

Metacard Concepts and Techniques

Reference Manual

Metacard Classes

Gh. C. -- 08/12/2016

Index A

Introduction

Objects

Stacks

The Home Stack

The defaultStack and the topStack

mainStacks, substacks, and stackFiles

Menu Panels

Cards

Groups and Backgrounds

Radio Groups and Traversal

Buttons

Menu Modes

Button Contents Menus

menuName Menus

Fields

Field Keyboard Functions

Scrollbars

Images

Icons, Patterns, Brushes, & Cursors

Painting

[Tools](#)

[Graphics](#)

[Players](#)

[EPS objects](#)

[AudioClips](#)

[VideoClips](#)

[Scripting](#)

[Terms](#)

[Container](#)

[Constant](#)

[Literal](#)

[Chunk](#)

[Factor](#)

[Expression](#)

[Properties and Custom Properties](#)

[Handlers](#)

[Message Handlers](#)

[Function Handlers](#)

[Setprop Handlers](#)

[Getprop Handlers](#)

[Debugging Scripts](#)

[Improving Performance](#)

[Geometry Management](#)

[Hypermedia](#)

[Building Hypertext Links](#)

[Sound](#)

[Animation](#)

[Object Based Animation](#)

[Frame Based Animation](#)

[The Outside World](#)

[Importing HyperCard and SuperCard stacks](#)

[open process and shell\(\)](#)

[Import/Export](#)

[External Commands and Functions](#)

[Communicating with MetaCard](#)

[MetaTalk as a CGI or batch scripting language](#)

Introduction

To get the most out of MetaCard, there are a few key concepts and techniques that you should master. This stack is a collection of those concepts and techniques. If you haven't yet, you should go through the first one or two **MetaCard Tutorials** stacks before trying to digest this one.

If you don't have time to read this whole section, at least quickly skim this whole stack the first time through. What you don't retain the first time, you will be able to find more easily if you need it later.

There are a couple of different ways to navigate in this stack. You can use the buttons provided at the bottom of each card or you can use the **Navigator** stack (from the "Tools" menu) or the keyboard equivalents of the Navigator buttons, the

arrow keys. See the help for the Navigator stack for details (press F1 or the keyboard Help key when the navigator stack has the keyboard focus to get help on that stack).

As in the **MetaCard Tutorials** stack, words in blue are hypertext links to other cards or stacks. Click on them to go to the referenced object. Words in the `Courier` font are MetaTalk terms and you can also look most of them up in the **MetaTalk Reference** stack by clicking on them.

The Find dialog (in the Tools menu) can also be used with this stack (and with the reference stack). Or you can type a "find" command into the Message Box to find a piece of information if you only know a word or two.

If you don't find what you need to know here or in the **MetaTalk Reference** or **Dialog Box Help** stacks, purchasing one of the many books on Apple/Claris HyperCard, such as Danny Goodman's *The Complete HyperCard 2.0 Handbook* or Dan Winkler's *HyperTalk 2.0: The Book* and *Cooking with HyperTalk 2.0*, may prove helpful. You can also contact MetaCard Corporation using the supplied **Support** stack.

Developing GUI applications that are easy to learn and easy to use can be a difficult task. If you don't have any experience in this area, there are many books that can help you avoid the most common pitfalls. Among them are:

The Art of human-computer interface design by Brenda Laurel.

The cross-GUI handbook : for multiplatform user interface design by Aaron Marcus, Nick Smilonich, and Lynne Thompson.

Tog on software design by Bruce Tognazzini.

Usability Engineering by Jakob Nielson

It would also be a good idea to have some background knowledge of the look and feel standards for whichever platform you're using. On UNIX/X11 systems this information is in the *OSF/Motif User's Guide* and the *OSF/Motif Style Guide*. There are similar guides for Microsoft Windows and MacOS.

Objects

MetaCard objects are what you see and interact with when using MetaCard. There are twelve classes of MetaCard objects:

stacks,

cards,

groups (backgrounds),

buttons,

fields,

images,

graphics,
players,
EPS objects,
scrollbars,
audioClips, and
videoClips.

Groups, buttons, fields, images, graphics, EPS objects, and scrollbars are also referred to as controls since they are the elements the user interacts with.

An object's *properties* govern the way it looks and the way it responds to the user actions. These properties can be defined or viewed using the dialogs in the MetaCard development environment (which itself is just a collection of MetaCard objects). You can also define or change an object's property with a script using the `set` command.

Objects are arranged in a hierarchy of the following classes in the following order: stack, background, card, group, control. The hierarchy is significant for two reasons. First, *messages* sent as the result of user action are passed up the hierarchy if they are not handled in scripts at a lower level.

Secondly, properties that govern the colors, patterns, and fonts used to draw an object are inherited from their `owner` if not set. This allows you to change the color scheme of a stack by simply changing the stack colors, rather than having to change the colors of every object in the stack. If you try to `get` a property that is inherited, the MetaTalk constant `empty` is returned. To find out the attribute actually used in these cases, use the modifier `effective` as shown in this example:

```
get the effective topColor of field 1
```

There are a number of ways to specify the properties of an object. In most cases the easiest method is to give the object a unique name and then use that name when you want to specify a property. For example:

```
put the rect of button "ButtonName"
```

In this example, "ButtonName" is a name that was given to the object by the stack developer (probably with the "Button Properties" dialog).

The second way to specify an object is with the `id` property. An `id` is assigned automatically when an object is created and is guaranteed to be unique within the stack. An example:

```
put the loc of button id 1023
```

The third, and least desirable, method of specifying an object is by using the object's `number` property, which is the position of the object relative to other objects of the same type. Some examples:

```
put the top of button 1
put the left of the first button
put the bottom of any control
put the right of button (5 * variable)
```

The problem with using this addressing method is that if the objects are rearranged, the number of an object may change, which will require changing all of the scripts that refer to that object.

Stacks

Every window MetaCard opens is a stack. This includes even dialog boxes and menus. Every stack contains one or more cards. The different cards in a stack can be viewed by navigating through them using one of the four techniques described on the Intro card of this stack.

Stacks can be opened in one of two ways. The first is to run them directly from a system command line (by specifying as command line arguments to the "mc" command), or by using the system file browser. Once MetaCard is running, scripts can execute MetaTalk commands to open stacks. These commands, like all MetaTalk commands, are either used in message handlers within scripts or can be executed directly by typing them into the **Message Box**. To open the **Tools palette** (a special type of stack), for example, you could type the following into the **Message Box**:

```
palette "Tools"
```

You must know a stack's name in order to open it. If you don't know the name of a stack, and it happens to be the substack of some main stack, you will probably be able to find its name listed in the "Components" dialog (accessible from the "Stack Properties" palette) of the Home stack or of one of your mainStacks.

Note that there is a difference between the stack name and the file name the stack is saved in. Stack names may not only exceed the 14 character name length limit found on some UNIX systems and the 8.3 limits in MSDOS, but may also include special characters like spaces, commas, and ampersands (&) that should not be used in file names. By convention, stack file names should also have a ".mc" extension. When you choose "Save As..." from the MetaCard File menu, MetaCard will translate your stack name into an acceptable file name by removing special characters and adding the .mc extension.

When you want to open a stack but don't know its name, you can substitute the name of the file for the name of the stack in the command you use to open the stack. This is not a good general technique, however, because there may be a delay each time you open the stack using its file name. This is because MetaCard may have to reload the stack to verify that it contains a stack that has been opened previously. For the same reason, it's not a good practice to give the file the same name as the stack stored in it, because it may cause multiple loads of the stack when objects in the stack are

accessed.

When a stack is opened the entire stack is loaded into memory. This includes all cards, controls, and substacks of that stack. Loading large stacks can therefore take considerable time and require large amounts of memory. To minimize the impact of this behavior, it is a good practice to minimize the size of stacks, breaking an application up into multiple stacks and saving those stacks into separate files if it grows beyond a certain size (a few MB). Stacks are retained in memory even when they are closed so that they'll open immediately the next time they are accessed. Set the `destroyStack` property of a stack to true if you instead want them to be purged from memory when they are closed.

Stacks can be opened in different modes. MetaCard has eight different modes: `topLevel`, `topLevel locked`, `modeless`, `palette`, `modal`, `pulldown`, `option`, and `popup`. Two additional modes, one for cascading (pull-right) menus and one for combo boxes, are also available, but there are no MetaTalk commands to open stacks in these modes.

Modes dictate what can and cannot be done to a stack, as well as determine what the `topStack` is. Stacks opened from the command line are always opened as `topLevel`. Stacks opened with the `topLevel` command will show the current card number in their title bar if they have more than 1 card.

If the `cantModify` property of the stack is set to false, the stack will also have a "*" in its title bar denoting that the stack can be edited. Editable stacks show the cursor corresponding to the tool hilited in the `TOOLS` palette. Non-editable stacks always use the "browse" tool so there is no danger that they will be edited accidentally.

The Home Stack

MetaCard automatically opens the Home stack when you run the "mc" command. This stack has three purposes. The most important of these is that it serves as the licensed unit of MetaCard: a special key stored with the Home stack enables the editing capabilities of the MetaCard engine. The Home stack can also serve as a directory for other MetaCard stacks. For example, you could put icon buttons in the Home stack for the stacks you use most often. This allows you to open those stacks simply by clicking on the icon.

The defaultStack and the topStack

MetaCard enables you to have multiple stacks open at the same time. When this is the case (which is most of the time) MetaCard needs some way to determine which stack a given command should be applied to. In these cases MetaCard uses the global property `defaultStack`. For example, to find a string in this stack, the following two commands could be put into a button script.


```
set the defaultStack \  
to "Concepts & Techniques"  
find "some string"
```

When the script is executed, the global property `defaultStack` is set to the stack containing the object whose script is executing.

When you want a button to perform some operation on an object in another stack, the button script should set the `defaultStack`. If the stack name doesn't change, then you can hard code it as was done in the example above. If however, you want a stack button to perform some operation on an object in a stack whose name is not known in advance, the `topStack` function returns the name of the stack that is opened with the lowest mode and which was last brought to the top by the user if two or more stacks have that mode. The mode of a stack is determined by the command used to open the stack, with `topLevel` stacks with their `cantModify` property set to false having the lowest mode.

Many of the MetaCard dialogs and menus (e.g., the "Save" button in the File menu and the "Stack Properties" dialog) operate on the `topStack`. This design makes common editing procedures simpler, but does have disadvantages. First, if you have an editable stack open and want to start editing a locked stack (a stack with its `cantModify` property set to true), you'll have to either close or lock the editable stack first.

Second, if you're using a UNIX/X11 system and your window manager is set to pointer focus policy (as opposed the standard mode of explicit or click-to-type focus), you'll have a lot of trouble since the `topStack` changes every time you move the mouse from one window to another. You will find directions for setting the focus policy of your window manager back to the recommended "explicit" focus mode in the MetaCard installation instructions.

mainStacks, substacks, and stackFiles

In addition to containing cards and the groups and controls on them, some stacks (called `mainStacks`) contain other stacks (called `substacks`). Dialog boxes and menus are commonly stored as substacks of the main application window, which is typically a `mainsStack`.

As discussed previously, rather than having to save each stack in a separate file, substacks are saved into the same file as their `mainStack`. This allows you to store all of the stacks that are used in an application in a single file. Note that whenever a `mainStack` is saved, so are all of the substacks of that stack. Also, *and this is important*, whenever any of the substacks is saved, the `mainStack` of that substack is saved, again along with all of the other substacks of that `mainStack`.

Each `mainStack` has a file name associated with it which is stored in the stack's `fileName` property. The file name is the full path used to load and save that stack.

To save a stack into its own file, it must be a mainStack. There are three ways you can make a substack into a mainStack. The easiest way is just to choose "Save As..." from the MetaCard File menu. When the "MainStack Option" dialog appears, click on the "Yes" button. You can also use the "Main Stack" dialog from the "Stack Properties" dialog to set the mainStack of a stack to itself. Finally, you can make a stack a mainStack by executing a script that performs the necessary steps. For example, you could do it by typing the following two commands into the Message Box:

```
set the mainStack of stack "my main" \  
to "my main"  
save stack "my main" as "filename"
```

If your stack has dialogs or menus associated with it, you'll need to move them into your new mainStack file so that they become substacks of the new mainStack. You can do this with the "Main Stack" dialog (from the "Stack Properties" palette) or using a script command:

```
set the mainStack of stack "my dialog" \  
to "my main"
```

If you need to go directly to a substack of another mainstack, add an entry to a stack's `stackFiles` property that includes the substack name and the name of the file that that stack is stored in. This technique is used in the MetaCard tools stack (the mainstack of which is the MetaCard Menu Bar) to go directly to substacks of the Help Directory stack.

An additional benefit of the main stack/substack architecture is that script handlers that must be callable from more than one stack can be stored in the script of the mainStack. These handlers can then be called from the scripts of any of the objects in any of the substacks. This is possible because messages sent to an object in a substack pass through that substack's mainStack, and then through the Home stack. This sharing of handlers is similar to that available via the `start` and `insert` commands, but is more selective because only messages sent to substacks go through the main stack's script.

This message passing architecture means that it's generally not a good idea to put generic handlers like `openStack` and `openCard` in the stack script of a mainStack with substacks, because these handlers will be run whenever any of the substacks opens or closes.

Menu Panels

Menu panels are just like any other stack, except that they are opened as `menuName Menus` by buttons rather than with a `go` or `topLevel` script command. The buttons in a menu panel must be set up in a certain way, however, in order to make them behave like Motif compatible menus.

The easiest way to create both menu panels and the menu items within them is to set a button's contents, and to let the button create the menu panel stack automatically when it is needed.

If you need to put controls in your menu panels that can't be defined with button contents, you must create a menu panel as a stack. The easiest way to do this is to clone one of the menu panels in the MetaCard development environment and then rename the buttons and rewrite the scripts to do what you need them to. For example, to clone the menu that pops up when you right click (control-click on the Mac) on a selected control:

```
clone stack "MC SelectedObject Menu"
```

After cloning a stack, you'll want to give it a unique name and then set the `menuName` of the button that will be used to open it to that unique name. You'll also want to edit the buttons in the stack to set their name and `script` properties. You can use the "Lay Out Panel" button in the **Utilities** stack to align and size the menu buttons after you have created the number of buttons you need.

See the `menuName` Menus card for information on how to create the button that will open the menu panel.

Cards

Cards are the simplest MetaCard objects. They have the fewest properties, and usually the smallest scripts (if they have scripts at all). Cards can contain as many buttons, graphics, scrollbars, groups and backgrounds as necessary. Double clicking with the pointer tool on a card will bring up the Card Properties dialog allowing you to apply new colors, change the font, edit the script, etc.

Of the messages sent to cards, one deserves special mention here: `resizeStack`. The `resizeStack` message is sent to the current card when a stack is opened, and whenever a stack is resized by the user. It can also be used to implement script-based Geometry Management (see that card for more details).

Groups and Backgrounds

Each card can have zero or more sets of controls on it. These sets of controls can be referred to either as backgrounds (the HyperCard and SuperCard compatible way) or as groups. Backgrounds are accessed in relation to the stack, whereas groups are accessed in relation to a card. So for example, *background 1* is the first group that was created in a stack, whereas *group 1* is the group with the lowest layer on the current card.

While this architecture can be difficult to understand at first, there are two benefits. The first is HyperCard compatibility. Many HyperCard, SuperCard, and MetaCard stacks make extensive use of backgrounds to minimize the duplication of objects. This is because the same background can be used on multiple cards. The second

benefit is the ability to use multiple groups on a single card, something HyperCard and SuperCard can't do.

There are three ways to create groups and add controls to them. The first is to use the **Edit Backgrounds** dialog to create a group and put the stack into `editBackground` mode. Any controls added or moved while in `editBackground` mode will become part of the new group.

The second way to create and edit groups is to create the controls that will go into the group first, and then select them by dragging over them or shift-clicking with the pointer tool and using the `group` command to group them (e.g., by choosing "Group" from the "Edit" menu). When you double click on the new group the **Group Properties** dialog opens and you can edit its properties or go into `editBackground` mode for that group by clicking on the "Edit" button.

The third way is to use the `create` command to create the group, and then use it create the individual controls directly into the group.

You can use the "Group" item in the "Edit" menu to group and ungroup controls at any time, but be sure you remember to regroup a set of controls again if you ungroup them. If you forget to do this and go to another card that had that group on it, the group will be permanently removed from that other card. The removal is reversible, however, since groups can be added to or removed from cards at any time using the **Edit Backgrounds** dialog or the **MetaTalk** `place` and `remove` commands.

Note that the layers of the controls within the group is defined by the order in which you selected the controls prior to grouping them. This is important if you want users of the stack to be able to use the tab key to move to those controls in a specific order.

Like other controls, groups have a layer property which governs the order in which they are drawn and the order in which the controls receive the keyboard focus when the tab key is pressed. It may be difficult to set the layer of a group in some situations. In these cases, selecting the controls that you want to go on top of the group and setting their layer properties is an easy way to change the layer of the group. You can also use the "relayer" button in the **Properties** palette or the **Control Browser** to change the layer of a group.

MetaCard supports nested groups (one group inside of another). The easiest way to manage nested groups is to stick to a single level of nesting and use `edit background` mode to edit the first layer and `group` and `ungroup` to edit the controls in the second level group.

Remember that cards don't have to have any groups (backgrounds) on them. Single card stacks, such as those used as dialog boxes, frequently don't.

Radio Groups and Traversal

MetaCard groups have built-in support for enforcing radio button behavior (only 1 button at a time is set to on). This behavior is automatic if you create a group that

contains only radio buttons. You can set the `hilitedButton` of a group directly to a value between 1 and the number of buttons in the group, or the `hilitedButtonName` to the name of the button you want hilited.

A related feature of groups is `tabGroupBehavior` property. Normally each of the controls on a card are given the keyboard focus in turn when you press the tab key. Setting the `tabGroupBehavior` of a group to true causes the *group* to be tabbed through, instead of the individual controls within it. The arrow keys are then used to move the keyboard focus among the controls in the group. According to the Motif Style Guide, groups of radio buttons and check boxes should have `tabGroupBehavior` set to true.

Buttons

MetaCard buttons are multipurpose objects. They can be used to start scripts or to open Menus. In addition to the standard configurations (set by the radio buttons in the "Style" group in the "Button Properties" dialog) most of the appearance and behavior properties can be set independently. See the Buttons card in the Properties By Object section of the MetaTalk Reference stack for details.

You may find it difficult to determine which properties need to be set to get a button to look or behave a certain way. In these cases try to find a button somewhere in the standard interface that does what you need and then copy and paste it. For example you could execute the following two commands in the Message Box to put a copy of the "File" button into the current topStack:

```
copy button "File"\  
of stack "MetaCard Menu Bar"  
paste
```

Here are some examples of button properties and how they can be used:

- A. Setting the `height` of a button to 4 makes it look like a menu separator.
- B. Setting the `autoHilite`, `showName`, `traversalOn`, and `opaque` of a button to false makes it work as a frame which can be placed around other controls, yet won't respond to keyboard or mouse input.
- C. Setting the `label` property of a button allows you to use a short string for the button name yet have the button display a long string for the user, making referring to a button by name in a script easier.
- D. Setting the `bottomMargin` of a button that it is also showing an icon moves the button label text farther away from the bottom of the button.

Menu Modes

Buttons can be used to open stacks as Menu Panels in one of five different modes:

pull-down, cascade, option, comboBox, and popup. In addition, there is a "tabbed" menuMode that is used to create tabbed/notebook dialogs.

Pull-down menus are the most commonly used type. They are operated by "pulling down" from a visible button in the stack. Menu panels opened as pull-downs can contain cascade menu buttons as well. Also known as "pull-right" menus, these buttons have an arrow indicating that a submenu can be opened by activating the button.

An option menu, which is also known as a drop-down list box, is used to select one item from a list of several items. A graphic element (an arrow or rectangle, depending on the current lookAndFeel setting) is drawn on the right side of the button to indicate its type to the user.

Popup menus "pop up" wherever the mouse button is depressed. Since there is no indication that the menu is available, the *OSF/Motif Style Guide* states that popups should only be provided *in addition to* other user interface techniques. Since experienced users can use keyboard accelerators even faster than they can use popups, it's a good idea to avoid using popup menus whenever possible.

Tabbed menus can only be created using button contents, and each line in the contents is used to create one "tab" in the area displayed at the top of the button.

Button Contents Menus

The easiest way to create a menu button is to set the button contents to a list of items that you would like to appear in a menu, and then set the button's `style` and `menumode` properties to the correct value for the type of menu you want. You would then put a `menuPick` message handler in the button's script to respond to the messages sent when the various menu items are chosen.

For example, you might do the following to set the button contents such that button 1 will display two menu items when it is clicked on:

```
put "one" & return & "two" into button 1
```

The button script would then probably have something like the following in it:

```
on menuPick which
switch which
case "one"
# do something for item "one"
break
case "two"
# do something for item "two"
break
```

```
end switch
end menuPick
```

There are several special characters that you can put at the start of a line in the button contents to indicate that an item should be something other than a simple rectangle button:

- makes a divider between groups of menu items
- !c makes the menu item a hilited check box
- !n makes the menu item an unhilited check box
- !r makes the menu item a hilited radio button
- !u makes the menu item an unhilited radio button
- (makes the menu item disabled

There are two other special characters that can appear anywhere in a line that change the behavior of that menu item. Putting the & character in a line makes the next character the mnemonic for the menu item: It will be underlined and activated if that character is typed with the alt key down while the menu is open. The / character makes the next character the keyboard accelerator for that menu item: the button will be activated if that character is typed while the control key is down, even if the menu is closed. Neither & nor / appear in the actual menu. To put a & or / character in a menu, double the character (&& or //).

You can build hierarchical (also call pull-right or cascading) menus by putting tab characters before the name. The depth of the submenus is determined by the number of tabs before the name, with the anchor item for a submenu being the line prior to an increase in the number of tabs. This means that the first line cannot contain tabs, and that any given line can have at most one more tab than the previous line had.

Note that none of the special characters appear in the message sent when a menu item is selected: The message is same as the item as it appears in the menu, not the corresponding line in the button contents.

You can change the size of the panel opened by drop-down list and combo box style buttons with the `menuLines` property. You can refer to buttons in the current `menuBar` using the term `menu` instead of `button`, and to individual menu items as `menuItem` instead of the term `line`. You can change the state of the menu items with the `disable`, `enable`, `hilite`, and `unhilite` commands. Use the `menuHistory` property to determine which menu item was most recently selected in a combo box or option menu, or to change the `label` of the button and positioning of the menu that opens.

menuName Menus

If your menu must have items other than simple rectangles, dividing lines, radio

buttons, or check boxes in it, you'll need to create a menu panel as a separate stack. See the **Menu Panels** card for instructions on creating and customizing the menu panel.

After creating a panel, you need to create the button that will open it. Create a button and set its style to "pulldown" using the properties palette. You'll also need to change the `menuName` to the name of your menu panel stack. You should also set the `mnemonic` property so the correct letter will be underlined on Windows and UNIX systems. In most cases this button will not need a script, as the scripts for the menu items will be in the buttons in the menu panel.

If you need to set up a button manually for your new menu, you will have to change some of the button's properties from their default values. The `autoArm` and `armBorder` properties make a button's border appear when the mouse is down in the button. The `rightMargin` property moves the `acceleratorText` into the button's rectangle (or out of it). And the `leftMargin` property allows you to align check boxes and radio buttons with buttons of other styles.

Fields

Text fields can be used in a number of ways. Their primary purpose is to display text, but they can also be used as containers in calculations and to retain information more permanently than is possible with variables.

For example, if you need to allow your user to edit a field which will later be restored to its original value, you can use a hidden field as a container of the original information and a second visible field as the container of the new information. To do this, first create a field called "editable text" which will accept the data your user inputs. Then create a field, name it "backup text", edit the contents of that field, and then hide it (which sets its `visible` property to false). When you need to restore your editable field, just put the contents of the hidden field into the editable field as shown below:

```
put field "backup text" into field\  
"editable text"
```

Fields can also be used as list boxes (set the `listBehavior` property of the field to true). For example, if you wanted to provide a list of files in the current directory you might put the following handler in the script of the first card in a stack (putting the command in an `openCard` handler ensures that the contents are updated every time that card is opened):

```
on openCard  
put the files into field "FNames"  
end openCard
```

You can also use fields for building hypertext links.

When fields are editable (`lockText` is set to `false`), clicking in them with mouse button 1 (usually the left mouse button) sets the text insertion cursor position at the current mouse cursor location.

Selecting a section of text in one field and then clicking with mouse button 2 (the middle button) will paste that selected text at the current mouse cursor location in a field. Clicks with mouse button 3 (the right mouse button) send `mouseDown` and `mouseUp` messages, allowing hypertext jumps using the `clickText` function even in unlocked fields.

Setting the `backgroundColor`, `topColor`, and `bottomColor` of a locked field in a `mouseDown` handler can be used to make a field simulate a button with the ability to use multiple fonts and colors for the label. To insure the screen does not flicker when using a field to emulate a button, use the `lock screen` and `unlock screen` commands to suppress updating the field appearance until all changes have been made.

Fields, and the text they contain, are easily formatted to meet your needs. The `formattedWidth` and `formattedHeight` properties can be used to determine the size of text in a field, so that you can resize the field (or even the stack) to make sure all the text in the field is visible without scrolling. Be sure that you have the field's `dontWrap` property set to `true` to get the width of the text before word wrapping. You can also use the `formattedText` property to import and export text such that the word-wrap locations are preserved.

Setting the `autoTab` property causes the cursor to advance to the next field when the return key is pressed in the last line of the field (normally the focus will only advance to the next control when the tab key is pressed).

Field Keyboard Functions

In the standard keyboard mode, MetaCard fields support the following keyboard functions:

arrow keys - move the cursor

Ctrl-arrow - move by words

Ctrl-A - select all text

Ctrl-B - backward character

Ctrl-C - copy selected text

Ctrl-D - delete character

Alt-D - delete word

Ctrl-E - end of line

Alt-E - end of sentence

Ctrl-K - kill to end of line (cuts text to clipboard)

Ctrl-N - next line

Ctrl-P - previous line

Ctrl-V - paste text

Ctrl-X - cut selected text

Ctrl-Y - yank (paste) killed (or cut) text

Ctrl-Z - undo

When the `emacsKeyBindings` property is set to true, the following standard Emacs keyboard functions replace the bindings specified above:

Ctrl-A - beginning of line

Alt-A - beginning of sentence

Ctrl-F - forward character

Ctrl-V - page down

Alt-V - page up

Note that navigation actions can be combined with the Shift key to select a range of text. You can also use the mouse to select text--drag selects by characters and double-click drag selects by word.

Some keyboards have keys for copy, cut, paste, and undo. Others have keys for page up, page down, home and end keys. In most cases these keys can be used in MetaCard fields.

To put extended (high bit set, e.g., á, æ, â, ö, ©) characters into the buffer, try using the **Character Chooser**. On UNIX/X11 systems, it is also possible to type these characters in by holding down the mod3 key. This modifier may be bound to the Alt Graph key, the Scroll Lock key, or the Num Lock key. Use the program `xmodmap` to see which key is bound to mod3. See your X Windows documentation for how to use the `xmodmap` utility to change or add the binding if your system does not support mod3 in a convenient way.

You can also enter high-bit set characters by quick pasting from another window, or using the `numToChar` function:

```
put numToChar(174) after field 2
```

Scrollbars

Scrollbars are the least complex of the MetaCard controls. All of their attributes and behaviors are described in the **MetaTalk Reference** stack. See the **Scrollbar Properties** card in the **Properties by Object** section of that stack for an index to the relevant properties. Also see the `scrollbarDrag` card in the **Messages** section

of the reference stack for a list of the messages that can be sent to scrollbar.

Images

Cards can contain as many images as you like, or none at all. Unlike HyperCard which limits the user to a single background and a single image (each the size of the card), MetaCard cards can have multiple images on them, and MetaTalk scripts have full control over the sizes, positions, and colors used in each image.

Images can be imported in most popular formats, but you should generally use only 2 formats for stacks you plan to distribute: GIF and JPEG. Other formats such as BMP, PICT, and XWD are not compressed, and using them will result in bloated, poorly performing applications.

For large projects, you should try to minimize the number of images imported into a stack. Instead, keep large images in separate files and set the `fileName` property of an image object to display that file at a particular place in your stack.

You should also minimize the number of images open at one time, and minimize the use of large multi-frame GIF files. Buffering these images requires large amounts of memory. Instead of importing and hiding a large number of images, create a single image object and use a script to change the `fileName` property of a that image. You can also use an image as a container and change its content by putting new data into it:

```
put myGIFdata into image "some image"
```

When you click with a painting tool in a stack that doesn't have an image, an image the size of the card is created. This image can be sized after it has been created by clicking on a painted-on area with the pointer tool to select it, and then dragging the size boxes. Note that when a stack is resized, the image is not resized. You'll have to resize the image either with the pointer tool, or by setting one of the size properties, e.g. by typing the following into the Message Box:

```
set the rect of image 1 \  
to the rect of this card
```

Images can be also be created with the "image" tool (next to the scrollbar in the Tools palette). Note that only the top most image (the image with the highest layer) can be edited if multiple images overlap. To edit the other images, you must either use the `hide` and `show` commands to hide the images above it, or set the `layer` of the image such that it is above all the other images.

Icons, Patterns, Brushes, & Cursors

MetaCard allows you to use any image object as an icon, fill pattern, brush shape, or cursor. The image that will be used when each of these properties is set is located by checking the id of the image objects. Images with ids less than 1000 are reserved for

the standard MetaCard icons, cursors, and fill patterns: cursors are ids 1-100, brush shapes 101-200, fill patterns 136-300. and icons 301-1000. The start value of each of these is subtracted when you set one of these properties to a value less than 1000. For example, setting an object's backPattern property to 2 causes the image with id 137 to be used to fill the background of that object.

All stacks and substacks are searched for an image of the correct id when you set one of these properties. To facilitate finding the right image, you can set the id property of image objects (they are the only MetaCard objects that allow this). To avoid the problem where you have multiple objects with the same id, you should only set ids to very large numbers (at least 6 digits). Note that if you set an id to a value used by another object in a stack (or another object is created with that id), one or both of the objects can be lost.

If you intend to distribute sets of custom cursors, patterns, icons, or brush shapes, contact MetaCard Corporation to reserve a block of ids for you to use to avoid conflicting with other users' images.

Images for patterns and icons are stored in two places in the standard distribution. Some are stored in the tools stack (mctools.mc contains the "Icons", "Cursors", and "Patterns" substacks), some in the Home stack (mchome.mc contains the "my icons" sub stack). Images stored in the Home stack can be edited by double clicking on the appropriate buttons in the **Icon Chooser** (accessed from the stack and button properties dialogs) and the **Pattern Chooser** palettes. Since the tools stack is not normally saved, editing images in this stack requires an extra step.

Note that if available, the tools stack (mctools.mc) or the mini tools (mcmini.mc) will be loaded when a standalone stack is run. If you don't want to distribute the tools stack with your stack, you'll need to make copies of the **Icons** and **Cursors** stacks and make them substacks of your stack. If you use icons that you have edited in the **My Icons** stack, you'll have to make a copy of that as well. For example, to put a copy of the "Cursors" stack into your stack use the following command:

```
clone stack "Cursors"  
set the name of stack "Copy of Cursors"\  
to "SS Cursors"  
set the mainStack of stack "SS Cursors"\  
to "standalone stack"
```

Since many systems won't accept large or odd sized images as cursors (some systems will only accept 16x16 pixel cursors) or fill patterns, MetaCard applications that must be portable should stick to smaller images that are multiples of 8 pixels in width and height.

Painting

The painting tools allow you to create and edit bitmap images. The best way to learn about painting a MetaCard image is by doing. In an unlocked stack (set the `cantModify` property to false or use the Stack Properties dialog), try out each of the tools in the **Paint Tools** palette. Use the undo menu item in the Edit menu to undo any messes you make.

You should also try out the **Color Chooser** and the **Pattern Chooser** palettes (both can be opened from the **Paint Tools** palette). Pressing F1 or the Help key when one of these palettes has the keyboard focus will bring up help on that palette. If after painting, you decide you don't want your picture, you can either exit MetaCard without saving, or type the following into the Message Box:

```
delete image 1
```

MetaCard has no text painting tool. Put text in a field underneath an image if you need to simulate painted text, or use the **Importer** to take a snapshot of text created in another application.

Tools

Here are several non-obvious techniques for those who like to paint. On the Mac, substitute "command" for "control", and control-click for the right mouse button:

1. Polygons created with polygon tool (bottom left of the **TOOLS** palette) can be closed either by clicking on the start point (use a big `gridSize` to make this easier), or by double-clicking.
2. Control clicking with the pencil tool brings up the magnifier window. Any of the painting tools can be used in this window. Control clicking again either in the window or on the image closes the window.
3. Dragging with the right mouse button using any of the tools erases using the current tool. Right-click with the bucket tool to erase the background of an imported image.
4. Double clicking on the brush, eraser, and spray-can tools brings up the **Brush Chooser** palette.
5. Holding down the shift key with some tools constrains the angles (try changing the "slices" field in the **Paint Properties** palette).
6. Holding down the control key while dragging with some tools draws the border of the object with the current `brushColor` (or `brushPattern`).
7. Holding down the control key when clicking with the dropper tool sets the brush color instead of the pen color.

Graphics

You can create a graphic with any of the filled shapes in the MetaCard Menu Bar or Tools palettes. Rather than have each object type (e.g. circle, square, polygon, etc.) have a separate MetaCard control, they are just different styles of the graphic control.

One of the most useful characteristics of the graphic control is that you can set the `points` property of the graphic dynamically from a script. See the scripts in the MetaCard Demo stack and in the MetaTalk Examples for some ideas. Note that when setting the `points` property of a graphic, leaving a blank line in the specification causes a "pen up" action, making it possible to create multiple non-connected line segments with a single graphic control.

You can optionally draw another series of line segments at each point in the graphic objects. These shapes are known as markers, and have their own list of points and colors. See the `markerPoints` property for more information.

There is no "text" style graphic, but by setting the `name` or `label` property of a graphic to a string, the `lineSize` to 0, and the `showName` property to true, you can make the graphic show just a text string. There is no "line" style either, but you can set the `points` of a "polygon" style graphic to a pair of points to draw a single line segment.

Players

The player control is used to play sound and movie files. Movies are played into the player's `rect`, and the player can display a controller bar that works with both sound and movie files. For sound files that don't require a controller, either set the `showController` property to false, or hide the player object.

After creating a player and setting its `fileName` property to the movie or sound file to play, you can start it with the `start` command. To play only a portion of a movie or sound file, create a selection by setting the `startTime` and `endTime` properties, and set the `playSelection` property to true.

A `playStopped` message is automatically sent when the `currentTime` reaches the `endTime` or the `duration` (whichever comes first), but you can arrange to have messages sent at other times by setting the player's `callbacks` property. A `selectionChanged` message is sent if the

The player object requires QuickTime 3 or later on MacOS and Windows. It will operate with limited functionality (AVI and WAV files only) on Windows if QuickTime is not available.

On UNIX/X11 systems, the player control uses the open source XAnim movie player. This means that the `xanim` binary is distributed as a separate executable, and that executable must be some place on the current `$PATH` before MetaCard will be able to play a `videoClip`. XAnim supports a variety of animation and video formats,

including AVI, QuickTime, and MPEG formats, but does not support a controller bar or selections.

EPS objects

On system with the Display PostScript extension (Sun SPARC Solaris and DEC Alpha), MetaCard can display PostScript files using the EPS object. Unlike the graphic control type, this object can't be created with a special tool. Instead the EPS control is created with the `import` command (the `Importer` dialog), or using the `create` command. After an EPS object has been created it can be moved and sized just like other control types.

Three different types of PostScript files can be used with the EPS object. The primary functional difference is that EPSF and EPSI files contain a `BoundingBox` comment that defines the coordinate system used. If you import a plain PostScript file, you'll probably have to set the `BoundingBox` coordinates yourself using the `EPS Properties` dialog.

You can set the `postScript` property from within a script, making it possible to generate graphs dynamically using graphing packages such as `gnuplot`, for example.

Keep in mind that only the PostScript in EPS files is retained (previews are discarded), so you'll only be able to view and edit EPS objects on systems that support the Adobe Display PostScript extension.

AudioClips

`AudioClips` hold audio data in MetaCard stacks. They are not controls, therefore they don't have any visual representation. `AudioClips` are accessed by name with the `play` command or created with the `import` command.

Sounds can be imported in WAV, AIFF or the Sun/Next (.au) format, or from HyperCard stacks in 8 bit linear format. Note that the Sun/Next 8-bit `mulaw` (mulaw) format is the only format supported on all platforms. In most cases, you should use `players` to play external sound files instead of importing audio clips because this reduces memory requirements.

VideoClips

`VideoClips` are objects used to hold frame-based animation into MetaCard stack windows. They can be created with the `import` command which moves the clip into a stack. Importing is only recommended if being able to distribute fewer files is worth the performance and size penalties incurred by importing a clip.

`VideoClips` don't have any visual representation when not playing (they aren't controls), and are usually only accessed by name with the `play` command. Temporary `players` are created to play back the clip at that point.

See the MetaCard FAQ list for more information on playing movies in MetaCard.

Scripting

Each MetaCard object can have a script made up of one or more message or function handlers. The **MetaTalk Reference** stack has descriptions of these two types of handlers, as well as descriptions of the standard messages, and of the commands, functions, and properties used in handlers.

Scripts are generally written using the Script Editor window which is accessible from the properties dialog of each object. You can also use the right mouse button to get to the objects script. The stack script can be opened directly with the shortcut control-alt-s and the card script with control-alt-c.

You can also edit the script of a particular object by typing something like the following into the **Message Box**:

```
edit the script of field 1
```

Although scripts can be edited by other scripts, this practice is discouraged. When a stack is run from the command line (i.e. without a Home stack), there are limits to the number of statements that can be in a script that is set from the scripting language. In most cases using the `do` command is a more straightforward way to achieve this dynamic behavior.

Be sure to document your scripts while you are writing them. Comments are set off from MetaTalk statements with either the '#' character or by the sequence '--'. Both of these cause MetaCard to skip to the end of the line:

```
put 10 -- this is one comment
get 10 # puts 10 into it
```

If you need to break very long statements into multiple lines to improve readability, use the backslash character:

```
put the name of button 1\
of card 2 of stack "My Stack"\
into somevariable
```

Terms

MetaTalk programming uses several basic terms: **Constant**, **Literal**, **Container**, **Chunk**, **Factor**, **Expression**, and **Properties and Custom Properties**. The most common use of many of these terms is in the syntax of a command or function in the **MetaTalk Reference**.

Container

MetaCard stores text and numbers in "containers" of which there are five types:

fields, buttons, images, variables, and urls. Fields not only *display* text and numbers, they can allow users to *edit* this information. Fields are specified either by name or by number. For example the following commands would have the same result if the field named "Addresses" is the first field on the current card:

```
put field 1 into it
put field "Addresses" into it
```

In general, it is a good practice to give your fields names and to use these names in scripts rather than the field number, since the number of a field can change (by setting the layer of another field to a value below it), whereas names won't.

Note that text attributes are not retrieved or saved when you use the field as a container. For example, if you put line 1 of field 1 into field 2, and the third word of line 1 has the bold attribute, that attribute will *not* carry over to field 2. If you must retain attributes, it may be possible to use the fields' `htmlText` property to move text from one field to another, but not all font and style information will be retained with this method. If all attribute information must be preserved, use the `select` command with the `cut` or `copy` commands to move text to the clipboard, and then `paste` it back into another field:

```
select text of field "source"
copy
select before char 1 of field "dest"
paste
```

When buttons are used as containers, the text they contain is usually used to build **Button Contents Menus**. They can be also be used as general-purpose containers, however. Images can be used as containers for binary image data such as images downloaded from from WWW servers.

Variables can also be used to store values, but they cannot display the information. Variables are created by either putting something into them, or by using the `global` or `local` keywords. For example all of three of these statements create a variable called "myVariable":

```
put 10 into myVariable
local myVariable
global myVariable
```

The difference among the variables created is that the `global` statement creates a variable that is shared between handlers whereas the `put` command and the `local` statement create variables that can only be used within the handler in which they appear. Note that every handler that uses a global variable must have a `global` statement that names the variables. See the `local` and `global` commands in the [MetaTalk Reference](#) for more information.

Variables with a \$ character as the first character in the name are exported to MetaCard's environment which is inherited by processes started with the shell function and `open` process command. You can also get the value of an environment variable to find the current users log in name (`$LOGNAME`), home directory (`$HOME`), and many other current settings. Environment variable names are all caps by convention, and you don't need to declare them with the `global` command before using them.

Any variable can be made into an array of containers indexed with a string or number by putting the index in brackets. For example, the statement `put 10 into somevar[3]` will put the number "10" into element "3" of the variable named "somevar". Use the `keys` function to determine which elements of a variable have values, and the `delete` command to delete individual elements.

It's a good idea to develop a personalized naming scheme for variables. For example, you might use the letter "t" as the first character in local variables (e.g. `tLocal`) and the letter "g" or name of a stack as a prefix for global variables (e.g. `gVariable` or `MetaTalkRefGlobal1`). There is no practical length restriction on variable names, and this type of naming scheme should reduce conflicts between scripts in different stacks that may otherwise try to use the same variable name for different things. It will also ease the process of upgrading to newer versions of MetaCard which will include new function and property names that may conflict with your variable names if you don't use such a naming system.

The final container type is the URL (Uniform Resource Locator). These can be files on your local system, or files on HTTP servers on your local network or the Internet. The four forms of the URL are "file:filename", "binfile:filename", "resfile:filename" and "http://some.machine/somefile". The first three forms access files on your local system. The "filename" parameter can include a path to the file of the form "drive:/directory/file". You should always use the URL standard "/" character to separate the directories in a URL.

```
put field 1 into url "file:test.txt"
put field 1 into url "file:c:/mc/test.txt"
put line 3 of url "file:test.txt" into field 1
```

The "file" URL type reads or writes the file in text mode. If you need to do a binary read or write, use the "binfile" or "resfile" types, the latter of which is only available on MacOS systems and which accesses the resource fork.

The "http://" form of URLs is used to download documents from WWW servers or to post documents back to them. You can use the `load` command to prefetch files that you'll need later, or just access them directly if you know that the wait won't be too long:

```
put url "http://www.some.org/somefile" into field 1
post field 1 \
```

```
to url "http://www.some.org/cgi-bin/script.mt"
```

Constant

A constant is a word that is defined in MetaCard to have a particular value. For example, the value of the constant `five` is 5. Refer to the **Constants** section of the **MetaTalk Reference** for a list of the predefined constants.

Literal

A literal is like a constant, the difference being that *you* define what it means. Literals can be numbers or strings. Literal numbers can be decimal numbers like 123 or real (floating point) numbers like 123.456. If the first two characters in a number are 0x (as in 0x123), the number is interpreted as hexadecimal (base 16). If the `convertOctals` property is set to true, a number with a leading zero will be interpreted as an octal (base 8) number. Literal numbers can be either quoted or unquoted.

Literal strings should always be enclosed in quotes. For example the statement:

```
put "hello" into field 1
```

will put the word "hello" into the first field on the current card. The following statement is equivalent:

```
put hello into field 1
```

but this way of specifying a literal is not recommended because:

```
put "four" into field 1
```

will put the word "four" into the field whereas

```
put four into field 1
```

will put the number "4" into the field since the word "four" is a MetaTalk constant. Unless you know every word in the MetaTalk vocabulary, being lazy about putting quotes around your literals will eventually result in producing a script that fails to do what you want. Setting the `explicitVariables` property to true will cause all unquoted literals to generate script errors.

Chunk

Chunk expressions are used to retrieve a small piece of text from a larger string. For example the command:

```
add 10 to the second word of \  
line 3 of field 1
```

adds 10 to a word (which must be a number) within a field.

There are five types of chunks: line, item, word, token, and character (or char). Line

chunks are separated by the return character; items by a comma (but see the `itemDelimiter` property); and words by spaces, tabs, or returns. The size of token chunks is defined by the MetaTalk language interpreter. In general, each continuous string of numeric or alphanumeric characters is a token, as is each punctuation mark.

When combining chunks of different types, they must be specified in smaller to larger order in the chunk statement:

```
char 1 of word 2 of item 3 of line 4
```

Chunk expressions are an extension of the normal method of specifying an object which also requires specifying types in "smaller" to "larger" order:

```
add 5 to line 1 of field "ZIP" of \  
card 1 of stack "Addresses"
```

(the "\" is a script continuation character, use it when a script statement gets too long for one line).

Chunk expressions can also specify ranges. For example, if field "ZIP" contains the string "80306", the following statement will add 10 to the number "03", and field "Zip" will then contain the string "81306":

```
add 10 to char 2 to 3 of field "ZIP"
```

Chunks can be specified using negative numbers, in which case the measurement is taken from the end of the string. For example "char -1" is the last character in a string, and "word -2" is the second-to-last word.

The number of chunks of a given type in a string can be determined using the number function. Some examples:

```
put the number of chars in it  
put the number of buttons on card 1  
put the number of lines in field "ToDo"
```

You can also specify chunks using ordinals instead of putting a number after the chunk type:

```
the second word of field 1  
the last item in it
```

To choose a chunk at random, use the word "any" as an ordinal:

```
put any char of it into randomchar
```

Factor

Factors are values that have no binary operators. Constants, literals, chunks, and functions all return factors.

There are two situations in which operations are performed on factors rather than

whole expressions, and it is important to distinguish between a factor and an expression in these cases. The first case in which you would need to make that distinction is with chunk expressions. For example, the chunk expression:

```
A) put field 1 + 10 into it
```

will put the contents of field 1, added to the number 10, into the local variable `it` rather than the contents of field 11 (since chunk expressions take factors). If you wanted field 11 rather than field 1 you would use parentheses:

```
put field (1 + 10) into it
```

The second situation in which you need to make a distinction between a factor and an expression is with functions that use `of` rather than parentheses (known as single parameter Functions):

```
B) put the sqrt of 4 + 5 into it
```

which puts 7 (not 3) into the variable `it`.

Expression

Expressions are made up of factors connected with binary Operators. See the [factor](#) card for descriptions of when it is important to distinguish a factor from an expression.

Properties and Custom Properties

All MetaCard objects have properties that you can get and set to change the appearance, contents, and behavior of the object. There are also many global properties, which when changed affect all objects in all stacks. For example setting the `CURSOR` property changes the cursor in all open stack windows.

Properties must be set using the `set` command, but can be retrieved using any type of expression (the `get` command can also be used to get the property and put it into the local variable "it"). In most cases, the important properties are set using a properties dialog box, but these dialogs just use the same `set` command your scripts must use.

You can get and set all of the important properties for an object at the same time using the `properties` property.

In addition to the predefined properties described in the [MetaTalk Reference](#), you can attach custom properties to any MetaCard object. Custom properties are usually used to store information with an object that must be persistent. For example, you might save version information or a date on a stack to keep a record of when it was modified. This could also be done by putting information in a hidden field (a commonly used technique in HyperCard), but using custom properties is more convenient and makes stacks easier to maintain. You can get a list of the properties that have been set for an object with its `customKeys` property.

Custom properties are organized into `customPropertySets`, which are accessed by setting the `customPropertySet` property, or by using an array syntax to refer to the custom property. For example, to access element "myprop" in a custom property set named "myset", use the expression:

```
myset [myprop]
```

You can copy an entire property set into an array with the `customProperties` property. You can get a specific set by requesting it using an array syntax:

```
get the customProperties[myset] of btn 1
```

Custom properties can also be used to implement behaviors that are triggered when a property is set, or to implement "virtual properties" that are calculated rather than actually stored with an object. You can create handlers that are called when custom properties are set or retrieved using **Setprop Handlers** and **Getprop Handlers**.

You can use variables to refer to both standard and custom properties. The following will put the rect of a stack into the Message Box:

```
get "rect"  
put the it of stack "mystack"
```

Handlers

Any MetaCard object (stack, card, group, button, menu, field, etc.) can have a script. Scripts are composed of one or more handlers, one for each of the messages that an object needs to respond to. Handlers are composed of a series of MetaTalk statements, each statement ending with a return or semicolon.

Messages not handled by an object are passed up the object hierarchy until a handler for that message is found. Messages sent to controls in groups pass from the control, to the group, to the card, to the stack, to the mainStack (if the stack is a substack), and then to the Home stack. Messages sent to controls on the card are passed from the control, to the card, to *each* of the backgrounds (groups) on that card, to the stack, to the mainStack (if the stack is a substack), and then to the Home stack.

There are two sources of messages. The first is the MetaCard engine, which sends messages whenever an action is taken by the user. For example, when the user clicks the mouse button down on a control in a stack, a `mouseDown` message is sent to that control.

MetaTalk scripts are the second source of messages. Scripts send messages either using the `send` command, or by just putting the handler name where a command would normally go. This latter method of sending messages is commonly known as calling a subroutine, and these subroutine calls can be made to handlers within a script, or to handlers in objects higher in the message-passing hierarchy. Using this technique you can avoid having to redefine a common series of operations in each script. Instead, just put a single handler in the script of an object (group, card, or

stack) higher in hierarchy.

Each handler can have zero or more parameters. Parameters are just like local variables except that their value is set before a handler begins execution. For example, in the following handler the stack is closed only if mouse button 2 is pressed and then released:

```
on mouseUp which
if which is 2
then close this stack
end mouseUp
```

In this case the number of the button the user released is put into the parameter "which" before the handler is called. If more than one parameter is required they are separated by commas.

Putting an @ character before the name of a parameter makes it a call-by-reference parameter, which means assigning a value to that parameter in the subroutine changes the value of the variable in the calling handler. Note that a variable must be used in the call to the subroutine when using call by reference parameters (you can't use an expression on the command line, nor can you pass a field). For example, setting the script of a button to the following and then clicking on the button will put "10" into the **Message Box**:

```
on setvar @which
put 10 into which
end setvar
on mouseUp
put 1 into somevar
setvar somevar # changes somevar
put somevar # puts "10"
end mouseUp
```

There are actually four types of handlers: **Message Handlers** such as the mouseUp handler shown above, **Function Handlers** which are used when a handler must return a value to a calling handler, **Setprop Handlers** which are called when setting custom properties, and **Getprop Handlers** which are called when getting custom properties.

Message Handlers

Message Handlers are statements beginning with the word on. The handler responds to the message following the word on -- mouseUp is the most common message -- but you can call a handler by placing the messageName (either a system message or

the name of a MetaTalk subroutine) as the first word on a statement line. For example, the command:

```
on returnInField
mouseUp
end returnInField
```

sends the mouseUp message up the hierarchy when the keyboard return key is pressed and a field has keyboard focus (the flashing bar cursor). You can also use the send command to send a message to a particular object. For example the following handler:

```
on returnKey
send mouseUp to button "OK"
end returnKey
```

sends the mouseUp message to the button named "OK" when the return key is pressed if no field has the keyboard focus. This handler is used to implement the default button in a dialog box, a keyboard shortcut. Be sure to set the default property on the button you will be sending the message to so that the user will be able to tell which button will be activated.

Function Handlers

The difference between a function handler and a message handler is that a message handler is a statement, (it is the first word on a line in a MetaTalk handler), whereas a function handler is called as part of an expression. Here is an example of a function handler:

```
function factorial x
if x <= 1
then return 1
else return x * factorial(x - 1)
end factorial
```

This handler is a recursive function (it calls itself) for computing factorials. To call this function you would write something like:

```
put factorial(5) into somevariable
```

This statement would call the function factorial, which would call itself 4 times, and put the resulting number (120) into a variable called somevariable.

A function handler should always have a return statement followed by an expression. The value of the expression is what is returned to the calling handler.

Setprop Handlers

Whenever a custom property is set, a message is sent to the object whose property is being set. You can write a setprop handler to catch this message and do validation on the new value of the property before it setting it. For example, if you wanted to make sure a custom property named "percent" was never set to a negative value, you could put the following script into the object's script:

```
setprop percent x
set the percent of me to max(0, x)
end percent
```

Note that when you set an object's property from within a setprop handler, setprop messages are not sent. This applies not only to the property being set, but also to any of that object's other custom properties.

You can also use a setprop property to implement "virtual properties" which are never really attached to the object but instead are triggers that an object should take some action. While in most cases this same behavior could be achieved using the send command, using custom properties makes the communication bi-directional: values can be returned to a calling handler by implementing Getprop Handlers.

Getprop Handlers

Whenever a custom property is used in an expression, a message is sent to the object whose property is being retrieved. Handlers for these messages are most often used to implement "virtual properties", which are properties that are not actually stored with an object, but are calculated when needed.

For example, if you wanted to be able to easily get the width of any control on a card as a percentage of the width of its parent object, regardless of whether the control was in a group or directly on the card, you could put the following handler in the card script:

```
getprop percent
return the width of the target * 100\
div the width of the owner of the target
end percent
```

This handler only needs to be in the card script because like regular messages, setprop and getprop messages are passed up to the parent object when not handled by the target object.

Debugging Scripts

If a script or a group of scripts fail to do what you intended, then you'll have to debug them. In general, spending extra time to write your scripts carefully in the first

place will save time in the long run, since debugging can be a very time consuming process. A well thought-out overall stack design can help too, since many bugs arise when some scripts depend on other scripts. These kinds bugs tend to be harder to find and fix because fixing a bug in one script will sometimes cause a different but to appear in another handler.

Sometimes when a script fails, it will get stuck in an "infinite loop", where the termination condition of a repeat loop never evaluates to "true". To break out of these loops on Windows or UNIX systems, press control-. (the control key and the period key), while a MetaCard window has the keyboard focus. On MacOS systems, the abort sequence is command-. On UNIX systems, you can also abort a script by running the command "kill -1 <pid>" where <pid> is the MetaCard process id.

The most commonly used debugging technique is to use the `put` command to periodically put debugging information into the **Message Box**:

```
put "variable var1 was" && var1
```

If these debugging messages go by too quickly, a `wait` command can be put after each one to delay execution long enough for you to read each message. Another alternative is to put the text *after* the previous message in the message box or into a scrolling field you've created for this purpose:

```
put var1 & return after \  
field 1 of stack "debugger"
```

More serious problems can be located using the Script Debugger, which can be opened from the "Tools" menu.

Improving Performance

After you have your scripts working, it may be necessary to fine tune them to make them run faster. In most cases, you can improve performance merely rearranging the statement scripts slightly. Here are some ways to do this:

1. Use variables instead of fields as containers whenever possible.
2. Precompute complex expressions. Any value that is referred to in your script more than once should be precomputed and put into a variable. If this value is needed over several executions of the script, consider storing it in a global variable or custom property.
3. Take statements out of loops whenever possible including the termination condition in an "until" repeat loop. For example, to look at every third line in a field, instead of:

```
put 3 into i  
repeat until i >= the number of lines\  
in field 1
```

```

# do something with line i of field 1
add 3 to i
end repeat
use:
put field 1 into fcontents
put the number of lines in fcontents\
into nlines
put 3 into i
repeat until i >= nlines
# do something with line i of fcontents
add 3 to i
end repeat

```

4. Use `repeat for each` when looping over chunks. For example, the following repeat loop will run much faster than the previous one:

```

put field 1 into fcontents
repeat for each line l in fcontents
# do something with variable l
end repeat

```

5. Put the conditions most likely to match first in `switch` statements and multiple `if-then-else` statements. Similarly, when using `if-then` with multiple expressions connected with the `and` or `or` operators, put the conditions most likely to match first when using `or`, and least likely to match first when using `and`. The fewer conditions that must be checked before a match is found, the faster the script will run.

6. Use associative arrays and custom properties instead of line or item chunk expressions and the `lineOffset` function to store and retrieve multiple chunks in the same container.

7. Refer to remote cards and stacks by name or number instead of going to them when you need to get a value from another card or stack.

8. Set `lockScreen` to true when performing more than one operation on the controls on the same card.

9. Use in-line statements to compute values instead of calling functions. Note that this may make your scripts harder to maintain, so only use this technique when performance is an utmost priority.

If implementing these tips doesn't speed up the scripts enough, you may have to use **External Commands and Functions** to speed up the execution even more.

Geometry Management

When a user resizes a stack, it is frequently desirable for the controls in the stack to be resized and/or repositioned to make maximum use of the available space. This is especially important for fields, which can display more text if they are resized larger.

The process of resizing controls in a `resizeStack` handler is called Geometry Management (the term is borrowed from the Xt toolkit process with the same goal). While writing a geometry management script for each dialog can be tedious, at least it's deterministic. The constraint based approach taken by the Xt toolkits is frequently incapable of managing complex layouts, requiring a fall back to the algorithmic approach taken by MetaCard.

The general approach to the problem of horizontal layout is to add up the widths of the objects, subtract that value from the total width of the stack and divide by the number of objects + 1. The result is the amount of space between an object and its neighbors. You should then set the `loc` of each of the objects to the appropriate value. The following is the geometry management script used in this stack for you to use as a template:

```
on resizeStack
get the rect of this card
add 8 to item 1 of it
add 32 to item 2 of it
subtract 8 from item 3 of it
subtract 48 from item 4 of it
set the rect of field 2 to it
put 0 into item 2 of it
put 32 into item 4 of it
set the rect of field 1 to it
set the right of button 1\
to the width of this stack - 48
set the right of button 2\
to the width of this stack - 12
repeat with i = 1 to the number of buttons
set the bottom of button i\
to the height of this stack - 12
end repeat
end resizeStack
```

See also the scripts in the MetaCard dialogs that have layout and behavior similar to what you're trying to achieve.

More elaborate geometry managers could use the `formattedWidth` and `formattedHeight` properties to determine the minimum size for buttons and fields in order to allow resizing of the objects in addition to moving them. These properties can also be used to resize stacks if all of the text in a field must be visible when the stack is opened.

Hypermedia

Hypermedia is the technique of linking sounds, images, and text to on-screen objects. The simplest form is hypertext, the linking of a word or phrase to another word or phrase. MetaCard's multiple card metaphor makes it easy to develop hypermedia documents and applications. You can simply place different topics on different cards, and use the `go` command to go to a card based on the user's choice.

There are several different ways the user can choose a topic. The easiest way (for the user, anyway) is to have buttons labeled with the destination. For each button you might write a script of the form:

```
on mouseUp
go to card "some topic"
end mouseUp
```

The disadvantage of this approach is that for some stacks a large number of buttons would have to be created. You could make this somewhat more efficient by giving your buttons the same name as the target cards and substituting the `short name` of the target for "some topic" in the above example. You would probably also want to put this handler into a card or stack script to reduce redundancy.

You could put the same type of handler into a field's script, but there is a more powerful technique. See the **Building Hypertext Links** card for more details.

Building Hypertext Links

There are several ways to build hypertext links in MetaCard. The easiest is to set the `textStyle` of the source text to `link` and handle the `linkClicked` message that will be sent when that text is clicked on. If you need some information other than what you can derive from the text clicked on, set the `linkText` property of that block of text and it will be passed with the `linkClicked` message instead of the text that was clicked.

A second method for building hypertext links is the `clickText` function. This function returns the text string that a user clicked on. If the text string is the name of a card, then you can just go to that card. For example:

```
on mouseUp
```

```
if the clickText is not empty
then if there is a card the clickText
then go to card the clickText
end mouseUp
```

It is important to check to see if there is a card with the correct name, so that if the user clicks on a word that does not have a destination, nothing happens. If your script doesn't check, an execution error may occur.

If your cards have multi-word names, you will have to group these words together using the `link` style (from the Style menu in the Menu Bar). That way, when any word in the phrase is clicked on, the whole phrase is returned.

There are two other techniques that can be used to implement hypertext links: indexes and the `find` command. An index could be kept in a field containing a list of paired items. For example, the first item would be the words that a user might click on, and the second item would be the destination card name (or number). To use the index, use a `repeat` loop to go through the lines of the index, looking for the word to match. When a match is found, go to the card specified by the second item in the index entry.

An index could also be stored as a custom property on a stack or card. The MetaCard help system uses this technique to link pages in the **MetaTalk Reference** to relevant examples in the **MetaTalk Examples** stack.

The second technique is to use the `find` command to find the `clickText` in another card. A more sophisticated version of this is to have a "keyword" field on each card that can be specified in the `find` command. For example, the keyword field might contain the words that are defined on a given card.

Sound

MetaCard supports a limited form of the HyperTalk `play` command. In MetaCard, the `play` command plays an audio file using the system speaker. See the **AudioClips** card for more information on storing sound files inside a stack.

Since the sound plays in the background, it is possible to play sound while displaying an animation. Though no synchronization is supported (other than the `sound` function which can be used to determine if a sound has finished playing), a little experimentation will usually yield a satisfactory animation/sound combination.

It will be necessary to use vendor supplied programs to record sound although the external program can of course be run via a MetaTalk script. Note that sound is the least portable of MetaCard features: many systems don't support sound, nor do most X terminals or PC X servers.

Animation

There are two basic types of animation possible in MetaCard: object based and frame based. In **Object Based Animation**, objects are moved around by setting their location properties or using the `drag` or `move` commands. In **Frame Based Animation**, the whole scene changes over time. This can be done with the `hide` and `show` commands, using `VideoClips`, by displaying animated GIF images, or with an external player.

A fourth type of animation is based on changing the `cursor` property. This is usually done to provide feedback to the user about the status of an operation. Two common uses are changing the cursor in a loop to make waiting more pleasant for the user, or to provide feedback for a drag-and-drop operation (the cursor changes to indicate that an object is being dragged).

Keep in mind that each type of animation has its strengths and weaknesses, but one thing is shared: producing effective animation is a time consuming process, and should not be attempted on a tight schedule.

Object Based Animation

Object based animation is the easiest and most flexible type of animation. The simplest example of this type of animation is the setting of an object's position in a loop:

```
on mouseUp
repeat with i = 50 to 100
set the loc of button "mover" to i,i
end repeat
end mouseUp
```

This script moves a button in a diagonal line from point 50,50 to point 100,100. A more interesting example is to capture points of a path beforehand, and then set the button position to each of the points in turn, by setting the `loc` of an object to the lines in a field in turn:

```
on gather
wait until the mouse is "down"
repeat while the mouse is "down"
put the mouseLoc & return\
after field "coords"
end repeat
end gather
on playback
```

```
put number of lines in field "coords"\  
into tlines  
repeat with i = 1 to tlines - 1  
set the loc of button "mover"\  
to line i of field "coords"  
end repeat  
end playback
```

The problem with this approach is that it is not speed compensated: it will run faster on some hardware than others. Use the `move` command when an animation must play back at the same speed on all hardware.

A final type of object based animation is to set the `icon` property of buttons within a matrix to various values. This type of animation is not speed compensated, but does allow multiple objects to be animated, and allows the object's appearance to change while it is moving.

Frame Based Animation

The most common type of frame-based animation playback system are video clip players like QuickTime and Video For Windows (which uses the AVI format). You can use the `play` command to play back clips in appropriate formats on all platforms. On Windows systems, you can use the `mciSendString` function to play back QuickTime movies. See tools.metacard.com for examples.

MetaCard can also play back animated GIF images. Playback starts automatically when images are opened, but you can control the presentation by setting properties such as `repeatCount` and `currentFrame` of the image object.

The simplest form of frame based animation is to draw an image into one card, clone the card a few times, alter the image slightly on each card after the first, and play the animation using the `show` command (flip book animation):

```
show 10 cards
```

If the animation can be slower, `visual` effects can be used with the `go` command to provide transitions between them. If speed is a problem, the images can all be placed on the same card and the `hide` and `show` commands used to flip between them. Using the `lock` command to lock the screen may make this even faster.

The Outside World

The `mc` command accepts any number of command line arguments. MetaCard attempts to load each of these arguments as stacks, ignoring those that can't be loaded. You can use this capability to pass command line options to your stacks.

All arguments are available from within the scripting language by prepending a `$`

before the number of the argument, with \$0 being the first argument on the command line. Note that if you load a stack by specifying only the stack's file name on the shell command line, \$0 will be the name of that file.

MetaCard can also access the environment by prepending the environment variable name with a \$. Note that all environment variables must be in upper case letters. For example, to get the PATH environment variable:

```
put $PATH
```

To set it:

```
put "/bin:/usr/bin" into $PATH
```

For data interchange with other applications, MetaCard can store and retrieve plain text from the system clipboard. Cut, copy, and paste all retain attribute information when moving data from place to place *within* MetaCard, however.

On X systems, MetaCard also supports "quick paste" for moving text from one application to another. To paste text from another window into any unlocked MetaCard field, select the text in the other window and then click with the middle mouse button at the location where you want the text to go. Text can be copied from a MetaCard field to another window by selecting the text in the field and clicking with the middle button in the destination window (but only if the destination application supports quick paste).

Importing HyperCard and SuperCard stacks

To import a HyperCard stack into MetaCard, first compact the stack in HyperCard to remove the extraneous data from the stack. Then just open the stack using "Open..." from the File menu. If you're running MetaCard on a UNIX or Windows system, you can import stacks in either MacBinary or BinHex formats.

To import a SuperCard project into MetaCard, use the SuperCard to MetaCard converter. This converter has two components. The exporter component writes out the SuperCard project to a set of text and image files, which are then used by the importer component to construct a MetaCard stack. This converter is available on the MetaCard WWW/FTP sites and via email from support@metacard.com.

After a stack or project has been imported, in most cases some changes will need to be made to the scripts and object properties to get the stack working properly in MetaCard. If there are problems compiling scripts, the "script errors" dialog will open and present you with a list of the objects with script problems. Double clicking on the items in this list will bring up the script editor and identify the location of the error.

The following are the most commonly encountered scripting and object display problems:

- 1) MetaCard's language compiler is much stricter than HyperCard's and SuperCard's.

Using function and property names as variables is the most likely source of problems. For this reason you should develop a naming scheme where variable names won't be confused with property and function names. For example, you might have all global variable names start with a "g", and all local variables have a "t" or "l" as their first character. The "colorize script" option in the MetaCard Script Editor can often be used to identify cases where MetaCard's script compiler is interpreting token names differently than HyperCard's or SuperCard's.

2) The MetaCard script compiler also requires more care in the use of prepositions. For example, property and function names must **always** be preceded by the word "the". Because the word "in" is used in many MetaTalk command structures, it is not interchangeable with the word "of" like it is in HyperCard.

3) The strictness of the MetaCard script language compiler also affects operation of the **do** command and **value** function. In some cases additional quotes and other punctuation will need to be added to the strings passed to these commands.

4) MetaCard's cross-platform support for menus is not completely compatible with either HyperCard's or SuperCard's menu architectures. In most cases commands that refer to menus will need to be modified.

5) Unlike HyperCard and SuperCard, you can't override standard command and function names in MetaCard by creating handlers of the same name. Any handlers that have the same names as built-in MetaCard commands or functions will never be executed.

6) Because Windows and UNIX systems have different fonts from those used on the Mac, in some cases buttons and fields will need to be resized, or their fonts changed.

7) MetaCard supports HyperCard-compatible XCMDs, but these are not moved to the resource fork of a stack when it is imported. You must first save the imported stack, move the external into the stack using Resedit or some other resource editor, and then close and reopen the stack. Before you do this, however, check to make sure that MetaCard doesn't have a feature built in that does what the HC external did (externals are not portable across platforms and so should be avoided whenever possible).

8) Because it was designed to work on multitasking OSes, MetaCard's support for features only appropriate for single-user single-tasking systems like MacOS is limited. For example, all scripts that have **idle**, **mouseStillDown**, or **mouseWithin** handlers should be rewritten to eliminate these handlers. In their place, use the **send** command, the **grab** command, or the **mouseMove** message.

9) MetaCard does not support color add-ons for HyperCard and so commands that access them will have to be removed from scripts and colors added back in using MetaCard's integrated color support.

10) MetaCard supports PICT images on MacOS systems, but not cross platform. Furthermore, PICT (like other OS-specific image formats including BMP and XWD)

is a very inefficient storage format for images. PICT images used in HyperCard or SuperCard applications should be converted to GIF/PNG or JPEG format and then imported into MetaCard.

For more information on scripting language incompatibilities between HyperCard, SuperCard, and MetaCard, see the script language guide available in the contrib directory on the MetaCard FTP sites and via email from support@metacard.com.

open process and shell()

On UNIX systems, the `open process` command and the `shell()` function can be used to run any UNIX command that can be run from the users login shell. The primary distinction is that the `shell()` function blocks (it waits) until the command has finished executing, whereas the `open process` command returns so that the external process can continue running in the background while MetaCard resumes running in the foreground.

MetaCard can send signals to processes started with `open process` using the `kill` command, which operates like the UNIX `kill` command. These processes can also send signals back to the MetaCard engine using `kill`, but only the `SIGUSR1` and `SIGUSR2` signals can be caught. These two signals cause a `signal` message to be sent to the current `defaultStack`.

The `shell()` function can also be used to read or write text using a UNIX command:

```
put shell("cat filename") into field 1
```

See the details of how to use these commands in the [MetaTalk Reference](#).

Import/Export

Images, EPS objects, AudioClips and VideoClips can be imported using the `import` commands or the `Importer` dialog. Images can also be exported with the `export` command. See the [MetaTalk Reference](#) stack for details on these commands.

Refer to the cards on the `open`, `read`, and `write` commands in the [MetaTalk Reference](#) stack if you want to save text into a file (in order to print it, for example), or read it into a field. See also the `open process and shell()` card for other ways to import and export text.

External Commands and Functions

The external procedure call interface can be used to access libraries or other 'C' code from within MetaCard. Calls to external commands and functions look just like calls to MetaCard handlers, but actually call compiled code in a separate library or subprocesses.

On UNIX systems, communication between the MetaCard engine and the external process is achieved using X properties. To improve performance and make debugging external commands easier, an embedded version of the MetaCard engine is available at extra cost. Embedded MetaCard is a library that you can link directly to your C applications.

On Windows systems, externals are built as DLLs.

The names of the executable or library to be loaded when a stack is opened are stored in a stack's `externals` property. These executables or libraries must be stored some place on the current `PATH` environment variable.

On UNIX systems, once externals are started up, the processes continue to run until the stack window is destroyed (which is when MetaCard exits, or when the window is closed if the stack's `destroyWindow` property is set to true). The external command template files are distributed in a file called `external.tar.Z`. After the files are extracted (with the `tar xf` command, you'll have to uncompress the tar file first if it has a `.Z` extension), you must edit the Makefile to set the appropriate compiler, and include file and library paths. See the README file in that distribution for more instructions.

Communicating with MetaCard

There are many different ways you can communicate with MetaCard from an external application on UNIX systems. Here they are in order of preference:

1. If your application can accept command line arguments and has a short run time, use the `shell()` function.
2. If it has a longer run time, use `open process` for neither.
3. If you need to pass information to or from the application, consider `open process` for `read`, `write`, or `update`.
4. On UNIX systems, you can also write information to a file with the `open` and `write` commands, and send a signal to the process to tell it to read from the file with the `kill` command.
5. On UNIX systems, you can do the reverse by sending a `SIGUSR1` or `SIGUSR2` signal to MetaCard and catching the `signal` message.
6. You can start up any process or use any library from **External Commands and Functions**.
7. The external interface is bi-directional, so the external can send messages to your stack. For example on UNIX systems you can use the template `XT` program to communicate with MetaCard from a program that uses the Xt/Motif toolkit.

MetaTalk as a CGI or batch scripting language

On UNIX systems the MetaCard executable "mc" can be used as a MetaTalk scripting language to run scripts that do not open windows. The most common use of this is to develop CGI applications for use with HTTP servers.

The script should start with the line "#!mc" to tell UNIX what interpreter to use to execute this file. The initial message sent when a script is loaded is the "startup" message, so your script should contain a handler for that message. In a CGI application, the input typically comes from stdin, and so that's what you should read when the script first starts up.

When run directly from a script file the "put" command writes its output to stdout (since there is no Message Box). This is equivalent to the command "write <expression> to stdout".

The format of the input that a CGI application receives is very specialized, as is the requirements for the output that is to be sent back to the WWW browser. See the examples on the MetaCard WWW site <http://www.metacard.com/> for details.

Reference Manual

Index B

accept

accept [datagram] connections on port <p> with message <m>

The accept command accepts TCP connections or UDP datagrams from other systems or processes and creates a new socket that can be used to read or write to that other process or system. When one arrives, the specified message is sent with a first parameter that is the IP address of the other end of the socket. If it is a datagram, a second parameter containing the actual data is also included.

binaryDecode, hostAddress, hostAddressToName, hostName, open, openSockets, peerAddress, read, wait, waitDepth, write

add

add <expression> to <container>

The add command adds an expression into a container. Both the expression and the container must have numbers in them. You can verify this with the is operator. An example:

```
if field 1 is a number
then add (field 1) * 5 to field 1
```

This function can also be used to add a scalar to each element in an array or to add the values in the corresponding elements of two arrays.

combine, divide, median, matrixMultiply, multiply, Operators, put, round, subtract, transpose, union

answer

answer [<m>] <q> [with <reply> [or <reply> [or <reply>]]] [titled <s>]

answer file <p> [with <def>] [with filter <f>] [of type <t>] [titled <s>]

answer folder <p> [with <def>]

answer printer | color | effect

The answer command opens a modal dialog which asks a user a question specified by the expression <q> to which they must respond by clicking on buttons specified by <reply>. The string in the title bar of dialog can optionally be set by including a string <s> as "titled" parameter, and the icon can be changed by including an optional <m> parameter, which can be "information", "question", "error", or

"warning".

The local variable "it" is set to the name of the button the user clicked on. Note that as is true for the `ask` and `read` commands, the string "it" will be treated as an unquoted literal until the statement *after* the answer command.

The "file" and "folder" options open a modal File Selector dialog so that the user can choose a file or folder. The prompt string <p> is displayed in the dialog or title bar if the system supports this feature. The <def> is the string that will appear in the editable field for the file name, and can also contain a full path that will be the directory the dialog opens to, if the current system supports these features. The filter <f> causes only those files that match the wild-card expression to be shown. On MacOS, the filter is a list of one or more 4 character file type specifications, and the list can be specified with "of type" rather than "with filter". For example to display both MetaCard stacks and text files, specify "TEXTMSTK". On Windows systems, the filter option must include a name (shown to the user) and one or more wild card expressions, separated by returns or commas. If more than one file type must be included, separate them with a semicolon. For example, to allow selection of MetaCard stacks or images, use the string "MetaCard Stacks (*.mc),*.mc,Images (*.gif;*.jpg),*.gif;*.jpg,All Files,*.*". On UNIX systems, the wild card string must be a single "glob" wildcard expression.

The answer folder command allows the user to select a folder.

The file or folder the user selects with answer file or answer folder is put into the local variable "it". The delimiter used in the returned path name will always be the UNIX standard / character instead of the DOS standard \ character or the MacOS standard : character (any / characters in a file name will be changed to the : character on MacOS). If the user cancels out of the dialog, the result will contain "Cancel".

The answer printer command opens a modal printer selection dialog on MacOS and Windows systems. Using it is the only way to set certain printing properties (such as `printRotated`) on MacOS systems. See also the `open printing` command.

```
answer question "Is this useful?" with "Yes" or "No"
```

```
if it is "No" then exit to MetaCard
```

```
answer file "Please select a file as input:"\
```

```
with "sample.txt" with filter "*.txt"
```

```
answer file "save as..." with "/initial/dir/"
```

The answer color command opens the system color chooser dialog and returns the RGB value of the color chosen. The answer effect command opens the QuickTime effects chooser dialog and returns an encoded string that specifies the effect name and option parameters that can be passed directly to the visual command.

`ask`, `dialogData`, `directory`, `dontUseNS`, `filter`, `formatForPrinting`, `go`, `modal`, `modeless`, `open`, `palette`, `platform`, `print`, `printRotated`, `qtEffects`, `request`, `send`,

systemFileSelector, topLevel

ask

ask <question> [with <reply>] [titled <string>]

ask file <prompt> [with <default>] [with filter <filter>]

ask password [clear] <prompt> [with <reply>] [titled <string>]

The ask command opens a modal dialog which asks a user a question specified by the expression <question> to which they must reply by entering a text string into a field. The <reply> parameter is an optional suggestion that will appear in the editable field. The contents of the field are put into the local variable "it" if the user presses the "OK" button, otherwise the constant empty is returned.

If the word "password" is added, the dialog echoes with the "*" character instead of the characters typed by the user. If the word "clear" is added, the characters typed by the user are put into the local variable "it", otherwise an encrypted version is returned.

"ask file" opens a modal File Selector dialog that allows the user to choose a file to save to. The <prompt> is an expression that explains what the user is supposed to do. The <default> is an expression whose value is placed into the field at the bottom of the dialog as a suggested file name and specifies the initial directory if it contains a path. The optional <filter> is a string containing wild-cards that is used to reduce the number of file names displayed as described in the answer command. The delimiter used in the returned path name will always be the UNIX standard / character instead of the DOS standard \ character or the MacOS standard : character. If the user cancels out of the dialog, the result will contain "Cancel".

Since you can build your own dialogs in MetaCard, this command is not as important as it is in HyperCard. You are encouraged to clone the "Ask Dialog" stack to give it a more meaningful name and a context-sensitive help button. Open the cloned stack with the modal command.

```
ask "What is your name?" with "John Smith"
```

```
ask password clear "Enter your personal ID:"
```

```
ask file "Please name the text file"\
```

```
with "sample.txt" with filter "*.txt"
```

answer, dialogData, dontUseNS, filter, modal, modeless, open, palette,
systemFileSelector, tempName, topLevel

beep

beep [<expression>]

The beep command causes the system speaker to emit a tone. The <expression> parameter, which must evaluate to a number, determines the number of beeps. If

expression is omitted, one beep is emitted.

```
beep 2
```

```
if alarm then beep 10
```

```
beepDuration, beepLoudness, beepPitch, play, sound
```

```
breakPoint
```

```
breakPoint
```

The `breakPoint` command causes the debugger to stop script execution as if a break point had been set at the statement *after* the `breakPoint` command.

If the script debugger is not running, this command has no effect.

```
ask, answer, do
```

```
call
```

```
call <handler-name> of <object>
```

The `call` command calls a handler in another object's script. The message can be either one of the predefined **Messages**, or the name of any other handler in the script. Any parameters to be passed to the handler should be part of the `<handler-name>` string, separated by commas. The whole message should be in quotes, but note that expressions within the message string are evaluated (see the `send` command for details)

Note that when you call a message handler in the script of an object on another card or another stack, the current context is not changed. So, for example, if a handler in stack "stack A" calls a handler in the script of an object in stack "stack B", the expression "field 1" will refer to the first field in stack "stack A". This is the opposite of the behavior of the `send` command.

```
script in button "B1":
```

```
on mouseUp which
```

```
#if mouse button 1 is clicked, change the color of
```

```
# button "B2" to red, otherwise to blue
```

```
if which is 1 then
```

```
call "changeColor red" of button "B2"
```

```
else call "changeColor blue" of button "B2"
```

```
end mouseUp
```

```
script in button "B2":
```

```
on changeColor newColor
```

```
#change my background color to newColor
```

```
set the backColor of me to newColor
end changeColor
```

click, defaultStack, do, drag, go, kill, type, param, pass, send, signal, start, stacksInUse, topStack, value

cancel

```
cancel <timer-id>
```

The cancel command is used to cancel a message put on the pendingMessages queue with the send command.

pendingMessages, send, wait

choose

```
choose <tool-name> tool
```

The choose command sets the active tool for editable windows (those with a * in the title bar). Other windows always have the "browse" tool in effect. Note that the chosen tool's action only applies when clicking button 1, so it is possible to have message handlers respond to mouse clicks with buttons 2 and 3 when using the "pointer" tool, for example. The valid <tool-names> are:

pointer -- move and size controls

browse -- operate controls

button -- create buttons

field -- create fields

scrollbar -- create scrollbars

image -- create images

graphic -- create graphic objects

player -- create media players

To create different styles of graphic objects, set the style of the templateGraphic before choosing the graphic tool.

The following tools are for painting on image objects:

select -- select an image area

pencil -- draw fine lines

brush -- paint with a brush

eraser -- erase part of images

spray can -- like brush, but not smooth

bucket -- fill an area

dropper -- pick up a color

line -- draw straight lines
oval -- draw circles and ovals
curve -- draw curved lines and areas
reg poly -- draw regular polygons
rectangle -- draw rectangles
polygon -- draw polygons
round rect -- draw rounded rectangles

Note that the HyperCard lasso and text tools are not supported (yet). However, text can be put into fields, and the selection area used as an image mask, thereby creating a similar effect as the actions the lasso.

choose pointer tool
choose brush tool

brush, brushColor, brushPattern, centered, cursor, editMenus, filled, flip, grid, lineSize, lockCursor, lockLoc, modal, modeless, mouseColor, newTool, penColor, penPattern, polySides, rotate, select, selected, selectionMode, templateGraphic, tool, topLevel

click

click [button <number>] at <point> [with key [, key2 [, key3]]]

The click command sends a synthetic mouse click to the **defaultStack**. The button parameter <number> is an expression that evaluates to an integer. If it is omitted, button 1 is used. The <key> parameters are chosen from the set {commandKey, controlKey, optionKey, shiftKey}. The commandKey and controlKey options are the same on platforms other than MacOS.

The most common use for the click command is creating graphics automatically by simulating user input to the image editing system. Note that using the **send** command to send messages to objects is more efficient, since only one message type is sent (instead of separate mouseEnter, mouseDown, and mouseUp messages).

commandKey, choose, clickChunk, clickText, defaultStack, drag, find, focus, mouseDown, mouseLoc, mouseUp, select, send, type
clone

clone <object>

The clone command makes a copy of the specified object. If the object specified is a control, it will be placed on the card slightly offset from the original. If the object is a card, the new card will share the groups of the specified card and will have copies of all controls used on the source card. If the object is a stack, the stack will be an exact duplicate of the original, and will be opened for editing.

After an object is created, a message is sent that can be used to prepare the environment to operate on the new object. For example, when a button is created, a `newButton` message is sent. The local variable "it" will also contain the long id of the newly created object.

Dragging a control with the control key held down, will clone the dragged object (**choose** pointer tool first). Note that state information (hilited, armed, selected, etc.) is not preserved when an object is cloned.

```
clone this card
```

```
clone button 1
```

```
backgroundBehavior, copy, cut, create, delete, get, paste, place, remove, select, undo
```

```
close
```

```
close <stack>
```

```
close file <filename>
```

```
close process <filename>
```

```
close printing
```

```
close socket <s>
```

The `close <stackname>` command will close a stack that was opened with the `go`, `topLevel`, `modal` or `modeless` commands. The `close` command can also be used to close a file or process opened with the `open` command.

Note that processes open for mode "neither" don't have to be closed: they exit automatically. You may not be able to immediately start up another process of the same name after closing a process, since MetaCard waits until the previous process has exited before freeing the name for reuse. If you know the process will exit, you can wait for it. Otherwise, you may need to use the `kill` command.

```
open process "sh"
```

```
# do some reads and writes to the shell
```

```
close process "sh"
```

```
repeat with i = 1 to 100
```

```
if "sh" is not in the openProcesses
```

```
then exit repeat
```

```
wait 1
```

```
end repeat
```

```
# now you can start up another copy of "sh"
```

The command "close printing" can be used to stop batching printed cards and send the job to the printer.

```
close file "testfile"
close this stack
close stack "myStack"
```

The "close socket" command is used to close a socket.

clone, closeStack, create, delete, go, kill, modal, modeless, open, openProcesses, openSockets, option, palette, popup, print, pulldown, revert, serialControlString, topLevel

combine

```
combine <var> using <pd> [and <sd>]
```

This command combines the elements of an array variable <var> into a single string using the delimiter <pd>. If the array is string indexed instead of numerically indexed, you can also specify an optional second delimiter <sd> which will be used to concatenate the index and the value of each element.

add, extents, intersect(C), split, transpose, union

compact

```
compact stack <stackname>
```

The compact command removes empty space from within a stack that wasn't freed when cards are deleted or rearranged with cut/copy/paste. Note that the standard interface does a compact before saving, so this command is only useful when writing scripts that will have a large number of cards repeatedly created and deleted in one MetaCard session.

```
compact this stack
```

```
compact stack "myStack"
```

clone, create, delete, go, kill, save, open, topLevel

constant

```
constant <name = value> [,<name = value>, ...]
```

You can declare your own constant values with the constant command. These values are similar to initialized variables declared with the local command, but an error occurs if you try to change their values.

Constants, explicitVariables, global, local, localNames, put, variableNames

convert

```
convert <container> to <format> [and <format>]
```

The convert command converts the text in a field or variable from one date/time form to another. <format> is either "time" or "date", possibly modified by the words

"abbreviated", "short", or "long". <format> can also be "dateItems" which is a comma separated list of date elements in the order year, month, day, hour, minute, second, day of week.

Be sure to check that the container contains a valid date by using the "is" operator.

```
if field 1 is a date then
```

```
convert field 1 to seconds
```

```
add 30000 to field 1
```

```
convert field 1 to long date
```

```
end if
```

```
convert "7/1/93" to long date
```

```
convert the date && long time to dateItems
```

```
convert 926671399454 to short time
```

centuryCutoff, charToNum, date, find, send, sort, seconds, time, Operators, twelveHourTime, useSystemDate

```
copy
```

```
copy
```

```
copy <object> [to <dest>]
```

The copy command copies either the currently selected text, image selection, or the object specified by <object> to the clipboard, or to the group, card, or stack specified by <dest>:

```
copy button "B1"
```

```
copy field 1 to stack "backup"
```

```
copy the selObj #copy the selected object
```

After an object is created, a message is sent that can be used to prepare the environment to operate on the new object. For example, when a button is created, a newButton message is sent. The local variable "it" will also contain the long id of the newly created object.

copyResource, clipboard, clone, commandKeyDown, copyKey, create, cut, get, delete, paste, place, put, rawKeyDown, remove, select, selectedChunk, selectedObject, selection, undo

```
create
```

```
create [invisible] <object-type> [<name>] [in group <groupname>]
```

```
create stack <name> with <background>
```

```
create directory <dirname>
```

create alias <a> to file <f>

The create command creates an object of type <object-type>. If <object-type> is a control, it will be placed on the current card. If it is a card, it will be placed in the **defaultStack** and opened. Note that when a card is created, the groups on the current card will be placed onto the new card.

After an object is created, a message is sent that can be used to prepare the environment to operate on the new object. For example, when a button is created, a **newButton** message is sent. The local variable "it" will also contain the long id of the newly created object.

The optional [invisible] parameter creates the object with its **visible** property set to false. The optional <name> parameter can be used to give the new object a name. The optional "in group <groupname>" clause can be used to create a control that will be contained by that group.

When creating a stack, you can specify a background expression that will be copied into the new stack (note however that it's usually easier to use the **clone** command to just clone the whole stack):

```
create stack "test" with this background
```

Each object type has a template that is used for creating objects of that type. If you want to change the properties of an object before it is created, set the property in the template first (be sure to set it back, or to check the current settings before creating your next object).

```
set the style of the templateButton to "radio"
```

```
create button "B1"
```

```
create field "newField"
```

After controls are created, a message is sent that can be used to prepare the environment to operate on the new control. For example, when a button is created, a **newButton** message is sent that is handled in the Home stack, which chooses the pointer tool. The local variable "it" will also contain the long id of the newly created object.

Create can also be used to create a directory (or a folder on MacOS). Specifying a full path creates that directory. Otherwise, the file is created as a subdirectory of the current **directory**. The delimiter used in the path name should always be the UNIX standard / character instead of the DOS standard \ character or MacOS standard : character. If you need to create a / character in a file name on MacOS, use a : in the path where the / should appear in the file name.

The create alias command creates an alias (known as a "shortcut" on Win32 systems and a "symbolic link" on UNIX systems) named <a> to an existing file <f>.

To create a file, use the **open** command.

aliasReference, backgroundBehavior, clone, copy, cut, delete, hide, paste, place, remove, rename, reset, select, show, templateButton, templateCard, templateField, templateImage, templateScrollbar, templateStack, tempName, tool, umask, undo

cut

cut

cut <object>

The cut command copies either the currently selected text, image selection, or the object specified by <object> to the clipboard.

```
cut field "newField" of stack "myStack"
```

```
cut button "B1"
```

```
cut the selObj #cut the selected object
```

clipboard, clone, commandKeyDown, copy, create, delete, paste, place, put, rawKeyDown, remove, select, selectedChunk, selectedObject, selection, undo

define

define <prop> of <object>

The define command is provided for OMO compatibility and is non-functional.

set, undefine

delete

delete

delete <object>

delete local | global <variable>

delete file | directory <name>

With no paramters, the delete command deletes either the currently selected text or image selection. If an <object> is supplied, it deletes that object:

```
delete button "B1" of stack "myStack"
```

```
delete field "newField"
```

```
delete the selObj
```

Delete can also be used to delete local and global variables, which will free the memory used by those variables and in the case of array variables delete an index from the **keys** of that variable:

```
delete global myglobal
```

```
delete local myarray["some index"]
```

Delete can also be used to delete files and directories:


```
delete file "/tmp/tmpfile"
```

```
delete directory "/tmp/tmpdir"
```

clone, copy, create, customProperties, cut, decompress, deleteKey, deleteResource, open, paste, place, remove, rename, select, selectedChunk, selectedObject, selection, tempName, umask, undo

disable

```
disable <object>
```

The disable command disables a control so that it will no longer respond to mouse clicks or receive keyboard or mouse messages.

```
disable button "B1" of stack "myStack"
```

```
disable the selObj #disable selected object
```

autoHilite, cantSelect, default, disabled, disabledIcon, enable, hide, hilited, ink, traversalOn, unhilite

divide

```
divide <container> by <expression>
```

The divide command divides the value of a container by an expression. Both the expression and the container must have numbers in them. You can verify this with the is operator, for example:

```
if field 1 is a number
```

```
then divide (field 1) * 5 by field 1
```

```
divide totalScore by numberOfPlayers
```

This function can also be used to divide each element in an array by a scalar or to divide the values in the corresponding elements of two arrays.

add, multiply, numberFormat, Operators, put, round, subtract, trunc

do

```
do <expression> [as <language>]
```

The do command executes a list of one or more statements specified by <expression>. The statements can include any statement that could appear in a handler. For example, the MetaCard **Message Box** uses the do command to execute commands typed into it.

While the do command is very powerful and can be used to save dozens or even hundreds of ordinary script statements, keep in mind that that executing a statement using do takes up to 30 times as long as executing the same statement in a regular script. This is because each statement must be compiled before it can be executed, and unlike normal statements which are only compiled once, statements executed

with `do` must be recompiled each time they are executed.

The script executed by `do` has access to all of the variables declared in the current handler, but cannot create new variables. Use the `local` command to create any new variables that the script will need.

```
get "put" && quote & "hello" & quote
do it
do "put 3+5 into field 2"
put "showBorder" into pname
put "false" into pvalue
put "field 1" into ptarget
do "set the" && pname && "of" && ptarget\
&& "to" && pvalue
```

The optional `<language>` parameter, which is only available on MacOS, executes the statements using an OSA extension which must be installed on the current system.

alternateLanguages, call, constant, Constants, get, global, local, lockErrorDialogs, merge, param, request, script, scriptLimits, send, value

doMenu

doMenu `<expression>`

The `doMenu` command is provided for compatibility with HyperCard. All of the functionality accessible with the HyperCard `doMenu` command have equivalent commands in MetaCard, therefore use of `doMenu` is discouraged.

cut, copy, create, delete, do, hcStack, paste, quit

drag

drag [button `<n>`] from `<start>` to `<finish>` [with key [, key2 [, key3]]]

The `drag` command sends synthetic mouse down, move, and up events to the `defaultStack`. It should only be used to create shapes using the painting tools, and not to create controls or move them around. Instead, use the `move` command, or set the `points` property of a graphic object.

The `button` parameter `<n>` is an expression that evaluates to an integer that determines which mouse button is simulated. Mouse button 1 is used if the button number is not specified. The `<start>` and `<finish>` parameters are expressions that evaluate to x,y points. The key parameters are chosen from the set of {`commandKey`, `optionKey`, `shiftKey`}.

Note that the `create` and `clone` commands are more efficient for creating controls, and that the `send` command is a more efficient way to call event handlers. Moving a

control is most efficiently done by setting its `loc` property or by using the `move` command.

```
choose pointer tool
```

```
set the dragSpeed to 10
```

```
#drag button 1 to location (725,256)
```

```
drag from the loc of button 1 to 725,256
```

choose, click, clone, create, dragSpeed, grab, paintCompression, select, send, set, type

edit

```
edit [the] script of <object>
```

The `edit` command brings up the MetaCard script editor with the script of the object specified. It is used primarily as a shortcut typed into the **Message Box** to avoid having to go through intermediate dialog boxes.

```
edit the script of button 1
```

```
editScript, script, scriptTextFont
```

enable

```
enable <object>
```

The `enable` command enables a control so that it will respond to mouse and keyboard events.

```
if the disabled of button 1
```

```
then enable button 1
```

```
enable field "myField"
```

disable, disabled, enabled, mouseUp

export

```
export paint | png | jpeg to file <iname> [with mask <mname>]
```

The `export paint` command writes the currently selected image to a disk file in the Portable Bitmap Format. For more information on the PBM format, find the `PBMplus` or `NetPBM image-processing / file-format-conversion` packages on the Internet, read the Usenet news group `alt.graphics.pixutils`, or email a support request to your support contact. Most commercial image conversion packages also have PBM support. The `png` and `jpeg` options export the file in those other formats.

The `<iname>` and optional `<mname>` parameters are expressions that evaluate to file names.

```
export paint to file "myFile"
```

```
export png to file "myfile.png"
```

```
fileName, import, open, read, text
```

```
filter
```

```
filter <container> with <expression>
```

The filter command can be used to perform wild card matching of the lines in a container. The special characters supported are the same as those used by the Bourne shell. For example, "mc*" will match all lines where the first two characters are 'm' and 'c'. The string "[a-c]?" will match all lines with two characters where the first character is 'a', 'b' or 'c'. For example, to filter a directory entry to show only the 'C' source files:

```
put the files into flist
```

```
filter flist with "*.c"
```

```
put flist into field "C files"
```

The `matchText` function offers more advanced capabilities, but must be applied to each line separately.

```
answer, ask, caseSensitive, directories, files, fileType, format, matchChunk,  
matchText, open, put, replace, sort
```

```
find
```

```
find [ <type> ] <expression> [in field <field>]
```

The find command searches through each text field on each card in the `defaultStack` until it finds the search string specified in `expression`. The `<type>` specifies the characteristics of the matching process. If it is omitted, the find command matches the beginnings of words in `<expression>` with the beginnings of words in fields.

The other possible values of `<type>` are `chars` (which matches characters anywhere in words), `string` (which matches a string exactly, including spaces), `whole` (which matches the begin and end of a string of words), and `word` (which matches only complete words, but multiple words don't have to be sequential to be matched).

If you only want to search a given field of each card (which will make the search go faster), you can specify this field by name, id, or number. Note that only the field part of the chunk expression is used: including a card or stack name in the expression will make the find fail to work properly. Set the `defaultStack` to the name of the stack you want to search if it isn't the current stack. If nothing is found, the `result` function will return the phrase `not found`, otherwise it will return `empty`.

Note that there is a `Find` stack that makes it easier to use the find command by allowing you to select the stack to search in, and to choose options by pressing buttons.

In most cases, using the `offset` or `matchText` functions will locate text faster than

the find command, but these functions only work on a single container at a time.

```
find "Bill"
find string "o sea" #finds "to search"
find "visual" in field 1
on replace old, new
set the caseSensitive to true #caseSensitive find
repeat
find string old #exact string match
if the result is not empty
then exit replace #done
do "put new into" && the foundChunk
end repeat
end replace
```

caseSensitive, click, defaultStack, dontSearch, foundChunk, foundField, foundLine, foundLoc, foundText, lock, lockScreen, matchText, offset, put, replace, result, select, sort, visual

flip

flip [image <i>] horizontal | vertical

This command flips a image selection either horizontally or vertically. The optional <i> parameter will select and then flip that entire image.

choose, rotate, select

focus

focus [on] <object>

The focus command moves the keyboard focus to the specified object. For fields, it's similar to the command "select after field x" except that it doesn't change the position of the insertion cursor.

focusIn, focusedObject, rawKeyDown, select, showFocusBorder, type

get

get <expression>

The get command is a shorthand way of writing:

```
put <expression> into it
```

The variable `it` is a local variable use for this command and for the `clone`, `copy`, `create`, `read`, and `paste` commands. The get command is used most frequently with

properties, since it is the complement of the `set` command.

```
get "Hi"  
get field 1  
get nameList  
get the visible of button 1
```

The `get` command is also often used with URLs, though they can be used as containers in any expression:

```
get url "http://www.metacard.com/"
```

`do`, `httpHeaders`, `post`, `propertyNames`, `put`, `read`, `replace`, `set`, `Properties by Name`
`global`

```
global <variable-name> [, <variable-name> ]
```

The `global` command creates a global variable (if it does not already exist) and makes it available for use in the current handler. The global declaration must appear in *every* handler that uses it or the variable will actually be a local variable.

The `global` command can be used within a handler but can also be used outside all handlers (such as at the top of a script) in which case the variables created can be used in any handler within the script. Any number of variables, separated by commas, can be placed on the command line.

It is a good idea to prepend a stack name or some other prefix to the name when declaring a global to prevent accidental reuse of a variable name in another stack or as a local variable. For example, a variable in the MetaTalk Reference might be named `MRvar1`, or `gVar1`.

```
on initGlobal  
global gcount #create global variable  
put 10 into gcount #initialize to 10  
halfGlobal #changes value  
put gcount #puts 5  
end initGlobal  
on halfGlobal  
global gcount #gcount is 10  
divide gcount by 2 #gcount is now 5  
end halfGlobal
```

`constant`, `container`, `explicitVariables`, `globalNames`, `local`, `localNames`, `put`, `variableNames`

```
go
```

```
go [ to ] [ invisible ] <stack> [ as <mode> ] [in window <wid>]
```

```
go [ to ] <card> [ of stack <name> [ as <mode> ] ]
```

```
go back | forth | forward [ <count> ]
```

```
go start | finish | home
```

The `go` command opens the specified card of the specified stack. Visual effects specified with the `visual` commands are executed if the destination is a card within the current stack. `<card>` can be either the name or the number of a card, or an ordinal specification such as `next card`, `first card`, `recent card`, `any card` or `last card`. The word `marked` can be inserted to only go to those cards that have been marked (using the `mark` command, for example).

The optional modifier "invisible" can be used to open a stack invisible directly without having to set its visible property first.

The `<mode>` parameter is the mode of the stack that is opened and should be one of: `modal`, `modeless`, `palette`, or `topLevel`. If it is omitted, the stack is opened as `topLevel` unless the stack's `style` property has been set. To open a stack as an option, popup, or pulldown menu, set the `menuName` and `menuMode` properties of a button instead of using the `go` command.

The `<wid>` parameter can be either the name or the `windowId` of a stack that the new stack should open into instead of opening a new window.

The backward and forward options can be used to move through the cards on the recent list. For example, the command `go back 1` has the same effect as `go recent`.

```
visual effect dissolve
```

```
go to next card
```

```
go to stack "y" in window of stack "y"
```

```
go marked card 2
```

```
go last #same as "go to last card"
```

```
alwaysBuffer, clone, close, create, delete, dynamicPaths, lock, lockRecent, mark, modal, mode, modeless, navigationArrows, number, open, option, palette, pop, popup, post, pulldown, push, recentNames, save, send, style, topLevel, unlock, unmark, visual, windowBoundingRect
```

```
grab
```

```
grab <object>
```

This command causes the control that was clicked to track the mouse until the mouse button is let up. You should only call `grab` in a `MouseDown` handler, and should consider using the `mouseMove` message instead because it's much more flexible:

on mouseDown

grab me

end mouseDown

choose, click, drag, grab, loc, mouseDown, moveControl, moveSpeed, moveStopped, move, movingControls, play, select, send, set, type, visual

group

group [**<c>**]

The group command makes a set of selected controls into a group/background. If a list of controls **<c>** is supplied, those controls will be added to the new group.

group button 1 and field 2

copy, cut, delete, editBackground, go, paste, place, remove, start, ungroup

hide

hide **<object>** [with visual effect **<effect>**]

hide menubar

The hide command sets the **visible** property of the specified object to false. Hiding stacks does not **close** them, it just unmaps their windows. Resources held by the stacks (including **Externals** and colors) will continue to be held until the stack is closed. Hide can also be used on MacOS systems to hide the system menubar.

When showing an object, an optional visual effect can be supplied that will be executed as the object is shown.

hide field 3 with visual effect dissolve

alwaysBuffer, bufferHiddenImages, cantSelect, create, dontDither, invisible, menubar, move, show, showInvisibles, textStyle, visible, visual

hilite

hilite **<button>**

This command sets the **hilited** property of a button specified by **<button>** to true.

disable, enable, default, hilited, unhilite

import

import **<format>** from file **<iname>** [with mask **<mname>**]

import snapshot [from rect **<rect>** [of window **<windowID>**]

The import command reads an image from a disk file. The **<format>** parameter is one of **paint**, **audioClip**, **EPS**, or **videoClip**. The **paint** format will import images in GIF, JPEG, PNG, BMP, XWD, XBM, XPM, or the Portable Bitmap

(PBM) format.

The <iname> parameter is an expression that evaluates to a file name. The <mname> is an expression that evaluates to a file name where a mask (a one bit image) is stored. Be sure to put the file name in quotes. Note that when importing masks, both the image and the mask must be in PBM format.

The import snapshot command puts MetaCard in snapshot taking mode. The cursor changes, and an image of the window (or area) the user selects will be imported into the current **defaultStack**. If the optional <rect> and/or <window> parameters are supplied, the specified area of the window or screen are grabbed as an image.

Importing audioClips and videoClips is done primarily to simplify distribution of stacks, since only the stack needs to be sent out. It may also improve playback performance in some cases (importing clips during delays in the presentation is an important technique for CD-ROM based stacks, for example). Importing clips does increase memory usage, however, and so should not be done indiscriminately.

```
import paint from file "picture" with mask "pmask"  
import audioClip from file "music"  
import videoClip from file "movie"
```

boundingBox, export, fileName, open, play, prepare, read, screenGamma, windowId
insert

insert [the] script of <object> into front | back

The insert command is used to insert an object's script into the message passing hierarchy. Inserting an object's script into the front causes all messages to be passed through that script before they are sent to the original **target**. Inserting a script into the back causes messages to be passed to that script after all other objects in the hierarchy have had a chance to handle the message.

Inserting a script into the message passing hierarchy can be used to get around the 64K script size limitation, and can be useful for building certain types of editors. It does, however, make it more difficult to understand and predict how a stack will behave.

There is no limit to the number of scripts that can be inserted into the front or back during development with a licensed Home stack, but the **scriptLimits** apply otherwise.

backScripts, frontScripts, remove, script, scriptLimits, stacksInUse, start
intersect(C)

intersect <a1> with <a2>

This command computes the intersection of the elements in array <a1> with the elements in <a2>. The result is stored in <a1>. Note that the values for each element

come from array <a1> even if they differ from the values in the corresponding elements of array <a2>.

combine, extents, intersect, split, union

kill

kill [signal number] process <process-name>

The kill command is used on UNIX to send a signal to another process, for example one created with `open process`. The signal number can be either a number, or one of the signal names defined in `/usr/include/sys/signal.h`, leaving off the initial "SIG". For example, to send a SIGUSR1 to a process:

```
kill USR1 process "some process"
```

If the signal number is omitted, SIGTERM is sent.

You may not be able to start up another process of the same name immediately after killing a process, since MetaCard waits until the previous process has exited before freeing the name for reuse. If you know the process will exit, you can wait for it using the script given on the `close` card.

close, externals, launch, open, openProcesses, openProcessIds, platform, processId, send, signal, sysError

launch

launch [<document> with] <application>

The launch command starts up another application. The optional <document> parameter will cause the application to open that document. With no document specified, launch is a synonym for `open process <application>` for neither

hideConsoleWindows, kill, open, shell

load

load [url] <url> [with message <message>]

This command downloads the file specified by <url> into a local cache. The file can then be used in a url expression, or with the `go`, `play`, or `set` commands. Note that you don't have to load urls before using them. The load command just provides a way to download them in the background rather than having the script wait for the complete download before continuing.

The current status of a downloading URL can be queried with the `urlStatus` function, or the optional <message> parameter can be supplied which will cause that message to be sent when the download is complete.

Be sure to `unload` the URLs when you're finished with them to free up the system resources required to keep them in the cache. You must do this even if the download

fails for some reason.

cachedUrls, close, create, decompress, delete, go, httpProxy, kill, play, post, open, shell(), send, set, unload, urlStatus

local

```
local <variable-name> [= <val>] [, <variable-name> ]
```

The local command creates local variables (if they do not already exist) and makes them available for use in the current handler. Any number of variables, separated by commas, can be placed on the line. The local command can be used in a handler, in which case the new variable has scope only within that handler (that is, it can only be used within that handler). Locals can also be declared outside all handlers, such as at the top of the script, in which case the variables created can be used in any handler in the script.

```
local newValue
on thisLocal
  put 5 into newValue
  put 5 into count #count is local to "thisLocal"
  otherLocal #handler changes newValue
  #display "count=5 newValue=30" in Message Box
  put "count=" & count && "newValue=" & newValue
end thisLocal
on otherLocal
  put 20 into count #local count is 20
  put 30 into newValue #newValue is now 30
end otherLocal
```

You can initialize the value of a local variable when declaring it with the following syntax:

```
local l1 = 3, l2 = 4
```

constant, explicitVariables, global, localNames, put, variableNames

lock

lock colormap | error dialogs | menus | messages | moves | recent | screen

The lock command sets a global property to true. Locking the colormap prevents objects from changing the current CLUT, which would cause a complete screen redraw on MacOS systems. Locking menus prevents system menu structures from being updated. Locking messages prevents messages from being delivered to handlers. Locking moves stops all objects set into motion with the `move` command.

Locking recent prevents cards shown with the go and show commands from being added to the list of recentCards. Locking the screen prevents objects from redrawing themselves when they are moved or sized or changed in any other way. Locking the screen may drastically improve performance when doing geometry management in a resizeStack message handler. All locks are removed when all pending handlers have finished execution. Still, it is a good idea to unlock when you no longer need the property locked.

An example using lock error dialogs (but see the try control structure for a better way to handle errors):

```
on mouseUp
lock error dialogs
someHandler
#if there's an error, it will trigger the
#errorDialog handler instead of putting up the
#standard error dialog box
end mouseUp
on errorDialog
answer "Script error occurred. Please try again"
end errorDialog
```

An example using lock screen and messages:

```
on mouseUp
lock screen # prevent stack flashing
lock messages # prevent message delivery
go to card 3 of stack "userPassword"
#do something.....
unlock screen
unlock messages
end mouseUp

allowInterrupts, executionError, go, lockColormap, lockErrorDialogs, lockMenus,
lockMessages, lockMoves, lockRecent, lockScreen, recentCards, recentNames,
resizeStack, scriptError, try, unlock, visual

mark

mark all cards
mark <card>
mark cards where <expression>
```

mark cards by finding <expression> [in <field>]

The mark command sets the **mark(P)** property of a card based on the results of a text search or an expression evaluation. For example, to mark all cards in an address book that have a telephone number in Nevada do one of the following:

```
mark cards where field "Phone" contains "702"
```

```
mark cards by finding "702" in field "Phone"
```

You can also mark specific cards:

```
mark this card
```

```
mark card "index"
```

After the cards have been marked they can be viewed by putting the word "marked" into a **go** command, or printed:

```
go to next marked card
```

```
print marked cards
```

convert, do, find, go, mark(P), print, sort, unmark, value

modal

modal <expression>

The modal command opens up a stack name specified by an **expression** as a modal dialog box. Any input to any **topLevel** windows is disabled while the modal dialog is open. Modeless and palette windows and other application windows can still be used, however. Note that statements following the **modal** command will not be executed until the modal dialog is closed. If information must be returned to the handler that executed the **modal** command, set a custom property on the stack and check that value after the modal command:

```
on mouseUp
```

```
modal "some stack"
```

```
if the retval of stack "some stack" is "OK"
```

```
then put "works like it's supposed to"
```

```
end mouseUp
```

In script of "OK" button of stack "some stack":

```
on mouseUp
```

```
set the retval of this stack to "OK"
```

```
close this stack
```

```
end mouseUp
```

In script of "Cancel" button of stack "some stack":

```
on mouseUp
set the retval of this stack to "Cancel"
close this stack
end mouseUp
```

Dialogs opened with the **modal** command always use the browse tool, so it is not necessary to **choose** another tool. In order to provide the user with maximum flexibility, **modeless** dialogs should be used whenever possible.

answer, ask, cantModify, clone, close, create, delete, dialogData, go, modeless, open, option, palette, popup, pulldown, resizable, show, topLevel, wait
modeless

modeless <expression>

The modeless command opens up a stack name specified by an **expression** as a modeless dialog box. Modeless dialogs allow interaction with all other modeless and palette windows and other application windows. They are similar to **topLevel** windows and **palette** windows, but can be resized (if the **resizable** property is true), but cannot be edited or iconified separately.

Dialogs opened with the **modeless** command always use the browse tool, so it is not necessary to choose another tool. In order to provide the user with maximum flexibility, modeless dialogs and **palette** windows should be used instead of **modal** dialogs whenever possible.

```
modeless "myStack"
```

answer, ask, cantModify, clone, close, create, delete, go, modal, open, palette, resizable, show, topLevel, wait

move

```
move <obj> [from <s>] to <p> [in <t>] [without messages | waiting]
```

```
move <obj> rel[ative] <displacement>
```

The move command moves an object from its current location to the x,y coordinates in <p>, which can be a single point or a list of points. The **relative** form specifies that the object should move from the current position to the dx,dy offset specified by <displacement>. The optional starting x,y coordinate <s> causes the object to be moved to that location before starting the move.

The speed of the movement is determined either by specifying a duration <t> for the movement, or by setting the **moveSpeed** which is used when no time is supplied.

A control's position can also be set by setting its **loc** property. Movement can also be simulated using **hide** or **show** with a **visual effect scroll**.

```
move btn 1 from 1,1 to 10,10 in 2.5 seconds
```

```
move fld 1 to the points of graphic "path"  
set the moveSpeed to 50 #50 pixels per second  
move button 1 from 10,10 to 350,250  
move fld 1 relative -3,-5 in 90 millisecs
```

Note that messages are delivered normally while the move is taking place, allowing animating a button by changing its icon while it is moving (e.g., use the "send" command to send a message after a certain period of time has passed). This also allows for the possibility of stopping a move once it is started, perhaps by handling a mouseUp message in a "Stop" button. To stop a move, set the loc of the object being moved to the <finish> point, or use the **stop** command.

You can disable dispatching of messages during the move by specifying the `without messages` clause. If you decide to allow messages while an object is moving, it is important to prevent another "move" command from being executed while an object is moving. To prevent this, disable the button that started the move while the move is executing, or set a custom property or global variable indicating that a move is taking place and don't execute a move command if this property or variable is set.

By default, the move will complete before the next script statement is executed. If you want script execution to continue immediately, use the `without waiting` clause. If you need to start moving several objects at the same time, put them in a group together, or use the **lockMoves** property to suppress the start of the move until after each object has been specified in a separate move command.

choose, click, drag, grab, lockMoves, lowResolutionTimers, moveSpeed, moveStopped, movingControls, play, select, send, set, stop, syncRate, type, visual multiply

multiply <container> by <expression>

The multiply command multiplies the value of a container by an expression. Both the expression and the container must have numbers in them. You can verify this with the `is` operator. An example:

```
if field 1 is a number  
then multiply field 1 by field 1  
multiply count by 5
```

This function can also be used to multiply each element in an array by a scalar or to multiply the values in the corresponding elements of two arrays.

add, divide, matrixMultiply, numberFormat, Operators, put, subtract
open

```
open file <expression> [ for [binary | text] <mode> ]
```

```
open process <expression> for <mode>
```

```
open printing [ with dialog ]
```

```
open [datagram] socket [to] <host:port> [with message <m>]
```

The `open` command opens a file or a separate process specified by **expression**. The **<mode>** parameter can be either `read`, `write`, `append`, `neither`, or `update`. The `update` mode allows both reading and writing. The `append` mode will force writing to the end of the file and can only be used with files. The delimiter used in the path name on all platforms should be the UNIX standard `/` character instead of the DOS standard `\` or MacOS standard `:` character. Use the `serialControlString` property to set the properties of serial ports before you open them.

By default, files are opened in text mode, which means linefeeds, carriage returns, or the combination of the two will be translated to a single line feed (MetaCard's native text format line delimiter). Supplying the optional "binary" argument prevents this translation from occurring.

The `neither` mode can be used when you don't want to collect the output from a process. Processes opened for `write` or `neither` don't have to be closed because their entries are removed from the `openProcesses` automatically when they exit (see also the `launch` command). Processes opened for `read` and `update` must be closed before they are removed from MetaCard's process table. The `append` mode isn't available for processes.

Only one process of a given name can be started. If your stack needs to start more than one process running the same command, append spaces to the end of the command name before calling `open process`. If the file does not exist (for `read`), or if the file or directory permissions prevent access the `result` function will return "Can't open that file.", otherwise it will return `empty`. The `sysError` global property can be used to determine the cause of the failure.

On Win32 systems, the `hideConsoleWindows` property can be used to hide the console window that opens when a "console" application is run. You can still `read` the standard output or `write` to the standard input of these processes even though you can't see them.

Normally the "print card" command will print each card on a separate page. The command "open printing" can be used to start batching printed cards into a single job, which will be sent to the printer when the "close printing" command is executed. The optional "with dialog" parameter will cause the printer properties dialog to open on MacOS systems.

```
open file "dataFile" for read
```

```
open process "myProcess" for read
```

```
read from process "myProcess" for 3 lines
```



```
close process "myProcess"  
on printCards  
open printing  
print 2 cards  
close printing  
end printCards
```

The "open socket" command opens a TCP socket to the specified host and port. If the optional "datagram" parameter is supplied, a connectionless UDP socket is created. The actual connection is made asynchronously, and a **socketError** message is sent if the connection cannot be established. The socket can be written to immediately after this command completes and the written data will be buffered until the connection is made. The optional "with message" parameter can be supplied if the first operation on the socket must be a read, if the data to be written is not yet available, or if the first operation requires information only available for a connected socket (see the **hostAddress** function).

To open multiple sockets to the same host:port address, an optional connection identifier can be supplied by appending it with the | character as a delimiter (e.g., "www.metacard.com:80|1").

accept, answer, ask, close, compress, create, delete, do, export, fileName, formatForPrinting, hideConsoleWindows, hostNameToAddress, import, kill, launch, openProcesses, openSockets, print, read, rename, replace, revert, result, save, secureMode, seek, shell, sysError, tempName, there, umask, write

option

option <expression>

The option command opens a stack as an option menu. An option menu is like a popup menu, except that it is centered on the button instead of around the mouse pointer position. To function properly, the stack should be opened in a **mouseDown** handler in a button's script:

```
on mouseDown  
option "Help Menu"  
end mouseDown
```

Note that it is almost always better to use the button contents or set the button's **menuName** property to implement menus. This command is provided for cases when the menu to be presented can't be determined until **mouseDown**.

cantModify, close, go, menuButton, menuLines, menuMode, menuName, modal, modeless, open, palette, popup, pulldown, resizable, style, topLevel, wait
palette

palette <expression>

The palette command opens up a stack name specified by an expression as a modeless dialog box. Modeless dialogs allow interaction with all other modeless and palette windows and other application windows. They are similar to topLevel windows and modeless windows, but can't be resized, edited, or iconified separately (at least not with window managers that work correctly).

Dialogs opened with the palette command always use the browse tool, so it is not necessary to choose another tool. In order to provide the user with maximum flexibility, palettes and modeless dialogs should be used instead of modal dialogs whenever possible.

activatePalettes, answer, ask, cantModify, clone, close, create, decorations, delete, go, hidePalettes, modal, modeless, open, raisePalettes, resizable, show, style, topLevel, wait

paste

paste

The paste command puts a copy of the clipboard contents onto the current card (if it is an object), into the current field (if it is text), or into the current image (if it is an image selection).

After an object is created, a message is sent that can be used to prepare the environment to operate on the new object. For example, when a button is created, a newButton message is sent. The local variable "it" will also contain the long id of the newly created object.

clipboard, clone, commandKeyDown, copy, create, cut, delete, get, place, put, rawKeyDown, remove, select, selectedChunk, selectedObject, selection, undo

place

place <background> onto <card>

The place command places a background (group) onto a card. A card can have zero or more backgrounds, use the place and remove commands to attach these backgrounds to cards. Use the start and stop commands to edit backgrounds, but remember that editing a background changes its appearance on all the cards on which it appears.

place background "flower" onto card "garden"

place bg "clouds" onto this card

backgroundBehavior, clone, copy, create, cut, delete, paste, remove, replace, start, stop, undo

play

```
play [<type>] <name> [looping] [at <point>] [options <xanim options>]
```

```
play [pause | resume | step forward | step back] videoClip <name>
```

```
play stop [<type> <name>]
```

The play command plays a sound through the system's speaker, or starts a movie directly from a movie file or imported videoClip. This command is provided only for backward compatibility and all new development should use the **start** command with a player object.

Note that since some systems do not have built-in support for sound, this command is not guaranteed to work on all systems. In addition, it won't work on X terminals or PC/Mac X servers. <type> is either audioClip or videoClip, with audioClip being the default if this parameter is omitted. If it is videoClip, a location can be specified. This point is the center of the movie, and the center of the card will be used if this parameter is omitted.

Playback of a movie or audio clip can be stopped with `play stop`, or using the `stop` command. A movie can be paused with `play pause`. A paused movie can be single stepped forward a single frame with `play step forward`, stepped back with `play step back`, or resumed with `play resume`.

The <name> parameter is an expression that evaluates to a file name, or the name of a previously imported object. Be sure to put the name in quotes.

The optional `looping` parameter will cause the sound or movie to loop until it is stopped or another sound or movie is played.

The <xanim options> are command line parameters that are passed directly to XAnim when the play command is executed on UNIX/X11 systems. Run "xanim -h" from your UNIX shell to get a list of the supported options. Note that the "xanim" binary must be on the current PATH. If you're distributing a stack on CD, you should change the PATH to ensure that only the correct xanim binary is on the PATH before executing the play command. You can set the PATH using a shell script that also starts up MetaCard, or by putting a path into the global variable \$PATH.

Sounds must be in a format appropriate for the platform that MetaCard is running on (e.g. .116 on HP 9000/700 workstations, .aiff on SGI IRIS). The Sun/NeXT .au/.snd format (8-bit 8KHz mulaw encoding) is supported on all platforms, and is the recommended format for sounds that must play on more than one platform. Note that the sound data must be in the data fork on MacOS systems and not in the resource fork.

If a play command for an audio clip is executed before a previous play has completed, the current sound will stop before the new one is started (no mixing is supported). The message `playStopped` is sent to the current card of the stack that was the `defaultStack` when the clip finishes playing.

```
play "music.au" # plays a sound
```

```
play audioClip "music.au"
play stop # stops sound playback
play videoClip "clown.flc" at 200,200
play stop videoClip
play stop videoClip "clown.flc"
play pause videoClip "clown.flc"
play step forward videoClip "clown.flc"
play resume videoClip "clown.flc"
beep, beepPitch, currentTime, dontRefresh, dontUseQT, duration, frameCount,
frameRate, import, mciSendString, movie, platform, playDestination, playLoudness,
playRate, playStopped, prepare, scale, screenVendor, sound, templateVideoClip,
videoClipPlayer, visual
```

pop

```
pop card [ <preposition> <container> ]
```

The pop command opens a card that was previously stored with the push command. If a preposition (after, before, into) and a container is specified, the name of the pushed card is put into the container.

```
on shopping
lock messages
push card #store location of this card
go to stack "shoppingList"
get field "grocery"
put "We are out of:" & return & it
pop card #restore the card
unlock messages
end shopping
```

clone, close, create, delete, go, lock, modal, modeless, open, option, palette, popup, pulldown, push, topLevel, unlock, visual

popup

```
popup <expression>
```

The popup command opens a stack as an popup menu. A common UI technique in Windows is to open a popup menu when the user presses the right mouse button on an object:

```
on mouseUp which
```

```
if which is 3
then popup "My Prop Stack"
end mouseUp
```

Note that it is usually easier build menus using the button contents or to set a button's `menuName` property. This command is provided for cases when the menu to be presented can't be determined until `mouseDown` and for those cases where the popup must be opened from an object other than a button.

The *OSF/Motif Style Guide* specifically prohibits putting functions in a popup menu that are not available through some other function. In addition, expert users can use accelerators much more efficiently than popups, therefore, it is a good general rule to avoid the use of popups.

`accelKey`, `cantModify`, `close`, `go`, `menuButton`, `menuMode`, `menuName`, `modal`, `modeless`, `open`, `option`, `palette`, `pulldown`, `resizable`, `topLevel`, `toolTip`, `wait`

`post`

```
post <expression> to url <url>
```

The `post` command posts the text specified by `<expression>` to the url specified by `<url>` using the HTTP POST protocol. The value returned is placed in the local variable `it`.

`accept`, `cachedUrls`, `go`, `get`, `httpHeaders`, `httpProxy`, `load`, `open`, `read`, `send`, `unload`, `urlStatus`

`prepare`

```
prepare [type] <file> [looping] [at <point>]
```

The `prepare` command prepares a movie for playback, which is actually started with the `play` command. It is used to reduce the delays between the execution of the `play` command and the actual start of movie playback. The command line of the `prepare` command must be identical to the command line supplied to the `play` command.

```
prepare videoClip "clown.flc"
```

```
go to next card
```

```
play videoClip "clown.flc"
```

`frameRate`, `import`, `movie`, `open`, `platform`, `play`, `read`, `screenVendor`, `sound`, `visual`

`print`

```
print <card or stack expression> [into <rect>]
```

```
print break
```

This command prints a card to the currently selected printer. On UNIX systems, the `print` command creates a PostScript file and runs the `printCommand` to send the file

to an output device. On Windows and MacOS systems, the page is sent to the default printer or the printer the user selected via the "answer printer" command.

The print command can print a single card, multiple cards, or a whole stack. The optional <rect> parameter prints a single card into a particular rectangle (specified in printer points) on the page. If a destination rectangle is not supplied, the `printScale` is used to determine the size of the card on the printed page. Examples:

```
print this card
print card 1 into rect 0,0,468,648
print marked cards
print this stack
```

If you want to print multiple cards on the same page, use `open` printing, then use `print` on whichever cards you want to print, then `close` printing. You can cause a page break with the command `print break`.

```
on printCards
open printing
print 2 cards
print break
print 2 cards
close printing
end printCards
```

`answer`, `close`, `formatForPrinting`, `open`, `pageHeights`, `printColors`, `printCommand`, `printFontTable`, `printGutters`, `printMargins`, `printPaperSize`, `printRotated`, `printRowsFirst`, `printScale`, `printTextAlign`, `printTextFont`, `printTextHeight`, `printTextSize`, `printTextStyle`, `value`, `write`

`pulldown`

`pulldown <expression>`

The `pulldown` command opens a stack as an pulldown menu. To function properly, the stack should be opened in a `mouseDown` handler in a button's script:

```
on mouseDown
pulldown "Help Menu"
end mouseDown
```

Note that it is almost always better to use the button contents or set the button's `menuName` property to implement menus. This command is provided for cases when the menu to be presented can't be determined until `mouseDown`.

It is also possible to simulate a menu by using the `hide` and `show` commands with a

field (see the "Object Font" dialog for an example).

cantModify, close, go, menubar, menuButton, menuMode, menuName, menuPick, modal, modeless, open, option, palette, popup, raiseMenus, resizable, topLevel, wait
push

push <expression>

The push command puts the name of the card specified by an **expression** into a stack of cards. The cards can be popped back out in reverse order using the pop command.

```
on shopping
lock messages
push card #store location of this card
go to stack "shoppingList"
get field "grocery"
put "We are out of:" & return & it
pop card #restore the card
unlock messages
end shopping
```

clone, close, create, delete, go, lock, modal, modeless, open, option, palette, pop, popup, pulldown, topLevel, unlock, visual

put

put <expression>

put <expression> [into | before | after] container

The put command puts the value of an **expression** into a **container**. It is one of the most commonly used MetaTalk commands. With no preposition, the put command puts the value into the msg variable (which is displayed by the **Message Box**). This is useful primarily as a debugging aid, since you can use it to keep track of the execution of a script. The "into" preposition causes put to replace the contents of the container with the new value. The "before" and "after" prepositions prepend and append the new value to the existing contents of the container, respectively.

```
put empty into char 4 to 9 of it
put "Hello!" #put "Hello!" into message box
put return & line 1 of field "flower" after plant
put it into tmpValue
```

add, clone, constant, copy, create, cut, delete, get, paste, place, post, remove, replace, select, set, text, undo, write

quit

quit

The quit command exits the MetaCard engine. A shutdown message is sent, the current handler exits immediately, and no other scripts will run afterwards. If a confirmation is required before exiting, put the quit command in the script for the confirm button. Note that closing all stacks is another way to force an application to exit.

```
on myExitHandler
answer "Exit MetaCard?" with "Yes" or "No"
if it is "Yes" then quit
end myExitHandler
close, go, open, openStacks, shell, topLevel
read
```

```
read from file | process <name> [at <offset>] until <char> [ in <time> ]
```

```
read from file | process <name> [at <offset>] for <count> [ in <time> ]
```

```
read from socket <s> [for | until <cond>] [with message <m>]
```

The read command reads from a file or process specified by the expression <name>, putting the characters read into the local variable "it". Note that <name> is case sensitive on all platforms, so you must pass exactly the same string passed to the open command.

The constant `stdin` can also be used to read from MetaCard's standard input on UNIX systems (`stdin` is opened by default, no need to open it with the open command).

The **result** will be set to the string `eof` when an end-of-file was encountered during the read, or to an error message if some other error occurred. An empty result indicates a successful read.

If an <offset> is supplied, MetaCard will seek to that offset before starting the read. If the offset is negative, the seek will be relative to the end of the file. See also the **seek** command.

The <char> is a character that the read should stop at. Useful values include the special constant `eof` which will read until the end of file and `empty` which will read from a process until there is no more to read. Note that the <char> value can be a string, which will be matched exactly and completely, with the file pointer ending up at the end of the string in the file.

The "for" form of read reads <count> characters from the input or until `eof`, whichever comes first. This form is most useful when the data file to be read has

fixed length records. You can also specify a unit type (character, word, item, line, int1, uint1, int2, uint2, int4, uint4), in which case the specified number of that chunk type are read. If one of binary formats, such as a 4 byte unsigned integer (as specified by uint4), a comma separated list of numbers is returned corresponding to the binary values read. The <time> parameter can be specified to allow a process time to respond after an earlier write. If the read can't be completed in that time, the result is set to "time out". For example, after writing a command to a shell process, a delay is required to give the process time to complete the command:

```
on fileList
  put "#####" into handshake #recognize end of ls
  open process "sh"
  write "ls -l" & return & "echo" && handshake\
  & return to process "sh"
  read from process "sh" until handshake \
  in 5 seconds
end fileList
read form file "data" until eof
put it into datafilecontents
open process "ls -l" for read
read from process "ls -l" for 4 line
```

The "read from socket" command is used to read data from a socket. Sockets are always opened in binary mode and so any required data conversion must be done in scripts. If the optional "message" parameter is supplied, a message will be sent when the read completes, otherwise the read blocks until the data has been read. If no "for" or "until" condition is supplied, the message will be sent when any data arrives at the socket.

accept, ask, binaryDecode, close, Constants, export, get, import, isoToMac, numToChar, open, post, put, result, seek, serialControlString, set, shell, socketTimeoutInterval, there, write
record

record sound file <f> [as 4CC <c> | with <q> quality]

This command uses QuickTime to record sounds from the currently selected audio input into a file specified by <f>. The optional 4 character code <c> determines the format of the resulting file and can be any of the values returned by the recordFormats function. The optional recording quality <q> can be "good", "better", or "best" with "good" being the default quality.

Recording continues until the command "stop recording" is executed or the recording

property is set to false.

move, play, qtVersion, recording, recordLoudness, stop

redo

redo

The redo command is reserved for future expansion.

clone, copy, create, delete, paste, place, put, remove, undo

remove

remove <background> from <card>

remove [the] script of <object> from front | back

This command removes a background from a card.

```
remove background "flower" from card "plants"
```

```
remove bg "clouds" from this card
```

It can also be used to remove the script of an object from the frontScripts or backScripts.

clone, copy, create, cut, delete, insert, paste, place, replace, start, stop, undo

rename

rename [file | folder] <oldname> to <newname>

This command changes the name of file specified by <oldname> to the string specified by <newname>. Either argument can specify a full or relative path to a directory (folder) other than the current directory.

On UNIX systems, you can move a file or directory to another directory by specifying different directories for <oldname> and <newname>, but the parent directory for <newname> must already exist. MacOS and Win32 systems do not support moving files or directories to other directories.

clone, copy, create, delete, directories, files, fileType, name, open, replace, umask, write

replace

replace <pattern> with <replacement> in <container>

The replace command replaces all occurrences of the string <pattern> with the string <replacement> in the variable or field specified by <container>. This command executes faster than the replaceText function, and does not require that special characters be escaped in the pattern string.

binaryEncode, caseSensitive, filter, find, format, lineOffset, matchText, offset,

Operators, replaceText

reply

reply <string> [with keyword <keyword>]

reply error <string>

The reply command is a MacOS-only command that returns data to the application that sent an appleEvent to MetaCard. It is similar to the **return** keyword, but returns data to the calling program instead of to another MetaTalk script. The optional <keyword> parameter can be used to return a specific type of data to the sending application.

address, appleEvent, ask, environment, request, return, send, value

request

request <expression> from program <program>

request appleEvent | ae class | id | returnid | data [with keyword <keyword>]

The request command, which is only available on MacOS systems, sends an "eval" appleEvent to a program. The target program should evaluate the expression sent to it and return the result in the local variable "it". Any errors will be returned in the **result**.

The request appleEvent command can be used to extract data from an appleEvent message sent to MetaCard. See the documentation for the sending application to determine which are valid keywords for the appleEvents it sends.

request ae data with keyword "trans"

address, answer, appleEvent, get, reply, send, value

reset

reset <templateObject>

reset paint

Resetting a template object returns it to its startup settings. It is a good practice to reset any template object changed in your scripts after you've created an object from the template.

This command can also be used to reset the painting properties to their start up values:

brush = 8

spray = 31

eraser = 2

centered = False

filled = False

grid = False

gridSize = 8

lineSize = 1

pattern = 1

polySides = 4

roundEnds = False

slices = 16

penColor = Black

brushColor = White

create, Properties by Name, templateButton, templateCard, templateField, templateGraphic, templateGroup, templateImage, templateScrollbar, templateStack

revert

revert

This command reverts to the last saved version of a stack. Note that all changes to all substacks of the current stack are also discarded.

close, directory, export, go, import, mainStack, open, read, result, save, seek, shell, stackFiles, substacks, there, topLevel, write

rotate

rotate [image <i>] by <d>

This command flips a rotates an image by <d> degrees. If necessary the size of the image will be increased to make room for the rotated pixels unless the image's lockLoc property is set to true. The optional <i> parameter will select and then rotate that entire image.

Note that some image quality is lost each time you rotate, so it is not practical to rotate a single selection multiple times. Instead, make multiple copies of the image and rotate part or all of each copy a single time, then hide and show the copies as necessary.

choose, clone, flip, rotate, select

save

save <stack> [as <filename>]

The save command saves a stack specified by <stack> either into the file it was read from (its fileName property), or to a file specified by <filename>. Note that substacks are saved with their mainStack.

To emulate the automatic save behavior of Apple Corporation's HyperCard, put the following handler into a stack's script:

```
on closeStack
save this stack
end closeStack
```

You could also use the **send** command to periodically send a message that would cause the stack to be saved.

Note that the **save** command cannot be used in applications built with the Standalone Builder because most operating systems do not allow writing to the executable of a currently executing program.

close, **directory**, **export**, **fileName**, **go**, **import**, **mainStack**, **open**, **read**, **result**, **revert**, **seek**, **send**, **shell**, **stackFiles**, **stackFileType**, **substacks**, **there**, **topLevel**, **write**

```
seek to | relative <position> in file <filename>
```

The **seek** command seeks to a position in a file specified by <filename>. If the "to" mode is specified, the value of <position> specifies a byte offset from the start of the file. If the "relative" mode is specified, the seek is relative to the current file position in the file.

```
seek to 200 in file "data"
on readFile
open file "data" for read
seek relative 200 in file "data"
read from file "data" for 3 lines
put it
end readFile
```

close, **export**, **import**, **open**, **read**, **result**, **save**, **shell**, **there**, **write**

```
select <object> [ and <object> ...]
```

```
select <chunk>
```

The **select** command selects the object specified by <object> or the chunk specified by the **chunk** expression <chunk>. The **copy**, **cut** or **delete** commands can then be used on the selection. For example, you can select objects as follows:

```
select button "blueColor"
select field 1
```

You can also use `select` to select text within a field, or to position the cursor at a particular location in a field:

```
select word 3 of field "some field"
select line 3 of field "text"
select before char 3 of field "whatever"
select after text of field 1
```

Note that you should use the `hilitedLines` to select lines in a `listBehavior` field instead of `select`. Use the `focus` command to set the keyboard focus to a field without changing where the insertion cursor is. To unselect all text, use "select empty".

`activatePalettes`, `centered`, `click`, `clone`, `copy`, `create`, `cut`, `delete`, `find`, `flip`, `focus`, `grab`, `listBehavior`, `paste`, `place`, `put`, `remove`, `rotate`, `selectedText`, `selected`, `selectedObjectChanged`, `selectGroupedControls`, `selection`, `selectionChanged`, `selectionMode`, `traversalOn`, `undo`

`send`

```
send <message> to <object> [in <time>]
send <message> to program <program> [with <type> | without reply]
```

The `send` command sends a message to an object. The message can be either one of the predefined `Messages`, or the name of another handler in the script. Any parameters to be passed to the handler should be part of the `<message>` string, separated by commas. The whole message should be in quotes, but note that expressions within the message string are evaluated just they are with the `do` command. For example, the following handler sends "mouseUp" with a parameter of 2 to the button:

```
put 1 into x
send "mouseUp x + 1" to button "example"
```

One common use of the command is to send a `mouseUp` message to the default button in a dialog box. Putting this handler in the card script means that you don't have to write a third handler that would be called by the `mouseUp` and `returnKey` message handlers:

```
on returnKey
send "mouseUp" to button "OK"
end returnKey
```

Note that when you send a message to an object in another stack, the object context is changed. So for example "field 1" refers to the first field in the current card of the stack that contains the object whose handler is executing. If you want to refer to objects in the stack containing the handler that sent the message, use the `call` command, or use the `go` command or set the `defaultStack` to change the current

context.

The send to program command (which can only be used on MacOS systems) sends an AppleEvent to another application. The target program can be specified simply using its name if it is installed or running on the same system as the MetaCard application, or with an address of the form "zone:machine:program" if it is running on another system. Normally MetaCard waits for the target program to send a reply (which will be put into the result) before continuing. Adding "without reply" will cause execution to continue immediately. You can specify an AppleEvent class and id other than the default "miscdosc" by supplying it in the expression <type>. The word "application" can be used as a synonym for "program" in this form of the send command.

The send command should be used sparingly, as it can make scripts difficult to understand and to debug. Send also executes somewhat more slowly than calling a handler using the normal message passing hierarchy. In most cases, it is possible to avoid using send by putting message or function handlers in the card or stack script.

The exception to this recommendation is when using send to cause a handler to be called at some point in the future. In this case, send is much more efficient, and much more flexible, than using the wait command to wait for a certain period of time to pass, or handling the idle message. The id of the message added to the pendingMessages can be retrieved with the result function, and passed to the cancel function if needed:

```
send "goNextCard" to me in 3 seconds
put the result into timerid
cancel timerid
```

You might also consider setting a custom property of the target object and using getProp and setProp message handlers instead of send. While the number of parameters that can be passed to these functions is limited to one, custom property handlers execute more quickly and can be easier to maintain.

```
script in button "B1":
on mouseUp which
#if mouse button 1 is clicked,
#change the color of button "B2"
#to red, otherwise to blue
if which is 1 then
send "changeColor red" to button "B2"
else send "changeColor blue" to button "B2"
end mouseUp
script in button "B2":
```

```
on changeColor newColor
#change my background color to newColor
set the backColor of me to newColor
end changeColor
```

answer, appleEvent, call, cancel, choose, click, defaultStack, do, drag, dynamicPaths, go, idle, kill, launch, move, param, pass, pendingMessages, request, result, save, seconds, signal, start, stacksInUse, time, type, value, wait, waitDepth

set

```
set [the] <property> [of <object>] to <value>
```

The set command sets a property specified by <property> to the value of an expression <value>. If the property is an object property, <object> specifies the object by name or by number.

```
set the moveSpeed to 10
set the cursor to watch
set the visible of me to true
set the loc of button "fly" to 120,230
```

customKeys, customPropertySet, define, get, hide, keys, put, propertyNames, read, replace, show, Properties by Name, Operators

show

```
show <object> [with visual effect <effect>]
```

```
show menubar
```

```
show <count> cards
```

The show <object> command sets the **visible** property of the specified object to true, reversing the action of a **hide** command. Note that show does not open stacks. Use the **go**, **modal**, **modeless**, **palette**, or **topLevel** commands to open a stack.

When showing an object, an optional visual effect can be supplied that will be executed as the object is shown.

```
show field "extraInfo" with visual effect dissolve
show button 1
```

```
show 3 cards #flip 3 cards of the current stack
```

The show menubar command shows the system menubar on MacOS systems.

The show <count> cards command flips through a number of cards specified by the expression <count>.

alwaysBuffer, bufferHiddenImages, create, go, hide, invisible, lock, lockMessages,

menubar, modal, modeless, move, palette, resizable, showInvisibles, toolTip, topLevel, visible, visual

sort

```
sort [ [ [ marked ] cards of ] <stack> [ <dir> ] [ <key> ] [ by <exp> ]
```

```
sort [ <chunks> of ] <container> [ <dir> ] [ <key> ] [by <exp>]
```

The sort command sorts either the cards in a stack or the lines or items in a container. <dir> is either "ascending" or "descending" with "ascending" being the default. <key> is either "datetime", "text" or "numeric" with "text" being the default. <exp> can be any expression which differentiates the cards or chunks in a container. For example, to sort the cards of a stack by a particular field treating the contents of each field as a number, you would use something like:

```
sort cards numeric by field "amount"
```

To sort the lines of a field in descending order alphabetically:

```
sort field "file names" descending
```

To randomize the order of the lines of a field:

```
sort field "numbers" by random(1000)
```

When sorting fields or variables, <chunks> is either the word "lines" or "items", and the "by" expression can use the term "each", which refers to the line or item being sorted. To sort the lines of a field by the second word in each line (by last name if the field contained names, for example):

```
sort lines of field 1 by word 2 of each
```

The sort command uses a stable sort, which means that it does not change the order of items that have the same sort key. For example, if you wanted to sort a field containing a list of names, you could sort by first name (word 1) and then sort again by last name (word 2). The result will be that lines with the same last name will be in order of first name.

compress, convert, copy, cut, delete, find, paste, put, random, undo

split

```
split <var> by <pd> [and <sd>]
```

This command splits a string variable into separate numerically-indexed array elements using the delimiter <pd>. If the optional secondary delimiter <sd> is supplied, the array will instead be addressed by string keys where the text before <sd> will be the key and the text after will be the value.

add, combine, extents, intersect(C), transpose, union

start

start editing <background>

start using <stack>

start <player>

The start editing command puts the `defaultStack` into background editing mode. In this mode, objects created will become part of the group specified by <background>. When done editing, the `stop` command will return the stack to normal mode (where background controls cannot be selected with the pointer tool). Note that the stack must be opened as a `topLevel` stack and must not have its `cantModify` property set to true.

The start using command is used to add another stack's script to the message passing hierarchy. The "start using" command can also be executed using the `command library`. Unlike the `insert` command, stacks are always added to the end of message passing hierarchy. For example, you could put frequently used functions in a single stack's script and call these functions without having to use `send`.

The start <player> command starts a `paused` player control.

```
start using stack "Utilities"
```

```
put the stacksInUse
```

```
stop using stack "Utilities"
```

```
start editing background "newBg" \
```

```
of stack "newStack"
```

```
stop editing bg "newBg" of stack "newStack"
```

```
start player "myplayer"
```

```
cardNames, clone, copy, create, cut, delete, dontUseQT, editBackground,  
defaultStack, insert, libraryStack, mainStack, mainStacks, paste, paused, place, put,  
remove, scriptLimits, send, stacksInUse, stop, there, undo
```

```
stop
```

```
stop editing | moving | playing | using <object>
```

```
stop <player>
```

The stop editing command turns off background editing for the `defaultStack` or the background specified by <object>. The stop moving command stops an object that was set into motion with the `move` command. The stop playing command stops playing a video or audio clip started with the `play` command. The stop using command is used to remove a stack's script from the message passing hierarchy that was added with `start using`. This can also be called with the `command release library`.

```
start using stack "Utilities"
```

```
put the stacksInUse
stop using stack "Utilities"
start editing background "newBg" of stack "newStack"
stop editing bg "newBg" of stack "newStack"
stop playing videoClip "myclip.mov"
stop player "myplayer"
```

clone, copy, create, cut, delete, editBackground, move, moveStopped, paste, paused, place, play, playStopped, put, releaseStack, remove, stacksInUse, start, there, undo

subtract

```
subtract <expression> from <container>
```

The subtract command subtracts an expression from a container. Both the expression and the container must have numbers in them. You can verify this with the is operator, for example:

```
if field 1 is a number
then subtract 5 from field 1
subtract refund from total
subtract withdrawal from line 3 of field "Status"
```

This function can also be used to subtract a scalar from each element in an array or to subtract the values in the corresponding elements of two arrays.

add, divide, multiply, Operators, put

topLevel

```
topLevel <stack>
```

The topLevel command opens up a stack name specified by an expression as a topLevel window. If the stack's **cantModify** property is set to false, the stack will be editable, and the stack's controls can be selected, moved and sized with the pointer tool. If the stack's **style** property has been set, it overrides the style specified by the command used to open the stack.

cantModify, clone, close, create, decorations, defaultStack, delete, go, group, modal, modeless, open, option, palette, place, popup, pulldown, remove, resizable, show, style, wait, windowBoundingRect

type

```
type <expression> [ with key [, key2 [, key3 ] ] ]
```

The type command is used to send synthetic keyboard events to the defaultStack. It can also be used to type text into a field. The key parameters are chosen from the set

of {commandKey, optionKey, shiftKey}. Note, using the put command is a much more efficient way to put text into a field.

on enterText

choose browse tool

click at the loc of field "Shape"

type "Triangle and Circle"

end enterText

choose, click, drag, focus, move, put, select, shiftKey, selectedChunk, typingRate, rawKeyDown

undefine

undefine <prop> of <object>

The undefine command is provided for OMO compatibility and is non-functional.

define, set

undo

undo

The undo command undoes the last painting action, object deletion or object movement, *when performed by the user*. Note that deletions or changes made with scripts (including those done with the standard menus) can't be undone. Undoing again undoes the undo (puts the object back to its state before the first undo).

clone, copy, create, cut, delete, paste, place, put, redo, remove, select, selectedChunk, selectedObject, selection

ungroup

ungroup

The ungroup command breaks the selected group up into a set of selected controls.

copy, cut, delete, go, group, paste, place, remove, start

unhilite

unhilite <button>

This command sets the **hilited** property of a button specified by <button> to false.

disable, enable, hilite

union

union <a1> with <a2>

This command computes the union of the elements in array <a1> with the elements

in <a2>. The result is stored in <a1>. Note that the values for each element come from array <a1> even if they differ from the values in the corresponding elements of array <a2>.

combine, extents, intersect(C), intersect, split
unload

unload [url] <url>

This command deletes the file specified by <url> from the local cache, canceling the download if it is not complete.

cachedUrls, close, delete, go, kill, load, play, post, open, send, set, urlStatus
unlock

unlock colormap|error dialogs|menus|messages|moves|recent|screen

unlock screen [with visual effect <effect>]

The unlock command sets a global property set with the lock command back to false. When unlocking the screen, a visual effect can be specified with the same syntax used in the visual command. All locks are removed when all pending handlers have been finished execution. Still, it is a good idea to unlock when you no longer need the property locked.

An example using unlock error dialogs:

```
on mouseUp
lock error dialogs
someHandler
#if there's an error, it will trigger the
#errorDialog handler instead of putting up the
#standard error dialog box
end mouseUp
on errorDialog
answer "Script error occurred. Please try again"
end errorDialog
```

An example using unlock screen and messages:

```
on mouseUp
lock screen #keep user from seeing stack redraw
lock messages #prevent message from sending
go to card 3 of stack "userPassword"
```

#do something.....

unlock screen

unlock messages

end mouseUp

go, lock, lockColormap, lockErrorDialogs, lockMenus, lockMessages, lockMoves, lockRecent, lockScreen, recentCards, recentNames, resizeStack, visual

unmark

unmark all cards

unmark <card>

unmark cards where <expression>

unmark cards by finding <expression> [in <field>]

The unmark command sets the **mark** property of a card based on the results of a text search or an expression evaluation. For example, to unmark all cards in an address book that have a telephone number in Nevada do one of the following:

unmark cards where field "Phone" contains "702"

unmark cards by finding "702" in field "Phone"

You can also unmark specific cards:

unmark this card

unmark card 5

convert, do, find, go, mark, mark(P), sort, value

visual

visual [effect] <name> [<speed>] [to <image>] [with sound <sound>]

Visual effects provide an interesting way to **go** between cards. The <name> parameter is one of the following:

barn door [close | open]

checkerboard

dissolve

iris [close | open]

plain

push [down | left | right | up]

reveal [down | left | right | up]

scroll [down | left | right | up]

shrink [to bottom | to center | to top]

stretch [from bottom | from center | from top]

venetian blinds

wipe [down | left | right | up]

zoom [close | in | open | out]

You can also pass an encoded effect description as returned by the answer effect command, or a 4 character code as specified in the QuickTime developer documentation.

The <speed> parameter is [fast | slow[ly] | very fast | very slow[ly]]. The <image> parameter is one of [black | white | gray | inverse | card]. An optional audioClip <sound> can be specified which causes the clip to be played along with the effect. This provides better synchronization than is possible using the play and visual commands separately.

Any number of visual commands can be executed before a go command and they are executed in sequence. For example, this handler fades out to black screen, and fades in to the next card:

```
on fadeOutIn
visual effect dissolve slowly to black\
with sound "birds.au"
visual effect dissolve slowly to card
go to next card
end fadeOutIn
```

This handler goes to next card with visual effect:

```
on nextCard
visual effect scroll left fast
go to next card
end nextCard
```

answer, do, effectRate, go, hide, lock, move, play, pop, push, qtEffects, show, unlock
wait

wait [for] <count> [milliseconds | seconds | ticks]

wait until <condition> [with messages]

wait while <condition> [with messages]

wait for messages

The wait command freezes the execution of MetaCard for a specified period of time,

or until a condition has been met. If the unit is not specified, the **expression** `<count>` specifies how many ticks to wait, a tick being 1/60 of a second. `<condition>` is any expression that evaluates to a boolean value (true or false). For example, to wait until the user presses a mouse button:

```
wait until the mouse is down
```

In most cases using the **send** command to send a message to the object after a certain period of time is a better technique than using **wait**, since **send** allows you to wait for both a certain period of time and at the same time wait for other events such as mouse clicks and key presses. You can also include the optional "with messages" clause which will allow messages to be delivered to other objects while waiting.

idle, idleRate, lowResolutionTimers, milliseconds, mouse, repeat, seconds, send, ticks, waitDepth

write

```
write <expression> to file | process <name> [at end | eof | <offset>]
```

```
write <exp> to socket <s> [with message <m>]
```

The write command writes to an open file or process. Note that `<name>` is case sensitive on all platforms, so you must pass exactly the same string passed to the **open** command. If there the file or process is not opened or there is an error writing to it, the **result** will contain an error message. An empty result indicates a successful write.

On UNIX systems, the constant `stdout` can also be used to write to MetaCard's standard output. Like `stdin`, `stdout` is opened when MetaCard starts up, there is no need to open it with the **open** command. This feature can be used to output debugging messages to monitor the progress of your scripts:

```
write "i is" && i & return to stdout
```

Note that the whole text value of the expression is written to the file, use a **chunk** expression to specify only part of the expression.

If an `<offset>` is supplied, MetaCard will seek to that offset before starting the write. If the offset is negative, the seek will be relative to the end of the file. See also the **seek** command.

```
write "more data" to file "dataFile" at 300
```

If an existing file is opened for write mode only, the file will be truncated after the last character written to the file. If the file is opened for update (the default), it will not be truncated.

The "write to socket" command is used to write binary data to a socket. Any data conversion required must be performed by a script. If the optional message parameter is supplied, that message will be sent when the write completes. Otherwise, the write blocks until all the data has been written.

binaryEncode, close, empty, export, format, formattedText, get, import, macToIso, open, put, read, save, seek, set, shell, serialControlString, socketTimeoutInterval, there

Functions

abs

abs(<expression>)

The abs function returns the absolute value (a positive value) of an expression that yields a number.

add the abs of value1 to value2

add abs(value1) to value2

is, round, sqrt, trunc

acos

acos(<expression>)

Returns the arc cosine of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant pi.

put acos(1) * 180 / pi into value2

asin, atan, atan2, Constants, cos, numberFormat, put, sin, tan

aliasReference

aliasReference(<alias>)

Returns the path to the file that the alias (also known as a shortcut on Win32 systems and a symbolic link on UNIX systems) file <a> points to.

create, directory, files, longFilePath, shortFilePath, specialFolderPath, there

alternateLanguages

the alternateLanguages or alternateLanguages()

Returns a return-delimited list of the alternate scripting languages available on the current system. This function is only supported on MacOS systems.

do, platform, systemVersion, value

altKey

the altKey or altKey()

Returns "up" or "down" depending on the position of the keyboard "Alt" key. This function is the same as the optionKey function.

```
on checkKey
if altKey() is "down" then exit checkKey
else put "the Alt-key is up"
end checkKey
commandKey, keysDown, metaKey, optionKey, shiftKey, mouse
annuity
```

```
annuity(<rate>, <periods>)
```

The annuity function returns the present or future value of an ordinary annuity. The two parameters are expressions that specify the rate and the number of periods, respectively.

```
annuity(.020, 36)
annuity(currentRate, monthsOfLoan)
compound, exp
asin
```

```
asin(<expression>)
```

Returns the arc sine of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant pi.

```
put asin(1) into value1
acos, atan, atan2, Constants, cos, put, sin, tan
atan
```

```
atan(<expression>)
```

Returns the arc tangent of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant pi.

```
put atan(sqrt(4)) * 3
put the atan of tempResult
acos, asin, atan2, cos, put, sin, tan
```

```
atan2
```

```
atan2(<y>, <x>)
```

Returns the arc tangent of two numeric expressions y/x (in radians) using the signs of both arguments to determine the quadrant of the result. To convert to degrees, multiply by 180 and divide by the constant pi.

```
put atan2(20, 40)
```

acos, asin, atan, cos, Constants, put, sin, tan
average

average(<list>)

The average function averages a comma separated list of expressions or the elements in an array variable. For example:

```
put average(1, 2, 3, 4)
```

would put 2.5 into the **Message Box**.

The list could also be a single container. For example

```
put "1, 2, 3, 4" into somevar
```

```
put average(somevar)
```

would also put 2.5 into the **Message Box**.

add, divide, extents, max, median, min, Operators, round, statRound,
standardDeviation, sum

backScripts

the backScripts or backScripts()

The backScripts function returns a list of the objects that will receive messages that are not handled by the **target** or any object above it in the message passing hierarchy.

frontScripts, insert, remove, script, start

base64Decode

base64Decode(<expression>)

This function decodes a base64 (RFC 2045) string, a format frequently used to encode binary data in MIME mail messages and HTTP transfers.

accept, base64Encode, binaryDecode, charToNum, convert, decompress, httpProxy,
load, numToChar, open, post, put, toLower, toUpper, urlDecode

base64Encode

base64Encode(<expression>)

This function encodes <expression> using base64 (RFC 2045), a format frequently used to encode binary data in MIME mail messages and HTTP transfers.

base64Decode, binaryEncode, charToNum, compress, convert, httpProxy,
macToISO, md5Digest, numToChar, put, toLower, toUpper, urlEncode

baseConvert

baseConvert(<expression>, <source>, <dest>)

This function converts a numeric expression from the base <source> to the base <dest>. For the following will convert the base 10 number 8 to base 2 (binary):

```
put baseConvert(8, 10, 2) #puts "1000"
```

```
put baseConvert(175, 10, 16) #puts "AF"
```

This function would be most useful for building hex calculators and other debugging aids. Note that the other mathematical **Commands and Functions** all operate in base 10, so conversions are required at both ends of an expression.

add, base64Encode, binaryEncode, charToNum, compress, convert, convertOctals, format, md5Digest, numToChar, put, replace, toLower, toUpper, urlEncode

binaryDecode

binaryDecode(<format>, <exp>, v1, ...)

This function takes the data supplied in the parameter <exp> and decodes it into variables v1 ... as specified by the <format> parameter. It is useful when binary data structures must be read from a file or socket and converted to a form that can be more easily manipulated.

The format string can contain one or more of the following types. Each type can be followed by an integer, specifying the number of chunks to convert, or a * specifying that the remainder of the input string should be decoded as that type. Note that the destination variables must be declared before they can be used.

a - decode a single character

A - decode a single character, but strip spaces

b - decode to a string of 1s and 0s for the number of bits specified

B - same as "b", but work from high end of each byte

h - decode to a string of hexadecimal digits

H - same as "h", but work from high end of each byte

c - decode as signed single-byte integers

C - decode as unsigned single-byte integers

s - decode signed two-byte integers in host order

S - decode unsigned two-byte integers in host order

i - decode signed four-byte integers in host order

I - decode unsigned four-byte integers in host order

n - decode signed two-byte integers in network order

N - decode signed four-byte integers in network order

- f - decode single precision (four-byte) floating point
- d - decode double precision (eight-byte) floating point
- x - skip a byte in the input

The return value is the number of arguments converted.

This example extracts the characters in a string back out into variables var1 (which gets "AB") and var2 (which gets "CDE"):

```
local v1, v2
get binaryDecode("a2a*", "ABCDE", v1, v2)
```

This example extracts the integer values of the first two characters in a string into variables v1 (which gets 65) and v2 (which gets 66). It's similar to the `charToNum` function, but works on multiple characters at once. The last 3 characters are ignored:

```
local v1, v2
get binaryDecode("cc", "ABCDE", v1, v2)
```

This example extracts the hexadecimal values of the first two characters in a string into variables v1 (which gets 41, the hex value of 65) and v2 (which gets 42, the hex value of 65). It's similar to the `baseConvert` function, but works on multiple arguments at once:

```
local v1, v2
get binaryDecode("H2H2", "AB", v1, v2)
```

`accept`, `base64Decode`, `baseConvert`, `binaryEncode`, `md5Digest`, `isoToMac`, `open`, `uniDecode`, `urlDecode`, `read`

`binaryEncode`

`binaryEncode(<format>, arg1, ...)`

This function takes the data supplied in the parameter list and encodes it into a binary value as specified by the `<format>` parameter. It is useful when binary data structures must be written to a file or socket.

The format string can contain one or more of the following types. Each type can be followed by an integer, specifying the number of chunks in each parameter to convert, or a `*` specifying that all the data in the parameter should be encoded as that type:

- a - encode a single character, pad with null bytes
- A - encode a single character, pad with spaces
- b - encode a string of 1s and 0s into bits
- B - same as "b", but work from high end of each byte
- h - encode hex digits

H - same as "h", but work from high end of each byte
c - encode signed single-byte integers
C - encode unsigned single-byte integers
s - encode signed two-byte integers in host order
S - encode unsigned two-byte integers in host order
i - encode signed four-byte integers in host order
I - encode unsigned four-byte integers in host order
n - encode signed two-byte integers in network order
N - encode signed four-byte integers in network order
f - encode single precision (four-byte) floating point
d - encode double precision (eight-byte) floating point
x - skip a byte in the input

This example concatenates the first two characters in the first argument to all of the characters in the second argument. The value returned is "ABXYZ":

```
put binaryEncode( "a2a*", "ABC", "XYZ" )
```

This example encodes two integers into two characters, undoing the charToNum operations. It returns the string "AB":

```
put binaryEncode( "cc", \  
charToNum( "A" ), charToNum( "B" ) )
```

This example encodes two two-character hexadecimal strings into the character string "AB":

```
put binaryEncode( "H2H2", "41", "42" )
```

base64Encode, baseConvert, binaryDecode, open, uniEncode, urlEncode, write
buildNumber

buildNumber() or the buildNumber

If more than one build of a particular version is required to provide short-term fixes to specific problems, the buildNumber property can be used to distinguish these versions from one another.

platform, qtVersion, systemVersion, version

cachedUrls

the cachedUrls or cachedUrls()

This function returns a list of the currently cached URLs. It can be used to locate URLs that are no longer needed so that they can be unloaded.

repeat for each line l in the cachedUrls
unload url l
end repeat
load, play, post, send, unload, urlEncode
capsLockKey

the capsLockKey or capsLockKey()

Returns "up" or "down" depending on the position of the keyboard "Caps Lock" key. This function is the same as the altKey and metaKey functions.

altKey, click, extendKey, commandKey, drag, metaKey, optionKey, shiftKey, mouse, type
charToNum

charToNum(<character>)

This function returns the numeric value of <character> based on its position in the current character set.

charToNum(10) #returns 49 (value of 1)

put the charToNum of "A" #returns 65

baseConvert, binaryDecode, charset, compress, isoToMac, macToISO, numToChar, toLower, toUpper, uniEncode, urlEncode

clickChar

the clickChar or clickChar()

The clickChar returns the character in a field that was clicked on.

clickCharChunk, clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseChunk, selectedChunk, textStyle

clickCharChunk

the clickCharChunk or clickCharChunk()

The clickCharChunk function returns a chunk expression that describes the clickChar.

clickChar, clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseChunk, selectedChunk, textStyle

clickChunk

the clickChunk or clickChunk()

The clickChunk function returns a chunk expression that describes the clickText.

One of the most useful application of this function is to **select** the word or phrase the user clicked on, for example:

```
select the clickChunk
```

The string returned is of the form "char x to y of field z" where x, y, and z are integers.

click, clickCharChunk, clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseChunk, selectedChunk, textStyle
clickField

the clickField or clickField()

Returns the name (or id if no name) of the field the user last clicked on.

clickChunk, clickText, foundField, mouseControl, selectedField
clickH

the clickH or clickH()

Returns the x coordinate of the last place the user clicked. The position is retained until the next mouse click, unlike the click functions that apply only to fields such as clickText which "forget" their values during some editing operations. Be sure to check that the clickStack is the correct value, since the clickH is relative to the upper left corner of the clickStack.

click, clickChunk, clickField, clickLine, clickLoc, clickStack, clickText, clickV, foundLoc, mouse, mouseH, mouseLoc, mouseUp, mouseV, select
clickLine

the clickLine or clickLine()

The clickLine function returns a chunk expression that describes the line of text the user last clicked on. One of the most useful application of this function is to select the line:

```
select the clickLine
```

The returned string is of the form "line x of field y" where x and y are integers. For a more precise specification of the words clicked on, use the clickChunk function.

click, clickChunk, clickField, clickH, clickLoc, clickText, clickV, foundLine, hilitedLines, linkColor, listBehavior, mouseLine, selectedChunk, textStyle
clickLoc

the clickLoc or clickLoc()

Returns the x,y coordinates of the last place the user clicked. The position is retained until the next mouse click, unlike the click functions that apply only to fields such as

`clickText` which "forget" their values during some editing operations. Be sure to check that the `clickStack` is the correct value, since the `clickLoc` is relative to the upper left corner of the `clickStack`.

`click`, `clickChunk`, `clickField`, `clickH`, `clickLine`, `clickStack`, `clickText`, `clickV`, `foundLoc`, `mouse`, `mouseH`, `mouseLoc`, `mouseUp`, `mouseV`, `select`

`clickStack`

the `clickStack` or `clickStack()`

The `clickStack` returns the name of the stack containing the `clickLoc`.

`clickLoc`, `defaultStack`, `mouseStack`, `topStack`

`clickText`

the `clickText` or `clickText()`

The `clickText` function returns the actual text the user clicked on. This function can be used to implement a simple hypertext capability. By naming the cards to appropriate values and putting the following in a `mouseUp` handler, the user can navigate in the stack merely by clicking on the appropriate words:

```
if there is a card the clickText
then go to card the clickText
```

Note that the field must be locked for button 1 to be used for hypertext jumps. Button 3, however, can be used for hypertext jumps even in unlocked fields since it is not used to manipulate text or graphics (button 2 does a "quick-paste" of text selected in another application).

`click`, `clickChar`, `clickChunk`, `clickField`, `clickH`, `clickLine`, `clickLoc`, `clickV`, `foundText`, `listBehavior`, `matchText`, `mouseChunk`, `mouseUp`, `mouseText`, `select`, `selectedText`, `textStyle`

`clickV`

the `clickV` or `clickV()`

Returns the y coordinate of the last place the user clicked. The position is retained until the next mouse click, unlike the `click` functions that apply only to fields such as `clickText` which "forget" their values during some editing operations. Be sure to check that the `clickStack` is the correct value, since the `clickLoc` is relative to the upper left corner of the `clickStack`.

`click`, `clickChunk`, `clickField`, `clickH`, `clickLine`, `clickLoc`, `clickStack`, `clickText`, `mouse`, `mouseH`, `mouseLoc`, `mouseV`, `select`

`clipboard`

the `clipboard` or `clipboard()`

The clipboard returns the type of the information on the clipboard.

copy, cut, paste

colorNames

the colorNames or colorNames()

The colorNames function returns a list of the color names known to the MetaTalk interpreter. You can use this function to determine whether MetaCard will be able to convert a given string into its RGB equivalent.

commandNames, foregroundColor, functionNames, propertyNames, htmlText

commandKey

the commandKey or commandKey()

Returns "up" or "down" depending on the position of the keyboard Control key on UNIX and Windows systems, and the Command (cloverleaf) key on MacOS. This function is the same as the controlKey function on UNIX and Windows systems. The name of this function can be abbreviated as cmdKey.

altKey, capsLockKey, click, commandKeyDown, controlKey, drag, extendKey, keysDown, metaKey, optionKey, rawKeyDown, shiftKey, mouse, type

commandNames

the commandNames or commandNames()

The commandNames function returns a list of the commands known to the MetaTalk interpreter. You can use this function to determine whether or not a handler name you are planning to use is already defined:

put "somename" is in the commandNames

colorNames, customKeys, functionNames, keys, propertyNames

compound

compound(<rate>, <periods>)

The compound function returns the present or future value of a compound interest bearing account. The two parameters are expressions that specify the rate and the number of periods, respectively.

annuity, exp

compress

compress(<exp>)

The compress function returns the string passed in <exp> compressed in the gzip format (RFC 1952). The degree of compression depends on the source string, but

ranges from a growth of 1% for already compressed data to a reduction to 1% or less of original size. A reduction of approximately 3 to 1 is typical.

base64Encode, charToNum, convert, decompress, load, md5Digest, put, open, toLower, urlEncode, write

controlKey

the controlKey or controlKey()

Returns "up" or "down" depending on the position of the keyboard "Control" key. The message `commandKeyDown` is sent when another key is pressed while holding down the Control key. This function is the same as the `commandKey` function on UNIX and Windows systems, but returns the state of the control key on the Mac.

altKey, commandKey, controlKeyDown, metaKey, optionKey, rawKeyDown, shiftKey, mouse

copyResource

`copyResource(<source>, <dest>, <type>, <name>[, <newid>])`

This function, which is only available on MacOS systems, copies a resource from file `<source>` to file `<dest>`. The resource is specified by `<type>` and `<name>`, where `<name>` can be either the name of the resource or its id. The optional `<newid>` argument is the id that will be given to the resource in the `<dest>` file. If no `<newid>` is supplied, the id of the resource in the `<source>` file will be used.

copy, deleteResource, getResources, hideConsoleWindows, open, platform

cos

`cos(<expression>)`

Returns the cosine of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant `pi`.

acos, asin, atan, atan2, Constants, numberFormat, put, sin, tan

date

the `[long | abbreviated | short | internet] [system] date` or `date()`

The `date` function returns the date according to the system clock. The modifier is optional and must be one of `long`, `abbreviated`, `short`, or `internet`. If omitted, the `short` date is returned. The optional modifier `system` can be supplied which will return the date using the language and item order specified by the date control panel or locale environment variable.

The long date returns dates of the form: Monday, April 1, 1992

The abbreviated (or `abbrev` or `abbr`) form is: Mon, Apr 1, 1992

The short form is: 4/1/92

The internet form is: Mon, 1 Apr 1992 15:55:21 -0700

centuryCutoff, convert, Operators, monthNames, seconds, time, useSystemDate, weekDayNames

decompress

decompress(<exp>)

The decompress function returns the string passed in <exp> decompressed from the gzip format (RFC 1952).

base64Decode, charToNum, compress, convert, put, open, toUpper, urlDecode, write deleteResource

deleteResource(<path>, <type>, <name>)

This function, which is only available on MacOS systems, deletes a resource from the file specified by <path>. The resource is specified by <type> and <name>, where <name> can be either the name of the resource or its id.

copyResource, delete, deleteKey, getResources

directories

the [detailed] directories or directories()

Returns the subdirectories of the current directory as a return separated list. The detailed directories returns additional information on each directory as described for the detailed files.

directory, dontUseNS, drives, files, filter, open, rename, secureMode, there drives

the drives or drives()

Returns the drives/volumes available on the current system. This function can also be called as "the volumes".

directories, directory, dontUseNS, files, filter, open, platform, rename, secureMode, there

environment

the environment or environment()

This function returns information about the current run time environment. It can be "MetaCard Development", "MetaCard Helper Application", "MetaCard Player", or "MetaCard Plug-in".

address, emacsKeyBindings, licensed, lookAndFeel, kill, machine, platform, shell,

screenVendor, shellCommand, signal, sysError, systemVersion, version
errorObject

the errorObject or errorObject()

This function returns the name of the object who's script had an execution error in it. It is used by the Execution Error stack and the Script Editor stack to determine which object's script to edit.

edit, editScript, errorDialog, executionError, lock, me, select, selectedObject, target
exists

exists(<object>)

This function returns true if the specified object exists, and false otherwise. It is an alternate form of the **there** operator and can also be called *existence*.

cardNames, rect, layer, loc, Operators, owner
exp

exp(<expression>)

This function returns e to the power of the expression.

baseConvert, exp1, exp10, exp2, ln, ln1, log2, log10, Operators
exp1

exp1(<expression>)

This function returns e to the power of the expression - 1.

baseConvert, exp, exp10, exp2, ln, ln1, log2, log10, Operators
exp10

exp10(<expression>)

This function returns 10 to the power of the expression.

baseConvert, exp, exp1, exp2, ln, ln1, log2, log10, Operators
exp2

exp2(<expression>)

This function returns 2 to the power of the expression.

baseConvert, exp, exp1, exp10, ln, ln1, log2, log10
extents

extents(<a>)

This function returns the dimensions of a numerically indexed array as a comma-delimited pair of numbers. The first number is the lowest numbered element with data in it, the second is the highest. If the array has multiple dimensions, the extents of each dimension will be returned on a separate line.

average, keys, matrixMultiply, max, median, sum, transpose
files

the [detailed] files or files()

Returns the names of the files in the current directory as a return separated list. The detailed files returns a comma-delimited list of the attributes for each file in the following order:

urlEncoded file name

size in bytes

size in bytes of the resource fork (MacOS)

creation date in seconds (Win32 and MacOS)

modification date in seconds

access date in seconds (Win32 and UNIX)

backup date in seconds (MacOS)

owner user id (UNIX)

owner group id (UNIX)

file permissions in octal

creator code and file type (MacOS)

If an attribute is not supported on the current platform, that item will be empty.

aliasReference, create, directories, directory, dontUseNS, drives, files, fileType, filter, open, rename, secureMode, stackFileType, there, umask, urlEncode

flushEvents

flushEvents(<types>)

The flushEvents function removes events from MetaCard's input event queue, preventing them from being generating script messages. The <types> can be one or more of all, mouseDown, mouseUp, keyDown, keyUp, autoKey, disk, activate, highLevel, system. Some of these messages are not available on all platforms. This function is most useful when suppressing button double clicks in navigation controls, but must be used with caution because it can disrupt the normal operation of buttons and fields.

on mouseUp

go to next card

get flushEvents("mouseDown, mouseUp")

end mouseUp

keyDown, keysDown, keyUp, lockMessages, mouseDown, mouseUp,
pendingMessages

focusedObject

the focusedObject or focusedObject()

This function returns the long id of the object in the **defaultStack** that currently has the keyboard focus.

focus, focusIn, select, type

fontNames

the fontNames or fontNames()

This function returns a return separated list of the font names (faces) available on the current system.

fontSizes, fontStyles, textFont

fontSizes

fontSizes(<facename>)

This function returns a return separated list of the font sizes available in the type face <facename> on the current system. The <facename> should be one of the names returned from the **fontNames** function. If a 0 is one of the elements of the list, the font can be scaled to any size.

fontNames, fontStyles, textFont

fontStyles

fontStyles(<facename>, <size>)

This function returns a return separated list of the font styles available in the type face <facename> and size <size> on the current system. The <facename> should be one of the names returned from the **fontNames** function, and the size from the **fontSizes** function.

fontNames, fontSizes, textFont

format

format(<expression> [, value 1, value 2...])

Format takes a format string and zero or more values and returns a string combining them. The format string has the same syntax as that use for the C language printf

function:

```
write format("Answer was %5d\n", 5) to stdout
```

In this example %5d means to print out the first value as a decimal number padded with spaces to be 5 characters long. The "\n" means to append a newline character, which is equivalent to the MetaCard constant `return`.

The format function understands %s for strings, %d and %u for integers, %o and %x for octal and hexadecimal numbers, %f and %g for floating point numbers, %e and %E for scientific notation, and %c for characters. Optional field width and precision parameters are also supported:

```
put format("%12.4e", 100/3) into field 1
```

In this example, the 12 specifies that the returned string should be 12 characters long, padded with spaces as needed. The 4 specifies that there should be 4 digits after the decimal point in the returned string (the resulting string is " 3.3333e+01").

The format function translates the following escape sequences to their single character equivalents:

\a (alert or bell)

\b (backspace)

\f (formfeed)

\n (newline or line feed)

\r (carriage return)

\t (horizontal tab)

\v (vertical tab)

\\ (\ character itself)

\? (question mark)

\' (single quote)

\" (double quote)

\ooo (octal number)

\xhh (hexidecimal number)

baseConvert, binaryEncode, convertOctals, formattedWidth, htmlText, matchChunk, matchText, md5Digest, numberFormat, numToChar, Operators, put, replace, tabStops, write

foundChunk

the foundChunk or foundChunk()

The foundChunk function returns a chunk expression that describes the foundText.

The string is of the form "char x to y of field z" where x, y, and z are integers.

clickChunk, find, foundField, foundLine, foundLoc, foundText, matchChunk, matchText, mouseChunk, selectedChunk

foundField

the foundField or foundField()

Returns the name (or id if no name) of the field the last find command found the foundText in.

clickField, find, foundChunk, foundLine, foundLoc, foundText, mouseControl, selectedField

foundLine

the foundLine or foundLine()

The foundLine function returns a chunk expression that describes the foundText. The string is of the form "line x of field y" where x and y are integers.

clickLine, find, foundChunk, foundField, foundLoc, foundText, selectedLine

foundLoc

the foundLoc or foundLoc()

The foundLoc function returns the x, y coordinate of the topLeft corner of the box around the foundChunk.

clickLoc, find, foundChunk, foundField, foundLine, foundText, selectedLoc

foundText

the foundText or foundText()

The foundText function returns the text matched by the last find command. Since the find command may only match parts of words, sometimes it is useful to be able to determine the whole word boxed as the result of a find command.

clickText, find, foundChunk, foundField, foundLine, foundLoc, matchText, mouseText, selectedText

frontScripts

the frontScripts or frontScripts()

The frontScripts function returns a list of the objects that will receive messages before the target receives them.

backScripts, editScript, insert, libraryStack, remove, script, start

functionNames

the functionNames or functionNames()

The functionNames function returns a list of the functions known to the MetaTalk interpreter. You can use this function to determine whether or not a function handler name you are planning to use is already defined:

```
put "somename" is in the functionNames
```

colorNames, commandNames, customKeys, keys, propertyNames, variableNames
getResources

```
getResources(<path>[, <type>])
```

This function, which is only available on MacOS systems, returns a list of the resource in file <path>. The optional <type> argument causes the function to only return information on that type of resource. Information on each resources is returned in 4 items on a line: name, id, type, and size.

copyResource, deleteResource, open, platform

globalLoc

```
globalLoc(<point>)
```

This function translates a point in a local coordinate system (0, 0 is the upper-left corner of a stack) to the global coordinate system (0, 0 is the upper-left corner of the main monitor). It uses the **topLeft** of the **defaultStack** as the reference point.

localLoc, mouse, mouseChunk, mouseClick, mouseControl, mouseH, mouseLoc, mouseMove, mouseStack, mouseText, mouseV, mouseWithin, screenRect, screenMouseLoc, selectedLoc, topLeft

globalNames

the globalNames or globalNames()

Returns a comma separated list of the global variables. This is primarily useful for debugging a script that may be setting an unknown global variable.

explicitVariables, global, local, localNames, variableNames

hostAddress

```
hostAddress(<sock>)
```

This function returns the IP address of the local host being used by a socket. Note that the socket <sock> must be connected, which means that an **open**, **read**, or **write** command on that socket must have been completed.

accept, hostName, hostNameToAddress, open, openSockets, peerAddress

hostAddressToName

hostAddressToName(<addr>)

This function does a DNS lookup on the IP address <addr> and returns the corresponding host name.

accept, hostName, hostNameToAddress, open, openSockets

hostName

the hostName or hostName()

This function returns the domain name of the current host.

accept, hostAddressToName, hostNameToAddress, open, openSockets

hostNameToAddress

hostNameToAddress(<addr>)

This function does a DNS lookup on the name specified in <addr> and returns the corresponding IP address. If more than 1 IP address is available for that name, they are returned on separate lines.

accept, hostAddress, hostAddressToName, hostName, open, openSockets

interrupt

the interrupt or interrupt()

This function returns true if the user pressed the interrupt key (control-. or control-break) while the `allowInterrupts` property was set to false.

allowInterrupts, cantModify, lock

intersect

intersect(<object 1>, <object 2>)

This function returns true if the `rect` properties of the two objects intersect and false otherwise.

intersect(C), rect, layer, loc, Operators, owner, selectionMode, within

isNumber

isNumber(<chunk>)

This function returns true if <chunk> evaluates to a number and false otherwise. It is provided for backward compatibility with SuperCard, and the "is a number" validation operator should be used for all new development.

Operators

isoToMac

isoToMac(<string>)

This function translates all the characters in <string> from the ISO 8859-1 character set to the MacOS character set.

baseConvert, binaryDecode, charset, charToNum, compress, macToISO, numToChar, platform, toLower, toUpper, uniEncode, urlEncode

itemOffset

itemOffset(<part>, <whole> [, <skip>])

This function returns the number of the item where string <part> is found within string <whole>. If the part string does not appear within the whole, zero is returned. If the optional third parameter <skip> is included, it specifies a number of items to skip before beginning the search. If it is omitted, it is set to 0. Remember to add the skip value to the value returned from offset to find the true item offset within the whole string.

caseSensitive, itemDelimiter, lineOffset, offset, Operators, matchChunk, matchText, put, switch, wholeMatches, wordOffset

keys

keys(<variable>)

The keys function returns a list of the elements that have been set in a variable used as an associative array. See the container card in the Concepts & Techniques stack for more information on variables.

commandNames, customKeys, customPropertySets, delete, extents, functionNames, properties, propertyName, set, variableNames

keysDown

the keysDown or keysDown()

This function returns a comma-delimited list of keys on the keyboard that are currently pressed down. Each item is the KeySym of the key, the same value that is passed as a parameter with the rawKeyDown and rawKeyUp messages.

altKey, commandKey, keyDown, flushEvents, keyUp, lockMessages, mouse, mouseDown, mouseUp, pendingMessages

length

the length or length()

The length of a string is the number of characters in the string. The following two statements are equivalent:

```
put the length of "abcde"
```

put the number of characters in "abcde"

The "number" method is more general since it can be used with any chunk units.

Operators, number, offset

licensed

the licensed or licensed()

This function returns true if a licensed Home stack has been loaded and false if otherwise. If no Home stack or an unlicensed Home stack has been loaded, the setting and executing scripts is subject to the length limits specified in the scriptLimits property.

environment, lookAndFeel, platform, save, scriptLimits, script

lineOffset

lineOffset(<part>, <whole> [, <skip>])

This function returns the number of the line where string <part> is found within string <whole>. If the part string does not appear within the whole, zero is returned. If the optional third parameter <skip> is included, it specifies a number of lines to skip before beginning the search. If it is omitted, it is set to 0. Remember to add the skip value to the value returned from offset to find the true line offset within the whole string.

caseSensitive, itemOffset, offset, Operators, matchChunk, matchText, put, switch, wholeMatches, wordOffset

ln

ln(<expression>)

This function returns the natural (base e) logarithm of a number specified in <expression>.

baseConvert, exp, exp1, exp10, exp2, ln1, log2, log10, Operators

ln1

ln1(<expression>)

This function returns $\ln(\text{expression} + 1)$

baseConvert, exp, exp1, exp10, exp2, ln, log2, log10, Operators

localLoc

localLoc(<point>)

This function translates a point in the global coordinate system (0, 0 is the upper-left corner of the main monitor) to a local coordinate system (0, 0 is the upper-left corner

of a stack). It uses the `topLeft` of the `defaultStack` as the reference point.

`globalLoc`, `mouse`, `mouseChunk`, `mouseClick`, `mouseControl`, `mouseH`, `mouseLoc`, `mouseMove`, `mouseStack`, `mouseText`, `mouseV`, `mouseWithin`, `screenMouseLoc`, `selectedLoc`, `topLeft`

`localNames`

the `localNames` or `localNames()`

Returns a comma separated list of the local variables. This is primarily useful for debugging a script that may be setting an unknown local variable.

`constant`, `explicitVariables`, `functionNames`, `global`, `globalNames`, `local`, `variableNames`

`log10`

`log10(<expression>)`

This function returns the log base 10 of a number specified in `<expression>`.

`baseConvert`, `exp`, `exp1`, `exp10`, `exp2`, `ln`, `ln1`, `log2`, `Operators`

`log2`

`log2(<expression>)`

This function returns the log base 2 of a number specified in `<expression>`.

`baseConvert`, `exp`, `exp1`, `exp10`, `exp2`, `ln`, `ln1`, `log10`, `Operators`

`longFilePath`

`longFilePath(<f>)`

This function returns the long file path corresponding to the short file path `<f>` on Win32 systems. A short file path is in the DOS-standard 8.3 format, and is most likely to be encountered as the `fileName` property of a stack that was passed on the command line. This function returns `<f>` on the other platforms, or if `<f>` already is a long file path.

`alternateLanguages`, `charset`, `dontUseNS`, `dontUseQT`, `drives`, `environment`, `charset`, `files`, `fileType`, `hideConsoleWindows`, `longFilePath`, `machine`, `macToISO`, `platform`, `queryRegistry`, `shell`, `shellCommand`, `sysError`, `systemFileSelector`, `systemVersion`

`machine`

the `machine` or `machine()`

This function returns the CPU type of the machine MetaCard is running on.

`environment`, `platform`, `systemVersion`, `version`

`macToISO`

macToISO(<string>)

This function translates all the characters in <string> from the MacOS character set to the ISO 8859-1 character set.

baseConvert, charset, charToNum, compress, isoToMac, numToChar, platform, toLower, toUpper, uniEncode, urlEncode, write

mainStacks

the mainStacks or mainStacks()

This function returns a list of the currently loaded main stacks. A stack file on disk has one main stack, which is the stack opened when the stack is loaded, and zero or more substacks. Substacks are generally used to implement dialog boxes, floating palettes, and menus. See the `mainStack` and `substacks` property descriptions for more details.

backdrop, defaultStack, mainStack, mode, openStacks, start, substacks, topStack

matchChunk

matchChunk(<source>, <regex>[, <output 1>, <output 2> ...])

The `matchChunk` function returns true or false depending on whether expression <source> matches a regular expression pattern <regex>. Optional output parameters, which must be variables and included in pairs, can be used to return the start and end character positions of matched substrings. For example, here is the example given for the `matchText` function, but adapted to return chunks instead of matched text:

```
local nameStart, nameEnd
local addressStart, addressEnd
put "From: John Doe <jdoe@somewhere.org>" \
into source
if matchChunk(source, \
"^From: (.*) <(.+@.+)>", \
nameStart, nameEnd, \
addressStart, addressEnd)
then put "Name starts at char" && nameStart
```

Be sure to declare the destination variables before the `matchChunk` function using the `local` command as they won't be created automatically as they are with commands like `put`.

filter, format, lineOffset, matchText, offset, Operators, wholeMatches

matchText

```
matchText(<source>, <regex>[, <output 1>, <output 2> ...])
```

The `matchText` function returns true or false depending on whether expression `<source>` matches a regular expression pattern `<regex>`. Optional output parameters, which must be variables, can be used to return matched substrings which are enclosed in parentheses in the pattern.

```
local userName, userAddress
put "From: John Doe <jdoe@somewhere.org>" \
into source
if matchText(source, \
"^From: (.*) <(.+@.+)>", \
userName, userAddress)
then put "found user name" && userName
```

In this example, the `matchText` function returns true. The first group `(.*)` matches the substring "John Doe" and the second match matches the substring "jdoe@somewhere.org". These matched substrings are put into the variables "username" and "useraddress" respectively. Be sure to declare the variables before the `matchText` function using the `local` command as they won't be created automatically as they are with commands like `put`.

The following special characters can be used in the regex pattern:

- . matches any character
- ^ forces match to be at beginning of string
- \$ forces match to be at end of string
- [chars] matches any of the characters in the set of chars. The characters can be either characters allowed to match, or if ^ is the first character in chars, not allowed to match. You can specify a range of characters by putting a - between them. For example [a-z] matches any lower case alphabetic character.
- (exp) matches the expression, and puts result in a variable
- * matches zero or more of the preceding special character
- + matches one or more of the preceding special character
- ? matches zero or more of the same characters matched by the previous special character
- regex 1 | regex 2 matches either regular expression

You can force a literal match of any of these special characters by preceding them with a `\` character (use `\\` to match the backslash character). So, for example, the regular expression "test\$" will match the word "test" only at the end of a line, where "test\$" will match the string "test\$" anywhere in the input.

caseSensitive, filter, find, format, lineOffset, matchChunk, offset, Operators, replace, replaceText

matrixMultiply

matrixMultiply(<a1>, <a2>)

This function performs a matrix multiplication of two numerically indexed arrays <a1> and <a2> and returns a new array. The number of rows in <a1> must match the number of columns in <a2>.

add, divide, extents, max, median, min, multiply, Operators, round, statRound, standardDeviation, sum, transpose

max

max(<list>)

The max function finds the biggest number in a comma-separated list of expressions or the elements in an array variable. If, for example, field "Items" had "1, 4, 7, 3" in it:

```
put max(field "Items")
```

would put 7 into the Message Box.

add, average, median, min, Operators, round, standardDeviation

mciSendString

mciSendString(<MCI command>)

This function, only available on Windows 95 and Windows NT, sends a string to the Microsoft Media Control Interface system. It can be used to control devices such as video digitizers, laser disks, and other multimedia input and output devices. Documentation on the device name and supported commands should be available from the device manufacturer.

The function return value will be the string returned from the MCI system. The result function will contain details of any error, or will be empty if there was no error.

dontRefresh, play, result, shell, templateVideoClip

md5Digest

md5Digest(<exp>)

This function computes a 128-bit (16 character) digest of the data in <exp>. It can be used to determine if a large value has changed without having to retain a backup copy of that value. Note that the returned value is a binary value, and will need to be converted to some other form if it must be viewed as text. A common technique is to convert it to a hexadecimal string with the binaryDecode function:

local tHexDigest

get binaryDecode("H*", md5digest(myvar), tHexDigest)

base64Encode, baseConvert, binaryDecode, compress, format, numToChar, urlEncode

menuButton

the menuButton or menuButton()

This function can be used to determine which button has been used to open a menu panel. It is most useful when implementing option menus where the menu panel may need to display different contents depending on which button opened it.

The most common use of this function in previous versions of MetaCard, was setting the label property of an option menu button, it is no longer relevant however since the menu panel will do this automatically in newer releases.

errorObject, selectedObject, me, mainStack, menubar, menuMouseButton, menuName, target

merge

merge(<expression>)

The merge function finds delimited expressions in a string and evaluates them, replacing the expression with this value. It supports both expressions, which are delimited by [[and]], and commands which are delimited by <? and ?>. Note that you must explicitly return a value with a return keyword for the latter type:

```
put merge("3 + 5 is [[3 + 5]] and 2 + 2 is <?return 2 + 2?>")
```

alternateLanguages, call, do, Functions, me, Operators, request, send, value

me

me

This function can be used to get properties of the object whose script is currently running. Note that for this function, neither the word "the" nor parentheses are required.

A common mistake made with this function and the **target** function is to forget that the value returned includes the type of the object:

```
put me -- returns the contents of field
```

```
put the name of me -- returns <field "name">
```

```
put the short name of me -- returns <"name">
```

editScript, errorObject, selectedObject, target, value

median

median(<list>)

This function compute the median of a comma separated list of expressions or the elements in an array variable. For example:

```
put average(1,2,3,4)
```

would put 2.5 into the **Message Box**.

The list could also be a single container. For example

```
put "1, 2, 3, 4" into somevar
```

```
put average(somevar)
```

would also put 2.5 into the **Message Box**.

add, average, divide, extents, max, median, min, Operators, round, statRound, standardDeviation, sum

metaKey

the metaKey or metaKey()

Returns "up" or "down" depending on the position of the keyboard "Alt" key. This function is the same as the optionKey function.

altKey, commandKey, extendKey, keysDown, optionKey, shiftKey, mouse

milliseconds

the milliseconds or milliseconds()

Returns the milliseconds from some arbitrary time in the past. This function is most useful in pairs:

```
put the milliseconds into startTime
```

```
wait until the mouse is down
```

```
put "That took" && the milliseconds - startTime\
```

```
&& "milliseconds"
```

effectRate, idleRate, seconds, send, ticks, wait

min

min(<list>)

The min function finds the smallest number in a comma separated list of expressions or the elements in an array variable. If, for example, field "Items" had "7, 1, 4, 3" in it:

```
put min(field "Items")
```

would put 1 into the Message Box.

average, max, median, Operators, round

monthNames

the [long | abbreviated | short] [system] monthNames or monthNames()

This function returns a return-delimited list of the month names in English or, if the "system" parameter is supplied, the language specified for dates in the system control panel or locale environment variable.

centuryCutoff, convert, date, Operators, platform, seconds, time, useSystemDate, weekDayNames

mouse

the mouse or mouse([<button>])

Returns "up" or "down" depending on the state of mouse. If you don't specify a button, MetaCard checks the state of mouse button 1. Otherwise it checks the state of the specified button (1, 2, or 3, corresponding to left, middle, and right on most Windows and UNIX systems).

A good general rule is to never use this function because it will disrupt the normal functioning of button hilites and other automatic operations. Instead, handle the mouseDown, mouseUp, and mouseRelease messages.

click, clickLoc, commandKey, grab, keysDown, mouseChunk, mouseClick, mouseColor, mouseDown, mouseLoc, mouseMove, mouseRelease, mouseText, optionKey, rawKeyDown, screenMouseLoc, shiftKey, wait

mouseChar

the mouseChar or mouseChar()

The mouseChar function returns the character under the mouse cursor.

clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseChunk, mouseLine, mouseLoc, mouseText, selectedChunk, textStyle

mouseCharChunk

the mouseCharChunk or mouseCharChunk()

The mouseCharChunk function returns a chunk expression that describes the mouseChar. The string is of the form "char x to y of field z" where x, y, and z are integers and x = y.

clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseLine, mouseLoc, mouseText, selectedChunk, textStyle

mouseChunk

the mouseChunk or mouseChunk()

The mouseChunk function returns a chunk expression that describes the mouseText. The string is of the form "char x to y of field z" where x, y, and z are integers.

clickField, clickH, clickLine, clickLoc, clickText, clickV, foundChunk, hilitedLines, listBehavior, mouseCharChunk, mouseLoc, mouseText, selectedChunk, textStyle
mouseClick

the mouseClick or mouseClick()

This function returns true if mouse button 1 is has been pressed and released since the event that led to the current handler being executed was sent.

A good general rule is to never use this function because it will disrupt the normal functioning of button hilites and other automatic operations. Instead, handle the mouseDown, mouseUp, and mouseRelease messages.

clickLoc, commandKey, optionKey, shiftKey, mouse, mouseLoc, wait
mouseColor

the mouseColor

Returns the color of the pixel the mouse is over as an RGB triplet.

clickChunk, clickText, clickV, colormap, cursor, foundText, globalLoc, hide, localLoc, mouseChar, mouseChunk, mouseLoc, mouseUp, mouseStack, screenMouseLoc, selectedText, textColor, tool

mouseControl

the mouseControl or mouseControl()

The mouseControl function returns the layer of the control that has the mouse pointer within it. When the mouse wheel is turned on a Windows system, a rawKeyDown message is sent to the mouseControl.

clickField, clickStack, defaultStack, mouseColor, mouseEnter, mouseLeave, mouseLine, mouseLoc, mouseMove, mouseStack, rawKeyDown, screenMouseLoc, selectedField, topStack

mouseH

the mouseH or mouseH()

Returns the current x coordinate of the mouse cursor relative to the top left corner of the defaultStack.

clickLoc, foundLoc, mouse, mouseClicked, mouseH, mouseLoc, mouseMove, mouseStack, mouseV, mouseWithin, screenMouseLoc, screenRect, selectedLoc
mouseLine

the mouseLine or mouseLine()

The mouseLine function returns a **chunk** expression that describes the line of text that the mouse cursor is positioned over. The string is of the form "line x of field y" where x and y are integers.

clickLoc, foundLoc, hilitedLines, mouse, mouseClicked, mouseH, mouseLoc, mouseMove, mouseStack, mouseV, mouseWithin, screenMouseLoc, selectedLoc
mouseLoc

the mouseLoc or mouseLoc()

Returns the current x, y coordinates of the mouse cursor relative to the top left corner of the **defaultStack**. It's generally more efficient to use the **mouseMove** message and save the x,y coordinates of the mouse than to call this in a loop. This function can also disrupt the normal operation of automatic features like button hilite tracking.

Note that you can't set the mouseLoc, because it is a function. You can, however, set the **screenMouseLoc**.

clickLoc, cursor, foundLoc, globalLoc, localLoc, mouse, mouseChunk, mouseClicked, mouseColor, mouseControl, mouseH, mouseStack, mouseText, mouseV, mouseWithin, rawKeyDown, selectedLoc

mouseStack

the mouseStack or mouseStack()

The mouseStack function returns the name of the stack that has the mouse pointer within it.

clickStack, defaultStack, intersect, mouseColor, mouseControl, mouseEnter, mouseLeave, mouseLoc, screenMouseLoc, topStack, within

mouseText

the mouseText or mouseText()

The mouseText function returns the word under the mouse cursor.

cantModify, click, clickChunk, clickField, clickH, clickLine, clickLoc, clickText, clickV, foundText, hide, mouseChar, mouseChunk, mouseColor, mouseLine, mouseLoc, mouseUp, selectedText, textStyle

mouseV

the mouseV or mouseV()

Returns the current y coordinate of the mouse cursor relative to the top left corner of the defaultStack.

clickLoc, foundLoc, mouse, mouseClicked, mouseH, mouseLoc, mouseStack, mouseWithin, screenMouseLoc, selectedLoc

movie

the movie or movie()

This function returns a list of the names of the currently playing videoClip objects, or "done" if no movie is playing.

beep, movingControls, dontRefresh, play, playStopped, sound, stop, videoClipPlayer, wait

movingControls

the movingControls or movingControls()

Returns a list of the controls that are currently moving. Use the move command to start the controls moving.

move, moveStopped, movie, stacksInUse, stop

numToChar

numToChar(<expression>)

This function returns the character corresponding to the number supplied in the current character set.

baseConvert, charToNum, extendKey, format, isoToMac, md5Digest, replace, toLower, toUpper, uniEncode, urlEncode

offset

offset(<part>, <whole> [, <skip>])

This function returns the character offset where string <part> is found within string <whole>. If the part string does not appear within the whole, zero is returned. For example, these statements delete a word from a field:

```
put offset("text", field "Source") into tOffset
delete char tOffset to tOffset \
+ length("text") of field "Source"
```

If the optional third parameter <skip> is included, it specifies an offset within the string <whole> to begin the search. If it is omitted, it is set to 0. Remember to add the skip value to the value returned from offset to find the true offset within the string <whole>.

itemOffset, lineOffset, Operators, chunk, matchText, put, wordOffset
openFiles

the openFiles or openFiles()

This function returns a return separated list of the currently open files. You can use this function in an error handler that will close files left open elsewhere in your scripts:

```
repeat while the openFiles is not empty
close file line 1 of the openFiles
end repeat
```

close, kill, open, openProcesses, openSockets

openProcesses

the openProcesses or openProcesses()

This function returns a return separated list of the currently open processes. You can use this function to determine which processes should have an action performed on them. For example this script might notify a series of processes that maintain locks on some system resource that the status has changed:

```
put the number of lines in (the openProcesses)\
into nlines
repeat with i = 1 to nlines
if line i of the openProcesses contains "lock"
then kill USR1 process line i of the openProcesses
end repeat
```

close, kill, launch, open, openFiles, openProcessIds, openSockets, processId, signal
openProcessIds

the openProcessIds or openProcessIds()

This function returns a return separated list of the ids of the currently open processes. They are returned in the same order as the process names in the openProcesses. For example, the script on the openProcesses card could be modified to use the shell() function:

```
put the number of lines in the openProcessIds\
into nlines
repeat with i = 1 to nlines
if line i of the openProcesses contains "lock"
```



```
then put shell("kill -USR1"\
&& line i of the openProcessIds)\
into dummyvar
end repeat
close, kill, open, openFiles, openProcesses, processId, signal
openSockets
```

the openSockets or openSockets()

This function returns a list of the currently open sockets.

```
accept, close, hostAddress, open, openFiles, openProcesses, peerAddress,
socketTimeoutInterval
openStacks
```

the openStacks or openStacks()

This function returns a return separated list of the currently open stacks. You find out how these stacks were opened with **mode** property, and whether or not they are iconified with the **iconic** property.

```
backdrop, close, defaultStack, iconic, mainStack, mainStacks, mode, open, quit,
stacks, substacks, topStack, visible
optionKey
```

the optionKey or optionKey()

Returns "up" or "down" depending on the position of the keyboard "Alt" key. This function is the same as the altKey and metaKey functions.

```
altKey, click, extendKey, commandKey, drag, keysDown, metaKey, shiftKey,
mouse, type
param
```

param(<expression>)

This function returns a parameter by number, 1 being the first parameter. param(0) returns the name of the current handler.

Most handlers will have their parameters defined by name. The exception will be those handlers that have a variable number of parameters passed to them. In these cases, the param function can be used to access the parameters by position. For example, the following two handlers are equivalent:

```
on noparams
put param(1) into x
```

```
put param(2) into y
put x && y
end noparams
on twoparams x, y
put x && y
end twoparams
average, Messages, paramCount, params, send
paramCount
```

the paramCount or paramCount()

This function returns the number of parameters passed to a handler. The individual parameters can be retrieved with the `param` function.

```
average, Handlers, param, params, send
params
```

the params or params()

The `params` function returns a comma separated list of all of the params passed to a handler. The first param is the message name and the parameters are enclosed in quotes. For example the `mouseUp` handler might return:

```
mouseUp "1"
param, paramCount, send
peerAddress
```

peerAddress(<sock>)

This function returns the IP address of the remote host to which a socket is connected.

```
accept, hostAddress, hostName, hostNameToAddress, open, openSockets
pendingMessages
```

the pendingMessages or pendingMessages()

The `pendingMessages` returns the list of message that have been added with the `send` command that haven't been sent yet. Each message is on a line, and has the id of the message (which can be supplied to the `cancel` command), the time (as the long `seconds`) when the message should be delivered, the message name, and the name of the object it should be sent to.

```
cancel, flushEvents, result, seconds, send, waitDepth
platform
```

the platform or platform()

This function returns a string describing the hardware the current engine is running on. It is similar to the UNIX "uname" function, but somewhat easier to understand. This information could be used to send different commands to the `shell` function depending on the hardware/OS the engine is running on.

address, alternateLanguages, buildNumber, charset, dontUseNS, dontUseQT, drives, environment, files, fileType, formatForPrinting, hideConsoleWindows, longFilePath, lookAndFeel, kill, machine, macToISO, queryRegistry, screenGamma, screenVendor, secureMode, shell, shellCommand, signal, specialFolderPath, sysError, systemFileSelector, systemVersion, useSystemDate, version

processId

the processId or processId()

This function returns MetaCard's process id. It can be used to allow a process opened with `open process` to send a signal to MetaCard.

close, externals, kill, open, openProcesses, openProcessIds, platform, screenVendor, signal, windowId

propertyNames

the propertyNames or propertyNames()

The `propertyNames` function returns a list of the properties known to the MetaTalk interpreter. You can use this function to determine whether or not a custom property name you are planning to use is already defined:

```
put "somename" is in the propertyNames
```

colorNames, commandNames, customKeys, customProperties, functionNames, keys, variableNames

qtEffects

the qtEffects or qtEffects()

This function returns a list of the visual effects supported by the QuickTime library installed on the current system.

answer, buildNumber, environment, lookAndFeel, machine, platform, play, qtVersion, record, screenVendor, start, systemVersion, version, visual

qtVersion

the qtVersion or qtVersion()

This function returns the version of QuickTime installed on the current system, or 0.0 if QuickTime is not available. It returns a fixed value of "2.0" on UNIX systems.

buildNumber, environment, lookAndFeel, machine, platform, play, record, screenVendor, start, systemVersion, version

queryRegistry

queryRegistry(<key>[, <type>])

This function, which is only available in the Win32 engine, returns the value of the specified <key> in the system registry. The <key> parameter should be a full path to the key, beginning with one of the predefined handle values. The optional <type> parameter is a variable that will contain the type of the data in that registry entry if the function succeeds.

Key paths ending with the "\" character will query the default value for a key. For example, to get the default executable for the extension ".mc" on the current system:

```
put queryRegistry("HKEY_CLASSES_ROOT\.mc\", it) && it
```

copyResource, hideConsoleWindows, lookAndFeel, lowResolutionTimers, open, platform, setRegistry, systemFileSelector, systemVersion

recordFormats

the recordFormats or recordFormats()

This function returns a list of the audio codecs that the version of QuickTime installed on the current system supports. Each line contains the common name of the codec and the QT 4 character code that can be supplied to the record command.

buildNumber, paintCompression, play, qtEffects, qtVersion, recordLoudness, start, systemVersion, version, visual

recordLoudness

the recordLoudness or recordLoudness()

This function uses QuickTime to determine the strength of the signal at the current recording input.

buildNumber, environment, lookAndFeel, machine, platform, play, qtEffects, qtVersion, record, recordFormats, screenVendor, start, systemVersion, version, visual

replaceText

replaceText(<source>, <regex>, <replacement>)

This function replaces all occurrences in string <source> of a pattern specified by <regex> with the text in <replacement> and returns that string. For example, the following statement will put "xesx" into the Message Box:

```
put replaceText("test", "t", "x")
```

The parameter <regex> can be any regular expression, as defined in the `matchText` function. Note that this means if you want to replace any character with special meaning in a regular expression pattern (. + * | ? () [] \$ ^ \), you must escape that character by preceding it with a \ character (see the `replace` command for a faster and easier method for substituting characters).

For example, to replace the periods in a number with commas, you must escape the "." character in the regular expression pattern:

```
put replacetext("1.0", "\.", ",")# returns 1,0
```

caseSensitive, filter, find, format, lineOffset, matchText, offset, Operators, replace
random

random(<maxvalue>)

The random function returns a positive integer between 1 and <maxvalue> (which itself must be a positive integer).

When trying to select a random object or chunk, the word "any" can also be used to achieve random selections:

```
put any line of field 1 into somevariable
```

abs, Operators, randomSeed, round, sort, trunc

result

the result or result()

Returns a string describing the status of the last find, go, open, send, or file operation command. If the operation was successful, the result returns empty. Otherwise an error occurred. Since many operations can set the result, you should check it immediately after any statement that may set it.

The result function can also be used to retrieve a value returned from a message handler. See the `return` key card for an example. Note that the result is cleared at the end of a handler, which means that you can't check the result in the **Message Box**. The result also contains the id of a message added to the `pendingMessages` with the `send` command.

answer, ask, dialogData, directory, find, mciSendString, open, read, request, send, shell, sysError, write

round

round(<expression>[, <precision>])

The round function rounds a real number specified by <expression> to the nearest integer. 0.5 or larger rounds up. To do statistical rounding (as the HyperCard `round()` function does), use the `statRound` function.

The optional <precision> argument determines what decimal place the rounding occurs. The default of 0 rounds at the ones digit. A value of -1 rounds at the 10s digit, and a value of 1 rounds at the tenths place. For example round(55.55, -1) returns "60" and round(55.55, 1) returns 55.56.

abs, add, average, max, min, numberFormat, Operators, statRound, trunc
screenColors

the screenColors or screenColors()

Returns the number of color cells (the number of different colors the display can show at one time) of the current display. This is primarily useful when setting the colormap property.

colormap, foregroundPixel, platform, privateColors, screenDepth, screenName, screenRect, screenType, screenVendor

screenDepth

the screenDepth or screenDepth()

Returns the depth of the current display. Given this information certain decisions about color use become possible. For example, a monochrome screen has a depth of 1, and may require that special icons be displayed.

alwaysBuffer, colormap, dontDither, ink, lockColormap, pixmapId, platform, privateColors, screenColors, screenGamma, screenName, screenRect, screenType, screenVendor

screenLoc

the screenLoc or screenLoc()

Returns a point (two comma separated values) defining the center of the current display.

loc, mouseLoc, platform, rect, screenDepth, screenName, screenRect, screenType, screenVendor

screenName

the screenName or screenName()

Returns the name of the current display as returned by the XDisplayName call. This name should be passed after the -d flag to other X applications started up with the open and shell calls if they are to appear on the same display that MetaCard is currently running on.

open, platform, screenDepth, screenNoPixmaps, screenRect, screenSharedMemory, screenType, screenVendor, shell

screenRect

the screenRect or screenRect()

Returns a rectangle (four comma separated values) defining the edges of the main monitor. The first two values are always 0. This function can be used to rescale dialogs so that they will fit on the screen.

backdrop, globalLoc, mouseLoc, platform, rect, screenDepth, screenGamma, screenLoc, screenName, screenType, screenVendor, windowBoundingRect
screenType

the screenType or screenType()

Returns a string that describes the colormap allocation capabilities of the current display. The possible values are StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor, or DirectColor. StaticGray and StaticColor visuals are fixed which means that colors selected in MetaCard will always be remapped to the closest available color. TrueColor and DirectColor visuals are typically 24 (32) bit depth and all colors can be realized at the same time.

PseudoColor is the most common visual type on 8-bit displays. This visual type allows applications to define until the colormap is full. On UNIX/X11 systems, setting the privateColors property when using a PseudoColor display allows MetaCard to use the full colormap.

On some UNIX/X11 systems, it is possible to select a different visual type using the -v command line option. The X11 command "xdpyinfo" lists the visuals that are available on the current display. See the X Window System documentation for more information on "xdpyinfo" and visual types.

mouseLoc, platform, rect, privateColors, screenDepth, screenName, screenRect, screenVendor
screenVendor

the screenVendor or screenVendor()

Returns the name of the organization that produced and the release of the current server on UNIX/X11 systems.

beepPitch, platform, screenDepth, screenName, screenNoPixmap, screenRect, screenSharedMemory, screenType, textFont, windowId
scriptLimits

the scriptLimits or scriptLimits()

This function returns the limits on script length in the current environment. The limits are returned as a four-item list. The elements are, in order, the maximum number of statements in a scripts that can be set, the number of statements allowed in a do command, the number of stacks that can be used with start using, and the

number of objects that can be added to the message passing hierarchy with insert script.

If there are no limits, as is the case when running with a licensed Home stack, this function returns empty.

do, environment, insert, licensed, lookAndFeel, platform, save, script, start seconds

the [long] seconds or seconds()

Returns the seconds from some arbitrary time in the past. The modifier long can be supplied which will return the time as a real number which includes fractions of a second. This function is most useful in pairs:

```
put the long seconds into startTime
wait until the mouse is down
put "That took" && the long seconds - startTime\
&& "seconds"
```

convert, effectRate, idleRate, milliseconds, send, ticks, time, wait
selectedButton

the selectedButton of family <fam>

This function returns the currently hilited button in a button family. It is provided for compatibility with HyperCard. New development should use a group to hold the radio buttons and the group's hilitedButtonName property to determine which button is hilited.

family, hilited
selectedChunk

the selectedChunk or selectedChunk()

The selectedChunk function returns a chunk expression that describes the currently selected text. The string is of the form "char x to y of field z" where x, y, and z are integers. If there is no selection, y will be 1 less than x, which can be used to determine the offset of the text insertion cursor.

clickChunk, foundChunk, hilitedLines, select, selectedField, selectedLine, selectedLoc, selectedText
selectedField

the selectedField or selectedField()

Returns the number of the field that has the keyboard focus.

clickField, focus, foundField, mouseControl, select, selectedChunk, selectedLine, selectedLoc, selectedText

selectedLine

the selectedLine or selectedLine()

The selectedLine function returns a chunk expression that describes the **selectedText**. If a single line is selected, the string is of the form "line x of field y" where x and y are integers. If more than one line is selected, the string is of the form "line x to y of field z".

clickLine, foundLine, hilitedLines, listBehavior, menuHistory, select, selectedChunk, selectedField, selectedLoc, selectedText

selectedLoc

the selectedLoc or selectedLoc()

The selectedLoc returns the x, y coordinate of the top left corner of the selection.

clickLoc, foundLoc, select, selectedChunk, selectedField, selectedLine, selectedText

selectedObject

the selectedObject or selectedObject()

This function returns the name(s) of the selected object(s), one per line. This function is used extensively in the standard interface for the object property dialogs. Another useful short-cut is to use the pointer tool to select a text field with its **lockText** property set and type into the **Message Box**:

```
put "some text" into the selectedObject
```

This function can be abbreviated selObj or selObjs.

edit, editScript, errorObject, focus, me, select, selectedObjectChanged, selected, selectedField, selectedText, selectGroupedControls, selectionHandleColor, selectionMode, target

selectedText

selectedText() or the selectedText [of <control>]

The selectedText returns the currently selected text or the constant empty if there is no selected text. Note that unlike the **selection**, values can't be stored into the selectedText. If the optional <control> is specified and it is a field, the text selected in a **listBehavior** field is returned. Note that a field will only retain a selection when it loses the keyboard focus if its listBehavior property is set. If multiple lines are selected, the selectedText function returns multiple lines.

If <control> is a button, the text returned is the name of the menu item that was chosen the last time the button was opened as a menu. Note that this function is

supported only for compatibility with HyperCard. Using the `label` or `menuHistory` properties is the recommended technique in MetaCard, since you can use the same name to both get and set the text displayed.

`clickText`, `focus`, `foundText`, `hilitedLines`, `label`, `listBehavior`, `menuHistory`, `menuMode`, `select`, `selectedChunk`, `selectedLine`, `selectedLoc`, `selection`, `selectedObject`

`selection`

the selection or `selection()`

The selection function is similar to the `selectedText`, the difference is that values can be stored into the selection, just as for any other container.

`accentColor`, `hiliteColor`, `selectedObject`, `selectedText`

`setRegistry`

`setRegistry(<key>, <value>[, <type>])`

This function, which is only available in the Win32 engine, sets the specified `<key>` in the system registry. It returns true if the value was set, and false otherwise. Extended error information is return in the `result`.

The `<key>` parameter should be a full path to the registry entry, beginning with one of the predefined handle values. The optional `<type>` parameter creates an entry of that type. The possible values are "binary", "dword", "dwordlittleendian", "dwordbigendian", "expandsz", "link", "multisz", "none", "resourcelist", "string", "sz", with "string" being the default.

Key paths ending with the "\" character will set the default value for a key. For example, to set the default executable for the extension ".mc" on the current system:

```
get setRegistry("HKEY_CLASSES_ROOT\\.mc\\", "MetaCard")
```

`hideConsoleWindows`, `lookAndFeel`, `open`, `platform`, `queryRegistry`, `secureMode`, `systemFileSelector`, `systemVersion`

`shell`

`shell(<expression>)`

The shell function returns the results of executing a shell command. On UNIX systems, `<expression>` is any command that would be valid at the command line prompt of a Bourne shell (but see the `shellCommand` property if you need to use another shell). For example, to present a list of all the .txt files in the current directory:

```
put shell("ls *.txt") into field "files"
```

On Win32 systems, `<expression>` is an MS-DOS command, which can be the name of another Windows application. The `hideConsoleWindows` property can be used

to hide the console window that opens when a "console" application is run. On NT systems, the `shellCommand` can be changed to `cmd.exe`, a more reliable command interpreter than the default `command.com`.

The `stdout` and the `stderr` of the process being run are combined, so error messages may be part of the returned value. The `result` function can be used to get the exit code of the process. Note that unlike the `open process` command, the `shell` function blocks until the command has finished executing.

`directory`, `files`, `filter`, `format`, `hideConsoleWindows`, `kill`, `launch`, `matchText`, `mciSendString`, `open`, `platform`, `result`, `screenName`, `secureMode`, `send`, `shellCommand`, `sysError`, `windowId`

`shiftKey`

the `shiftKey` or `shiftKey()`

Returns "up" or "down" depending on the position of the keyboard "Shift" key.

`click`, `commandKey`, `capsLockKey`, `drag`, `keysDown`, `optionKey`, `mouse`, `type`

`shortFilePath`

`shortFilePath(<f>)`

This function returns the short file path corresponding to the long file path `<f>` on Win32 systems. A short file path is in the DOS-standard 8.3 format, and is most likely to be encountered as the `fileName` property of a stack that was passed on the command line. This function returns `<f>` on the other platforms, or if `<f>` already is a short file path.

`alternateLanguages`, `charset`, `dontUseNS`, `dontUseQT`, `drives`, `environment`, `charset`, `files`, `fileType`, `hideConsoleWindows`, `longFilePath`, `machine`, `macToISO`, `platform`, `queryRegistry`, `shell`, `shellCommand`, `sysError`, `systemFileSelector`, `systemVersion`

`sin`

`sin(<expression>)`

Returns the sine of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant `pi`.

`acos`, `asin`, `atan`, `atan2`, `Constatns`, `cos`, `numberFormat`, `put`, `tan`

`sound`

the `sound` or `sound()`

This function returns the name of the `audioClip` that is currently playing, or "done" if no sound is playing. `AudioClips` are started using the `play` command.

`beep`, `movie`, `play`, `playStopped`, `wait`

specialFolderPath

specialFolderPath(<name>)

This function takes a name as a parameter and returns the path to that folder on the current system. The supported folder names on MacOS systems are "Apple", "Desktop", "Control", "Extension", "Fonts", "Preferences", "Temporary", and "System". On Win32 systems, the supported names are "desktop", "fonts", "documents", "start", "system", and "temporary". This function is not supported on UNIX systems.

answer, create, open, platform, tempName

sqrt

sqrt(<expression>)

This function returns the square root of an expression.

baseConvert, exp, ln, Operators

stacks

the stacks or stacks()

This function returns a return separated list of the files where the currently open stacks are saved. This function is provided for HyperCard compatibility and in most cases the **openStacks** returns more useful information.

close, defaultStack, iconic, mainStack, mainStacks, mode, open, openStacks, substacks, topStack, visible

standardDeviation

standardDeviation(<list>)

This function computes the standard deviation of a comma separated list of expressions or the elements in an array variable.

add, average, divide, extents, max, median, min, Operators, round, statRound, sum

statRound

statRound(<exp>[, <precision>])

This function performs statistical rounding of a number as is implemented in the HyperCard round() function. Numbers of the form x.5 are rounded to x when x is even and x+1 when x is odd.

The optional <precision> argument is described in the round function.

abs, add, average, max, min, numberFormat, Operators, round, sum, trunc

sum

sum(<list>)

The sum function sums a comma separated list of expressions or the elements in an array variable. For example:

```
put sum(1, 2, 3, 4)
```

would put 10 into the Message Box.

The list could also be a single container. For example

```
put "1, 2, 3, 4" into somevar
```

or

```
put sum(somevar)
```

would also put 10 into the Message Box.

add, average, divide, max, median, min, Operators, round, statRound, standardDeviation, trunc

sysError

the sysError or sysError()

The sysError returns the current value of the system error variable (the errno variable on UNIX/X11 systems, and GetLastError() on Windows systems). It can be used to determine the reason for a failure of an open command.

errorDialog, kill, lock, open, platform, result, shell

systemVersion

the systemVersion or systemVersion()

This function returns the version of the operating system.

alternateLanguages, buildNumber, charset, environment, lookAndFeel, machine, platform, qtVersion, shellCommand, version

tan

tan(<expression>)

Returns the tangent of a numeric expression (in radians). To convert to degrees, multiply by 180 and divide by the constant pi.

acos, asin, atan, atan2, Constants, cos, numberFormat, put, sin

target

the target or target()

This function can be used to get properties of the object that originally received the message that started a script executing. Compare with the me function which accesses the object whose script is currently executing. Note, a common mistake

made with this function and the me function is to forget that the value returned includes the type of the object:

put the target -- returns field "name"

put the short name of me -- returns "name"

backScripts, editScript, errorObject, frontScripts, insert, me, selectedObject, text
templateAudioClip

the templateAudioClip or templateAudioClip()

This function allows setting the properties of the audioClips that will be imported.

templateButton, play, playDestination

templateButton

the templateButton or templateButton()

This function allows setting the properties of the buttons created with the create command or by dragging with the button tool. For example, instead of having to create a rectangle button and then change its style with the "Button Properties" dialog box you can set the templateButton to create a checkBox button directly:

set the style of the templateButton \

to "checkBox"

create button

reset templateButton

choose, create, me, Properties by Name, reset, style, target, templateCard,
templateField, templateGraphic, templateGroup, templateImage, templatePlayer,
templateScrollbar, templateStack

templateCard

the templateCard or templateCard()

This function allows setting the properties of the cards created with the create command. Note that cards created will also have the backgrounds used on the current card placed onto them.

create, me, Properties by Name, reset, style, target, templateButton, templateField,
templateGraphic, templateGroup, templateImage, templatePlayer, templateScrollbar,
templateStack

templateEPS

the templateEPS or templateEPS()

This function allows setting the properties of the EPS objects created with the create

command or the `import` command.

`create`, `me`, `Properties by Name`, `style`, `target`, `templateButton`, `templateField`, `templateGraphic`, `templateGroup`, `templateImage`, `templatePlayer`, `templateScrollbar`, `templateStack`

`templateField`

the `templateField` or `templateField()`

This function allows setting the properties of the fields created with the `create` command or by dragging with the field tool. For example, instead of creating a rectangle field and then changing its style with the "Field Properties" dialog box, the following function:

```
set the style of the templateField \  
to "scrolling"  
create field  
reset templateField
```

will create a scrolling field directly.

`choose`, `create`, `me`, `Properties by Name`, `reset`, `style`, `target`, `templateButton`, `templateCard`, `templateGraphic`, `templateGroup`, `templateImage`, `templatePlayer`, `templateScrollbar`, `templateStack`

`templateGraphic`

the `templateGraphic` or `templateGraphic()`

This function allows you to set the properties of graphics either created with the `create` command or created by dragging with the graphic tool. For example, instead of having to create a rectangle graphic and then having to change its style with the "Graphic Properties" dialog box, you can simply set the style of the `templateGraphic` and create an oval directly:

```
set the style of the templateGraphic to "oval"  
create graphic  
reset templateGraphic
```

`choose`, `create`, `me`, `Properties by Name`, `reset`, `style`, `target`, `templateButton`, `templateCard`, `templateEPS`, `templateField`, `templateGroup`, `templateImage`, `templatePlayer`, `templateScrollbar`, `templateStack`

`templateGroup`

the `templateGroup` or `templateGroup()`

This function allows you to set the properties of the groups created with the `create` command.

create, me, Properties by Name, radioButton, target, templateButton, templateCard, templateField, templateGraphic, templateImage, templatePlayer, templateScrollbar, templateStack

templateImage

the templateImage or templateImage()

This function allows you to set the properties of the images created with the **create** command.

create, me, Properties by Name, reset, target, templateButton, templateField, templateCard, templateGraphic, templateGroup, templatePlayer, templateScrollbar, templateStack

templatePlayer

the templatePlayer or templatePlayer()

This function allows you to set the properties of the players created with the **create** command.

create, me, Properties by Name, reset, target, templateButton, templateField, templateCard, templateGraphic, templateGroup, templateScrollbar, templateStack

templateScrollbar

the templateScrollbar or templateScrollbar()

This function allows you to set the properties of the scrollbars created with the **create** command.

create, me, Properties by Name, reset, target, templateButton, templateField, templateCard, templateGroup, templateImage, templatePlayer, templateStack

templateStack

the templateStack or templateStack()

This function allows you to set the properties of the stacks created with the **create** command. One of the most common applications of this function is to set the initial position of a stack:

```
set the visible of the templateStack to false
```

```
create stack "my stack"
```

```
set the rect of stack "my stack" to\
```

```
to "50,50,400,400"
```

```
reset templateStack
```

create, me, Properties by Name, reset, target, templateButton, templateCard, templateField, templateGraphic, templateGroup, templateImage, templatePlayer,

templateScrollbar

templateVideoClip

the templateVideoClip or templateVideoClip()

This function allows you to set the properties of the videoClips that will be imported or played directly from disk:

set the dontRefresh of the templateVideoClip to true

set the scale of the templateVideoClip to 2.0

play videoClip "mymovie.avi"

create, dontRefresh, frameRate, import, mciSendString, play, scale, templateButton, videoClipPlayer

tempName

the tempName or tempName()

This function returns a file name in the OS-specific temporary directory that is guaranteed to be unique.

create, delete, fileName, open, specialFolderPath

textHeightSum

textHeightSum(<object>)

This function returns the total height of all the text in a button or field. It is provided primarily for backward compatibility with SuperCard, and using the more natural formattedHeight property is preferred for new development.

formatForPrinting, formattedHeight, formattedWidth, pageHeights, textHeight

there

there is a <thing>

This validation operator is used to verify that an object exists before trying to access it. <thing> is any object (background, button, card, field, group, image, scrollbar, stack) or the word "file", "directory", or "process". For example, to prevent errors when a user clicks on a term that does not have a card describing it, the following statements are used in a mouseUp handler in this stack:

if there is a card the clickText

then go to card the clickText

aliasReference, cardNames, convert, create, directories, exists, files, go, intersect, longFilePath, number, open, Operators, rename, shortFilePath, within

ticks

the ticks or ticks()

Returns the clock ticks (1/60 of a second) from some arbitrary time in the past. This function is most useful in pairs:

```
put the ticks into startTime  
wait until the mouse is down  
put "That took" && the ticks - startTime\  
&& "ticks"
```

effectRate, idleRate, milliseconds, seconds, send, wait
time

the [long | abbreviated] time or time()

The time function returns the time according to the system clock. The modifier is optional and must be one of long, short, or abbreviated. If omitted, the short time is returned.

The long time returns the time of the form: HH:MM:SS [AM | PM]

The abbreviated (or abbrev or abbr) and short form is: HH:MM [AM | PM]

AM or PM will not be shown if the twelveHourTime property is set to false.

convert, date, seconds, send, twelveHourTime, wait
toLower

toLower(<expression>)

The toLower function converts a string to all lower case characters.

baseConvert, charset, charToNum, compress, format, isoToMac, macToISO,
numToChar, toUpper

tool

the tool or tool()

This function returns the name of the current tool. See the choose command for details. For SuperCard compatibility, tool can also be used as a stack property.

choose, create, editMenus, mode, mouseColor, newTool, select, selectionMode,
templateGraphic

topStack

the topStack or topStack()

This function returns the name of the top-most stack. The top-most stack is defined to be the open stack with the lowest mode. If more than one stack has that mode, the stack that was most recently activated (had the keyboard focus) is the topStack.

backdrop, close, defaultStack, editMenus, go, mainStack, mainStacks, menubar, mode, open, openStacks, substacks, topLevel, visible

toUpper

toUpper(<expression>)

The toUpper function converts a string to all upper case (capital) characters.

baseConvert, charset, charToNum, format, isoToMac, macToISO, numToChar, toLower

transpose

transposr(<a>)

This function transposes a multidimensional numerically indexed array <a> so that, for example, a[2,1] becomes a[1,2].

add, divide, extents, max, median, min, matrixMultiply, multiply, Operators, round, statRound, standardDeviation, sum

trunc

trunc(<expression>)

The trunc function truncates the fractional part of a real number, leaving a whole number (integer).

abs, average, max, min, numberFormat, Operators, round, statRound

uniDecode

uniDecode(<expression>)

This function converts a UniCode string back to a normal character string. Little-endian (Intel, Alpha, etc.) format is assumed.

base64Decode, baseConvert, binaryDecode, charToNum, convert, load, numToChar, post, put, toLower, toUpper, uniEncode, urlDecode, urlStatus

uniEncode

uniEncode(<expression>)

This function converts a string to a UniCode string. It does this by adding a null (0) byte after each character. The resulting format is what is required when writing to files on little-endian systems (Intel, Alpha, etc.). No provision is made for producing the format required for UniCode files on big-endian systems (SPARC, PA-RISC, PowerPC).

base64Encode, baseConvert, charToNum, convert, isoToMac, load, numToChar, post, put, toLower, toUpper, uniDecode, urlEncode, urlStatus

urlDecode

urlDecode(<expression>)

This function decodes a string posted from an HTML form. All "+" characters are converted to spaces, and all hex encodings of the form %XX are decoded to their character equivalents.

base64Decode, baseConvert, binaryDecode, charToNum, convert, decompress, load, numToChar, post, put, toLower, toUpper, uniDecode, urlEncode, urlStatus

urlEncode

urlEncode(<expression>)

This function encodes a string as if it were posted from an HTML form. All spaces are replaced with "+" characters, and all special characters are converted to hex encodings of the form %XX.

base64Encode, baseConvert, binaryEncode, charToNum, compress, convert, decompress, files, httpHeaders, load, macToISO, numToChar, post, put, toLower, toUpper, urlDecode, urlStatus

urlStatus

urlStatus(<url>)

This function returns the status of the URL specified by <url> in the local cache. The return value can be any of the following:

contacted - the site has been contacted

requested - the URL has been requested

not found - the URL was not found in the cache

loading,x,y - y is total size, x is bytes downloaded

cached - the URL download is complete

error - an error occurred during download

timeout - a timeout error occurred

cachedUrls, compress, httpHeaders, load, post, put, urlEncode

value

value(<expression> [, <object>])

The value function returns the value of an expression stored in a container. For example if field "equation" had "1 + 3 / 2" in it:

put the value of field "equation"

would put "2.5" into the Message Box.

The optional <object> expression uses that object as the context for the evaluation instead of the current object.

alternateLanguages, call, do, Functions, me, merge, Operators, request, send
variableNames

the variableNames or variableNames()

Returns a list of the all the variables. The first line contains the parameters passed to the current handler. The second contains the local variables created in the current handler. The third line contains the local variables for the entire script. The fourth line contains global variables that can be used in the current script.

See the **container** description for more information on variables.

constant, functionNames, global, globalNames, keys, local, localNames,
propertyNames

version

the version or version()

This function returns the version number of the currently executing MetaCard engine.

buildNumber, environment, lookAndFeel, machine, platform, qtVersion,
screenVendor, systemVersion

waitDepth

the waitDepth or waitDepth()

This function returns the number of "wait for messages" commands that are currently executing.

accept, pendingMessages, send, time, wait

weekdayNames

the [long | abbreviated | short] [system] weekdayNames

This function returns a return-delimited list of the weekday names in English or, if the "system" parameter is supplied, the language specified for dates in the system control panel or locale environment variable.

centuryCutoff, convert, date, monthNames, Operators, platform, seconds, time,
useSystemDate

windows

the windows or windows()

The windows function is a synonym for the openStacks function.

mode, openStacks, topStack

within

within(<object>, <point>)

This function returns true if the point is within the visible region of the object, and false otherwise.

```
if within(button "OK", the mouseLoc)
```

```
then show field "help"
```

```
else hide field "help"
```

rect, intersect, layer, loc, mouseStack, Operators, owner

wordOffset






wordOffset(<part>, <whole> [, <skip>])



This function returns the number of the word where string <part> is found within string <whole>. If the part string does not appear within the whole, zero is returned. Words are delimited by spaces, returns, and tabs and *not* by most punctuation.

If the optional third parameter <skip> is included, it specifies a number of words to skip before beginning the search. If it is omitted, it is set to 0. Remember to add the skip value to the value returned from offset to find the true word offset within the whole string.

caseSensitive, itemOffset, lineOffset, offset, Operators, matchChunk, matchText, put, switch, wholeMatches

Metaclasses

	Article	Description	Rating
9	The Basics: Understanding stacks, subStacks and mainStacks	Fluency with MetaCard depends on a knowledge of the Card and Stack metaphor. Essential reading for all beginners.	
1	The Message Hierarchy	A basic article explaining the key concept of messages and the message hierarchy.	
2	Using the Send Command	Forget idle handlers! Use the Send command to implement asynchronous events and scheduling.	
3	StackFiles	The magic formula for easily and efficiently splitting up your stacks prior to standalone building.	
6	Groups and Backgrounds	An introduction to groups and backgrounds, what you can do with them and what sets them apart.	
4	Text Management	An in depth look at the matchText , matchChunk and replaceText functions.	
5	Finding and Replacing Text	Covers the find command, and explores the offset function as a more efficient alternative. Also covers the replace command.	

7	Using Files with MetaCard	Reading and Writing to files using MetaCard including downloading from the Internet and launching applications.	
8	Doing Menus in MetaCard	Everything you need to know about menus, covering menubars, pulldown and popup menus, combo boxes etc.	

Article

Difficulty Rating

2

The Basics: Understanding stacks, subStacks and mainStacks

A window in MetaCard is called a stack. Each stack can contain any number of screens, known as cards. Cards contain objects such as buttons, text fields, images, etc. By changing the card currently being displayed in a stack, you change the entire contents of the screen being displayed in that stack (window).

Stacks have a number of properties to control how they are displayed. These include basics such as width, location and height. There are also commands to control what mode stacks are opened in, i.e. whether they are displayed as palettes, normal (toplevel) windows or dialog boxes.

Not all stacks are created equal. There are two types: the **mainStack** and the **subStack**. The analogy is simple. A mainStack is stored in a file. SubStacks are stored in mainStacks. When you open a MetaCard "document" on disk, the mainStack will be displayed. Scripts in the mainStack can then open any subStack in that file.

SubStacks are used to implement any other window in the application you are creating, for example dialog boxes, menus, or other program screens.

Lets walk through creating a new mainStack, and then the process of attaching a subStack to it. Open MetaCard, and create a stack by choosing "New stack" from the file menu. A new stack will appear, called something like "Stack 00000000". The stack properties palette for that stack is automatically opened when you create a new stack, so use it to rename the stack so that we can refer to it easily again.

Now choose Save from the file menu. You will be presented with a standard Save As dialog box, which allows you to give the stack a file name and save it on disk. Type in "My program.mc" and press the Save button.

Now, create a new stack. You could just save this stack into another file as another mainStack by choosing Save from the File menu again. However, this time, lets save it into the same file as the mainStack "My stack" we just created.

Article	Difficulty Rating	
9	Messages and the Message Hierachy	

The scripting language in MetaCard is based on the principal of messages. A good understanding of this principal and of the hierarchy or path messages take is essential to efficiently learning MetaCard.

Whenever something happens, a message gets sent. These messages can be generated directly by the user, or by a script. For example, when someone clicks on a button, it will receive a `mouseDown` message as the mouse is pressed, and then a `mouseUp` message when the mouse is released over the button. The `mouseUp` message is the most usual message to use to respond to a user clicking on a button. Other messages include `mouseEnter` for when a mouse moves over an object, `mouseLeave` for when a mouse moves out of an object, `mouseMove` for when the mouse moves whilst over an object, `keyDown` for when a key is pressed, `preOpenCard` for just before a card is displayed on the screen, and many, many more.

To respond to a message, you write a script. At its most basic, you select the object you want to write the script for with the pointer tool, right-click (or control-click if you are using the MacOS) and choose Edit Script from the contextual menu that comes up. You then type in a routine or *handler* that starts with the name of the message you want to respond to.

Here is an example of a very basic handler you can place in any button:

```
on mouseUp
  beep
  move me relative 100,0 in 5 seconds without waiting
end mouseUp
```

This handler will cause the system to beep and the object to move 100 pixels to the right, when the mouse is pressed and released on the object that contains this script.

You can make the behavior a little more complex by including another handler.

```
on moveStopped
  set the location of me to 200,200
end moveStopped
```

This handler will cause the button to jump to a location 200 pixels right and down

from the top left of the card. It is an example of a message generated by MetaCard in response to an event (the button stopping moving), as opposed to a message directly sent by the user (e.g. when the user pressed the button).

The message hierarchy

In the example above, it is very clear where messages go. The user clicks on the button, and the `mouseUp` message is received by the button, which then does the `mouseUp` handler. However, it is not always so simple. Suppose there isn't a `mouseUp` handler in the button. When the user presses the button, the message gets sent to the button anyway. *As there is no handler for the message in the button, MetaCard then sends the message to the next object in the message path.*

The message path (also know as the message hierarchy) is a logical system for working out what object gets what message. Here is the path:

button -> card -> stack -> mainStack -> Home Stack -> MetaCard

Note that the first item in that path, the button, could be any object residing on a card, such as a field, image, etc.

There are a simple set of rules to determine where in that path a message will get sent and intercepted.

- Messages are always sent to the lowest possible object in that path. For example, if a button is clicked on, a `mouseUp` message will be sent to that button. If a card is opened, a `preOpenCard` message is sent to that card.
- Messages that are intercepted by handlers do not get any sent further up the path, so all messages only get processed once
- Messages that are not intercepted get sent up the path until they are intercepted
- If messages are never intercepted, they reach MetaCard, which may or may not do something with the message

Why would you want a message to be passed up a path? Surely if you haven't placed a message handler in a button for a particular message, you don't intend the message to be acted on? This is not the case. Message passing is actually a very useful feature, because it means that you can write the minimum number of scripts to achieve the effect you want. For example, if you have 10 buttons on a card that all do the same thing when clicked, you can put a `mouseUp` handler in the card, and put the common commands in it.

What if you have some objects that all use the same script, but others that don't? A simple way of dealing with this is to insert the objects into a group, and place the common script into the group object. When an object is grouped, the message path looks like this:

button -> **group** -> card -> stack -> mainStack -> Home Stack -> MetaCard

Another way of doing it is to check the `target` function. That function contains the name of the object that would originally have received the message, and can be

used by handlers further up the message path to determine if they need to take action, and what action to take. Place the following example into a card script:

```
on mouseUp
  put the target
end mouseUp
```

Now create multiple objects on the card, choose the browse tool to switch to run time mode, and click each object. The message box will display the object clicked on. (The `put` command is used to place text into the message box.)

What happens when the message reaches MetaCard? That depends on the message. In most cases, MetaCard doesn't do anything. However, in some cases it does. For example, if MetaCard receives a `closeStackRequest` message, it will close the current stack. And if it receives a `keyDown` message it will place the key that has just been typed into the field with the insertion point. Intercepting these messages is useful to prevent a user doing something, e.g. putting inputting invalid data into a field (you can check what the user typed before putting anything in the field) or closing a stack without saving (you can ask the user if he or she wants to save before closing).

Special cases

What if you have a handler in an object that intercepts a message, does something, but then you want the message to continue up the hierarchy so that a handler further up can also process the message? This is useful if you have a common script that does processing after specific processing has been performed by each object. It is also useful in cases where you have used a message such as `keyDown` to validate a key for entry into a field, but now want to allow the data entry to proceed having checked it. The solution is to use the `pass` command. This command causes the message to be sent up the hierarchy, just as if it had never been intercepted. This script would only enter text into a field if the user typed a number:

```
on keyDown theKey
  if theKey is a number then pass keyDown
  else beep
end keyDown
```

Note the third word on the first line of that script "theKey". TheKey is a variable that contains the actual key the user pressed. Many messages come with optional parameter information like this. For example, the `mouseUp` message comes with the number of mouse button that was pressed. On Mac and Windows systems that will usually be 1 or 3 (use the control-key to do a right click on MacOS systems), on UNIX workstations, there may also be a button number 2. The following script displays which mouse button was used:

```
on mouseUp whichButton
  put whichButton
end mouseUp
```

What if you want to include a script before or after all other scripts temporarily? For example, you might want to capture every user action before it was sent to any object and do something with it. An example would be displaying a coordinates window with the mouse coordinates. You would want to intercept the mouseMove message before it got sent, and potentially used, by any other object. The solution to this type of problem is to use frontScripts and backScripts. FrontScripts are object scripts inserted *before* the message hierarchy, and backscripts are scripts inserted *after* the message hierarchy, but *before* MetaCard itself.

frontScript -> button -> card -> stack -> mainStack -> Home Stack -> **backscript** -> MetaCard

Here is an example of a frontScript in use. Create a button, name it "coordinates" (select it and choose Object Properties... from the Edit menu to change its name) and place the following script into it:

```
on mouseMove
  put the mouseLoc
end mouseMove
```

If you choose the browse tool and move the mouse over that button, you will see that the location of the mouse is displayed in the message box. Now, type the following into the message box and press return:

```
insert the script of button "coordinates" into front
```

Now move the mouse around the card and over other objects. The coordinates are displayed as you move. To remove this script:

```
remove the script of button "coordinates" from front
```

To remove all scripts from front:

```
remove all scripts from front
```

It is always a good idea to include a pass statement at the end of a frontScript. This allows the message intercepted in the frontscript to go on to be received by the object that would normally expect that message, such as the button under the mouse. So in the coordinates example above, you should insert a line before end mouseMove that says pass mouseMove.

Backscripts are the same as frontScripts, except that these get called only at the end of the message path, and therefore won't be called if any handler intercepts the message. They are often useful to prevent a user from closing any stack in large series of stacks, or to do a check on a certain type of message that must always be intercepted somewhere in the path. Passing backScript messages is only necessary where you want the message to continue on to MetaCard itself to be acted on.

The mainStack script


The mainStack script is worthy of particular note. Any message sent to any object, card or subStack stored in the same file as the mainStack, that is not intercepted by an object will be sent through the mainStack. This has two important ramifications. The first is that you can use the mainStack to store frequently used commands and functions - basically anything that multiple parts of your program might want access to. The second is that the mainStack may intercept and act on messages you don't want it to act on if you're not careful.

For example, a handler for the preOpenStack message (sent to all stacks when they are opened) will be executed *every* time a subStack stored in the same file as the mainStack is opened, providing the message isn't intercepted by that subStack first. Sometimes that's useful; often it isn't. To avoid it, set a variable in the mainStack script like this:

```
on preOpenStack
  global gHaveDonePreOpenStack
  if gHaveDonePreOpenStack is true then exit to MetaCard
  put true into gHaveDonePreOpenStack

--put the rest of the handler here.....
```

As you probably guessed, that handler will only execute once - i.e. the first time the mainStack is loaded. The global variable gHaveDonePreOpenStack retains its value until you quit, and gets set to true the first time the handler is executed. The handler will always be exited before execution in subsequent cases.

Article	Difficulty Rating	
8	Using the Send Command	

► Using the "Send" Command to Schedule Future Events and Animation

You wouldn't guess from the name, but the send command it is actually one of MetaCard's most important features. This article details using it to schedule events and animations.

This is a basic send command:

```
send "mouseUp" to me in 5 seconds
```

That statement causes the message "mouseUp" to be delivered to the same object that sent it, 5 seconds from when the command was executed. The MetaCard engine is not occupied during that period, and is therefore free to do other things. After 5 seconds have elapsed, that message will be delivered to the object.

The advantages of this approach are many. You can cause messages to be sent at any time in future. There is no measurable performance decrease in the interim between sending and receiving the message. MetaCard remains free to do other tasks during that period. This completely eliminates the need to use messages such as "idle", a common, complicated and processor intensive kludge used in many other tools.

► Scheduling a Repeat Loop

Setting something to occur regularly is easy. The following script updates a field on screen to show the time, once every ten seconds.

```
on mouseUp
  --this handler starts the timer
  setClock
end mouseUp
```

```
on setClock
  put the time into fld 1
```

```
    send "setClock" to me in 10 seconds
end setClock
```

This example would repeat forever, putting the time into field 1 every 10 seconds.

We can easily modify this example to be more useful. We'll include a facility to cancel the message, and also make the time update much more frequently, and include seconds.

```
local lTimerID
```

```
on mouseUp
  --this handler starts the timer
  setClock
end mouseUp
```

```
on setClock
  put the long time into fld 1 --long time also has
seconds
  send "setClock" to me in 600 milliseconds --600
milliseconds or 0.6 seconds
  --will be enough to keep the seconds on the clock
current
  put the result into lTimerID
  --the above line stores the "ID" of the message just
sent
end setClock
```


The result contains a unique ID for the message you have sent. You can stop that message from being delivered at any time by passing that ID to the cancel command:

```
cancel lTimerID
```

The pendingMessages function contains a list of all the messages scheduled to be delivered in the future. Messages are added to it every time an event is scheduled, and removed when the message is delivered (or cancelled with the cancel command).

Here is a line from the pendingMessages:

```
12,913382037.933333,setClock,button id 1003  
of card id 1002  
of stack "Stack 913381993"
```

The first item is the ID of the message. You can cancel the message using this ID just in the same way you would cancel it using the result function to retrieve and store the id.

```
cancel (item 1 of the pendingMessages)
```

The second item contains the time the message will be delivered at. The third item contains the name of the message being delivered, and the fourth item contains the path for the object that the message will be delivered to.

The pendingMessages function can be used to check if a message has already been scheduled. In the above example of updating a clock, a problem may arise if the user presses the original button used to trigger the script for a second time. Two sets of the same message will then start being sent, and the clock will be updated twice as frequently as originally specified. Repeated clicking of the button will eventually cause messages to be sent almost constantly, locking up MetaCard. Inserting the following line into the top of the mouseUp message prevents this problem:

```
if "setClock" is in the pendingMessages then  
exit mouseUp
```

This will prevent the mouseUp handler from executing if the cycle has already been started. Of course, advanced users reading this may have noticed it is a few milliseconds more efficient to set up a variable the first time the handler is executed, and check

```
on closeCard
  repeat for each line l in the pendingMessages
    cancel (item 1 of l)
  end repeat
end closeCard
```

That repeat loop is useful to remember. Run it in the message box or keep it handy in a button when you're debugging any complex send based script. You may want to use it in an emergency to clear the `pendingMessages` when you get into something you find you can't get out of.

► **Doing animation with "send"**

An obvious use for the send command is to do animation. Carefully done, animation can be virtually asynchronous, meaning that MetaCard can respond to events while the animation is taking place. For a smooth animation, it is important that the individual handlers executed by the send command are as short as possible. Where multiple things need to be done, it is best split into multiple handlers.

The reason for this is simple: only one script can be executing at a time. Send is no exception. Like any other command, when a send command is running, other scripts cannot run. In fact, a send command can only be delivered if there is no other handler running. Otherwise, it will wait until the end of whatever handler is running to be delivered. Keeping all the handlers that execute short, and splitting longer scripts into multiple handlers delivered by `send`, allows you to maintain user interaction whilst updating the screen.

Why is MetaCard not truly multi-threaded? A multi-threaded engine would be capable of running multiple scripts at once. MetaCard does not support this, because of the complexities of creating and debugging a multi-threaded application. The `send` command is provided as an alternative, and if used well, can achieve excellent results without the hassle of keeping track of multi-threaded script or code. Note that the GIF animation commands and the `move` command are exceptions to this: they will both continue to run when other scripts are running (with a few exceptions such as using the file access commands to read in huge files).

It is a good idea to use the `move` command in conjunction with animated GIFs to do the bulk of any animation wherever possible. You may want to start and stop GIF animations, start and stop move commands, manually alter the frame being shown in a GIF animation, alter button icons in a moving button, start and stop sounds, show and hide objects, scroll fields, or use various other techniques or a combination of all of these to produce a presentation. All of these can be controlled, while still allowing user interaction, with a series of carefully scheduled

```

local lCurrentFieldScroll, lCurrentEndValue

on mouseUp
  put 0 into lcurrentFieldScroll --start with the field
  scrolled to 0
  put 500 into lCurrentEndValue --the vScroll of the
  field when fully scrolled down
  updateField
end mouseUp

on updateField
  add 10 to lCurrentFieldScroll
  if lCurrentFieldScroll > lCurrentEndValue then exit
updateField
  set the vScroll of fld "example" to lCurrentFieldScroll
  send "updateField" to me in 100 milliseconds
end updateField

```

This example will scroll a field, incrementing every 100 milliseconds. For basic animation, this is often enough. The script will take the same order of magnitude of time to run regardless of the speed of machine it is run on, and varying the delay between messages can be used to alter the speed of the animation.

However, if you need to guarantee that the script will take a certain amount of time to run, you need to do a little more work. Other messages could get delivered when that field is scrolling, or the machine might be busy and therefore slow down slightly. If you are playing back audio or other parts of a presentation to be exactly in sync, you cannot allow this field to be left behind or gradually become out of sync if anything else on the computer causes a glitch or slow down.

Here is a new script that will ensure that the field scrolling is complete inside ten seconds. If there is a delay or interruption causing messages to be delivered late, the scrolling will jump to the correct position when processor time is returned, rather than increase the overall time taken to complete the effect.

```

local lCurrentFieldScroll, lCurrentEndValue, lTotalTime,
lStartTime

```

```

on mouseUp
    put 10000 into lTotalTime -- the total time allowed in
milliseconds
    put 0 into lcurrentFieldScroll --start with the field
scrolled to 0
    put 500 into lCurrentEndValue --the vScroll of the
field when fully
                                --scrolled down
    put the milliseconds into lStartTime --measure the
animation from
                                --this start time


    updateField
end mouseUp

on updateField
    put the milliSeconds - lStartTime into tCurrentTime
    if tCurrentTime > lTotalTime then --we've reached the
end
        set the vScroll of fld "example" to tEndValue --
ensure that the
                                --field
is at the end
        exit updateField --don't send this message again
    end if
    put tCurrentTime / lTotalTime * lCurrentEndValue into
lCurrentFieldScroll
    --thats the position the scroll bar should be based on
the time elapsed
    set the vScroll of fld "example" to round
(lCurrentFieldScroll)
    -- rounding is required or the script will not work
    send "updateField" to me in 50 milliseconds
end updateField

```

You can use the above script for just about any time related activity that needs to be performed on time. Just alter the variables in the first handler, such as the total length, the total number of frames (in this case the total scroll value of the field). You can also change how frequently the message is delivered. This message was delivered every 50 milliseconds, changing that value down the way will result in a smoother animation, and changing the value up the way will leave more processor time free. But altering that interval will *not* alter the *total* length of time to complete the animation.

Enjoy experimenting and using the send command! And look out for a timeline animation tool that automates writing the scripts in our forthcoming Editor for MetaCard.

Article	Difficulty Rating	
7	StackFiles	

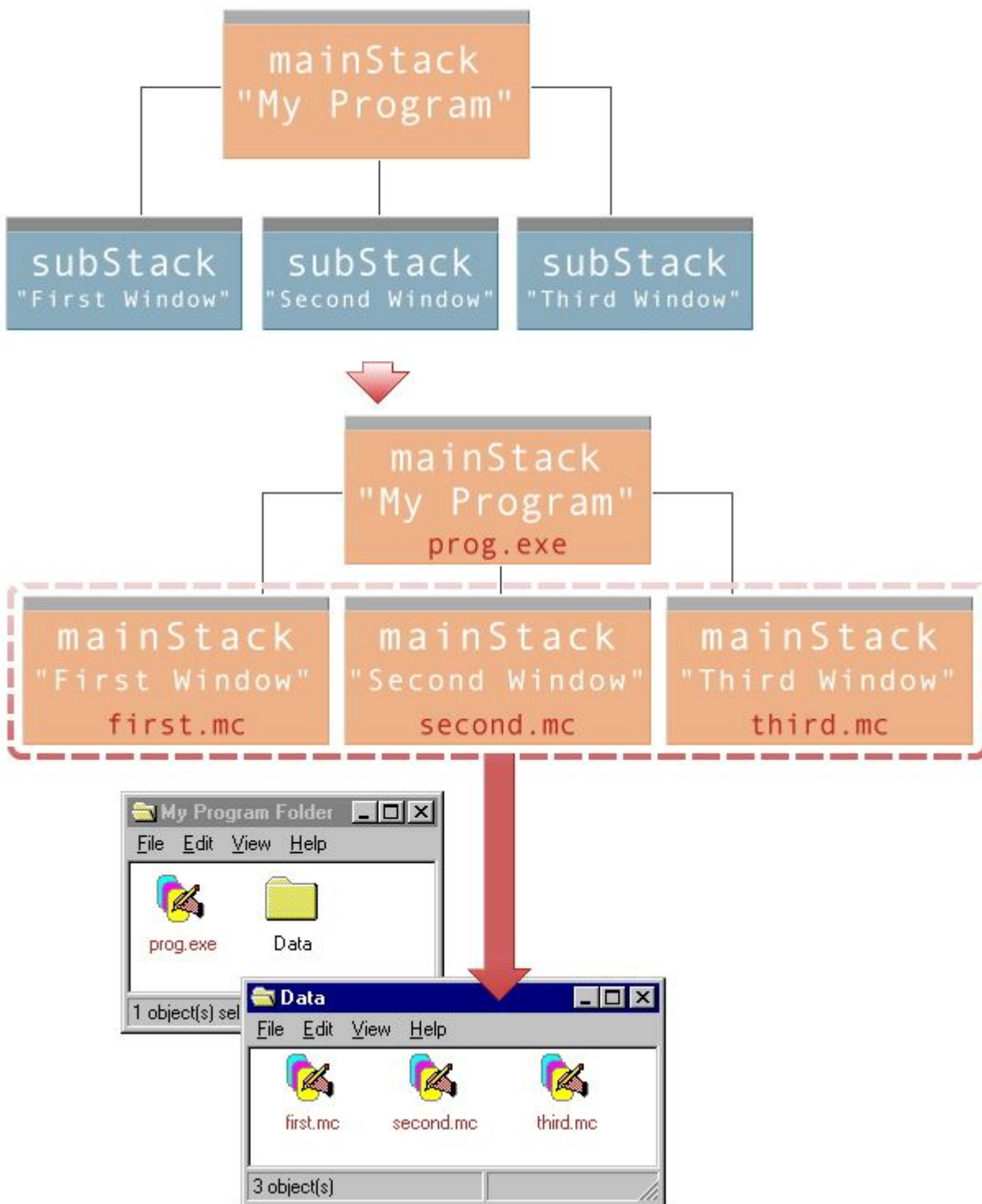
Is your stack a bloated, heavyweight RAM guzzler? Want to save changes to individual subStacks in your standalone on a users machine? Can't stand the inconvenience of removing images from your stack and referencing them by fileName prior to delivery? Split your single mainStack/subStack combination into multiple files (each of which can be loaded and unloaded from memory individually), and use `stackFiles` to prevent scripts that refer to these subStacks being broken.

stackFiles is a stack property that is used to reference separate stacks stored in separate files by name, just as if they were subStacks.

Here's how to utilise the power of `stackFiles`.

► Split your large stack up into multiple files, making the subStacks into mainStacks (though you can leave little stacks like ask and answer as subStacks of the mainStack if you like). To do that, open each stack and choose "Save As..." from the File menu. Note that Save As... will remove a subStack from its original mainStack and save it into a file (it does not leave a copy in the original mainStack).

Create a new folder called "Data" or something that equally conveys "*don't go here*" to the end user. Move all the subStacks that have newly been turned into files (of single mainStacks) into the new folder. This folder should be in the same folder as the standalone, but more about that later.



- ▶ Toplevel your *original* mainStack, called "My Program" in the diagram. This should be the stack that actually gets built into the standalone, i.e. the one that loads from the desktop when you open up. (A good use for a stack like this is to display a splash screen or welcome notice.)
- ▶ Select Stack Properties from the Edit menu, then click the Stack Files... button. Make sure the menu is showing the name of the current mainStack selected.
- ▶ Type the stack names of each of the subStacks you just saved into separate files, one per line. Those are the actual names you have for each of these stacks, the ones you use on the stacks and in your scripts *not* the file path for each.
- ▶ Next, go back up the list and add a comma to the end of each line.
- ▶ Now, go through each line and add the file path to the file that each stack is now stored in.

For example:

```
C:/Program
Files/MetaCard/Program/MyApp/Data/myfirstwin
dow.mc
```


However, a complete path like that is actually *not* a good idea. That's because when you install the application, you're not likely to be installing it in that directory. Your install directory will probably read something like..

```
C:/Program Files/MyApp/Data/myfirstwindow.mc
```

Don't try to guess this complete path, as the installation directory might be changed by the user. For example, the user might be instal onto drive "D" or run your program directly from the CD...

The solution to this problem is to use *relative paths*. Relative paths start at the current directory and move up from there. *The current directory is set on start up by MetaCard to the directory the MetaCard application is running from.* When you load a stack as a non-standalone, the directory will be set to the directory that the MetaCard application is in. When you load as a standalone, it will be the directory to the actual standalone application file.

Let's assume we're working with a standalone for the moment.

Article	Difficulty Rating	
4	Groups and Backgrounds	

► An introduction to Groups

A group is a type of object that contains other controls. It can contain any controls - buttons, fields, graphics, scrollbars, and eps objects, in any combination and number. As a result, groups are powerful and flexible objects. Here are some examples of what you can do with them:

- Hide and show multiple objects with a single command
- Move multiple objects with a single command
- Creating "panes" in a window to allow scrolling sets of objects
- Display the same object or set of objects on multiple cards (it is possible to group a single object)
- Create a common script that can be used by multiple objects
- Automatically track hiliting in a set of radio buttons
- Create tabbed menus
- Place menus into the Mac system menubar

When you group objects, they get placed in another object, which comes complete with selection handles, the ability to show a border, an option to show a scrollbar etc. Just like other objects you can assign groups scripts and custom properties, send messages to them, move them etc. To edit the properties of a group, double click it with the pointer tool like any other object.

However, groups are really a unique cross between a card and an object. Like cards, they are attached to the stack. Unlike cards, they can be smaller than the window frame, can be moved around as objects on cards, can scroll and do most of the things an object can do. Like a card, their scripts get inserted into the message hierarchy, just above all the objects inside them, but before the card.

► Groups and Backgrounds, why and whats the difference?

Once created, a group can be displayed on any card in the current stack. Try it now: create a new stack with some objects and group them (select them and choose "Group" from the Edit menu), then choose "Backgrounds..." from the "Edit" menu. Select the group you have created and press "Remove". The group will disappear from the card (move the dialog out of the way if you can't see the card). You will note that the group has appeared in the lower half of the box. Select it, and press "Place". It will reappear on the current card. Now, create a few more cards and go through each one. You will be able to place and remove the group on any card in that

stack using the Backgrounds dialog. Note you can only place a group once on any one card. That is why the group disappears from the bottom half of the dialog when you place it: the group is still available for placement on other cards, but can't be placed again on the current card.

As you will already have noted, in MetaCard there are two names for groups: *groups* and *backgrounds*. Both are used in the Metatalk language. Confusing? The two terms are useful. **Group** refers to group objects counted relative to a card, **background** refers to group objects relative to the stack.

Groups are displayed as objects on the *card*. Asking for the number of groups will give you the total number of groups on the current card.

Backgrounds refers to the total number of groups in the stack. Asking for the number of backgrounds will return the total number of unique groups in the stack, regardless of whether or not they are on the current (or any other) card.

Here are two examples. Firstly, the `backgroundNames` property returns a list of all the groups anywhere in the stack, whereas the `groupNames` returns a list of all the groups that are on a particular card. Secondly, referring to backgrounds and groups by number works differently. Referring to backgrounds by number works in *order of creation* of any group anywhere in the stack, but referring to groups by number works by using the *layer* of the group on a *card*. Thus `background 1` is the first group created anywhere in the stack, whereas `group 1` is the group with the lowest *layer* on the card.

You need the two terms when using scripts to place groups on a card. To determine which group to place, you need to be able to count it relative to all the other groups in the stack.

► Manipulating Groups

When creating groups, it is important to remember that the `layer` of controls in each group is determined by their order of selection before they are grouped, so if you want to make it possible for the user to tab between controls in a group simply select them in the appropriate order. Of course, you can change this order at any point by editing the `layer` of the individual objects within the group. Remember that the `layer` of grouped objects only refers to their layer within the group, not in relation to other objects in the card. However, the number of a grouped object refers to its number relative to all the objects on the card.

TIP: The function `the selectedobject` returns the name of the currently selected object. If multiple objects are selected that are not grouped it returns the names of all the selected objects with one on each line, in order of selection.

To edit a group, open its properties palette, click the Properties tab, and press Edit. You will go into edit background mode, and only the contents of the group will be displayed - all other objects will disappear. To stop editing a group, choose "Stop

editing BG" from the edit menu.

To do this by script:

```
start editing group 1
start editing background "example"
stop editing this bg
```

To create a group and objects in a group by script:

```
create group "My Group" -- creates a new group called "My
Group"
create button "My Button" in group "My Group" -- creates
a button
create fld "My Field" in group "My Group"
```

If you have a nested group (a group within a group) , it will not appear in the Edit Background dialogue. Edit the first level group and then select and edit the nested group.

You can also add or remove groups to cards by script:

```
place group "My Group" onto card 1
place background "My Group" onto card 2
remove background "My Group" from card 1
```

Remember that a group is one object. So be wary about deleting groups by selecting and pressing the delete key because this permanently deletes them from every card in the stack.

TIP: Setting the `selectGroupedControls` global property to true

```
set the selectGroupedControls to true
```

allows you select individual controls within a group without having to ungroup or go into Edit Background mode. (This requires MetaCard 2.2.1B1 or above.)

Individual controls within a group can be deleted by script, just as if they were separate objects.

One particularly useful function of groups is to automatically make radio buttons hilite correctly. If you create some radio buttons and group them, they will automatically enforce radio button behavior without any scripts. It is possible to navigate between radio buttons in the same group using the arrow keys on Windows and UNIX. To do this set the `tabGroupBehavior` of the group to true. This will

also makes the tab key skip over the whole group.

▶ **Tabbed menus**

Groups are important for creating tab menus and menu bars. To create a tabbed menu create a button and using the Button Properties dialogue, set the style to tabbed. Set the menu contents of the button to the name of the different tabs you require, and create groups named the same as each tab. It is easy to hide and show the groups corresponding to each tab. Place the following script in the tabbed button:

```
on menuPick newTab, oldTab
  hide group oldTab
  show group newTab
end menuPick
```


In the above example, the parameters passed with the menuPick message (containing the name of the tab clicked on and the name of the tab that was previously in front) are used to hide and show groups with the same name as the tabs.

▶ **Displaying text on multiple cards**

When placing a group on multiple cards, you will often find you want to fields such as labels to display the same text on each card in the stack, while others fields (usually for user entry) should be editable and display different text on each card. Fields which display the same text on each card have their sharedText property set to true, and those that display different text have their sharedText property set to false.

Similarly, if you want to display the same state of hiliting in buttons and checkboxes on each card, set their sharedHilite property to true. (You will find that if you create labels using the Show as Label button in the Button Properties dialogue their sharedText property is set to true, but the default setting for fields is false.)

You can sort cards by the contents of their fields using the sort command. Sorting can be alphabetical or numeric, and ascending or descending. The sort command can be complimented with the identically named mark command and mark property. For example if you want only to sort certain cards, they can be marked and you can specify that only the marked cards are sorted.

Article	Difficulty Rating	
6	Text Management	

There are many features available in MetaCard for doing complex matches on text. From the simple `find` and `replace` search routines to the extraction of key entries from complex formatted text, MetaCard has the features you need. This article examines the `matchText`, `matchChunk` and `replaceText` functions. It also discusses splitting a large file into chunks suitable for processing, using an array. For information about the more basic search and replace features see the article on [finding and replacing text](#).

► The `matchText` Function

The `matchText` function can be used for advanced text matching. Its has two main areas of functionality. First, it validates whether text contains a given pattern, and second, it can return certain patterns within a that string to specific variables if it makes a match. If the text passed to it is a multiple-line container, the validation applies to the entire container. For output to variables, however, it applies only to one occurrence of the match, so if you want to repeatedly extract the same pattern, you need to apply the `matchText` function for each line. The syntax is as follows:

```
matchText(<source>, <regularExpression>[ ,  
<output variable 1>,  
<output variable 2>...])
```

A basic example of validation..

```
matchText("hello there", "hello")
```

This will return true since the string "hello" is a substring of "hello there".

```
matchText(field "My Field", "hello")
```

Similarly, if there is a field called "My Field" containing the string "hello", this will return true.

But `matchText` is not really intended to perform such basic matching. *To give the exact characteristics of the text match <regularExpression> can be formed from any combination of special characters, each of which has a specific function.*

► The Special Characters for use with the `matchChunk`, `matchText`, and `replaceText` functions.

The following table shows all the special characters that can be

Special Character	Description
(exp)	matches the expression, and puts result in a variable*
. (<i>a period</i>)	matches any character
^	forces match to be at beginning of string
\$	forces match to be at end of string
[chars]	matches any of the characters in the set of chars. The characters can be either characters allowed to match, or if ^ is the first character in chars, not allowed to match. You can specify a range of characters by putting a - between them. For example [a-z] matches any lower case alphabetic character.
*	matches zero or more of the preceding special character
+	matches one or more of the preceding special character
?	matches zero or more of the same characters matched by the previous special character
regEx 1 regEx 2	matches either regular expression

*It is important to remember that everything enclosed within parentheses in this expression denotes an output to a variable. The first set of parentheses is output to the first variable <output 1> and the second set of parentheses is output to the second variable <output 2> and so on.

Example 1: matchText with Email Addresses

IMPORTANT: Unlike commands such as the `put` command, the `matchText` function does not automatically declare variables. Therefore, remember to declare the variables prior to usage.


```

local actualName, addressName, ispName,
classification
put "From: Joe Bloggs <jbloggs@someISP.com>"
into source
put matchText(source, "^From: (.*) <(.)@
([^\.]+)\.(>)", \
actualName, addressName, ispName,
classification)

```

The <regularExpression> component in the above pattern is broken up in the table below.

Component of String	Match in Source	Description of Special Chars
^ From:	From:	the ^ character forces the string "From:" to match the beginning of the <source> expression
(.*)	Joe Bloggs	the () enclose an output to the variable <i>actualName</i> the . (period) character matches any character and the * character allows zero or more unspecified subsequent characters before the next specified character (in this case < is next specified)
<	<	a literal match of the < character

(.+)	jbloggs	<p>an output to the variable <i>addressName</i>, in this case matching if there are any characters before the @ symbol</p> <p>Note the use of + instead of *. The + requires that at least one character be present. E.g. <code>matchText("", ".+")</code> returns false, but <code>matchText("", ".*")</code> returns true. This example of <code>matchText</code> will match a string with no name before the actual email address, but requires the email address to be present.</p>
@	@	a literal match
([^\.]*)	someISP	<p>the [] encloses a character match</p> <p>the ^ specifies that the string must <i>not</i> match the period</p> <p>the \ specifies that the . is escaped since it is a special character but a literal match is wanted</p> <p>the () specify the value is to be output to the variable <i>ispName</i></p>
\.	.	a literal match of the period .
(.+)	com	an output to the variable <i>classification</i>
>	>	a literal match of the > character

Thus in the above example output to variables is as follows:

Variable	Contents of Variable
actualName	Joe Bloggs
addressName	jbloggs
ispName	someISP
classification	com

Example 2: matchText with Phone Numbers

```

local international, county, district
put "+44131 672 2909" & return & "0131 554 2961" into
source
  repeat for each line l in source
    -- matchText must be applied to each line
    if matchText(l, "(^\\+[0-9]+|[0-9]+) ([0-9]+) ([0-9]
+)", \
        international, county, district) then
      put "national code:" && international & return
after output
      put "county code:" && county & return after output
      put "district code:" && district & return & return
after output
    end if
  end repeat
answer output

```

Component of string	Match in string 1	Match in string 2	Description of special chars
(^\\+[0-9]+	+44131		^\\+ literally matches the + character at the start of the string - the ^ forces the match to be at the start of the string, the \\ escapes the + character [0-9]+ matches any numeric characters
			allows a match of the string either before or after it; whichever matches is put into the variable <i>international</i>

[0-9]+)		0131	matches any numeric characters.
([0-9]+)	672	554	matches any numeric characters the matching string is put into the variable <i>county</i>
([0-9]+)	2909	2961	matches any numeric characters the matching string is put into the variable <i>district</i>

TIP: To match the " (quote) character use the quote constant. To make an actual match with one of the special characters, you must first escape that character by inserting the \ character before it.

► The matchChunk Function

```
matchText(<source>, <regularExpression>[,  
<output1FirstChar>,  
<output1LastChar>, output2FirstChar,  
output2LastChar...])
```

The matchChunk function works in a similar fashion to the matchText function. The same special characters work, and an output is made to variables, but the matchChunk function outputs a chunk expression to describe the matched text, rather than the actual text matched. Output occurs to pairs of variables. The first of the pair is the number of the first character in the matching string, the second is the number of the last character in the matching string. Thus in equivalent matchChunk and matchText strings the matchChunk has twice the number of output variables.

Example 3: matchChunk with Email Addresses

If Example 1 is modified to use matchChunk instead of matchText we have the following:

```
local actualNameFirst, actualNameLast, addressNameFirst,\  
addressNameLast, ispNameFirst, ispNameLast,  
classificationFirst,\  
classificationLast
```

```

put "From: Joe Bloggs <jbloggs@someISP.com>" into source
put matchChunk(source, "^From: (.*) <(.*?)@([^\.]*)\.(.+)
>", \
actualNameFirst, actualNameLast, addressNameFirst, \
addressNameLast, ispNameFirst, ispNameLast, \
classificationFirst, \
classificationLast)

```

Now the output to variables is as follows:

String	Matching Text	Variable Pair	Contents of Variable
(.*)	Joe Bloggs	actualNameFirst actualNameLast	7 16
(.+)	jbloggs	addressNameFirst addressNameLast	19 25
([^\.]*)	someISP	ispNameFirst ispNameLast	27 33
(.+)	com	classificationFirst classificationLast	35 37

► The `replaceText` Function

```

replaceText(<source>, <regularExpression>,
<replacement>)

```

The `replaceText` function can be used to replace all occurrences of a text string in a source. All the same special characters as in `matchText` and `matchChunk` can be used in the `regularExpression`, and the same rules apply for escaping special characters. However, for most uses, the `replace` command is better because of its speed and simplicity.

► Splitting up large amounts of data to perform text matching operations

Say you're importing a database, or some other large file with a lot of data in it. What do you do with it when you've read it in? In most cases, the first thing to do before processing anything is to split it up into manageable chunks, rather than have it all sitting in one large variable. If you want to do any chunk expressions or text matching on the data, e.g. searching for particular strings using `lineOffset()`,

you'll find it inefficient to work with large quantities of data in one variable. If you're using `lineOffset()` to pick out all the lines with a particular string, for example, you have to be aware that it **always** starts searching from the start of the container. Even if you specify a number of lines to skip, whilst these lines aren't searched, they still have to be read through, so this does not lead to performance improvements. By the time you've got a reasonable way down the file, you'll be unnecessarily reading through large amounts of data at the start of the file each time you pick out another line.

Typically, you'll want to use an array to split the data up into neat chunks, each of which can be accessed as an independent unit. The time it takes to split the data up like this is usually insignificant compared to the time savings made on processing the data afterwards.

Lets say that we're dealing with a return delimited file, where each database entry begins with a line containing the word "start" and a name or identifying string for each entry:

```
start myDataBaseRecord1
record 1 contents...
record 1 contents...
more lines of contents...
start myDataBaseRecord2
record 2 contents
etc...
```

The following script will split up the data, waiting until the word "start" appears in a line then placing the text (up until the next time it meets the word start) into a separate array element:

```
repeat for each line l in tFileContents
  if word 1 of l is "START" then
    put l into currentobj
  else
    put l&cr after gBigArray[currentobj]
  end if
end repeat
```

The first line tells MetaCard to cycle through each line in the field contents, putting each line into the variable l. An alternative way of doing this would have been to use:

```
repeat with i = 1 to the num of lines in tFileContents
```

However, the latter method should only be when you **absolutely** need to know the

line number of each line for some reason. Using the "repeat for each line..." construct, as in the first example, is always *very* considerably faster.

The second line checks if the line starts with the word "start". You can use anything here, e.g. a delimiter you expect to find in the text, or even a complex match on the line (as described earlier in this article). If the data you're picking out is fairly simple and doesn't require more than simple processing (e.g. it just gets split up into fields), you may want to do that here instead of split the data up into an array for further processing at all.

The third line contains the name of the array element currently in use. This gets updated every time the word "start" is encountered. Because arrays in MetaCard can be indexed with strings, there is no need to condense this line into anything - the entire line can be used to reference the array element.

The fifth line places the current line in the variable `l` into the element of the array currently in use (named in `currentObj`).

The result of using that repeat loop on the small set of example data above would be an array with two elements.

Element Name	Contents
start	record
myDataBaseRecord	contents...
1	record
	contents...
	more lines of
start	contents
myDataBaseRecord	record 2
2	contents
	etc...

To process the data now that its been split up, we need a list of all the elements in the array:

```
put keys(gBigArray) into tListOfElements
```

The variable `tListOfElements` now contains a complete list of each record in the array. Processing that is a simple matter of cycling through each line and passing the array element to a function that does the processing. For example:

```
repeat for each line l in tListOfElements
  doMyDataProcessingRoutine gBigArray[l]
end repeat
```

The end result? To `doMyDataProcessingRoutine` would be sent each chunk of the original file separated between lines containing the word "start". Because you're now


dealing with a smaller chunk of data, its efficient to do any intensive processing routines on each record: e.g. chunk expressions or regular expression matching (REGEX) to draw graphs, display information, build objects, or whatever you like...

If you've been following along closely this far, you'll have spotted one problem. The elements passed to the doMyDataProcessingRoutine won't be passed in the order they were in, in the original file. Why? The keys() function doesn't return the elements in the order they were created in the array. If you need to process the file in order, then alter the original script to:

```
put 1 into tCounter
  repeat for each line l in
tFileContents
  if word 1 of l is "START"
then
  put l && tCounter into
currentobj
  add 1 to tCounter
  else
  put l&cr after gBigArray
[currentobj]
  end if
end repeat
```

The variable tCounter increments each time a new element is created, and appends the current element number as a separate word to the end of the element name. When getting the list of array elements, you need to sort them:

```
put keys(gBigArray) into tListOfElements
sort lines of tListOfElements numeric by last word of
each
```

Article	Difficulty Rating	
5	Finding and Replacing Text	

Metacard offers flexible and effective features to search and replace text. There are various methods to go about this, from relatively straightforward commands like `find` and `replace`, to more advanced functions like `matchText()` and `replaceText()`. This article focuses on the more straightforward searching features. For more information about advanced features see the article on [text management](#).

► The Find Command

One way to search for a text string is to use the `find` command. It is what the MetaCard Find tool uses, and has a number of advantages. Firstly, it can be used to search multiple fields in the `defaultStack`, as well as specific fields.

```
find "Joe" -- searches all the fields in the stack card
by card
find "Joe" in field "Name" -- searches the "Name" field
in all the cards
```

Secondly, you can control characteristics of the search by specifying the `<type>`, in the form:

```
find <type> <expression>
```

`<Type>` can be `chars`, `string`, `whole`, or `word`. Using `chars` finds exact character matches anywhere in words, but does not match text expressions longer than a single word. `String` finds exact character matches anywhere in words, and includes spaces and multiple words. `Whole` only matches the entire expression if it is complete words, and `word` matches whole words but only matches the first word in the expression.

The following table gives examples of `find` commands and the text which is matched using different search characteristics. The first line in the table is equivalent to typing:

```
find chars "man"
```

Examples of the Find Command, operating in a field containing the words "The yellow man".

Type	Search	the result	the foundText
chars	man	empty	man

string	man	empty	man
whole	man	empty	man
word	man	empty	man
chars	e yellow	empty	e
string	e yellow	empty	e yellow
whole	e yellow	not found	
word	e yellow	not found	
chars	yellow man	empty	yellow
string	yellow man	empty	yellow man
whole	yellow man	empty	yellow man
word	yellow man	empty	yellow
chars	man yellow	empty	man
string	man yellow	not found	
whole	man yellow	not found	
word	man yellow	empty	man

There are a number of functions which give information about the text which has been found with the `find` command. The `foundChunk` returns a chunk expression describing where the text was found, e.g. char 2 to 3 of field 1.

The `foundText` (used in the field above) contains the actual text found. The `foundLine` returns the number of the line the text was on. The `foundField` returns the field name. The `foundLoc` returns the coordinates of the border drawn around the text that was found.

If you do not want certain fields such as labels to be included in a general search, set their `dontSearch` property to true. If you want to discriminate capital letters and lowercase letters set the `caseSensitive` local property to true. This property also applies to other methods of searching besides the `find` command, such as the `offset` functions.

TIP: If you want to use the `find` command, but you want the text which is found to be selected as a normal text selection, rather than the standard black border, use the following script:

```
lock screen -- avoids any screen flicker
find <expression>
select the foundChunk
unlock screen
```

► The Offset Function

Despite the flexibility of the `find` command, it is not always the best feature for searching.

The `offset` function is a simple alternative to `find`. It is generally faster than the `find` command, and has the advantage that it can be used in other containers besides fields, such as variables. It also does not display any results, leaving you free to generate user feedback to the search in any way you like.

`Offset` returns the character number of a specified string within a whole string.

```
put offset("there", "hello there")
-- returns 7 since the t in "there" is character 7 in
"hello there"
```

If the specified string is not within the whole string 0 is returned.

To use the `offset` function like the `find` command, try something similar to the following script:

```
ask "Enter the text to find?"

if the result is not "Cancel" then \
select char offset(it, field 1) \
to offset(it, field 1) + length(it) - 1 of field 1

-- select from the first character to the last character
to be found
-- the length function returns the number of characters
in a string
-- the \ character is simply used to separate long lines
```

The `offset` functions returns character offsets, but there are a set of other similar functions that work in relation to different chunks: `itemOffset`, `wordOffset`, `lineOffset`. A useful property to remember when using these is the `wholeMatches` property. When set to `true` it forces all matches made to be complete chunks, i.e. complete items, words, or lines. When `false`, it allows partial matches.

► Useful Operators

There are various operators which can be useful during text searches. The `contains`, `is in` and `is among` operators can be used to make text comparisons. Of these, only `is among` merits further explanation. It will only return `true` when a complete match is made.

```
if field 1 contains "hello" then beep
if "ello" is in "hello" then beep
if "hello" is among the items of fld 1 then beep
if "This complete line." is among the lines of fld
"example" then beep
```


► The Replace Command

The `replace` command can be used to replace text strings in containers. It replaces all instances of a text string in the specified container. For more complex replacements, see the article on text management. The syntax is as follows:

```
replace <text> with <replacement text> in <container>
```

For example:

```
replace "his" with "her" in field 1
replace "this" with "that" in MyVariable
```

Article	Difficulty Rating	
3	Using Files With MetaCard	

For file access, MetaCard uses the URL system. In keeping with this, all of MetaCard's file access commands use a "/" character as the directory delimitator, regardless of platform. To read a file from somewhere on your computer, you simply:

```
put url "file:c:/test.txt" into theVariable
put url "file:/Macintosh HD/Files/example" into
theVariable
put line 3 of url "file:c:/example.txt" into theVariable
put "A line of text" into url "file:c:/examplewrite.txt"
```

```
answer file "Select a document to open:"
put it into tPath
put url ("file:"&tPath) into theVariable
```

```
ask file "Save this document as:"
put it into tPath
put field "text to save" into url ("file:"&tPath)
```

Note the use of the HyperCard compatible ask and answer file commands. Further information on these commands is available in the MetaTalk reference stack.

All of the above examples open files in text mode. If you want to access a binary file:

```
put url "binfile:c:/test.txt" into theVariable
```

To access something from the internet, it changes slightly to the familiar:

```
put url "http://www.mysite.com/" into theVariable
```

That command would download the HTML page at the site "www.mysite.com" and place it into theVariable.

If you want a file on the internet to be downloaded asynchronously, you can use the load command instead. This downloads the file and places it in a cache. The cache can contain multiple files, and can be read multiple times. It is important to make sure that you empty the cache whenever you are done using something in it, otherwise it will consume a lot of memory. The urlStatus () function can be used in conjunction with the load command to give a feedback bar which shows a

file downloading. An example of this can be found in the MetaCard tools stack. To access the script, open the MetaCard Menu Bar as normal, then type:

```
edit script of stack "Download stack"
```

into the message box. To see it in action, choose either "tools.metacard.com" or "help.metacard.com" from the appropriate menu on the MetaCard menu bar.

Tip: It is possible to download images from the web then import them into MetaCard for display. Simply "put" the image data you downloaded into an image object.

As ever, copying this stack and scripts for use in your own stacks is probably the best way to get started.

You can download any type of document from the web. The most common use for this feature is to load MetaCard stacks that are served somewhere on the internet.

You can put things into files by using URL too:

```
put "some text" into word 3 of line 2 of url  
"file:c/test.txt"
```

► Other File Functions

URL is the normal way of accessing files in MetaCard. Traditional (HyperCard compatible) file access functions are also supported. In almost all cases, you will find the URL function faster and more convenient to use.

Here are the basic set of HyperCard compatible file functions:

```
open file "c:/test.abc" for text read  
read from file "c:/test.abc" until eof --eof stands for  
"end of file"  
put it into tFileContentsVariable  
close file "c:/test.abc"
```

Of course, you can also open files for writing:

Tip: Make sure you understand all of these file modes, and pick the right one each time. Its tempting to omit the parameters (so both allowing read and write of any file you open), but this is much less efficient then specifying exactly what you want to do. Also, failing to specify binary when you want to work with binary data will not give the desired results.

```
open file "c:/test.abc" for write
```

The append mode always adds to the file at the end, not the start, which avoids overwriting the contents:

```
open file "c:/test.abc" for append
```

Both reading and writing:

```
open file "c:/test.abc" for update
```

Finally, you can do all of the above in binary mode, e.g.:

```
open file "c:/test.abc" for binary read  
open file "c:/test.abc" for binary append
```

Once a file has been opened for writing, appending or updating, you can of course write to it:

```
write theVariable to file theFilePath
```

Finally, an often forgotten command is seek:

```
seek to 200 in file theFilePath  
read from file filePath for 5 lines
```

In the above example, the first 200 characters in the file are skipped, then the next five lines read. The advantages of doing this rather than reading the file is that its faster, and doesn't take up memory to store the contents of the file. Its a bit like moving an invisible insertion point marker. You can also seek *backwards* from the end of a file, in exactly the same way you use any chunk expression backwards. For example, to seek to the character 50 characters before the end of a file:

```
seek to -50 in file theFilePath
```

Always remember to close any file you've finished working with. This will make the file available to other applications, and free up memory.

► **Managing files: copying, deleting, and searching, compressing**

MetaCard uses only a handful of commands for managing files. However, when you put them together, you can do quite a few things. For example, to copy a file you need five lines:

```
answer file "Select a file to copy:"
put it into tOriginalFilePath
ask file "Save the copy as:"
put URL ("binfile:"&tOriginalFilePath) into URL
("binfile:"&tNewFilePath)
```

To compress a file:

```
answer file "Select a file to
compress:"
put it into tOriginalFilePath
put it & ".gz" into tNewFilePath --add a .gz extension
put compress(URL ("binfile:"&tOriginalFilePath)) into URL
("binfile:"&tNewFilePath)
```

To decompression is similar:

```
answer file "Select a file to decompress:"
put it into tOriginalFilePath
put it into tNewFilePath
if char -3 to -1 of tNewFilePath is ".gz" then delete
char -3 to -1 of tNewFilePath
put decompress(URL ("binfile:"&tOriginalFilePath)) into
URL ("binfile:"&tNewFilePath)
```


On the Macintosh platform, its a little more complex. Mac files contain two forks: the data fork and the resource. In this case, you must use the `getResource()`, `copyResource()` and the URL type "resfile:" to copy the resources in the first file to the second.

Tip:

You can find out if a particular file exists using the syntax:

```
if there is a file someFile
then...
```

You can rename a file using the `rename` command.

```
rename file "C:/example.txt" to "example2.txt"
```

To delete a file use:

```
delete file theFilePath
```

Be careful with this command!

You can use `MetaCard` to read in lists of files in directories, or search through directories for files on disk. The principle is simple. Use the `directory` function to change to the directory you want to query, e.g.:

```
set the directory to "C:/"
```

Then get a list of the files in that directory using the `files` function:

```
put the files into theListOfFilesOnDriveC
```

Now, get the all the sub-directories:

```
put the directories into tDirectoriesList
```

► Using files with external applications

To open a file with another application, e.g. to view an Acrobat document in Acrobat, you use the `shell` function:

```
get shell(quote & tAcrobatReaderPath & quote && quote &
```

```
tAcrobatDocumentPath & quote)
```

On MacOS systems you use the launch command:

```
launch "/Macintosh HD/My Application"
```

The use of quotes is required or the function will fail on paths that contain spaces. On Windows 95, you'll need to set the `hideConsoleWindows` global property to true, to avoid seeing the "MS-DOS" prompt screen that runs when you use `shell()`.

You can also use the `open process` command to load applications in a similar way.

You can use the `shell()` function to run any MS-DOS command on Windows based systems, or a command line instruction on UNIX based systems.

Tip:


To find the path to an application (such as Acrobat Reader, above) on Windows 95, use the `queryRegistry` function. Acrobat can be found with:

```
put word 1 to -2 of queryRegistry  
("HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AcroExch.Document\  
shell\open\command") into gAcrobatPath
```

Why word 1 to -2? The registry entry is stored with position of the allowable parameters at the end. Taking word 1 to -2 takes the first word up to the word second from the end, removing those parameters.

Another useful one is being able to get the default web browser on Windows systems:

```
put word 1 to -2 of queryRegistry  
("hkey_local_machine\software\classes\http\shell\open\  
command") into gWebBrowserPath
```

Article	Difficulty Rating	
1	Doing Menus in MetaCard	

Menus are really very simple to do in MetaCard. However, they often prove to be one of the more difficult concepts to get to grips with if you are just starting out. The important thing to understand is that menus in MetaCard are really just buttons with a few fancy properties set. This even applies to the MacOS version of MetaCard - editing the menu bar is just a case of moving around buttons on the stack to which the menus are "attached". Even if you are an old hand with menus, you'll want to check out our scrolling menu stack, which is available for download (see the bottom of this document).

MetaCard supports two different methods of doing menus. The first method is known as a button contents menu. Use it to construct simple menus which use only the normal set of options - radio buttons, check boxes, dividers and enabled or disabled items and hierarchical (or pull right) menus. The second, the menu panel, is used for doing more complex menus that support any kind of scripting or control. Use these to make pop up palettes, menus with graphics, images, icons, or any other complex type of menu.

► Button Contents Menu

We'll start off by looking at button contents menus. Button contents menus are buttons, with their text property set to contain a list of items to pop up. Use the properties palette to create a button, then set its style to pulldown. Then, put the name of each menu item on a separate line in the menu contents field. Try out that button: clicking on it will cause the menu to open. If you want to dynamically alter the contents of the menu before display, use a mouseDown handler in the button to set the `text` property of the button to whatever you want. Use this technique to add, delete, check, uncheck, enable or disable items.

To make an item...	Do the following...
a dividing line	use "-" as the name of the item.
disabled	place a "(" at the start of the item name.

a hilited checkbox	place a "!c" at the start of the item name.
an unhilited checkbox	use "!n" at the start of the item name.
a hilited radio button	place a "!r" at the start of the item name.
an unhilited radio button	place a "!u" at the start of the item name.

Example:

Item 1
 (Item 2
 -
 !cFourth item
 Fifth item
 -
 Seventh Item
 Eight item
 (!rItem nine



Button contents menus on Windows and the MacOS.

MetaCard can use HyperCard compatible syntax for menu operations. Here's a sample script to disable an item (in this case, item 5):

```
on mouseDown
  disable menuItem 5 of me
end mouseDown
```

And a few more HyperCard compatible examples:

```
disable menu "Style"
enable menuItem 3 of menu "Style"
hilite menuItem 1 of menu "Font"
```

MetaCard's native way of handling menus is actually a little harder to use. The following script does the same as the first example above, disabling menu item 5:

```
on mouseDown
  get the text of me
  if char 1 of line 5 of it is not "(" then put "("
  before line 5 of it
  set the text of me to it
end mouseDown
```

You may need to use this method to control some MetaCard specific features, such as placing checkboxes before menu items.

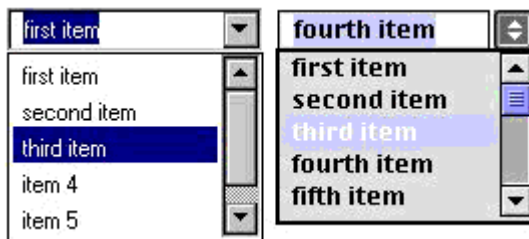
When an item gets chosen, it sends a "menuPick" message to the button. The menuPick message is sent with the name of the item chosen. Here is an example script:

```
on menuPick pWhichItemPicked
  switch pWhichItemPicked
    case "Item 1"
      --do something to handle the first item picked
      break
    case "Second item"
      --do something to handle the second item getting
picked
      break
    ... etc...
  end switch
end menuPick
```

Tip: If you prefer to work with item numbers instead of names, use the menuHistory property. It contains the number of the item last chosen, instead of the name. Use a switch statement:

```
switch the menuHistory of me
```

Button contents menus can also be used to do option style menus, and comboBox style menus. Use an option menu to create a list of items. Use a comboBox to create a scrolling list of items where the user has the option to type the name of the item in instead of choose it from a list. Use the menuLines property to set how many lines in a comboBox are visible at one time. Note that disabling items, checkboxes, dividers, etc. are not supported in this type of menu.



ComboBoxes on Windows and the MacOS. The menuLines has been set to 5.

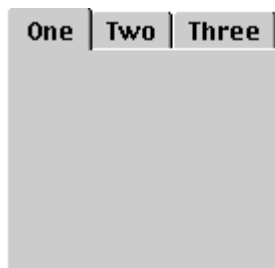
When an item is chosen, the label property of the button is automatically set to the name of the item chosen. In option menus, when you re-open the menu, it will be positioned with the item most recently chosen over the mouse. You can get and set this using the menuHistory property of the button.



Option menus on Windows and the MacOS.

The final type of button contents menu is the tabbed button. This type of menu can be used to display a tabbed interface, often used for doing program Preferences screens. The tab items are set as contents in the normal way. Like option menus and comboBoxes, using dividers, disabling items and other such features are not supported.

When a tab is chosen, a menuPick message is sent to the button. In this case it is sent with two parameters, the name of the tab chosen, and the name of the tab previously chosen. The most common use for this is to hide and show a group of controls.



If you want to set which tab is currently active, you can set the menuHistory property of the button to the number of the tab, e.g.:

```
set the menuHistory of btn
"tabbed
example" to 2
```

Note that setting the menuHistory causes a menuPick message to be sent to the button, just as if one of the tabs had been pressed with the mouse.

▶ Menu panel menus

Menu Panels are very different from button contents menus. They are actual MetaCard stacks, as opposed to a property attached to a single button. This means you can open them in multiple places throughout an application without having to duplicate anything. It also means that you have to draw out the panel yourself, and you can place any standard MetaCard control in it if you wish.

To create a menu panel, it is simplest to clone the existing menu panel structure found in MetaCard. Type:

```
clone stack "MC SelectedObject Menu"
```

to create an editable copy of the MetaCard object selection menu. You'll find that all the buttons in the stack have mouseUp scripts, instead of menuPick handlers. You can use mouseUp messages attached to each button in the menu, or use a single menuPick handler like in button contents. In addition, normal messages are sent to controls in a panel stack. This means you can use the messages mouseEnter, mouseLeave, etc. to provide additional functionality if you wish.

To display a menu panel, create a button on the stack you want to display it. Set the style to either pulldown or popup, and then set the menuName property to the name of the stack you've created. You can do all of these operations using the button properties palette. For example, you could create a button to pop up the MetaCard object selection menu in anywhere - simply create a new button, set the style to pulldown and the menuName to "MC SelectedObject Menu".

If you need to display a panel by script, rather than attached to a button, you can open the menu stack using a command in a mouseDown handler. For example:

```
popup "myStackMenu"
```

will open the stack "myStackMenu" as a popup menu under the position of the cursor.

Tip: If you only want your panel to open on a button when a particular mouse button is used, set the `menuMouseButton` property. For example, select a pulldown panel button and type:

```
set the menuMouseButton of the selobj to 3
```

The button will now only display the menu if you use the third mouse button. Note that the third mouse button is actually the second mouse button on Windows systems, and the mouse-Control key combination on the Mac.

► **Sub-menu (or "pull right" menus)**

If you want to create menus inside menus, you can use either contents menus with tabs inserted, or create the first menu as a menu panel. Note that sub-menus inside button contents menus are only supported in 2.2.1B1 or above, previous versions require you to use panel menus.

For button contents menus, place tabs before the menu items that you want to be displayed in a sub-menu. The number of tabs can be used to indicate the number of levels deep an item is.

For panel menus, the item where you want the second menu to be displayed should have its style set to cascade. You can either set the button contents of that button to form a button-contents sub-menu, or set the `menuName` property to display a panel inside the panel. The sub-menu will be displayed automatically when the mouse moves over the item. Note that you can't use the cascade style menu in a normal stack - you must use it inside a menu panel.

► **Menu Bars**

A menu bar is made from a group of buttons. To create a menu bar, create buttons corresponding to each menu and, using the Button Properties dialogue, set the style of the buttons to pulldown menu. If you're using Windows, set the text and font to the Windows standard (MS Sans Serif). If you're using the MacOS, you don't need to set the font - the system font will be used when MetaCard places your buttons into the MacOS standard menu bar at the top of the screen.

Set the menu contents of each button to the menu items you require. Arrange the buttons as a menu bar and group them. Name the group. To display a menu bar as the system menu bar on MacOS, use the Stack Properties dialog in the Edit menu to set the `menubar` property of the stack to the name of the group. Whenever the stack is active that group will be used for the system menu bar.

IMPORTANT: On MacOS when you set the `menubar` property of a stack to a group which exists in the same stack, the whole stack is scrolled so that the menu bar group and everything above it disappears from the window. The height of the stack is actually decreased by the height of the menu bar group and everything above it. This is a useful time saver for cross-platform projects because the same menu bar appears inside the window in Windows and UNIX, and on the system menubar in MacOS. However, if you want the group to remain in the stack window where it is editable on MacOS systems, set the `editMenus` property of the stack to true.

A good example of an existing menu bar you can take apart and examine is the MetaCard standard menu bar. The entire MetaCard interface is written in MetaCard. The menus you see while using MetaCard are a MetaCard group. The menubar of stack "Home" is set to the group "MetaCard Menu Bar". If you want to see this group type the following into the Message Box:

```
toplevel "MetaCard Menu Bar"-- stack name  
happens to be group name
```

Select the group named "MetaCard Menu Bar" inside this stack. These are the actual menus MetaCard displays in its editor. Be cautious as editing this group will alter the MetaCard interface .

▶ **Menu bars on MacOS systems**

The menu bar includes the MacOS standard Help menu with the Apple default items (usually Show Balloons, About Balloon Help). The righter-most menu button in the menubar group always operates as the Help menu (and is named Help, regardless of what you call it). The last item in the menu contents of the righter-most button is used as the About... menu item in the Apple menu. To capture a user selection of that item, place a `menuPick` message handler in the group containing the menu bar.

If you want to update the contents of the menu bar before displaying the menus, place a `mouseDown` message in the group containing the menus. Note that the `mouseDown` message is only sent to the group, not the individual menus. This is a limitation of the MacOS. Another limitation for the 2.2 release of MetaCard is that you can't use menu panels in the Mac menu bar. All Mac menu bar menus must be standard button contents menus (cascade and option styles are not supported either). If you want to do sub-menus, these must be done using the sub-menu button contents properties in the 2.2.1B1 release or above.

If any of the subStacks in your project don't have a menu bar, but you want one to be displayed set the `defaultMenuBar` global property to the one you want displayed. That menu bar will then be displayed for stacks without a menu bar.

Important Note:

Make sure that the buttons in any menu bar you create don't overlap. If they do, it may cause strange behavior - including multiple panels being displayed at once. If you're having problems with a menu bar, check to see that the buttons aren't overlapping.

► Scrolling Menus

One limitation of MetaCard is that you can't do scrolling menus, except as the comboBox type. There are plans to add scrolling menu support to the MetaCard engine. In the mean time, Cross Worlds has produced a free utility for creating menu panel menus, which includes full support for scrolling. Fully compatible with MetaCard on any platform, you can use it to create menus with greater ease than producing even button contents menus. The panels created can easily also be edited easily by script on the fly - just like button contents menus. This stack comes complete with support for doing cascade (sub-menu) style menus inside the scrolling menu, with support for scrolling sub-menus inside sub-menus. (Of course, actually doing this in your program would make it just about impossible to navigate :-)

Download it [here as a Stuffit](#) archive, or [here as a zip](#) archive.