This has two benefits: It allows the programmer to easily eliminate all error alert dialogs. This is very important for server applications because an alert dialog requires human intervention to get the server going again.

It makes it easy to build a log of errors.

**Parameters:** This statement has one parameter: `statements`.

**statements** is a text string that contains one or more Panorama statements to be executed when an error occurs. Notice that this is not the name of a procedure, but the actual statements themselves (as a string of text). This is similar to the `execute` statement. Once an error has occurred these statements will run. Within these statements you can use the `info("error")` function to find out what the error was, if necessary.

**Action:** This statement allows you to specify what happens when Panorama encounters an error as part of running a procedure. When Panorama encounters an error, it checks to see if the next line is `if error`. If not, it usually stops and displays an error message. However, if an `OnError` statement has been encountered, Panorama will not stop and will not display an error message. Instead, it will execute the statements specified as the parameter to the `OnError` statement.

The effect of the `OnError` statement ends when the main procedure stops running. In other words, `OnError` isn't a permanent error handler — you must specify it for each procedure you wish to have error trapping. If you plan to use `OnError`, it is probably best to put it in the first line of any procedure that needs error trapping. If you are going to use the same statements with `OnError` in several different procedures, you may want to set up the statements in a variable in your `.Initialize` procedure, then use that variable as the parameter to `OnError`.

It's important to consider the possible environment that may exist when an error is created. Depending on the flow of your main procedure, Panorama may not be in the same window or even in the same database. Your `OnError` program should generally not make any assumptions about what windows or databases will be active or available when the error occurs.

**Examples:** Here is an example of how `OnError` could be used in a CGI (web server) application. In this example if there is an error Panorama will return an error message to the web server and also log the error along with the date and time.

```
global cgiResult,errorLog
errorLog=errorLog /* make sure errorLog exists */
if error
 errorLog="" /* initialize errorLog */
endif
onerror {cgiResult="Panorama Error: "+info("error") }+
 {errorLog=sandwich(" ",errorLog,¶)+}+
 {datepattern(today(),"DD/MM/YYYY ")+}+
 {timepattern(now(),"hh:mm:ss")+}+
 {info("error")}
```

```
/* error logging is set up, now we can continue with our tasks */
...
... rest of this procedure
```

**Views:** This statement may be used in any view.

**See Also:** [info\("error"\)](#) function  
[if](#) statement  
[alertmode](#) statement

# OPENAS

**Syntax:** OPENAS name

**Description:** The **openas** statement works with the [openfile](#) statement or [opensecret](#) statement. After the **openas** statement, the next file will be opened in memory with a different name from its disk name. This allows you to open two databases with the same name at the same time (they must be in different folders)

**Parameters:** This statement has one parameter: name.

**name** is the name that should be used for the next file instead of its disk name. The name should be a text string that is up to 31 characters long, and should be different from the name of any database that is already open.

**Action:** This statement allows two databases with the same name to be opened at the same time. For example, you may want to open an old version of a database at the same time as a newer version of the same database.

**Examples:** The example loads a new phone book file with the new data from another person's phone book file. Both files are named **Phone Book**, so normally they cannot both be open at the same time. Using the **openas** statement this procedure opens one of the databases as **Other Phone Book**, allowing both to be open in memory simultaneously.

```

openfile "Phone Book"
openas "Other Phone Book"
openfile "Scott's Disk:Phone Book"
select Status="New"
window "Phone Book"
openfile "+Other Phone Book"
save
window "Other Phone Book"
closefile

```

**Views:** This statement may be used in any view.

**See Also:** [openfile](#) statement  
[opensecret](#) statement



# OPENCROSSTAB

**Syntax:** `OPENCROSSTAB crosstab`

**Description:** The `opencrosstab` statement opens a crosstab from the current database in a new window.

**Parameters:** This statement has one parameter: `crosstab`.  
`crosstab` is the name of the crosstab to open.

**Action:** This statement opens a crosstab in a new window. The effect is similar to selecting the crosstab from the View menu (the pop-up menu in the window title) with the Control key held down. If the crosstab is already open, it is simply brought to the front. By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the [setwindowrectangle](#), [setwindow](#), or [windowbox](#) statements.

**Examples:** The procedure below opens the crosstab **Budget** in a 4 inch by 6 inch window centered on the main screen.

```
local newWindowRect
newWindowRect=rectanglecenter(
 info("screenrectangle"),
 rectangle(1,1,4*72,6*72))
setwindowrectangle newWindowRect," "
opencrosstab "Budget"
```

This example opens the **Budget** crosstab from the Checkbook database. This procedure will work from any database, even if the Checkbook database doesn't have any windows open.

```
window "Checkbook:Secret "
openform "Budget"
```

**Views:** This statement may be used in any view

**See Also:** [opensheet](#) statement  
[opendesignsheet](#) statement  
[openprocedure](#) statement  
[openform](#) statement  
[gosheet](#) statement  
[godesignsheet](#) statement  
[goform](#) statement  
[goprocedure](#) statement  
[gocrosstab](#) statement  
[setwindow](#) statement  
[setwindowrectangle](#) statement  
[windowbox](#) statement  
[info\("windows"\)](#) function  
[listwindows\(\)](#) function

# OPENDESIGNSHEET

- Syntax:** OPENDESIGNSHEET
- Description:** The `opendesignsheet` statement opens the design sheet window for the current database in a new window.
- Parameters:** This statement has no parameters.
- Action:** This statement opens the design sheet. The effect is similar to selecting Design Sheet from the View menu (the pop-up menu in the window title) with the Control key held down. If the design sheet is already open, it is simply brought to the front. By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the [setwindowrectangle](#), [setwindow](#), or [windowbox](#) statements.
- Examples:** The procedure below opens the design sheet in a 4 inch by 6 inch window centered on the main screen.

```

local newWindowRect
newWindowRect=rectanglecenter(
 info("screenrectangle"),
 rectangle(1,1,4*72,6*72))
setwindowrectangle newWindowRect," "
opendesignsheet

```

This example opens the design sheet for the Price List database. This procedure will work from any database, even if the Price List database doesn't have any windows open.

```

window "Price List:Secret"
opendesignsheet

```

**Views:** This statement may be used in any view.

**See Also:** [newgeneration](#) statement  
[opensheet](#) statement  
[openform](#) statement  
[openprocedure](#) statement  
[opencrosstab](#) statement  
[gosheet](#) statement  
[godesignsheet](#) statement  
[goform](#) statement  
[goprocedure](#) statement  
[gocrosstab](#) statement  
[setwindow](#) statement  
[setwindowrectangle](#) statement  
[windowbox](#) statement  
[info\("windows"\)](#) function  
[listwindows\(\)](#) function

# OPENDIALOG

**Syntax:** OPENDIALOG form

**Description:** The **opendialog** statement opens a form from the current database in a new window. The new window will have no scroll bars, tool palette or drag bar, and will behave like a modal dialog box.

**Parameters:** This statement has one parameter: form.

**form** is the name of the form to open.

**Action:** This statement opens a form in a modal dialog window. As long as this window is open, all other Panorama windows are inactive. The only way to close this dialog window is with the [closewindow](#) statement. While the dialog window is open you cannot click on another Panorama window to bring it in front of the dialog window. (Exception: If you hold down the COMMAND key you can click on other windows and bring them to the top.) If you click outside the dialog window, the **.OutOfBounds** procedure (if any) will be triggered. By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the [setwindowrectangle](#), [setwindow](#), or [windowbox](#) statements.

**Examples:** The procedure below opens the form Transaction Preferences in a 4 inch by 6 inch dialog window centered on the main screen. The procedure then pauses so the user can fill in the dialog (see pause and resume).

```
global prefState
local newWindowRect
newWindowRect=rectanglecenter(
 info("screenrectangle"),
 rectangleSize(1,1,4*72,6*72))
setwindowrectangle newWindowRect,"noHorzScroll noVertScroll noPalette"
opendialog "Transaction Preferences"
pause prefState
```

**Views:** This statement may be used in any view.

**See Also:** [pause](#) statement  
[resume](#) statement  
[closewindow](#) statement  
[opensheet](#) statement  
[openform](#) statement  
[openprocedure](#) statement  
[opencrosstab](#) statement  
[gosheet](#) statement  
[godesignsheet](#) statement  
[goform](#) statement  
[goprocedure](#) statement  
[gocrosstab](#) statement  
[setwindow](#) statement  
[setwindowrectangle](#) statement

windowbox statement  
info("windows") function  
listwindows() function

# OPENFILE

**Syntax:** OPENFILE file

**Description:** The **openfile** statement opens a database file. It can also import data from a text file, another database, or a variable into the current database. If the text contains one or more [html tables](#) it will import the first table found.

**Parameters:** This statement has one parameter: file.

**file** identifies the file you want to open. If the file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "Disk:Accounting:Invoice".

If file is the keyword **dialog**, Panorama will pause the procedure and present the user with the standard **Open...** dialog. This allows the user to choose the file on the fly.

If file begins with a + symbol, Panorama will append the file to the current database, with the first field appended to the first field, second field appended to second field, etc. If file begins with ++, Panorama will match the field names between the two databases as it appends the data. If a field name does not match, that field will not be appended.

If file begins with a & symbol, Panorama will replace the data in the current database with the data in the file. The fields will be replaced in order, with the first field replacing the first field, second field replacing the second field, etc. If file begins with &&, Panorama will match the field names between the two databases as it replaces the data.

If file begins with a @ symbol, Panorama will import from a variable instead of from a text file. For example, @mydata tells Panorama to import from the variable mydata. This symbol may also be combined to append (+@) or replace (&@) the text from the variable.

**Action:** This statement can be used two ways: 1) It can open a database, 2) it can import into an existing database. These operations are the same as the different options available when using the Open command in the File menu. When used to open a database, the openfile statement loads the database into memory, opens the windows for the database and automatically calls the .Initialize procedure (if any). If you don't want the windows to open use the [opensecret](#) statement.

**Examples:** The example procedure below opens the price list and customer list databases.

```
openfile "Price List"
openfile "Customer List"
```

The procedure allows the user to select a TEXT file, then appends that text file to the current database.

```
local file,folder,type
openfiledialog folder,file,type,"TEXT"
if file=""
 stop /* the user pressed the CANCEL button */
endif
openfile "+" + folderpath(folder) + file
```

The example procedure below replaces the contents of the file **Telemetry Data** with the new data in **Telemetry.TXT**.

```
openfile "Telemetry Data"
openfile "&Telemetry.TXT"
```

If you are using the Windows operating system and you want to import a text file that does not have the `.txt` extension (or has a different extension, like `.ini` or `.html`) you must use the [opentextfile](#) statement instead of the **openfile** statement.

The **openfile** statement may be used to simulate the **Open** command in your own custom **File** menu. Here are the statements to use in your **.CustomMenu** procedure.

```
if info("trigger") beginswith "Menu.File.Open"
 openfile dialog
 stop
endif
```

**Views:** This statement may be used in any view

**See Also:** [opentextfile](#) statement  
[closefile](#) statement  
[openas](#) statement  
[opensecret](#) statement  
[importusing](#) statement  
[newdatabase](#) statement  
[openfiledialog](#) statement  
[opentextfile](#) statement  
[openresource](#) statement  
[fileinfo\(\)](#) function  
[fileload\(\)](#) function  
[folder\(\)](#) function  
[folderpath\(\)](#) function  
[import\(\)](#) function  
[importcell\(\)](#) function  
[info\("files"\)](#) function  
[html tables](#)

# OPENFILEDIALOG

**Syntax:** OPENFILEDIALOG folder,filename,type,typelist

**Description:** The `openfiledialog` statement pauses a procedure and displays the standard “open file” dialog. This is the same dialog that most applications use for opening or selecting a file. The user can then select a file from the disk. Once the user has picked a file the procedure resumes and can process the file with the `openfile` statement or `fileload()` function.

**Parameters:** This statement has four parameters: `folder`, `filename`, `type` and `typelist`.

**folder** should be a variable. When the statement is finished this variable will contain a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the selected file is located. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. The path id can be converted into a text description of the path with the `folderpath()` function.

**filename** should be a variable. When the statement is finished the variable will contain the name of the file that was selected by the user. If the `filename` parameter is empty the user pressed the **Cancel** button.

**type** is a four character code that identifies the type of file the user selected. There are hundreds of possible type codes. Here is a list some of the more common types you may encounter:

| Type Code | Description              |
|-----------|--------------------------|
| TEXT      | Text File                |
| PICT      | Picture File             |
| ESPF      | Encapsulated Postscript  |
| APPL      | Application (Program)    |
| ZEPD      | Panorama Database        |
| KSET      | Panorama File Set        |
| GIFf      | GIF Image File           |
| JPEG      | JPEG Image File          |
| MOOV      | QuickTime Movie          |
| PDF       | Adobe Acrobat PDF Format |

**typelist** is the only parameter that you actually supply. This is a list of the types of files that should be displayed in the dialog. If you want the dialog to display all files, the `typelist` should be an empty string (""). If you wanted to display only text files, the `typelist` should be "TEXT". If you want to display only picture, postscript and GIF files, the `typelist` should be "PICTESPFGIFf". The `typelist` may be as long as you want, but it should always be a multiple of four characters (4, 8, 12, 16, etc.).

**Action:** This statement causes the standard “open file” dialog to appear. This allows the user to select a file from the disk

**Examples:** The procedure allows the user to select a TEXT file, then appends that text file to the current database.

```

local file, folder, type
openfiledialog folder, file, type, "TEXT"
if file=""
 stop /* the user pressed the CANCEL button */
endif
openfile "+" + folderpath((folder) + file

```

## Custom Dialogs

If you are using Panorama 3.1 or later on a Macintosh, you can customize the open file dialog by using the `customdialog` statement. The customization options available include changing the layout of the dialog, adding extra text to the dialog and adding extra push buttons to the dialog. (You cannot add other kinds of controls to the dialog, for example checkboxes, radio buttons, or pop-up menus.)

Most of the work in setting up a custom dialog involves creating a resource template for the dialog. To do this you will need a resource editing program like ResEdit or Resourcer. (See Appendix A of the Panorama Real World Programming Guide for more information on these programs, or consult the documentation for the programs themselves.) Once the resource template is set up, it can be used in any procedure by inserting the `customdialog` statement just before the `openfiledialog` or `savefiledialog` statements.

The resource for an open file dialog must contain at least the required items listed below. Once the required items are set up in this order you can add additional items of your own. The easiest way to do this correctly is to make a copy of DLOG 9000 in the File Dialogs.rsrc file, then adjust the layout and add your own items as necessary.

- 1) Open Button (you may rename this button)
- 2) Invisible Button
- 3) Cancel Button
- 4) Disk Name
- 5) Eject Button
- 6) Drive Button
- 7) Filename List
- 8) Scroll Bar
- 9) Dotted Line
- 10) Invisible text

To use your custom file dialog in a procedure you must place the `customdialog` statement just before the `openfiledialog` statement, like this:

```

local folder, file
customdialog 9037
openfiledialog folder, file, "TEXT", ""
case info("dialogtrigger") contains "Open"
 ...
case info("dialogtrigger") contains "Select Folder"
 ...
endcase

```

As this example shows, the `info("dialogtrigger")` will contain the name of the push button that the user pressed. (Note: If the user clicks on a pushbutton without selecting a file, the procedure can still find out what folder was selected, as shown in this example.)



**Views:** This statement may be used in any view

**See Also:** [customdialog](#) statement  
[savefiledialog](#) statement  
[openfile](#) statement  
[filesave](#) statement  
[filerename](#) statement  
[filetrash](#) statement  
[folder\(\)](#) function  
[folderpath\(\)](#) function  
[fileload\(\)](#) function  
[filesize\(\)](#) function

# OPENFORM

**Syntax:** OPENFORM form

**Description:** The **openform** statement opens a form from the current database in a new window.

**Parameters:** This statement has one parameter. **form**  
**form** is the name of the form to open.

**Action:** This statement opens a form in a new window. The effect is similar to selecting the form from the View menu (the pop-up menu in the window title) with the **Control** key held down. If the form is already open, it is simply brought to the front. By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the [setwindowrectangle](#), [setwindow](#), or [windowbox](#) statements. These statements can also make the new window appear without scroll bars or a tool palette.

**Examples:** The procedure below opens the form **Utilities** in a 4 inch by 6 inch window centered on the main screen. The new form window will not have any scroll bars or a tool palette.

```
local newWindowRect
newWindowRect=rectanglecenter(
 info("screenrectangle"),
 rectangle(1,1,4*72,6*72))
setwindowrectangle newWindowRect,"noHorzScroll noVertScroll noPalette"
openform "Utilities"
```

This example opens the Adjust form from the Price List database. This procedure will work from any database, even if the Price List database doesn't have any windows open.

```
window "Price List:Secret"
setwindowrectangle rectangle(40,50,200,250)," "
openform "Adjust"
```

**Views:** This statement may be used in any view even.

**See Also:** [opendialog](#) statement  
[opensheet](#) statement  
[opendesignsheet](#) statement  
[openprocedure](#) statement  
[opencrosstab](#) statement  
[gosheet](#) statement  
[godesignsheet](#) statement  
[goform](#) statement  
[goprocedure](#) statement  
[gocrosstab](#) statement  
[setwindow](#) statement  
[setwindowrectangle](#) statement

[windowbox](#) statement  
[info\("windows"\)](#) function  
[listwindows\(\)](#) function

# OPENPROCEDURE

- Syntax:** OPENPROCEDURE procedure
- Description:** The **openprocedure** statement opens a procedure from the current database in a new window.
- Parameters:** This statement has one parameter: procedure.  
**procedure** is the name of the procedure to open.
- Action:** This statement opens a procedure in a new window. The effect is similar to selecting the procedure from the View menu (the pop-up menu in the window title) with the **Control** key held down. If the procedure is already open, it is simply brought to the front. (Note: This statement may only be used if the database is in Author mode. If the database is in User Mode or Custom Mode the **openprocedure** statement will not open the window, and an error will occur (which may be trapped with `if error`). The procedure can find out what mode the database is in with the `dbinfo("level",...)` function.)
- By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the `setwindowrectangle`, `setwindow`, or `windowbox` statements.
- Examples:** The procedure below opens the procedure .CustomMenu in a 4 inch by 6 inch window centered on the main screen.
- ```

local newWindowRect
newWindowRect=rectanglecenter(
    info("screenrectangle"),
    rectangle(1,1,4*72,6*72))
setwindowrectangle newWindowRect,""
openprocedure ".CustomMenu"

```
- Views:** This statement may be used in any view
- See Also:** [opensheet](#) statement
[opendesignsheet](#) statement
[opencrosstab](#) statement
[openform](#) statement
[gosheet](#) statement
[godesignsheet](#) statement
[goform](#) statement
[goprocedure](#) statement
[gocrosstab](#) statement
[setwindow](#) statement
[setwindowrectangle](#) statement
[windowbox](#) statement
[info\("windows"\)](#) function
[listwindows](#)((function

OPENRESOURCE

Syntax: OPENRESOURCE file

Description: The **openresource** statement opens a resource file. Once the file is opened the resources inside the file can be accessed and used.

Parameters: This statement has one parameter: file.

file identifies the resource file you want to open. If the resource file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "Disk:Sounds:Star Trek". (On Windows, resource files always have the extension .RSR. The openresource statement will add this extension for you.)

Action: The Macintosh has a special kind of file that allows many items to be stored inside a single file. This is called a resource file. Each individual item within the file is called a resource. Each resource may be anything from a single character to tens of thousands of bytes of information, and may contain menus, text, pictures, sounds...virtually any kind of data the Macintosh understands.

Panorama has functions and statements for accessing the items inside a resource file. Before these statements can be used, the resource file must be opened with the **openresource** statement. This statement loads a directory of information in the resource file into memory, where Panorama can access it. The resources remain available until the file is closed with the [closeresource](#) statement. It is possible to open more than one resource at once, however in this case you must be careful to make sure that the resource items inside the file do not conflict with each other.

Resource files are created and modified with special resource editor programs like ResEdit and Resourcer™.

Examples: The example procedure below opens the resource file My Menu.

```
openresource "My Menu"
```

Views: This statement may be used in any view

See Also: [openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function
[getString\(\)](#) function
[getnstring\(\)](#) function
[getStringmatch\(\)](#) function
[resources\(\)](#) function
[resourcetypes\(\)](#) function

OPENRESOURCERW

Syntax: OPENRESOURCERW file

Description: The `openresourcerw` statement opens a resource file. Once the file is opened the resources inside the file can be accessed and modified.

Parameters: This statement has one parameter: file.

file identifies the resource file you want to open. If the resource file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "`Disk:Sounds:Star Trek`".

Action: The Macintosh has a special kind of file that allows many items to be stored inside a single file. This is called a resource file. Each individual item within the file is called a resource. Each resource may be anything from a single character to tens of thousands of bytes of information, and may contain menus, text, pictures, sounds...virtually any kind of data the Macintosh understands.

Panorama has functions and statements for accessing the items inside a resource file. Before these statements can be used, the resource file must be opened with the `openresource` statement (read only) or `openresourcerw` statement (read and write). This statement loads a directory of information in the resource file into memory, where Panorama can access it. The resources remain available until the file is closed with the `closeresource` statement. It is possible to open more than one resource at once, however in this case you must be careful to make sure that the resource items inside the file do not conflict with each other.

Resource files are created and modified with special resource editor programs like ResEdit and Resourcerer™.

Examples: The example procedure below opens the resource file My Menus.

```
openresource "My Menus"
```

Views: This statement may be used in any view.

See Also: [openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function
[getString\(\)](#) function
[getnstring\(\)](#) function
[getStringmatch\(\)](#) function
[resources\(\)](#) function
[resourcetypes\(\)](#) function

OPENSECRET

Syntax: OPENSECRET filename

Description: The `opensecret` statement is just like the [openfile](#) statement, but it does not open any windows or launch the `.Initialize` procedure.

Parameters: This statement has one parameter: filename.

filename is the name of the file to be opened. If this file is not in the same folder as the current database you must supply both the path and the file, for example

```
"My Disk:Manhattan Project:Isotopes"
```

If you have the path id for the folder you can convert it into a path with the [folderpath\(\)](#) function like this:

```
folderpath(ProjectFolder)+"Isotopes"
```

Action: This statement opens a database without opening any windows or launching the `.Initialize` procedure. Using this statement, a procedure can open any database without windows, whether or not the "No Windows" option was set in the **Save As** dialog. If a file has been opened with `opensecret` and you later open it normally (either with the [openfile](#) statement or manually with the **Open** dialog) the windows will be opened at that time.

Examples: The example opens a database called Shipping Rates, but does not open any of the windows for that database. It then looks up a rate from the table.

```
opensecret "Shipping Rates"  
Shipping=lookup("Shipping Rates",Zone,Zip[1,3],Rate,0,0)
```

Views: This statement may be used in any view.

See Also: [openfile](#) statement
[openas](#) statement
[makesecret](#) statement

OPENSHEET

- Syntax:** OPENSHEET
- Description:** The `opensheet` statement opens the data sheet window for the current database in a new window.
- Parameters:** This statement has no parameters.
- Action:** This statement opens the data sheet. The effect is similar to selecting Data Sheet from the View menu (the pop-up menu in the window title) with the Control key held down. If the data sheet is already open, it is simply brought to the front.
- By default, the new window will take up most of the screen, covering all of the other windows. To specify the size and location of the new window in advance, use the [setwindowrectangle](#), [setwindow](#), or [windowbox](#) statements.
- Examples:** The procedure below opens the data sheet in a 4 inch by 6 inch window centered on the main screen.

```
local newWindowRect
newWindowRect=rectanglecenter(
info("screenrectangle"),
rectanglesize(1,1,4*72,6*72))
setwindowrectangle newWindowRect, ""
opensheet
```

This example opens the data sheet for the Price List database. This procedure will work from any database, even if the Price List database doesn't have any windows open.

```
window "Price List:Secret"
opensheet
```

- Views:** This statement may be used in any view.

- See Also:** [opendesignsheet](#) statement
[openform](#) statement
[openprocedure](#) statement
[opencrosstab](#) statement
[gosheet](#) statement
[godesignsheet](#) statement
[goform](#) statement
[goprocedure](#) statement
[gocrosstab](#) statement
[setwindow](#) statement
[setwindowrectangle](#) statement
[windowbox](#) statement
[info](#)("windows") function
[listwindows](#)((function

OPENSOUND

Syntax: OPENSOUND file

Description: The **opensound** statement opens a resource file containing digitally recorded sounds. Use this command when you want to play several sounds in a row with the [playsound](#) statement. Sounds can be recorded using **Sound** control panel.

Parameters: This statement has one parameter. file

file is the name of the resource file that contains the sounds you want to play. If the resource file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "Disk:Sounds:Star Trek".

Action: This statement opens a resource file containing sounds. A sound resource file contains a SND resource for each sound and each SND resource is given a name. Sound resources may be created with a sound program like Farallon's MacRecorder or shareware programs like Sound->snd or SoundMover. System 7 users may even create sound resources with the Sound control panel and a Macintosh which supports a microphone.

Examples: This example opens a sound resource called Bird Calls, plays the sound titled Robin and then closes Bird Calls.

```
opensound "Bird Calls"
playsound "Robin"
closesound
```

This example asks the user a question to answer, opens a sound resource called Remarks, and depending on the answer plays the sound called **Correct** or **Wrong**. It then closes Remarks.

```
local Answer,Reply
message "The Alamo is in:"+¶+"1. New Mexico"+
        ¶+"2. Texas"+¶+"3. California"
gettext "Is the answer 1, 2, or 3",Answer
opensound "Remarks"
case Answer = "1"
    Reply = "Wrong"
case Answer = "2"
    Reply = "Correct"
case Answer = "3"
    Reply = "Wrong"
endcase
playsound Reply
closesound
```

Views: This statement may be used in any view even if there are no visible windows open.

See Also: [playsound](#) statement
[closesound](#) statement
[sound](#) statement

OPENTEXTFILE

- Syntax:** OPENTEXTFILE file
- Description:** The `opentextfile` statement imports data from a text file, another database, or a variable into the current database. If the text contains one or more [html tables](#) it will import the first table found.
- Parameters:** This statement has one parameter: file.
- file** identifies the file you want to open. If the file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "Disk:Accounting:Invoice".
- If file begins with a + symbol, Panorama will append the file to the current database, with the first field appended to the first field, second field appended to second field, etc.
- If file begins with a & symbol, Panorama will replace the data in the current database with the data in the file. The fields will be replaced in order, with the first field replacing the first field, second field replacing the second field, etc.
- Action:** This statement imports into an existing database. Unlike the [openfile](#) statement, the `opentext` statement treats all files as text files to be imported. This allows you to import files with any extension on Windows PC systems.
- Examples:** The example procedure procedure below opens the [Price List](#) and [Telemetry](#) text files.
- ```
opentextfile "Price List.html"
opentextfile "Telemetry.ini"
```
- Views:** This statement may be used in any view
- See Also:** [openfile](#) statement  
[importusing](#) statement  
[openfiledialog](#) statement  
[fileinfo\(\)](#) function  
[fileload\(\)](#) function  
[folder\(\)](#) function  
[folderpath\(\)](#) function  
[import\(\)](#) function  
[importcell\(\)](#) function  
[info\("files"\)](#) function  
[html tables](#)

# OUTLINELEVEL

**Syntax:** OUTLINELEVEL level

**Description:** The **outlinelevel** statement expands or collapses the entire database to show a specific level of data or summary level. In other words, this statement lets you select if you want to work with the forest or with the trees.

**Parameters:** This statement has one parameter: level.

**level** is a text item that specifies the minimum summary level to be displayed. This value may be either "Data" or a summary level from "1" (lowest level) and "7" (highest level summary).

If the level parameter is the word dialog (no quotes), the procedure will stop and display the standard **Outline Level** dialog. The user may select a level using the buttons. The procedure will then select the level and continue.

**Action:** This statement performs the same action as the **Outline Level** command in the **Sort** menu.

**Examples:** This example calculates summaries for cities and states, then displays the summary information. The original data is hidden.

```

field State
group
field City
group
field Amount
total
outlinelevel "1"

```

To see the original data and the summaries, use this example.

```
outlinelevel "data"
```

**Views:** This statement may be used in any view.

**See Also:** [group](#) statement  
[removesummaries](#) statement  
[removedetail](#) statement  
[summarylevel](#) statement  
[info\("summary"\)](#) function

# OVERFLOW(...)

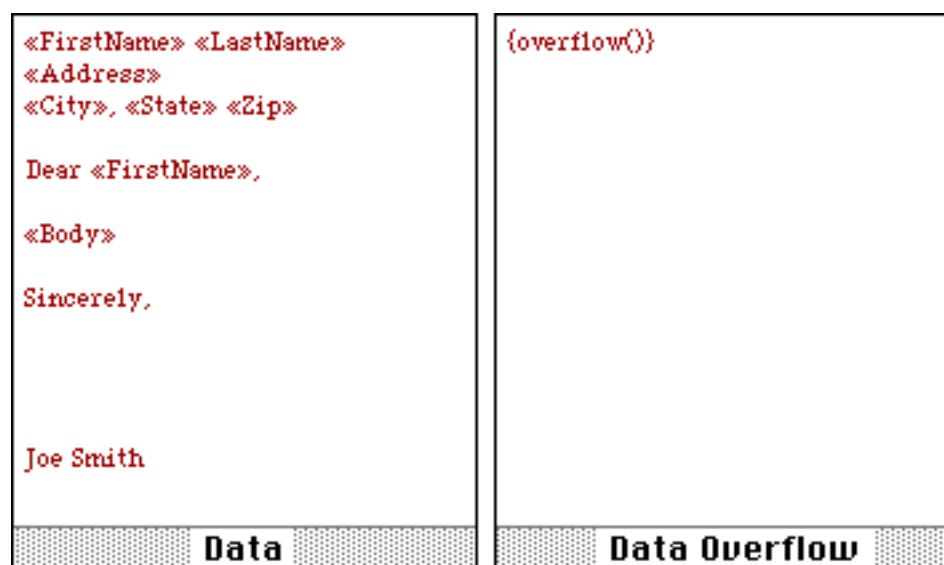
**Syntax:** OVERFLOW()

**Description:** The `overflow()` function is used with auto-wrap text objects and an overflow report tile to print text that won't fit on a single page. For example, you can use this function to help print multiple page letters.

**Parameters:** This function has no parameters.

**Result:** The `overflow()` function returns a text type data item.

**Examples:** The most common application for the `overflow()` function is to print multiple page letters. To set up a multiple page letter you'll need two report tiles: a data tile and an overflow tile. The overflow tile is labeled (...) in the Specialized Tile dialog. Both of these tiles should be large enough that only one record will print per page (usually somewhere near 8.5" by 11", perhaps 8" by 10"). Each of these two tiles can have various graphic objects on them, but each should have a single large auto-wrap text object that holds all of the text that is designed to overflow from page to page. (Important note: the overflow feature only works if the auto-wrap text objects are filled with NONE. If the text objects are filled with white or any other pattern the text will not overflow properly.) The auto-wrap text object on the data tile should have data cells and/or formulas merged into it.



When the form is printed Panorama will print as much of this text as it can fit on the first page (within the boundaries of the auto-wrap text object on the data tile). The rest of the text is set aside so that it can be printed on the overflow tile. (If there is no leftover text then Panorama is done and the overflow tile is not used. The auto-wrap text object on the overflow tile should have only one formula in it: `overflow()`. The `overflow()` function picks up the text that was leftover from the previous page. If the rest of the text doesn't fit on the overflow page the whole process will be repeated over and over again (using the same overflow tile) until all the text is printed.

**Errors:** This function does not produce any errors.

**See Also:** [extrapages\(\)](#) function

# PAGESETUP

**Syntax:** PAGESETUP

**Description:** The `pagesetup` statement displays the page setup dialog, allowing the page setup options to be changed for the current window. This is the same as choosing Page Setup from the File menu.

**Parameters:** This statement has no parameters.

**Examples:** The most common reason to use the `pagesetup` statement in a procedure is to simulate the **Page Setup** command in your own custom **File** menu. Here are the statements to use in your `.CustomMenu` procedure. (You could also trigger page setup with a button.)

```
if info("trigger") beginswith "Menu.File.Page Setup"
 pagesetup
 stop
endif
```

**Views:** This statement may be used in any view.

**See Also:** [print](#) statement  
[printpreview](#) statement  
[printonerecord](#) statement  
[printonemultiple](#) statement

# PANORAMA CGI

**Background:** Panorama 3.1 includes a CGI (Common Gateway Interface) that interfaces between Panorama and WebSTAR (or any WebSTAR compatible server). Using this CGI you can write procedures to make your Panorama databases accessible from any web browser anywhere on the web (NetScape Navigator/Communicator, Microsoft Internet Explorer, etc.).

**Installation:** To install the Panorama CGI you must copy at least three files into the WebSTAR folder on your hard disk:

- 1) Panorama CGI (a Panorama database)
- 2) Panorama.cgi (an application)
- 3) The Panorama database(s) you want to interface to the web

If you don't want to copy the actual Panorama database itself, you can create an alias and copy the alias into the WebSTAR folder.

Before you can access any databases from the web you must make sure that the Panorama.cgi application and the Panorama CGI database are both open. To ensure that these items are automatically opened simply create aliases to them and place the aliases in the Startup folder inside the System folder. You may also want to place aliases to your Panorama databases in the Startup folder.

An alternative method for opening your databases and the Panorama CGI database is to create an .AutoLoad file set (see documentation for this feature elsewhere in this document). This file set will be in the WebSTAR folder, so you will need to create an alias to the file set in the Panorama folder. This will cause Panorama and all necessary databases to run whenever the Panorama.cgi is accessed. (You must still place an alias to the Panorama.cgi application in the Startup folder, and may wish to place an alias to Panorama in the Startup folder as well.

**URL's:** To use the Panorama CGI from a web browser you must use a URL something using this pattern.

```
http://$Panorama.cgi~<database>~<procedure>
```

For example, suppose your URL is [www.acme.com](http://www.acme.com) and you have created a database called Registration that you want to interface to the web. This database contains a procedure called Recent that generates HTML.

```
http://www.acme.com$Panorama.cgi~Registration~Recent
```

You can enter this URL manually in your browser (handy for debugging), but for most applications you will code the URL into a web page using the `<A HREF="URL">` tag.

**Generating** The procedure specified by your URL must create a page of HTML and place the page into a pre-defined global variable called `cgiHTML`. This very simple example creates an HTML page that simply displays the database name and the number of records in the database.

```
cgiHTML=
{<HTML>
<HEAD>
<TITLE>Record Count</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
```

```

The }+
info("databasename")+
 { database contains }+
 pattern(info("records"),"# record~.")+
 {</BODY>
</HTML>
}

```

The arraybuild statements are very useful for taking the content of a database and turning it into HTML format. This example generates a list of names and phone numbers in list element format.

```

local phoneList
arrayselectedbuild phoneList,¶," ",
 ""+Last+", "+First+" "+Phone+¶
cgiHTML=
 {<HTML><HEAD><TITLE>Phone List</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Departmental Phone Directory</H1>

}+phoneList+
{
</BODY></HTML>}

```

Here is a similar example that builds an HTML table from a price list database.

```

local priceList
arrayselectedbuild priceList,¶," ",
 "<TR><TD>"+
 OrderNum+"</TD><TD>"+
 Description+"</TD><TD>"+
 Price+"</TD></TR>"+¶
cgiHTML={<HTML><HEAD><TITLE>Price List</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
<H1>Price List</H1>
<table>
<tr><td>Order#</td><td>Description</td><td>Price</td><tr>
}+priceList+
{</table>
</BODY></HTML>}

```

## Processing

The Panorama CGI pre-processes input from an HTML form, making that input easily accessible to your procedure. To review HTML forms, each input item (text, button, pop-up menu, etc.) has both a name and a value. The Panorama CGI has a procedure for retrieving the form data values: the `.cgiParameter` procedure. This procedure itself has two parameters:

**farcall "Panorama CGI",.cgiParameter,<input item name>,<input value>**

You must supply the input item name. The procedure will take this name and retrieve the input value. This value will be placed in the field or variable specified by the second parameter. Here is an example of a procedure that uses an HTML form to add new data to a Guest Book database.

```

addrecord
farcall "Panorama CGI",.cgiParameter,"Name",Name
farcall "Panorama CGI",.cgiParameter,"Address",Address
farcall "Panorama CGI",.cgiParameter,"City",City
farcall "Panorama CGI",.cgiParameter,"State",State
farcall "Panorama CGI",.cgiParameter,"Zip",Zip
farcall "Panorama CGI",.cgiParameter,"Comments",Comments
save
cgiHTML={<HTML><HEAD><TITLE>Thanks</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF">
 Thanks for signing into our guest book!
</BODY></HTML>}

```

Notice that the procedure saves the database immediately after making the changes. This is a good idea because you never know when this database will be accessed again. Also notice that the procedure generates an HTML page, in this case thanking the user for their input. Your procedure must always generate an HTML page.

## Processing

The Panorama CGI is capable of keeping a log of its activity. To keep a log of all activity, open the PanoramaCGI Log. When this database is open, Panorama will keep a log of every request made to the CGI.

To see more detail on a particular log item, simply double click on the item. The window displays the exact time and date, URL and form parameters (if any).

To keep a log of CGI errors, open the PanoramaCGI Error Log. This database looks very similar to the PanoramaCGI Log file, but also has a column for the error message. When this database is open Panorama keeps a log of any errors that occur in CGI procedures (type mismatch, etc.). This log can help you track down any problems in the HTML generation procedures you create. As with the PanoramaCGI Log file, simply double click on any line to see the full detail for that item, including the exact time and date, URL, error message, and form parameters (if any).

## See Also:

[arraybuild](#) statement  
[arrayselectedbuild](#) statement  
[htmlencode\(\)](#) function  
[exportcell\(\)](#) function  
[onerror](#) statement



# PARAMETER(...)

**Syntax:** `PARAMETER(parameter#)`

**Description:** The `parameter()` function is used to transfer data between a main procedure and a subroutine. The main procedure can set up one or more data item parameters as part of the call statement. The subroutine can retrieve and use these data items using the `parameter()` function.

**Parameters:** This function has one parameter: `parameter#`.

`parameter#` is a number specifying what parameter you want to retrieve. All parameters are numbered, starting with 1 (1, 2,3, 4, etc.).

**Result:** This function returns a data item. This data item may be text or numeric, depending on what kind of data is passed to the subroutine.

**Examples:** Here is a main procedure that calls the subroutine `.GetNumber` with two parameters. The two parameters are highlighted in orange below:

```

local addCount
addCount=1
call .GetNumber, "Add how many records?", addCount
loop
stoploopif addCount=0
addrecord
addCount=addCount-1
while forever

```

Now let's look at the `.GetNumber` subroutine, which retrieves these two parameters with the `parameter()` function.

```

local temptext
temptext=str(parameter(2))
gettext parameter(1),temptext
setparameter 2,val(temptext)

```

**Errors:** **Parameter does not exist.** This error occurs if there is no parameter with the specified number.

**See Also:** [call](#) statement  
[farcall](#) statement  
[setparameter](#) statement

# PASTE

- Syntax:** PASTE
- Description:** The `paste` statement pastes the contents of the clipboard into the database, usually into the current cell. The old contents of the current cell are lost. If the current cell has one or more automatic formulas associated with it, these formulas will be calculated.
- Parameters:** This statement has no parameters.
- Action:** This statement has the same effect as choosing the **Paste** command from the Edit menu.
- Examples:** This example copies the contents of the clipboard into the `Qty` field in the current database.

```
field Qty
paste
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[pastecell](#) statement  
[pasterecord](#) statement  
[clipboard\(\)](#) function

# PASTECELL

**Syntax:** PASTECELL

**Description:** The `paste` statement pastes the contents of the clipboard into the current cell. The old contents of the current cell are lost. If the current cell has one or more automatic formulas associated with it, these formulas will be calculated.

**Parameters:** This statement has no parameters.

**Action:** This statement has the same effect as choosing the **Paste** command from the Edit menu. The `paste` statement does the same job as the `paste` statement, but will work correctly in all views.

**Examples:** This example copies the contents of the clipboard into the Qty field in the current database.

```
field Qty
paste
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[pasterecord](#) statement  
[clipboard\(\)](#) function

# PASTEFORM

**Syntax:** PASTEFORM

**Description:** The `pasteform` statement creates a new form. The new form is not empty, but is copied from the clipboard.

**Parameters:** This statement has no parameters.

**Action:** This statement creates a new form from the contents of the clipboard. The clipboard must be set up in advance with the `copyform` statement. Using the `copyform` and `pasteform` statements creates an exact duplicate of a form, including the graphic objects on the form, the page setup, the custom menu setup, and form and report preferences.

This statement has the same effect as choosing the **Paste Form** command from the Edit menu (graphics mode).

**Examples:** This example will make a copy of the form **Report Template**. The `pasteform` statement will stop and ask the user what to call the new form.

```
openform "Report Template"
copyform
pasteform
```

**Views:** This statement may be used in a Form view.

**See Also:** [copyform](#) statement  
[openform](#) statement  
[copy](#) statement  
[cut](#) statement  
[paste](#) statement

# PASTERECORD

**Syntax:** PASTERECORD

**Description:** The `pasterecord` statement inserts a new record into the database. The new record will contain the contents of the clipboard. The clipboard must contain tab delimited text (see `copyrecord`).

**Parameters:** This statement has no parameters.

**Action:** This statement has the same effect as clicking on the **Paste Record** tool on a tool palette (when available).

**Examples:** This example copies the current record from the Current Records database, goes to the Old Records database, pastes the copy into that file, and returns to Current Records.

```
copyrecord
window "Old Records"
pasterecord
window "Current Records"
```

**Views:** This statement may be used in any view.

**See Also:** [clear](#) statement  
[clearcell](#) statement  
[clearrecord](#) statement  
[copy](#) statement  
[copycell](#) statement  
[copyrecord](#) statement  
[cut](#) statement  
[cutcell](#) statement  
[cutrecord](#) statement  
[deleteabove](#) statement  
[deleteall](#) statement  
[deleterecord](#) statement  
[paste](#) statement  
[pastecell](#) statement

# PATHID(...)

**Syntax:** PATHID(folder)

**Description:** The `pathid()` function creates a binary data item that unambiguously describes the location of a folder on the hard disk. This pathid can be used in other functions and statements. Note: This function is identical to the `folder()` function.

**Parameters:** This function has one parameter: folder.

**folder** is a complete description of the path to this folder, for example `HD:System Folder:Extensions:`.

**Result:** This function returns a 6 byte binary data item that unambiguously describes the location of the folder. However, if the folder does not exist the function returns an empty binary data item ("").

**Examples:** This example checks to see if a folder exists.

```
if ""=pathid("HD:Panorama Accounting:Order Entry:")
 message "You do not have the Order Entry option."
 stop
endif
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder parameter.

**See Also:** [pathstr\(\)](#) function  
[listfiles\(\)](#) function  
[dbinfo\(\)](#) function  
[info\("panoramafolder"\)](#) function  
[info\("systemfolder"\)](#) function  
[openfiledialog](#) statement  
[savefiledialog](#) statement

# PATHSTR(...)

**Syntax:** PATHSTR(folder)

**Description:** The `pathstr()` function takes a path id and converts it to a description of the path to that folder. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. **Note:** This function is identical to the `folderpath()` function.

**Parameters:** This function has one parameter: `folder`.

**folder** is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder.

**Result:** This function returns complete description of the path to this folder, for example  
`HD:System Folder:Extensions:`

**Examples:** This example displays the folder the currently running copy of Panorama is located in.

```
message "This database is in the "+
 pathstr(info("panoramafolder"))+" folder."
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the folder parameter. This error can also occur if the folder parameter is more or less than 6 bytes long.

**See Also:** [pathid\(\)](#) function  
[listfiles\(\)](#) function  
[dbinfo\(\)](#) function  
[info\("panoramafolder"\)](#) function  
[info\("systemfolder"\)](#) function  
[openfiledialog](#) statement  
[savefiledialog](#) statement

# PATTERN(...)

**Syntax:** PATTERN(number,pattern)

**Description:** The `pattern()` function converts a number into text using a pattern (see [numeric patterns](#)).

**Parameters:** This function has two parameters: `number` and `pattern`.  
**number** is the number that you want to convert to text.  
**pattern** is text that contains a pattern for formatting the number (see [numeric patterns](#)).

**Result:** This function returns an item of text containing the formatted number.

**Examples:** The `pattern()` function is very useful when used in a formula in an auto-wrap text object or Text Display SuperObject™. Here is a pattern that will output prices with a dollar sign and with a comma every three digits (for example \$4,285.25)

```
pattern(Price,"$#,##")
```

For more information on the variety of patterns that are possible, see [numeric patterns](#).

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the pattern parameter.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the number parameter.

**See Also:** [numeric patterns](#)  
[str\(\)](#) function



# PAUSE

**Syntax:** PAUSE state

**Description:** The `pause` statement temporarily pauses a procedure. The procedure can be restarted from this point with the `resume` statement.

**Parameters:** This statement has one parameter: state.

**state** is a global variable. This global variable is used to store information about where and how the procedure was paused. Later, the `resume` statement (which is usually used as part of another procedure) uses the same variable to know how and where to re-start the original procedure.

**Action:** This statement temporarily pauses a procedure. Later the procedure can be started up again from this point. For example you might want to pause a procedure while the user fills in a dialog, then continue when the user presses the Ok button.

**Examples:** The example opens a form called Sound & Video. To the user, this form will appear to be a standard dialog box. Once the dialog box is open the procedure will pause, allowing the user to type values into the box, select checkboxes, etc.

```
global dialogPause
setwindow 100,100,300,400," "
opendialog "Sound & Video"
pause dialogPause
/* pause here for user to fill in dialog */
closewindow
if info("trigger") contains "Ok" or
 info("trigger")="Key.Enter"
 playsound dialogSound
endif
```

If you want a button in the Sound & Video dialog to close the dialog, that button should be linked to a procedure that contains the following statement:

```
resume dialogPause
```

The `resume` statement causes the original procedure to continue, starting from right after the `pause` statement. In this case the procedure closes the dialog and plays a sound if the user pressed Ok. The advantage of using the `pause` and `resume` statements is that a single dialog may be used with many procedures. The buttons in the dialog aren't actually linked to any specific procedure, they simply let whatever procedure opened the dialog in the first place continue. In Panorama 3.0, the `resume` statement had to reference the same global variable as the `pause` statement. In Panorama 3.1 and later, this is no longer required. In fact, Panorama ignores the variable name you supply (although you must supply a valid variable name).

**ENTER key:** Many dialogs allow the user to press the **ENTER** or **RETURN** key instead of pressing the Ok button. Starting with Panorama 3.1, this is possible using the `pause` and `resume` statements. In fact, Panorama will do this automatically if you create your dialog using Text Editor SuperObjects and use the `pause` and `resume` statements. When the user presses the ENTER key (or the RETURN key if the Text Editor SuperObject is one line high and is set to terminate editing when RETURN is pressed) Panorama will automatically `resume`, even though your `resume` procedure has not been triggered. When the pro-

cedure starts up after the pause statement, the [info\("trigger"\)](#) function will return Key.Enter as shown in the example above. (This is the reason why the variable after the resume statement is now ignored. Panorama must keep track of the variable itself so that it can resume properly when the ENTER or RETURN key is pressed.)

**Nested**

It is possible to nest dialogs (i.e. one dialog is activated within another dialog) when using pause and resume. If you do this, you must be careful to use a separate variable in the pause statement of the nested dialog. (The variable used in the resume statement does not matter, because Panorama ignores it.)

**Views:**

This statement may be used in any view, but it usually only make sense in the form view.

**See Also:**

[resume](#) statement  
[opendialog](#) statement  
[info\("trigger"\)](#) function

# PERMANENT

**Syntax:** `PERMANENT variables`

**Description:** The `permanent` statement creates one or more permanent variables. Unlike other variables which disappear when Panorama is closed, permanent variables are saved as part of the database they were created in. Note: Starting with Panorama 3.1, permanent variables act like [fileglobal](#) variables. In other words, they can only be accessed from within the database in which they were created (except with the [grabfilevariable\(\)](#) function).

**Parameters:** This statement has one parameter: `variables`.

`variables` is a list of variables to be created. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

**Action:** This statement creates one or more permanent variables. The variables will be permanently attached to the current database. Whenever that database is saved the variables are saved along with it. When the database is opened these variables will be automatically re-created and filled with the previous data. Permanent variables are very useful for preferences, passwords, etc. Note: The [dbinfo\(\)](#) function can create a list of the permanent variables associated with a database.

**Examples:** A permanent variable is forever, unless you destroy it with the [unpermanent](#) statement. The example creates two permanent variables, `Sales Tax Rate` and `EarlyDiscount`.

```
permanent «Sales Tax Rate», EarlyDiscount
```

Permanent variables may be used just like any other variable. You may change the value of the variable with an assignment, like this:

```
«Sales Tax Rate»=.075
```

When the database is saved, the `Sales Tax Rate` will also be saved. Note: If you close the file without saving, the new `Sales Tax Rate` will not be saved!

**Views:** This statement may be used in any view.

**See Also:** [unpermanent](#) statement  
[global](#) statement  
[fileglobal](#) statement  
[local](#) statement  
[globalize](#) statement  
[dbinfo\(\)](#) function  
[grabfilevariable\(\)](#) function  
[undefine](#) statement

# PLAYSOUND

- Syntax:** `PLAYSOUND sound`
- Description:** The `playsound` statement plays a digitally recorded sound in a resource file. The resource file must be opened with the `opensound` statement before the sound can be played. On Macintosh systems sounds can be recorded using the **Sound** control panel.
- Parameters:** This statement has one parameter: `sound`  
`sound` is the name of the resource that contains the sound.
- Action:** This statement plays a sound from the resource file currently opened by an [opensound](#) statement. A sound resource contains one or more SND resource for each sound and each SND resource is given a name. Sound resources may be created with a sound program like Farallon's MacRecorder or shareware programs like Sound->snd or Sound-Mover. System 7 users may even create sound resources with the Sound control panel and a Macintosh which supports a microphone.
- Examples:** This example opens a sound resource called Bird Calls, plays the sound titled Robin and then closes Bird Calls.

```
opensound "Bird Calls"
playsound "Robin"
closesound
```

This example asks the user a question to answer, opens a sound resource called Remarks, and depending on the answer plays the sound called **Correct** or **Wrong**. It then closes Remarks.

```
local Answer,Reply
message "The Alamo is in:"+¶+"1. New Mexico"+
 ¶+"2. Texas"+¶+"3. California"
gettext "Is the answer 1, 2, or 3",Answer
opensound "Remarks"
case Answer = "1"
 Reply = "Wrong"
case Answer = "2"
 Reply = "Correct"
case Answer = "3"
 Reply = "Wrong"
endcase
playsound Reply
closesound
```

- Views:** This statement may be used in any view even if there are no visible windows open.
- See Also:** [opensound](#) statement  
[closesound](#) statement  
[sound](#) statement

# PMT(...)

**Syntax:** PMT(rate,payments,amount,fv,begin)

**Description:** The **pmt()** function (short for payment) calculates the periodic payment required to pay off a loan.

**Parameters:** This function has five parameters: rate, payments, amount, fv and begin.

**rate** is the interest rate of the loan (per period). For example, if there is one payment per year, this is the annual percentage rate. If there is one payment per month, then this is the monthly percentage rate.

**payments** is the number of payments required to pay off the loan. For example, if this is a 36 month loan with one payment per month, this value is 36. If this is a 30 year loan with one payment per month this value is 360.

**amount** is the amount being borrowed, the original amount of the loan. For example, if you borrow \$22,000 to purchase a car, the loan amount is 22000.

**fv** (future value) is the value of the loan at the end of the period. This is almost always zero.

**begin** specifies whether payments are made at the beginning (1) or end of each period (0). Most loans are paid at the end of each period (the first payment is made at the end of the first month, etc.) so this value should be zero

**Result:** The result of this function is always a numeric floating point value.

**Examples:** If the payment period is annually the calculation is simple. Suppose you take out a \$50,000 loan at 12% for 10 years, with one payment per year. This formula will calculate the payments.

```
pmt(.12,10,50000,0,0)
```

Most loans are paid more frequently than once a year...usually once a month. To calculate the payments for such a loan you must convert the annual percentage rate into a monthly percentage rate by dividing by 12. Suppose you are taking out a 36 month loan of \$20,000 to purchase a car. If the annual interest rate is 13.5%, here is the formula for calculating the monthly payments:

```
pmt(0.135/12,36,20000,0,0)
```

Our final example is for a \$180,000 real estate loan for 30 years at a fixed rate of 9%. In this case the number of years is multiplied by 12 to get the number of monthly payments, and the annual interest rate is divided by 12 to calculate the monthly interest rate.

```
pmt(.09/12,30*12,180000,0,0)
```

Of course the **pmt()** function only works with fixed interest rates.

**Notes:** Here is the formula that Panorama uses to calculate payments.

$$\text{payment} = \frac{\text{rate} * (\text{amount} * (1 + \text{rate})^{\text{periods}} - \text{fv})}{(1 + \text{rate} + \text{begin} * (1 + \text{rate})^{\text{periods}} - 1)}$$

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value with this function, for example `pmt("12%", )`. If you have a number in a text item you must convert the text to a numeric value before calculating the payments, for example `pmt(val("12%"[1,-2]), )`.

**See Also:** [fv\( function](#)  
[pv\( function](#)

# POINT(...)

**Syntax:** POINT(v,h)

**Description:** The `point()` function combines vertical and horizontal co-ordinates into a single number that describes the position of a point (see [graphic coordinates](#)).

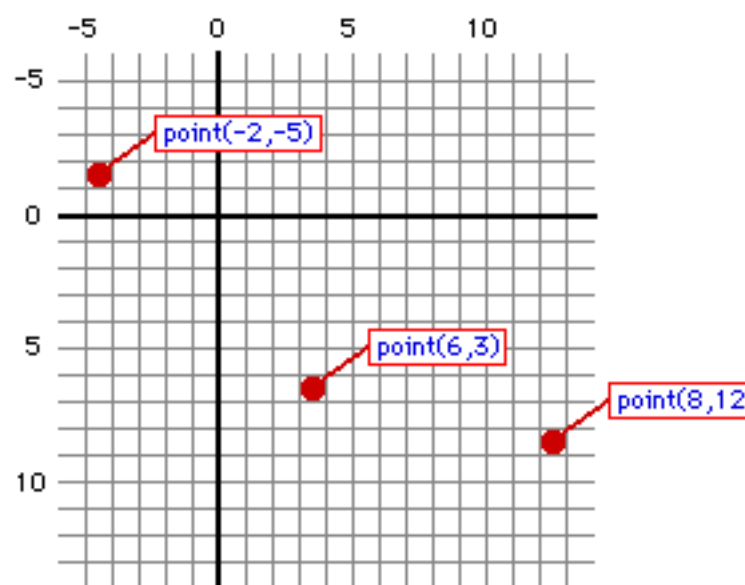
**Parameters:** This function has two parameters: `v` and `h`.

`v` is the vertical position of the point. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.) All dimensions are in pixels (1 pixel=1/72 inch).

`h` is the horizontal position of the point. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.) All dimensions are in pixels (1 pixel=1/72 inch).

**Result:** This function returns a number (an integer) that describes the location of the point. You can use this number in any function or statement that accepts a point as a parameter.

**Examples:** The greatly magnified illustration below shows several sample points and the functions used to create them. Note that the actual point “hangs” down and to the right of the co-ordinate grid lines.



The procedure below displays a message if you manage to click on a spot exactly 50 pixels from the top of the screen and 100 pixels from the left edge of the screen.

```
if info("click") = point(50,100)
 message "You hit the secret spot!"
else
 beep
endif
```

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the `v` or `h` parameters.

**See Also:**

[xytoxy\(\)](#) function  
[v\(\)](#) function  
[h\(\)](#) function  
[info\("click"\)](#) function  
[info\("mouse"\)](#) function  
[rectangle\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[rright\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function



# POPUP

**Syntax:** `POPUP menu,v,h,initial,result`

**Description:** The `popup` statement makes a pop-up menu appear anywhere on the screen, without a Pop Up Menu SuperObject. The popup statement is usually used with a standard transparent button with the click/release option turned off.

**Parameters:** This statement has five parameters: `menu`, `v`, `h` `initial` and `result`.

**menu** is either a list of items to be included in the menu or a menu number.

If the menu parameter is a list, that list must contain the actual text of the menu, with each menu item separated by a carriage return (a text array). To create a dividing line in the menu use "-" as a menu item.

If the menu parameter is a number, that number must correspond to a menu in a resource file. The resource file must be open (see [openresource](#)). This option allows you to pre-build menus with a resource editor like ResEdit.

**v** is the vertical position of the pop-up menu. The position is measured in pixels from the top of the form. These are form relative co-ordinates, 72 pixels per inch (see [xytoxy\(\)](#), [info\("click"\)](#), and [getlocalbutton](#)).

**h** is the horizontal position of the pop-up menu. The position is measured in pixels from the left edge of the form. These are form relative co-ordinates, 72 pixels per inch (see [xytoxy\(\)](#), [info\("click"\)](#), and [getlocalbutton](#)).

**initial** is the initial value of the menu. This value should be one of the items in the menu. The menu will pop up with the arrow over this item. For example, suppose the pop-up menu contains the items Red, Blue and Green and the current value is Blue. "Blue" should be the initial value.

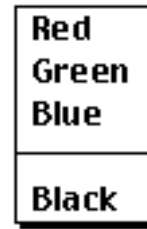
**result** is a variable where the user's choice from the pop-up menu will be stored. If the user releases the mouse without making a choice, the result will be left untouched.

**Action:** This statement is designed to be used in conjunction with a transparent button that has the click/release option turned off. In other words, as soon as someone clicks on the button, the procedure triggers and the pop-up menu pops up. (Note: It is usually easier to use a Pop-Up Menu SuperObject instead of using the popup statement. However, you must use the popup statement if the menu items cannot be calculated with a formula, but require one or more procedure statements to compute.)

**Examples:** This example allows the user to choose a color.

```
local btop,bleft,bheight,bwidth,NewColor
getlocalbutton btop,bleft,bheight,bwidth
NewColor=" "
popup "Red"+¶+"Green"+¶+"Blue"+¶+"(-"+¶+"Black",
 btop,bleft,"Blue",NewColor
```

The pop-up menu from this statement will look like this:



**Views:** This statement may be used in a Form view.

**See Also:** [popupbynumber](#) statement  
[openresource](#) statement  
[text arrays](#)

# POPUPBYNUMBER

**Syntax:** `POPUPBYNUMBER menu,v,h,initial,result`

**Description:** The `popupbynumber` statement makes a pop-up menu appear anywhere on the screen, without a Pop Up Menu SuperObject. The `popupbynumber` statement is usually used with a standard transparent button with the click/release option turned off.

**Parameters:** This statement has five parameters: `menu`, `v`, `h`, `initial` and `result`.

**menu** is either a list of items to be included in the menu or a menu number.

If the `menu` parameter is a list, that list must contain the actual text of the menu, with each menu item separated by a carriage return (a text array). To create a dividing line in the menu use "-" as a menu item.

If the `menu` parameter is a number, that number must correspond to a menu in a resource file. The resource file must be open (see [openresource](#)). This option allows you to pre-build menus with a resource editor like ResEdit.

**v** is the vertical position of the pop-up menu. The position is measured in pixels from the top of the form. These are form relative co-ordinates, 72 pixels per inch (see [xytoxy\(\)](#), [info\("click"\)](#), and [getlocalbutton](#)).

**h** is the horizontal position of the pop-up menu. The position is measured in pixels from the left edge of the form. These are form relative co-ordinates, 72 pixels per inch (see [xytoxy\(\)](#), [info\("click"\)](#), and [getlocalbutton](#)).

**initial** is the initial value of the menu. This value should be a number from one to the number of items in the menu. The menu will pop up with the arrow over this item. For example, if the initial value is 3 the menu will pop-up over the third item.

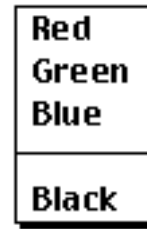
**result** is a variable where the user's choice from the pop-up menu will be stored. The result is returned as a number: 1 for the first menu item, 2 for the second menu item, etc. If the user releases the mouse without making a choice, the result will be left untouched.

**Action:** This statement is designed to be used in conjunction with a transparent button that has the click/release option turned off. In other words, as soon as someone clicks on the button, the procedure triggers and the pop-up menu pops up. (Note: It is usually easier to use a Pop-Up Menu SuperObject instead of using the `popup` statement. However, you must use the `popup` statement if the menu items cannot be calculated with a formula, but require one or more procedure statements to compute.)

**Examples:** This example allows the user to choose a color.

```
local btop,bleft,bheight,bwidth,NewColor
getlocalbutton btop,bleft,bheight,bwidth
NewColor=0
popupbynumber "Red"+¶+"Green"+¶+"Blue"+¶+"(-"+¶+"Black",
 btop,bleft,3,NewColor
```

The pop-up menu from this statement will look like this:



**Views:** This statement may be used in a Form view.

**See Also:** [popup](#) statement  
[openresource](#) statement  
[text arrays](#)

# POPUPSTYLE

- Syntax:** `POPUPSTYLE font,size,forecolor,backcolor`
- Description:** The `popupstyle` statement allows a procedure to control the font, text size, and color of pop-up menus created with the `popup` and `popupbynumber` statements.
- Parameters:** This statement has four parameters: `font`, `size`, `forecolor` and `backcolor`.
- font** is the name of the font to use in the menu, "Chicago", "Geneva", "Helvetica", etc.
- size** is the size of the text in the menu, in points. Standard menus are 12 pt. The smallest practical size for most fonts is 9 pt.
- forecolor** is the color of the text in the menu. You can generate a color with the `rgb()` or `hsb()` functions. If you supply empty text ("" ) for the foreground color it will default to black.
- background** is the color of the background of the menu. You can generate a color with the `rgb()` or `hsb()` functions. If you supply empty text ("" ) for the background color it will default to white.
- Action:** This statement controls the appearance of pop-up menus created with the `popup` and `popupbynumber` statements. It does not affect SuperObject pop-up menus. The procedure can control the font and size of the text in the menu, as well as the color. The `popupstyle` statement should be placed immediately before the `popup` or `popupbynumber` statements.
- Examples:** The procedure will display a pop-up menu with 9pt Geneva red text on a gray background. (Note: This procedure is designed to be attached to a transparent button with the click/release option turned off. See the `popup` statement for more information.)
- ```

local btop,bleft,bheight,bwidth,ShipBy
getlocalbutton btop,bleft,bheight,bwidth
NewColor=""
PopUpStyle "Geneva",9,
rgb(65535,0,0),rgb(50000,50000,50000)
popup "Air"+¶+"Rail"+¶+"Sea"+¶+"Ground",btop,bleft,"Air",ShipBy

```
- Views:** This statement may be used in a Form view
- See Also:** [popup](#) statement
[popupbynumber](#) statement
[colors](#)
[rgb\(\)](#) function
[hsb\(\)](#) function

PRINT

Syntax: PRINT options

Description: The **print** statement prints the all selected records to the current printer. (Use the Chooser to select a different printer.)

Parameters: This statement has one parameter: options

options is a item that specifies the printing options. However, the Apple ROM's do not allow Panorama to control printing options, so there are only two valid options: empty text (") or the keyword **dialog**.

When **options** is the word **dialog** Panorama will pause the procedure and present the user with the standard **Print...** dialog. This allows the user to select the number of copies, print quality, and any other options that may be available for the currently selected printer.

When **options** is empty text (") Panorama simply use whatever print options were used the last time this window was printed. (Note: Depending on the printer driver software your printer uses, some options may not be saved from print to print. These options will use default settings.)

Action: This statement is the same as choosing the **Print** tool from the **File** menu.

Examples: This example prints a report.

```
openform "90 Day Report"
select Date>today()-90
print dialog
selectall
closewindow
```

Views: This statement may be used in any view.

See Also: [printonerecord](#) statement
[pagesetup](#) statement
[printonemultiple](#) statement
[printpreview](#) statement

PRINTONEMULTIPLE

Syntax: PRINTONEMULTIPLE variable,start,end,bump,copies

Description: The `printonemultiple` statement prints a form over and over again without advancing from record to record. Instead of advancing from record to record, a variable is incremented each time the form is printed. This statement is designed for printing calendars or thumbnails.

Parameters: This statement has five required parameters: `variable`, `start`, `end`, `bump`, and `copies`.

variable is the name of the variable you wish to increment each time the form is printed.

start is the beginning sequence number or date value. `start` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. `start` must be less than and not equal to `end`

end is the ending sequence number or date value. `end` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. `end` must be greater than and not equal to `start`.

bump is the increment value for your sequence. `bump` may be a number, a numeric variable, or a formula which results in a positive numeric integer. The `bump` value must be a positive integer for a numeric field. For a date field `bump` may also be one of the following:

```
"M" - for month (the M must be in quotes)
"Y" - for year (the Y must be in quotes)
1   - for day
7   - for week
```

copies is the number of times the form is printed for each sequence number. `copies` may be a number, a numeric variable, or a formula which results in a positive numeric integer (in most cases this value will be 1).

Action: This statement will print a form a predetermined number of times. Each printing of the form may be sequenced with incrementing integer or date values in a specified variable. Note: the `printonemultiple` statement does not actually print, but must be followed by a `printonerecord` statement

Examples: This example prints the next 3 months of a monthly calendar. The example assumes that the form Monthly Calendar will display the month specified by the variable CalendarDate.

```
global CalendarDate
openform "Monthly Calendar"
printonemultiple CalendarDate,today(),today()+90,"m",1
printonerecord dialog
closewindow
```

This example prints all the picture files in the current folder. The example assumes that the form `Picture Matrix` will display 20 pictures per page, probably using a `SuperMatrix` object. The picture in the top left corner of each is controlled by the global variable `PicNumber`.

```
global PicNumber,PicMax  
PicMax=array_size( listfiles("","PICT"),1)  
openform "Picture Matrix"  
printonemultiple PicNumber,1,PicMax,20,1  
printonerecord dialog  
closewindow
```

Views: This statement may be used in a Form view.

See Also: [printonerecord](#) statement
[print](#) statement
[pagesetup](#) statement
[printpreview](#) statement
[addlines](#) statement

PRINTONERECORD

Syntax: PRINTONERECORD options

Description: The `printonerecord` statement prints the currently active record to the current printer. (Use the Chooser to select a different printer.) This statement may only be used with a form window.

Parameters: This statement has one parameter: options

options is a item that specifies the printing options. However, the Apple ROM's do not allow Panorama to control printing options, so there are only two valid options: empty text (""), or the keyword `dialog`.

When options is the word `dialog` Panorama will pause the procedure and present the user with the standard **Print...** dialog. This allows the user to select the number of copies, print quality, and any other options that may be available for the currently selected printer.

When options is empty text (""), Panorama simply use whatever print options were used the last time this window was printed. (Note: Depending on the printer driver software your printer uses, some options may not be saved from print to print. These options will use default settings.)

Action: This statement is the same as choosing the **Print This Record** tool from the tool palette.

Examples: This example prints the current invoice.

```
openform "PaperInvoice"  
printonerecord dialog  
closewindow
```

Views: This statement may be used in the Form view.

See Also: [print](#) statement
[pagesetup](#) statement
[printonerecord](#) statement
[printpreview](#) statement

PRINTPREVIEW

Syntax: PRINTPREVIEW

Description: The `printpreview` statement opens a special preview window. This window displays a preview of the printed results for the current view. This is the same as choosing **Preview** from the File menu. The user can flip forward to see additional pages of the previewed report. When the user closes the preview window, the procedure continues with the statement after the `printpreview` statement.

Parameters: This statement has no parameters.

Examples: The most common reason to use the `printpreview` statement in a procedure is to simulate the **Print Preview** command in your own custom File menu. Here are the statements to use in your `.CustomMenu` procedure. (You could also trigger print preview with a button.)

```
if info("trigger") beginswith "Menu.File.Print Preview"  
    printpreview  
    stop  
endif
```

Views: This statement may be used in any view.

See Also: [print](#) statement
[pagesetup](#) statement
[printonerecord](#) statement
[printonemultiple](#) statement

PRINTUSINGFORM

Syntax: PRINTUSINGFORM file,form

Description: The `printusingform` statement allows the current database to be printed using a different form than the one currently being displayed. It is designed to be used in combination with the [print](#), [printonerecord](#), or [printpreview](#) statements.

Parameters: This statement has two required parameters: file and form.

file is the name of the database file that contains the form to be printed. The database file must be open. Usually the form will be in the current database, and in that case you can simply use an empty string ("") for the file name.

form is the name of the form to be printed.

Action: The print command normally prints whatever window is currently active. If you want to print a different window, you must first open that window and then print. The `printusingform` statement is another way to print an alternate form. Note: The `printusingform` statement may only be used when a form window is currently on top.

Examples: The procedure below will print **My Report**, even if another form is currently visible.

```
printusingform "", "My Report" print dialog
```

The procedure below will print **Standard Report #4** from the **Reports** database. Although the form is from the Reports database, the data will be from the current database. This usually only makes sense if the two databases have the same fields.

```
printusingform "Reports", "Standard Report #4"  
print dialog
```

Views: This statement may be used in a Form view.

See Also: [print](#) statement
[printonerecord](#) statement
[printpreview](#) statement

PROPAGATE

Syntax: PROPAGATE

Description: The **propagate** statement fills all the empty cells in the current field. Each empty cell is filled with the value of the first non-empty cell above it.

Parameters: This statement has no parameters.

Examples: This example computes the outstanding balance for each company in an invoice database. It uses the **propagate** statement to copy the address information from the data records into the newly created summary records.

```
field Company
groupup
field Balance
total
field Address
propagate
field City
propagate
field State
propagate
field Zip
propagate
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [propagateup](#) statement
[unpropagate](#) statement
[unpropagateup](#) statement

PROPAGATEUP

Syntax: PROPAGATEUP

Description: The `propagateup` statement fills all the empty cells in the current field. Each empty cell is filled with the value of the first non-empty cell below it.

Parameters: This statement has no parameters.

Examples: This example fills any empty dates with the date from the next record.

```
field Date  
propagateup
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [propagate](#) statement
[unpropagate](#) statement
[unpropagateup](#) statement

PV(...)

Syntax: PV(rate,periods,payment,balloon,begin)

Description: The **pv**(function (short for present value) calculates the present value of an income or debit stream of payments. Present value is an economic calculation that is the equivalent of a bird in the hand is worth two in the bush, or is that a bird in the hand is worth 1.67 in the bush? It's better to receive \$1000 right now than \$1000 next year, but how much better? The **pv**(function will tell you.

Parameters: This function has five parameters: **rate**, **periods**, **payment**, **balloon** and **begin**.

rate is the interest rate of the investment (per period). For example, if there is one payment per year, this is the annual percentage rate. If there is one payment per month, then this is the monthly percentage rate.

periods is the number of payments that will be made during the life of the investment. For example, if this is a 36 month investment with one payment per month, this value is 36. If this is a 30 year investment with one payment per month this value is 1080.

payment is the amount being invested each period. If this is money that is coming to you this value should actually be negative. For example, if you are receiving \$500 per month, the payment amount should be -500.

balloon (future value) is any lump sum payment at the end of the investment (actually the negative of the balloon payment). For example if you are receiving \$25,000 at the end of ten years, this value should be -25000.

begin specifies whether payments are made at the beginning (1) or end of each period (0).

Result: The result of this function is always a numeric floating point value.

Examples: If the payment period is annually the calculation is simple. Suppose someone offers to pay you either \$2,500 right now, or \$1000 per year for the next three years. Which should you take? The answer depends on the interest rate you can get for your money (and, of course, how much you need cash now, but that's another question). If you can invest your money at 7% then the three payments are worth \$2624 now (see below), but if you can get 10% then three payments later are only worth \$2486 now.

```
pv(.07,3,-1000,0,0)
```

Most investments are paid more frequently than once a year...usually once a month. To calculate the present value for such an investment you must convert the annual percentage rate into a monthly percentage rate by dividing by 12. Suppose someone offers you \$200 a month at the beginning of each month for 5 years, and you know that you can put your money into an investment that returns 13.5% annual interest. What is the cash payment that would be equivalent to that income stream?

```
pv(0.135/12,5*12,-200,0,1)
```

You can also use this function to calculate the current value of a single balloon payment, with no payments. Suppose someone offers to pay you \$50,000 5 years from now. You know that you can invest your money at 14% annual interest. This formula will calculate the value of the equivalent cash value of this \$50,000 future payment (\$25,968) :

```
pv(0.14,5,0,-50000,0)
```

Of course the pv(function only works with fixed interest rates.

Notes:

Here is the formula that Panorama uses to calculate present value.

$$\text{present value} = - \frac{\text{payment} * ((1 + \text{rate} * \text{begin}) * (1 + \text{rate})^{\text{periods} - 1}) + (\text{fv} * \text{rate})}{\text{rate} * (1 + \text{rate})^{\text{periods}}}$$

Errors:

Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value with this function, for example `fv("12%",)`. If you have a number in a text item you must convert the text to a numeric value before calculating the future value, for example `fv(val("12%"[1,-2]),)`.

See Also:

[pmt\(function](#)
[fv\(function](#)

QUARTER1ST(...)

Syntax: QUARTER1ST(date)

Description: The `quarter1st`(function computes the first day of a quarter.

Parameters: This function has one parameter: `date`.
`date` is a number representing the date.

Result: This function calculates the first day of the quarter. For example, if the date passed to this function is August 18, 1997, this function will return the date July 1, 1997. The date is returned as a number.

Examples: The example below selects the orders placed this quarter, then displays the count.

```
select
  OrderDate>quarter1st(today()) and
  OrderDate<today()
message str( info("records")+ " orders this quarter"
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

See Also: [monthlength](#)(function
[monthmath](#)(function
[week1st](#)(function
[month1st](#)(function
[year1st](#)(function
[date](#)(function
[datepattern](#)(function

QUIT

Syntax: QUIT

Description: The **quit** statement stops Panorama and returns to the Finder. As a side effect, it also stops the procedure! This command has the same effect as choosing the **Quit** command from the **File** menu.

The **quit** statement will not normally ask the user if they want to save changes in any open databases before stopping Panorama. However, if the **quit** statement is the last statement in the procedure, or is followed by a stop statement, it will ask the user, and if they say yes, save the files for them.

Parameters: This statement has no parameters.

Examples: Usually the only reason to use the **quit** statement in a procedure is to simulate the **Quit** command in your own custom **File** menu. Here are the statements to use in your `.CustomMenu` procedure. Since the **quit** statement is followed by a stop statement, it will ask the user to confirm the operation, just like the real **Quit** command.

```
if info("trigger") beginswith "Menu.File.Quit"  
    quit  
    stop  
endif
```

Views: This statement may be used in any view.

See Also: [save](#) statement
[saveall](#) statement
[stop](#) statement
[rtn](#) statement
[info\("changes"\)](#) function

RADIAN

Syntax: RADIAN

Description: The **radian** statement tells Panorama that all values in trigonometric functions should be treated as radian's rather than degree's. (Radian is the program's default).

Parameters: This statement has no parameters.

Action: This statement doesn't perform any visible action on its own. However, this is the only way to turn off the [degree](#) statement and have all attributes for any trig functions in the procedure treated as radian values. Panorama will revert back to **radian** after the procedure is finished; **radian** is the default setting.

Examples: This example tells Panorama to calculate the tan of 30 degrees and put the answer in Height and later on convert to radian's to calculate the tan of $\pi/2$ radian's and put the answer in Rheight.

```

degree
Height = tan(30)
...
...
radian
Rheight = tan( $\pi/2$ )

```

Views: This statement may be used in any view.

See Also: [arccos\(\)](#) function
[arccosh\(\)](#) function
[arcsin\(\)](#) function
[arcsinh\(\)](#) function
[arctan\(\)](#) function
[arctanh\(\)](#) function
[cos\(\)](#) function
[cosh\(\)](#) function
[sin\(\)](#) function
[sinh\(\)](#) function
[tan\(\)](#) function
[tanh\(\)](#) function

RADIX(...)

Syntax: RADIX(radix,text)

Description: The `radix()` function converts a text item containing a hex, octal, or binary number into a standard Panorama number (decimal). See [non decimal numbers](#) for background information on hex, octal and binary numbers.

Parameters: This function has two parameters: `radix` and `text`.

radix is the base for the numbering system you are converting from. Legal radix values are 2, 4, 8, 16 or 32. Or you can specify the radix as "binary" (same as 2), "octal" (same as 8) or "hex" (short for hexadecimal, same as 16).

text is a text item that contains the non-decimal number you want to convert.

Result: This function normally returns an integer that contains the decimal (base 10) number corresponding to the hex, octal, or binary number input to the function. If the radix is hex and there are more than 8 digits in the input text, or if the radix is binary and there are more than 32 digits, this function will return a raw binary value instead of a number. This binary value may be of unlimited length. Like all binary values, it cannot be calculated with, but should be handled as a text item.

Examples: The example below allows the user to enter two hex values. It then converts the two values to decimal, adds them, and then converts them back to hexadecimal and displays them.

```
local someText,hexNum1,hexNum2,Sum
someText=""
gettext "Enter A+B",someText
hexNum1=radix(16,array(someText,0,"+"))
hexNum2=radix(16,array(someText,1,"+"))
Sum=hexNum1+hexNum2
message someText+"="+radixstr(16,Sum)
```

If the user enters 1E+4 the procedure will display 1E+4=22.

If the result would be bigger than a longword (4 bytes), the `radix()` function will produce a text value that contains the raw binary information. The following three statements are all exactly identical.

```
rawStuff=byte(65)+byte(97)+byte(167)+
byte(45)+byte(109)+byte(159)
rawStuff=radix("hex","4161A72D6D9F")
rawStuff="Aaß-mü"
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the text parameter.

Illegal number. This error occurs if the input value contains a character that is not a legal digit for the radix. For example, the character X is not a legal digit in any radix. The character 9 is not a legal digit in binary or octal.

See Also: **Illegal radix.** This error occurs if you attempt to use a radix other than 2, 8, 16, 32, "binary", "octal", or "hex".

[radixstr\(\)](#) function

[byte\(\)](#) function

[word\(\)](#) function

[longword\(\)](#) function

[binary data](#)

[c/pascal structures](#)

RADIXSTR(...)

Syntax: RADIXSTR(radix,number)

Description: The `radixstr()` function converts a number into a text item containing the equivalent hex, octal, or binary number. See [non decimal numbers](#) for background information on hex, octal and binary numbers.

Parameters: This function has two parameters: `radix` and `number`.

radix is the base for the numbering system you are converting from. Legal radix values are `2`, `4`, `8`, `16` or `32`. Or you can specify the radix as `"binary"` (same as `2`), `"octal"` (same as `8`) or `"hex"` (short for hexadecimal, same as `16`).

number is the number you want to convert to hex, octal, or binary. If the radix is `2`, `16`, `"binary"`, or `"hex"` the number can be a raw binary data (text) value.

Result: This function returns a text item that contains the hex, octal, or binary number equivalent to the number (or binary data) passed to the function.

Examples: The first example converts the decimal value 256 to hexadecimal.

```
radixstr(16,256)
```

This function will calculate that 256_{10} is 100 hex.

The second example converts a decimal value to binary.

```
radixstr("binary",5)
```

This will calculate that 5_{10} is

```
000000000000000000000000000000000000000000000000000000000101 binary.
```

This function will convert the binary value `"abcdef"` into hex.

```
radixstr("hex","abcdef")
```

The result will be `616263646566` hex.

The example below allows the user to enter two hex values. It then converts the two values to decimal, adds them, and then converts them back to hexadecimal and displays them.

```
local someText,hexNum1,hexNum2,Sum
someText=""
gettext "Enter A+B",someText
hexNum1=radix(16,array(someText,0,"+"))
hexNum2=radix(16,array(someText,1,"+"))
Sum=hexNum1+hexNum2
message someText+"="+radixstr(16,Sum)
```

If the user enters `1E+4` the procedure will display `1E+4=22`.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text for the number parameter (unless the radix is hex or binary, in which case text is Ok).

Illegal radix. This error occurs if you attempt to use a radix other than 2, 8, 16, 32, "binary", "octal", or "hex".

See Also:

[radix\(function](#)

[byte\(function](#)

[word\(function](#)

[longword\(function](#)

[binary data](#)

[c/pascal structures](#)

RBOTTOM(...)

Syntax: RBOTTOM(rectangle)

Description: The `rbottom()` function extracts the position of the bottom edge of a rectangle (see [rectangle\(\)](#), [graphic coordinates](#)).

Parameters: This function has one parameter: `rectangle`.
`rectangle` is the rectangle you want to get information about.

Result: This function returns a number between -32,768 and 32768. This is the position of the bottom edge of the rectangle (in pixels).

Examples: The procedure below zooms the window to full size if the bottom of the window is more than 100 pixels from the bottom edge of the screen.

```
if rbottom(info("windowrectangle")+100
    < rbottom(info("screenrectangle")))
    zoomwindow rtop( info("screenrectangle")),
        rleft( info("screenrectangle")),
        rheight( info("screenrectangle")),
        rwidth( info("screenrectangle"), "" )
endif
```

Errors: **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.

See Also: [point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rleft\(\)](#) function
[rtop\(\)](#) function
[rright\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RECTANGLE(...)

Syntax: `RECTANGLE(top,left,bottom,right)`

Description: The `rectangle()` function defines a rectangle from four dimensions. A rectangle is 8 bytes or raw binary data (see [binary data](#), [graphic coordinates](#)). Panorama has many functions and statements that use rectangles for working with graphic elements.

Parameters: This function has four parameters: `top`, `left`, `bottom` and `right`. These parameters may be in screen, window, or form relative co-ordinates as long as you make sure all four use the same co-ordinate system. All measurements are in pixels (1 pixel = 1/72 inch).

top is the position of the top edge of the rectangle. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

left is the position of the left edge of the rectangle. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

bottom is the position of the bottom edge of the rectangle. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

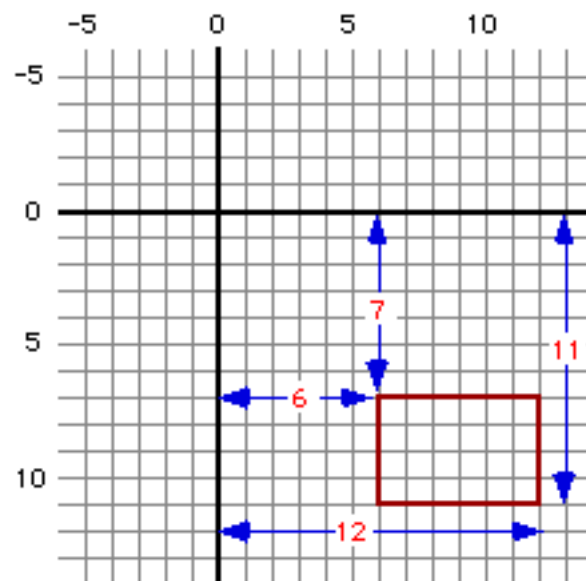
right is the position of the right edge of the rectangle. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

Result: This function returns a rectangle. A rectangle is an 8 byte binary data item. Like all other binary data items, rectangles are actually stored as text (see [binary data](#))

Examples: The procedure below creates a rectangle that is 4 pixels high and 6 pixels wide.

```
MyRectangle=rectangle(7,6,11,12)
```

Here is a magnified view of what this rectangle would look like if it was displayed on the screen:



Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for any of the four parameters.

See Also:

[point\(\)](#) function
[rectanglesize\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function
[rright\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[rectangleadjust\(\)](#) function
[rectanglecenter\(\)](#) function
[adjustxy\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RECTANGLEADJUST(...)

Syntax: `RECTANGLEADJUST(rect,d_top,d_left,d_bottom,d_right)`

Description: The `rectangleadjust()` function adjusts all four edges of a rectangle independently.

Parameters: This function has five parameters: `rect`, `d_top`, `d_left`, `d_bottom`, and `d_right`. These parameters may be in screen, window, or form relative co-ordinates as long as you make sure all four use the same co-ordinate system. All measurements are in pixels (1 pixel = 1/72 inch).

rect is the original rectangle.

d_top is the distance the top edge of the rectangle should be moved. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

d_left is the distance the left edge of the rectangle should be moved. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

d_bottom is the distance the bottom edge of the rectangle should be moved. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

d_right is the distance the right edge of the rectangle should be adjusted. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

Result: This function returns a rectangle. A rectangle is an 8 byte binary data item. Like all other binary data items, rectangles are actually stored as text (see [binary data](#))

Examples: The procedure below creates a rectangle that is inset 20 pixels from all four edges of the screen.

```
MyRectangle=rectangleadjust(
  info("screenrectangle"),20,20,-20,-20)
```

The procedure below creates a rectangle that is the same size as the button rectangle but shifted 1 inch (72 pixels) to the right.

```
MyRectangle=rectangleadjust(
  info("buttonrectangle"),0,72,0,72)
```

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for any of the four `d_` (delta) parameters.

Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value for the rectangle parameter.

See Also: [rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rectanglecenter\(\)](#) function
[adjustxy\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function

rright(function
rheight(function
rwidth(function
inrectangle(function
unionrectangle(function
intersectionrectangle(function
rectanglecenter(function
info("screenrectangle") function
info("windowrectangle") function
info("buttonrectangle") function
info("cursorrectangle") function

RECTANGLECENTER(...)

- Syntax:** `RECTANGLECENTER(largerect,smallrect)`
- Description:** The `rectanglecenter()` function adjusts a small rectangle so that it is centered inside of a larger rectangle.
- Parameters:** This function has two parameters: `largerect` and `smallrect`.
- largerect** is a large rectangle. How large is large? Well, it should at least be larger than the `smallrect` rectangle.
- smallrect** is a small rectangle. How small is small? Well, it should at least be small enough to fit inside the `largerect` rectangle, although the function will do its best to center it even if it does not fit.
- Result:** This function returns a rectangle. A rectangle is an 8 byte binary data item. Like all other binary data items, rectangles are actually stored as text (see [binary data](#))
- Examples:** The procedure below creates a 1 inch square rectangle that is centered within the current screen dimensions.
- ```
MyRectangle=rectanglecenter(
 info("screenrectangle"),rectanglesize(0,0,72,72))
```
- Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for either of the rectangle parameters.
- See Also:** [rectangle\(\)](#) function  
[rectanglesize\(\)](#) function  
[adjustxy\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[rright\(\)](#) function  
[rheight\(\)](#) function  
[rwidth\(\)](#) function  
[inrectangle\(\)](#) function  
[unionrectangle\(\)](#) function  
[intersectionrectangle\(\)](#) function  
[rectanglecenter\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function

# RECTANGLESIZE(...)

**Syntax:** `RECTANGLESIZE(top,left,height,width)`

**Description:** The `rectanglesize()` function defines a rectangle from four dimensions. A rectangle is 8 bytes or raw binary data (see [binary data](#), [graphic coordinates](#)). Panorama has many functions and statements that use rectangles for working with graphic elements.

**Parameters:** This function has four parameters: `top`, `left`, `height` and `width`. The `top` and `left` parameters may be in screen, window, or form relative co-ordinates as long as you make sure both use the same co-ordinate system. All measurements are in pixels (1 pixel = 1/72 inch)

**top** is the position of the top edge of the rectangle. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

**left** is the position of the left edge of the rectangle. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

**height** is the height of the rectangle. This must be a number between 0 and +32,767.

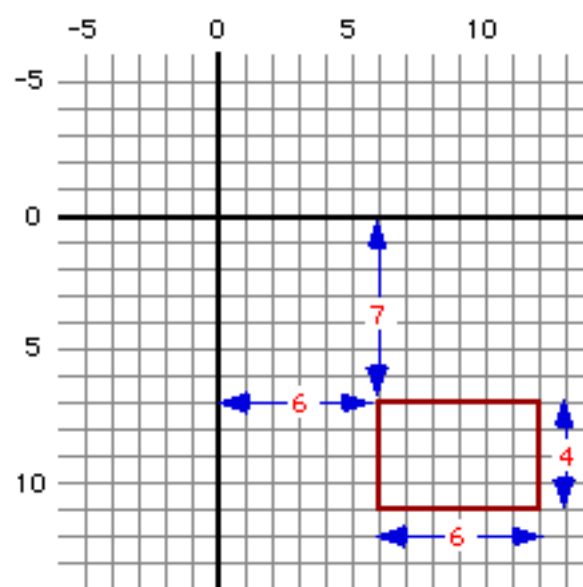
**width** is the width of the rectangle. This must be a number between 0 and +32,767.

**Result:** This function returns a rectangle. A rectangle is an 8 byte binary data item. Like all other binary data items, rectangles are actually stored as text (see [binary data](#))

**Examples:** The procedure below creates a rectangle that is 4 pixels high and 6 pixels wide.

```
MyRectangle=rectanglesize(7,6,4,6)
```

Here is a magnified view of what this rectangle would look like if it was displayed on the screen:



**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for any of the four parameters.

**See Also:**

[point\(\)](#) function  
[rectangle\(\)](#) function  
[rectangleadjust\(\)](#) function  
[rectanglecenter\(\)](#) function  
[adjustxy\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[rright\(\)](#) function  
[rheight\(\)](#) function  
[rwidth\(\)](#) function  
[inrectangle\(\)](#) function  
[unionrectangle\(\)](#) function  
[intersectionrectangle\(\)](#) function  
[rectanglecenter\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function

# RED(...)

**Syntax:** RED(color)

**Description:** The `red()` function extracts the red intensity from a color.

**Parameters:** This function has one parameter: `color`.

**color** is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

**Result:** This function extracts the intensity of the red component of this color. This intensity is a number between 0 (black) and 65535 (full intensity).

**Examples:** The example below calculates the red intensity of the color (in percent, from 0 to 100%).

```
Intensity=red(HighlightColor)*100/65535
```

**Errors:** For more examples of color, see [colors](#).

**Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the color parameter.

**See Also:** [rgb\(\)](#) function  
[hsb\(\)](#) function  
[green\(\)](#) function  
[blue\(\)](#) function  
[hue\(\)](#) function  
[saturation\(\)](#) function  
[brightness\(\)](#) function  
[objectinfo\(\)](#) function  
[changeobjects](#) function  
[colorwheel](#) statement  
[colors](#)

# REGISTRYDELETE

**Syntax:** REGISTRYDELETE path

**Description:** The `registrydelete` statement deletes a registry key or registry value (Windows only).

**Parameters:** This statement has one parameter: path.

**path** is the path to an item in the registry. The path consists of one or more items separated by backslashes (\). The first item must be one of these six classes in the table below (or the abbreviations shown on the right).

Root	Abbreviation
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKUS
HKEY_CURRENT_CONFIG	HKCC
HKEY_DYN_DATA	HKDD

**Action:** This statement will delete a registry key or registry value. The item is deleted immediately, there is no undo.

**Examples:** This example deletes a registry value:

```
registrydelete "HKLM\Software\Acme\SuperWidget:WindowLocation"
```

This example deletes a registry key, along with any values associated with it:

```
registrydelete "HKLM\Software\Acme\SuperWidget"
```

**Views:** This statement may be used in any view.

**See Also:** [registryinfo\(\)](#) function  
[registrywrite](#) statement



# REGISTRYINFO(...)

**Syntax:** REGISTRYINFO(path)

**Description:** The `registryinfo()` function returns information from the Windows registry. The function may return a directory of subkeys, a directory of values within a registry key, or a specific value within a key. The function does not work on MacOS computers.

**Parameters:** This function has one parameter: `path`.

**path** is the path to an item in the registry. The path consists of one or more items separated by backslashes (\). The first item must be one of these six classes in the table below (or the abbreviations shown on the right).

Root	Abbreviation
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKUS
HKEY_CURRENT_CONFIG	HKCC
HKEY_DYN_DATA	HKDD

**Result:** The result of this function depends on the `path` parameter. If the path specifies a registry folder the function will return a list of subkeys in this folder. If the path specifies a specific key it the function will return a list of the subkeys in this key. If the path specifies a specific subkey the function will return the value of the subkey.

**Examples:** The example below returns a list of control panels.

```
registryinfo("HKEY_CURRENT_USER\Control Panel")
```

The example below returns a list of of the subkeys within the `Mouse` key.

```
registryinfo("HKEY_CURRENT_USER\Control Panel\Mouse:")
```

The example below returns the value in the `MouseSpeed` subkey within the `Mouse` key.

```
registryinfo("HKCU\Control Panel\Mouse:MouseSpeed")
```

The example below returns the default value of the `.aif` key. If you have QuickTime installed the result of this function will be `QuickTime`.

```
registryinfo("HKEY_CLASSES_ROOT**.aif:<DEFAULT>")
```

**Errors:** **Registry Error.** This error occurs if the root of the registry path is not one of the six roots listed above, or if you attempt to use this function on a non MacOS computer.

**See Also:** [registrywrite](#) statement  
[registrydelete](#) statement

# REGISTRYWRITE

**Syntax:** REGISTRYWRITE path,type,data

**Description:** The `registrywrite` statement allows you to create registry keys and registry values, or to modify an existing registry value (Windows only).

**Parameters:** This statement has three parameter: `path`, `type` and `data`.

**path** is the path to an item in the registry. The path consists of one or more items separated by backslashes (\). The first item must be one of these six classes in the table below (or the abbreviations shown on the right).

Root	Abbreviation
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKUS
HKEY_CURRENT_CONFIG	HKCC
HKEY_DYN_DATA	HKDD

**type** is the type of data being written. The possible choices are shown below. If you specify "", Panorama will default to `REG_SZ` (text).

0	REG_NONE
1	REG_SZ
2	REG_EXPAND_SZ
3	REG_BINARY
4	REG_DWORD
5	REG_DWORD_BIG_ENDIAN
6	REG_LINK
7	REG_MULTI_SZ
8	REG_RESOURCE_LIST
9	REG_RESOURCE_LIST
10	REG_RESOURCE_REQUIREMENTS_LIST

**data** is the actual data being written. No matter what data format you are writing, this should be text. For other data types you can fill the text item with binary values (see [binary data](#)).

**Action:** This statement modifies the value of a registry key. If the key does not exist it will be created.

**Examples:** This example changes the mouse speed. Notice that this statement supports the same abbreviations allowed by the `registryinfo()` function (see “[Getting Information About Registry Items](#)” on page 1696).

```
registrywrite "HKCU\Control Panel\Mouse:MouseSpeed", "", "2"
```

You can also change the default value associated with a registry key.

```
registrywrite "HKCR**.aif:<DEFAULT>", "", "Quick Time Movie"
```

This example creates a registry entry named `Acme`, but does not create or modify any values associated with that key.

```
registrywrite "HKLM\Software\Acme", "", ""
```

**Views:** This statement may be used in any view.

**See Also:** [registryinfo\(\)](#) function  
[registrydelete](#) statement

# REGULARDATE(...)

**Syntax:** `REGULARDATE(superdate)`

**Description:** The `regulardate()` function extracts a regular date (number of days from January 1, 4713 B.C.) from a superdate. SuperDates combine the date and time into a single number...the number of seconds since January 1, 1904. SuperDates make it easy to calculate time intervals across multiple days. However, SuperDates take up more storage than regular dates, and are not as easy to work with. In addition, SuperDates are limited to dates between 1904 and 2040.

**Parameters:** This function has one parameter: `superdate`.

**superdate** is a number that combines the date and time into a single value. See the [superdate\(\)](#) function.

**Result:** This function extracts the date from a superdate. You can convert this date to text with the [datepattern\(\)](#) function.

**Examples:** In its heyday, the Santa Fe Super Chief train would travel between Chicago and Los Angeles in 39 and 1/2 hours. This example uses SuperDates to calculate the arrival time and day after the user enters the departure time.

```
local Arrival,Departure,xTime
xTime="7:30 pm"
gettext "Departure Time",xTime
Departure=superdate(today(),time(xTime))
Arrival=Departure+superdate(0,time("39:30:00"))
message "Arrives "+
 datepattern(regulardate(Arrival),"DayOfWeek")+ " at "+
 timepattern(regulartime(Arrival),"hh:mm am/pm")
```

If the train leaves at 7:30 pm on Monday, the message will be Arrives Wednesday at 10:00 am. As you can see, SuperDate arithmetic is very easy, just add or subtract. There's no need to worry about crossing midnight, because that simply is the start of a new day.

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the superdate parameters.

**See Also:** [superdate\(\)](#) function  
[regulartime\(\)](#) function  
[date\(\)](#) function  
[time\(\)](#) function  
[today\(\)](#) function  
[now\(\)](#) function

# REGULARTIME(...)

**Syntax:** `REGULARTIME(superdate)`

**Description:** The `regulartime()` function extracts a regular Panorama time (seconds since midnight) from a `superdate`. SuperDates combine the date and time into a single number...the number of seconds since January 1, 1904. SuperDates make it easy to calculate time intervals across multiple days. However, SuperDates take up more storage than regular dates, and are not as easy to work with. In addition, SuperDates are limited to dates between 1904 and 2040.

**Parameters:** This function has one parameter: `superdate`.

`superdate` is a number that combines the date and time into a single value. See the [superdate\(\)](#) function.

**Result:** This function extracts the time from a superdate. You can convert this time to text with the [timepattern\(\)](#) function.

**Examples:** In its heyday, the Santa Fe Super Chief train would travel between Chicago and Los Angeles in 39 and 1/2 hours. This example uses SuperDates to calculate the arrival time and day after the user enters the departure time.

```

local Arrival,Departure,xTime
xTime="7:30 pm"
gettext "Departure Time",xTime
Departure=superdate(today(),time(xTime))
Arrival=Departure+superdate(0,time("39:30:00"))
message "Arrives "+
 datepattern(regulardate(Arrival),"DayOfWeek")+ " at "+
 timepattern(regulartime(Arrival),"hh:mm am/pm")

```

If the train leaves at 7:30 pm on Monday, the message will be Arrives Wednesday at 10:00 am. As you can see, SuperDate arithmetic is very easy, just add or subtract. There's no need to worry about crossing midnight, because that simply is the start of a new day.

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the superdate parameters.

**See Also:** [superdate\(\)](#) function  
[regulardate\(\)](#) function  
[timepattern\(\)](#) function  
[date\(\)](#) function  
[time\(\)](#) function  
[today\(\)](#) function  
[now\(\)](#) function

# REMINDER

- Syntax:** REMINDER reminder,message
- Description:** The **reminder** statement allows the user to edit a reminder with a dialog (see [reminder data](#)).
- Parameters:** This statement has two parameters: **reminder** and **message**.
- reminder** is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. See [reminder data](#) for detailed information about reminders. In this case you are not specifying the reminder itself, but the field containing the reminder data.
- message** is the message that goes with the alarm, for example Sue's flight arrives or Pick up kids. In this case you are not specifying the message itself, but the field containing the messages.
- Action:** This statement displays the dialog for editing a reminder. This dialog allows the user to change the date, time, message and attributes of the reminder. When the user presses the Ok button the statement will update the database and it will also tell the Team Alarm extension (if installed) to modify its copy of the reminder.
- Examples:** This example creates a new reminder for tomorrow at 9am. It then allows the user to modify the reminder with a dialog.
- ```
buildreminder today()+1,time("9am"),0,Reminder
Message=grabdata("Contacts","Name")
reminder Reminders,Message
```
- Views:** This statement may be used in a Data Sheet or Form view.
- See Also:** [reminder\(function](#)
[reminderdate\(function](#)
[remindertime\(function](#)
[remindertype\(function](#)
[buildreminder statement](#)
[alarmedit statement](#)
[reminder data](#)

REMINDER DATA

Background: A reminder is a special data type that holds scheduling information. Reminders are usually used in calendar database applications. A reminder is a raw binary data item 30 bytes long (stored in a text field or variable), and contains the following information:

- The reminder type (either appointment or to-do)
- The reminder date, or recurring date information (july 12, every tuesday, etc.)
- The reminder time (3:30pm, 7:20am, etc.)
- Alarm status
- Completion status (to-do only)
- Priority (to-do only)

Notice that the reminder only contains scheduling information. It does not contain any message or other information about the event. If there is a message associated with a reminder (for example, Lunch with Bob) it should be stored in a separate field or variable.

Although reminders can be kept in a variable, they are usually kept in a database field.

Appointments There are two different types of reminders: Appointments and To-do's. Appointment reminders are used for anything that has a definite, fixed, time: appointments, birthdays, meetings, etc. Once the time has passed the appointment is no longer relevant. For example, it won't do much good to be reminded that your spouse's birthday was yesterday!

To-do reminders have a completion status as well as a time and date. For example, suppose you set up a to-do reminder to order parts on Monday. If you don't get around to it, you'll still want it your to-do list on Tuesday, and again on Wednesday etc. until you actually do order the parts. To-do reminders remain active until the task is completed (or at least it is marked as completed!)

Alarms: If you have the optional Team Alarm extension installed, you can be notified of your reminders even when Panorama is not currently running. To do this, the Team Alarm extension keeps a separate private list of pending alarms. This list is in a special format that cannot be accessed by Panorama. However, this extra alarms database is updated automatically when a reminder is updated with the [reminder](#) statement. However, if you modify a reminder yourself without using the [reminder](#) statement, you'll need to make sure the Team Alarm list is also updated. There are three statements for doing this: [alarmedit](#), [alarmdelete](#), and [alarmreset](#).

See Also: [reminder\(\)](#) function
[remindercaption\(\)](#) function
[reminderdate\(\)](#) function
[remindertime\(\)](#) function
[remindertype\(\)](#) function
[remindertodo\(\)](#) function
[remindercompare\(\)](#) function
[reminderpriority\(\)](#) function
[lookupcalendar\(\)](#) function
[lookupptime\(\)](#) function
[buildreminder](#) statement
[reminder](#) statement

todopriority statement
todo statement
alarmedit statement
alarmdelete statement
alarmreset statement

REMINDER(...)

Syntax: REMINDER(date,time,type)

Description: The `reminder()` function builds a new reminder (see [reminder data](#)).

Parameters: This function has three parameters: date, time and type.

date is the date for the new reminder (the `reminder()` function cannot create recurring reminders). However, you should not use a number here as you do with most date functions. You should use text that describes the date, for example "5/25/03" or "next tuesday".

time is the time for the new reminder. However, you should not use a number here as you do with most time functions. You should use text that describes the time, for example "5:22 pm".

type is the type of the new reminder: "a" for appointments or "t" for to-dos (see [reminder data](#)).

Result: This function returns a reminder, which can be stored in a text field.

Examples: This example adds a new reminder next tuesday at 2pm. The procedure stores this reminder in a field called `Schedule`:

```
addrecord  
Schedule=reminder("next tue","2:00 pm","a")
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for any parameter.

See Also: [reminderdate\(\)](#) function
[remindertime\(\)](#) function
[remindertype\(\)](#) function
[buildreminder](#) statement
[reminder](#) statement
[reminder data](#)

REMINDERCAPTION(...)

Syntax: REMINDERCAPTION(reminder)

Description: The `remindercaption()` function extracts the date from a reminder as formatted text that describes when the reminder will occur (see [reminder data](#)).

Parameters: This function has one parameter: `reminder`.

reminder is the reminder you want to extract information from.

Result: This function returns text describing when the reminder will occur. The table below shows typical examples of how different reminder frequencies will be formatted by this function:

| Frequency | Typical Examples |
|-----------|--|
| Once only | Tuesday, May 16th, 1995 |
| Annual | August 8th of each year |
| Quarterly | First day of each quarter |
| Monthly | 5th day of each month
Last day of each month
2nd Wed of each month |
| Weekly | Tue of each week |
| Daily | Every day |

Examples: The formula below displays the reminder date in a field called Schedule. This formula could be used in an auto-wrap text object or a Text Display SuperObject™.

```
remindercaption(Schedule)
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

See Also: [reminderdate\(\)](#) function
[remindertime\(\)](#) function
[remindertype\(\)](#) function
[buildreminder](#) statement
[reminder](#) statement
[reminder data](#)

REMINDERCOMPARE(...)

Syntax: REMINDERCOMPARE(reminder,date)

Description: The `remindercompare()` function checks to see if a reminder occurs on a specified date (see [reminder data](#)).

Parameters: This function has two parameter: `reminder` and `date`.
reminder is the reminder you want to compare.
date is the date you want to compare with the reminder

Result: This function returns either true or false. It will return true if the reminder will occurs on the specified date, including a repeating reminder that falls on that date.

Examples: The example below selects all reminders that will occur on next tuesday.

```
select remindercompare(Schedule,date("next tuesday"))
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a text value for the date parameter.

See Also: [reminder\(\)](#) function
[reminderdate\(\)](#) function
[remindertime\(\)](#) function
[?\(\)](#) function
[buildreminder](#) statement
[reminder](#) statement
[if](#) statement
[reminder data](#)

REMINDERDATE(...)

Syntax: REMINDERDATE(reminder)

Description: The `reminderdate()` function extracts the date from a reminder (see [reminder data](#)).

Parameters: This function has one parameter: reminder.

reminder is the reminder you want to extract information from.

Result: This function returns a number corresponding to the date for this reminder. This number can be used with functions like `datepattern()`. If a reminder repeats, the function will try to come up with the most appropriate single date. The table below shows how different reminder frequencies will be handled by this function:

| Type of Reminder | Result |
|------------------|----------------------------------|
| Once only | Actual Date |
| Annual | Next occurrence of this reminder |
| Quarterly | 0 |
| Monthly | 0 |
| Weekly | 0 |
| Daily | 0 |

Examples: The example below uses the `reminderdate()` function to help sort the database by date. The procedure copies the dates to a temporary field called `ReminderDate`. (The actual reminder data is in a field called `Schedule`.)

```
field ReminderDate
formulafill reminderdate(Schedule)
sortup
```

This procedure will sort the quarterly, monthly, weekly and daily reminders to the top of the database. The other reminders will be sorted by date.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

See Also: [reminder\(\)](#) function
[remindertime\(\)](#) function
[remindertype\(\)](#) function
[buildreminder](#) statement
[reminder](#) statement
[reminder data](#)

REMINDERPRIORITY(...)

- Syntax:** REMINDERPRIORITY(reminder)
- Description:** The `reminderpriority()` function extracts the priority of a to-do reminder (completed/not completed) from a reminder (see [reminder data](#)).
- Parameters:** This function has one parameter: `reminder`.
reminder is the reminder you want to extract information from.
- Result:** This function returns a number from 0 to 3: 0 for the lowest priority to 3 for the highest priority.
- Examples:** The example below uses the `reminderpriority()` function to select high priority to-do items.
- ```
select reminderpriority(Schedule)=3 and
remindertype(Schedule)=1
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

**See Also:** [reminder\(\)](#) function  
[reminderdate\(\)](#) function  
[remindertime\(\)](#) function  
[remindertodo\(\)](#) function  
[buildreminder](#) statement  
[reminder](#) statement  
[todo](#) statement  
[todopriority](#) statement  
[reminder data](#)

# REMINDERTIME(...)

**Syntax:** REMINDERTIME(reminder)

**Description:** The `remindertime()` function extracts the time from a reminder (see [reminder data](#)).

**Parameters:** This function has one parameter: `reminder`.  
**reminder** is the reminder you want to extract information from.

**Result:** This function returns a number corresponding to the time for this reminder (the number of seconds since midnight). This number can be used with functions like [timepattern\(\)](#).

**Examples:** The example below uses the `remindertime()` function to select only those reminders that occur during work hours.

```
select remindertime(Schedule)>time("8am") and
remindertime(Schedule) <= time("5pm")
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

**See Also:** [reminder\(\)](#) function  
[reminderdate\(\)](#) function  
[remindertype\(\)](#) function  
[buildreminder](#) statement  
[reminder](#) statement  
[reminder data](#)

# REMINDERTODO(...)

**Syntax:** REMINDERTODO(reminder)

**Description:** The `remindertodo()` function extracts the status of a to-do reminder (completed/not completed) from a reminder (see [reminder data](#)).

**Parameters:** This function has one parameter: `reminder`.  
**reminder** is the reminder you want to extract information from.

**Result:** This function returns a number: 0 if the to-do has not been completed (or the reminder is an appointment type), or 1 if the to-do has been completed

**Examples:** The example below uses the `remindertodo()` function to select all uncompleted to-do items.

```
select remindertodo(Schedule)=0 and
remindertype(Schedule)=1
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

**See Also:** [reminder](#) function  
[reminderdate\(\)](#) function  
[remindertime\(\)](#) function  
[buildreminder](#) statement  
[reminder](#) statement  
[todo](#) statement  
[todopriority](#) statement  
[reminder data](#)

# REMINDERTYPE(...)

**Syntax:** REMINDERTYPE(reminder)

**Description:** The `remindertype()` function extracts the type (to-do or appointment) from a reminder (see [reminder data](#)).

**Parameters:** This function has one parameter: `reminder`.  
**reminder** is the reminder you want to extract information from.

**Result:** This function returns a number: 0 if the reminder is an appointment type, or 1 if it is a to-do.

**Examples:** The example below uses the `remindertype()` function to select today's to-do list.

```
select remindertype(Schedule)=1 and
remindercompare(Schedule,today())
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the reminder parameter.

**See Also:** [reminder](#) function  
[reminderdate\(\)](#) function  
[remindertime\(\)](#) function  
[buildreminder](#) statement  
[reminder](#) statement  
[todo](#) statement  
[todopriority](#) statement  
[reminder data](#)



# REMOVEDDETAIL

**Syntax:** REMOVEDDETAIL level

**Description:** The **removedetail** statement removes data records from the current database, leaving only summary records. It can also delete low level summary records, leaving only higher levels.

**Parameters:** This statement has one parameter: level.

**level** is a number that specifies the minimum summary level that will not be removed. In other words, if you want to remove only data records this value should be zero. If you want to remove data records and the first level of summary this value should be 1, etc. This value may be between 0 (data records) and 7 (highest level summary). To make sure that all summary records are removed, use 7.

If the level parameter is the word dialog (no quotes), the procedure will stop and display the standard **Remove Detail** dialog. The user may select a level using the buttons. The procedure will then remove the data and summary records and continue.

**Action:** This statement performs the same action as the **Remove Detail** command in the **Sort** menu.

**Examples:** This example calculates summaries for each company, then saves those summaries in a separate database. The Company Totals database will not have any of the original data, only the summary information (which will have been converted into data records for possible further processing.)

```

field Company
group
field Amount
total
removedetail 0
saveas "Company Totals"

```

**Views:** This statement may be used in any view.

**See Also:** [group](#) statement  
[removesummaries](#) statement  
[outlinelevel](#) statement  
[summarylevel](#) statement  
[info\("summary"\)](#) function

# REMOVESUMMARIES

**Syntax:** REMOVESUMMARIES level

**Description:** The **removesummaries** statement removes summary records from the current database.

**Parameters:** This statement has one parameter: level.

**level** is a number that specifies the minimum summary level that will not be removed. In other words, if you want to remove only the first level of summary this value should be 1. This value may be between 1 (lowest summary) and 7 (highest level summary). To make sure that all summary records are removed, use 7.

If the level parameter is the word **dialog** (no quotes), the procedure will stop and display the standard **Remove Summaries** dialog. The user may select a level using the buttons. The procedure will then remove the summaries and continue.

**Action:** This statement performs the same action as the **Remove Summaries** command in the **Sort** menu.

**Examples:** This example calculates summaries, prints a report, then restores the database to its original condition.

```
field Date
group by year
group by month
openform "Annual Report"
print dialog
closewindow
removesummaries 7
```

**Views:** This statement may be used in any view.

**See Also:** [group](#) statement  
[removedetail](#) statement  
[outlinelevel](#) statement  
[summarylevel](#) statement  
[info\("summary"\)](#) function

# REMOVEUNSELECTED

**Syntax:** REMOVEUNSELECTED

**Description:** The `removeunselected` statement deletes all unselected records from the database. (Unselected records are invisible, but they are still part of the database and take up space.) Once they are removed, the records cannot be recovered unless you have previously saved them on the disk. The `removeunselected` statement will not ask the user to confirm before deleting the records; it assumes that the procedure writer knows what he or she is doing.

**Parameters:** This statement has no parameters.

**Examples:** This example deletes all records from previous years.

```
select date(year1st(today())
removeunselected
```

**Views:** This statement may be used in any view

**See Also:** [info\("records"\)](#) function  
[info\("selected"\)](#) function  
[select](#) statement  
[selectadditional](#) statement  
[selectall](#) statement  
[selectreverse](#) statement  
[selectsummaries](#) statement  
[selectwithin](#) statement

# RENAMERESOURCE

- Syntax:** `RENAMERESOURCE type,id,number,name`
- Description:** The `renameresource` statement changes the number and/or the name of a resource item. The resource file must be opened with the `openresourcerw` statement.
- Parameters:** This statement has four parameter: `type`, `id`, `number` and `name`.
- type** is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).
- id** is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).
- number** is the new id number for the resource.
- name** is the new name for the resource. If this parameter is "" the name will be left alone. If this parameter is ¶ the name will be erased.
- Action:** This statement renumbers and/or renames a resource item. It is mostly useful for building your own resource editor program (like the **Custom Menu Editor** Wizard that comes with Panorama).
- Examples:** The procedure below changes resource `MENU 498` into `MENU 1917`, and gives the resource the name `Option Menu`.
- ```
renameresource "MENU",498,1917,"Option Menu"
```
- The procedure below changes resource `MENU 498` into `MENU 8367` while leaving the name alone.
- ```
renameresource "MENU",1917,8367,""
```
- The procedure below erases the name from `MENU 8367`.
- ```
renameresource "MENU",8367,8367,¶
```
- Views:** This statement may be used in any view.
- See Also:** [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[writeresource](#) statement
[deleteresource](#) statement
[activeresource](#) statement

RENAMEWINDOW

Syntax: RENAMEWINDOW

Description: The `renamewindow` statement allows you to rename the active form.

Parameters: This statement has no parameters.

Action: This statement pauses the procedure and allows you to rename the active form; the form's name appears in the drag bar after the database's name and a colon (:).

The statement will first present you with a dialog box which highlights the current form's name allowing you to type in a new name. You may then either press **OK** or **Cancel**. If you press **OK** the form will be renamed, if you press **Cancel** the statement will be dismissed and the procedure will continue on. If you choose a name already in use Panorama will warn you against using duplicate names and allows you to try again.

Note: if the active window is not a form window this statement will generate an alert message dialog telling you that "You can't do that in this window."

Examples: This example will rename the active window provided it is a form window and that you respond by hitting the OK button.

```
renamewindow
```

This example allows you to choose the form you wish to rename, provided it exists.

```
getscrap "Enter form name."
openform clipboard()
if error
    message "Form: "+clipboard()+" not found."
    stop
endif
renamewindow
```

This example allows you to select a form from the list provided by the `formselect` statement and, if the **Rename** button is selected, will open the form window, rename the form, and then closes the form window.

```
local B,F
formselect 2086,0,B,F
if B contains "rename" and F ≠ ""
    windowbox "99 77 244 435"
    openform F
    renamewindow
    closewindow
endif
...
...
...
```

Views: This statement may be used in a Form view **only**.

See Also: [deletewindow](#) statement
[goform](#) statement
[graphicsmode](#) statement
[openform](#) statement
[window](#) statement

REP(...)

Syntax: REP(root,count)

Description: The **rep()** function (short for repeat) assembles a text item by repeating a smaller text item over and over again.

Parameters: This function has two parameters: **root**. and **count**.
root is the main character, word or phrase that you want to repeat over and over again. This must be a text item.
count is the number of times you want the root to be repeated.

Result: The result of this function is always a text item.

Examples: The example below creates a text item with 20 asterisks in a row.

```
rep(" * ",20)
```

The previous example is exactly the same as this formula below:

```
"*****"
```

The **rep()** function however, is less prone to error and the count can be changed easily or even vary dynamically. Here is a formula which adds leading asterisks to a number so that there are always 15 characters:

```
rep(" * ",15-length(pattern(Amount,"$#, .##")))+  
pattern(Amount,"$#, .##")
```

The output from this formula will look like this:

```
***** $4,983.45  
**** $22,456.74  
***** $482.51  
*** $467,380.12
```

The root does not have to be a single character. It can be a complete word or phrase.

```
rep("Love ",5)
```

This formula repeats the word over and over again:

```
Love Love Love Love Love
```

The **rep()** function can be used to display a crude bar graph.

```
rep(" * ",Rating)
```

If **Rating** is a numeric field with a value between 1 and 4 this formula will display a visual representation. For example, you might use a formula like this to display ratings of movies (1 star, 2 stars, etc.)

Errors:

Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value for the root, for example `rep(4567,12)`. If you have a number you must convert the number to text before using it with this function, for example `rep(str(4567),12)`.

Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a text value for the number of repetitions, for example `rep("$", "25")`. If you have a number you must convert the number to text before using it with this function, for example `rep("$",val("25"))`.

REPEATLOOPIF

Syntax: REPEATLOOPIF true-false test

Description: The `repeatloopif` statement decides whether to continue with a loop or to start over again from the top.

Parameters: This statement has one parameter: true-false test.

true-false test is a formula that should result in a true (-1) or false (0) answer. Usually the formula is created with a combination of comparison operators (=, ≠, , etc.) and boolean combinations (and, or, etc.) For example the formula `Name="Smith"` will be true if the field or variables Name contains the value **Smith**, and false if it contains any other value.

Action: This statement decides whether to continue with a loop or to start over again from the top. If the test is true, the loop will start over again. If the test is false, the loop will continue normally

Examples: The example builds a list of the alphabetic letters used in the field `Notes`. The `repeatloopif` statement is used to check whether or not a character is alphabetic. If it is not alphabetic, the loop starts over again from the top.

```

local X,aLetter,LetterList
X=1
LetterList=""
loop
  aLetter=upper( array(Notes,X,1))
  stoploopif aLetter=""
  X=X+1
  repeatloopif aLetter≠striptoalpha(aLetter)
  if LetterList notcontains aLetter
    LetterList=LetterList+aLetter
  endif
while forever

```

Views: This statement may be used in any view.

See Also: [loop](#) statement
[until](#) statement
[while](#) statement
[stoploopif](#) statement
[if](#) statement
[else](#) statement
[endif](#) statement

REPLACE(...)

Syntax: REPLACE(text,old,new)

Description: The `replace()` function partially replaces text with new text. It searches through an item of text looking for a character, word or phrase. If the function finds an exact match (including upper/lower case) the function replaces the old character, word or phrase with a new character, word or phrase.

Parameters: This function has three parameters: `text`, `old`, and `new`.

text is the item of text that you want to search through and possibly replace part of.

old is the character, word or phrase that you want to search for and replace. This must be the exact character, word, or phrase, including upper or lower case. The `replace()` function will not match `Dr` with `DR` or `dr`.

new is the new character, word or phrase that you want to substitute for the old character word or phrase.

Result: The result of this function is always a text item where the new character, word or phrase has been substituted in every spot where the old character, word or phrase was located.

Examples: This simple procedure uses the `replace()` function to change 50 watt light bulbs into 75 watt light bulbs.

```
field Item
formulafill replace(Item,"50 watt","75 watt")
```

The `replace()` function will replace every occurrence of the old character, word or phrase—even if it occurs many times. The procedure below will convert regular quotes into smart quotes.

```
field Description
formulafill replace(
  replace(" "+Description+" ",{ " },{ "}),
  {" },{ " })[2,-2]
```

The `replace()` function can also be used to eliminate a character, word or phrase—simply by replacing it with nothing! Here's a formula that removes all the trademark symbols from the Description field.

```
field Description
formulafill replace(Description,"™","")
```

Errors: Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value with this function, for example `replace(Notes,10,20)`. If you have a number you must convert the number to text before using it with this function, for example `replace(Notes,str(10),str(20))`.

See Also: [replacemultiple\(\)](#) function
[search\(\)](#) function
[change](#) statement

REPLACEMULTIPLE(...)

Syntax: `REPLACEMULTIPLE(text,old,new,separator)`

Description: The `replacemultiple()` function is similar to the `replace()` function. However, instead of simply replacing one word or phrase with another, the `replacemultiple()` function takes an entire list of words or phrases and replaces them with the corresponding words and phrases in a second list.

Parameters: This function has four parameters: `text`, `old`, `new` and `separator`.

text is the item of text that you want to search through and possibly replace part of.

old is an array of characters, words or phrases that you want to search for and replace. Each entry is separated from the next by the separator character.

new is an array of new characters, words or phrases that you want to substitute for the old characters, words or phrases. This array must have the same number of items as the old parameter. The `replacemultiple()` function will scan through the text parameter. When it finds an item in the old parameter, it will replace that item with the corresponding item in the new parameter.

separator is the separator for the old and new arrays. See [text arrays](#) for more information about separators.

Result: The result of this function is always a text item where the new characters, words or phrases have been substituted in every spot where the old characters, words or phrases were located.

Examples: This example uses the `replacemultiple()` function to replace state abbreviations with the fully spelled out state names.

```
local shortStates,longStates
shortStates="AK;AZ;CA;NV;OR;WA"
longStates="Alaska;Arizona;California;Nevada;Oregon;Washington"
Letter=replacemultiple(Letter,shortStates,longStates,";")
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `replace(Notes,10,20)`. If you have a number you must convert the number to text before using it with this function, for example `replace(Notes,str(10),str(20))`.

See Also: [replace\(\)](#) function
[search\(\)](#) function
[change](#) statement

RESOURCEFORK

Syntax: RESOURCEFORK

Description: The `resourcefork` statement allows you to access the resource fork of a Macintosh file as if it was a data fork. This statement tells Panorama that the `fileload()` and `filesize()` functions and `filesave` and `fileappend` statements should access the resource fork instead of the data fork. To tell Panorama to go back to accessing the data fork, use the `datafork` statement. (On Windows systems, this statement is ignored.)

Parameters: This statement has no parameters.

Action: This statement has no direct action of its own. However, this statement modifies any file access functions that are used later in the procedure. Instead of accessing the data fork, Panorama will access the resource fork.

Examples: This simple example will make a copy of the resource fork of the file named MyResources.

```
local rezdata
resourcefork
rezdata=fileload("","MyResources")
filesave "","My Resources Copy","",rezdata
datafork
```

This example copies the resource fork of MyResources into the data fork of MyResources.RSR. This .RSR file can be transferred to a PC for use with the PC version of Panorama.

```
local rezdata
resourcefork
rezdata=fileload("","MyResources")
datafork
filesave "","My Resources.RSR","",rezdata
```

Views: This statement may be used in any view.

See Also: [datafork](#) statement
[filesave](#) statement
[fileappend](#) statement
[fileload\(\)](#) function
[filesize\(\)](#) function
[openresource](#) statement
[openresourcerw](#) statement
[resources\(\)](#) function
[resourcetypes\(\)](#) function
[getresource\(\)](#) function
[getstring\(\)](#) function
[getnstring\(\)](#) function

RESOURCES(...)

- Syntax:** RESOURCES(type)
- Description:** The `resources()` function creates a text array containing a list of resources of a particular type.
- Parameters:** This function has one parameter: `type`.
- `type` is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).
- Result:** This function returns a text array containing a carriage return delimited list of all the resources of the specified type. Each element of this list is itself a tab delimited array. The first item is the resource item number. The second item is the resource name (if any).
- Examples:** This example builds a list of the TEXT resources in the currently open resource files. (The currently open resource files include Panorama itself and the Macintosh system file, as well as any resource files you have opened with the [openresource](#) statement.)
- ```
local rezStrings
rezStrings=resources("TEXT")
```
- This will fill `rezStrings` with an array like this.
- ```
2001 Error Messages
2002 Command List
2003 Conversion Options
```
- Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the type parameter. The type must be a four letter text item.
- See Also:** [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function
[getstring\(\)](#) function
[getnstring\(\)](#) function
[getstringmatch\(\)](#) function
[resourcetypes\(\)](#) function

RESOURCETYPES(...)

Syntax: RESOURCETYPES()

Description: The `resourcetypes()` function creates a text array containing a list of the resource types in the currently open resource files.

Parameters: This function has no parameters.

Result: This function returns a carriage return delimited text array. Each element in the array contains a resource type. Each resource type is a four letter text item, for example "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).

Examples: You can use this function to check if a particular resource type exists, or you can use the function with a pop-up menu or List SuperObject™ to allow the user to select a type of resource for any reason. The formula below will create a text array with resource types.

```
local rezTypes
rezTypes=resourcetypes()
```

The `rezTypes` variable will be filled with a list of resource types, like this:

```
CNTL
CURS
FKEY
INIT
KCAP
KCHR
LDEF
MACA
PACK
PTCH
ROv#
TPLT
SIZE
LBAR
octb
DLGX
dctb
cocm
TEXT
STR#
PICT
PAT#
PAPR
MENU
MDEF
```

As you can see, the resource types are not listed in any particular order.

Errors: This function does not produce any error messages.

See Also: [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[getresource\(\)](#) function

[getstring\(\)](#) function
[getnstring\(\)](#) function
[getstringmatch\(\)](#) function
[resources\(\)](#) function

RESUME

- Syntax:** RESUME state
- Description:** The **resume** statement resumes a procedure that has been temporarily halted with the [pause](#) statement. The procedure will be restarted from the point immediately after the original [pause](#) statement.
- Parameters:** This statement has one parameter: state.
- state** is a global variable. This global variable is used to store information about where and how the procedure was temporarily halted with the [pause](#) statement. (Note: Starting with Panorama 3.1, the state variable is ignored. You still must supply a valid variable name, but it does not need to match the variable used in the [pause](#) statement.)
- Action:** This statement resumes a procedure that has been temporarily halted with the [pause](#) statement. For example you might want to pause a procedure while the user fills in a dialog, then continue when the user presses the Ok button.
- Examples:** The example opens a form called Sound & Video. To the user, this form will appear to be a standard dialog box. Once the dialog box is open the procedure will pause, allowing the user to type values into the box, select checkboxes, etc.

```

global dialogPause
setwindow 100,100,300,400," "
opendialog "Sound & Video"
pause dialogPause
/* pause here for user to fill in dialog */
closewindow
if info("trigger") contains "Ok"
    playsound dialogSound
endif

```

If you want a button in the Sound & Video dialog to close the dialog, that button should be linked to a procedure that contains the following statement:

```

resume dialogPause

```

Notice that the **resume** statement must reference the same global variable as the [pause](#) statement. (Actually, any valid variable name may be used if this is Panorama 3.1 or later, but we still suggest that you use the correct name for clarity and for compatibility with older versions.) The **resume** statement causes the original procedure to continue, starting from right after the pause statement. In this case the procedure closes the dialog and plays a sound if the user pressed Ok.

The advantage of using the [pause](#) and **resume** statements is that a single dialog may be used with many procedures. The buttons in the dialog aren't actually linked to any specific procedure, they simply let whatever procedure opened the dialog in the first place continue.

- Views:** This statement may be used in any view.

See Also:

[pause statement](#)
[opendialog statement](#)
[info\("trigger"\) function](#)

RESYNCHRONIZE

Syntax: RESYNCHRONIZE

Description: The **resynchronize** statement synchronizes the local Panorama database with the SQL server database (client/server databases only). This is the same as choosing **Synchronize** from the File menu.

Parameters: This statement has no parameters.

Examples: This procedure synchronizes with the server to get the latest possible data, then prints a report with monthly totals.

```
resynchronize  
openform "My Report "  
field "Date"  
groupup by month  
field "Amount "  
total  
outlinelevel 1  
print dialog
```

Views: This statement may be used in any view.

See Also: [subsetselect](#) statement
[subsetselectall](#) statement

RETURNKEY

Syntax: RETURNKEY

Description: The `returnkey` statement adds a new record just below the current record.

Parameters: This statement has no parameters.

Action: This statement adds a new record just below the current record. It has the same effect as pressing the **RETURN** key when you are in the data sheet.

Examples: This simple example adds twelve new records just below the current record.

```
loop  
    returnkey  
until 12
```

Views: This statement may be used in the Data Sheet, Design Sheet, and Form views.

See Also: [insertrecord](#) statement
[insertbelow](#) statement
[addrecord](#) statement
[deleterecord](#) statement
[info\("records"\)](#) function

REVERT

Syntax: REVERT

Description: The **revert** statement reloads the current database from the disk. It has almost the same effect as closing and then re-opening the database (the only difference is that the **.Initialize** procedure is not triggered). Any changes that have been made since the last time this database was saved will be lost. This command has the same effect as choosing the **Revert to Saved** command from the File menu.

The **revert** statement will not normally ask the user to confirm before loading the database from the disk. However, if the **revert** statement is the last statement in the procedure, or is followed by a [stop](#) or [nop](#) statement, it will ask the user to confirm that they want to revert to saved before proceeding.

Examples: Usually the only reason to use the revert statement in a procedure is to simulate the **Revert to Saved** command in your own custom File menu. Here are the statements to use in your .CustomMenu procedure. Since the **revert** statement is followed by a [stop](#) statement, it will ask the user to confirm the operation, just like the real **Revert to Saved** command.

```
if info\("changes"\) beginswith "Menu.File.Revert"  
    revert  
    stop  
endif
```

Views: This statement may be used in any view.

See Also: [save](#) statement
[saveall](#) statement
[saveas](#) statement
[saveacopyas](#) statement
[filesave](#) statement
[info\("changes"\)](#) function

RGB(...)

Syntax: RGB(red,green,blue)

Description: The `rgb()` function creates a color by combining red, green, and blue primary colors. See [colors](#).

Parameters: This function has three parameter: red, green and blue.

red is the intensity of the **red** component of this color. This must be a number from 0 (black) to 65535 (full intensity).

green is the intensity of the **green** component of this color. This must be a number from 0 (black) to 65535 (full intensity).

blue is the intensity of the blue component of this color. This must be a number from 0 (black) to 65535 (full intensity).

Result: This function returns 6 bytes of raw binary data (see [binary data](#)).

Examples: The example below changes the color of any object named Border to orange.

```
local Orange
Orange=rgb(65535,23356,2936)
selectobjects objectinfo("name") = "Border"
changeobjects "color",Orange
```

For more examples of colors see [colors](#).

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the any parameter.

See Also: [hsb\(\)](#) function
[red\(\)](#) function
[green\(\)](#) function
[blue\(\)](#) function
[hue\(\)](#) function
[saturation\(\)](#) function
[brightness\(\)](#) function
[objectinfo\(\)](#) function
[changeobjects](#) function
[colorwheel](#) statement
[colors](#)

RHEIGHT(...)

Syntax: RHEIGHT(rectangle)

Description: The `rheight()` function extracts the height of rectangle (see [rectangle\(\)](#), [graphic coordinates\(\)](#)).

Parameters: This function has one parameter: `rectangle`.
rectangle is the rectangle you want to get information about.

Result: This function returns a number between 0 and 32767. This is the height of the rectangle (in pixels).

Examples: The procedure displays the size of the computers screen, for example Your screen is 640 by 480 pixels.

```
message "Your screen is "+
  str( rwidth( info("screenrectangle")))+ " by "+
  str( rheight( info("screenrectangle")))+ " pixels."
```

Errors: **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.

See Also: [point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rleft\(\)](#) function
[rtop\(\)](#) function
[rright\(\)](#) function
[rbottom\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RIGHT

Syntax: RIGHT

Description: The **right** statement moves the cursor to the next field in the active window. To decide what the next field is, Panorama uses the data sheet order of the fields. This is the opposite of the [left](#) statement.

Parameters: This statement has no parameters.

Action: This statement moves the cursor to the next field in the active window. If the current window is the data sheet the cursor will appear to move to the right one column. If the cursor is already on the last visible column this statement will do nothing. You can use this statement in conjunction with the [info\("fieldname"\)](#) or [info\("stopped"\)](#) functions to test to see if you are on the last visible record in the window.

Examples: This example converts all text fields in the current database to all upper case. It starts with the leftmost field and moves to the right column by column. (Note: The «» symbol, as used in the [upper\(\)](#) function, always refers to the current field.)

```

field (array( dbinfo("fields",""),1,¶))
loop
  if info("datatype")=0
    formulafill upper(«»)
  endif
  right
until info("stopped")

```

Here is a similar example that sets up the field names in a newly imported database. Many programs export field names as the first line of a text file. This procedure takes those field names and copies them into Panorama's field names, then deletes the line containing the field names.

```

field "A"
loop
  fieldname «»
  right
until info("stopped")
deleterecord

```

Views: This statement may be used in any view.

See Also: [info\("fieldname"\)](#) function
[info\("stopped"\)](#) function
[dbinfo\(\)](#) function
[field](#) statement
[left](#) statement

RLEFT(...)

Syntax: RLEFT(rectangle)

Description: The `rleft()` function extracts the position of the left edge of a rectangle (see [rectangle\(\)](#), [graphic coordinates](#)).

Parameters: This function has one parameter: `rectangle`.
rectangle is the rectangle you want to get information about.

Result: This function returns a number between -32,768 and 32768. This is the position of the left edge of the rectangle (in pixels).

Examples: The procedure below zooms the window to full size if the left of the window is more than 100 pixels from the left edge of the screen.

```
if rleft(info("windowrectangle"))>100
    zoomwindow rtop( info("screenrectangle"),
        rleft( info("screenrectangle")),
        rheight( info("screenrectangle")),
        rwidth( info("screenrectangle")), " "
    endif
```

Errors: **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.

See Also: [point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rheight\(\)](#) function
[rtop\(\)](#) function
[rright\(\)](#) function
[rbottom\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RND(...)

Syntax: RND()

Description: The `rnd()` function generates a random number between 0 and 1.

Parameters: This function has no parameters

Result: The result of this function is always a floating point numeric value between 0 and 1. This function will generate all sorts of strange values like 0.453321465099 and 0.9844334343219.

Examples: This example generates a random number between 0 and 1.

```
rnd()
```

This possibly more useful example generates a random integer between 1 and 10 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10).

```
int(1+10*rnd())
```

This more general example generates a random integer between `START` and `START+COUNT`. For example, if `START` is 500 and `COUNT` is 501 this formula will generate a random integer between 500 and 1000.

```
int(START+COUNT*rnd())
```

This example generates a random even integer between 2 and 50 (2, 4, 6, 8, ... 46, 48, 50).

```
round(2+48*rnd(),2)
```

Errors: This function does not generate any errors.

See Also: [int\(\)](#) function
[round\(\)](#) function

ROUND(...)

Syntax: ROUND(value,step)

Description: The **round()** function rounds a number to the nearest step

Parameters: This function has two parameters: value and step.

value is the value you want to round. This must be a number, not text.

step is the increment value for the steps you want to round to. For example if the step is 1 the value will be rounded to an integer, if the step is 12 the value will be rounded to the nearest dozen.

Result: The result of this function is always a numeric value. If the input value was an integer the result will be an integer, if the input was floating point the result will be floating point.

Examples: This example rounds the quantity to the nearest dozen.

```
round(Quantity,12)
```

The output of this formula will be 0, 12, 24, 36, etc. The formula above may round the number down, for example 14 will round down to 12. Here is another formula that will always round up to the next higher dozen (i.e. 13 will round up to 24):

```
round(Quantity+6,12)
```

Non-integer step values are Ok. This next formula rounds the temperature to the nearest 1/2 degree.

```
round(Temperature,0.5)
```

Temperature must contain a numeric value.

Since dates are really numbers, the formula below rounds the StartDate to the first day of the week (Sunday).

```
round(StartDate,7)
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text values with this function, for example `round("Bob","Sue")`. If you have a number in a text item you must convert the text to a numeric value before taking the absolute value, for example `round(val("34"),val("12"))`.

See Also: [int\(\)](#) function

RRIGHT(...)

Syntax: `RRIGHT(rectangele)`

Description: The `rright()` function extracts the position of the right edge of a rectangle (see [rectangle\(\)](#), [graphic coordinates](#)).

Parameters: This function has one parameter: `rectangle`.
`rectangle` is the rectangle you want to get information about.

Result: This function returns a number between -32,768 and 32768. This is the position of the right edge of the rectangle (in pixels).

Examples: The procedure below zooms the window to full size if the right of the window is more than 100 pixels from the right edge of the screen.

```
if rright(info("windowrectangle"))+100
  < rright(info("screenrectangle"))
  zoomwindow rtop( info("screenrectangle")),
  rleft( info("screenrectangle")),
  rheight( info("screenrectangle")),
  rwidth( info("screenrectangle")), ""
endif
```

Errors: **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.

See Also: [point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rleft\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RTN

Syntax: RTN

Description: The `rtn` statement may be used to end a subroutine at any point. When this statement is encountered Panorama stops the current procedure. If the current procedure was called as a subroutine by another procedure, the calling procedure continues from the point just after the [call](#), [farcall](#), or [shortcall](#) statement that started the procedure. If the current procedure was not called as a subroutine, it simply stops (see the [stop](#) statement). (Note: The statement `rtn` is an abbreviation for `return`, because this statement “returns” from the current subroutine to the original procedure.)

Parameters: This statement has no parameters.

Examples: This following adjusts the inventory level if there is enough in stock to complete the order. Otherwise the subroutine simply returns to the original procedure that called it.

```
if OnHand < parameter(1)
  rtn
endif
OnHand=OnHand-parameter(1)
```

Views: This statement may be used in any view, and also works when no windows are open at all.

See Also: [call](#) statement
[farcall](#) statement
[shortcall](#) statement
[stoploopif](#) statement
[debug](#) statement
[if](#) statement
[case](#) statement

RTOP(...)

Syntax: `RTOP(rectangle)`

Description: The `rtop()` function extracts the position of the top edge of a rectangle (see [rectangle\(\)](#), [graphic coordinates](#)).

Parameters: This function has one parameter: `rectangle`.
`rectangle` is the rectangle you want to get information about.

Result: This function returns a number between -32,768 and 32768. This is the position of the top edge of the rectangle (in pixels).

Examples: The procedure below zooms the window to full size if the top of the window is more than 100 pixels from the top of the screen.

```
if rtop(info("windowrectangle"))>100
    zoomwindow rtop(info("screenrectangle"),
        rleft( info("screenrectangle")),
        rheight( info("screenrectangle")),
        rwidth( info("screenrectangle")), " "
    endif
```

Errors: **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.

See Also: [point\(\)](#) function
[rectangle\(\)](#) function
[rectanglesize\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function
[rright\(\)](#) function
[rheight\(\)](#) function
[rwidth\(\)](#) function
[inrectangle\(\)](#) function
[unionrectangle\(\)](#) function
[intersectionrectangle\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

RUNNINGDIFFERENCE

Syntax: RUNNINGDIFFERENCE

Description: The **runningdifference** statement calculates the running difference for the current field. Every cell in the current field is filled with the difference between it and the previous cell. Use this statement when you want to compute the interval or spread between consecutive values.

Parameters: This statement has no parameters.

Action: This statement starts at the top of the database and subtracts each cell from the cell above it. As it scans the database it fills the field with the difference, destroying the original data in the process. It has the same effect as using the **Running Difference** command from the Math menu. Note: This statement may only be used if the current field is a Numeric field. The **runningdifference** statement will not work with text, dates, or pictures.

Examples: This example calculates the gas mileage after each fillup. Since the **runningdifference** statement will destroy the field it is used on, the procedure copies the Odometer data into a separate field called Miles. The **runningdifference** statement computes the number of miles driven on a tank of gas (the difference between the two odometer readings).

```

field Miles
formulafill Odometer
runningdifference
field Mileage
formulafill Miles/Gallons

```

Views: This statement may be used in any view.

See Also: [fill](#) statement [sequence](#) statement
[runningtotal](#) statement
[emptyfill](#) statement
[formulafill](#) statement
[propagate](#) statement
[propagateup](#) statement
[unpropagate](#) statement
[unpropagateup](#) statement

RUNNINGTOTAL

Syntax: RUNNINGTOTAL

Description: The **runningtotal** statement calculates the running total for the current field. Every cell in the current field is filled with the accumulated total from the top of the database (or from the previous summary record).

Parameters: This statement has no parameters.

Action: This statement starts at the top of the database and adds up the numbers in the current field. As it scans the database it fills the field with the running total, destroying the original data in the process. It has the same effect as using the **Running Total** command from the Math menu. Note: This statement may only be used if the current field is a Numeric field. The **runningtotal** statement will not work with text, dates, or pictures.

Examples: This example calculates the balance after each transaction in a checkbook database.

```
field Balance
formulafill Credit-Debit
runningtotal
```

Views: This statement may be used in any view.

See Also: [fill](#) statement
[sequence](#) statement
[runningdifference](#) statement
[emptyfill](#) statement
[formulafill](#) statement
[propagate](#) statement
[propagateup](#) statement
[unpropagate](#) statement
[unpropagateup](#) statement

RWIDTH(...)

- Syntax:** `RWIDTH(rectangle)`
- Description:** The `rwidth()` function extracts the width of rectangle (see [rectangle\(\)](#), [graphic coordinates](#)).
- Parameters:** This function has one parameter: `rectangle`.
`rectangle` is the rectangle you want to get information about.
- Result:** This function returns a number between 0 and 32767. This is the width of the rectangle (in pixels).
- Examples:** The procedure displays the size of the computers screen, for example `Your screen is 640 by 480 pixels`.
- ```
message "Your screen is "+
 str(rwidth(info("screenrectangle")))+" by "+
 str(rheight(info("screenrectangle")))+" pixels."
```
- Errors:** **Illegal function argument.** This error occurs if you attempt to use a parameter that is not a rectangle.
- See Also:** [point\(\)](#) function  
[rectangle\(\)](#) function  
[rectanglesize\(\)](#) function  
[rleft\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[rright\(\)](#) function  
[rheight\(\)](#) function  
[inrectangle\(\)](#) function  
[unionrectangle\(\)](#) function  
[intersectionrectangle\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function

# SANDWICH(...)

**Syntax:** SANDWICH(prefix,root,suffix)

**Description:** The **sandwich()** function assembles a text item from three smaller text items. The prefix and suffix are slapped on the ends of the root, just like a sandwich. However, if the root is empty, the prefix and suffix are also left off (the result is an empty text item), just as you wouldn't make a sandwich without any meat.

**Parameters:** This function has three parameters: **prefix**, **root**, and **suffix**.

**prefix** is the item of text that the final text will start with.

**root** is the main character, word or phrase that you want to include in the final text. If the root is empty, the entire assembled text will be empty.

**suffix** is the item of text that the final text will end with.

**Result:** Suppose you have a database with names and titles, and you want to display this information in a report with the titles surrounded by parentheses. The formula below could be used with an auto-wrap text object or text display SuperObject.

```
Name+" ("+Title+")"
```

The only problem with this formula is if some of the people don't have a title. Then you'll get results like these:

```
Steve Johnson (Sales Mgr)
Bill Langly ()
Mary Wilson (VP Engineering)
Stan Franklin ()
```

The **sandwich()** function can fix this problem. The prefix is (, the suffix is ), and the root is the title.

```
Name+sandwich(" (",Title,")")"
```

This new formula eliminates the superfluous ( and ) symbols:

```
Steve Johnson (Sales Mgr)
Bill Langly
Mary Wilson (VP Engineering)
Stan Franklin
```

The **sandwich()** function is useful any time you have optional data items combined together with punctuation in between. This example displays a name with optional middle name. If the middle name is missing there will be only one space between the first and last names, not two.

```
First+" "+sandwich("",Middle,"")+Last
```

**Errors:**      **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `sandwich("$", 34, "")`. If you have a number you must convert the number to text before using it with this function, for example `sandwich("$", str(34), "")`.

**See Also:**      [?\( function](#)

# SATURATION(...)

**Syntax:** SATURATION(color)

**Description:** The `saturation()` function extracts the saturation intensity from a color. **Saturation** specifies how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? When using the standard Apple color picker wheel, the Saturation would specify the distance of the color from the center of the wheel. This is a number from 0 to 65535.

**Parameters:** This function has one parameter: `color`.

**color** is the color you want to extract information from. This must be a six byte binary data value (see [binary data](#)).

**Result:** This function extracts the intensity of the saturation of this color. This intensity is a number between 0 and 65535.

**Examples:** The example below calculates the saturation intensity of the color (in percent, from 0 to 100%).

```
Intensity=saturation(HighlightColor)*100/65535
```

For more examples of color, see [colors](#).

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the color parameter.

**See Also:** [rgb\(\)](#) function  
[hsb\(\)](#) function  
[green\(\)](#) function  
[blue\(\)](#) function  
[red\(\)](#) function  
[hue\(\)](#) function  
[brightness\(\)](#) function  
[objectinfo\(\)](#) function  
[changeobjects](#) function  
[colorwheel](#) statement  
[colors](#)

# SAVE

**Syntax:** SAVE

**Description:** The **save** statement saves the current database to the disk. This command has the same effect as choosing the **Save** command from the File menu.

**Parameters:** This statement has no parameters.

**Examples:** The procedure removes all records more than one year old, then saves the database. After the database is saved there is no way to get the old data back again (unless you have a backup).

```
select Date>today()-365
removeunselected
save
```

**Views:** This statement may be used in any view.

**See Also:** [saveall](#) statement  
[saveas](#) statement  
[savecopyas](#) statement  
[revert](#) statement  
[filesave](#) statement  
[info\("changes"\)](#) function

# SAVEACOPYAS

- Syntax:** SAVEACOPYAS file
- Description:** The `savecopyas` statement saves a copy of the currently active file under a new name. The original remains open in memory under its original name.
- Parameters:** This statement has one parameter: file.
- file** is the name of the new copy of the file you wish to save. The file name may be up to 31 characters long, and may not contain the `:` character.
- If the new file should be saved in a different folder than the current database. The file name must contain a complete path description. A path description is a list of the folders the file is in, with each folder separated by a colon (for example `Disk:Joe:January`). You can create a path from a folder id with the [folderpath\(\)](#) function. The path and file name may be up to 255 characters long.
- If the file parameter is the word `dialog` (no quotes), the procedure will stop and display the standard **Save A Copy As** dialog. The user may type in a name and select a folder. The procedure will then save the database and continue.
- Action:** This statement performs the same action as the **Save a Copy As** command in the File menu. The statement makes a copy of the current database on the disk, while leaving the original untouched and ready for further work. (Note: The original database is not saved to the disk, but it can be by simply using the [save](#) statement.)
- Examples:** This example saves a copy of the current file. If the original database is `Sales`, the copy will be something like `Copy of Sales 3/16`.
- ```
saveas "Copy of "+info("databasename")+
datepattern( today(), "mm/dd")
```
- This example lets the user choose the new name and location for the database, just as if they had chosen the **Save A Copy As** command in the File menu.
- ```
savecopyas dialog
```
- Views:** This statement may be used in any view.
- See Also:** [save](#) statement  
[saveas](#) statement

# SAVEALL

**Syntax:** SAVEALL

**Description:** The **saveall** statement saves every open database to the disk. (However, if a database has not been modified it is not saved.) This command has the same effect as choosing the **Save ALL** command from the File menu

**Parameters:** This statement has no parameters.

**Examples:** The procedure removes all records more than one year old from two databases, then saves both databases. After the database is saved there is no way to get the old data back again (unless you have a backup). **Warning:** The **saveall** statement will also save any other databases that happen to be open in addition to the **Inventory** and **Invoices** databases. Be careful with this statement, or you may save more than you intended.

```
window "Inventory"
select Date>today()-365
removeunselected
window "Invoices"
select Date>today()-365
removeunselected
saveall
```

**Views:** This statement may be used in any view.

**See Also:** [save](#) statement  
[saveas](#) statement  
[savecopyas](#) statement  
[filesave](#) statement

# SAVEAS

- Syntax:** SAVEAS file
- Description:** The **saveas** statement saves a copy of the currently active file under a new name. The copy remains in memory.
- Parameters:** This statement has one parameter: file.
- file** is the name of the file you wish to save. The file name may be up to 31 characters long, and may not contain the : character.
- If the file should be saved in a different folder than the current database. The file name must contain a complete path description. A path description is a list of the folders the file is in, with each folder separated by a colon (for example Disk:Joe:January). You can create a path from a folder id with the [folderpath\(\)](#) function. The path and file name may be up to 255 characters long.
- If the file parameter is the word dialog (no quotes), the procedure will stop and display the standard **Save As** dialog. The user may type in a name and select a folder. The procedure will then save the database and continue.
- Action:** This statement performs the same action as the **Save As** command in the **File** menu. The current database is changed to a new name, then saved. (Note: The original database is not saved, so it will not have the most recent changes.)
- Another way to export data is using the [export](#) statement. The [export](#) statement is often easier to use because it does not rely on a special form.
- Examples:** This example assumes that the current database has a name like **Sales 07/96**. The procedure will save the database on the disk. If this is a new month it will save the database under a new name for the new month (for example **Sales 08/96**), leaving the old month on the disk.
- ```

local fileroot,filemonth
fileroot=info("databasename")[1,-6]
filemonth=info("databasename")[-5,-1]
if filemonth=datepattern( today(),"MM/YY")
    save
else
    saveas fileroot+datepattern( today(),"MM/YY")
endif

```
- This example lets the user choose the new name and location for the database, just as if they had chosen the **Save As** command in the File menu.
- ```

saveas dialog

```
- Views:** This statement may be used in any view.
- See Also:** [save](#) statement  
[savecopyas](#) statement



# SAVEASTEXT

**Syntax:** SAVEASTEXT file

**Description:** The `saveastext` statement exports the current database into a text file, using a special form to control what data is exported. If the text file does not already exist it will be created. **If it does already exist its contents will be replaced!**

**Parameters:** This statement has one parameter: file.

**file** is the name of the file you wish to save. The file name may be up to 31 characters long, and may not contain the `:` character.

If the file should be saved in a different folder than the current database. The file name must contain a complete path description. A path description is a list of the folders the file is in, with each folder separated by a colon (for example `Disk:Joe:January`). You can create a path from a folder id with the `folderpath()` function. The path and file name may be up to 255 characters long.

**Action:** This statement exports data by scanning the current database and outputting each record with a special form. The form may contain up to 3 report tiles—a data tile (required), a summary tile (optional) and a group header tile (optional). Each tile must contain one (and only one) auto-wrap text object. The exact positioning of the auto-wrap text object on the tile is unimportant, as long as there is one text object per tile. The `saveastext` statement will use the auto-wrap text object as a template to output the data from the file.

Another way to export data is using the `export` statement. The `export` statement is often easier to use because it does not rely on a special form.

**Examples:** This example exports all fields from all selected records in the current database. The form Customer Export Template is used as a template, and must be set up as described in the previous section. The procedure asks the user to specify the name and folder for the new file.

```
local file, folder
openform "Customer Export Template"
savefiledialog folder, file, "Export file name"
if file="" stop endif
saveastext folderpath(folder)+file
```

The procedure below creates a text file named List.TXT in that contains data for every record in Iowa.

```
select State="IA"
openform "Customer Export Template"
saveastext "List.TXT"
```

**Views:** This statement must be used in a Form view. The form must be specially prepared for use with this statement (see Action, above).

**See Also:**

- [export](#) statement
- [filesave](#) statement
- [filetrash](#) statement
- [openfiledialog](#) statement
- [savefiledialog](#) statement
- [exportline\(\)](#) function
- [folder\(\)](#) function
- [folderpath\(\)](#) function
- [fileload\(\)](#) function
- [filesize\(\)](#) function

# SAVEFILEDIALOG

- Syntax:** SAVEFILEDIALOG folder,filename,prompt
- Description:** The `savefiledialog` statement pauses a procedure and displays the standard “save file” dialog. This is the same dialog that most applications use for saving a file. The user can then type in the name of a file and select a folder from the disk. Once the user is done the procedure resumes and can process the file with the [export](#) or [filesave](#) statements.
- Parameters:** This statement has three parameters: `folder`, `filename` and `prompt`.
- folder** should be a variable. When the statement is finished this variable will contain a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. The path id can be converted into a text description of the path with the [folderpath\(\)](#) function.
- filename** should be a variable. The procedure should place a default file name in this variable (or fill with empty text (“”) if you want to force the user to type something in. When the statement is finished the variable will contain the name of the file that was the user typed in. If the filename parameter is empty the user pressed the Cancel button.
- prompt** is the only parameter that you actually supply. This is a short message that will appear in the dialog, for example [Export as:](#), or [Save Picture](#).
- Action:** This statement causes the standard “save file” dialog to appear. This allows the user to select a folder and type in a file name. The statement will check to see if this file already exists, if it does, it will ask the user if he or she really wants to replace this file with a new one.
- Examples:** The procedure exports this months invoice data into a text file. The user may select the name and folder for the new text file.

```

local file,folder
select Date>month1st(today())
/* set up default filename, (Example: "Invoices Aug 92" */
file="Invoices "+datepattern(today(),"Mon yy")
savefiledialog folder,file,"Export file name"
if file=""
 stop /* pressed cancel, so stop */
endif
export folderpath(folder)+file,exportline()+¶

```

## Custom Dialogs

If you are using Panorama 3.1 or later, you can customize the open file dialog by using the [customdialog](#) statement (Macintosh only). The customization options available include changing the layout of the dialog, adding extra text to the dialog and adding extra push buttons to the dialog. (You cannot add other kinds of controls to the dialog, for example checkboxes, radio buttons, or pop-up menus.)

Most of the work in setting up a custom dialog involves creating a resource template for the dialog. To do this you will need a resource editing program like ResEdit or Resorcerer. (See “[Working with Resources](#)” on page 1688 for more information on these programs, or consult the documentation for the programs themselves.) Once the resource template is set up, it can be used in any procedure by inserting the `customdialog` statement just before the [openfiledialog](#) or `savefiledialog` statements.

The resource for an save file dialog must contain at least the required items listed below. Once the required items are set up in this order you can add additional items of your own. The easiest way to do this correctly is to make a copy of DLOG 9001 in the File Dialogs.rsrc file, then adjust the layout and add your own items as necessary.

- 1) Save Button (you may rename this button)
- 2) Cancel Button
- 3) Prompt Text (This is overwritten by the 2nd parameter of the savefiledialog statement)
- 4) Disk Name
- 5) Eject Button
- 6) Drive Button
- 7) File Name
- 8) Dotted Line

To use your custom file dialog in a procedure you must place the customdialog statement just before the savefiledialog statement, like this:

```

local folder,file
customdialog 9038
savefiledialog folder,file,"Export File:"
case info("dialogtrigger") contains "Save Tab Delimited"
...
case info("dialogtrigger") contains "Save Comma Delimited"
...
endcase

```

As this example shows, the `info("dialogtrigger")` will contain the name of the push button that the user pressed.

**Views:** This statement may be used in any view.

**See Also:**

- [customdialog](#) statement
- [openfiledialog](#) statement
- [openfile](#) statement
- [filesave](#) statement
- [filerename](#) statement
- [filetrash](#) statement
- [folder\(\)](#) function
- [folderpath\(\)](#) function
- [fileload\(\)](#) function
- [filesize\(\)](#) function

# SAVEVARIABLES

- Syntax:** SAVEVARIABLES variablelist,combinedarray,separator
- Description:** The **savevariables** statement takes a list of variables and combines the values of all the variables into a single array. (Later you can unpack the array back onto variables with the [loadvariables](#) statement.)
- Parameters:** This statement has three parameters: **variablelist**, **combinedarray** and **separator**.
- variablelist** is an array containing the names of the variables to be "saved" in the output array. Each item in the array must be separated from the next by the **separator** character.
- combinedarray** is a field or variable name. The statement will build the final array of values in this field or variable, using the **separator** character to divide each item. Any numbers will be converted to text as the array is built.
- separator** is the character that will be used to separate the values in the combinedarray. Common separators include carriage return (¶) and tab (→). You should be careful to make sure that the separator character is a character that will not appear in any of the variables being saved. One way to make sure of this is to use a character that cannot normally be generated from the keyboard, for example [chr\(1\)](#) or [chr\(255\)](#). For more information on separators see [text arrays](#).
- Action:** This statement combines two or more variables into a single variable containing an array. You could perform the same operation with a loop, but the **savevariables** statement is much faster. One use for the **savevariables** statement is exporting a group of variables to a field or file. Then later you can use [loadvariables](#) to recover the previous variable values.
- Examples:** The example below saves all of the [fileglobal](#) variables for the currently active database into a disk file. The disk file will have the name of the database plus Variables, for example Contact Variables or Invoice Variables. You can look at this file and see the variable values using any text editor (SimpleText, BBEdit, etc.).

```
local fileExtraData
savevariables info("filevariables"),fileExtraData,¶
filesave " ",info("databasename")+ " Variables", " ",fileExtraData
```

The following example is similar but it saves both the variable name and the data in the format **variable=value**. The variables will be listed in alphabetical order.

```
local varNames,varData
varNames=info("filevariables")
arraysort varNames,varNames,¶
savevariables varNames,varData,¶
arrayfilter varData,varData,¶,
array(info("filevariables"),seq(),¶)+"="+iimport()
filesave " ",info("databasename")+ " Variables", " ",varData
```

The resulting file will be named something like Contact Variables and will look something like this:

```
ActiveForm=Contacts
LocalAreaCode=714
SearchText=Chicago
```

For an example of how to load this file back into the individual variables, see the [load-variables](#) statement.

**Views:** This statement may be used in any view.

**See Also:** [loadvariables](#) statement  
[loadglobalvariables](#) statement  
[loadfilevariables](#) statement  
[loadlocalvariables](#) statement  
[loadwindowvariables](#) statement

# SCRAPCALC

**Syntax:** SCRAPCALC formula

**Description:** The `scrapcalc` statement calculates a formula and stores the result in the clipboard. (Note: The same effect can be achieved by putting the word `clipboard` on the left side of an equation.)

**Parameters:** This statement has one parameter: formula.

**formula** calculates the value that will be placed into the clipboard.

**Examples:** Panorama normally copies data into fields or variables with an assignment, for example `City="Westside"`. However, early versions of Panorama did not have assignments. Instead, they used the `scrapcalc` and `paste` statements. For example, this procedure will put the value `Westside` into the City field.

```
field City
scrapcalc "Westside"
paste
```

Assignments make this use of the `scrapcalc` statement obsolete. Sometimes, however, you may want to use the clipboard to communicate with other programs. This example copies the name and address onto the clipboard in mailing label format. The user could then paste the address into a letter or other word processing document.

```
scrapcalc Name+¶+Address+¶+City+", "+State+" "+Zip
```

Here is the same procedure rewritten using an assignment instead of using the `scrapcalc` statement.

```
clipboard=Name+¶+Address+¶+City+", "+State+" "+Zip
```

**Views:** This statement may be used in any view.

**See Also:** [formulacalc](#) statement  
[set](#) statement  
[clipboard\(\)](#) function

# SCRATCHMEMORY

- Syntax:** `SCRATCHMEMORY memory`
- Description:** The `scratchmemory` statement changes the amount of memory reserved as scratch memory.
- Parameters:** This statement has one parameter: `memory`.  
**memory** is the amount of scratch memory you want Panorama to use (in bytes). The minimum value is 200000 (200k).
- Action:** This statement changes the amount of memory reserved as scratch memory. Scratch memory is used for fonts, displaying pictures, printing, and other system tasks. Increasing scratch memory reduces the amount of memory available for databases. The minimum scratch memory setting is 200000 (200K). If Panorama cannot change the scratch memory size to the new value (not enough memory) an alert will appear (The alert may be bypassed with the `if error` statement.) (Note: To increase the scratch memory allocation, you may need to increase Panorama's overall memory allocation. To do this you must select Panorama with the Finder, choose **Get Info** from the File menu, then increase the Preferred memory size.)
- Examples:** The example attempts to set the scratch memory size to 400k. If there is not enough memory to do this it tries smaller amounts in 25K increments (375k, 350k, 325k, etc.) until it is successful.
- ```

local bigScratch
bigScratch=400000
loop
  scratchmemory bigScratch
  if error
    bigScratch=bigScratch-25000
  else
    bigScratch=0
  endif
while bigScratch>200000

```
- Views:** This statement may be used in any view.
- See Also:** [info\("scratchmemory"\)](#) function
[scratchmemorytemporary](#) statement

SCRATCHMEMORYTEMPORARY

- Syntax:** `SCRATCHMEMORYTEMPORARY memory`
- Description:** The `scratchmemorytemporary` statement temporarily changes the amount of memory reserved as scratch memory. The temporary setting remains until you quit from Panorama. The next time Panorama is launched the original scratch memory settings will return.
- Parameters:** This statement has one parameter: `memory`.
memory is the amount of scratch memory you want Panorama to use (in bytes). The minimum value is 200000 (200k).
- Action:** This statement changes the amount of memory reserved as scratch memory. Scratch memory is used for fonts, displaying pictures, printing, and other system tasks. Increasing scratch memory reduces the amount of memory available for databases. The minimum scratch memory setting is 200000 (200K). If Panorama cannot change the scratch memory size to the new value (not enough memory) an alert will appear (The alert may be bypassed with the `if error` statement.) (Note: To increase the scratch memory allocation, you may need to increase Panorama's overall memory allocation. To do this you must select Panorama with the Finder, choose **Get Info** from the File menu, then increase the Preferred memory size.)
- Examples:** The example attempts to set the scratch memory size to 400k. If there is not enough memory to do this it tries smaller amounts in 25K increments (375k, 350k, 325k, etc.) until it is successful.
- ```

local bigScratch
bigScratch=400000
loop
 scratchmemorytemporary bigScratch
 if error
 bigScratch=bigScratch-25000
 else
 bigScratch=0
 endif
while bigScratch>200000

```
- Views:** This statement may be used in any view.
- See Also:** [info\("scratchmemory"\)](#) function  
[scratchmemory](#) statement

# Scroll Bar Programming

**Background:** The [superobject](#) and [activesuperobject](#) statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

**Quick**

```
"GetScrollMin" ,<Value>
"SetScrollMin" ,<Value>
"GetScrollMax" ,<Value>
"SetScrollMax" ,<Value>
"GetScrollPage" ,<Value>
"SetScrollPage" ,<Value>
"GetScrollValue" ,<Value>
"DisableScrollBar"
"EnableScrollBar"
"GetScrollEnable" ,<Value>
```

**GetScrollMin** ,<Value>

This command gets the minimum scroll bar value and places into the field or variable specified by <Value>.

**SetScrollMin** ,<Value>

This command sets the minimum scroll bar value to any numeric value (must be integer) between 1 and 65535. (This value is normally set by the Min: value in the Scroll Bar dialog.)

**GetScrollMax** ,<Value>

This command gets the maximum scroll bar value and places into the field or variable specified by <Value>.

**SetScrollMax** ,<Value>

This command sets the maximum scroll bar value to any numeric value (must be integer) between 1 and 65535. (This value is normally set by the Max: value in the Scroll Bar dialog.) Here is an example that sets the maximum value of the scroll bar named Slider to the number of elements in the array People:

```
superobject "Slider" ,"SetScrollMax" ,arraysize(People,¶)
```

**GetScroll-Page** ,<Value>

This value is the amount the scroll bar value will increase or decrease if the user clicks on the gray area above or below the thumb of the scroll bar.

**SetScrollPage** ,<Value>

This command sets the scroll bar page amount to any numeric value (must be integer) between 1 and 65535. This value is the amount the scroll bar value will increase or decrease if the user clicks on the gray area above or below the thumb of the scroll bar. (This value is normally set by the Page Up/Down: value in the Scroll Bar dialog.)

- GetScrollValue**     , <Value>
- This command gets the current position of the scroll bar. This command is redundant because you can always get the position by examining the field or variable linked to the scroll bar.
- DisableScroll-  
Bar**             This command disables the scroll bar. The scroll bar is still visible, but it turns white and the thumb disappears.
- EnableScroll-  
Bar**             This command enables the scroll bar (see previous command).
- GetScrollEna-  
ble**             , <Value>
- This command checks to see if a scroll bar is enabled or disabled, and sets the field or variable specified by <Value> with a true or false result accordingly.

# SEARCH(...)

**Syntax:** SEARCH(text,phrase)

**Description:** The search( function searches through an item of text looking for a character, word or phrase. If it finds an exact match (including upper/lower case) with the character, word or phrase it returns it's position within the text item. If it does not find the character, word or phrase it returns zero.

**Parameters:** This function has two parameters: text. and phrase.

**text** is the item of text that you want to search through.

**phrase** is the character, word or phrase that you want to search for. This must be the exact character, word, or phrase, including upper or lower case. The search( function will not match **Dr** with **DR** or **dr**.

**Result:** The result of this function is always a positive integer numeric value, for example 0, 1, 2, 3, etc.

**Examples:** This simple procedure uses the search( function to attempt to locate and display a fax number in the Notes field. This procedure assumes that the fax number will look something like this: fax (999) 555-0123.

```

local X
X=search(lower(Notes),"fax (")
if X≠0
 message "Fax Number: "+Notes[X+4;14]
endif

```

The example starts by searching for the phrase fax (. If it finds this phrase, it displays the 14 characters starting with the ( symbol. (By using the lower( function the procedure makes sure that search( will find a match even if the word fax is all or partially in upper case.) By removing the text that has already been searched through a procedure can locate multiple occurrences of a character, word or phrase. This procedure attempts to locate every phone number and collect them all into a list.

```

local X,XNotes,XPhone,PhoneList
X=1 XNotes=Notes
PhoneList=""
loop
 X=search(XNotes,"(")
 stoploopif X=0
 XPhone=XNotes[X;14]
 if XPhone match "(???) ???-????"
 PhoneList=sandwich(" ",PhoneList,"")+XPhone
 X=X+14
 endif
 XNotes=XNotes[X+1,-1]
while forever
message "Phone Numbers: "+PhoneList

```

After each successful search the procedure lops off the section of the text that has already been searched, and then starts over again. The process continues until there are no more matches.

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `search(Notes,34)`. If you have a number you must convert the number to text before using it with this function, for example `length(Notes,str(34))`.

**See Also:** [replace](#)( function  
[replacemultiple](#)( function  
[extract](#)( function  
[array](#)( function

# SECONDS(...)

**Syntax:** SECONDS(TEXT)

**Description:** The `seconds()` function converts text into a number representing a time. See also [time\(\)](#)

**Parameters:** This function has one parameter: `text`.

`text` is the text that you want to convert to a number representing a time. The text must contain a valid time. Here are some examples of valid times:

```
4:13 PM
11:00 AM
2:30
18:45
```

The [time\(\)](#) function also converts text into a number, but is more flexible about the time formats it will accept.

**Result:** This function returns a number representing the time. The number is the number of seconds since midnight. For example, if the time is 10:23 AM this function will return the number 37,380.

**Examples:** The example below asks the user to enter a time, then selects all the flights departing between one hour before and one hour after the specified (1 hour contains 3600 seconds). The text entered by the user is converted to a number by the `seconds()` function before it is compared with the `DepartureTime` field.

```
local xTime
xTime=""
getttext "Select flights around what time?",xTime
select DepartureTime>seconds(xTime)-3600 and
DepartureTime<seconds(xTime)+3600
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to convert a numeric value.

**See Also:** [timepattern\(\)](#) function  
[now\(\)](#) function

# SELECT

**Syntax:** `SELECT true-false test`

**Description:** The **Select** statement makes visible only those records for the active database which match the true-false test. This statement will work on all fields except **Picture** type fields.

**Parameters:** This statement has one parameter: true-false test.

**true-false test** may be one or more functions or equations which result in a true or a false condition. Multiple true-false tests must be separated by an **and** or an **or** operator. Grouping true-false tests inside parenthesis ( ) will give those tests priority in the processing order when Panorama evaluates them.

**Action:** This statement is used to make visible all records that match the true portion of the true-false test. All records not matching the select criteria will be invisible. If a subset of records are already selected from the database select will examine **all** records looking for a match. You do not need to have the cursor on the field you are performing the select on prior to executing the select.

Compound true-false tests connected by an **or** operator(s) requires only one of the tests to be true to make the test true. Compound true-false tests connected by an **and** operator(s) requires all tests to evaluate true to make the test true.

The record count, in the horizontal scroll bar, will reflect any change in the visible record count after the result of a successful select operation. The record count option can be turned on or off by using the **Show Record Count** command under the Setup menu.

**Note:** If no records match a true test and the `select` statement is the last procedure statement executed Panorama will display an **alert dialog** warning your that no records were selected and reverts you back to the previous selection. If the `select` statement is not the last procedure statement and the select fails the warning will be suppressed. You may use the `info("empty")` function to test to see if a select fails (see examples below.)

**Examples:** This statement has the same effect as clicking on the **Select** button in the **Find/Select** dialog, Search menu. This example selects all records for John Smith.

```
select Customer = "John Smith"
```

This example selects all records for Votes over 12000.

```
select Votes > 12000
```

This example selects all records on or after **January 1 1995**.

```
select «Date Shipped» ≥ date("1/1/95")
```

This example shows how you can test to see if records were selected matching the input Work Order No. using the `info("empty")` function. Work Order No. is a numeric field.

```
getscrap "Enter Find value."
select «Work Order No.» = val(clipboard())
if (not info("empty"))
 field «Mark Record»
 fill "X"
else
 message "There are no records for Work Order: "+ clipboard()
stop
```

```
endif
...
...
...
```

In this example the `info("empty")` function is used to test to see if no records were selected for the date entered.

```
local ADate
gettext "Enter date ex. 12/31/95",ADate
select «Process Date» = date(ADate)
if info("empty")
 beep
 message "No records match this date: "+ADate
 stop
endif
field «Sales Rep»
...
...
...
```

This example uses a compound true-false test that must match a name to either Customer or Company and who has made a Purchase Price more than \$500.

```
local Name
gettext "Enter name:",Name
select (Customer contains Name or Company contains Name)
 and «Purchase Price» > 500
if info("empty")
 beep
 message "No records selected."
endif
```

This example uses a formula to test multiple text fields to select the records where the variable's contents matches any one of those fields.

```
local TheText
gettext "Enter selection criteria:",TheText
select " "+Customer+" "+Company+" "+Address+" "+Comments contains TheText
if info("empty")
 beep
 message "No records selected."
endif
```

**Views:** This statement may be used in any view.

**See Also:**

- [find](#) statement
- [findselect](#) statement
- [formulafindselect](#) statement
- [info\("empty"\)](#) function
- [info\("found"\)](#) function
- [info\("records"\)](#) function
- [info\("selected"\)](#) function
- [next](#) statement
- [selectadditional](#) statement
- [selectall](#) statement



selectreverse statement  
selectsummaries statement  
selectwithin statement

# SELECTADDITIONAL

**Syntax:** SELECTADDITIONAL true-false test

**Description:** The `selectadditional` statement may add unselected records to a previously selected group if they match the true-false test.

**Parameters:** This statement has one parameter: true-false test.

**true-false test** may be one or more functions or equations which result in a true or a false condition. Multiple true-false tests must be separated by an `and` or an `or` operator. Grouping true-false tests inside parenthesis ( ) will give those tests priority in the processing order when Panorama evaluates them.

**Action:** This statement is used to add records to a previously selected group that match the true portion of the true-false test. All records not matching the true-false test will remain invisible. If a subset of records are already selected from the database `selectadditional` will examine all invisible records looking for a match. You do not need to have the cursor on the field you are performing the select on prior to executing the `selectadditional`.

Compound true-false tests connected by an `or` operator(s) requires only one of the tests to be true to make the test true. Compound true-false tests connected by an `and` operator(s) requires all tests to evaluate true to make the test true.

The record count, in the horizontal scroll bar, will reflect any change in the visible record count after the result of a successful `selectadditional` operation. The record count option can be turned on or off by using the **Show Record Count** command under the **Setup** menu.

**Note:** If no records match a true test and the `selectadditional` statement is the last procedure statement executed Panorama will display an **alert dialog** warning your that no records were selected and reverts you back to the previous selection. If the `selectadditional` statement is not the last procedure statement and the `selectadditional` fails the warning will be suppressed. You may use the `info("empty")` function to test to see if a `selectadditional` fails (see examples below.)

This statement will work on all fields except **Picture** type fields.

This statement has the same effect as clicking on the **Select Additional** button in the **Find/Select...** dialog, **Search** menu.

**Examples:** This example selects all records for **Hawaii** and then selects additional records for **California**.

```
select State = "HI"
selectadditional State = "CA"
```

This example selects all records for shipments on or after **January 1 1995** and then selects all records marked **Backorder**.

```
select «Date Shipped» ≥ date("1/1/95")
selectadditional «Status» = "Backorder"
```

This procedure will allow you to keep adding records to the selection until you enter a blank (null) for the City name, then it stops.

```

getscrap "Enter City"
select «City» = clipboard()
SelectMore: ;This is a label for the goto
getscrap "Enter City"
if clipboard() ≠ ""
 selectadditional «City» = clipboard()
 goto SelectMore
else
 stop
endif

```

In this example the info("empty") function is used to test to see if no records were selected for the dates entered.

```

local ADate
gettext "Enter Process Date ex. 12/31/95",ADate
select «Process Date» = date(ADate)
if info("empty")
 beep
 message "No records match this date: "+ADate
 stop
else
 gettext "Enter Release Date ex. 12/31/95",ADate
 selectadditional «Release Date» = date(ADate)
 if info("empty")
 beep
 message "No records match this date: "+ADate
 stop
 endif
endif
field «Sales Rep»
...
...

```

**Views:** This statement may be used in any view.

**See Also:**

- [find](#) statement
- [findselect](#) statement
- [formulafindselect](#) statement
- [info\("empty"\)](#) function
- [info\("found"\)](#) function
- [info\("records"\)](#) function
- [info\("selected"\)](#) function
- [next](#) statement
- [select](#) statement
- [selectall](#) statement
- [selectreverse](#) statement
- [selectsummaries](#) statement
- [selectwithin](#) statement

# SELECTALL

**Syntax:** SELECTALL

**Description:** The **Selectall** statement allows you to make all records in the database previously unselected visible.

**Parameters:** This statement has no parameters.

**Action:** This statement is used to make all previously unselected (or invisible) records selected (or visible). After successful completion the record count, in the horizontal scroll bar, should reflect success by showing that the number of visible records is the same as the total number of records. The record count option can be turned on or off by using the **Show Record Count** command under the Setup menu.

**Note:** If the record count shows a discrepancy between the number of visible records and the total number of records after the selectall statement is used it may be due to having previously collapsed a database to a summary level with either the [collapse](#) or the [outlinelevel](#) statements or commands. Combining the statement [outlinelevel](#) "data" in your procedure either before or after a selectall statement should make visible all records previously collapsed and made unselected. See example two below for more information.

This statement has the same effect as choosing the **Select All** command from the Search menu.

**Examples:** This example selects all records in the active database that were previous unselected.

```
selectall
```

This example illustrate three statements you should place at the beginning of a procedure as a general housekeeping task to make certain you are working with all of the data records only in your database.

```
selectall
outlinelevel "data"
removesummaries 7
```

This example uses the previous example combined with a procedure to export today's sales records, if they exist, and then selecting all the records at the end.

```
selectall
outlinelevel "data"
removesummaries 7
select «Purchase Date» = today()
if info("empty")
 opensound "SoundStuff"
 playsound "Bummer"
 message "No sales today."
 stop
endif
openform "Export Sales Data"
saveastext datepattern(today(), "mm/dd")+"Sales.txt"
closewindow
selectall
```

**Views:** This statement may be used in any view.

**See Also:** [find](#) statement  
[findselect](#) statement  
[formulafindselect](#) statement  
[info\("empty"\)](#) function  
[info\("found"\)](#) function  
[info\("records"\)](#) function  
[info\("selected"\)](#) function  
[next](#) statement  
[select](#) statement  
[selectadditional](#) statement  
[selectreverse](#) statement  
[selectsummaries](#) statement  
[selectwithin](#) statement

# SELECTALLOBJECTS

**Syntax:** SELECTALLOBJECTS

**Description:** The `selectallobjects` statement will select all the graphic objects in the current form. (This is the same concept as going into graphics mode and choosing the **Select All** command.) After the objects are selected you can get information about them or change the attributes of certain objects.

**Parameters:** This statement has no parameters.

**Action:** This statement allows a procedure to select all the graphic objects within a form. Once the objects are selected they can be examined or changed. This gives Panorama a limited capability to actually change forms on the fly.

**Examples:** This example will reduce the brightness of every object in the form by 20%.

```
selectallobjects
changeobjects "color",hsb(
hue(objectinfo("color")),
saturation(objectinfo(),0.8*brightness(objectinfo("color")))
selectnoobjects
```

**Views:** This statement may be used in Form views.

**See Also:** [changeobjects](#) statement  
[selectobjects](#) statement  
[selectnoobjects](#) statement  
[object](#) statement  
[objectid](#) statement  
[objectnumber](#) statement  
[objectinfo\(\)](#) function

# SELECTDUPLICATES

**Syntax:** `SELECTDUPLICATES formula`

**Description:** The `selectduplicates` statement locates duplicate information in the database.

**Parameters:** This statement has one parameter: `formula`.

**formula** tells Panorama what data to check for duplicates. If the formula is empty ("") Panorama will check in the current field. If the formula is not empty the formula will build the data that is actually checked for duplicates. For example to look for records where both the first name and last name are duplicated the formula would be `FirstName+" "+LastName`. (Note: The formula must produce a text item as the result. If you want to include numeric fields they must be converted to text with the `str()` or `pattern()` function. If you want to include date fields they must be converted to text with the `date-pattern()` function.)

**Action:** This statement selects duplicate records. To work properly, the database must be sorted according to the formula used (see the examples below).

This statement has the same effect as choosing the **Select Duplicates** command in the **Search** menu.

**Examples:** This example selects records with duplicate check numbers.

```
field «Check#»
sortup
selectduplicates ""
```

This example selects all records with duplicate names (both first and last name must be duplicated).

```
field FirstName
sortup
field LastName
selectwithin
selectduplicates FirstName+" "+LastName
```

**Views:** This statement may be used in any view.

**See Also:** [find](#) statement  
[findselect](#) statement  
[formulafindselect](#) statement  
[info\("empty"\)](#) function  
[info\("found"\)](#) function  
[info\("records"\)](#) function  
[info\("selected"\)](#) function  
[next](#) statement  
[select](#) statement  
[selectall](#) statement  
[selectreverse](#) statement

selectsummaries statement

selectwithin statement

sortup statement



# SELECTNOOBJECTS

**Syntax:** SELECTNOOBJECTS

**Description:** The `selectnoobjects` statement will unselect all the graphic objects in the current form. (This is the same concept as going into graphics mode and clicking on an empty spot.)

**Parameters:** This statement has no parameters

**Action:** This statement allows a procedure to unselect all the graphic objects within a form. Usually this is done to “clean up” after a procedure has selected graphic objects.

**Examples:** This example will reduce the brightness of every object in the form by 20%. When it is done it politely unselects the graphic objects.

```
selectallobjects
changeobjects "color",hsb(
hue(objectinfo("color")),
saturation(objectinfo(),0.8*brightness(objectinfo("color")))
selectnoobjects
```

**Views:** This statement may be used in Form views.

**See Also:** [changeobjects](#) statement  
[selectobjects](#) statement  
[selectallobjects](#) statement  
[object](#) statement  
[objectid](#) statement  
[objectnumber](#) statement  
[objectinfo\(\)](#) function

# SELECTOBJECTS

**Syntax:** SELECTOBJECTS formula

**Description:** The selectobjects statement will select one or more objects in the current form based on the formula supplied. (This is the same concept as going into graphics mode and clicking on objects to select them.) The formula will usually use the [objectinfo\(\)](#) function to get and compare information about each object. After the objects are selected you can get information about them or change the attributes of certain objects.

**Parameters:** This statement has one parameter: formula.

**formula** is a formula that is used to decide which objects in the current form will be selected. The **selectobjects** statement scans the form object by object. For each object it checks the formula to see if the result is true or false. If the result is true, the object is selected, otherwise it is not selected.

In order to make a true-false decision about a graphic object, the formula needs to be able to get information about the object. The only way to do this is with the [objectinfo\(\)](#) function. This function can return one of 17 different object attributes, including the location, size, name, font, text size, color, and more. Within the formula you can compare these attributes with the attributes you are looking for to decide which objects should be selected.

**Action:** This statement allows a procedure to select graphic objects within a form. Once the objects are selected they can be examined or changed. This gives Panorama a limited capability to actually change forms on the fly.

**Examples:** This example will change all objects in the top 2 inches (144 pixels) of the form to red.

```
selectobjects rtop(objectinfo("rectangle"))<144
changeobjects "color",rgb(65535,0,0)
selectnoobjects
```

This example will change all objects that contain 9 point text to Monaco.

```
selectobjects objectinfo("textsize")=9
changeobjects "font","Monaco"
selectnoobjects
```

**Views:** This statement may be used in Form views.

**See Also:** [changeobjects](#) statement  
[selectallobjects](#) statement  
[selectnoobjects](#) statement  
[object](#) statement  
[objectid](#) statement  
[objectnumber](#) statement  
[objectinfo\(\)](#) function

# SELECTREVERSE

**Syntax:** SELECTREVERSE

**Description:** The `selectreverse` statement allows you to make all visible records invisible and all invisible records visible for the active database.

**Parameters:** This statement has no parameters.

**Action:** This statement is used to reverse the selected and unselected with each other in the active database. For example, if a database has 10 records and 3 are selected prior to doing a `selectreverse`, afterwards 7 records will be selected and 3 will be unselected. After running this statement in a procedure the record count, in the horizontal scroll bar, should reflect the correct number of visible records. The record count option can be turned on or off by using the **Show Record Count** command under the Setup menu.

Since you must have at least one visible record in a Panorama database at any time, if all records are selected prior to doing a `selectreverse` the `selectreverse` will have no effect.

**Note:** If the visible record count differs from the viewed records on screen it may be due to having previously collapsed a database to a summary level with either the collapse or the `outlinelevel` statements or commands. Combining the statement `outlinelevel "data"` in your procedure either before or after a `selectreverse` statement should make the visible count match the viewed records on screen.

This statement has the same effect as choosing the **Select All** command from the Search menu.

**Examples:** This example reverses the selection, if one took place, in the active database.

```
selectreverse
```

This example uses the `select`, `selectadditional`, and `selectreverse` statements to make visible group of records **not** from **California, Alaska, or Hawaii** and who have paid more than **\$50.00** for their product, so that you can print mailing labels for them.

```
selectall
outlinelevel "data"
removesummaries 7
select «Purchase Price» > 50
selectadditional State="CA" or State="AK" or State="HI"
selectreverse
openform "Avery Labels 5160"
print ""
closewindow
selectall
```

**Views:** This statement may be used in any view.

**See Also:** [find](#) statement  
[findselect](#) statement  
[formulafindselect](#) statement  
[info\("empty"\)](#) function  
[info\("found"\)](#) function

info("records") function  
info("selected") function  
next statement  
select statement  
selectadditional statement  
selectall statement  
selectsummaries statement  
selectwithin statement

# SELECTSUMMARIES

**Syntax:** SELECTSUMMARIES

**Description:** The `selectsummaries` statement selects all the summary records, and makes the data records invisible. It then converts the summary records into data records.

**Parameters:** This statement has no parameters

**Examples:** The `selectsummaries` statement lets you use summary records in a non standard way. In the data sheet you can convert an individual record into a summary record by clicking on the left edge of the record. Using this technique you can arbitrarily scan through a database and convert a few records to summary records. Then you could use this procedure to select the summary records and turn them back into data records.

```
selectsummaries
```

ProVUE does not recommend this use for summary records. Instead, we suggest that you add a field to your database for marking records. You can easily set up a checkbox to mark or unmark a record, and then use a procedure like this to select the marked records.

```
select Marked≠" "
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [info\("empty"\) function](#)  
[info\("found"\) function](#)  
[info\("records"\) function](#)  
[info\("selected"\) function](#)  
[next](#) statement  
[outlinelevel](#) statement  
[removedetail](#) statement  
[select](#) statement  
[selectadditional](#) statement  
[selectall](#) statement  
[selectreverse](#) statement  
[selectwithin](#) statement  
[summarylevel](#) statement

# SELECTWITHIN

**Syntax:** SELECTWITHIN true-false test

**Description:** The `selectwithin` statement may reduce a selected group of records to a smaller group provided they match the true-false test.

**Parameters:** This statement has one parameter: true-false test.

**true-false** test may be one or more functions or equations which result in a true or a false condition. Multiple true-false tests must be separated by an **and** or an **or** operator. Grouping true-false tests inside parenthesis ( ) will give those tests priority in the processing order when Panorama evaluates them.

**Action:** This statement is used to remove records from a previously selected group. The records kept in the group must match the true portion of the true-false test for `selectwithin`. All records not matching the true-false test will become invisible. If a subset of records are already selected from the database `selectwithin` will examine only the visible records looking for a match. You do not need to have the cursor on the field you are performing the select on prior to executing the `selectadditional`.

Compound true-false tests connected by an **or** operator(s) requires only one of the tests to be true to make the test true. Compound true-false tests connected by an **and** operator(s) requires all tests to evaluate true to make the test true.

The record count, in the horizontal scroll bar, will reflect any change in the visible record count after the result of a successful `selectwithin` operation. The record count option can be turned on or off by using the **Show Record Count** command under the Setup menu.

**Note:** If no records match a true test and the `selectwithin` statement is the last procedure statement executed Panorama will display an **alert dialog** warning your that no records were selected and reverts you back to the previous selection. If the `selectwithin` statement is not the last procedure statement and the `selectwithin` fails the warning will be suppressed. You may use the `info("empty")` function to test to see if a `selectwithin` fails (see examples below.)

This statement will work on all fields except **Picture** type fields.

This statement has the same effect as clicking on the **Select Within** button in the **Find/Select** dialog, Search menu.

**Examples:** This example selects all records for **Hawaii** and then narrows that group down to records in **Maui** only.

```
select State = "HI"
selectwithin City = "Maui"
```

The previous example's results could also have been achieved by this single procedure statement.

```
select State = "HI" and City = "Maui"
```

This example selects all records for shipments on or after **January 1 1995** and then reduces that group to shipments in **New Jersey**.

```
select «Date Shipped» ≥ date("1/1/95")
selectwithin «City» = "NJ"
```

This procedure will allow you to reduce the selected group by a maximum purchase price.

```
getscrap "Enter cut-off date."
select «Purchase Date» = date(clipboard())
if info("empty")
 message "No records selected, try again."
 stop
endif
TryAgain:
getscrap "Enter maximum price."
selectwithin «Price» < val(clipboard())
if info("empty")
 message "No records below "+clipboard()+", try again."
 goto TryAgain
endif
```

This procedure uses a compound true-false test for the selectwithin statement to narrow down the selection of records to ones with a blank Ship To field and that match the first three digits of Zip to the numbers entered.

```
getscrap "Enter cut-off date."
select «Purchase Date» = date(clipboard())
if info("empty")
 message "No records selected, try again."
 stop
endif
getscrap "Enter Zip[1,3]."
```

```
selectwithin «Ship To» = "" and Zip[1,3] = clipboard()
if info("empty")
 message "No records selected, try again."
 stop
endif
```

**Views:** This statement may be used in any view.

**See Also:**

- [find](#) statement
- [findselect](#) statement
- [formulafindselect](#) statement
- [info\("empty"\)](#) function
- [info\("found"\)](#) function
- [info\("records"\)](#) function
- [info\("selected"\)](#) function
- [next](#) statement
- [select](#) statement
- [selectadditional](#) statement
- [selectall](#) statement
- [selectreverse](#) statement
- [selectsummaries](#) statement

# SEQ(...)

- Syntax:** SEQ()
- Description:** The `seq()` function returns a sequential numbers (1, 2, 3, etc.). This function only works in conjunction with the [formulafill](#), [select](#), [find](#), and [arrayfilter](#) statements.
- Parameters:** This function has no parameters.
- Result:** When it is used with the [formulafill](#), [find](#) or [select](#) statements, the `seq()` function return a sequential number for each record (the first selected record is 1, the second is 2, etc.).
- When it is used with the [arrayfilter](#) statement, the `seq()` function returns a sequential number for each element in the array being processed (the first array element is 1, the second is 2, the third is 3, etc.).
- When it is used at any other time, the `seq()` function returns the number 1.
- Examples:** This procedure selects the first 10 records in the database:
- ```
select seq()≤10
```
- The procedure below will select every 5th record in the database.
- ```
select (seq() mod 5)=0
```
- By changing the 0 in this formula to 1, 2, 3, or 4 you can select different sets of data, but still 20% of the database each time. This procedure can be used as a subroutine. It takes the array passed to the subroutine as parameter 1 and adds line numbers to it. Parameter 2 is the separator character for the array. The line numbers will be surrounded by parenthesis, for example (1) New York, (2) Tokyo, etc.
- ```
local tempArray
tempArray=parameter(1)
arrayfilter tempArray,tempArray,parameter(2),(" "+str(seq())) "+import()
setparameter 2,tempArray
```
- Errors:** This function does not produce any errors.
- See Also:** [select](#) statement
[find](#) statement
[formulafill](#) statement
[import](#)(function

SEQUENCE

Syntax: SEQUENCE values

Description: The **sequence** statement fills every visible cell in the current field with a numeric sequence, for example 1, 2, 3 or 12, 24, 36,

Parameters: This statement has one parameter: values

values is a text item that specifies the starting and increment values for the sequence. If **values** contains two numbers the first number is the starting value and the second number is the increment value (amount that will be added each time). If **values** contains one number this number is used as the starting value and the increment value is one. Here are some examples of **values** parameters and the resulting sequence of numbers:

| Values | Generated Sequence |
|-----------|-----------------------------------|
| "1" | 1, 2, 3, 4, 5, 6, ... |
| "100" | 100, 101, 102, 103, 104, 105, ... |
| "5 5" | 5, 10, 15, 20, 25, 30, 35, ... |
| "100 -1" | 100, 99, 98, 97, 96, 95, ... |
| "0.1 0.1" | 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, ... |

When **values** is the word **dialog** Panorama will pause the procedure and present the user with the **Sequence** dialog. This allows the user to enter his or her own starting value and increment. Clicking on the **Ok** button will allow the procedure to continue.

Action: This statement fills the current field with numbers according to an arithmetic sequence. It has the same effect as using the **Sequence** command from the Math menu.

Note: This statement may only be used if the current field is a Numeric field. The sequence statement will not work with text, dates, or pictures.

Examples: This example assigns numbers to invoices that don't have a number yet, starting with 1000.

```
field InvoiceNumber
select sizeof(InvoiceNumber)=0
sequence "1000 1"
```

This example adds a new field for customer number, and automatically assigns numbers to each customer.

```
addfield CustNumber
fieldtype "0 digits"
sequence "1 1"
```

Views: This statement may be used in any view.

See Also:

- [fill](#) statement
- [runningtotal](#) statement
- [runningdifference](#) statement
- [emptyfill](#) statement
- [formulafill](#) statement
- [propagate](#) statement
- [propagateup](#) statement
- [unpropagate](#) statement
- [unpropagateup](#) statement

SERVERFILE

Syntax: SERVERFILE filename

Description: The **serverfile** statement changes the name of the SQL server file that this Partner/Server database is attached to (if any). This statement should only be used after you rename the SQL server file using the Finder. It cannot be used to attach the Panorama database to a different SQL server database.

Parameters: This statement has one parameter: filename.

filename is the name of the SQL server database that this Panorama database should be linked to. The file name must contain only upper case letters, numbers, and underscore (_) characters. It must not contain any spaces, lower case letters, or any other punctuation or symbol. Here are some examples of valid SQL file names:

```
INVOICES_96
HUMAN_RESOURCES
FALL_REGISTRATION
```

Here are some examples of illegal SQL file names:

```
INVOICES 96
Human Resources
FALL_REGISTRATION@UCSD
```

Action: This statement changes the name of the server file that this database is linked to. **This is a very dangerous statement!** You should only use this statement if you have renamed the server database file on the server computer. (**Note:** The [info\("serverfile"\)](#) function allows a procedure to find out the current name of the server file linked to this database.)

Examples: Suppose you have an SQL database file named STUDENT_RECORDS, and you rename that file as TRANSCRIPTS on the server machine (using the Finder to rename the file). The statement shown below would re-establish the connection between the local Panorama file and the server database.

```
serverfile "TRANSCRIPTS"
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [info\("serverfile"\)](#) function
[sqlconnection](#) statement

SERVERLOOKUP

Syntax: `SERVERLOOKUP status`

Description: The `serverlookup` statement controls whether lookups are made in the local database or refer to the SQL server

Parameters: This statement has one parameter: `status`.

`status` should be either 0 or 1. If the value is 0 then lookups in this procedure will come from the local database. If the value is 1 then lookups will be made directly from the SQL Server database. Note: In addition to 0 and 1, you may also use "off" or "on", "no" or "yes", or "false" or "true".

Action: This statement allows the database designer to trade off speed vs. up-to-the-minute accuracy in lookups made in a procedure. This option only affects Partner/Server databases that are linked to an SQL server database. For up-to-the minute accuracy lookups should be made directly from the server. However lookups from the server are substantially slower than lookups from the local database. This statement only affects lookups made in this procedure.

Examples: The example forces the status to be looked up from the local database instead of from the server.

```
serverlookup "off"  
Status=lookup("Patients", "SSN", "SSN", "Status", "", 0)  
serverlookup "on"
```

Views: This statement may be used in any view.

See Also: [lookup](#)(function

SERVERPLUGOPTIONS

- Syntax:** SERVERPLUGOPTIONS options
- Description:** The **serverplugoptions** statement allows a procedure to control how Panorama will resolve conflicts when synchronizing between the client and server (after the client database has been modified off-line).
- Parameters:** This statement has one parameter: options.
- options** is one of three possible values.
- client** This means that if a record has been modified by both the client and the server, the clients changes will be kept and the servers changes will be discarded.
- server** This means that if a record has been modified by both the client and the server, the servers changes will be kept and the clients changes will be discarded.
- manual** This means that if a record has been modified by both the client and the server, the user will be presented with a list of the changed record and allowed to "cherry pick" which records to keep.
- Action:** This statement controls how Panorama resolves conflicts between the server and client. This option can also be set manually with the Local Options dialog (in the design sheet Server menu.) (Note: The [info\("plugandrun"\)](#) function allows a procedure to find out the current settings for this database.)
- Examples:** This example tells Panorama to resolve conflicts in favor of the client.
- ```
serverplugoptions "client"
```
- Views:** This statement may be used in the Data Sheet and Form views.
- See Also:** [info\("plugandrun"\)](#) function  
[lockrecord](#) statement  
[lockorstop](#) statement  
[unlockrecord](#) statement

# SERVETIMEOUT

**Syntax:** `SERVETIMEOUT seconds`

**Description:** The `servertimeout` statement sets the maximum time that the local computer will keep a record locked if there is no keyboard or mouse activity.

**Parameters:** This statement has one parameter: `seconds`.

**seconds** is the maximum amount of time that the local computer will keep a record locked if there is no keyboard or mouse activity. If you do not want any timeout (infinite seconds) set this parameter to 0.

**Action:** This statement sets the maximum time that the local computer will keep a record locked if there is no keyboard or mouse activity. This option can also be set manually with the Local Options dialog (in the design sheet Server menu.) (**Note:** The [info\("servertimeout"\)](#) function allows a procedure to find out the current timeout value for this database.)

**Examples:** This example sets the database record locking timeout to 3 minutes.

```
servertimeout 180
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [info\("servertimeout"\)](#) function  
[lockrecord](#) statement  
[lockorstop](#) statement  
[unlockrecord](#) statement

# SERVERUPDATE

**Syntax:** SERVERUPDATE truefalse

**Description:** The `serverupdate` statement allows you to temporarily disable record locking and server updates when using Panorama with an SQL server.

**Parameters:** This statement has one parameter: truefalse.

**truefalse** indicates whether you want to disable or enable record locking and server updates. To disable updates and record locking, use "off", "no", or "false". To enable updates and record locking, use "on", "yes", or "true"

**Action:** When using a client/server database, Panorama normally updates the server with full record locking every time any change is made to the database. If this is unnecessary, you can make your procedure run much faster by temporarily disabling record locking and server updates with the `serverupdate` statement. Important: You should only disable server updates if you don't plan to keep the changes you have made. For example, perhaps you need to use the `formulafill` statement to prepare data for printing, but will no longer need the calculated data after the report is printed.

**Examples:** The example below turns off server updates, then uses the `formulafill` statement to calculate P/E ratios. Since server updates are turned off, the `formulafill` will be very fast. In this case we don't need to keep the calculated P/E ratios after we are finished printing, so it is acceptable to turn off the server update.

```
serverupdate "off"
field Ratio
formulafill Price/Earnings
print dialog
serverupdate "on"
subsetselectall /* restore original data from server */
```

The last line in this example may not be necessary. If you don't use this line, the data in the Ratio field will gradually change back to the server values as you work with the database.

**Views:** This statement may be used in any view.

**See Also:** [attachserver](#) statement  
[detachserver](#) statement  
[formserverlookup](#) statement  
[info\("serverfile"\)](#) statement  
[info\("serverstatus"\)](#) statement  
[lockorstop](#) statement  
[lockrecord](#) statement  
[subsetselect](#) statement  
[subsetselectall](#) statement  
[unlockrecord](#) statement

# SET

**Syntax:** SET destination,formula

**Description:** The **set** statement performs an assignment, much like an equals sign. However, the destination of the assignment can be calculated on the fly.

**Parameters:** This statement has two parameters: **destination** and **formula**.

**destination** is a formula that calculates the name of the field or variable that you want to modify.

**formula** calculates the value that will be placed into the destination.

**Action:** Panorama normally copies data into fields or variables with an assignment. For example, this assignment statement copies the value **Westside** into the field (or variable) named **City**.

```
City="Westside"
```

Assignment statements work fine as long as it is known where the data needs to be copied into when the program is written. But sometimes this is not known, or it needs to change on the fly. The set statement essentially lets the left hand side of the = change on the fly.

**Examples:** The procedure below assumes that the current database contains a field for each day of the week: Sunday, Monday, Tuesday, etc. The example copies the variable **DepartureTime** into the field for the current day (the second line of the procedure calculates the name of the day).

```
set
datepattern(today(), "DayOfWeek"),
DepartureTime
```

Here is the same procedure rewritten without the set statement. This illustrates the power of the set statement, which in this case is doing the same work as 17 statements.

```
local dayName
dayName=datepattern(today(), "DayOfWeek")
case dayName="Sunday"
 Sunday=DepartureTime
case dayName="Monday"
 Monday=DepartureTime
case dayName="Tuesday"
 Tuesday=DepartureTime
case dayName="Wednesday"
 Wednesday=DepartureTime
case dayName="Thursday"
 Thursday=DepartureTime
case dayName="Friday"
 Friday=DepartureTime
case dayName="Saturday"
 Saturday=DepartureTime
endcase
```



**Views:** This statement may be used in any view.

**See Also:** [formulacalc](#) statement  
[define](#) statement

# SETABOUTMENU

**Syntax:** SETABOUTMENU name

**Description:** The `setaboutmenu` statement allows you to customize the first item of the Apple Menu. This item normally displays **About Panorama** or **About Panorama Direct**, but you can customize it to display anything you want.

**Note:** By using the `.CustomAbout` and `.About` procedures you can almost completely hide the fact that your application is created in Panorama. (However, the Application Menu in the upper right hand corner of the screen will always show the name Panorama.)

**Don't forget that you must include the Panorama copyright message in any custom About window that you create with the `.About` procedure!**

**Parameters:** This statement has one parameter: name.

**name** is a formula that specifies what should be displayed in the first item of the Apple Menu. The formula must generate text, for example "My Killer Application" or `info("formname")`.

**Action:** The `setaboutmenu` statement can only be used in a special procedure named **..Custom-About**. This procedure should only have one statement in it, the `setaboutmenu` statement. If the `..CustomAbout` procedure contains any other statements, Panorama may crash (and possibly burn).

**Examples:** This example sets the About menu item to **About Jim's Killer Database**.

```
setaboutmenu "About Jim's Killer Database"
```

This example shows how you can create an About menu that dynamically changes. Depending on the currently active form the Apple menu will display a different name, for example About Calcu-Metric or About Calcu-Time.

```
setaboutmenu "About Calcu-"+info("formname")
```

The `.CustomAbout` procedure only applies to the forms and data sheet in the same database as the procedure. As you click from window to window, the About item in the Apple Menu may change (as in the previous example). When a procedure, flash art, or design sheet window is open the Apple Menu will display the standard **About Panorama...or About Panorama Direct...message**. The standard message will also be displayed when a window from any other database is active (unless that database also has a `..CustomAbout` procedure).

**Views:** Does not apply. This statement must only be used in the `..CustomAbout` procedure.

**See Also:** None

# SETAUTONUMBER

**Syntax:** SETAUTONUMBER *number*

**Description:** The `setautonumber` statement changes the automatically generated number for the next record that will be added to database. This allows you to generate numbers out of sequence, or to start the sequence at a specific value.

**Parameters:** This statement has one parameter: *number*.

**number** is the number that should be assigned to the next record that is created. This must be an integer.

**Action:** Panorama can automatically number new records as they are added to the database. This statement allows a procedure to change what the next number will be. A procedure can find out what the next number will be with the `getautonumber` statement.

**Examples:** The example skips autonumbering ahead by 100.

```
local xNumber
getautonumber xNumber
setautonumber xNumber+100
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [getautonumber](#) statement  
[addrecord](#) statement  
[insertrecord](#) statement

# SETAUTOSAVE

**Syntax:** SETAUTOSAVE *file,time*

**Description:** The `setautosave` statement allows a procedure to turn auto save on or off, and to control the frequency of auto save.

**Parameters:** This statement has two parameters: *file* and *time*.

**file** is the name of the database. This database must be open (in memory).

**time** is the number of minutes between automatic saves. To turn autosave off, set the time to zero (0). (Note: A procedure can find out the current auto-save setting for any database with the `dbinfo("autosave",...)` function.

**Action:** Panorama can automatically save a database every few minutes. This feature is normally controlled from the **Save As** dialog, but can also be controlled by a procedure.

**Examples:** This example turns on auto-save for the current database. The database will be saved every 10 minutes.

```
setautosave info("databasename"),10
```

This example turns off auto-save for the current database.

```
setautosave info("databasename"),0
```

**Views:** This statement may be used in a procedure run from any view.

**See Also:** [save](#) statement  
[dbinfo\(\)](#) function

# SETCANGES

**Syntax:** `SETCANGES count`

**Description:** The `setchanges` statement allows a procedure to arbitrarily change the count of the number of changes made to the current database since the last save.

**Parameters:** This statement has one parameter: `count`.  
`count` is a positive integer number.

**Action:** As you work, Panorama keeps track of how many changes have been made to each database file. It keeps track of this number so that it knows whether or not to ask if you want to save changes when you close the file. If the number of changes is zero, it will not ask and will not save the file (since it is already saved).

This statement allows a procedure to arbitrarily change this count. The primary use for this is to force Panorama to not count any changes the procedure itself makes. (Note: A procedure can find out the current count of changes using the `iinfo("changes")` function.

**Examples:** This example analyzes the data and prints a report. Even though the database is the same before and after the report is printed, Panorama would normally think the database had been changed. The example “fools” Panorama into thinking the database has not been changed by the seven statements in between `info("changes")` and `setchanges`.

```

local WasChanges
WasChanges=info("changes")
field Date
group by month
field Amount
total
outlinelevel 1
print dialog
removesummaries 7
setchanges WasChanges

```

**Views:** This statement may be used in any view.

**See Also:** `info("changes")` function  
`save` statement

# SETFILEFINDERINFO

- Syntax:** `SETFILEFINDERINFO folder,filename,typecreator,position, flags, creationdate, modificationdate`
- Description:** The `setfilefinderinfo` statement modifies a collection of information about a file, including when it was created and last modified and its position within the window.
- Parameters:** This statement has seven parameters: `folder`, `filename`, `typecreator`, `position`, `flags`, `creationdate`, and `modificationdate`.
- folder** is a 6 byte binary data item (a path id) that unambiguously describes the location of the folder where the file should be saved. A path id is a binary data item that unambiguously describes the location of a folder on the hard disk. Path id's are created by the `folder()`, `dbinfo()` and some `info()` functions, and the `openfiledialog` and `savefiledialog` statements. If this parameter is empty text ("") the folder containing the current database is assumed.
- filename** is the name of the file you wish to save. The file name may be up to 31 characters long, and may not contain / characters.
- typecreator** is the new 4 character type code and 4 character creator code for this file. If you don't want to change these codes you can simply specify "".
- position** is the visual x-y position of this file within the folder (see [graphic coordinates](#)). If you don't want to change the position use 0.
- flags** is a number that specifies operating system specific options for this file. If bit 14 of this value is set then the file is invisible. If you don't want to change the flags use 0.
- creationdate** is contains the time and date the file was created, in SuperDate format (see `superdate()`). If you don't want to change this date use 0.
- modificationdate** contains the time and date the file was last modified, in SuperDate format (see `superdate()`). If you don't want to change this date use 0.
- Action:** The `setfilefinderinfo` statement allows a procedure to modify the visible properties of a disk file. It can be used with the `getfilefinderinfo` statement to examine and modify those properties.
- Examples:** This program sets the creation date/time and modification date/time to 9 am today while leaving all of the other file options undisturbed.
- ```
setfilefinderinfo "", "Sunset.jpg", "", 0, 0,
    superdate( today(), time("9am")),
    superdate( today(), time("9am"))
```
- Views:** This statement may be used in any view.
- See Also:** [getfilefinderinfo](#) statement
[filerename](#) statement
[filetypecreator](#) statement
[filesave](#) statement
[filetrash](#) statement
[folder\(\)](#) function

SETMENUMARK

- Syntax:** `SETMENUMARK menu,item,mark`
- Description:** The `setmenumark` statement adds, changes or removes a mark to a menu item. The mark is usually a checkmark (for example ✓Fast), but may be any character.
- Parameters:** This statement has three parameters: `menu`, `item` and `mark`.
- menu** is the name or ID number of the menu that contains the item to be marked. The menu ID is assigned in ResEdit.
- item** is the name of the menu item, or the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu is **Cleared**. This menu item may be specified as either "Cleared" or 3.
- mark** is the character to be used to mark the menu item. To mark the menu item with a checkmark (✓) use the formula `chr(18)`. To mark the item with a diamond (◆) use the formula `chr(19)`. To remove any mark from this menu item use the formula "" (empty text).
- Action:** This statement adds a mark to a menu item, changes the mark associated with a menu item, or removes the mark associated with a menu item. (Note: **Only custom menus can be marked by a procedure**. A procedure cannot add a mark to one of Panorama's standard menus or to an item in the **Action** menu.)
- Examples:** Suppose a database has a **Rush** menu item in the **Order** custom menu. The example below could be part of the `.CustomMenu` procedure, and handles adding and removing a checkmark from the **Rush** menu item (**Rush/Rush**).

```

if info("trigger") = "Menu.Order.Rush"
  getmenumark "Order", "Rush"
  if clipboard() = "
    setmenumark "Order", "Rush", chr(18)
  else
    setmenumark "Order", "Rush", ""
  endif
endif
endif

```

Another common use for menu checkmarks is to check one item from a group. The example below is designed to work with a Shipper custom menu that contains a list of shipping companies. This example is designed to be part of the `.CustomMenu` procedure.

```

global PreferredShipper
local MenuName, MenuItemName
MenuName=info("trigger")[6,"-."][1,-2]
MenuItemName=info("trigger")["-.",-1][2,-1]
if MenuName="Shipper"
  clearmenumarks "Shipper"
  PreferredShipper=MenuItemName
  setmenumark "Shipper",MenuItemName,chr(18)
  stop
endif
endif

```

Views: This statement may be used in any view that has custom menus installed.

See Also: [getmenumark](#) statement
[clearmenumarks](#) statement
[getmenutext](#) statement
[setmenutext](#) statement
[menudisable](#) statement
[menuenable](#) statement
[menubuild](#) statement
[getmenus](#) statement
[setmenus](#) statement

SETMENUS

Syntax: `setmenus menulist`

Description: The `setmenus` statement sets up a new custom menu bar configuration. Using this statement you can add or remove menus from the top of the screen.

Parameters: This statement has one parameter: `menulist`.

menulist is a text item that must contain a list of the menus you want to appear in the menu bar (across the top of the screen). The menu numbers in the list should be separated by one or more spaces. Menu numbers below 128 are standard Panorama menus (see the list of standard menus below). Menus above 128 are custom menus which you create with a resource editing program like ResEdit.

| Menu | Number | Notes |
|--------|--------|------------------------------------|
| Apple | 1 | Apple Menu |
| File | 7 | Use in Data Sheet window |
| File | 27 | Use in Form windows |
| Window | S18 | Arrange (submenu of File menu) |
| Edit | E19 | Use in Data Sheet window |
| Edit | E37 | Use in Form windows |
| Fields | 28 | Normally used in Data Sheet window |
| Text | 73 | Normally used in Data Sheet window |
| Font | S3 | Submenu of Text menu |
| Size | S4 | Submenu of Text Menu |
| Search | 8 | |
| Sort | 9 | |
| Math | 10 | |
| Setup | 68 | Normally used in Data Sheet window |
| Setup | 70 | Normally used in Form window |

Action: This statement allow menus to be added, removed or re-arranged in the menu bar at any time. Only the menus for the current form or data sheet are affected. This statement is the same as using the **Install Custom Menus** command in the Setup Menu.

Examples: The example below will change the configuration of the menu bar. If the `userMode` variable indicates that the user is an expert, the menu will contain the menus: **Apple, File, Edit** and four custom menus. If the user is not an expert there will be only one custom menu.

```
if userMode="Expert"
  setmenus "1 27 S18 E37 3004 3005 3006 3008"
else
  setmenus "1 27 S18 E37 3006"
endif
```

Views: This statement may be used in Data Sheet and Form views.

See Also: [menubuild](#) statement
[getmenus](#) statement
[setmenutext](#) statement
[getmenutext](#) statement
[setmenumark](#) statement
[getmenumark](#) statement
[clearmenumarks](#) statement
[menudisable](#) statement
[menuenable](#) statement

SETMENUTEXT

- Syntax:** `SETMENUTEXT menu,item,text`
- Description:** The `setmenutext` statement changes the text of a menu item. For example you could use this to change **Locked** to **Unlocked**, or **Fast** to **Slow**.
- Parameters:** This statement has three parameters: `menu`, `item` and `text`.
- menu** is the name or ID number of the menu that contains the item to be changed. The menu ID is assigned in ResEdit.
- item** is the name of the menu item, or the number of the menu item within the menu (starting with 1 at the top). For example, suppose the third item in the Books menu is **Post Now**, and you want to change this item to **Post Later**. The item parameter may be either "**Post Now**" or 3.
- text** is the new text for the menu item. This may be up to 40 characters long.
- Action:** This statement renames a single item within a menu. Be careful when you change menu items on the fly. This can create a very confusing user interface. (Note: **Only custom menus can be changed**. You cannot change Panorama's standard menus or the items in the **Action** menu with this statement.)
- Examples:** This example changes the third menu item in the **Books** custom menu.
- ```
if Cash>2500
 setmenutext "Books",3,"Post Now"
else
 setmenutext "Books",3,"Post Later"
endif
```
- This example toggles the fifth menu item in the **Preferences** custom menu between **Fast** and **Slow**.
- ```
getmenutext "Invoice",5
if clipboard() contains "Fast"
    setmenutext "Preferences",5,"Slow"
else
    setmenutext "Preferences",5,"Fast"
endif
```
- Views:** This statement may be used in any view that has custom menus installed.
- See Also:** [getmenutext](#) statement
[setmenumarkk](#) statement
[getmenumark](#) statement
[clearmenumarks](#) statement
[menudisable](#) statement
[menuenable](#) statement
[menubuild](#) statement
[getmenus](#) statement
[setmenus](#) statement

SETPARAMETER

- Syntax:** `SETPARAMETER parameter#,value`
- Description:** The `setparameter` statement is used to transfer data from a subroutine back to the main procedure that called it.
- Parameters:** This statement has two parameters: `parameter#` and `value`.
- parameter#** is a number specifying what parameter you want to modify. All parameters are numbered, starting with 1 (1, 2,3, 4, etc.). A parameter may only be modified if it is a field or variable. If a parameter is a more complex formula, it cannot be modified. For example if the parameter is `Name` or `BookCount` it can be modified. If the parameter is `upper(Name)`, `"Frank"`, or `City+`, `"State` it cannot be modified. If you attempt to modify a parameter that is not a simple field or variable, an error will occur (which can be trapped with `if error`.)
- value** is a formula that calculates the value that will be placed into the parameter. If the parameter is a variable this value must be a text value. If it is a numeric value it will be converted into a text value before it is placed in the variable.
- Action:** When a procedure uses the `call` statement to activate a subroutine, it can pass one or more values to the subroutine. These values are called parameters. If a parameter is in a field or variable, the subroutine can use the `setparameter` statement to modify the value and pass it back to the original procedure.
- Examples:** The procedure below (called `.GetNumber`) assumes that two parameters will be passed to it, and it modifies the second parameter.

```

; Subroutine: .GetNumber
;
; This subroutine expects two parameters:
; (1) prompt text
; (2) new value (a number)
;
; Example:
; call .GetNumber,"How many weeks",WeekCount
;
local temptext
temptext=str( parameter(2))
gettext parameter(1),temptext
setparameter 2,temptext

```

The procedure below uses the `.GetNumber` procedure to ask the user to enter a number. It then uses that number (which is placed in the `addCount` variable, the second parameter to the `call` statement) to determine how many records to add to the database. Notice that the `addCount` value must be converted to a number after it is returned by the subroutine.

```

local addCount
addCount=1
call .GetNumber,"Add how many records?",addCount
addCount=val(AddCount)
loop
  stoploopif addCount=0

```

```
addrecord  
addCount=addCount-1  
while forever
```

Views: This statement may be used in any view.

See Also: [call](#) statement
[farcall](#) statement
[parameter\(\)](#) function

SETPLUGANDRUN

Syntax: SETPLUGANDRUN mode

Description: The `setplugandrun` statement controls how Panorama will resolve conflicts between the client and server when the client database is reconnected to the server after being used off line.

Parameters: This statement has one parameter: mode.

mode specifies the way this database will handle records that have changed on both the server and client when synchronizing with the server database. There are four possible mode:

| Mode | Description |
|----------|--|
| "off" | Don't allow plug-and-run (database may not be modified if not connected to server). |
| "client" | In this mode if a record has been modified by both the client and the server, the client's changes will be kept and the server's changes will be discarded. |
| "server" | In this mode if a record has been modified by both the client and the server, the server's changes will be kept and the client's changes will be discarded. |
| "manual" | In this mode if a record has been modified by both the client and the server, the user will be presented with a list of the changed record and allowed to "cherry pick" which records to keep. |

Action: This statement changes the plug-and-run mode of the current database. You can also change this mode manually using the **Local Setup** dialog (in the Server menu of the Design Sheet window).

Examples: This statement changes the plug-in-run mode so that changes made to the current database will override changes made by other users when synchronizing.

```
setplugandrun "client"
```

Views: This statement may be used in any view.

See Also: [info\("plugandrun"\)](#) function

SETREPORTCOLUMNS

Syntax: SETREPORTCOLUMNS *columns*

Description: The `setreportcolumns` statement allows a procedure to control the number of columns of a report, and the direction (across or down). (Note: These options can be set manually with the **Report Preferences** dialog.)

Parameters: This statement has one parameter: *columns*.

columns is the number of columns in the report. This should be a text value, for example "2", not 2. Use "0" to print as many columns as will fit on the page (the automatic option). The report normally defaults to printing down, but you may follow the number with either down or across, for example "3 across" or "2 down". You may abbreviate down with **d** and across with **a**, for example "3a" or "2d".

Action: This statement changes the report preferences for the current form. The statement can change the number and direction of printed columns.

Examples: The example shows four different options for report columns.

```
setreportcolumns "2 Down"  
setreportcolumns "3 Across"  
setreportcolumns "0" /* defaults to down */  
setreportcolumns "3a" /* a=across */
```

Views: This statement may be used in a form view.

See Also: [print](#) statement
[printonerecord](#) statement
[printonemultiple](#) statement

SETRULERS

Syntax: `SETRULERS units`

Description: The `setrulers` statement changes the measurement units (inches, centimeters, etc.) for the rulers in the current form.

Parameters: This statement has one parameter: `units`.

units is the measurement unit type to select. There are five choices for measurement units:

```
Inches
Centimeters
Pixels
Deca-Pica
Deca-Elite
```

The first three choices may be abbreviated with their first letter: `I`, `C`, or `P`.

Action: This statement changes the measurement units (inches, centimeters, etc.) for the rulers in the current form. This setting is also used for the rulers in any word processing objects in the current form.

Examples: All three of the examples below set the measurement units to centimeters.

```
setrulers "centimeters"
setrulers "C"
setrulers "c"
```

This example sets the units to Deca-Elite.

```
setrulers "Deca-Elite"
```

This example toggles the units between inches and centimeters. Each time the procedure is run it flips to the opposite setting.

```
if info("rulers") beginswith "i"
    setrulers "centimeters"
else
    setrulers "inches"
endif
```

Views: This statement may be used in a form view.

See Also: [info\("rulers"\)](#) function

SETTRIGGER

Syntax: SETTRIGGER value

Description: The `settrigger` statement changes the value returned by the `iinfo("trigger")` function. Usually this value is set by Panorama when it triggers a procedure. The `settrigger` statement allows a procedure to simulate this action.

Parameters: This statement has one parameter: value.

value is the text to be placed into the `info("trigger")` buffer. The value may be up to 40 characters long.

Action: This statement can fake other procedures into thinking that the user has selected menu items, pressed buttons, etc.

Examples: The example below triggers the print section of the `.CustomMenu` procedure. As far as the `.CustomMenu` procedure is concerned, it thinks that the user selected **Print** from the File menu.

```
settrigger "Menu.File.Print"  
call .CustomMenu
```

Views: This statement may be used in any view.

See Also: `info("trigger")` function

SETWINDOW

Syntax: SETWINDOW top,left,height,width,options

Description: The `setwindow` statement specifies the dimensions (size and location) of the next window that is opened with the [openform](#), [opensheet](#), [opencrosstab](#), [openprocedure](#), [opendesignsheet](#), or [opendialog](#) commands

Parameters: This statement has five parameters: `top`, `left`, `height`, `width` and `options`.

top is the position of the top edge of the rectangle. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

left is the position of the left edge of the rectangle. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

height is the height of the rectangle. This must be a number between 0 and +32,767.

width is the width of the rectangle. This must be a number between 0 and +32,767.

options is an item of text that optionally turns off elements of the new window. If the text contains `NoPalette`, the window will not have a tool palette. If the text contains `NoVertScroll`, the window will not have a vertical scroll bar. If the text contains `NoHorzScroll`, the window will not have a horizontal scroll bar. If the text contains `NoDragBar`, the window will not have a drag bar across the top (the window will look like a dialog box). A procedure may combine several options separated by spaces. If the option text is empty ("") the window will appear normal.

Action: When a new window opens it normally fills almost the entire screen. By using the `setwindow` statement before opening the window, the procedure can control where the window appears on the screen. Note: There are two other statements that can perform the same function: [setwindowrectangle](#) and [windowbox](#). Another statement, [fitwindow](#), can be used to make sure that the new window is completely visible on the screen.

Examples: The procedure below opens the form Balance Options as a 3 inch by 4 inch window with no scroll bars.

```
setwindow 72,96,216,288,"noHorzScroll noVertScroll"
openform "Balance Options"
```

Views: This statement may be used in any view.

See Also: [setwindowrectangle](#) statement [windowbox](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[openform](#) statement
[opencrosstab](#) statement
[openprocedure](#) statement
[opendialog](#) statement
[fitwindow](#) statement
[zoomwindow](#) statement

[rectangle\(](#) function
[rectanglesize\(](#) function
[rectanglecenter\(](#) function
[rectangleadjust\(](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("maximumwindow"\)](#) function
[info\("minimumwindow"\)](#) function

SETWINDOWRECTANGLE

- Syntax:** SETWINDOWRECTANGLE rectangle,options
- Description:** The `setwindowrectaangle` statement specifies the dimensions (size and location) of the next window that is opened with the [openform](#), [opensheet](#), [opencrosstab](#), [openprocedure](#), [opendesignsheet](#), or [opendialog](#) commands
- Parameters:** This statement has two parameters: rectangle and options. rectangle defines the size and location of the new window. Most procedures will use the `rectangle`(or `rectanglesize`(functions to create the rectangle.
- options** is an item of text that optionally turns off elements of the new window. If the text contains `NoPalette`, the window will not have a tool palette. If the text contains `NoVertScroll`, the window will not have a vertical scroll bar. If the text contains `NoHorzScroll`, the window will not have a horizontal scroll bar. If the text contains `NoDragBar`, the window will not have a drag bar across the top (the window will look like a dialog box). A procedure may combine several options separated by spaces. If the option text is empty ("") the window will appear normal.
- Action:** When a new window opens it normally fills almost the entire screen. By using the `set-window` statement before opening the window, the procedure can control where the window appears on the screen. Note: There are two other statements that can perform the same function: [setwindow](#) and [windowbox](#). Another statement, [fitwindow](#), can be used to make sure that the new window is completely visible on the screen.
- Examples:** The procedure below opens the form Check List in a new window. The new window is the same size as the current window, but offset 10 pixels down and to the right. The new Check List window will not have a tool palette or horizontal scroll bar, but it will have a vertical scroll bar and a drag bar across the top.
- ```
local newWindowRect
newWindowRect=rectangleadjust(info("windowrectangle"),10,10,10,10)
setwindowrectangle newWindowRect,"nopalette nohorzscroll"
openform "Check List"
```
- The procedure below opens the form Balance Options as a 3 inch by 4 inch dialog box centered on the main screen. Notice that since the form is opened with the `opendialog` statement, the options parameter of the `setwindowrectangle` statement is ignored.
- ```
local newWindowRect
newWindowRect=rectanglecenter(
info("screenrectangle"),
rectanglesize(1,1,3*72,4*72))
setwindowrectangle newWindowRect,""
opendialog "Balance Options"
```
- Views:** This statement may be used in any view.
- See Also:** [setwindow](#) statement [windowbox](#) statement
[opensheet](#) statement
[opendesignsheet](#) statement
[openform](#) statement
[opencrosstab](#) statement

[openprocedure](#) statement
[opendialog](#) statement
[fitwindow](#) statement
[zoomwindow](#) statement
[rectangle](#)(function
[rectanglesize](#)(function
[rectanglecenter](#)(function
[rectangleadjust](#)(function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("maximumwindow"\)](#) function
[info\("minimumwindow"\)](#) function

SHELLOPENDOCUMENT

- Syntax:** SHELLOPENDOCUMENT document
- Description:** The `shellopendocument` statement opens a document in another application. This statement works on Windows only. On Macintosh systems you must use an AppleScript to open a document in another application.
- Parameters:** This statement has one parameter: `document`.
`document` is the name and location of the document to be opened.
- Action:** Use the `shellopendocument` statement when you need to view a document in another application, including HTML pages.
- Examples:** This example opens the Adobe Acrobat document `Manual.pdf` in the folder `My Documents`.
- ```
shellopendocument "C:\My Documents\Manual.pdf"
```
- This example opens the HTML document `Roadshow.html` in the folder `My Test Site`. The HTML document will be opened using Internet Explorer.
- ```
shellopendocument "C:\My Test Site\Roadshow.html"
```
- Views:** This statement may be used in any view (but only on Windows based PC systems).
- See Also:** [fileinfo\(\)](#) function
[fileload\(\)](#) function
[folder\(\)](#) function
[folderpath\(\)](#) function
[import\(\)](#) function
[importcell\(\)](#) function
[info\("files"\)](#) function
[html tables](#)

SHORTCALL

Syntax: SHORTCALL label

Description: The **shortcall** statement allows a procedure to call a sequence of statements within the current procedure as a “mini-subroutine”. If you want to call an entire procedure as a subroutine, use the **call** statement. (Note: Unlike the **call** statement, the **shortcall** statement does not allow parameters to be passed to the subroutine.)

Parameters: This statement has one parameter: label.

label identifies the beginning of the mini-subroutine. (The end of the mini-subroutine is defined by a **rtn** statement or the end of the procedure.) A label is a unique series of letters and numbers that identifies a location within the procedure. The label may not contain any spaces or punctuation except for . and %, and must always end with a colon. The colon is not actually part of the label, it simply identifies the series of letters and numbers as a label instead of a field or variable.

Action: This statement allows a procedure programmer to call a mini-procedure within the current procedure as a subroutine. Subroutines make it easy to use the same sequence of statements at different times within the same procedure. This allows you to write the statements once and use them again and again rather than duplicating the same statements over and over again.

Subroutines normally finish when the end of the procedure is reached. To stop the subroutine before the end of the procedure, use the **rtn** statement. The **rtn** statement makes Panorama return control to the statement just after the **shortcall** statement.

Examples: The example groups and totals two fields in the database. Instead of repeating the statements that group and total, they have been placed in a mini-subroutine. They can be called as many times as needed with the shortcall statement. The arrows on the left show how the procedure will skip back and forth from the main program to the mini-subroutine.

```

field "GL Category"
shortcall GroupTotal
field "Pay To"
shortcall GroupTotal
stop

```

```

GroupTotal:
groupup
field "Debit"
total
rtn

```

This example has only one mini-subroutine, but a procedure may contain as many mini-subroutines as you wish. Each one must begin with a label and end with a **rtn** statement.

Views: This statement may be used in any view.

See Also:

[call](#) statement
[farcall](#) statement
[goto](#) statement
[rtn](#) statement

SHOW

Syntax: SHOW

Description: The **show** statement restores normal screen drawing after the [hide](#) statement has been used.

Note: Panorama 3.1 introduced the [noshow](#) and [endnoshow](#) commands, which we recommend you use instead of [hide](#) and **show**.

Parameters: This statement has no parameters.

Action: To eliminate unnecessary erasing and display of screens, you should use the [hide](#) and **show** statements. The [hide](#) statement tells Panorama not to redraw windows after each procedure statement or command. Redrawing remain off until the end of the procedure, or until the show or [showfields](#) statement turns them back on again. The **show** statement redraws the window immediately and restores normal operation. **Warning: Never switch windows or open new windows while redrawing is turned off!** This may cause crashes, or in rare cases, data corruption.

Examples: The example below would normally erase and re-display the window four times. Adding the [hide](#) and **show** statements suppresses the extra drawing. The window is only erased and re-displayed once at the end of the procedure.

```

hide
field State
groupup
field City
groupup
field Fees
total
outlinelevel "1"
Show

```

Views: This statement may be used in the Data Sheet, Design Sheet, Crosstab or Form Views.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[noundo](#) statement

SHOWCOLUMNS

Syntax: SHOWCOLUMNS fields

Description: The **showcolumns** statement forces Panorama to display specified fields. If a data sheet or view-as-list form is active, the entire column of each field will be updated. This statement should be used when you use the [noshow](#) statement and update a field with a Math Menu statement ([total](#), [formulafill](#), etc.).

Parameters: This statement has one parameter: fields.

fields is a list of fields to be updated. Each field should be separated from the next by a comma. If a field name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: To eliminate unnecessary erasing and display of screens, you can use the [noshow](#) and **showcolumns** statements. The [noshow](#) statement tells Panorama not to redraw windows after each procedure statement or command. The **showcolumns** statement redraws the entire column of the specified fields immediately.

Examples: The example below would normally erase and re-display the window two times. Adding the [noshow](#) and **showcolumns** statements suppresses the extra drawing. The Balance field is erased and re-displayed only once at the end of the procedure.

```
noshow
field Balance
Balance=Credit-Debit
runningtotal
showcolumns Balance
endnoshow
```

Views: This statement may be used in the Data Sheet, Design Sheet, Crosstab or Form Views.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showvariables](#) statement
[showfields](#) statement
[showrecordcounterr](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[noundo](#) statement

SHOWFIELDS

Syntax: SHOWFIELDS fields

Description: The **showfields** statement forces Panorama to display specified fields. When used with the [noshow](#) statement, showfields restores normal screen drawing. When used with the [endnoshow](#) statement, showfields does not restore normal screen drawing.

Parameters: This statement has one parameter: fields.

fields is a list of fields to be updated. Each field should be separated from the next by a comma. If a field name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: To eliminate unnecessary erasing and display of screens, you can use the [hide](#) or [noshow](#) and **showfields** statements. The [hide](#) or [noshow](#) statement tells Panorama not to redraw windows after each procedure statement or command. The **showfields** statement redraws the specified fields immediately. If the [hide](#) command is being used, showfields restores normal display from this point forward. If using the [noshow](#) statement, only the [endnoshow](#) statement turns the display back on. **Warning: When using [hide](#), never switch windows or open new windows while redrawing is turned off!** This may cause crashes, or in rare cases, data corruption.

Examples: The example below would normally erase and re-display the window three times. Adding the [noshow](#) and **showfields** statements suppresses the extra drawing. Even though three fields are modified, only the Balance field is erased and re-displayed once at the end of the procedure.

```
noshow
Date=today()
Time=now()
Balance=Credit-Debit
showfields Balance
endnoshow
```

Views: This statement may be used in the Data Sheet, Design Sheet, Crosstab or Form Views.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[noundo](#) statement

SHOWLINE

- Syntax:** SHOWLINE
- Description:** The `showline` statement forces Panorama to redisplay the current record in all windows in the current database.
- Parameters:** This statement has no parameters.
- Action:** This statement tells Panorama to redisplay all of the data in the current record in every open window of the current database. Use this command after the [noshow](#) command when you have only modified the current record. ShowLine does not turn the display back on. To do that you must use the [endnoshow](#) statement.
- Examples:** Here is an example that clears the current record in the database, but doesn't display anything until it is completely finished.

```

noshow
field (array( dbinfo("fields",""),1,1)) /* goto first field */
loop
  clearcell
  right
until stopped
showline
endnoshow

```

- Views:** This statement should only be used when a form or data sheet is active.
- See Also:** [noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[nundo](#) statement

SHOWOTHER

Syntax: SHOWOTHER field,code

Description: The **showother** statement forces Panorama to update some data on the screen. The showother statement allows you to access Panorama's internal display routines. However, you should avoid using this routine when one of the other show statements can be used instead ([showpage](#), [showline](#), [showvariables](#), [showcolumns](#), [showvariables](#), or [showrecordcounter](#)).

Parameters: This statement has two parameters: **field** and **code**.

field is the name of the field to update. If you want all fields to be updated, use «».

code is a number that specifies what information should be displayed. The available codes are:

```
0 Current Cell (use ShowFields instead)
1 Entire page (use ShowPage instead)
2 cursor moved, data sheet must be updated
3 cursor moved, data sheet already updated
4 update after insertline
5 move cursor up/down
6 new line with cursor move
97 record count (use ShowRecordCounter instead)
98 column header change (changed field name)
99 database redesign (insert field, etc)
```

Action: This statement forces Panorama to re-display some data. All windows in the current database will be affected.

Examples: This example renames all of the fields in the database, using the data in the current record as the new field names. The new field names are not displayed until the end of the procedure.

```
noshow
field array( dbinfo("fields",""),1,¶)
loop
  fieldname «»
  right
until stopped
showother «»,98 /* display all new field names */
endnoshow
```

Views: This statement may be used in any view, but it usually only make sense in the form or data sheet views.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showline](#) statement
[showpage](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement

showrecordcounter statement

hide statement

show statement

noundo statement

SHOWPAGE

Syntax: SHOWPAGE

Description: The **showpage** statement forces Panorama to redisplay all windows in the current database.

Parameters: This statement has no parameters.

Action: This statement tells Panorama to redisplay all of the data in every open window of the current database. ShowPage does not turn the display back on. To do that you must use the [endnoshow](#) statement.

Examples: Here is an example that performs several operations on the current database, but only updates the display once.

```
noshow  
field Date  
groupup by month  
field Category  
groupup  
field Amount  
total  
outlinelevel 2  
showpage  
showfields
```

Views: This statement should only be used when a form or data sheet is active.

See Also: [noshow](#) statement
[endnoshow](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[nundo](#) statement

SHOWRECORDCOUNTER

Syntax: SHOWRECORDCOUNTER

Description: The `showrecordcounter` statement forces Panorama to redisplay the record counter in all windows in the current database.

Parameters: This statement has no parameters.

Action: This statement tells Panorama to redisplay the record counter in every open window of the current database. Use this command after the [`noshow`](#) command when you have added or deleted records.

Examples: Here is an example that adds three new records to the database but only updates the display once.

```
noshow  
addrecord  
addrecord  
addrecord  
showpage  
showrecordcounter  
endnoshow
```

Views: This statement should only be used when a form or data sheet is active.

See Also: [`noshow`](#) statement
[`endnoshow`](#) statement
[`showline`](#) statement
[`showpage`](#) statement
[`showfields`](#) statement
[`showvariables`](#) statement
[`showcolumns`](#) statement
[`showother`](#) statement
[`hide`](#) statement
[`show`](#) statement
[`nundo`](#) statement

SHOWVARIABLES

Syntax: SHOWVARIABLES variables

Description: The **showvariables** statement forces Panorama to update the display of one or more variables on a form

Parameters: This statement has one parameter: **variables**.

variables is a list of variables to be updated. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: When a variable is changed by a procedure, any display of that variable on a form is not normally updated to show the new value. (Continuously updating all variables would make Panorama procedures much slower.) However, sometimes you do need Panorama to update the display of a variable, so that is what the **showvariables** statement does. Every graphic object that displays the variable will be updated, including auto-wrap text, Text Display SuperObjects, and Flash Art. (Note: The **showvariables** statement will cause the variable to update in all windows of the current database. It will not affect windows of other database files, even if they happen to display one or more of the specified variables.

Examples: The example counts and displays the number of times the word **um** appears in a field named **Transcript**. If there is a form open that displays the variable **umCount**, the number will update to show the new value when the procedure is finished. However if a form displays the variable **wordCount**, that number will not update.

```
global umCount,wordCount
umCount=0
wordCount=1
loop
  if array(Transcript,wordCount," ") contains "um"
    umCount=umCount+1
  endif
wordCount=wordCount+1
while wordCount<arraysize(Transcript," ")
showvariables umCount
```

The example is a variation on the previous example. It also counts and displays the number of times the word **um** appears in a field named **Transcript**. This procedure will update the counts continuously as the procedure loops around and around, letting the user watch the progress of both the **umCount** and **wordCount** variables. The disadvantage is that this procedure will take longer to finish.

```
global umCount,wordCount
umCount=0
wordCount=1
loop
if array(Transcript,wordCount," ") contains "um"
  umCount=umCount+1
endif
showvariables umCount,wordCount
wordCount=wordCount+1
while wordCount<arraysize(Transcript," ")
```

Views: This statement may be used in a Form view.

See Also: [global](#) statement
[local](#) statement
[noshow](#) statement
[endnoshow](#) statement
[showline](#) statement
[showpage](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[noundo](#) statement

SIMULATEDIRECT

Syntax: SIMULATEDIRECT

Description: The `simulatedirect` statement temporarily downgrades a full version of Panorama and makes it behave as if it was Panorama Direct. This is sometimes useful for testing a database that is going to be used with Panorama Direct.

Parameters: This statement has no parameters.

Examples: This procedure tells Panorama to simulate Panorama Direct. If you are already using a copy of Panorama Direct or Panorama Engine this statement will be ignored.

```
simulatedirect
```

This procedure restores full Panorama operation (assuming that you started with a full copy of Panorama).

```
endsimulate
```

Views: This statement may be used in any view.

See Also: [simulateengine](#) statement
[endsimulate](#) statement
[info\("serialnumber"\)](#) function

SIMULATEENGINE

Syntax: SIMULATEENGINE

Description: The **simulateengine** statement temporarily downgrades a full version of Panorama and makes it behave as if it was Panorama Engine (the free trial version of Panorama). This is sometimes useful for testing a database that is going to be used with the free Panorama Engine.

Parameters: This statement has no parameters.

Examples: This procedure tells Panorama to simulate Panorama Engine. If you are already using a copy of Panorama Direct or Panorama Engine this statement will be ignored.

```
simulateengine
```

This procedure restores full Panorama operation (assuming that you started with a full copy of Panorama).

```
endsimulate
```

Views: This statement may be used in any view.

See Also: [simulatedirect](#) statement
[endsimulate](#) statement
[info\("serialnumber"\)](#) function

SIN(...)

Syntax: SIN(angle)

Description: The `sin()` function calculates the sine of an angle.

Parameters: This function has one parameter: angle.

angle is a numeric value, an angle. The angle is usually specified in a mathematical unit of measurement called radians, however, within a procedure you can temporarily force Panorama to use degrees (see below). One radian is equal to approximately 57.2958 degrees (the exact value is $180/\pi$).

Result: The result of this function is a numeric floating point value between -1 and 1.

Examples: The graph below shows the result of the sine function given input values from 0 to +20 radians.



This formula calculates the sine of an angle in degrees.

```
sin(Angle*180/π)
```

In this example the angle is in a field or variable named Angle, however, you may use any formula that produces a numeric result in this location. The pi symbol (π) is produced by pressing OPTION-P. Here is another way to calculate angles in degrees in a procedure.

```
degree
NewAngle=sin(Angle)
```

The `degree` statement tells Panorama to use degrees instead of radians in all trigonometry calculations. Panorama will continue to use degrees until the end of the procedure, or until a `radian` statement is encountered.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `sin("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating the sine, for example `sin(val("34"))`.

See Also:

[cos\(\)](#) function
[tan\(\)](#) function
[degree](#) statement
[radian](#) statement
[arcsin\(\)](#) function
[arccos\(\)](#) function
[arctan\(\)](#) function
[val\(\)](#) function

SINH(...)

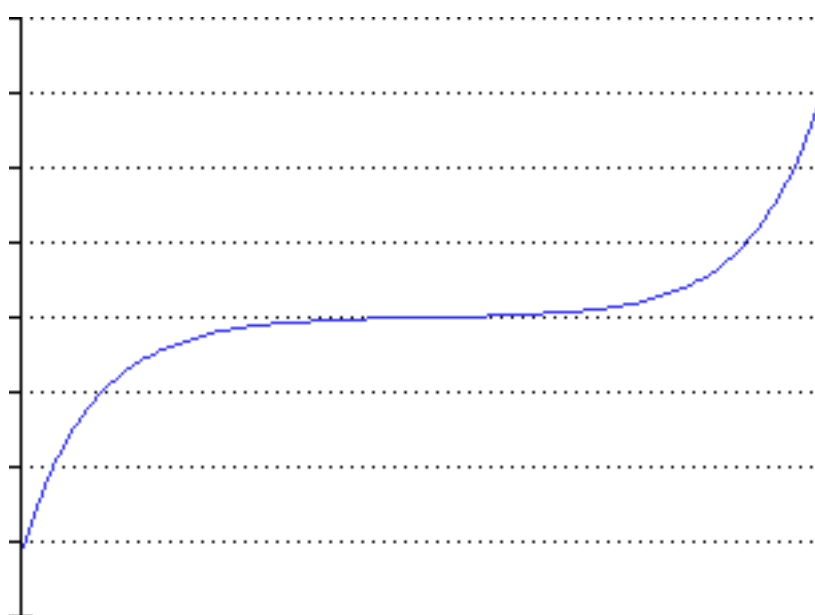
Syntax: `SINH(value)`

Description: The `sinh()` function calculates the hyperbolic sine of a numeric value.

Parameters: This function has one parameter: `value`.
`value` is a numeric value.

Result: The result of this function is a numeric floating point value.

Examples: The graph below shows the result of the hyperbolic sine function given input values from -6 to +6.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `sinh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the hyperbolic sine, for example `sinh(val("34"))`.

See Also: [cosh\(\)](#) function
[tanh\(\)](#) function
[arcsinh\(\)](#) function
[arccosh\(\)](#) function
[arctanh\(\)](#) function
[val\(\)](#) function

SIZE

- Syntax:** `SIZE size`
- Description:** The `size` statement specifies the font's point size for the current Data Sheet, Design Sheet, or Cross Tab window.
- Parameters:** This statement has one parameter: `size`
- `size` assigns the point size of the font you've chosen for the active Data Sheet, Design Sheet, or Cross Tab window. This parameter may be a literal value, field, variable, or formula which results in an integer value for the point size required. System loaded print fonts will show available point sizes in outline style on Panorama's **Size** submenu (**Text** menu), non-outline point sizes will need to be calculated at print time. However, any integer point size may be specified for display purposes.
- Action:** This statement will immediately set the font's point size for the active window to the one specified provided the window is a Data Sheet, Design Sheet, or Cross Tab window. All other views will ignore the size statement.
- You may also change the font for the window using the `font` statement or change the style using the `style` statement.
- This statement has the same effect as using the **Size** submenu from the **Text** menu.
- Examples:** This example changes the point size for the Data Sheet to 10.
- ```
opensheet
size 10
```
- This example allows you to specify the point size you wish to set provided you are in the proper window.
- ```
getscrap "Enter point size: "
size clipboard()
```
- This example opens a cross tab window called Budget, changes the font to palatino, changes the point size to 14, prints the window and then returns to the original form.
- ```
local form
form = info("formname")
gocrosstab "Budget"
calccrosstab
font "palatino"
size 14
print ""
goform form
```
- Views:** This statement may be used in the Data Sheet, Design Sheet, or Cross Tab views **only**.



**See Also:** [fieldstyle\(\)](#) function  
[font](#) statement  
[style](#) statement

# SIZEOF(...)

**Syntax:** `SIZEOF(fieldvariablename)`

**Description:** The `sizeof()` function calculates the amount of memory used by a field cell or a variable.

**Parameters:** This function has one parameter: `fieldvariablename`.  
**fieldvariablename** is the name of the field or variable that you want to calculate the size of.

**Result:** The function returns the number of bytes of memory used by the variable or field cell.

**Examples:** The `sizeof()` function can be used to decide if a numeric or date field is empty or not. The example below selects all the records with no price (not the same as records with a price of zero).

```
select sizeof(Price)=0
```

Another use for the `sizeof()` function is to check if a variable is taking up too much scratch memory. This example checks to see if the variable `importLetter` is more than 500 bytes long. If it is, the procedure clears the variable.

```
if sizeof(importLetter)>500
 importLetter=""
endif
```

**Errors:** **Field or variable does not exist.** This error occurs if there is no variable or field in the current database with the name you have specified. You probably misspelled the field or variable name.

**See Also:** [length\(\)](#) function

# SORTBYCOLOR

**Syntax:** SORTBYCOLOR

**Description:** The `sortbycolor` statement sorts the database by the color of cells in the current field. This is useful if you have assigned a meaning to each color. The sort order is: black, red, green, blue, cyan, magenta, yellow. To assign colors to individual cells use the [style](#) statement.

**Parameters:** This statement has no parameters.

**Examples:** This example assumes that the City field is color coded. Black cities will move to the top, with yellow cities at the bottom.

```
field City
sortbycolor
```

**Views:** This statement may be used in the Data Sheet or Form views.

**See Also:** [groupbycolor](#) statement  
[sortup](#) statement  
[sortupwithin](#) statement  
[sortdown](#) statement  
[sortdownwithin](#) statement  
[groupup](#) statement

# SORTDOWN

**Syntax:** SORTDOWN

**Description:** The `sortdown` statement sorts the database by the current field. The database is sorted in descending order (high to low).

**Parameters:** This statement has no parameters.

**Examples:** This example sorts checks from largest to smallest. To sort in normal order use [sortup](#). To sort additional fields use [sortdownwithin](#). To sort and divide the database into groups with summary records use [groupup](#).

```
field Debit
sortdown
```

**Views:** This statement may be used in the Data Sheet or Form views.

**See Also:** [sortup](#) statement  
[sortupwithin](#) statement  
[sortdownwithin](#) statement  
[sortbycolor](#) statement  
[groupup](#) statement  
[groupbycolorr](#) statement

# SORTDOWNWITHIN

**Syntax:** SORTDOWNWITHIN

**Description:** The **sortdownwithin** statement sorts the database from high to low by the current field. However, it leaves any previous sort intact. For example, suppose a procedure sorts by state, then uses **sortdownwithin** on the City field. The states will remain in order, but within each state the cities will now be sorted from Z to A. This process can be repeated as many times as necessary.

**Parameters:** This statement has no parameters.

**Examples:** This example sorts checks by category, then sorts by check amount from hi to low within each category.

```

field Category
sortup
field Debit
sortdownwithin

```

The **sortdownwithin** can be used over and over again to sort 3, 4, or more fields with each other. Always start with the regular [sortup](#) or [sortdown](#) statement followed by as many **sortdownwithin** statements as required for additional fields. Since Panorama uses a “stable” sort algorithm there is another way to sort multiple fields. Instead of using the **sortupwithin** statement, a procedure can use the regular sort statement, but sort the fields in reverse order. Like the previous example, this example sorts by check amount within categories.

```

field Debit
sortdown
field Category
sortup

```

**Views:** This statement may be used in the Data Sheet or Form views.

**See Also:** [sortup](#) statement  
[sortupwithin](#) statement  
[sortdown](#) statement  
[sortbycolor](#) statement  
[groupup](#) statement  
[groupbycolorr](#) statement

# SORTUP

**Syntax:** SORTUP

**Description:** The **sortup** statement sorts the database by the current field. The database is sorted in ascending order (low to high).

**Parameters:** This statement has no parameters.

**Examples:** This example sorts a database in zip code order from low to high. To sort in reverse order use [sortdown](#). To sort additional fields use [sortupwithin](#). To sort and divide the database into groups with summary records use [groupup](#).

```
field Zip
sortup
```

**Views:** This statement may be used in the Data Sheet or Form views.

**See Also:** [sortdown](#) statement  
[sortupwithin](#) statement  
[sortdownwithin](#) statement  
[sortbycolor](#) statement  
[groupup](#) statement  
[groupbycolorr](#) statement

# SORTUPWITHIN

**Syntax:** SORTUPWITHIN

**Description:** The **sortupwithin** statement sorts the database from low to high by the current field. However, it leaves any previous sort intact. For example, suppose a procedure sorts by state, then uses **sortupwithin** on the City field. The states will remain in order, but within each state the cities will now be sorted. This process can be repeated as many times as necessary.

**Parameters:** This statement has no parameters.

**Examples:** This example sorts a database by both city and state at the same time.

```
field State
sortup
field City
sortupwithin
```

The **sortupwithin** can be used over and over again to sort 3, 4, or more fields with each other. Always start with the regular sortup or sortdown statement followed by as many **sortupwithin** statements as required for additional fields. Since Panorama uses a “stable” sort algorithm there is another way to sort multiple fields. Instead of using the **sortupwithin** statement, a procedure can use the regular sort statement, but sort the fields in reverse order. Like the previous example, this example sorts by cities within states.

```
field City
sortup
field State
sortup
```

**Views:** This statement may be used in the Data Sheet or Form views.

**See Also:** [sortdown](#) statement  
[sortup](#) statement  
[sortdownwithin](#) statement  
[sortbycolor](#) statement  
[groupup](#) statement  
[groupbycolorr](#) statement

# SOUND

**Syntax:** SOUND file sound

**Description:** The **sound** statement plays a digitally recorded sound stored in a resource file. Sounds can be recorded using **Sound** control panel.

**Parameters:** This statement has two parameters: file and sound.

**file** is the name of the resource file that contains the sound. If the resource file is not in the same folder as the current database, you must specify the entire path name in addition to the file name, for example: "[Disk:Sounds:Star Trek](#)".

**sound** is the name of the resource that contains the sound.

**Action:** This statement opens a resource file, plays a sound in that file, then closes the resource file. If a procedure needs to play several sounds in a row it is faster to explicitly open the resource file with the [opensound](#) statement, then use [playsound](#) to play the sound.

**Examples:** This procedure complains audibly if you attempt to sell an item for less than its cost.

```
if Price<Cost
 sound "Greetings" "Breaking Glass"
endif
```

**Views:** This statement may be used in any view, and also works when no windows are open at all.

**See Also:** [opensound](#) statement  
[playsound](#) statement  
[closesound](#) statement  
[resources\(\)](#) function



# SPEEDCOPY

**Syntax:** SPEEDCOPY start,end,from

**Description:** The `speedcopy` statement can copy several fields from one database to another very quickly, or between different records in the same database. The fields in both database files must be in the same order and be of the same type. **WARNING: This statement is for expert users only!** There is very little error checking, and this statement can quickly turn your database into swiss cheese.

**Parameters:** This statement has three parameters: `start`, `end` and `from`.

**start** is the first field to be replaced in the current database. The field name must be surrounded with quotes, for example: "City", not City.

**end** is the last field to be replaced in the current database. This field must be to the right of the start field in the data sheet. The field name must be surrounded with quotes, for example: "Zip", not Zip.

**from** is the field in the second database that corresponds to the start field in the current database. The field name must be surrounded with quotes, for example: "City", not City.

**Action:** This statement is designed to be used in combination with a `lookup()` function. First you lookup, then you copy. The `speedcopy` will copy additional data from the record that was located by the `lookup()` function. The `speedcopy` statement is much faster than multiple lookups.

**Examples:** Assume that you have two databases named `Customers` and `Organizer`, and both have fields name `Address`, `City`, `State` and `Zip`. The example below will quickly look up a customer and copy their address into the `Organizer` database.

```
Address = lookup("Customers",Company,Company,Address,"",0)
if Address=""
 speedcopy "City","Zip","City"
endif
```

Here is the same procedure but written without using the `speedcopy` statement.

```
Address = lookup("Customers",Company,Company,Address,"",0)
City= lookup("Customers",Company,Company,Address,"",0)
State= lookup("Customers",Company,Company,Address,"",0)
Zip= lookup("Customers",Company,Company,Address,"",0)
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [lookup\(\)](#) function  
[lookuplast\(\)](#) function  
[lookupselected\(\)](#) function  
[table\(\)](#) function

# SPELLING

- Syntax:** `SPELLING text,position,wordlength`
- Description:** The **spelling** statement checks the spelling of a section of text. If it finds a spelling error it identifies the location of the error. This statement requires that the optional Panorama dictionary be installed.
- Parameters:** This statement has three parameters: **text**, **position** and **wordlength**.
- text** is a formula that creates the text the procedure wants to check. This may be a field, a variable or a more complex formula.
- position** is a variable that identifies a location within the text. Unlike most other parts of Panorama, the spelling statement considers the first character to be number 0, the second to be number 1, etc. See the Action: section for details.
- wordlength** is a variable that identifies the length of a word that starts at **position**. See the Action: section for details.
- Action:** This statement checks a section of text for spelling errors. If it finds an error, it returns the position of the incorrect word in the **position** parameter and the length of the incorrect word in the **wordlength** parameter. If it does not find any errors, the **wordlength** parameter will be zero.
- The **spelling** statement does not start with the first word of the text. Instead, it computes the starting point from the **position** and **wordlength** parameters. The starting point is **position+wordlength**. This allows the spelling statement to continue scanning after it has found an error. To start at the beginning set both the **position** and **wordlength** parameters to zero.
- Examples:** This procedure makes a list of all the misspelled words in the Transcript field. It ignores words that contain non-alphabetic characters.
- ```

global misSpellings
local position,size,badWord
position=0 size=0
misSpellings=""
loop
    spelling Transcript,position,size
    stoploopif size=0
    badWord=Letter[position+1;size]
    if badWord=striptoalpha(badWord)
        misSpelled=sandwich(" ",misSpellings,",")+badWord
    endif
while forever

```
- Views:** This statement may be used in any view
- See Also:** [wordlist](#) statement

SPLITLINES

Syntax: `SPLITLINES raw,pattern1,line1,...,patternN,lineN`

Description: The `splitlines` statement takes a big chunk of text and splits it into two or more smaller chunks. The statement works by starting at the top and splitting off one or more lines at a time (hence the name `splitlines`).

Parameters: This statement has one fixed parameter: `raw`. It also has two repeating parameters: `pattern` and `line`.

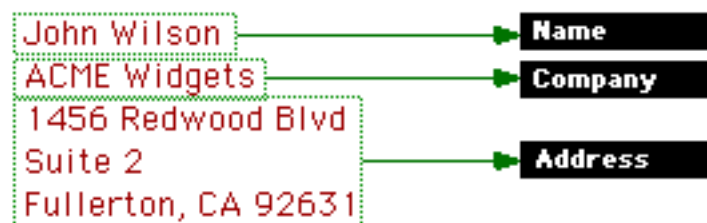
raw is the big chunk of text that you want to split into two or more smaller chunks.

pattern is a template for taking the next slice out of the `raw` chunk of text. The pattern may be one or two characters long. The first character tells Panorama how many lines long the next chunk should be (from 1 to 9 lines). You can also specify 0 lines, which tells Panorama to include all the remaining `raw` text. The second character, which is optional, specifies whether or not Panorama should force the text to upper case as it splits it. If this character is "U" the line will be converted to all upper case. If this character is "W" the first character of each word in each line will be converted to upper case. However, in either case this upper case conversion will happen only if the `raw` text being split off is entirely lower case to begin with. If the second character is not U or W the text will be left as is.

line is the name of a field or variable where this slice of the `raw` text should be placed.

Action: This statement can be used to split up data that is organized line by line.

Examples: Suppose you had an address like the one shown below and you wanted to split it up into individual fields.



The example below will quickly split the data into the three fields. (Note: The procedure could split these fields up further with the [getname](#) and [getaddress](#) statements.)

```
splitlines Combined,"1W",Name,"1W",Company,"0",Address
```

Views: This statement may be used in any view.

See Also: [getname](#) statement
[getcitystatezip](#) statement
[getaddress](#) statement
[getphone](#) statement
[array\(\)](#) function
[arrayrange\(\)](#) function
[extract\(\)](#) function

SQLCOMMAND

Syntax: SQLCOMMAND command

Description: The `sqlcommand` statement sends one or more SQL commands directly to the SQL server. This command is for SQL propeller heads only (you know who you are)!

Parameters: This statement has one parameter: `command`.
`command` is one or more SQL commands.

Action: This statement allows you to program directly in SQL, bypassing all of Panorama's automatic Partner/Server features. You can use this command to access SQL databases that are not part of the Partner/Server system. If your SQL command generates any data, make sure that you read all of the data with the `sqlread` command. Any leftover data will confuse the logic of Panorama's Partner/Server system and possibly cause lost data or data corruption.

The `SQLCommand` and `sqlread` statements must be used in a Partner/Server database. You may want to set up a "dummy" Partner/Server database just for this purpose. Since the "dummy" database doesn't contain any data that you will use, it can simply contain one field and one record. Opening the "dummy" database opens the connection to the server, allowing SQL commands to be sent to the server.

Examples: This example "prints" a list of field names and types for the SALES table in the FY96 database. The "printed" data must be read by the `sqlread` command.

```
sqlcommand "describe columns of FY96.SALES;" +  
"for each print -> name,type;"
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [sqlread](#) statement

SQLCONNECTION

Syntax: SQLCONNECTION

Description: The `sqlconnection` statement opens the Server Connection dialog. This is only necessary if the SQL server file linked to this Panorama database has been moved to a different server computer.

Parameters: This statement has no parameters.

Action: This statement opens the Server Connection dialog. This dialog is normally opened by the Server Connection command in the design sheet, however you may use this statement in the data sheet or any form. The only time this statement is needed is if you move the server to a different machine or change the network connection to the server.

Examples: This example simply opens the Server Connection dialog. This example might be part of a larger procedure, perhaps the `.CustomMenu` procedure.

```
if info("trigger") contains "Server Connection"
    sqlconnection
    save
endif
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [attachserver](#) statement
[serverfile](#) statement
[info\("serverstatus"\)](#) function

SQLREAD

Syntax: `SQLREAD buffer,separator,count`

Description: The `sqlread` statement directly reads one or more items of data from the SQL server. This command is designed to be used with the `sqlcommand` statement, and is for SQL propeller heads only (you know who you are)!

Parameters: This statement has three parameters: `buffer`, `separator` and `count`.

buffer is a field or variable that will be used to hold the data from the SQL server. Only text fields can be used as buffers, not numeric or date fields. If you use a variable, keep in mind the amount of data you plan to transfer from the server. If you plan to transfer large amounts of data in one chunk, you may need to increase Panorama's scratch memory allocation. This can be done by holding down the SHIFT key while selecting Memory Usage from the Apple menu, or in a procedure with the `scratchmemory` statement.

separator is the separator character that will be used in the buffer between SQL data items. Typical separators include carriage returns (¶), tabs (→), commas, semicolons and slashes.

count is the number of SQL data items you want to read into the buffer. The SQL server sends data to the server item by item. For example, suppose you have a database with fields Name, Address, City, State and Zip and you tell the server to print a record. The server will send 5 items to the client. You have the choice of reading all five of these items at once, or of reading them one at a time.

If you don't know how many data items to expect, you can use 0 as the count. In this case Panorama will transfer all of the data sent by the SQL server into the buffer. Be careful, however, that there is enough memory to fit all the data in the buffer. You can use the `arraysize()` function to find out how many data items were actually transferred.

Action: This statement allows you to program directly in SQL, bypassing all of Panorama's automatic Partner/Server features. You can use this command to access SQL databases that are not part of the Partner/Server system. To generate data, use the `sqlcommand` statement with SQL `print` or `printall` statements. Make sure that you read all of the data you generate with the `sqlread` command. Any leftover data will confuse the logic of Panorama's Partner/Server system and possibly cause lost data or data corruption.

The `sqlcommand` and `sqlread` statements must be used in a Partner/Server database. You may want to set up a "dummy" Partner/Server database just for this purpose. Since the "dummy" database doesn't contain any data that you will use, it can simply contain one field and one record. Opening the "dummy" database opens the connection to the server, allowing SQL commands to be sent to the server.

Examples: This example displays a list of all the SQL databases on the server, with each file name separated by a comma.

```
local dbList
sqlcommand "describe databases;printall;"
sqlread dbList,"",0
message dbList
```

This procedure returns a carriage return separated array of tables in a databases.

```
/* Parameters: (1) Name of database, (2) list of tables */  
local db,tblList,qc  
db=parameter(1)  
qc=  
  "open database "+{"}+db+{"}+";"+  
  "describe tables of "+{"}+db+{"}+";"+  
  "for each print -> name;"+  
  "close database "+{"}+db+{"}+";"  
clipboard=qc  
sqlcommand qc  
sqlread tblList,¶,0  
setparameter 2,tblList
```

Views: This statement may be used in the Data Sheet and Form views.

See Also: [sqlcommand](#) statement

SQR(...)

Syntax: SQR(

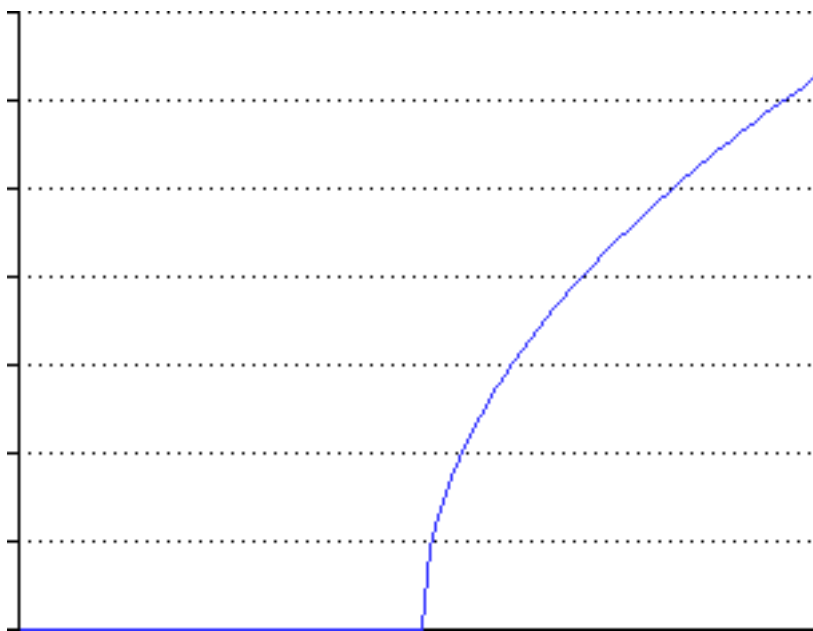
Description: The `sqr(` function computes the square root of a value.

Parameters: This function has one parameter: value.

value is a numeric value. Only positive numbers can be used as the value.

Result: The result of this function is a numeric floating point value.

Examples: This function calculates the square root of a value. The graph below shows the result of the square root function given input values from -10 to +10.



If you want to calculate the square root of a negative value you must convert it to a positive value first using the `abs(` function.

```
sqr(abs(myNegativeValue))
```

(Of course the square root of a negative number is really a complex number, but we'll leave that to the mathematics majors!)

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `sqr("23")`. If you have a numeric value in a text item you must convert the text to the number data type before calculating a square root, for example `sqr(val("34"))`.

Floating point error. The common logarithm of values 0 or less (negative numbers) is undefined. If you attempt to calculate the common logarithm of such a number, a floating point error will occur.

See Also: [val\(](#) function
[abs\(](#) function

STARTSCRIPT

Syntax: `STARTSCRIPT script`

Description: The `startscript` statement launches an AppleScript script.

Parameters: This statement has one parameter: `script`.

script is the name of the script that you want to launch. This script should be a file in the same folder as the current database. If the script is in a different folder you must supply the complete path to the script, for example "`Disk:My Scripts:Check E-Mail`".

Action: Starting with System 7.1 the Macintosh has a built in programming language called AppleScript. The `startscript` statement allows a Panorama procedure to use AppleScript's to do work that cannot be done by Panorama alone. Panorama itself is scriptable, so the AppleScript you trigger may change fields or variables within Panorama, or even call another Panorama procedure.

Examples: The example below is from a database of e-mail messages. When the procedure is started it finds the first message that has not been sent yet. It then starts an AppleScript called Send E-Mail, which is supposed to grab the address and body from the letter and send the e-mail, then mark the Status field to indicate the letter is sent. At this point Panorama loops back to the top and looks for another unsent message.

```
loop
  find Status="Unsent"
  stoploopif notfound
  startscript "Send E-Mail"
while forever
save
```

Views: This statement may be used in any view

See Also: [call](#) statement
[farcall](#) statement

STATE(...)

Syntax: STATE(zip)

Description: The `state()` function uses Panorama's optional zip code dictionary to look up the name of a state associated with a zip code.

Parameters: This function has one parameter: `zip`.

zip is a US 5 digit zip code. You can specify the zip code either as a number or as text.

Result: The function returns the name of the state for the zip code. If the optional zip code dictionary is not installed, the function will return -- instead of a state name.

Examples: One primary use for the `state()` function is to enter the state automatically when the zip code is entered, saving keystrokes and reducing the probability of data entry errors. This example assumes that the database has a `ZipCode` field that contains text. The example uses a text funnel to strip off any extra characters in the zip code (for example 9 digit zip codes).

```
City=city(ZipCode[1,5])
State=state(ZipCode[1,5])
```

Another use for the `state()` function is to double-check data entry. This example locates all records where the zip code does not match the state name.

```
select State≠state(ZipCode[1,5])
```

Errors: This function does not generate any error message. However, if the zip code dictionary is not installed the function will always return -- instead of a state name.

See Also: [city\(\)](#) function
[county\(\)](#) function

STATEMENTS

Introduction: A statement is a single step in a procedure. Most statements start with a special word (sometimes called a keyword) that identifies this statement, for example sortup, select, print, or open. Panorama 3 understands over 300 different keywords.

A statement may consist of a keyword all by itself, but many statements require additional options. These additional options are called parameters. If a keyword uses parameters they must immediately follow the keyword in the program.

Statements are not functions! A statement can only be used as a step in a procedure—not as an element within a formula. Since a procedure may use functions within a formula, statements and functions are sometimes confused.

(Note to experienced programmers: Unlike many programming languages, Panorama's keywords are not reserved words. This means, for example, that a database field or variable may be called print or open, even though these are keywords. However, using field or variable names this way can easily result in programs that are very confusing to read, so we recommend that you avoid keywords when you are defining fields and variables.

See Also: [functions](#)

STATUSMESSAGE

Syntax: STATUSMESSAGE message

Description: The `statusmessage` statement displays a message in the status bar at the bottom of the procedure window. If the procedure window is not open this statement is ignored.

Parameters: This statement has one parameter message.

`message` is the text that is to be displayed. You may use any formula to create the text.

Action: Use the `statusmessage` statement to display information in the status bar at the bottom of the procedure window. Panorama normally displays the result of each assignment in this status bar while single stepping, the `statusmessage` statement allows you to write anything you want.

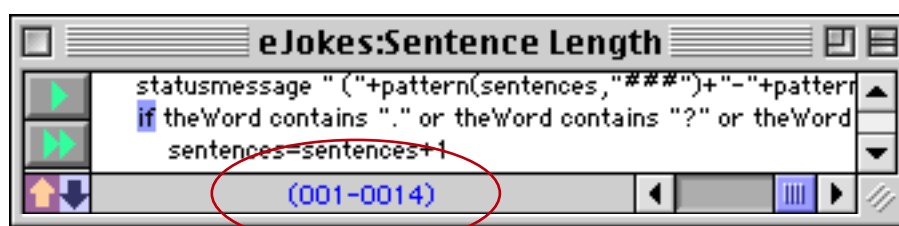
Examples: This example uses the `statusmessage` statement to help monitor the progress of a loop.

```

local theWord,n,words,sentences
n=1 words=0 sentences=0
loop
  theWord=array( replace( replace(Joke,¶," "), " ", " "),n," ")
  stoploopif theWord=""
  theWord=strip(theWord)
  words=words+1
  statusmessage " (" +pattern(sentences,"###")+
    "-"+pattern(words,"####")+")"
  if theWord contains "." or theWord contains "?" or theWord contains "!"
    sentences=sentences+1
  endif
  n=n+1
while forever
  message "Average number of words per sentence: "+str(words/sentences)

```

Here is an example of the status bar display after the `statusmessage` statement.



Views: This statement may be used in any view, but it is ignored if the procedure window is not open.

See Also: [debug](#) statement
[message](#) statement
[logmessage](#) statement

STOP

Syntax: STOP

Description: The **Stop** statement stops the current procedure immediately.

Parameters: This statement has no parameters.

Action: A procedure continues running until it reaches the last statement in the procedure or it encounters a **stop** statement. The [rtn](#) statement will also stop a procedure if the procedure has not been called as a subroutine. If you only want the procedure to stop under certain conditions you should use the **stop** statement with [if](#) or [case](#) statements.

Examples: This following example will cause the procedure to stop running if the current value of prices is less that the current value of cost.

```
if price<cost
  message "That's all folks"
  stop
endif
Qty=Qty+1
```

Views: This statement may be used in any view, and also works when no windows are open at all.

See Also: [rtn](#) statement
[debug](#) statement
[if](#) statement
[case](#) statement

STOPLOOPIF

- Syntax:** STOPLOOPIF true-false test
- Description:** The **stoploopif** statement decides whether to continue with a loop or to stop and skip to the end of the loop.
- Parameters:** This statement has one parameter: true-false test.
- true-false test** is a formula that should result in a true (-1) or false (0) answer. Usually the formula is created with a combination of comparison operators (=, ≠, <, >, etc.) and boolean combinations (and, or, etc.) For example the formula **Name="Smith"** will be true if the field or variables **Name** contains the value Smith, and false if it contains any other value.
- Action:** This statement decides whether to continue with a loop or to stop and skip to the end of the loop. If the test is true, the loop will stop and Panorama will skip to the first statement after the loop. If the test is false, the loop will continue normally
- Examples:** The example builds a list of the alphabetic letters used in the field Notes. The stoploopif statement is used to test if there are no more characters to check.

```

local X,aLetter,LetterList
X=1
LetterList=""
loop
  aLetter=upper(array(Notes,X,¶))
  stoploopif aLetter=""
  X=X+1
  repeatloopif aLetter≠striptoalpha(aLetter)
  if LetterList notcontains aLetter
    LetterList=LetterList+aLetter
  endif
while forever

```

Views: This statement may be used in any view.

See Also: [loop](#) statement
[until](#) statement
[while](#) statement
[repeatloopif](#) statement
[if](#) statement
[else](#) statement
[endif](#) statement

STOPTAB

- Syntax:** STOPTAB
- Description:** The **stoptab** statement aborts any pending TAB operation.
- Parameters:** This statement has no parameters.
- Action:** The StopTab statement is designed to be used in an automatic procedure that is triggered by data entry. If the procedure was triggered by the **Tab** key, this statement will prevent the **Tab** from occurring after the procedure finishes.
- Examples:** This example automatically rejects negative values. If the user enters a negative value, an alert appears, then the input box re-opens so the data can be edited. (Note: In a formula chevrons with nothing in between («») returns the value of the current field.)
- ```
if «» < 0
 message "Negative values are not allowed"
 stoptab
 editcellstop
endif
```
- Views:** This statement may be used in a Data Sheet, Design Sheet or Form view.
- See Also:** [editcell](#) statement  
[editcellstop](#) statement  
[info\("trigger"\)](#) function

# STR(...)

**Syntax:** STR(number)

**Description:** The `str()` function converts a number into text. A plain format is used. If you want to format the number, use the [pattern\(\)](#) function.

**Parameters:** This function has one parameter: `number`.  
**number** is the number that you want to convert to text.

**Result:** This function returns an item of text containing the converted number.

**Examples:** The `str()` function is useful when you need to combine numbers and text together. This example converts the number `Orders` to text so that it can be placed in the middle of a sentence.

```
message "There have been "+str(Orders)+" orders this week."
```

If you need to format the number instead of converting with a plain number format, use the [pattern\(\)](#) function.

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the number parameter.

**See Also:** [pattern\(\)](#) function  
[val\(\)](#) function



# STRING255(...)

**Syntax:** `STRING255(text,space)`

**Description:** The `string255(` function converts text into a Pascal string. A Pascal String is a special text format that is sometimes used by the Macintosh ROM's (also called a `String255` or `Str255` because the text is limited to a maximum length of 255 characters). (Pascal is the name of a computer language, which in turn is named after a famous mathematician.)

**Parameters:** This function has two parameters: `text` and `space`.

**text** is the text that you want to convert into a Pascal string. This text should be less than 255 characters long.

**space** is a number defining the amount of space taken up by the Pascal string. If `space` is zero, the string may be up to 255 characters, and is not padded. If `space` is from 1 to 255, the `string255(` function makes sure that the string takes up exactly this amount of space. If the string is too long, it will be cut off. If the string is too short, it will be padded with nulls (bytes containing zeroes).

**Result:** This function returns a text data item containing a Pascal string.

**Examples:** See [c/pascal structures](#) for examples of the `string255(` function.

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to convert a number to a Pascal string.

**Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `space` parameter.

**See Also:** [text255\(](#) function  
[byte\(](#) function  
[word\(](#) function  
[longword\(](#) function  
[radix\(](#) function  
[radixstr\(](#) function  
[c/pascal structures](#)

# STRIP(...)

**Syntax:** STRIP(text)

**Description:** The `strip()` function strips off leading and trailing blanks and other whitespace (carriage returns, tabs, etc.)

**Parameters:** This function has one parameter: `text`.  
`text` is the item of text that you want to strip.

**Result:** The result of this function is always a text item.

**Examples:** The table below shows some examples of how the `strip()` function affects various items of text:

```
strip(" John Smith") John Smith ←
strip("net 30 ") →net 30 ←
strip(" New York ") →New York ←
```

As you can see, this function removes blanks at the beginning or end of the text, but does not affect blanks in the middle of the text. It also removes carriage returns, tabs, or any character with an ASCII value less than 32. This function can be used to modify fields or variables, or to display data. This example makes sure that there are no extra blank characters at the beginning or end of a name.

```
field Name
formulafill strip(Name)
```

For example, you might use this procedure after you imported data that had extra blanks.

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `strip(34)`. If you have a number you must convert the number to text before using it with this function, for example `strip(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

**See Also:** [striptoalpha\(\)](#) function  
[striptonum\(\)](#) function  
[stripchar\(\)](#) function  
[replace\(\)](#) function

# STRIPCHAR(...)

**Syntax:** STRIPCHAR(text,range)

**Description:** The `stripchar()` function removes characters you don't want from a text item. You specify exactly what kinds of characters you want and don't want included in the final output.

**Parameters:** This function has two parameter: `text` and `range`.

**text** is the item of text that you want to strip.

**range** specifies what kinds of characters you want to keep and what kinds of characters you want to strip away. The range consists of one or more pairs of characters. Each pair specifies a set of characters you want to keep. For example, the pair `AZ` means that you want to keep the characters from `A` to `Z`. For alphanumeric characters the set is pretty obvious. For other types of characters you should check an [ascii](#) chart. For example the pair `#&` specifies a set of four characters: `#`, `$`, `%` and `&`.

If a pair consists of the same character repeated twice in a row, the set is just that single character. For instance the pair `##` means you want to keep one character: `#`.

The range may consist of several pairs put together. For example the range `AZaz09..` consists of four pairs, and specifies that all letters, numbers, and periods will be kept, with all other characters stripped away.

**Result:** The result of this function is always a text item.

**Examples:** One handy use for this function is to quickly check if a field or variable contains any inappropriate characters. If a field or variable changes when you run it through the `stripchar()` function it must contain characters that are not part of the specified range. The procedure below checks to make sure that the field `StartTime` does not have any non-time characters in it (a `%`, for example).

```
if StartTime#stripchar(StartTime,"09:: AAaaMMmmPPpp")
 message "Invalid character in start time!"
endif
```

The partial [ascii](#) charts below shows exactly how text is affected by different combinations of this function. Characters in the colored area get through, everything else is removed. (Any part of the [ascii](#) chart that is not displayed consists of characters that are removed.)

**Positive/negative numbers with decimal point**

```
stripchar(...,"09-.")
```

10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SPC	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

## Floating point numbers

```
stripchar(..., "09-.EEee")
```

10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SPC	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

## Letters and spaces

```
stripchar(..., "AZaz ")
```

10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SPC	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
80	À	Á	Ç	È	É	Ë	Ê	Ë	Ì	Í	Î	Ï	Ñ	Ò	Ó	Ô

## US and International Letters

```
stripchar(..., "AZazÄü")
```

30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
80	À	Á	Ç	È	É	Ë	Ê	Ë	Ì	Í	Î	Ï	Ñ	Ò	Ó	Ô
90	È	É	Ë	Ê	Ë	Ì	Í	Î	Ï	Ñ	Ò	Ó	Ô	Õ	Ù	Ú
A0	†	°	¢	£	§	•	¶	ß	®	©	™	ˆ	˜	≠	ℙ	ℚ

## Everything Except Spaces

```
stripchar(...,"!ÿ")
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20	SPC	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
80	À	Á	Â	Ã	Ä	Å	Ë	È	É	Ê	Ë	Ì	Í	Î	Ï	Ñ
90	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	à	á
A0	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ñ	ò
B0	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ß	à	á	â
C0	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ñ	ò	ó
D0	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ß	à	á	â	ã
E0	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ñ	ò	ó	ô
F0	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ñ	ò	ó	ô	õ

### Errors:

**Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `stripchar(34,"09")`. If you have a number you must convert the number to text before using it with this function, for example `stripchar(str(34),"09")`.

### See Also:

[ascii](#)  
[striptoalpha\(\)](#) function  
[striptonum\(\)](#) function  
[replace\(\)](#) function



**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `striptoalpha(34)`. If you have a number you must convert the number to text before using it with this function, for example `striptoalpha(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

**See Also:** [ascii](#)  
[striptonum\(\)](#) function  
[stripchar\(\)](#) function  
[replace\(\)](#) function





**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `striptonum(34)`. If you have a number you must convert the number to text before using it with this function, for example `striptonum(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

**See Also:** [ascii](#)  
[striptoalpha\(\)](#) function  
[stripchar\(\)](#) function  
[replace\(\)](#) function

# STYLE

**Syntax:** `STYLE options`

**Description:** The `style` statement changes the style and color of one or more data cells. (Note: This is identical to the [stylecolor](#) statement.)

**Parameters:** This statement has one parameter: `options`.

**options** is a text item that controls what cells get changed (cell, record, field, all), what color the cells should be changed to (black, red, green, blue, cyan, yellow, magenta) and what style (bold, italic, underline, shadow).

If the options start with the word **cell**, only the current cell will be changed. If the options start with the word **record**, all the cells in the current record will be changed. If the options start with the word **field**, all the selected cells in the current field will be changed. If the options start with the word **all**, every cell in every selected record will be changed. Here are some examples of different option combinations.

```
style "field blue bold"
style "all black"
style "cell red italic"
style "record bold"
```

**Action:** This statement changes the color and/or style of certain cells. (The color appears in the data sheet and in data cells displayed in a form if the **Use Data Style/Color** option in **Form Preferences** is turned on.) The cell will retain this color and style until it is changed again, or until the cell is edited. (When a cell is edited it reverts to plain black.)

**Examples:** This example makes all negative values in the Balance field red.

```
select Balance
field Balance
style "field red"
selectall
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [sortbycolor](#) statement  
[groupbycolor](#) statement  
[fieldstyle\(\)](#) function

# STYLECOLOR

**Syntax:** STYLECOLOR options

**Description:** The `stylecolor` statement changes the style and color of one or more data cells. (Note: This is identical to the [style](#) statement.)

**Parameters:** This statement has one parameter: options.

**options** is a text item that controls what cells get changed (cell, record, field, all), what color the cells should be changed to (black, red, green, blue, cyan, yellow, magenta) and what style (bold, italic, underline, shadow).

If the options start with the word **cell**, only the current cell will be changed. If the options start with the word **record**, all the cells in the current record will be changed. If the options start with the word **field**, all the selected cells in the current field will be changed. If the options start with the word **all**, every cell in every selected record will be changed.

Here are some examples of different option combinations.

```
stylecolor "field blue bold"
stylecolor "all black"
stylecolor "cell red italic"
stylecolor "record bold"
```

**Action:** The statement changes the color and/or style of certain cells. (The color appears in the data sheet and in data cells displayed in a form if the **Use Data Style/Color** option in **Form Preferences** is turned on.) The cell will retain this color and style until it is changed again, or until the cell is edited. (When a cell is edited it reverts to plain black.)

**Examples:** This example makes all negative values in the Balance field red.

```
select Balance field Balance
stylecolor "field red"
selectall
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [sortbycolor](#) statement  
[groupbycolor](#) statement  
[fieldstyle\(\)](#) function

# SUBSETFORMULASELECT

**Syntax:** SUBSETFORMULASELECT

**Description:** The `subsetformulaselect` statement opens the **Formula Select from Server** dialog. This dialog is normally opened from the Search Menu.

**Parameters:** This statement has no parameters.

**Action:** This statement opens the **Formula Select from Server** dialog. The user may then type in a formula which is used to extract a subset from the SQL server database.

**Examples:** This example simply opens the dialog.

```
subsetformulaselect
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [subsetselect](#) statement  
[selectall](#) statement  
[subsetselectdialog](#) statement  
[info\("serverstatus"\)](#) function

# SUBSETSELECT

**Syntax:** SUBSETSELECT formula

**Description:** The `subsetselect` statement selects a subset of the SQL server database and loads it into the local Panorama database.

**Parameters:** This statement has one parameter: formula.

**formula** is a formula that specifies what subset of the SQL server database is to be loaded into the local Panorama database. There are some very significant restrictions on this formula compared to a regular Panorama formula. The formula must consist of one or more comparisons between a field and a value, like this

```
Field compare-operator Constant
```

The Field must be just that—one field. You cannot concatenate multiple fields, use text funnels, or modify the field value with a function. The Constant must also be a single value like "Smith" or 89. You may not use fields, variables, operators or functions as part of the constant. The one exception is if the comparison is with a date field. In that case you may use the `date()` function, for example `date("1/1/97")`. However, in this case the text inside the date function ("1/1/"97") must be a single value with no fields, variables or operators.

The compare-operator is also restricted. You may not use the `soundslike`, `match`, `matchexact`, `notmatch`, or `notmatchexact` operators. In addition the `endswith`, `contains`, `notcontains`, `like` and `notlike` operators will not be able to take advantage of any indexes, so they may be quite slow depending on the size of the SQL database and whether any other indexed compares are part of your formula.

You may combine multiple comparisons with the `and` or `or` operators, and you may use parentheses to control the order in which comparisons are combined.

**Action:** This statement selects a subset of the SQL server database and loads it into the local Panorama database. If there is not enough room for the entire subset Panorama will load as much as possible. As a side effect all summary records will be removed from the local database.

**Examples:** This example will load all Oregon records into the local Panorama database.

```
subsetselect State = "OR"
```

Here are some examples of invalid subsetselect formulas. This example will not work because the left hand side of the comparison contains the + operator.

**subsetselect City+", "+State="Carson, NV"**

This example can be rewritten to work correctly like this.

```
subsetselect City="Carson" and State="NV"
```

Here is another bogus example that attempts to select all records within the last 30 days.

**subsetselect Date>today()-30**

Making this example work correctly is a bit trickier. To do so we must assemble the selection formula in the clipboard.

```
clipboard={Date>date(" }+
datepattern(today()-30,"mm/dd/yyyy")+{" }
subsetselect clipboard
```

If today's date is August 31, 1997 the first line of this procedure will put `Date>date("8/2/1997")` on the clipboard. The `subsetselect` then uses this formula directly from the clipboard. Warning: If you use `clipboard()` instead of `clipboard` this example will not work!

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [select](#) statement  
[subsetselectall](#) statement  
[subsetselectdialog](#) statement  
[subsetformulaselect](#) statement  
[info\("subsetformula"\)](#) statement

# SUBSETSELECTALL

**Syntax:** SUBSETSELECTALL

**Description:** The **subsetselectall** statement loads the entire SQL server database into the local Panorama database (if there is enough memory).

**Parameters:** This statement has no parameters.

**Action:** This statement copies the entire SQL server database into the local Panorama database. If there is not enough room for the entire database Panorama will load as much as possible. As a side effect all summary records will be removed from the local database

**Examples:** This example loads the entire SQL server database into the local Panorama database, then analyzes the data.

```
subsetselectall
 field "Date"
 groupup by month
 field "Amount"
 total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [subsetselect](#) statement  
[subsetselectdialog](#) statement  
[selectall](#) statement  
[subsetformulaselect](#) statement  
[info\("subsetformula"\)](#) statement

# SUBSETSELECTDIALOG

**Syntax:** SUBSETSELECTDIALOG

**Description:** The **subsetselectdialog** statement opens the **Select from Server** dialog. This dialog is normally opened from the Search Menu.

**Parameters:** This statement has no parameters.

**Action:** This statement opens the **Select from Server** dialog. The user may then specify comparisons which are used to extract a subset from the SQL server database.

**Examples:** This example simply opens the dialog.

```
subsetselectdialog
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [subsetselect](#) statement  
[selectall](#) statement  
[subsetformulaselect](#) statement  
[info\("serverstatus"\)](#) function



# SUM(...)

**Syntax:** SUM(field)

**Description:** The `sum()` function adds up all instances of a line item field in the current record. For example, `Sum(QtyΩ)` will add up all Qty1+Qty2+Qty3+... for as many line items fields are in the database. The `sum()` function has two advantages over simple addition: 1) it's easier to type, and 2) it actually calculates faster

**Parameters:** This function has one parameters: `field`.

**field** is the line item field you want to add up. This must be a numeric field, not text. You must use the Ω symbol (type option-Z) at the end of the field name.

**Result:** The result of this function is always a numeric value. If the line item field is an integer the result will be an integer, if the line item field is floating point the result will be floating point.

**Examples:** This example adds up the Amount line item fields, `Amount1+Amount2+Amount3+... .`

```
sum(AmountΩ)
```

This formula adds up the Tax line item fields.

```
sum("TaxΩ")
```

The quotes shown in this example are optional (although they were required in Panorama 2.1 and earlier versions).

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `sum(Description%)`.

# SUMMARYLEVEL

**Syntax:** SUMMARYLEVEL level

**Description:** The `summarylevel` statement changes the summary level of the current line.

**Parameters:** This statement has one parameter: level.

**level** is a number that specifies the new summary level of the record. This value may be between 0 (data record) and 7 (highest level summary).

**Action:** This statement will arbitrarily change the summary level of the current line. This is usually not a very useful thing to do. It is generally better to let summary records be manipulated by the [groupup](#), [removesummaries](#), and [removedetail](#) statements. Note: A procedure can find out the summary level of the current record with the [info\("summary"\)](#) function.

**Examples:** This example increases the summary level of the current record. If the current record is a data record, it becomes a level 1 summary, if it is a level 1 summary it becomes level 2. (Don't ask me why you would do this, it's just an example!)

```
if info("summary") < 7
 summarylevel 1+info("summary")
endif
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [groupup](#) statement  
[groupdown](#) statement  
[groupbycolor](#) statement  
[removesummaries](#) statement  
[removedetail](#) statement  
[outlinelevel](#) statement  
[info\("summary"\)](#) function

# Super Flash Art Programming

**Background:** The `superobject` and `activesuperobject` statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

**Quick Reference:**  
`"FindText"`  
`"ExtractText"`

**FindText**      `,<POINT>,<TEXT>`

This command returns the text at the specified point in the picture.

Finding out what text a user clicked on takes three components: 1) a Super Flash Art object, 2) a transparent Standard Button with the click/release option turned off, and 3) a procedure triggered by the Standard Button. The transparent button should be overlaid exactly on top of the Super Flash Art object...use the Align command to get exact alignment. If the Flash Art object has scroll bars, however, they should not be covered by the button. Only cover the area where the picture is displayed. The example below shows a procedure that will figure out what text was clicked on, and what font, size, and style the text is. The example assumes that the Super Flash Art object is named HyperFlash. (To give an object a name, first select the object, then use the Object Name command in the Edit menu or click on the object name in the Graphic Control Strip.)

```

local v,h,clickPoint
local clickText, clickFont, clickSize, clickStyle
v=v(info("mouse"))-rtop(info("buttonrectangle"))
h=h(info("mouse"))-rleft(info("buttonrectangle"))
clickPoint=point(v,h)
superobject "HyperFlash", "FindText", clickPoint, clickText
clickFont=objectinfo("font")
clickSize=objectinfo("textsize")
clickStyle=objectinfo("textstyle")

```

**ExtractText**      `,<FONT>,<SIZE>,<STYLE>,<CONNECTOR>,<RESULT>`

The SuperObject "ExtractText" command allows all the text that matches specified criteria to be extracted from the picture currently displayed in a Super Flash Art object. This command has several parameters as shown here:

The first parameter, `<FONT>`, specifies the font of the text you want to extract. If you don't care what the font is, leave this parameter empty ("").

The second parameter, `<SIZE>`, specifies the size of the text you want to extract. If you don't care, this parameter should be zero.

The third parameter, `<STYLE>`, specifies the style of the text you want to extract. This parameter allows you extreme flexibility in selecting what styles you want to extract.

If the Style parameter is zero, any style is ok. For example, if you want to extract all Monaco 9 point text of any style into a variable named Samples, here's how you would do it:

```

superobject "HyperFlash", "ExtractText", "Monaco", 9, 0, ";", Samples

```

If the Style parameter is 1-255, it specifies the exact style you want. Add up the numbers for each individual style. For example, for underlined text you would specify 4, for bold text, 1. The example below will extract all bold text, but not bolditalic or bold underlined.

```
superobject "HyperFlash", "ExtractText", "", 0, 1, ¶, Samples
```

If the Style parameter is 256 or greater, it specifies both the style and a mask for the style. The mask allows you to isolate individual styles. The mask uses the same style numbers as the individual styles, but multiplied by 256. (Why 256? 256 is 2 the 8th power (2<sup>8</sup>), an even number in the computer's binary numbering system.) For example, suppose you wanted to extract all bold text, even bold text that is combined with other styles. By using a mask of 4\*256 you tell the ExtractText command that you only care about the underlined style. The example below will extract all underlined, underlined-italic, underlined-bold; any text that is underlined no matter what other attributes it may have.

```
superobject "HyperFlash", "ExtractText", "", 0, (4*256)+4, ¶, Samples
```

The fourth parameter, <CONNECTOR>, specifies what connector character(s) should be used between segments as they are extracted. Usually this is a carriage return (¶), comma, space, slash, etc. The character "t" is a special separator. When this separator is used, Panorama checks each piece of extracted text to see if it is on the same line as the previously extracted piece of text. If it is on the same line, Panorama will connect the pieces with a space. If the two pieces are on different lines, they will be connected with a carriage return. This allows the extracted text to be more or less reconstructed in its original form. (Note: Either "t" or "T" will trigger this special operation.) The final parameter, Result, is the field or variable that you want the extracted text placed into. The procedure below displays the number of underlined segments in the current picture.

```
local KeywordList
superobject "HyperFlash", "ExtractText", "", 0, 4, ¶, KeywordList
message arraysize(KeywordList, ¶)
```

The next example takes all the Monaco 9 point text in the current picture and combines it. It then copies the text onto the clipboard.

```
local SampleText
superobject "HyperFlash", "ExtractText", "Monaco", 9, 0, "t", SampleText
clipboard=SampleText
```

# Super Matrix Programming

**Background:** The `superobject` and `activesuperobject` statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

**Quick Reference**

```
"ReDraw",<AREA>,<START>,<END>
"CellRectangle",<CELL>,<RECTANGLE>
"CellToXY",<CELL>,<ROW>,<COL>
"MatrixSize",<CELLS>
```

**Redraw** ,<AREA>,<START>,<END>

This command redraws some or all of the cells in a super matrix. The first parameter, <AREA>, defines the area that will be redrawn. Legal options for this parameter are: "all", "column", "row", and "cell".

The <START> and <END> parameters define the start and end of the area to be redrawn. For example, if the <AREA> parameter was "column" and the last two parameters were 3 and 5, then columns 3 thru 5 would be redrawn. (Note: The start and end values are ignored if the "all" area is chosen.)

The following examples illustrate different ways a matrix might be updated. This calendar example redisplay the entire month.

```
superobject "Month","redraw","all",0,0
```

This example redisplay only weekdays.

```
superobject "Month","redraw","column",2,6
```

This example works with a matrix of photographs. The procedure redisplay photo 7 only.

```
superobject "Photographs","redraw","cell",7,7
```

This example redisplay all photos after photo 12. This procedure would be used if someone inserts or deletes a photograph at position 12.

```
superobject "Thumbnails","redraw","cell",12,9999
```

**CellRectangle** ,<CELL>,<RECTANGLE>

This command returns the dimensions of an individual cell in the matrix. The dimensions are in window co-ordinates. The <CELL> parameter should be a number from 1 to the maximum number of cells in the matrix. The <RECTANGLE> parameter should be a field or variable where the rectangle will be stored.

This example uses the "CellRectangle" command to open a new window over the current matrix cell (the matrix cell that was clicked on).

```
local subWindowRectangle
superobject "Calendar",
"CellRectangle",info("matrixcell"),subWindowRectangle
setwindowrectangle subWindowRectangle," "
openform "Day"
```

**CellToXY** ,<CELL>,<ROW>,<COL>

This command converts a matrix cell number into a row and column. The example below displays what row and column were clicked on.

```
local mRow,mCol
superobject "Thumbnail","CellToXY",info("matrixcell"),mRow,mCol
message "You clicked on row "+str(mRow)+" and column "+str(mCol)
```

**MatrixSize** ,<CELLS>

This command calculates the current number of cells in the matrix. Here is a procedure that displays all the vital statistics for a matrix.

```
local mCells,mRows,mCols
superobject "Images","MatrixSize",mCells
superobject "Images","CellToXY",mCells,mRows,mCols
message "This matrix contains "+str(mCells)+" cells ("+
str(mRows)+" rows by "+str(mCols)+" columns).
```

This information can be very useful if you want to attach the matrix to a scroll bar.

# SUPERDATE(...)

**Syntax:** SUPERDATE(date,time)

**Description:** The `superdate()` function converts a regular date and a regular time into a superdate. SuperDates combine the date and time into a single number...the number of seconds since January 1, 1904. SuperDates make it easy to calculate time intervals across multiple days. However, SuperDates take up more storage than regular dates, and are not as easy to work with. In addition, SuperDates are limited to dates between 1904 and 2040.

**Parameters:** This function has two parameters: `date` and `time`.

**date** is a regular Panorama date (the number of days since January 1, 4713 B.C.). This date must be between 1904 and 2040 A.D.

**time** is a regular Panorama time (the number of seconds since midnight).

**Result:** This function returns a value that combines both the date and time into a single number.

**Examples:** In its heyday, the Santa Fe Super Chief train would travel between Chicago and Los Angeles in 39 and 1/2 hours. This example uses SuperDates to calculate the arrival time and day after the user enters the departure time.

```
local Arrival,Departure,xTime
xTime="7:30 pm"
gettext "Departure Time",xTime
Departure=superdate(today(),time(xTime))
Arrival=Departure+superdate(0,time("39:30:00"))
message "Arrives "+datepattern(regulardate(Arrival),"DayOfWeek")+
 " at "+timepattern(regulartime(Arrival),"hh:mm am/pm")
```

If the train leaves at 7:30 pm on Monday, the message will be Arrives Wednesday at 10:00 am. As you can see, SuperDate arithmetic is very easy, just add or subtract. There's no need to worry about crossing midnight, because that simply is the start of a new day.

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the date or time parameters.

**See Also:** [regulardate\(\)](#) function  
[regulartime\(\)](#) function  
[date\(\)](#) function  
[time\(\)](#) function  
[today\(\)](#) function  
[now\(\)](#) function

# SUPEROBJECT

**Syntax:** SUPEROBJECT object,command,param1,param2,...paramN

**Description:** The **superobject** statement allows a procedure to communicate with a SuperObject on the current form. This statement is similar to the [activesuperobject](#) statement, but requires you to specify a named object in the form (instead of using the currently active object.) For information on using this command with specific types of SuperObjects, see:

[text editor programming](#)  
[word processor programming](#)  
[super flash art programming](#)  
[list superobject programming](#)  
[super matrix programming](#)  
[scroll bar programming](#)

**Parameters:** This statement has a variable number of parameters, but always at least two: **object** and **command**.

**object** is the name of the graphic object you want to send the command to. If a graphic object does not have a name, you cannot send a command to it with the **superobject** statement.

There are two ways to give a name to an object. Both start by selecting the object (in Graphics Mode). Once the object is selected, you can use the **Object Name** command (in the Edit menu) to assign the name. Or you can click on the object name portion of the Graphic Control Strip along the bottom of the window. (If the object name is not visible, click the triangle on the right side of the strip until you see the object name.

It is possible to have two or more graphic objects on a form with the same name, but this is usually not a good idea. If the SuperObject statement finds more than one object with the same name it will attempt to send the command to all of them, which may or may not make sense. It definitely doesn't make sense if the command is supposed to return information about the object (GetSelection, FindCell, etc.) or if this is a command that requires the SuperObject to be active for editing (since only one SuperObject may be active at once).

**command** is an instruction that will be sent to a specific SuperObject. Different types of SuperObjects understand different types of commands. For example the Text Editor SuperObject understands commands like "InsertText" and "GetSelection", while the List SuperObject understands commands like "AddCell" and "FindCell".

**param1...paramN** are additional parameters used by individual commands, if required. For example the Text Editor's "InsertText" command requires one additional parameter which specifies the text to be inserted. The List SuperObject's "DeleteCell" command requires two additional parameters, the numbers of the first and last cells to be deleted. Parameters may also receive values from the SuperObject. For example the List SuperObject's "GetCount" command has one parameter into which the number of items in the list is stored. If a parameter is used to receive a value from the SuperObject, that parameter must be a single field or variable with no operators (`myValue`, not `myValue+yourValue` or `strip(myValue)`).



**Action:** This statement allows a procedure to communicate with a SuperObject on the current form. You specify what object you want to communicate with by its name. Many SuperObjects have one or more commands that they understand. For example, the Text Editor SuperObject has commands for selecting text, locating text, modifying text, etc.

**Examples:** This example insert the current date and time into whatever SuperObject is currently being edited. This procedure will work with both Text Editor and Word Processor SuperObjects. The first few lines of the procedure check to make sure that there actually is an active SuperObject (i.e. something is really being edited at this time).

```
local EditingObject
EditingObject=info("activesuperobject")
if EditingObject=""
 superobject EditingObject,"InsertText",
 datepattern(today(), "mm/dd/yy")+"@"+
 timepattern(now(), "hh:mm am/pm")
endif
```

**Views:** This statement may be used in Form views.

**See Also:** [activesuperobject statement](#)  
[info\("activesuperobject"\) statement](#)  
[text editor programming](#)  
[word processor programming](#)  
[super flash art programming](#)  
[list superobject programming](#)  
[super matrix programming](#)  
[scroll bar programming](#)

# SUPEROBJECTCLOSE

**Syntax:** SUPEROBJECTCLOSE

**Description:** The `superobjectclose` statement closes any Text Editor SuperObject or Word Processing Superobject that is currently open. If nothing is being edited, this statement does nothing.

**Parameters:** This statement has no parameters.

**Action:** The `superobjectclose` statement is equivalent to:

```
if info("activesuperobject")≠"
 activesuperobject "close"
endif
```

**Examples:** This example uses the `superobjectclose` statement to make sure that editing is finished before saving the file.

```
superobjectclose
save
```

**Views:** This statement must only be used when a form is active.

**See Also:** [activesuperobject](#) statement  
[info\("activesuperobject"\)](#) function

# TABDOWN

**Syntax:** TABDOWN

**Description:** The **tabdown** statement toggles between **tab across** and **tab down** modes. This is the same as pressing the **Tab Down** tool in the tool palette. Note: A procedure can find out the current status of the tab down option with the [info\("tabdown"\)](#) function.

**Parameters:** This statement has no parameters.

**Examples:** The procedure below turns off the tab down option. In other words, this procedure makes sure that the tab down option is OFF.

```
if info\("tabdown"\)
 tabdown
endif
```

**Views:** This statement may be used in a Data Sheet or View-As-List form view.

**See Also:** [info\("tabdown"\)](#) function

## TABLE(...)

**Syntax:** TABLE(file,keyField,keyValue,dataField,default,level)

**Description:** The `table()` function searches a database for a value, then returns other information from the same record. Unlike the `lookup()` function, the `table()` function does not require an exact match. If it does not find an exact match the `table()` function will use the closest match. For example, the `table()` function can look up a tax rate given an income amount, or look up a shipping price given a zip code and weight.

**Parameters:** This function has six parameters: `file`, `keyField`, `keyValue`, `dataField`, `default` and `level`.

**file** is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use `info("database")`.

**keyField** is the name of the field that you want to search in. For example if you want to look up a shipping price by weight, this should be the field that contains weights. The field must be in the database specified by the first parameter.

**keyData** is the actual data that you want to search for. For example if you want to look up a shipping price by weight, this should be the actual weight of the package. This parameter is often a field in the current database.

**dataField** is the name of the field that you want to retrieve data from. For example if you want to retrieve a shipping price, this should be the name of the field that contains prices. This must be a field in the database specified by the first parameter.

**default** is the value you want this function to return if it is unable to find the information specified by the `keyField` and `keyData` parameters. This will only happen if the `keyData` value is smaller than the smallest value in the `keyField` field. The data type of the default value should match the data type of the `dataField`. If the `dataField` is numeric, the default should usually be zero. If the `dataField` is text, the default should usually be "".

**level** is the minimum summary level to be searched. Usually this parameter is zero so that the entire database will be searched. If the level is set to 1 through 7, only summary records will be searched.

**Result:** If the function is able to locate the information specified by the `keyField` and `keyData` parameters it returns the contents of the specified field in the specified database. If it cannot find an exact match, it finds the closest value (but not greater than the `keyData` value. If the `keyData` value is smaller than any value in the `keyField` the function returns the default value.

For example, suppose the key field contains the values 5, 25, 100, 250 and 1000. If the key value is 47, the `table()` function will match with the record containing 25 in the key field. If the key value is 4700, the `table()` function will match with the record containing 1000 in the key field. If the key value is 4, there is no match, because there is no value in the key field less than 4. In this case the default value will be used.

**Examples:**

The table function is designed to be used with rate lookup tables like tax tables, shipping tables, volume discount tables etc. Our example will calculate shipping prices. Suppose you have a database called Shipping Rates that contains the fields and values shown here.

Weight	Zone1	Zone2	Zone3
0	2.50	4.00	5.00
50	2.35	3.80	4.70
100	2.25	3.60	4.50
250	2.12	3.40	4.25
500	2.03	3.00	4.05
1000	1.94	2.85	3.85
2000	1.86	2.70	3.70

The `table()` function interprets this table like this: From 0-49 pounds in Zone 1, the rate is \$2.50 per pound. From 50-99 pounds the rate is \$2.35/pound. From 100-249 the rate is \$2.25 per pound, and so on. Items 2,000 pounds and over are shipped for \$1.86 per pound. The other zones are similar.

The procedure below calculates the shipping charges for a package using the database shown above.

```

local PackageWeight, DestinationZone, ShippingCharge
PackageWeight="" DestinationZone=""
gettext "Package weight:", PackageWeight
PackageWeight=val(PackageWeight)
if PackageWeight<=0
 message "Sorry, anti-gravity option not available."
 stop
endif
gettext "Zone Number (1-3)", DestinationZone
if length(DestinationZone)<>1 or DestinationZone<"1" DestinationZone>"3"
 message "Zone must be from 1 to 3"
 stop
endif
ShippingCharge=Weight*table("Shipping Rates", Weight, PackageWeight,
 "Zone"+DestinationZone, 0, 0)
message "Shipping charge is: "+pattern(ShippingCharge, "$#, .##")

```

Notice that this example actually calculates the name of the data field on the fly: either Zone1, Zone2, or Zone3. The data field is still a single field (remember, only one item can be transferred at a time) but we are using a formula to calculate what the name of that field is.

In a real database you probably would not ask the user to enter the zone, but would have another database that would relate zones to zip codes. Here's a simple Zone Chart database that divides the entire USA into three zones based on the first three digits of the zip code.

Zip3	Zone
000	1
300	2
700	3
99:	0

The last value in this table, 99:, is the smallest value that is greater than the last legal zip code (999) according to the [ascii](#) character order. This record can help catch illegal zip codes. For instance, ABC is greater than 99:, so the Zone will be 0 for this illegal zip code.

The assignment below will turn a regular zip code (Zip) into a zone number according to the Zone Chart database.

```
DestinationZone=table("Zone Chart",Zip3,Zip[1,3],Zone,0,0)
```

This assignment can easily be plugged into the previous example to calculate the shipping charges given the weight and zip code.

**Errors:**

**Database does not exist.** This error occurs if there is no open database with the name you have specified. You have either misspelled the name, or the database is not currently open.

**Field or variable does not exist.** This error occurs if there is no field in the specified database with the name you have specified. You probably misspelled the field name.

**See Also:**

[lookup\( function](#)  
[lookuplast\( function](#)  
[lookupselected\( function](#)  
[grabdata\( function](#)  
[lookupall\( function](#)  
[ascii](#)

# TAGARRAY(...)

**Syntax:** TAGARRAY(text,header,trailer,separator)

**Description:** The `tagarray()` function builds an array (see Text Arrays) containing the body of all the specified tags (usually HTML tags) in the text. Each element in the array is separated from the next with the separator character (usually ¶ or ,). See [html tag parsing](#) for more information on HTML tags.

**Parameters:** This function has four parameters: `text`, `header`, `trailer` and `separator`.

**text** is the item of text that contains the data you want to extract.

**header** is the text that you want to use as a tag header. For example, if you want to extract all HTML tags, use "<" as the header. If you wanted to extract all bold text you would use "<B>" as the header. (Note: Upper or lower case is ignored, so "<b>" will also work.)

**trailer** is the text that you want to use as a tag trailer. For example, if you want to extract all HTML tags, use ">" as the trailer. If you wanted to extract all bold text you would use "</B>" as the trailer. (Note: Upper or lower case is ignored, so "</b>" will also work.)

**separator** is the separator character for the output array. This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ␣ character (option-L).

**Result:** This function returns an array that contains the contents of every specified tag, one tag per array item. The headers and trailers themselves are not included as part of each array item.

**Examples:** This example assumes you have a field or variable named `myPage` that contains an HTML page. The example will find and list all of the HTML tags in the page.

```
local myTags
myTags=tagarray(myPage,"<",">",";")
arrayfilter myTags,myTags,";","?(import() notmatch "/*",import(),"")
arraydeduplicate myTags,myTags,";"
message myTags
```

The output from this example will be something like this (any tags starting with / have been removed by the [arrayfilter](#) statement):

```
B,BODY,CENTER,FONT,HTML,TITLE
```

The more useful example below displays all of the links from this page to any other page.

```
message tagarray(myPage,{href="},{"},¶)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text`, `header`, `trailer` or `separator` parameters.

**See Also:** [html tag parsing](#)  
[text arrays](#)  
[tagcount\(\)](#) function  
[tagdata\(\)](#) function

[tagstart\(\)](#) function  
[tagend\(\)](#) function  
[tagnumber\(\)](#) function  
[tagparameter\(\)](#) function  
[tagparameterarray\(\)](#) function



# TAGCOUNT(...)

**Syntax:** TAGCOUNT(text,header,trailer)

**Description:** The `tagcount()` function counts the number of times a specified tag (usually an HTML tag) appears in the text. See [html tag parsing](#) for more information on HTML tags.

**Parameters:** This function has three parameters: `text`, `header` and `trailer`.

**text** is the item of text that contains the tags you want to count.

**header** is the text that you want to use as a tag header. For example, if you want to count all HTML tags, use "<" as the header. If you wanted to count all bold text items you would use "<B>" as the header. (Note: Upper or lower case is ignored, so "<b>" will also work.)

**trailer** is the text that you want to use as a tag trailer. For example, if you want to count all HTML tags, use ">" as the trailer. If you wanted to count all bold text items you would use "</B>" as the trailer. (Note: Upper or lower case is ignored, so "</b>" will also work.)

**Result:** This function returns the number of times the specified tag occurs in the text (or zero if it never occurs).

**Examples:** This example counts the number of tags on a page, assuming you have a field or variable named `myPage` that contains an HTML page.

```
message "This page contains "+
tagcount(myPage,"<",">")+ " tags."
```

This example counts the pictures on the page.

```
message "This page contains "+tagcount(myPage,"")+ " images."
```

**Errors:** Type mismatch: numeric argument used when text was expected. This error occurs if you attempt to use a numeric value for the `text`, `header` or `trailer` parameters.

**See Also:** [html tag parsing](#)  
[text arrays](#)  
[tagarray\(\)](#) function  
[tagdata\(\)](#) function  
[tagstart\(\)](#) function  
[tagend\(\)](#) function  
[tagnumber\(\)](#) function

# TAGDATA(...)

**Syntax:** `text,header,trailer,item)`

**Description:** The `tagdata()` function extracts the body of the specified tag (usually an HTML tag) in the text. See [html tag parsing](#) for more information on HTML tags.

**Parameters:** This function has four parameters: `text`, `header`, `trailer` and `item`.

**text** is the item of text that contains the data you want to extract.

**header** is the text that you want to use as a tag header. For example, if you want to extract an HTML tag, use "<" as the header. If you wanted to extract a bold text item you would use "<B>" as the header. (Note: Upper or lower case is ignored, so "<b>" will also work.)

**trailer** is the text that you want to use as a tag trailer. For example, if you want to extract an HTML tag, use ">" as the trailer. If you wanted to extract a bold text item you would use "</B>" as the trailer. (Note: Upper or lower case is ignored, so "</b>" will also work.)

**item** is the number of the tag you want to extract. For example, suppose you are using "<" and ">" as the tag header and tag trailer. Using an item number of 1 will extract the first tag, 2 the second tag, 3 the third tag, etc.

**Result:** This function returns the contents (`text`) of the specified tag. The headers and trailers themselves are not included as part of the extracted text.

**Examples:** This example assumes you have a field or variable named `myPage` that contains an HTML page. The example will display the title of the page.

```
message tagdata(myPage,"<title>","</title>",1)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text`, `header` or `trailer` parameters.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the `item` parameter.

**See Also:** [html tag parsing](#)  
[text arrays](#)  
[tagarray\(\)](#) function  
[tagcount\(\)](#) function  
[tagstart\(\)](#) function  
[tagend\(\)](#) function  
[tagnumber\(\)](#) function  
[tagparameter\(\)](#) function  
[tagparameterarray\(\)](#) function

# TAGEND(...)

- Syntax:** TAGEND(text,header,trailer,item)
- Description:** The `tagend()` function returns the ending position of the specified tag (usually an HTML tag) in the text. See [html tag parsing](#) for more information on HTML tags.
- Parameters:** This function has four parameters: `text`, `header`, `trailer` and `item`.
- text** is the item of text that contains the tag you want to locate.
- header** is the text that you want to use as a tag header. For example, if you want to locate an HTML tag, use "<" as the header. If you wanted to locate a bold text item you would use "<B>" as the header. (Note: Upper or lower case is ignored, so "<b>" will also work.)
- trailer** is the text that you want to use as a tag trailer. For example, if you want to locate an HTML tag, use ">" as the trailer. If you wanted to locate a bold text item you would use "</B>" as the trailer. (Note: Upper or lower case is ignored, so "</b>" will also work.)
- item** is the number of the tag you want to extract. For example, suppose you are using "<" and ">" as the tag header and tag trailer. Using an item number of 1 will locate the first tag, 2 the second tag, 3 the third tag, etc.
- Result:** This function returns the position of the end of the specified tag within the text (just before the tag trailer). If the tag is not found, the result will be zero.
- Examples:** This example assumes you have a field or variable named `myPage` that contains an HTML page that is being edited with a text superobject named `PageEditor`. The example will locate and select the title of the web page.
- ```

local tStart,tEnd
tStart=tagstart(myPage,"<title>","</title>",1)
if tStart=0
    rtn
endif
tEnd=tagend(myPage,"<title>","</title>",1)
superobject "PageEditor","setselection",tStart-1,tEnd-1

```
- Errors:**
- Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text`, `header` or `trailer` parameters.
- Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the `item` parameter.
- See Also:** [html tag parsing](#)
[text arrays](#)
[tagarray\(\)](#) function
[tagcount\(\)](#) function
[tagdata\(\)](#) function
[tagstart\(\)](#) function
[ttagnumber\(\)](#) function

TAGNUMBER(...)

Syntax: TAGNUMBER(text,header,trailer,position)

Description: The `tagnumber()` function checks to see if a specified position is inside of a tag (usually an HTML tag). Please see [html tag parsing](#) for more information on HTML tags.

Parameters: This function has four parameters: `text`, `header`, `trailer` and `position`.

text is the item of text that contains the tag you want to locate.

header is the text that you want to use as a tag header. For example, if you want to check if the position is within any HTML tag, use "<" as the header. If you wanted to check if the position is inside a bold text area you would use "" as the header. (Note: Upper or lower case is ignored, so "" will also work.)

trailer is the text that you want to use as a tag trailer. For example, if you want to check if the position is within any HTML tag, use ">" as the trailer. If you wanted to check if the position is inside a bold text area you would use "" as the trailer. (Note: Upper or lower case is ignored, so "" will also work.)

position is the position within the text parameter, starting with 0 for the first character.

Result: This function returns the tag number (1, 2, 3, etc.) or 0 if the position is not within any specified tag.

Examples: This example assumes you have a field or variable named `myPage` that contains an HTML page that is being edited with a text superobject. The example will check the currently selected text to see if it is inside a tag. If it is, the body of that tag will be selected.

```

local tStart,tEnd,clickStart,clickEnd,thetag
activesuperobject "getselection",clickStart,clickEnd
thetag=tagnumber(Page,"<",">",clickStart)
if thetag>0
    tStart=tagstart(myPage,"<title>","</title>",1)
    if tStart=0
        rtn
    endif
    tEnd=tagend(myPage,"<title>","</title>",1)
    activesuperobject "setselection",tStart-1,tEnd-1
endif

```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text`, `header` or `trailer` parameters.

Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value for the `item` parameter.

See Also: [html tag parsing](#)
[text arrays](#)
[tagarray\(\)](#) function
[tagcount\(\)](#) function

[tagdata\(\)](#) function
[tagstart\(\)](#) function
[tagend\(\)](#) function

TAGPARAMETER(...)

Syntax: TAGPARAMETER(text,name,item)

Description: The `tagparameter()` function extracts the value of a tag parameter embedded in some text, where the tag parameter takes the form `name=value`.

Parameters: This function has three parameters: text, name and item.

text is the item of text that contains the data you want to extract.

name is the name of the tag parameter you want to extract, including the trailing = sign (you may use other terminators, for example colons (`font:Helvetica`) or period (`font.Helvetica`). For example, if you want to extract the font name from the text `font=Helvetica size=12` the name would be `font=`.

item is the number of the parameter you want to extract. This is usually 1, but may be other values if you expect multiple instances of the parameter (multiple fonts, for example).

Result: This function returns the contents (text) of the specified parameter. If the parameter is surrounded by quotes, the quotes are removed. (Quotes are necessary if the parameter value contains spaces, for example `font="Times Roman"`.)

Examples: This example very quickly builds a list of all the GIF and JPEG images displayed in the page HTMLPage.

```
local imageTags,imageNames
imageTag=tagarray(HTMLPage,"<IMG",">",1)
arrayfilter imageTag,imageNames,1,
tagparameter( import(), "src=",1)
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text, header or trailer parameters.

Type mismatch: text argument used when numeric was expected. This error occurs if you attempt to use a text value for the item parameter.

See Also: [html tag parsing](#)
[text arrays](#)
[tagarray\(\)](#) function
[tagcount\(\)](#) function
[tagstart\(\)](#) function
[tagend\(\)](#) function
[tagnumber\(\)](#) function
[tagparameterarray\(\)](#) function

TAGPARAMETERARRAY(...)

- Syntax:** TAGPARAMETERARRAY(text,name,separator)
- Description:** The `tagparameter()` function extracts the value of multiple tag parameter embedded in some text, where each tag parameter takes the form `name=value`.
- Parameters:** This function has three parameters: text, name and separator.
- text** is the item of text that contains the data you want to extract.
- name** is the name of the tag parameter you want to extract, including the trailing = sign (you may use other terminators, for example colons (`font:Helvetica`) or period (`font.Helvetica`). For example, if you want to extract the font name from the text `font=Helvetica size=12` the name would be `font=`.
- separator** is the separator character for the output array. This should be a single character. For carriage return delimited arrays, use the ¶ character (**Option-7/Alt-0182**). For tab delimited arrays use the ␣ character (**Option-L/Alt-0172**).
- Result:** This function returns an array of all instances of the specified parameter within the text. If the parameter is surrounded by quotes, the quotes are removed. (Quotes are necessary if the parameter value contains spaces, for example `font="Times Roman"`.)
- Examples:** This example assumes that you have a field name `Orders` that contains food orders like this:

```
<pizza topping=pepperoni topping=mushroom>
<pizza topping=olive>
<rigatoni sauce=red>
<pizza topping=clam topping=garlic>
<linguine sauce=white>
```

The procedure shown below can process this data

```
local pizzaOrders,pizzaToppings
pizzaOrders=tagarray(Orders,"<pizza",">","¶")
arrayfilter pizzaOrders,pizzaToppings,¶,
tagparameterarray( import(), "topping=","")
```

The final result is a list of toppings

```
pepperoni,mushroom
olive
clam,garlic
```

Although this example is not all that practical, this is a very powerful function for parsing languages like HTML, XML, or your own specification languages.

- Errors:**
- Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the text, header or trailer parameters.
- Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the item parameter.

See Also:

- [html tag parsing](#)
- [text arrays](#)
- [tagarray\(function](#)
- [tagcount\(function](#)
- [tagstart\(function](#)
- [tagend\(function](#)
- [tagnumber\(function](#)
- [tagparameter\(function](#)

TAGSTART(...)

- Syntax:** TAGSTART(text,header,trailer,item)
- Description:** The `tagstart()` function returns the starting position of the specified tag (usually an HTML tag) in the text. See [html tag parsing](#) for more information on HTML tags.
- Parameters:** This function has four parameters: `text`, `header`, `trailer` and `item`.
- text** is the item of text that contains the tag you want to locate.
- header** is the text that you want to use as a tag header. For example, if you want to locate an HTML tag, use "<" as the header. If you wanted to locate a bold text item you would use "" as the header. (Note: Upper or lower case is ignored, so "" will also work.)
- trailer** is the text that you want to use as a tag trailer. For example, if you want to locate an HTML tag, use ">" as the trailer. If you wanted to locate a bold text item you would use "" as the trailer. (Note: Upper or lower case is ignored, so "" will also work.)
- item** is the number of the tag you want to extract. For example, suppose you are using "<" and ">" as the tag header and tag trailer. Using an item number of 1 will locate the first tag, 2 the second tag, 3 the third tag, etc.
- Result:** This function returns the position of the specified tag within the text (just after the tag header). If the tag is not found, the result will be zero.
- Examples:** This example assumes you have a field or variable named `myPage` that contains an HTML page that is being edited with a text superobject named `PageEditor`. The example will locate and select the title of the web page.

```

local tStart,tEnd
tStart=tagstart(myPage,"<title>","</title>",1)
if tStart=0
    rtn
endif
tEnd=tagend(myPage,"<title>","</title>",1)
superobject "PageEditor","setselection",tStart-1,tEnd-1

```

- Errors:**
- Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `text`, `header` or `trailer` parameters.
- Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the `item` parameter.

- See Also:**
- [html tag parsing](#)
 - [text arrays](#)
 - [tagarray\(\)](#) function
 - [tagcount\(\)](#) function
 - [tagdata\(\)](#) function
 - [tagend\(\)](#) function
 - [tagnumber\(\)](#) function

TAN(...)

Syntax: TAN(angle)

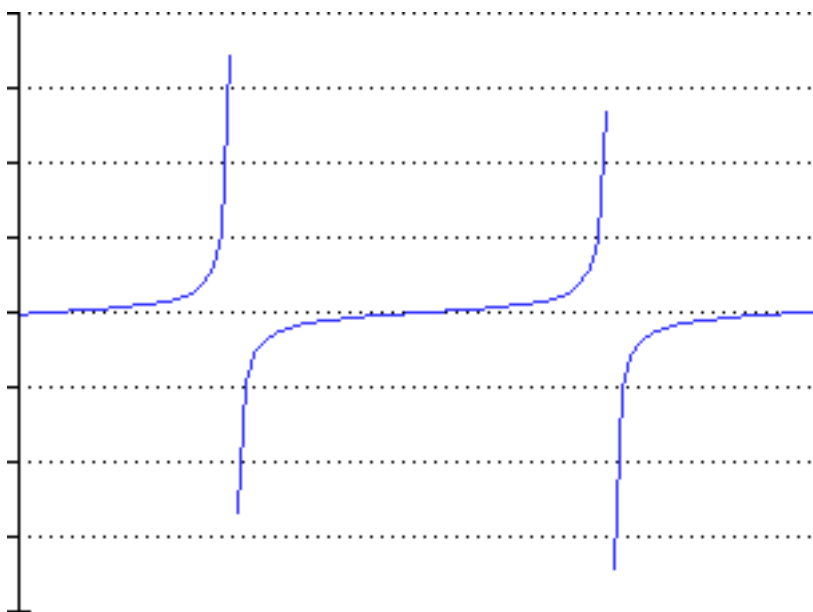
Description: The `tan()` function calculates the tangent of an angle.

Parameters: This function has one parameter: `angle`.

angle is a numeric value, an angle. The angle is usually specified in a mathematical unit of measurement called radians, however, within a procedure you can temporarily force Panorama to use degrees (see below). One radian is equal to approximately 57.2958 degrees (the exact value is $180/\pi$).

Result: The result of this function is a numeric floating point value.

Examples: The graph below shows the result of the sine function given input values from -3 to +3 radians.



This formula calculates the tangent of an angle in degrees.

```
tan(Angle*180/π)
```

In this example the angle is in a field or variable named `Angle`, however, you may use any formula that produces a numeric result in this location. The pi symbol (π) is produced by pressing `OPTION-P`.

Here is another way to calculate angles in degrees in a procedure.

```
degree
NewAngle=tan(Angle)
```

The `degree` statement tells Panorama to use degrees instead of radians in all trigonometry calculations. Panorama will continue to use degrees until the end of the procedure, or until a `radian` statement is encountered.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example. If you have a numeric value in a text item you must convert the text to the number data type before calculating the tangent, for example `tan(val("34"))`.

Floating point error. The tangent of $\pi/2$ radians (90°) is ∞ . If you attempt to calculate the tangent of $\pi/2$ a floating point error will occur.

See Also:

[sin\(\)](#) function
[cos\(\)](#) function
[degree](#) statement
[radian](#) statement
[arcsin\(\)](#) function
[arccos\(\)](#) function
[arctan\(\)](#) function
[val\(\)](#) function

TANH(...)

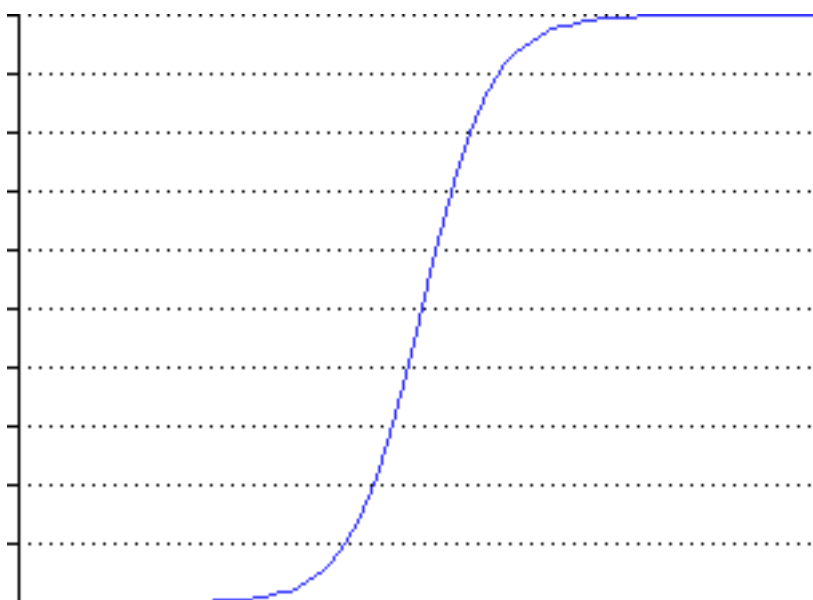
Syntax: TANH(value)

Description: The `tanh()` function calculates the hyperbolic tangent of a numeric value.

Parameters: This function has one parameter: value.
value is a numeric value.

Result: The result of this function is a numeric floating point value.

Examples: The graph below shows the result of the hyperbolic tangent function given input values from -6 to +6.



Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you use a text value with this function, for example `tanh("23")`. If you have a numeric value in a text item you must convert the text to the number data type before taking the hyperbolic tangent, for example `tanh(val("34"))`.

See Also: [sinh\(\)](#) function
[cosh\(\)](#) function
[arcsinh\(\)](#) function
[arccosh\(\)](#) function
[arctanh\(\)](#) function
[val\(\)](#) function

TEXT ARRAYS

Background: An array is a numbered collection of data items. Panorama includes a number of functions and statements that treat a single text data item as if it were a numbered collection of smaller items. The smaller text data items must be separated from each other by a delimiter, for instance a comma or carriage return.

A typical text array is shown below. Panorama would normally treat this as a single text item with a length of 40 characters. When treated as a text array, however, this text would be considered as a collection of 7 elements separated by slashes.

```
white/red/orange/yellow/green/blue/black
```

In this example, the / is the separator character. You can use any character you want for a separator character. Here is the same array, but using the • character (option-8) as the separator character.

```
white•red•orange•yellow•green•blue•black
```

It is possible to use different separator characters at different times. You could even build a multi-level array by using two different separator characters.

Since text arrays are a secondary data type, it is up to you to keep track of the fact that you are using an array and what the separator character is. Panorama won't stop you from trying to access the array of colors above as if it were delimited with commas instead of semicolons, but you probably won't get the results you wanted unless you use the correct separator character.

Picking a Separator

Any ASCII character can be used as a separator character, so you have 256 possible choices. Common separators include commas, semicolons, slashes, carriage returns, spaces and tabs.

It's important to pick a separator character that will not occur in the data elements of your array. If your data may include commas, don't use the comma as a separator character. If the data might include carriage returns, don't use a carriage return. If you want to be extra sure to avoid conflicts, pick a non-printing character. You can use the chr(function to generate non-printing characters, for example chr(1), chr(2), chr(3). Most chr values below 32 are non-printing except for chr(9) and chr(13), which correspond to tab and carriage return.

Some Panorama user interface elements and functions use text arrays as parameters or to hold a list of values. For these applications the separator character is usually required to be a carriage return. For example, the Pop-Up Menu SuperObject uses a carriage return delimited array to define the list of pop-up menu choices. The lookupall(function extracts information from another database and places it into an array with whatever separator you specify. Consult the documentation for each individual statement, function or SuperObject to see the exact specifications for any arrays they may use.

You can easily change the separator character of an array with the replace(function. The assignment statement below changes the separator character of the array Elements from semicolons to carriage returns.

```
Elements=replace(Elements,";",␣)
```

See Also:

[array\(\)](#) function
[arraysize\(\)](#) function
[arrayrange\(\)](#) function
[arraychange\(\)](#) function
[arraydelete\(\)](#) function
[arrayelement\(\)](#) function
[arrayinsert\(\)](#) function
[arrayscan\(\)](#) function
[arrayreverse\(\)](#) function
[arraysearch\(\)](#) function
[arraystrip\(\)](#) function
[extract\(\)](#) function
[arraysort](#) statement
[arrayfilter](#) statement
[arraydeduplicate](#) statement
[arraybuild](#) statement
[arrayselectedbuild](#) statement
[arraylinebuild](#) statement
[chr\(\)](#) function

Text Editor Programming

Background: The `superobject` and `activesuperobject` statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

Quick Reference:

```
"Open"
"Close"
"Cut"
"Copy"
"Paste"
"Clear"
"GetSelection", <START>, <END>
"SetSelection", <START>, <END>
"GetText", <TEXT>
"SetText", <TEXT>
"InsertText", <TEXT>
"GetSelectedText", <TEXT>
"Find"
"FindNext"
"Change"
"Spell"
"SetScroll", <VERTICAL>, <HORIZONTAL>
"GetScroll", <VERTICAL>, <HORIZONTAL>
"GetLineCount", <COUNT>
```

Open This command opens the SuperObject for editing, if it is not already active. This command is the equivalent of clicking on the object to start editing it. Since the object isn't active yet, you can't use the `activesuperobject` statement. The example below opens the Memo field.

```
superobject "Memo", "Open"
if info("activesuperobject")≠"Memo"
    beep
    stop
endif
```

If another data cell or SuperObject is currently active, it's possible that Panorama won't be able to open the SuperObject. If there is an error while attempting to close the currently active item (for example, incorrect date format or an illegal character in a number), the user may choose to cancel and re-edit the incorrect data. The example above checks to make sure that the SuperObject has really been opened for editing—if not, the procedure beeps and stops.

Close This command closes the SuperObject. This is equivalent to pressing the **Enter** key.

```
activesuperobject "Close"
if info("activesuperobject")≠"
    stop /* whoops, could not close because of an error */
endif
```

If there was an error in the data that was being edited (for example, incorrect date format or an illegal character in a number), the user may choose to cancel and re-edit the incorrect data. The procedure above checks for this and stops if this happens.

Cut This command copies the currently selected text to the clipboard, then deletes the selected text. This is the same as choosing **Cut** from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)

Copy This command copies the currently selected text to the clipboard, but does not delete the text. This is the same as choosing **Copy** from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)

Paste This command pastes the text in the clipboard into the text being edited. The new text will replace any currently selected text, or the text will be inserted at the current insertion point if no text is currently selected. This is the same as choosing **Paste** from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)

Clear This command deletes the selected text (without copying it to the clipboard). This is the same as choosing **Clear** from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)

GetSelection ,<START>,<END>

This command gets the start and end points of the currently selected text. For the purpose of the GetSelection command (and the SetSelection command) each character is numbered, starting with zero in front of the first character. For example, if the first character was currently selected, GetSelection will return 0 and 1. If the 3rd through 8th characters are currently selected, GetSelection will return 2 and 8. If there is currently an insertion point, the starting and ending point will be the same. This command only returns the position of the selected text; if you want to get the text itself, use the GetSelectedText command.

The example procedure below counts and displays the number of characters selected.

```

local SelStartPoint,SelEndPoint
SelStartPoint=0
SelEndPoint=0
if info("activesuperobject")≠" "
    activesuperobject "GetSelection",SelStartPoint,SelEndPoint
endif
message str(SelEndPoint-SelStartPoint)+" characters selected"

```

This procedure checks to make sure that a SuperObject is active. If there is no SuperObject active, it will display the message 0 characters selected. If the procedure did not check, Panorama would stop the procedure and display an error message if there was no active SuperObject.

SetSelection ,<START>,<END>

This command allows you to change the selection area. It is equivalent to clicking or dragging on the text to select it. For the purpose of the SetSelection command (and the GetSelection command), each character is numbered, starting with zero in front of the first character. For example, the procedure below would put the insertion point in front of the first character in the text.


```

if info("activesuperobject")≠" "
  activesuperobject "SetSelection",0,0
endif

```

The next example will select all of the text. Notice that the end position may be past the end of the text...Panorama will automatically adjust this for you.

```

if info("activesuperobject")≠" "
  activesuperobject "SetSelection",0,32768
endif

```

Here's a similar example that places the insertion point at the end of the text.

```

if info("activesuperobject")≠" "
  activesuperobject "SetSelection",32768,32768
endif

```

This final example will increase the length of the current selection by one character.

```

local SelStartPoint,SelEndPoint
SelStartPoint=0
SelEndPoint=0
if info("activesuperobject")≠" "
  activesuperobject "GetSelection",SelStartPoint,SelEndPoint
  SelEndPoint=SelEndPoint+1
  activesuperobject "SetSelection",SelStartPoint,SelEndPoint
endif

```

GetText

,<TEXT>

This command gets all of the text being edited and puts it in a variable you specify. (Note: If you want only the selected text, use the GetSelectedText command.) The example procedure below searches for text in chevrons («») and if found, selects it. Using this procedure you could create templates with blanks to be filled in, for example ...«Gallery»...«Artist»...«Title». (Of course it might be better to store this information in fields and merge it into the text with a formula.)

```

local someText,selStart,selEnd
if info("activesuperobject") = " " stop endif
activesuperobject "GetText",someText
selStart=search(someText,"«")
selEnd=search(someText,"»")
if selStart>0
  selStart=selStart-1
endif
if selEnd<selStart
  selEnd=selStart+1
endif
activesuperobject "SetSelection",selStart,selEnd

```

SetText

,<TEXT>

This command replaces the text currently being edited with completely new text! This is a very powerful command.

Here is a very simple example that simply erases all of the text. This is similar to Clear, except that all the text is erased, not just the selected text.

```
if info("activesuperobject")= "" stop endif
activesuperobject "SetText", ""
```

The next example adds a new line with a date and time stamp to the currently edited text. It also moves the insertion point to the end of the new time and date stamp, so the user can immediately type in a note.

```
local someText
if info("activesuperobject") = "" stop endif
activesuperobject "GetText",someText
activesuperobject "SetText",someText+¶+
    datepattern( today(), "mm/dd/yy")+ " @"+
    timepattern( now(), "hh:mm am/pm")+ " - "
activesuperobject "SetSelection",32768,32768
```

InsertText

,<TEXT>

This command inserts text. The new text replaces the currently selected text, or is inserted at the insertion point if no text is selected. The example below inserts the current time into the text.

```
if info("activesuperobject") = "" stop endif
activesuperobject "InsertText",
timepattern( now(), "hh:mm am/pm")
```

GetSelected-Text

,<TEXT>

This command gets the selected text and puts it into a variable. The example below uses this command to change the case of the selected text. Each time the procedure is used the case will toggle: if the text is all lower case, it will be converted to initial caps; if it is initial caps, it will be converted to all upper case; otherwise it will be converted to all lower case.

```
local someText
if info("activesuperobject") = "" stop endif
ActiveSuperObject "GetSelectedText",someText
case someText=lower(someText)
    someText=upperword(someText)
case someText=upperword(someText)
    someText=upper(someText)
defaultcase
    someText=lower(someText)
endcase
activesuperobject "InsertText",someText
```

Find

This command displays a dialog asking the user what they would like to find, then locates the word or phrase within the text being edited. This is the same as using the Find in Cell command in the Edit Menu.

Another way to find is to use the [search\(\)](#) function. For an example of this, see the GetText command earlier in this section.

FindNext This command locates the next occurrence of the word or phrase searched for with the Find command. This is the same as using the **Find Next in Cell** command in the Edit Menu.

Change This command displays a dialog asking the user what they would like to change, then changes every occurrence it finds in the text being edited. This is the same as using the **Change in Cell** command in the Edit Menu.

Another way to change is to use the GetText command and the `replace()` function. The example below replaces the initials `rdp` with `Robert D. Bryce`, then moves the insertion point to the end of the text.

```
local someText
if info("activesuperobject") = "" stop endif
activesuperobject "GetText",someText
activesuperobject "SetText",replace(someText,"rdp","Robert D. Bryce")
activesuperobject "SetSelection",32768,32768
```

Spell This command locates the next misspelled word in the text being edited. This is the same as using the **Spelling** command in the Edit Menu.

SetScroll ,<VERTICAL>,<HORIZONTAL>

This command scrolls the text, just like dragging on the scroll bar. The scrolling is specified in pixels (one pixel equals 1/72 inch). The example below scrolls to the top of the text.

```
activesuperobject "setscroll",0,0
```

GetScroll ,<VERTICAL>,<HORIZONTAL>

This command gets the current amount of scrolling in both the vertical and horizontal directions. The parameters should be variables into which the values will be placed. The example below scrolls down one inch:

```
local vScroll,hScroll
activesuperobject "getscroll",vScroll,hScroll
activesuperobject "setscroll",vScroll+72,hScroll
```

GetLineCount ,<COUNT>

This command calculates the actual number lines in the text being edited, taking automatic line wraps into account. For example, even if the text has no carriage returns in it, the text may wrap over several lines. The GetLineCount command will return the actual number of lines, not the number of carriage returns.

```
local LCount
activesuperobject "getlinecount",LCount
```

TEXT255(...)

Syntax: TEXT255(binarydata)

Description: The `text255()` function converts binary data containing a Pascal String into regular text. A Pascal String is a special text format that is sometimes used by the Macintosh ROM's (also called a String255 or Str255 because the text is limited to a maximum length of 255 characters). (Pascal is the name of a computer language, which in turn is named after a famous mathematician.)

Parameters: This function has one parameter: `binarydata`.
`binarydata` is a text data item containing a Pascal String.

Result: This function returns the text equivalent of the Pascal String passed to it.

Examples: See [c/pascal structures](#) for examples of the `text255()` function.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to pass a number as the Pascal String.

See Also: [string255\(\)](#) function
[byte\(\)](#) function
[word\(\)](#) function
[longword\(\)](#) function
[radix\(\)](#) function
[radixstr\(\)](#) function
[c/pascal structures](#)

TEXTDISPLAY(...)

Syntax: TEXTDISPLAY(color,style)

Description: The `textdisplay(` function works with Text Display SuperObjects™. By using this function as the first part of the formula in a Text Display SuperObject™ you can control the color and style of the text on the fly. For example, you can automatically display all negative numbers in red. (Advanced note: The `textdisplay(` function actually generates a special header that is intercepted and removed by the Text Display SuperObject™. The header contains information the Text Display SuperObject™ uses to select the style and color.)

Parameters: This function has two parameters: color and style.

color is the color that should be used to display the text. See the `rgb(` function. If you pass "" for this parameter the text will be displayed in the normal color for this object.

style is the style or combination styles that should be used to display the text. For a single style by itself simply use the name of the style: "Plain", "Bold", "Italic", "Underline", "Outline" or "Shadow". If you want to combine multiple styles together you must specify the style numerically. Add up the numbers for the styles you want from the table listed below. For example, for bold italic text the style should be 3.

```
0 Plain
1 Bold
2 Italic
4 Underline
8 Outline
16 Shadow
```

Examples: Here's an example formula which displays negative Balance values in red. (Remember, this formula only works in a Text Display SuperObject™.)

```
textdisplay(rgb(?(Balance<0,65535,0),0,0),0)+pattern(Balance,"$#,##")
```

This example formula displays the Response field in italics if the Priority is Hot.

```
textdisplay(" ",?(Priority="Hot","italic","plain"))+Response
```

This slightly revised example displays the Response field in bold-italic if the Priority is Hot. It sets the style to 3 if the Priority is Hot (1=bold, 2=italic, 1+2=bold italic).

```
textdisplay(" ",?(Priority="Hot",3,0))+Response
```

Errors: This function does not generate any errors. However, if the parameters are not set up properly, the Text Display SuperObject™ will display the text using the standard color and style.

See Also: [rgb\(](#) function
[colors](#)

TEXTSTUFF(...)

Syntax: TEXTSTUFF(maintext,newtext,position)

Description: The `textstuff()` function replaces one or more characters in the middle of a piece of text

Parameters: This function has three parameters: `maintext`, `newtext`, and `position`.

maintext is the original text data item that contains one or more characters you want to replace.

newtext is the new text that you want to use to replace characters in the original text data item.

position is the location within the original text where you want to replace text. The position is a number starting with zero.

Result: This function returns a copy of the original text item with one or more characters replaced.

Examples: This example replaces two characters in a 24 character text item.

```
TEMP=textstuff("Temperature: 87 degrees"," 92",13)
```

The operation of this formula is shown in the table below.

```
Original Text: Temperature: 87 degrees
New Text: 92
Position: 13
-----
Result (TEMP): Temperature: 92 degrees
```

If the new text is positioned beyond the end of the original text, the characters in between are undefined.

```
TEMP=textstuff("Temp: "," 92",13)
```

The operation of this formula is shown in the table below.

```
Original Text: Temp:
New Text: 92
Position: 13
-----x-----
Result (TEMP): Temp:\#0020f 92
```

The characters in between (in yellow) may be anything. (Of course you could use another `textstuff()` function to fill them in, or you could add characters to the original text before using `textstuff()` in the first place.)

See [c/pascal structures](#) for additional examples of the `textstuff()` function.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a number for the `maintext` or `newtext` parameters.

Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a number for the `position` parameter.

See Also:

[chr\(function](#)
[byte\(function](#)
[word\(function](#)
[longword\(function](#)
[c/pascal structures](#)

TIME(...)

Syntax: TIME(text)

Description: The time(function converts text into a number representing a time. See also [seconds\(](#).

Parameters: This function has one parameter: text.

text is the text that you want to convert to a number representing a time. The text must contain a valid time. The time function allows you to leave out the colons in the time, and also allows you to leave off the am/pm. Here are some examples of valid times:

```
4:13 PM
11:00 AM
2:30
18:45
230
4p
midnight
noon
morning
afternoon
evening
night
nite
```

The [seconds\(](#) function also converts text into a number, but is more strict about the time formats it will accept.

Result: This function returns a number representing the time. The number is the number of seconds since midnight. For example, if the time is 10:23 AM this function will return the number 37,380.

Examples: The time(function is very lenient about the format you use to enter the time. It will accept a time without colons, for example 425 pm instead of 4:25 pm. If there is no am or pm the time function will try to make an intelligent guess. For example, 230 is almost certainly 2:30 pm, not 2:30 am. By default, the time(function assumes that any time from 6:00 to 11:59 is AM, and any time from 12:00 to 5:59 is PM, but you can change these assumptions with the [timedefaults](#) statement described below.

The time(function will also convert “named” times: noon, midnight, morning, afternoon, evening, and night. This function assumes that morning is 9:00 am, afternoon is 1:00 pm, evening is 6:00 pm, and night is 10:00 pm. These assumptions can be changed with the [timedefaults](#) statement. This statement has five parameters: the earliest hour that should default to AM, and the times for morning, afternoon, evening, and night. Here is a procedure that uses the time(function to help locate flights departing in a ±3 hour period from a specified time of day.

```
local xTime
timedefaults 7,time("7a"),time("3p"),time("8p"),time("1a")
xTime=""
getttext "Select flights around what time?",xTime
select DepartureTime>time(xTime)-3*3600 and
    DepartureTime<time(xTime)+3*3600
```


Since the first parameter of the `timedefaults` statement is 7, any time from 7:00 to 11:59 will default to AM unless the user types PM, and any time from 12:00 to 6:59 will default to PM unless the user types AM. If the user types morning the procedure will select all flights from 4:00 am to 10:00 am (within three hours of 7:00 am). If the user types evening the procedure will select all flights from 5:00 pm to 11:00 pm (within three hours of 8:00 pm).

If the `time()` function is supplied with text it cannot interpret as a time, the procedure will stop and an error message is displayed. You can trap this error with the `if error` statement and handle it yourself within the procedure, as you can see in this example.

```

local xTime,depTime
timedefaults 7,time("7a"),time("3p"),time("8p"),time("1a")
xTime=""
gettext "Select flights around what time?",xTime
depTime=time(xTime)
if error
message "Invalid time...all flights will be displayed."
selectall
else
select DepartureTime>=depTime-(3*3600) and
       DepartureTime<=depTime+(3*3600)
endif

```

This example has another advantage: the `select` statement will run slightly faster. This is because it does not have to convert the text to seconds over and over again for each record in the database. Note: This example will not work properly if the user selects a time within 3 hours of midnight. See [timedifference\(\)](#) and [timeinterval\(\)](#) for solutions to this problem.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to convert a numeric value.

Illegal time. This error occurs if the text does not contain a valid time.

See Also: [seconds\(\)](#) function
[timepattern\(\)](#) function
[now\(\)](#) function
[timedefaults](#) statement

TIME24(...)

Syntax: TIME24(TIME)

Description: The **time24()** function takes a time and makes sure it falls within a 24 hour period. If the time is less than 24 hours, it is unchanged. If the time is greater than 24 hours, it is converted to the equivalent time in a 24 hour period (for example 30:00:00 is converted to 6:00:00).

Parameters: This function has one parameter: **time**.

time is a time (number of seconds) that may be greater than 23 hours, 59 minutes, 59 seconds

Result: This function returns a time between 0 and 86,399 seconds (23:59:59). If the original time is greater than this value, the time is reduced in 24 hour increments until it is smaller than 24 hours.

Examples: The **time24()** function can help with calculations of an ending time from a start time and duration. The basic formula for such a calculation is shown here.

```
EndTime=StartTime+Duration
```

This formula works fine unless the interval extends over midnight. The **time24()** function adjusts the result to make sure it starts over at zero as it crosses midnight.

```
EndTime=time24(StartTime+Duration)
```

This formula will correctly calculate that 10:30 PM + 4 hours is 2:30 AM.

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the time parameters. You must convert text values to numbers with the [seconds\(\)](#) or [time\(\)](#) function before they can be used with this function.

See Also: [timeinterval\(\)](#) function
[timedifference\(\)](#) function
[seconds\(\)](#) function
[time\(\)](#) function
[now\(\)](#) function

TIMEDEFAULTS

- Syntax:** `TIMEDEFAULTS default12,morning,afternoon,evening,nite`
- Description:** The `timedefaults` statement sets the defaults for the [time\(\)](#) function.
- Parameters:** This statement has five parameters: `default12`, `morning`, `afternoon`, `evening` and `nite`.
- default12** is an integer specifying the first hour of the default 12 hour period if no am/pm is supplied. For example, if you want the default period to be 8am to 8pm this parameter should be 8.
- morning** is the time that should be used if the text `morning` is passed to the [time\(\)](#) function.
- afternoon** is the time that should be used if the text `afternoon` is passed to the [time\(\)](#) function.
- evening** is the time that should be used if the text `evening` is passed to the [time\(\)](#) function.
- nite** is the time that should be used if the text `night` (or `nite`) is passed to the [time\(\)](#) function.
- Action:** This statement has no immediate effect, but changes the way the [time\(\)](#) function behaves.
- Examples:** The procedure below uses the [time\(\)](#) function to help locate flights departing in a ± 3 hour period from a specified time of day. If the user enters `evening` the procedure will select flights from 5pm to 11pm, for `afternoon` flights from noon to 6pm will be selected.
- ```

local xTime
timedefaults 7,time("7a"),time("3p"),time("8p"),time("1a")
xTime=""
getttext "Select flights around what time?",xTime
select DepartureTime>time(xTime)-3*3600 and
DepartureTime<time(xTime)+3*3600

```
- Views:** This statement may be used in any view.
- See Also:** [time\(\)](#) function  
[seconds\(\)](#) function  
[timepattern\(\)](#) function  
[now\(\)](#) function

# TIMEDIFFERENCE(...)

**Syntax:** `TIMEDIFFERENCE(startTime,endTime)`

**Description:** The `timedifference()` function calculates the difference between two times. It works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between -12 and +12 hours. See also the [timeinterval\(\)](#) function, which returns a time interval between 0 and 24 hours.

**Parameters:** This function has two parameters: `startTime` and `endTime`.

**startTime** is a number (number of seconds) representing the starting point of the time interval.

**endTime** is a number (number of seconds) representing the ending point of the time interval.

**Result:** This function returns the number of seconds between the two times. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the difference would be 4:35. But if the parameters are reversed and the start time is 2:05 AM and the end time is 9:30 PM, the difference is -4:35. If the result is positive, the `endTime` is after the `startTime`. But if the result is negative, the `startTime` is after the `endTime`.

**Examples:** This sample procedure selects flights that depart within  $\pm 3$  hours of a specified time. By using the `timedifference()` function this procedure is able to work correctly even if the specified time is near midnight.

```
local xTime
timedefaults 7,time("7a"),time("3p"),time("8p"),time("1a")
xTime=""
getttext "Select flights around what time?",xTime
select
abs(timedifference(DepartureTime,time(xTime)))
```

The result of the `timedifference()` function is positive if the second time is later than the first time, and negative if the first time is later. The example uses the `abs()` function to make the difference always positive, making the comparison simpler.

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for either of the time parameters. You must convert text values to numbers with the [seconds\(\)](#) or [time\(\)](#) function before they can be used with this function.

**See Also:** [seconds\(\)](#) function  
[time\(\)](#) function  
[now\(\)](#) function

# TIMEINTERVAL(...)

**Syntax:** `TIMEINTERVAL(startTime,endTime)`

**Description:** The `timeinterval()` function calculates the time interval between two times. It works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between 0 and 24 hours. See also the [timedifference\(\)](#) function, which returns a time interval between -12 and +12 hours.

**Parameters:** This function has two parameters: `startTime` and `endTime`. `startTime` is a number (number of seconds) representing the starting point of the time interval. `endTime` is a number (number of seconds) representing the ending point of the time interval.

**Result:** This function returns the number of seconds in the interval between the two times. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the interval would be 4:35. But if the parameters are reversed and the start time is 2:05 AM and the end time is 9:30 PM, the interval is 19:25.

**Examples:** This example calculates the flight time of an airplane flight. This procedure will work correctly even if the flight leaves in the evening and arrives the next morning.

```

local fltHours,fltMinutes,fltSeconds
fltSeconds=timeinterval(ArrivalTime,DepartureTime)
fltMinutes=fltSeconds/60
fltHours=fltMinutes/60
fltMinutes=fltMinutes mod 60
message "Flight time from "+OriginCity+" to "+
DestinationCity+" is "+
 str(fltHours)+" hours "+str(fltMinutes)+" minutes"

```

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for either of the time parameters. You must convert text values to numbers with the [seconds\(\)](#) or [time\(\)](#) function before they can be used with this function.

**See Also:** [timeinterval\(\)](#) function  
[seconds\(\)](#) function  
[time\(\)](#) function  
[now\(\)](#) function

# TIMEPATTERN(...)

**Syntax:** TIMEPATTERN(number,pattern)

**Description:** The `timepattern()` function converts a number representing a time into text. The function uses a pattern to control how the date is formatted.

**Parameters:** This function has two parameters: `number` and `pattern`.

**number** is the number that you want to convert to text. This number is treated as the number of seconds since midnight. (the `seconds()` and `time()` functions can convert text into such a number).

**pattern** is text that contains a pattern for formatting the date. The pattern is assembled from four components: hh (hours), mm (minutes) ss (seconds), and am/pm. Some of the more common time patterns are listed here:

| Pattern          | Typical Output |
|------------------|----------------|
| "hh:mm:ss am/pm" | 4:32:17 pm     |
| "hh:mm am/pm"    | 4:32 pm        |
| "hh:mm:ss"       | 16:32:17       |

If am/pm is left off the pattern the time will be formatted in 24 hour format, as shown on the last line of the table above. You should also leave off am/pm for converting elapsed times.

**Result:** This function returns an item of text containing the formatted date.

**Examples:** Here's a simple example that displays a flight arrival time.

```
message "Flight "+FlightNumber+" from "+
OriginCity+" is due at "+
timepattern(ArrivalTime,"hh:mm am/pm")
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the pattern parameter.

**Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the number parameter.

**See Also:** [seconds\(\)](#) function  
[time\(\)](#) function  
[now\(\)](#) function

# TODAY(...)

**Syntax:** TODAY()

**Description:** The `today()` function returns a number corresponding to today's date.

**Parameters:** This function has no parameters

**Result:** This function calculates today's date. The date is represented as a number which is the number of days since January 1, 4713 B.C. For example, if today's date is August 7, 1991 this function will return the number 2,448,476.

**Examples:** This example procedure selects records that have been printed today (assuming that `PrintDate` is a date field).

```
select PrintDate=today()
```

This formula could be used in an auto-wrap text object or Text Display SuperObject™ to display today's date.

```
datepattern(today(),"Month ddnth, yyyy")
```

**Errors:** \*\*\* This function does not produce any errors. \*\*\*

**See Also:** [date\(\)](#) function  
[datepattern\(\)](#) function  
[now\(\)](#) function

# TODO

- Syntax:** `TODO reminder,status`
- Description:** The **todo** statement allows a procedure to change the status (incomplete vs. complete) of a to-do reminder (see [reminder data](#)).
- Parameters:** This statement has two parameters: **reminder** and **status**.
- reminder** is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. See [reminder data](#) for detailed information about reminders. In this case you are not specifying the reminder itself, but the field containing the reminder data.
- status** is a number which controls the completion status for this reminder. The status may be **0** (not complete), or **1** (complete) Note: A procedure can find out the current status of a to-do reminder with the [remindertodo\(\)](#) function.
- Action:** This statement changes the status of a to-do reminder. It is ignored for appointment reminders.
- Examples:** This example marks all reminders for [Frank Johnson](#) as completed.
- ```

find Contact = "Frank Johnson"
loop
  stoploopif (not info("found"))
  todo Reminders,1
next
while forever

```
- Views:** This statement may be used in a Data Sheet or Form view.
- See Also:** [remindertodo\(\)](#) function
[reminder\(\)](#) function
[reminderdate\(\)](#) function
[remindertime\(\)](#) function
[remindertype\(\)](#) function
[buildreminder](#) statement
[alarmedit](#) statement
[reminder data](#)

TODOPRIORITY

- Syntax:** `TODOPRIORITY reminder,priority`
- Description:** The `todopriority` statement allows the procedure to change the priority (hi-medium-low) of a to-do reminder (see [reminder data](#)).
- Parameters:** This statement has two parameters: `reminder` and `priority`.
- reminder** is a special data type that holds scheduling information about an appointment or to-do item. Reminders are usually used in calendar database applications. See [reminder data](#) for detailed information about reminders. In this case you are not specifying the reminder itself, but the field containing the reminder data.
- priority** is a number which describes how important this reminder is. The priority may be **0** (blank), **1** (low), **2** (medium), or **3** (high). Note: A procedure can find out the current priority of a to-do reminder with the [reminderpriority\(\)](#) function.
- Action:** This statement changes the priority of a to-do reminder. It is ignored for appointment reminders.
- Examples:** This example changes all reminders for Frank Johnson to high priority.
- ```

find Contact = "Frank Johnson"
loop
 stoploopif (not info("found"))
 todopriority Reminders,3
next
while forever

```
- Views:** This statement may be used in a Data Sheet or Form view.
- See Also:** [reminderpriority\(\)](#) function  
[reminder\(\)](#) function  
[reminderdate\(\)](#) function  
[remindertime\(\)](#) function  
[remindertype\(\)](#) function  
[buildreminder](#) statement  
[todo](#) statement  
[alarmedit](#) statement  
[reminder data](#)

# TOTAL

**Syntax:** TOTAL

**Description:** The **total** statement calculates totals and subtotals for the current field.

**Parameters:** This statement has no parameters.

**Action:** This statement calculates totals for the current field. The current field must be numeric. If the database contains summary records, this statement will calculate subtotals for each summary record, along with an overall total for the whole database. If there are not any summary records in the database, one will be added at the end of the database and the overall total calculated and placed into the summary record. This statement has the same effect as choosing the **Total** command in the **Math** menu.

**Examples:** This simple example calculates the total of all checks in the database.

```
field Debit
total
```

This example calculates the total sales for each state, along with the overall grand total.

```
field State
group
field Sales
total
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [average](#) statement  
[sum\(\)](#) function  
[count](#) statement  
[minimum](#) statement  
[maximum](#) statement  
[group](#) statement  
[outlinelevel](#) statement

# UNDEFINE

**Syntax:** UNDEFINE variables

**Description:** The **undefine** statement destroys one or more variables.

**Parameters:** This statement has one parameter: variables.

**variables** is a list of variables to be destroyed. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

**Action:** This statement destroys one or more variables. The variables may have been created with the [local](#), [windowglobal](#), [fileglobal](#) or [global](#) statements, or any combination. Destroying a variable completely removes it as if it had never been created in the first place.

**Examples:** The example destroys two variables, [Counter](#) and [Operating Ratio](#).

```
undefine Counter, «Operating Ratio»
```

Before you undefine a permanent variable you should make it unpermanent, like this:

```
unpermanent timeStamp
undefine timeStamp
```

**Views:** This statement may be used in any view.

**See Also:** [global](#) statement  
[fileglobal](#) statement  
[local](#) statement  
[globalize](#) statement  
[windowglobal](#) statement  
[permanent](#) statement  
[unpermanent](#) statement

# UNDO

**Syntax:** UNDO

**Description:** The **undo** statement reverses the effect of the previous statement. Not all statements can be reversed, however you can undo the effect of sort, select, and fill statements. The **undo** statement would not normally be used in a procedure. However, it may be recorded in a procedure if the user chooses the **Undo** item in the Edit menu. If this happens you should remove the **undo** statement and the procedure statement from the procedure, they will simply waste time.

**Parameters:** This statement has no parameters.

**Examples:** The examples below shows an undo recorded by the procedure recorder. In effect the first three lines of this procedure don't do anything, and they should probably be removed.

```
field "Price"
sortup
undo
field "Name"
sortup
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [noundo](#) statement

# UNIONRECTANGLE(...)

**Syntax:** UNIONRECTANGLE(rectangle1,rectangle2)

**Description:** The `unionrectangle()` function creates a rectangle by combining two rectangles. The new rectangle is large enough to exactly cover both of the input rectangles. A rectangle is 8 bytes or raw binary data (see [binary data](#), [graphic coordinates](#)).

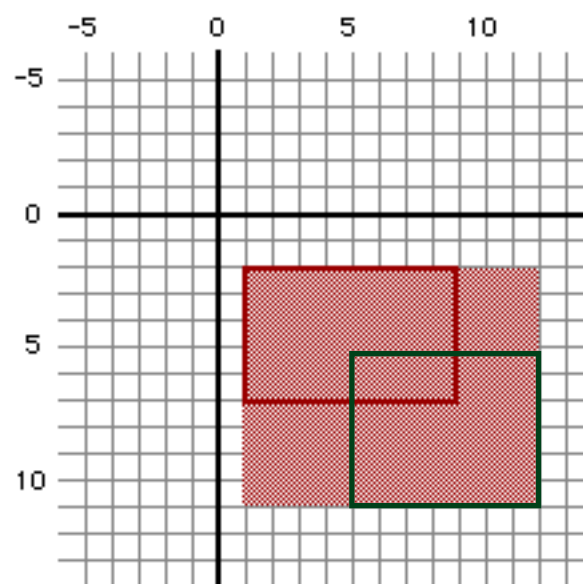
**Parameters:** This function has two parameters: `rectangle1` and `rectangle2`.

**rectangle1** is the first rectangle.

**rectangle2** is the second rectangle.

**Result:** This function returns a rectangle that is just large enough to cover both of the input rectangles.

**Examples:** The illustration below shows how this function combines two rectangles, creating a third rectangle where the original two rectangles overlap:



The `unionrectangle()` function can be used to check if a smaller rectangle is inside a larger rectangle. If the smaller rectangle is completely inside the larger rectangle, the result of this function will be the same as the larger rectangle. The procedure below checks to see if the current window is completely inside the main screen, or if it partially extends past the edge of the main screen.

```
if unionrectangle(
 info("windowrectangle"),
 info("screenrectangle")
)≠info("screenrectangle")
message "This window sticks out from the main screen"
endif
```

**Errors:** **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for any of the four parameters.

**See Also:**

[point\(\)](#) function  
[rectangle\(\)](#) function  
[rectanglesize\(\)](#) function  
[rtop\(\)](#) function  
[rbottom\(\)](#) function  
[rleft\(\)](#) function  
[rright\(\)](#) function  
[rheight\(\)](#) function  
[rwidth\(\)](#) function  
[inrectangle\(\)](#) function  
[intersectionrectangle\(\)](#) function  
[info\("screenrectangle"\)](#) function  
[info\("windowrectangle"\)](#) function  
[info\("buttonrectangle"\)](#) function  
[info\("cursorrectangle"\)](#) function

# UNIQUEID(...)

**Syntax:** UNIQUEID(field,root)

**Description:** The **uniqueid()** function is designed for generating unique ID codes for each record in a database. The function generates ID codes with a text root and a numeric suffix (for example Jeff261). By using the machine name as the text root you can guarantee that the ID will be unique even for multiple copies of the database on different machines.

**Parameters:** This function has one parameters: **field** and **root**.

**field** is the name of the field that will contain the ID code. The function needs to know the name of this field so that it can scan the field to find an ID code that has not been used yet. The field name should be surrounded by quotes. For example, if the name of the field is ID, you should use "ID" as the parameter.

**root** is the text root that the ID code will be based on. This root may contain any kind of character, but it should not end with a numeric digit. To get a root that will be unique for each different computer you have, use the **info("user")** function for the root. This function returns the user name specified in the Sharing Setup control panel.

**Result:** The function returns a unique ID code based on the root you have supplied.

**Examples:** Although you may find other uses for it, the **uniqueid()** function was designed specifically for creating unique Smart Merge serial numbers. Whenever a new record is added to a database that supports Smart Merge you must make sure that the ID and Modified fields are filled in. The best way to do this is to add a **.AddRecord** automatic procedure to your database. The two lines shown below will fill in the proper values.

```
Modified=superdate(today(),now())
ID=uniqueid("ID",info("user"))
```

The **uniqueid()** function will scan the ID field to find the next serial number available. For example, if you are using a computer with a chooser name of Sam and the highest Sam serial number is 296, the **uniqueid()** function will return the value Sam297. Creating an **.AddRecord** procedure may not be enough to insure that the ID and Modified fields are always filled in. If your database has procedures that create new records, the **.AddRecord** procedure will not automatically be called. You must modify these procedures to call the **.AddRecord** procedure (using the call statement). Another possible problem area is imported data. When you import data into the database you must make sure that the ID and Modified fields are filled in. The procedure listed below will do the job. You should also run this procedure when you first add Smart Merge to your database, so that all your existing data will be properly identified.

```
select ID=" "
field Modified
formulafill superdate(today(),now())
field ID
formulafill uniqueid("ID",info("user"))
selectall
```

**Errors:** **Field or variable does not exist.** This error occurs if there is no field in the current database with the name you have specified. You probably misspelled the field name.

**See Also:** [info\("user"\) function](#)



# UNLOCKRECORD

**Syntax:** UNLOCKRECORD

**Description:** The `unlockrecord` statement unlocks the currently active record. This statement only applies to Partner/Server databases. It is ignored when used with a standard Panorama database.

**Parameters:** This statement has no parameters.

**Action:** This statement unlocks the current record. If the current record is not locked, this statement does nothing. (Note: The current record is automatically unlocked when Panorama moves to another record.)

**Examples:** The first line of this example automatically locks the record. Since this procedure is not planning to make any further modifications to the record, it immediately unlocks it so that other users can access the record.

```
Status="Complete" /* automatically locks record */
unlockrecord
```

**Views:** This statement may be used in the Data Sheet and Form views.

**See Also:** [lockrecord](#) statement  
[lockorstop](#) statement  
[info\("serverstatus"\)](#) function  
[info\("servertimeout"\)](#) function

# UNPERMANENT

**Syntax:** UNPERMANENT variables

**Description:** The **unpermanent** statement converts one or more permanent variables into regular file-global variables. The variables will no longer be saved as part of the database.

**Parameters:** This statement has one parameter: **variables**.

**variables** is a list of permanent variables to be converted into regular fileglobal variables. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

**Action:** This statement “destroys” one or more permanent variables. Actually, the variables still exist, but they are no longer linked to a database, and thus no longer saved with the database. To use this statement the database that contains the permanent variables must be the current database. If you want to completely destroy the variable use the [undefine](#) statement.

**Examples:** The example converts the permanent variable [EarlyDiscount](#) into a fileglobal variable.

```
unpermanent EarlyDiscount
```

**Views:** This statement may be used in any view.

**See Also:** [fileglobal](#) statement  
[global](#) statement  
[local](#) statement  
[dbinfo\(\)](#) function  
[undefine](#) statement

# UNPROPAGATE

**Syntax:** UNPROPAGATE

**Description:** The **unpropagate** statement removes repeating data from the currently selected field. If several records in a row contain the same data, this statement erases all but the first (top) value.

**Parameters:** This statement has no parameters.

**Examples:** This example removes duplicate names from a mailing list. If two records have the same name, this procedure will keep the one that was originally closer to the top of the database. (If you want to keep the one that was bottom to the top of the database, use the [propagateup](#) statement.)

```
field Name
sortup
unpropagate
select Name ≠ ""
removeunselected
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [unpropagateup](#) statement  
[propagate](#) statement  
[propagateup](#) statement  
[selectduplicates](#) statement

# UNPROPAGATEUP

**Syntax:** UNPROPAGATEUP

**Description:** The **unpropagateup** statement removes repeating data from the currently selected field. If several records in a row contain the same data, this statement erases all but the last (bottom) value.

**Parameters:** This statement has no parameters.

**Examples:** This example removes duplicate names from a mailing list. If two records have the same name, this procedure will keep the one that was originally closer to the top of the database. (If you want to keep the one that was bottom to the top of the database, use the [propagate](#) statement.)

```
field Name
sortup
unpropagateup
select Name ≠ ""
removeunselected
```

**Views:** This statement may be used in a Data Sheet or Form view.

**See Also:** [unpropagate](#) statement  
[propagate](#) statement  
[propagateup](#) statement  
[selectduplicates](#) statement

# UNTIL

**Syntax:** UNTIL true-false test

**Description:** The **until** statement is used at the end of a loop. A loop is a sequence of statements that are executed over and over again.

**Parameters:** This statement has one parameter: true-false test.

**true-false test** is a formula that should result in a true (-1) or false (0) answer. Usually the formula is created with a combination of comparison operators (=, ≠, , etc.) and boolean combinations (and, or, etc.) For example the formula `Name="Smith"` will be true if the field or variables Name contains the value Smith, and false if it contains any other value.

**Action:** The true-false test may be replaced with an integer. In that case, the loop will repeat the number of times specified by the number.

The **until** statement decides whether to jump back up to the top of the loop, or to continue with the next statement after the loop. The formula in the **until** statement is evaluated each time Panorama reaches the end of the loop. Panorama will continue looping as long as the result of the formula remains false. If the result of the formula is true, the loop stops and the procedure continues with the next statement after the **until** statement.

**Examples:** This example adds 10 new records to the current database.

```
local NewCount
NewCount=10
loop
 NewCount=NewCount-1
 addrecord
until NewCount=0
```

Here is a simpler example that also adds 10 records to the database.

```
loop
 addrecord
until 10
```

**Views:** This statement may be used in a procedure run from any view, and also works when no windows are open at all.

**See Also:** [loop](#) statement  
[while](#) statement  
[stoploopif](#) statement  
[repeatloopif](#) statement

# UPDATEWIZARDMENU

**Syntax:** UPDATEWIZARDMENU

**Description:** The `updatewizardmenu` statement tells Panorama to update the contents of the Wizard menu with the items in the variable `PanoramaWizardMenu`.

**Parameters:** This statement has no parameters.

**Action:** This statement is used by the **Wizard Manager** wizard, but it can be used by anyone. The `PanoramaWizardMenu` variable must contain a list of wizards to list in the menu, one line per wizard.

**Examples:** This example saves the contents of the wizard menu and then zaps the wizard menu.

```
global wasWizards
wasWizards=PanoramaWizardMenu
PanoramaWizardMenu=" "
updatewizardmenu
```

This example restores the wizard menu.

```
global wasWizards
PanoramaWizardMenu=wasWizards
updatewizardmenu
```

**Views:** This statement may be used in any view.

**See Also:** [global](#) statement

# Updating Database Structures

## Background:

Panorama 3.1 includes a mechanism that lets you update the structure of a database while retaining the data. Let's say that you have created a database and distributed it to many users far and wide...perhaps you are even selling the database. Your many users are each filling their databases with their own data. In the meantime, you are creating a new version. This new version of the database may have new fields, new forms, new procedures, and there are probably changes to existing forms/procedures/fields as well. Once you have finished your update you need a way to distribute the update and let each user update his or her copy of the database so that it has the new structure but retains the old data. The following paragraphs will explain how this can be done.

## ChangeName, DetachName, and HiJack

These three statements are the key to letting one database take over the structure of another. The three statements are designed to work together. Instead of attempting a detailed explanation of each statement, take a look at this example. This procedure assumes the old version of the database is currently open. The procedure allows the user to locate the update file, then updates the structure.

```

local oldFile,newFolder,newFile,newFType
/* let user locate the update file */
openfiledialog newFolder,newFile,newFType,"KAS1ZEPD"
if newFile="" stop endif /* user pressed cancel */
/* save name of original database*/
oldFile=info("databasename")
/* change name of original database IN MEMORY ONLY */
changenname oldFile+".old"
/* open the file with the new structure */
openfile folderpath(newFolder)+newFile
/* suck the data from the old file into the new structure */
openfile "&"+oldFile+".old"
/* name of new database=name of old database (IN MEMORY ONLY) */
detachname oldFile
/* now connect to the original databases file on disk */
/* it's a "filejacking"! */
hijack oldFile+".old"
/* save the new, update file */
save
/* finally close the old database - we're done with it */
window oldFile+".old:SECRET"
closefile /* this file is really gone now */

```

If you look closely at this example, you will see that it doesn't really update the structure of the original database. Instead, it loads the data from the old database into the new database using Panorama's standard "append with matching names" feature. Once this is done the old database is "detached" from its disk file. The new file then takes over or "hijacks" the detached disk file. As part of this "hijack" process Panorama also copies the auto-increment value, so if the database uses auto-numbering the numbers will continue to be generated in sequence.

## Permanent Variables

If the original database has permanent variables that you want to keep, insert the following statements just before the detachname statement in the procedure above. This procedure assumes that the new updated database has at least the same permanent variables as the original database, and it copies the values from the old database to the new.

```

local oldPermanentVariables,opv,pv
/* build a list of the permanent variables */
oldPermanentVariables=dbinfo("permanent",oldFile+".old")
opv=1
loop
/* get name of permanent variable */
pv=array(oldPermanentVariables,opv,1)
stoploopif pv=""
/* transfer value from old to new */
set pv,grabfilevariable(oldFile+".old",pv)
opv=opv+1
while forever

```

The procedure above will not work if the old database is not in author mode, since the dbinfo( function will not be able to build a list of permanent variables. In that case you must rely on your knowledge of the original database and hard code the permanent variable names, like this:

```

pAreaCode=grabfilevariable(oldFile+".old","pAreaCode")
pDialingPrefix=grabfilevariable(oldFile+".old","pDialingPrefix")
pCallingCard=grabfilevariable(oldFile+".old","pCallingCard")

```

This procedure transfers three permanent variables from the old database to the new: pAreaCode, pDialingPrefix, and pCallingCard.

### Verifying Database Identity

The procedure listed above for updating the database relies on the user to pick the correct update database. If they pick the wrong database, there will be a big problem. You can use permanent variables to create a database identity system that will permanently identify a database, even if it has been renamed. We recommend creating three permanent variables with the names dbVendor, dbName and dbVersion. Here is an example showing how these variables can be created in the .Initialize procedure.

```

permanent dbVendor,dbName,dbVersion
dbVendor="ProVUE Development"
dbName="Power Team Phone Book"
dbVersion="2.0"

```

Once these variables have been created they can be used to verify the identity of a database. In the database update routine you can add verification code in between the two openfile statements. This verification code will stop the update if the user selected the wrong database.

```

if dbVendor#grabfilevariable(oldFile+".old",dbVendor) or
dbName#grabfilevariable(oldFile+".old",dbName)
message "Please pick another database. "+
"The file you picked is not an update for "+oldFile+"."
closefile /* close the bogus update file */
changenname oldFile /* and restore the original name */
endif

```

You could even make this procedure more robust by adding a check to make sure that the version number of the update file is newer than the version number of the old file.



# UPPER(...)

**Syntax:** `UPPER(text)`

**Description:** The `upper()` function converts text to UPPER CASE (all caps).

**Parameters:** This function has one parameter: `text`.  
`text` is the item of text that you want to force to all upper case.

**Result:** The result of this function is always a text item.

**Examples:** This function can be used to modify fields or variables, or to display data. This example makes sure that every state abbreviation is capitalized, i.e. `CA` not `Ca` or `ca`.

```
field State
formulafill upper(State)
```

For example, you might use this procedure after you imported data that was not properly capitalized. Another handy use for this function is to make comparisons when you don't know how the data is capitalized. The procedure below will select all data where the terms are `NET 30`, `Net 30`, or `net 30`.

```
select upper(Terms)="NET 30"
```

The table below shows how the `upper()` function affects various items of text:

```
upper("John Smith") → JOHN SMITH
upper("NET 30") → NET 30
upper("NEW York") → NEW YORK
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `upper(34)`. If you have a number you must convert the number to text before using it with this function, for example `upper(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

**See Also:** [upperword\(\)](#) function  
[lower\(\)](#) function

# UPPERWORD(...)

**Syntax:** UPPERWORD(text)

**Description:** The `upperword()` function converts text to Initial Caps. In other words, the first letter of each word is changed to upper case, and all other letters are changed to lower case.

**Parameters:** This function has one parameter: `text`.  
`text` is the item of text that you want to force to word capitalization.

**Result:** The result of this function is always a text item.

**Examples:** This function can be used to modify fields or variables, or to display data. This example makes sure that every city name is properly capitalized, i.e. San Francisco not SAN FRANCISCO or san francisco.

```
field City
formulafill upperword(City)
```

For example, you might use this procedure after you imported data that was not properly capitalized. Another handy use for this function is to make comparisons when you don't know how the data is capitalized. The procedure below will select all data where the terms are NET 30, Net 30, or net 30.

```
select upperword(Terms)="Net 30"
```

The table below shows how the `upperword()` function affects various items of text:

```
upper("John Smith") → John Smith
upper("NET 30") → Net 30
upper("NEW York") → New York
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value with this function, for example `upperword(34)`. If you have a number you must convert the number to text before using it with this function, for example `upperword(str(34))`. Of course this function really doesn't make much sense when applied to a number, even if it is converted to text first.

**See Also:** [upper\(\)](#) function  
[lower\(\)](#) function

# UPRECORD

**Syntax:** UPRECORD

**Description:** The **uprecord** statement moves the cursor up one visible record in the active window. This is the opposite of the [downrecord](#) statement.

**Parameters:** This statement has no parameters.

**Action:** This statement moves the cursor up one visible record in the Data Sheet, Design Sheet, Cross Tab view, or View-as-list Form view. In a Individual Record Form view the view will change to the next record up in the database. If the cursor is already on the first visible record this statement will do nothing.

You can use this statement in conjunction with the [info\("bof"\)](#) or [info\("stopped"\)](#) functions to test to see if you are on the first visible record in the window.

This statement has the same effect as clicking on the **Up Record** tool on a tool palette (when available).

**Examples:** This simple example could be used in either the Data Sheet, Form view or Cross Tab view to move the cursor to the next visible record below the current record, making this next record the current record.

```
uprecord
```

This example adds up the Qty field for the current record and every record above it.

```
local UpQty
UpQty=Qty
loop
 uprecord
 UpQty=UpQty+Qty
until info("bof")
```

**Views:** This statement may be used in any view.

**See Also:** [firstrecord](#) statement  
[info\("eof"\)](#) function  
[info\("stopped"\)](#) function  
[lastrecord](#) statement  
[downrecord](#) statement

# URLDECODE(...)

**Syntax:** URLDECODE(url

**Description:** The `urldecode()` function takes standard ASCII text and converts into a format guaranteed to be legal in an internet URL (Universal Resource Locator). For example the url `my%20web%20page` is converted to `my web page`.

**Parameters:** This function has one parameter: `url`.  
`url` is the ASCII text you want to convert from URL format.

**Result:** This function returns regular [ascii](#) text. Any special characters in the text are converted to regular ASCII format.

**Examples:** The example assumes that you have a field named `WebAddress` that contains web page addresses. The example uses the `urldecode()` function to convert the url to a more readable format before storing it in the database. Be sure to use the `urlencode()` function to convert back to url format before passing this web address to the browser again.

```
global ActiveURL
startscript webScriptPath+"GetCurrentURL"
WebAddress=urldecode(ActiveURL)
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `url` parameter.

**See Also:** [urlencode\(\)](#) function

# URLENCODE(...)

**Syntax:** URLENCODE(URL)

**Description:** The `urlencode()` function takes standard ASCII text and converts into a format guaranteed to be legal in an internet URL (Universal Resource Locator). For example the url `my web page` is converted to `my%20web%20page`.

**Parameters:** This function has one parameter: `url`.

`url` is the ASCII text you want to convert to URL format.

**Result:** This function returns regular [ascii](#) text. Any special characters in the text are converted to their URL equivalent (% followed by the hexadecimal value of the character).

**Examples:** The example assumes that you have a field named `WebAddress` that contains web page addresses. The example uses the `urlencode()` function to make sure that the url is valid before passing it to the browser.

```
global ActiveURL
ActiveURL=urlencode(WebAddress)
startscript webScriptPath+"SetCurrentURL"
```

**Errors:** **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the `url` parameter.

**See Also:** [urldecode\(\)](#) function

## V(...)

- Syntax:** V(point)
- Description:** The `v`( function extracts the vertical position from a point (see `point`(, [graphic coordinates](#)).
- Parameters:** This function has one parameter: `point`
- point** is a number that includes both the vertical and horizontal components of a position. This number is usually created with the [point](#)(, [info\("click"\)](#), or [info\("mouse"\)](#) functions.
- Result:** This function returns a number (an integer) that describes the vertical position of the point. This number will be between -32,768 and +32,767. (Unlike standard cartesian coordinates, positive is down and negative is up.)
- Examples:** The procedure below displays a message if you click on a spot less than 50 pixels from the top of the screen.
- ```
if v( info("click") ) ≤ 50
    message "You're near the top!"
endif
```
- Errors:** Type mismatch: text argument used when number was expected. This error occurs if you attempt to use a text value for the `point` parameter.
- See Also:** [xytoxy](#)(function
[point](#)(function
[h](#)(function
[info\("click"\)](#) function
[info\("mouse"\)](#) function
[rectangle](#)(function
[rtop](#)(function
[rbottom](#)(function
[rleft](#)(function
[rright](#)(function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

VAL(...)

Syntax: VAL(text)

Description: The val(function converts text into a number. The text must contain a valid number.

Parameters: This function has one parameter: text.

text is the text that you want to convert to a number. The text must contain a valid number, with no additional characters on the beginning or the end

Result: This function returns the number that was converted from the text.

Examples: The example below assumes that the current database has a numeric field called Distance. The procedure will ask the user for the speed, then display a message like this: At 45mph the elapsed time to travel 92 miles is 123 minutes.

```

local Speed
Speed="60"
getttext "Enter the Speed",Speed
Speed=val(Speed)
message "At "+str(Speed)+
    "mph the elapsed time to travel "+
    str(Distance)+" miles is "+
    str(Distance*60/Speed)+" minutes."

```

The getttext statement (line 3) allows the user to enter the speed, but as its name implies, the result is a text data item. Line 4 uses the val(function to convert the Speed from text to numeric. (Note: If the user typed in any non-numeric characters, the procedure will stop at this point and display an error message.)

Trapping Numeric Conversion Errors:

The val(function converts a text data item into a numeric value, but what if the text doesn't contain a valid number. For example, what is the numeric value of abc? According to Panorama it doesn't have any value at all, and the procedure will normally stop and display an alert. If you don't want the procedure to stop, you can trap the error with the if error statement. The example below traps the error and forces the user to enter the speed over and over again until they enter a valid numeric value.

```

local Speed
loop
    Speed="60"
    getttext "Enter the Speed",Speed
    Speed=val(Speed)
    if error
        message "Please enter a numeric value for the speed!"
        Speed=0
    endif
while Speed=0
message "At "+str(Speed)+
    "mph the elapsed time to travel "+
    str(Distance)+" miles is "+
    str(Distance*60/Speed)+" minutes."

```

Another possible way to handle this error is to fill in a default value and continue with the procedure.

You can also attempt to strip out conversion errors in advance, by using the [striptonum\(\)](#) function. This function removes all non-numeric characters from a text item.

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to convert a value that is already a number.

See Also: [pattern\(\)](#) function
[str\(\)](#) function
[striptonum\(\)](#) function

WATCHCURSOR

Syntax: WATCHCURSOR

Description: The `watchcursor` statement re-enables the watch cursor (after it has been disabled with the `nowatchcursor` statement)..

Parameters: This statement has no parameters.

Action: This statement enables the watch and pie cursors that Panorama normally uses when performing a potentially slow operation. The watch cursor is also automatically re-enabled when the procedure is finished, even if you do not use the `watchcursor` statement.

Examples: While the operations in this example are performed the cursor stays as an arrow or cross until the data is exported, at which time it will flip into a watch.

```

noshow
nowatchcursor
field Date
groupup by month
field Category
groupup
field Amount
total
outlinelevel 2
watchcursor
export "Summaries.txt", exportline()+¶
showpage
endnoshow

```

Views: This statement may be used in any view.

See Also: [nowatchcursor](#) statement
[noshow](#) statement
[endnoshow](#) statement
[showpage](#) statement
[showline](#) statement
[showfields](#) statement
[showvariables](#) statement
[showcolumns](#) statement
[showrecordcounter](#) statement
[showother](#) statement
[hide](#) statement
[show](#) statement
[nundo](#) statement

WEEK1ST(...)

Syntax: WEEK1ST(date)

Description: The `week1st`(function computes the first day of a week (Sunday).

Parameters: This function has one parameter: `date`.
`date` is a number representing the date.

Result: This function calculates the first day of the month. For example, if the date passed to this function is July 12, 1995 (a Wednesday), this function will return the date July 9, 1995 (a Sunday). The date is returned as a number.

Examples: The example below selects the orders placed this week, then displays the count.

```
select OrderDate>week1st(today()) and  
OrderDate<week1st(today()+7)  
message str( info("records") )+" orders this week"
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

See Also: [monthlength](#)(function
[monthmath](#)(function
[dayofweek](#)(function
[month1st](#)(function
[quarter1st](#)(function
[year1st](#)(function
[date](#)(function
[datepattern](#)(function

WHILE

Syntax: `WHILE true-false test`

Description: The **while** statement is used at the end of a loop. A loop is a sequence of statements that are executed over and over again.

Parameters: This statement has one parameter: true-false test.

true-false test is a formula that should result in a true (-1) or false (0) answer. Usually the formula is created with a combination of comparison operators (=, ≠, , etc.) and boolean combinations (and, or, etc.) For example the formula `Name="Smith"` will be true if the field or variables `Name` contains the value Smith, and false if it contains any other value. The test may also consist of the word forever, in which case the loop will never stop (unless the loop has a [stop](#), [rtn](#), or [stoploopif](#) statement inside it).

Action: The **while** statement decides whether to jump back up to the top of the loop, or to continue with the next statement after the loop. The formula in the **while** statement is evaluated each time Panorama reaches the end of the loop. Panorama will continue looping as long as the result of the formula remains true. If the result of the formula is false, the loop stops and the procedure continues with the next statement after the **while** statement.

Examples: This example adds 10 new records to the current database.

```

local NewCount
NewCount=10
loop
  NewCount=NewCount-1
  addrecord
while NewCount>0

```

This example prints all unprinted records using the appropriate form. In this case the loop can only stop at the top (because of the [stoploopif](#)), because this **while** is forever!

```

find PrintedStatus=""
loop
  stoploopif (not info("found"))
  openform PrintForm
  print ""
  PrintedStatus="Complete"
  closewindow
  next
while forever

```

Views: This statement may be used in a procedure run from any view, and also works when no windows are open at all.

See Also: [loop](#) statement
[until](#) statement
[stoploopif](#) statement
[repeatloopif](#) statement

WINDOW

Syntax: WINDOW

Description: The **window** statement brings a Panorama window to the front. It can also open a “secret” invisible window.

Parameters: This statement has one parameter, **name**.

name is the exact name of the window that is to be brought to the front. This must be a window that is currently open. If necessary, the procedure can find out the name of the current window with the [info\("windowname"\)](#) function. It can get a list of all open windows with the [info\("windows"\)](#) or [listwindows\(\)](#) function.

To open a “secret” window, the window name should be `<dbname>:Secret`. A secret window is just like any other window that contains a form, except that it is invisible. A procedure can flip to a secret window in another database (the database must be open), perform some operation on that database (search, sort, etc.) then go back to the original window—all without the flashing and updating that usually occur when you flip from window to window. Secret windows are temporary. The secret window ceases to exist as soon as the procedure brings another window to the top (or as soon as the procedure stops.) Note: The word **Secret** in the window name may be capitalized any way you want: `secret`, `Secret` or `SECRET`.

Action: This statement will bring a window to the front, on top of all other windows. The window must already be open (unless it is a secret window) and its name must be spelled correctly with all punctuation. If the window is not already open, an error will be generated, which can be trapped by `if error` (otherwise the procedure will stop). A window that is not already opened may be opened using the [openform](#), [opensheet](#), [opencrosstab](#), [openprocedure](#) or [opendesignsheet](#) statements.

If you simply want to open any window in another database but you don't care what window, use the [openfile](#) statement. If the database is already open this statement will simply bring one of its windows to the top.

Examples: This example prints using the **Label** form. If the **Label** window is already open it simply brings it to the top, otherwise it opens the form.

```

window "Clients:Label"
if error
  openform "Label"
endif
print dialog

```

This example updates an inventory database. It temporarily switches to the **Inventory** data sheet window to do the update, then switches back to the original window when it is done.

```

local wasWindow
wasWindow=info("windowname")
window "Inventory"
if error
  message "Sorry, cannot update inventory."
  stop
endif
find Description=grabdata("Invoice",Description)
if info("found")

```

```

    QtyOnHand=QtyOnHand-grabdata("Invoice","Qty")
endif
window wasWindow

```

This example is exactly the same as the previous example, except that it uses a secret invisible window instead of the data sheet.

```

local wasWindow
wasWindow=iinfo("windowname")
window "Inventory:SECRET" /* use secret window */
if error
    message "Sorry, cannot update inventory."
    stop
endif
find Description=grabdata("Invoice",Description)
if info("found")
    QtyOnHand=QtyOnHand-grabdata("Invoice","Qty")
endif
window wasWindow

```

Views: This statement may be used in any view

See Also: [windownto](#)back statement
[openfile](#) statement
[opensecret](#) statement
[opencrosstab](#) statement
[opendesig](#)sheet statement
[opendialog](#) statement
[openform](#) statement
[openprocedure](#) statement
[info](#)("windowname") function
[info](#)("windows") function
[listwindows](#)((function

WINDOWBOX

- Syntax:** WINDOWBOX coordinates
- Description:** The `windowbox` statement allows a procedure to specify the size and location of a new window. It should be used just prior to one of the following statements: `opensheet`, `openform`, `opendialog`, `openprocedure`, `opencrosstab`, or `opendesignsheet`.
- Parameters:** This statement has one parameter, `coordinates`.
- `coordinates` is a text item that must contain four numbers within the item. The four numbers are the top, left, bottom, and right pixel coordinates of the new rectangle.
- The coordinates may also contain window options, which must be at the end (after the four numbers). If the text contains `NoPalette`, the window will not have a tool palette. If the text contains `NoVertScroll`, the window will not have a vertical scroll bar. If the text contains `NoHorzScroll`, the window will not have a horizontal scroll bar. If the text contains `NoDragBar`, the window will not have a drag bar across the top (the window will look like a dialog box). A procedure may combine several of these options separated by spaces.
- Action:** This statement allows the procedure to specify the size and location of the next window to be opened.
- Examples:** This simple example opens a new window for the Credit Card form. The top is 120 pixels down, the left side at 70 pixels in and the bottom at 280 pixels down and the right side at 440 pixels.
- ```

windowbox "120 70 280 440"
openform "Credit Card"

```
- This example opens the same window but without a tool palette or horizontal scroll bar.
- ```

windowbox "120 70 280 440 noPalette noHorzScroll"
openform "Credit Card"

```
- Views:** This statement may be used in any view, and also works when no windows are open at all.
- See Also:** [setwindow](#) statement
[setwindowrectangle](#) statement
[zoomwindow](#) statement
[fitwindow](#) statement
[opensheet](#) statement
[opencrosstab](#) statement
[opendesignsheet](#) statement
[opendialog](#) statement
[openform](#) statement
[info\("windowrectangle"\)](#) function

WINDOWGLOBAL

Syntax: WINDOWGLOBAL variables

Description: The **windowglobal** statement creates one or more variables that are specific to the current window. Window global variables may be used by any procedure as long as the same window is open, and remain active until you close the window.

Parameters: This statement has one parameter: variables.

variables is a list of variables to be created. Each variable should be separated from the next by a comma. If a variable name contains spaces or punctuation it should be surrounded by chevron (« ») characters.

Action: This statement creates one or more window specific variables. Variables can be used to hold pieces of information (numbers or text). Each variable has a name.

The **windowglobal** statement reduces that chance for conflict between windows in the same database. It allows you to define the same variable name over and over again in different windows, with each window having a separate value. This is very useful when using window clones.

Window Clones

Panorama normally allows only a single window per form. However, Panorama 3.1 allows a single form to be opened over and over again into multiple windows. This is called window "cloning." To allow a form to be cloned you must open the **Form Preferences** dialog and select the **Allow Clones** option.

A window clone cannot be opened manually...clone windows must be created with the **openform** statement in a procedure. Here is a typical procedure that opens a slightly off-set clone of the current window:

```
setwindowrectangle rectangleadjust( info("windowrectangle"),10,10,10,10)
openform info("formname")
```

This procedure will not create a clone window unless the **Allow Clones** option is turned on.

Designing Forms for Clone Windows

Although any form can be cloned if the **Allow Clones** option is turned on, most forms will not work very intelligently if they are cloned. In general, a form that is designed to be cloned should not contain any fields or global variables, only **windowglobal** variables. If your form contains SuperObjects and the **Allow Clones** option is turned on, the SuperObjects will automatically create **windowglobal** variables instead of global variables. Since the **windowglobal** variables can be manipulated separately for each clone window you can control each clone window individually, even though all the clone windows use the same form template.

Examples: The example assumes that you have create a form called **File Display** that displays the contents of a text file using a Text Display object with the formula

```
fileload(wFolder,wFile)
```

The procedure below will open a new window and display the file.


```
local folder,file,type  
openfile folder,file,type,"TEXT"  
if file="" stop endif  
openform "File Display"  
windowname file  
windowglobal wFolder,wFile  
wFolder=folder  
wFile=file  
showvariables wFolder,wFile
```

Each time you run this procedure another window will open. The only limit to the number of windows that can be opened is Panorama's ultimate limit of 32 windows.

Views: This statement should only be used in a form view.

See Also: [global](#) statement
[local](#) statement
[windowglobal](#) statement
[permanent](#) statement
[globalize](#) statement
[grabfilevariable](#)(function
[grabwindowvariable](#)(function
[info](#)("windowvariables") function
[undefine](#) statement

WINDOWNAME

Syntax: WINDOWNAME name

Description: The **windowname** statement changes the name of the current window.

Parameters: This statement has one parameter: name.

name is the text that will be used as the new name for the window.

Action: Panorama windows have a name that appears in the middle of the title bar. Usually the window name is a combination of the database name and the current form (or other view) name, for example Invoice:Report. The **windowname** statement allows you to change the name of the window to anything you want. However, the change is temporary. As soon as the view is changed (for example with the goform statement, the window name will revert to the usual database/view combination (until you use the **windowname** statement again).

Examples: The example below is from a database called Finances. Normally the window name for this form would be **Finances:Summary**. However, in this case the window name will be something like **1997**.

```
local year
year=datepattern( today(), "yyyy" )
gettext "What Year", year
openform "Summary"
select datepattern(Date, "yyyy")=year
windowname year
```

Views: This statement may be used in any view.

See Also: [info\("windowname"\)](#) function

WINDOWPROC

Syntax: WINDOWPROC

Description: The `windowproc` statement allows a procedure to open windows using any kind of Macintosh window, not just the standard windows normally supported. For example a procedure can open a rounded corner window (like a calculator) or a window with no border at all. If you are an advanced C or assembly language programmer you can even write your own window proc and use it with Panorama (refer to Inside Mac for information on writing your own window procs). **This statement is ignored on Windows systems.**

Parameters: This statement has one parameter: `procid`.

procid is a number that specifies the Window Proc ID. You can use the window procedures built in to the Mac ROM (listed below) or specify your own. The standard built in mac window procedures are listed in Inside Mac Vol I pg 273, but here is a quick summary of what is available:

Window Proc **0** - This is the standard window. If you want a standard window don't use this, because Panorama has it's own standard window that supports the pop-up view menu.

Window Proc **1** - This is the standard dialog window. You can get the same effect simply by specifying "nodragbar" with the [setwindow](#) command.

WindowProc **2** - This is a plain dialog box with no border.

WindowProc **3** - This is a plain dialog box with no border but with a drop shadow.

WindowProc **16** - This is a rounded corner window with a black drag bar. This window is often used for calculator DA's.

Action: Most Macintosh windows have a standard appearance: a drag bar across the top and a slight shadow on the bottom and right hand sides. However, there are several other styles available, and a C or Pascal programmer can even create their own style of window. The `windowproc` statement allows Panorama to use any style of window available. This statement must be used just before the statement that actually opens the window (usually [openform](#)). Once the window is opened, you cannot change its style. The only way to change the style is to close the window and then re-open it.

If you use a custom window proc, keep in mind that the window proc is not saved as part of the database file. If the file is saved with a custom window open, then re-opened, the window will re-open using the standard window procedure. Only windows that have been opened with the proper procedure statements will use custom window procedures.

Examples: The example below will open a form named Calculator. The form will appear in a window with rounded corners, just like the Calculator desk accessory.

```

windowproc 16
openform "Calculator"

```

Views: This statement may be used in any view.

See Also: [openform](#) statement
 [opendialog](#) statement

WINDOWTOBACK

Syntax: WINDOWTOBACK *name*

Description: The `windowback` statement moves a Panorama window to the back, behind all other Panorama windows.

Parameters: This statement has one parameter, *name*.

name is the exact name of the window that is to be sent to the back. This must be a window that is currently open. If necessary, the procedure can find out the name of the current window with the `info("windowname")` function. It can get a list of all open windows with the `info("windows")` or `listwindows((` function.

Action: This statement will send a window to the back, below all other windows. The window must already be open and its name must be spelled correctly with all punctuation. If the window is not already open, an error will be generated, which can be trapped by `if` error (otherwise the procedure will stop).

Examples: This example prints using the `Label` form. When it is finished, it sends the window behind all other windows.

```
window "Clients:Label"  
if error  
    openform "Label"  
endif  
print dialog  
windowtoback "Clients:Label"
```

Views: This statement may be used in any view.

See Also: `window` statement
`closewindow` statement
`info("windowname")` function
`info("windows")` function
`listwindows((` function

Word Processor Programming

Background: The [superobject](#) and [activesuperobject](#) statements allow a procedure to communicate and send commands to SuperObjects. Each type of SuperObject has its own list of commands and parameters for those commands.

Quick Please refer to [text editor programming](#) for details of the following commands:

```
"Open "
"Close "
"Cut "
"Copy "
"Paste "
"Clear "
"GetSelection" ,<START> ,<END>
"SetSelection" ,<START> ,<END>
"GetText " ,<TEXT>
"SetText " ,<TEXT>
"InsertText " ,<TEXT>
"GetSelectedText " ,<TEXT>
"Find "
"FindNext "
"Change "
"Spell "
```

The commands shown below are defined in this section.

```
"GetFont " ,<TEXT>
"SetFont " ,<TEXT>
"GetFontSize" ,<INTEGER>
"SetFontSize" ,<INTEGER>
"GetJustification" ,<TEXT>
"SetJustification" ,<TEXT>
"GetLeading" ,<NUMBER>
"SetLeading" ,<NUMBER>
"GetLeftIndent" ,<NUMBER>
"GetRightIndent" ,<NUMBER>
"GetFirstIndent" ,<NUMBER>
"SetLeftIndent" ,<NUMBER>
"SetRightIndent" ,<NUMBER>
"SetFirstIndent" ,<NUMBER>

"ClearTabs "
"GetTab" ,<Tab> ,<Position> ,<Type> ,<Leader>
"AddTab" ,<Position> ,<Type> ,<Leader>
"GetStyle" ,<Style Name> ,<Status>
"SetStyle" ,<Style Name> ,<Status>
"GetTextColor" ,<Color>
"GetTextBackgroundColor" ,<Color>
"SetTextColor" ,<Color>
"SetTextBackgroundColor" ,<Color>
"ShowRuler" ,<Status>
"LockDocument " ,<Status>
```

GetFont

,<TEXT>

This command gets the font of the selected text. If the selected text contains more than one font, only the first font is listed. The example below displays the font of the selected text.

```
local MyFont,MyFontSize
activesuperobject "GetFont",MyFont
activesuperobject "GetFontSize",MyFontSize
message "The font is: "+str(MyFontSize)+"pt "+MyFont
```

SetFont

,<TEXT>

This command changes the font of the selected text. All the selected text is changed to the same font. The example below inserts the current time into the text using 15 point American Typewriter.

```
if info("activesuperobject") = "" stop endif
activesuperobject "SetFont","American Typewriter"
activesuperobject "SetFontSize",15
activesuperobject "InsertText",
timepattern( now(),"hh:mm am/pm")
```

GetFontSize

,<INTEGER>

This command gets the font size of the selected text. If the selected text contains more than one size, only the first size is listed. See GetFont (above) for an example of this command.

SetFontSize

,<INTEGER>

This command changes the size of the selected text. All the selected text is changed to the same size. See GetFontSize (above) for an example of this command.

GetJustification

,<TEXT>

This command gets the text justification status of the selected text. The result may be one of these values: Left, Center, Right or Full. If the selected text contains more than one justification, only the first justification is listed.

SetJustification

,<TEXT>

This command changes the justification of the selected text. The new justification may be one of these values: Left, Center, Right or Full. All the selected text is changed to the same justification. The example below adds a new, right justified line to the end of the document, then types the current time into that line.

```
if info("activesuperobject") = "" stop endif
activesuperobject "SetSelection",999999,999999
activesuperobject "InsertText",¶
activesuperobject "SetJustification","Right"
activesuperobject "InsertText",
timepattern( now(),"hh:mm am/pm")
```

GetLeading , <NUMBER>
This command gets the leading of the selected text. If the selected text contains more than one leading, only the first leading is returned. For normal single spaced text the leading value is zero.

SetLeading , <NUMBER>
This command changes the leading of the selected text. All the selected text is changed to the same leading. For normal single spaced text the leading value should be zero.

GetLeftIndent , <NUMBER>
GetRightIn-
dent
GetFirstIn-
dent
These command gets the indents of the selected text. If the selected text contains more than one indent value, only the first indent value is returned. The GetLeftIndent and GetRightIndent commands return the left and right indent values, respectively. The GetFirstIndent command returns the indent of the first line of the paragraph. All indent values are specified in points (72 points per inch).

SetLeftIndent , <NUMBER>
SetRightIn-
dent
SetFirstIndent
This command changes the indents of the selected text. All the selected text is changed to the same indents. All indent values are specified in points (72 points per inch). The example below sets the margins for the currently selected text at 1/2 inch (36 points).

```
activesuperobject "SetLeftIndent",36
activesuperobject "SetRightIndent",36
```

ClearTabs This command clears all tabs from the selected text.

GetTab , <Tab>, <Position>, <Type>, <Leader>
This command gets information about a tab stop active with the currently selected text. The first parameter, <Tab>, is the number of the tab you want to get information about (starting with 1). The remaining three parameters are filled in by the command. The <Position> is the position of the tab, in points. The <Type> is the type of tab. The possible types are Left, Center, Right, Decimal and None. A <Type> of None indicates that the requested tab does not exist. In that case values of the <Position> and <Leader> characters are not defined. The <Leader> parameter is the tab leader character, if any. The example below will display a list of the current tab stops.

```
local tabList,theTab,tabSpot,tabType,tabLeader
theTab=1
tabList=""
tabType="None"
loop
  activesuperobject "GetTab",theTab,
  tabSpot,tabType,tabLeader
  stoploopif tabType = "None"
  tabList=sandwich("",tabList,", ")+
  tabType+" "+pattern(tabSpot/72,"#.##")+{" "}
  theTab=theTab+1
while forever
message tabList
```


AddTab

,<Position>,<Type>,<Leader>

This command adds a new tab stop. The <Position> is the position of the tab, in points. The <Type> is the type of tab. The possible types are Left, Center, Right and Decimal. The <Leader> parameter is the tab leader character, if any. The example below will add a tab stop and then add several lines of pricing information.

```
if info("activesuperobject") = "" stop endif
activesuperobject "SetSelection",999999,999999
activesuperobject "InsertText",¶
activesuperobject "ClearTabs"
activesuperobject "AddTab",220,"Decimal", ""
activesuperobject "InsertText",
  "Widget"++"6.56"++¶+
  "Micro Widget"++"3.12"++¶+
  "Deluxe Widget"++"18.63"
```

GetStyle

,<Style Name>,<Status>

This command will check the selected text to see if it is a certain style. If there is more than one style in the selected text, the style of the first character will be returned. If the selected text matches the specified cell the result is -1, if it does not match, the result is 0. The style names are:

```
Plain
Bold
Italic
Outline
Shadow
Condensed
Extended
Hidden Text
Strikeout
SuperScript
SubScript
SmallCaps
AllCaps
AllLowerCase
FormulaMerge
UnderLine
DoubleUnderLine
WordUnderLine
DottedUnderLine
OverLine
```

A text selection may contain more than one of these styles. You must test for each style separately. Here is an example that checks for bold strikeout text.

```
local isBold,isStrike
activesuperobject "GetStyle","Bold",isBold
activesuperobject "GetStyle","Strikeout",isStrike
if isBold=-1 and isStrike-1
  yesno "Delete this?"
  if clipboard() contains "yes"
    activesuperobject "Cut"
  endif
endif
```

SetStyle

,<Style Name>,<Status>

This command will change the style of the selected text. If the status is -1, the specified style is turned on. If the status is 0, the specified style is turned off. The style names are listed in the previous section.

The SetStyle command adds or subtracts the specified style from the styles the selected text already has. If you want to make sure the selected text has only the styles you specify, start by making the text plain. The example below sets the selected text to bold double underline.

```
activesuperobject "SetStyle","Plain",-1
activesuperobject "SetStyle","Bold",-1
activesuperobject "SetStyle","DoubleUnderLine",-1
```

**GetTextColor
GetTextBack-
groundColor**

,<Color>

These commands will return the color of the selected text (See [colors](#)).

**SetTextColor
SetTextBack-
groundColor**

,<Color>

These commands will set the color of the selected text (See [colors](#)). The example below sets the selected text to a pure blue.

```
activesuperobject "SetTextColor",rgb(0,0,65535)
```

ShowRuler

,<Status>

This command will turn the display of the ruler on and off. If the <Status> is -1, the ruler will be shown; if the <Status> is 0, the ruler will not be shown. The example below makes sure the ruler is visible.

```
activesuperobject "ShowRuler",-1
```

**LockDocu-
ment**

,<Status>

This command allows the document to be locked. If the <Status> is -1, the document will be locked and cannot be edited. If the <Status> is 0, the document will be unlocked and may be edited again. The example below locks the current document.

```
activesuperobject "LockDocument",-1
```

WORD(...)

Syntax: WORD(number)

Description: The `word()` function converts a number into a single word (2 bytes) of binary data (see [binary data](#)).

Parameters: This function has one parameter: `number`.

number is the value that you want to convert into a binary number. This value must be between 0 and 65,535.

Result: This function converts the number into a single word of binary data (16 bits). This binary data should be handled as text data.

Examples: This example converts the number 9837 into a binary word, then copies that binary data into the variable X.

```
local X  
X=word(9837)
```

If you check the size of X with the [sizeof\(\)](#) function, you'll find that it is 2 bytes long.

See [c/pascal structures](#) for additional examples of the `word()` function.

Errors: **Type mismatch: text argument used when number was expected.** This error occurs if you attempt to use a text value for the number parameter.

Illegal number. This error occurs if you attempt to convert a value less than 0 or greater than 65,535.

See Also: [byte\(\)](#) function
[longword\(\)](#) function
[radix\(\)](#) function
[radixstr\(\)](#) function

WORDLIST

- Syntax:** WORDLIST array,separator,word
- Description:** The `wordlist` statement builds an array of words from the dictionary. This statement requires that the optional Panorama dictionary be installed.
- Parameters:** This statement has three parameters: array, separator and word.
- array** is a field or variable. The wordlist statement will put the list of words into this field or variable.
- separator** is the character that will be placed in between each word (see [text arrays](#)).
- word** specifies what words are to be included in the list of words. You can pick either words that sound like a word, or words that start with certain letters.
- To build a list of words that sound like a word simply use that word as the parameter. For example to find words that sound like `abalone` use the parameter `"abalone"`.
- To build a list of words that start with certain letters the parameter should be those letters followed by an asterisk. For example to find all words that begin with non use the parameter `"non*"`. (Note: Unlike the match operator, you may use only one `*` and it must be at the end of the text.)
- Action:** This statement uses the Panorama dictionary to build a list of English words.
- Examples:** This procedure will check the spelling in the currently active text editor or word processor SuperObject™. If an incorrect word is located this procedure will fill the variable `suggestionList` with a list of words that sound like the incorrect word. This list could be used for a pop-up menu or List SuperObject.
- ```
global suggestionList
local badWord
activesuperobject "spell"
activesuperobject "getselectedtext",badWord
wordlist suggestionList,␣,badWord
```
- Views:** This statement may be used in any view.
- See Also:** [spelling](#) statement

# WRITERESOURCE

- Syntax:** WRITERESOURCE type,id,data
- Description:** The `writeresource` statement modifies and/or creates a resource item. The resource file must be opened with the [openresourcerw](#) statement.
- Parameters:** This statement has three parameter: `type`, `id` and `data`.
- type** is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).
- id** is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).
- data** is the actual data that will be placed into the item. This may be in a field, variable, or constructed with a formula.
- Action:** This statement saves data directly into a resource item. If there is more than one resource file open the resource will be written into the file that was most recently opened (see the [activeresource](#) statement).
- Examples:** The procedure below writes some text (for example `Last update: 10/18/02`) into STR resource number `2000`. If the resource does not exist it will be created.
- ```
writeresource "STR ",2000,
  string255("Last update: "+datepattern( today(),"mm/dd/yy")
```
- Views:** This statement may be used in any view.
- See Also:** [openresource](#) statement
[openresourcerw](#) statement
[closeresource](#) statement
[deleteresource](#) statement
[renameresource](#) statement
[activeresource](#) statement

XCALL

Syntax: XCALL procedure,parameters

Description: The `xcall` statement calls an external procedure. External procedures are usually written in C or Pascal. Before an external procedure can be used it must be installed in a resource file, and the resource file must be opened with the [openresource](#) statement.

Parameters: This statement has at least one parameter: `procedure`. It may have additional parameters which are not processed by the `xcall` statement itself but simply passed through to the external procedure.

procedure is name of the external procedure. This is actually the name of the resource containing the external procedure.

parameters are additional parameters used by the external procedure. The exact parameters required and their meanings will be different depending on what external procedure you are calling. If a parameter is used to pass information to the external procedure you may use any formula to calculate the value of the parameter. Parameters may also receive values from the external procedure. If a parameter is used to receive a value, that parameter must be a single field or variable with no operators (`myValue`, not `myValue+yourValue` or `strip(myValue)`).

Action: This statement calls an external procedure.

Examples: The example uses an external procedure called `SerialIO` in a resource file called `Serial Access Package` to read a line of data from the serial port.

```

openresource "Serial Access Package"
local SerialErr
SerialERR=0
xcall "SerialIO","Open","PortA",SerialERR
local AC, ioerr, line, char
line=""
char="" AC=0
ioerr=0
loop
    xcall "SerialIO","GetAvailable","PortA",AC,ioerr
    if AC>0
        xcall "SerialIO","ReadChar","PortA",char,ioerr
        line=line+char
    endif
while char≠chr(13)

```

Views: This statement may be used in any view.

See Also: [openresource](#) statement

XYTOXY(...)

Syntax: XYTOXY((point/rectangle,fromrelative,torelative)

Description: The `xytoxy()` function converts a point or rectangle from one co-ordinate system to another. There are three possible co-ordinate systems: **Screen Relative**, **Window Relative**, and **Form Relative** (see [graphic coordinates](#)).

Parameters: This function has three parameters: `point/rectangle`, `fromrelative` and `torelative`.

point/rectangle is the point or rectangle that you want to convert to another co-ordinate system.

fromrelative is the co-ordinate system that the point or rectangle is in now. The three options for this parameter are:

"Screen" (may be abbreviated "S" or "s")

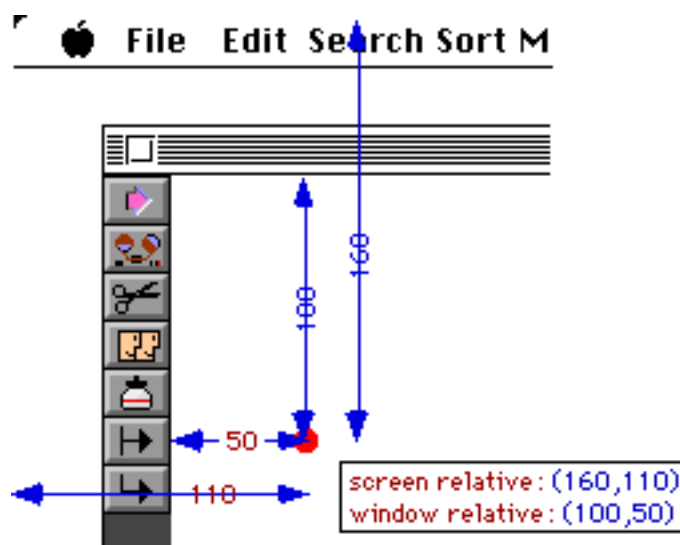
"Window" (may be abbreviated "W" or "w")

"Form" (may be abbreviated "F" or "f")

torelative is the co-ordinate system that you want to convert the point or rectangle into. This parameter accepts the same three options listed above.

Result: This function returns a point or rectangle (whatever was passed to it). This point or rectangle has been converted to a different co-ordinate system (for example screen relative to window relative).

Examples: The point shown below has window relative co-ordinates of (100,50). Its screen relative co-ordinates are (160,110).



The formula below will convert this point from window co-ordinates (100,50) to screen co-ordinates (160,110).

```
xytoxy(point(100,50),"w","s")
```

The `info("click")` function returns the screen relative co-ordinates of the last mouse click. This procedure will convert the mouse click point to form relative, then check to see if the mouse was clicked in the top inch of the form.

```
local mousePoint  
mousePoint=xytoxy(info("click"), "s", "f")  
if v(mousePoint)<=72  
    message "You clicked in the top inch of the form"  
endif
```

Errors: **Type mismatch: numeric argument used when text was expected.** This error occurs if you attempt to use a numeric value for the fromrelative or torelative parameters.

See Also: [point\(\)](#) function
[v\(\)](#) function
[h\(\)](#) function
[info\("click"\)](#) function
[info\("mouse"\)](#) function
[rectangle\(\)](#) function
[rtop\(\)](#) function
[rbottom\(\)](#) function
[rleft\(\)](#) function
[rright\(\)](#) function
[info\("screenrectangle"\)](#) function
[info\("windowrectangle"\)](#) function
[info\("buttonrectangle"\)](#) function
[info\("cursorrectangle"\)](#) function

YEAR1ST(...)

Syntax: YEAR1ST(date)

Description: The year1st(function computes the first day of a year.

Parameters: This function has one parameter: date.
date is a number representing the date.

Result: This function calculates the first day of the year. For example, if the date passed to this function is July 12, 1995, this function will return the date January 1, 1995. The date is returned as a number.

Examples: The example below calculates the number of days remaining in the current year.

```
Remaining=year1st(today()+366)-today()
```

Errors: **Type mismatch: text argument used when numeric was expected.** This error occurs if you attempt to use a text value for the date parameter.

See Also: [monthlength\(function](#)
[monthmath\(function](#)
[week1st\(function](#)
[month1st\(function](#)
[quarter1st\(function](#)
[date\(function](#)
[datepattern\(function](#)

YESEVENT

Syntax: YESEVENT

Description: The `yesevent` statement enables Panorama's event processing after it has been disabled with the [noevent](#) statement.

Parameters: This statement has no parameters.

Action: This statement turns on Panorama's event processing after it has been turned off with the [noevent](#) statement. You should only use this statement in one place—at the beginning of the `.CustomMenu` procedure as shown in the example below. In this application the [noevent](#) statement allows AppleEvents to open a database properly via your `.CustomMenu` procedure. This is important because the Finder uses AppleEvents to open a file when you double click on its icon. Using [noevent](#) in any other way will probably cause Panorama to crash! No kidding.

Examples: If your database uses a custom menu for the File menu your `.CustomMenu` procedure should contain the following statements at the very beginning of the procedure. The [noevent](#) statement must be the very first statement in the procedure. This will allow the Finder to open files with AppleEvents.

```
noevent
if info("trigger") beginswith "Menu.File.Open"
    openfile dialog
    stop
endif
yesevent
```

Views: This statement may be used in any view.

See Also: [noevent](#) statement
[openfile](#) statement

YESNO

- Syntax:** YESNO text
- Description:** The `yesno` statement displays an alert with a message and two buttons: **Yes** and **No**. The default is **Yes**.
- Parameters:** This statement has one parameter: text.
- text** is the message that will appear in the dialog when it is displayed. You may use any formula to create this text, but usually a text constant is used (text surrounded by double quote marks (Example: "Do you want to continue?").
- Action:** This statement allows the procedure to pause and asks a question requiring one of two responses: **Yes** or **No**. The response will be written to the clipboard so that it can be tested for later in the procedure.
- Examples:** This example asks the user if they want to remove old records. If they press the Yes button the procedure will remove all records that are more than 1 year old, otherwise the procedure will do nothing.
- ```
yesno "Remove old records?"
if clipboard() contains "Yes"
 select Date>today()-365
 removeunselected
endif
```
- Views:** This statement may be used in any view, and also works when no windows are open at all.
- See Also:** [noyes](#) statement  
[alert](#) statement  
[cancelok](#) statement  
[customalert](#) statement  
[customdialog](#) statement  
[getscrap](#) statement  
[gettext](#) statement  
[message](#) statement  
[okcancel](#) statement  
[alertmode](#) statement  
[clipboard\(\)](#) function  
[info\("dialogtrigger"\)](#) function

# ZEROBLANK(...)

**Syntax:** ZEROBLANK(value)

**Description:** The `zeroblank()` function tells Panorama to treat zeroes as empty space.

**Parameters:** This function has one parameter: `value`.

**value** is any numeric value or formula. If any field in this value or formula is blank (empty), the result of the formula will be blank (empty).

**Result:** The result of this function is either a numeric value, or empty if one of the source fields is empty.

**Examples:** This example calculates a line item in an invoice. The amount is calculated by multiplying the quantity times the price. However, if either the quantity or the price is blank (not zero, but actually blank) the amount will also be blank. This avoids the cluttered look in your invoices.

```
AmountΩ=zeroblank(QtyΩ*PriceΩ)
```

**Notes:** If this is a SQL database, blank numbers are not allowed. The number will appear blank temporarily, but will turn into a zero the next time the database is synchronized with the SQL server.

**Errors:** **Type mismatch: text argument used when numeric was expected.** This error occurs if you use text fields with this function, for example `zeroblank(FName+LName)`.

# ZOOMWINDOW

**Syntax:** ZOOMWINDOW Top,Left,Height,Width,Options

**Description:** The `zoomwindow` statement allows a procedure to move, resize, and change the attributes of an existing window.

**Parameters:** This statement has five parameters: `top`, `left`, `height`, `width` and `options`.

**top** is the position of the top edge of the rectangle. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)

**left** is the position of the left edge of the rectangle. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)

**height** is the height of the rectangle. This must be a number between 0 and +32,767.

**width** is the width of the rectangle. This must be a number between 0 and +32,767.

**options** is an item of text that optionally turns off elements of the new window. If the text contains `NoPalette`, the window will not have a tool palette. If the text contains `NoVertScroll`, the window will not have a vertical scroll bar. If the text contains `NoHorzScroll`, the window will not have a horizontal scroll bar. The `NoDragBar` option, which can be used with the `setwindow` and `setwindowrectangle` statements, has no effect when used with the `zoomwindow` statement. A procedure may combine several options separated by spaces. If the option text is empty (""), the window will appear normal.

**Action:** This statement allows the procedure to move a window to a new location, change the size of a window, or add or remove scroll bars and tool palettes.

**Examples:** This example enlarges a window to its maximum size for 10 seconds, then reduces it back to its original size.

```

local wasWindowRect,startDelay
wasWindowRect=info("windowrectangle")
zoomwindow
 rtop(info("maximumwindow")),
 rleft(info("maximumwindow")),
 rheight(info("maximumwindow")),
 rwidth(info("maximumwindow")),
 ""
startDelay=now()
loop until now(>startDelay+10
zoomwindow
 rtop(wasWindowRect),
 rleft(wasWindowRect),
 rheight(wasWindowRect),
 rwidth(wasWindowRect),"

```

**Views:** This statement may be used in any view.

**See Also:**      [setwindowrectangle](#) statement  
                  [setwindow](#) statement  
                  [window](#) statement  
                  [getwindow](#) statement  
                  [closewindow](#) statement  
                  [info\("windowrectangle"\)](#) function  
                  [info\("minimumwindow"\)](#) function  
                  [info\("maximumwindow"\)](#) function  
                  [info\("windowbox"\)](#) function