# mFactory Object Model Reference

GUIDE•

**version 1.1**
**August 1996 Edition**

m TROPOLiS

MOM Reference Guide

© 1996 mFactory, Inc. All rights reserved.

mFactory, Inc.
1440 Chapin Avenue, Suite 200
Burlingame, CA 94010
USA
E-mail: info@mfactory.com
World Wide Web: http://www.mfactory.com

# Contents

# Chapter 1. Introduction

The mFactory Object Model (MOM) is a mechanism by which users can programmatically extend the functionality of mTropolis, using a language such as C or C++. With MOM, components can be built that seamlessly integrate with the mTropolis environment, from the underlying architecture to the visual interface.

When using MOM to extend mTropolis, you are in essence declaring new object classes. These classes are assembled into kits, placed in the mTropolis "Resource" folder, and initialized when mTropolis is launched. When mTropolis finishes launching, these classes are available for use by the mTropolis end user. MOM classes can be instantiated in a number of different ways.

Instantiated MOM classes are referred to as *components*. MOM components are organized into categories of classes, such as modifiers, tools, players and assets, and services as described below.

## MOM COMPONENT DESCRIPTIONS

There are four basic types of components that can be created with the MOM software development kit:

- **Modifiers**: Palette-based objects that can be dragged and dropped onto objects in any of mTropolis' editing views, just like the built-in mTropolis modifiers. Modifiers are instantiated by dragging them into a mTropolis project and dropping them on an object in an editing view.
- **mTools**: Editor-hosted plug-in tools, that add new features to the mTropolis editing environment and enhance or replace existing mTropolis features. mTools are similar to plug-ins for other products such as QuarkXTensions or Adobe Plug-ins.
- **Services**: MOM-based collections of routines that can be used by other MOM components. Services can appear in their kit's palette like a modifier (and they are instantiated in the same way as modifiers). Alternatively, services can be "faceless" and instantiate automatically (e.g., when the kit is first initialized).
- **Players and assets**: MOM components that can be used to play back non-native asset types in mTropolis. Using these types of components, developers can extend the types of media that mTropolis can display and manipulate.

These components are described in more detail below.

### Modifiers

The most common form of MOM component. mTropolis derives much of its power from modifiers. These components are usually accessed via a floating palette or library, and can be dropped onto an element in any of mTropolis' editing views. There are many types of modifiers including messengers, variables, graphical modifiers, action modifiers, behaviors, and mFunks. Modifiers are arranged in groups known as *kits*. Kits can have as few as one modifier, or may contain several modifiers. It is usually best to group modifiers that share similar functionality into kits. For example, a set of modifiers that perform realtime audio manipulation could be grouped into a "Sound" kit.

### mTools

mTools are MOM components that can be used to implement new features in the mTropolis editor. mTools can attach themselves to menus, get and modify information about the current project, etc.

### Services

A special kind of MOM component, services are modifiers that provide special functions to other modifiers. For example, a "Gravity" service might provide information about the gravitational aspect of a given scene. Modifiers that have been placed on the elements in that scene could provide data about how "massive" an element is, thereby providing a mechanism to interact with the gravity service.

There are two types of services—global and project. Global services have a scope across all projects. Project services have a scope that exists only for a given project. In addition to MOM based services, there are a number of built-in services in the core that MOM components can take advantage of.

### Players and Assets

MOM players provide playback capability for asset types that are not native to the mTropolis editor. For example, a JPEG player could provide the ability to play JPEG files in a mTropolis title. Support for the assets that they play are also implemented in MOM.

## HOW MOM COMPARES TO C++

Although MOM is a class hierarchy that allows you to create subclasses and provides methods that you can override, you do not necessarily need to write C++ code when creating a MOM class (although you can). Instead, you use a collection of macros and other mechanisms to write code that is object-oriented and completely cross-platform.

Many utility classes (such as list management and math services) are provided for you, so you do not need to duplicate the work done by others. Common ANSI routines are also implemented in the MOM interface, which in many cases eliminates the need to link in the ANSI libraries (thus keeping component size to a minimum).

Like C++, MOM components can be derived from other MOM components. However, unlike C++, you can derive MOM classes from other MOM classes without worrying about the traditional fragility of the base class.

When using C++ to build a class hierarchy, a change made to a base class requires recompilation of its descendant classes. When a MOM base class is changed, the subclasses do not necessarily need to be recompiled. They can automatically take advantage of improvements made to the base classes. For example, a “timer” messenger might be derived from a MOM superclass, such as a basic messenger, and inherit any future improvements made to the basic messenger.

The MOM API works on both Macintosh and Windows, so you can create a single component code base and build it for different platforms.

## BASIC CAPABILITIES OF MOM COMPONENTS

All MOM components share some basic capabilities. Some of these are:

- MOM components may (depending upon the type of component) affect many different parts of the mTropolis editor environment, open projects, and running titles.
- MOM components may inherit capabilities from other MOM components, in effect becoming descendants from their MOM superclass. They may call their superclass’ inherited methods and access their inherited class’ member variables.
- MOM components have the basic ability to create and send messages to other objects in the project, and process messages received from other objects.
- Components can traverse the entire mTropolis project hierarchy and ascertain information about each object located in it.
- Components may provide support for accessing their attributes and controlling their behavior via Miniscript, mTropolis’ scripting language.
- Components can save their private data into the project when it is saved, and restore it when the project is loaded.
- Components have full access to the system-specific toolbox API.

## BUILDING MOM COMPONENTS

MOM components can be built using C or C++ using just about any development environment that supports shared libraries and stand-alone code resources (for 68K Macintoshes). A detailed discussion of setting up the development environment is provided in the technical documentation.

Your MOM component will make use of the built-in core services that mTropolis provides, and may provide its own services or use those of other MOM components as it needs them. You do not need to provide any special linking to take advantage of the built-in services in MOM—just make the necessary function calls and they will be resolved for you.

Developing a component or set of components (a "kit") involves the following general steps:

- Decide upon the functionality required for the component(s). Does the desired functionality require more than one component to be built? If so, decide which functions each component will perform. Organize related components into individual kits.
- Decide what types of components will be needed, such as messengers, variables, services, etc.
- Design, implement, and test the components.
- Debug and make code changes as necessary.

## WHERE TO GO FROM HERE

Read Chapter 2, "Implementing MOM Components—A Technical Overview" to learn the technical details of implementing a MOM Component.

# Chapter 2. Implementing MOM Components—A Technical Overview

This chapter provides a general overview of the technical details of implementing MOM components. It discuss some of the features of the MOM architecture and the mTropolis application environment. Also included are discussions of some basic MOM Component capabilities, and how each is implemented. Examples are presented using C-style code.

After reading the introductory sections, you should read the section related to the type of MOM component that you wish to develop. These sections provide more information directly related to implementing that type of component:

- "Implementing Services" on page 2.39
- "Implementing MOM Modifiers" on page 2.33
- "Editor Construction Guide" on page 2.42
- "Implementing mTools" on page 2.52

## MOM COMPONENT IMPLEMENTATION

There are four basic steps involved with defining a MOM component:

- defining the component itself and its characteristics
- defining the component's superclass
- defining the methods that the component overrides and introduces, and
- connecting the component to the mTropolis-provided services.

This section describes these four basic steps, but first some definitions are in order.

### Basic Types and Definitions

The following basic definitions must be understood before continuing with this document.

#### The MOM ID

A MOM ID (also referred to as a CompID, or component ID) is a 16-byte sequence that serves as an identifier for a number of different purposes, which will be more fully explained later.

#### The "MSelf" Pointer

MSelf is used as a reference to your component's "self" pointer (C++ programmer's call it a "this" pointer; it's used to identify the pointer to the instance of an object). Many of the MOM API methods require an MSelf pointer as their first parameter.

#### The MOM Data Type

MOM provides data types for use in your components that are more complex than ordinary data types such as integers and strings. Most basic data types are mirrored in MOM and are named appropriately—integers are MIntegers, strings are MStrings, etc. These data types have an associated type field with them, making them implicitly aware of their type. This field allows the data to be passed around and correctly interpreted elsewhere, and also facilitates operations such as coercion and type negotiation. Several macros are provided for you to work with MOM data types, such as initialization, allocation, destruction, coercion, etc. Many MOM API functions accept only MOM data types as arguments (more on this later).

### Method Naming Conventions

In a MOM component, methods that are overridden or can be overridden by subclasses should begin with "MComp", followed by the component name. In order to call a function, you eliminate the "Comp" portion of the call.

## Defining the Component

The following is a typical header file for a MOM component:

```
#pragma once

#ifndef _Beep_
 #define _Beep_

 typedef struct BeepComp BeepComp;

 #define BeepCompID ((CompID *)”beepcomp\0\0\0\0\0\0\0\0”)

 #ifndef MSelf
     #define ComponentName Beep
     #define MSelf BeepComp
     #include "MFusion.h"
 #endif

 typedef struct BeepComp {
     MEventf_activateEvent;
 } BeepComp;

 const shortkBeepRev = 0;
 enum {
     kBeepNumProcs
 };

 const     kBeepSlot = kMBaseCompSlot + 1;

#endif
```

Nothing too exotic here. The header file simply defines a structure called BeepComp, which will be used to determine our instance member variables. We also define kBeepRev to be a revision identifier for our component (some MOM service calls will require this, particularly saving to and restoring from a project), kBeepNumProcs to be the number of methods that our component will introduce (none, in this case), and define kBeepSlot to be the "slot" of our component (which is 1 greater than the slot of our superclass). We also define BeepCompID to be the MOM ID of our component, which is "beepcomp" padded with 0s.

This is a typical source file snippet. MOM components have a main entry point, typically declared as follows:

```
MCompMainFuncDcl MCompMainName(MInitInfo* initInfo)
{
 MDefineComponent(BeepComp, kMPaletteBased);

 MInheritClass(MComponent, kMCompNumProcs, kMBaseCompSlot);

 // MComponent methods we must to override

 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompConstructor, MCompConstructor);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompCopy, MCompCopy);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompCopyConstructor, MCompCopyConstructor);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompDestructor, MCompDestructor);

 // MComponent methods we choose to override

 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompEditorOpen, MCompEditorOpen);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompEditorAccept, MCompEditorAccept);

 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompGetSaveInfo, MCompGetSaveInfo);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompSaveComponent, MCompSaveComponent);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompRestoreComponent, MCompRestoreComponent);

 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompEnabled, MCompEnabled);
 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompDisabled, MCompDisabled);

 MDefineMethod(kBeepSlot, kMBaseCompSlot, kMCompProcessMessage, MCompProcessMessage);

 MConnectCommonServices(initInfo);

 MEndComponentDef;
}
...

// Method definitions follow here...
```

The first line, MDefineComponent, takes two parameters: a structure definition that defines the components' instance variables, and a combination of flags that determine the type of component that is being declared. Valid values for the flags parameter are:

- `kMDefaultOptions`: No specific options specified
- `kMPaletteBased`: Component is instantiated by drag and drop from its palette
- `kMAutoSpawn`: Component is automatically spawned at startup
- `kMPropertyType`: Component can be listed in a 'With Data' popup
- `kMFloatingWindow`: Component's editor window is floating
- `kMNoLayer`: Component does not appear in Layout or Layers window
- *Note: More than one component in a kit may declare itself as palette-based. One palette will be created for each kit that contains at least one palette-based component. Developers may also specify the order in which components are listed in the floating palette; more on how to do this later (it basically involves a resource definition).*

### Inheriting Properties from the Superclass

The next line, MInheritClass(MComponent, kMCompNumProcs, kMBaseCompSlot), defines the superclass that this component will inherit from, which in this case is MComponent. It takes three arguments: the superclass, the number of procedures in the superclass (provided as a constant), and the "slot" of the superclass (also provided for you as a constant).

### Overriding the Required Methods

The method definition and overrides come next, in the form of several calls to MDefineMethod(), which takes four arguments: your component's "slot" (defined as the superclass slot + 1), the slot of the superclass, the method index to override, and the name of the actual method as it appears in your code.

Your component must override four methods in order to work properly (note that in order to actually be useful your component will have to override more methods; this is just the bare minimum): MCompConstructor, MCompDestructor, MCompCopyConstructor, and MCompCopy. The constructor methods deal with creating a newly instantiated component: the user has dragged and dropped it from a palette, or kMAutoSpawn was specified in the definition. Here you should initialize and allocate any necessary space for your instance variables. In the case of the copy constructor, you should copy the values of the copied instance (which is provided) to the new instance. You may decide to do some general initialization here, but there are other points in time where you may wish to do initialization instead.

The destructor method is called when your component is destroyed. You should deallocate any space that was previously allocated.

The MCompCopy method is called when an alias to your component is updated. As with the copy constructor, you should copy the values of the source instance to the destination instance.

### Overriding Other Methods

In addition to the required methods, your component should override any other methods it needs to in order to carry out its tasks. Some methods must be overridden depending upon the type of component and what features it implements. Modifiers, for example, must override the MCompGetAttribute method so that the Behavior modifier can identify its execution and termination events.

### Defining Your Own Methods

You can also use the MDefineMethod() call if you want to introduce your own methods into the component that other components may eventually override. You do not need to do this, however, if you simply want to define methods that are private to the component; just declare the functions as you normally would and call them like any other function.

### Connecting to mTropolis

The last line, MConnectCommonServices, connects your component to the mTropolis services. This call, along with MEndComponentDef, should always be the last calls in your main routine.

## PROVIDING SUPPORT FOR BASIC mFACTORY TECHNOLOGIES

mFactory provides some very basic technologies in mTropolis that enable developers to create powerful, independent, and reusable objects. Supporting these technologies is very important, because users will expect these capabilities, but even more so because they are fairly straightforward to implement and cost little in the way of development resources.

MOM Components should make use of the provided MDataType structures, which simplify working with the MOM toolbox. MOM components should provide support, where appropriate, for persistence, editing, messaging, attributes, and Miniscript. Another basic technology which your component may optionally choose to support is the mFunk.

Each of these functional areas is briefly discussed below. Details on each area are provided in separate sections of this book.

## MDATATYPE STRUCTURES—MOM DATA TYPES

MOM provides predefined structures called MDataTypes that your component can use to simplify communicating with mTropolis and the MOM toolbox. mFactory's MOM Data Types exist to make it easier for a developer to work with data types that are “smarter” than the built-in types that C and C++ typically use. MOM Data types are implicitly aware of their own type, because each type contains a “member” variable labeled f_type, which is set to one of the known MOM data types listed in the MData.h file. This implicit type awareness facilitates many operations, such as type negotiation, coercion, and abstraction. You can often write one method that simply accepts a MOM Data Type as an argument and return any kind of data in it.

### Ownership of Memory

Certain MDataTypes require memory to be allocated in order to use them, such as MStrings. These MDataTypes have a field called f_owner, which is set by the MSelf pointer of the component that creates the memory.

In some cases, however, the component that creates the memory may not ultimately be the component that is responsible for destroying it later. This presents a problem—determining the ownership of the memory object.

## COMPONENT PERSISTENCE—SAVING AND RESTORING INFORMATION IN THE PROJECT

Your component, if it is part of a project, must save its information into the project when it is saved and restore it when it is loaded. There are three methods that need to be overridden to support this feature.

Your component’s MCompGetSaveInfo method will be called when mTropolis needs to know the size of the data that you will be writing out. After responding with the data size, the MCompSaveComponent method will be called, in which you will write your component data to the project stream. When the project is reloaded, then the component’s MCompRestoreComponent method will be called, and you can then read your data back in.

### Related Methods

MCompGetSaveInfo,
MCompSaveComponent,
MCompRestoreComponent

## PROVIDING SUPPORT FOR USER EDITING

You may wonder, “What does editing have to do with my component?” The answer is, “a lot,” if your component is palette-based.

Users can open and edit the attributes of palette-based components by double-clicking on them. You do not need to provide any explicit support for handling the editing window; this is done automatically for you. What you do need to provide is a dialog template that defines the window and the user interface elements that it contains. You also need to override certain methods in your component to provide support for actions such as getting and setting fields and controls of the dialog, handling any custom clicks or controls, and validating the content of the window.

### The Editing Window

Handling the editing window is relatively straightforward. Your component's MCompEditorOpen method is called when the dialog is opened, allowing you to set up the fields and controls it contains. If the user clicks on the "OK" or "Cancel" buttons, then the MCompEditorAccept and MCompEditorDecline methods, respectively, are called. When the user types a key or clicks on something, the MCompEditorItemChanged() method is called.

Creating the dialog template is discussed at length in "Editor Construction Guide" on page 2.42.

### Related Methods

MCompEditorOpen, MCompEditorItemChanged, MCompEditorAccept, MCompEditorDecline, MCompEditorPreValidateItem

## PROVIDING SUPPORT FOR MESSAGING

Messaging is the glue that holds a mTropolis project together. Messaging is the ability to package a message along with some accompanying data and send that message to another object in the system. This capability allows for very flexible programming and the creation of reusable elements.

Your modifiers should provide robust support for messaging wherever appropriate by overriding the MCompProcessMessage method and by using the MInitMessage and MSendMessage methods.

### Related Methods

MCompProcessMessage, MInitMessage, MSendMessage

## IMPLEMENTING ATTRIBUTES

Attributes are a very powerful aspect of the mTropolis environment, and are used heavily throughout the application and mFactory-built components. It's difficult to implement a good, robust mTropolis component without supporting attributes, so it's very important to understand how they work and what role they play.

### Attributes Defined

An *attribute* is a lowercase (important!) MOM ID that is used to uniquely identify an "attribute", or property, within your component.

For example, in the Beep component, we may define an attribute such as the number of times to beep when activated as:

```
#define kBeepCountAttrib(CompID
*)”beepcount\0\0\0\0\0\0\0”)
```

### Why Attributes are Important

Attributes are important because they can be manipulated by users (using Miniscript) and other by components (using the MGetAttribute and MSetAttribute methods), thus allowing a certain degree of "programmable" access to the internal settings of your component, similar to the way AppleScript provides control over applications.

### How Attributes Are Used

MOM provides several methods for using, manipulating, and enumerating the attributes that a component responds to. As a general rule, you should provide the capability of manipulating the attributes that you make available to users in your component's editing dialog (if it has one). You should also provide access to any attributes that you want the user to be able manipulate using Miniscript.

Some commonly used attribute names are supplied in the MOM SDK header files that end with "attr.h", such as MPlyAttr.h, which lists attributes for use with player objects.

### Related Methods

MCompGetAttribute, MCompSetAttribute, MCompGetAttributeCount, MCompGetNthAttributeName.

## PROVIDING SUPPORT FOR MINISCRIPT

Miniscript is mFactory's scripting component and language. Users can write Miniscripts to control their projects and send messages to objects in the system.

### How To Support It

Support for MiniScript is fairly basic and straightforward, and requires almost no additional effort on the part of the developer. In fact, if you've already provided support for attributes in your component, then you've pretty much implemented the necessary code to interface with miniscript.

Miniscript will call your component's MCompGetAttribute and MCompSetAttribute methods when the user refers to them in a Miniscript modifier.

For example, suppose that our Beep component is attached to an element and it is named "beep", and the user types the following into a miniscript:

```
set beep.beepcount to 5
```

Your MCompGetAttribute method will be called with the attribName field set to "beepcount" and the dataValue field set to be an MInteger with a value of 5.

### Related Methods

MCompGetAttribute, MCompSetAttribute

## SUPPORTING MFUNKS

mFunks are a special type of callback function that your component may support. Currently, mTropolis supports mFunks that affect the way that elements are drawn. MFunks are discussed in detail in "mFunks" on page 3.61

### Related Methods

MCompPreDrawBuffered, MCompPostDrawBuffered, MCompDrawBuffered, MRegisterFunk, MUnregisterFunkByID

## OTHER GENERAL IMPLEMENTATION ISSUES

This section discusses practical issues involved with implementation of MOM Components.

### Well-Behaved Components

Writing code that performs well under a variety of situations is important, since your MOM component may be loaded and unloaded at different times and under a variety of circumstances. Your component should not make any assumptions about its current environment unless they are specifically documented, and even then you should always proceed carefully. The instantiation lifetime of a MOM component cannot be guaranteed, because memory situations and other considerations may cause your component to become instantiated and destroyed unpredictably.

### How and When Components are Instantiated

Components can be created and destroyed at almost any time. Due to the non-linear nature of mTropolis and the titles it enables, your component will have to be prepared to deal with being instantiated and destroyed under a variety of circumstances.

Generally speaking, your components will be instantiated when the user drags them from a palette (during editing time) and when a scene containing them is loaded (during runtime). Destruction occurs when the user deletes your modifier from an element (and then commits the process by performing another action), or when a scene containing the component is unloaded. In the future, however, components will be able to instantiate and destroy other components.

Your MCompConstructor and MCompCopyConstructor methods will be called when instantiation occurs. You may do most of your initialization at this point. Do not, however, confuse construction with connection: your component may not be fully hooked up to its parent at this point in the process. When your component receives a parent, its MCompConnect method will be called (and conversely, its MCompDisconnect method will be called when you are disconnected from your parent). You may do any initialization that requires that your component be connected to its parent at this point.

### Player-Vs.-Editor Issues

MOM allows developers to build components that will run under both the mTropolis Editor and the mTropolis Player. Kits that have a suffix of ".e68" and ".ePP" are editor-based components. The suffixes ".r31", ".r95", ".r68", and ".rPP" are used for runtime components (the "e" and "r" denote "editor" and "runtime", the other two characters denote the platform—68 for 68K, PP for PowerPC, 31 for Windows 3.x, and 95 for Windows95/NT). The main difference is that since the player does not have a user interface, you can omit several methods from your component's code when it is built for use during runtime. Two header files are provided for Macintosh development (they aren't needed under Windows). These files are named MOMEditorPrefix.h and MOMRuntimePrefix.h. They contain the necessary compiler switches to build your components correctly. There is a compiler switch provided called _MFEditor_ that should be set to TRUE when building editor-based components. Note that mTools, by definition, are editor-based components, and should therefore have a suffix of ".e68" or ".ePP".

### Memory Management of Components

Memory management is, for the most part, left up to the component. Components may acquire and release memory as they need it. All memory management that is performed by your component should be done using the built-in MOM Memory service, which provides cross-platform memory management routines using a single API.

The Memory service will be enhanced in the future to provide object tracking and other statistical tools.

Be sure to release any memory that your component acquires. If you do not, then mTropolis will have no way of reclaiming the memory, and leak will occur. Over time, users may experience performance hits and other worse behavior if memory gets too low.

### Using the Built-In Services

In general, you should use the built-in MOM services where possible to accomplish your programming tasks. A list of services can be found in Chapter 4, "Service Reference".

Services are provided to handle a variety of jobs, such as list and string manipulation, memory management, and debugging.

Before creating your own code to handle a particular job, be sure to check to see if there is a service available, either built-in or third party, that implements functionality that you need.

### Installing Periodic Tasks

Components may install periodic tasks that will be called when a time period that the component specifies has elapsed. Your component registers its timer tasks using the Task Service described in "MOM Task Services" on page 4.250.

The example Gravity Service and Modifier make use of these timer tasks.

### Handling Environment Changes

An "Environment Change" is a type of notification that your component receives from mTropolis. Most MOM components will have to override the MCompEnvironmentChanged method, which informs your component that its surroundings have changed somehow, and that it needs to update certain types of data in response.

This subject is discussed in detail in "Implementing MOM Modifiers" on page 2.33.

### Use mFactory Data Types

Although you are generally not forced to use MOM data types throughout your own code, there are certain instances where you must use MOM data types (in communicating with the Core, for example). Many MOM API calls accept only MOM data types, such as MWriteValue(). Generally speaking, if space and speed are not of the utmost importance in a given situation, you should strive to make use of the mFactory-provided data types, because it will save you the trouble of having to convert them whenever you want to pass them to an API call. The mFactory data types also work across platforms, providing another compelling reason to use them.

Perhaps the most compelling reason of all, however, to use MOM data types is that interface elements that are used by some types of MOM Components can be linked to corresponding MOM data types in your code. For example, a popup menu that displays static strings in a modifier's editing dialog can be set and queried using the MSymbol data type.

For your own internal mathematical calculations or for data that you won't need to pass to the Core, however, you can feel free to use the built-in C data types.

### Cross-Platform Code

The MOM SDK provides several mechanisms for you to produce code that should work across multiple platforms, including Macintosh 68K and PowerPC, and Windows 32 bit and 16 bit.

Several data types that are used throughout the MOM SDK are defined as so-called "PL" types. These types will compile to their native platform types, but are defined so that your code does not need to worry about their specific implementation. For example, the Macintosh "Rect" and the Windows "RECT" structures are both defined to be PLRect, and will compile correctly for each platform. mFactory has also provided some complex data types to represent structures such as bitmaps and drawing contexts that can be used across platforms. Your code should take advantage of these definitions. The files "MType.h" and "MWinType.h" and "MMacType.h" contain many of these predefined data types.

### mTropolis Events

"Events" are messages that are sent from one object to another, or by the system to an object. There are several different kinds of events: mouse events, which are related to the user's mouse actions; scene control events, such as "scene started" and "scene ended", and author events, which are defined by the user. The file "MConst.h" contains event code definitions for the events that your component may receive or send.

## HANDLING mTROPOLIS OBJECTS

mTropolis projects are made up of many different types of objects: elements, scenes, sections, modifiers, etc. All but the most basic of your components will need to manipulate objects in order to perform useful tasks.

Object handling in mTropolis is done in several ways, depending upon how you want to retrieve an object and how you need to manipulate and keep track of it. Objects can be dynamically loaded and unloaded during the editing and title execution processes, and this is especially true of large projects. Your component can make use of the same methods that mTropolis uses to manage objects that may or may not be in memory at any given time. Your components may also retrieve and set attributes for mTropolis objects.

### Object IDs

Object IDs are long values assigned to objects when they are created. These values uniquely identify an object within its project. These values are persistent; they are saved with the project and restored when the project is reloaded.

### Using MAdaptData()

Since ObjectIDs are unique to a project, it is likely that an object's ID will need to change when it is moved to a new project. A new object ID also needs to be generated when an object is duplicated. Similarly, event IDs will need to be updated in this manner.

If your component keeps track of an object ID, then it may need to be updated in response to one of the aforementioned actions taking place. This is the purpose of the MAdaptData() method. An overview of MAdaptData() is provided below. A complete description can be found in "MAdaptData" on page 4.212.

You should call the MAdaptData() method when your component's MCompEnvironmentChanged() method gets called. You should pass the data given to your component by MCompEnvironmentChanged on to MAdaptData, along with the MDataType that needs to be updated. The sample component code illustrates how to do this and what types of data need to be "adapted."

### MDataTypes for Working With Objects

Your component will use the mFactory-provided MDataTypes MObjectRef, MPartialIDPath, MIDPath, and MObjectPtr to keep track of its objects. These data types are described below.

### MObjectRef

```
typedef long MObjectRef;
```

An MObjectRef is a direct pointer to a mTropolis object.

### MObjectPtr

```
typedef struct MObjectPtr {
 short f_type; // Set to kMObjectPtr
 struct MObjectPtr *f_previous; // Private, do not alter
 struct MObjectPtr*f_next;// Private, do not alter
 MObjectRef*f_obj;// A ref to a c++ element
 MPartialIDPath*f_partialIDPath;// Optional, if allocated & MRegisterObjectPtr
                                // called, it is filled in if f_obj unloads
                                // so that it can be resurrected if required
} MObjectPtr;
```

An MObjectPtr is a type of pointer that can be registered with the object that it points to. When that object is unloaded, the registered MObjectPtr is set to NULL. In addition, a partial ID path may optionally be included with an MObjectPtr, so that the object that was unloaded can be reloaded if it is needed.

### MIDPath

```
typedef struct {
 short   f_type;     // set to kMIDPath
 short   f_nItems;// number of IDs in the path
 long      *f_pathIDs;// list of object IDs, (usually starting with section)
} MIDPath;
```

An MIDPath specifies a location in a project using object IDs.

### MPartialIDPath

```
typedef struct {
 long   f_objID;       // The f_obj ID
 long   f_pathIDs[kNPartialIDPath];// partial list of object IDs, starting
                                   // with projectPlayer
} MPartialIDPath; // Use MPartialIDPath along with an f_objID as a fairly fast &
                  // compact version of MIDPath
```

A shortened version of MIDPath, an MPartialIDPath has a fixed depth, and refers to a specific object that is somewhere below that level in the hierarchy.

## Retrieving Objects

There are two methods that can be used to retrieve objects:

#### Using MGetElement()

One of the simplest ways that your component can get access to a mTropolis object is via the MGetElement() method, which returns an MDataType of type MObjectRef to the parent object of your component. This object pointer is a "hard" object pointer.

#### Using Attributes

Components can also retrieve pointers to objects by issuing MGetElementAttribute() calls to a given object (which may have been obtained by MGetElement()).

## Keeping Track of Objects

Because large projects may demand more memory than mTropolis has available to keep objects loaded, mTropolis may be forced at times to "unload" certain objects from memory. The objects still exist, of course, but any pointers that you had to them will become invalid at this point. In addition to this condition, an object may be disconnected from its current parent and re-parented elsewhere in the project hierarchy. Objects may also be duplicated, or "cloned", in which case a new object ID will be given to the newly created object.

## Hard, Rigid, and Soft Object Pointers

mTropolis allows you to reference objects using different types of pointers, referred to as "hard", "rigid", and "soft" pointers, each of which is used in different types of situations.

#### Hard Pointers

A "hard" object pointer is a direct pointer to a mTropolis object, usually specified with an MObjectRef pointer. You use an MObjectRef pointer when you need to directly manipulate and object immediately after retrieving it. Note, however, that if you return control to mTropolis, the object that you are pointing to may be killed off, at which point your MObjectRef pointer will become meaningless. This problem can be solved using "rigid" pointers.

For example, suppose that a component wants to set the attributes of a mTropolis object—the object that is the parent of the component. It does this by calling MGetElement() to retrieve the component's owner, then calls MSetElementAttribute and MGetElementAttribute to it's heart's delight, changing whatever it wishes.

#### Soft Pointers

Also known as "location" pointers, soft pointers should be used to find out if any object exists at a given location in a project. They are specified using MIDPaths and MPartialIDPaths. They are commonly obtained using the MGetObjectPath and MGetObjectPartialIDPath() methods.

Note that you can use the methods and data structures to quickly access an object at any particular point in a project, provided that you have the paths to those objects. If you *do not* have these paths, you may need to do a hierarchical search of the title to find a given object the first time.

**Rigid Pointers**

"Rigid" pointers are a cross between a hard pointers and a soft pointers, in that they can automatically update themselves to reflect the fact that the object that they point to no longer exists, and that they can be used to reload an object that has been unloaded.

Rigid pointers are specified using the MObjectPtr MDataType. The MObjectPtr variable should be initialized with the MInitObjectPointer() macro. You should then call the method MRegisterObjectPtr() with your MObjectPtr variable, which will fill in the MObjectPtr's private data fields and notify the object that it must reset the fields of your MObjectPtr to NULL when it is deleted. Your component should check for a null value in the MObjectPtr's f_obj field before attempting to use the object.

Consider the following example. A modifier component provides functionality that allows an object to detect collisions with other objects. In the process of doing so, the modifier maintains a list of the objects that it's owner is currently colliding with. The problem is that any one of those objects may be killed while the modifier component has relinquished control to mTropolis. To solve this problem, upon executing its code again, the modifier first checks to see if an object it is keeping track of (by using MObjectPtrs) has been killed by checking the f_obj field of the MObjectPtr data structure. If it is null, then the modifier does not try to access that object—it is gone, and the modifier can remove it from the list.

Note that the above approach works well only if your component doesn't care what happened to the object it was keeping track of, and was only interested in whether the object existed or not.

What should your component do if it needs to access an object, even if it has been unloaded? It should make use of an optional field provided in the MObjectPtr structure: an MPartialIDPath structure.

An advantage of using rigid pointers, or MObjectPtrs, is that they contain an MPartialIDPath, which your component can optionally fill in. If this portion of the MObjectPtr has been filled in, then upon checking the f_obj field of the MObjectPtr and finding it to be null, your component can call the MFindObjectByPartialIDPath() method to retrieve a pointer to it and have the object reloaded into memory. To obtain a partial ID path for a given object, your component should use the MGetObjectPartialIDPath method, which is part of the MOM Query service.

- *Note: Your component must call MUnregisterObjectPtr() if it dies before the object that it is keeping track of, so that the object knows not to try to reset your MObjectPtr variable. You must also call MUnregisterObjectPtr() if your component's container becomes disabled (e.g. you receive a container disabled event).*

## WHERE TO GO FROM HERE

You should read the sections on implementing the various types of MOM components that are related to the components you need to develop:

- “Implementing MOM Modifiers” on page 2.33 provides a detailed discussion of the issues related to developing modifier components.
- “Implementing Services” on page 2.39 talks about the unique details of service-style components. “Implementing mTools” provides information related to developing editor-hosted components, which are intended to enhance the mTropolis editing environment.
- “Editor Construction Guide” on page 2.42 describes how to create user interfaces for your MOM components.
- “Implementing mTools” on page 2.52 describes the process of adding new functionality to the mTropolis editing environment.

You should also become familiar with the mFactory User Interface guidelines, and follow the recommendations for components that need to implement a user interface.

There are various sections explaining the details involved with implementing the various MOM technologies that are related to all component types. You should become familiar with these at a basic level to begin with, then try using the methods from each section to implement new features in your components.

The sample code provided with the SDK is always a good place to look to glean information about creating MOM components, and the examples can be used as a starting point for your own components.

Finally, Chapter 6, “MOM FAQ” contains several frequently asked questions that may provide some insight into a specific problem that you are facing.

## IMPLEMENTING MOM MODIFIERS

This section discusses implementation issues related directly to the construction of modifiers.

## WHAT ARE MODIFIERS?

The term “modifier” is used throughout the mTropolis vocabulary, and is generally used to describe objects that can be dropped onto elements to “modify” their behavior in some way. Modifiers are usually distinguished from other MOM components by their appearance in a floating palette in the mTropolis editor. Groups of modifiers can be arranged into “kits”, each of which have one floating palette and one icon per modifier.

There are, in fact, several different types of modifiers—including variables, messengers, and functions—each of which perform a different type of action but are constructed and coded the same way.

## BASIC PROPERTIES OF MODIFIERS

In addition to the basic properties of MOM components, MOM modifiers exhibit the following properties:

- Modifiers are typically palette-based. They can be dragged from a palette and dropped onto an object in a project.
- Modifiers have icons associated with them that convey information related to what type of modifier they are and the type of function that they perform.
- A modifier may have a user interface associated with it. It is accessed by double-clicking an instantiated modifier to open its editing window (called a *configuration dialog* in the mTropolis Reference Guide).
- Modifiers may provide a “preview” of what their runtime effect will be during editing by overriding the MCompApplyEditorEffect method, discussed in detail later.
- Users may view a modifier’s name by holding down the Control key while the cursor is positioned over the modifier on its palette.

## RESOURCE USAGE AND NAMING

Your component will need to include certain resources to operate properly in the mTropolis editing environment. Since the mTropolis editor currently runs only on the Macintosh, this discussion will focus on Macintosh resources. Your component will also need to define the symbol MComponentName with the name of the component. This name will be used to access your component’s resources. For example:

```
#define MComponentName MyComp
```

See the documentation that came with your resource editing program (e.g., ResEdit or Resorcerer) for instructions on how to name resources. Naming resources usually involves a simple step, such as choosing the resource, displaying its information dialog, and entering a name.

### Colorized Icon ('cicn')

Modifier components have colorized icons that appear in their kit's floating palette. This icon is supplied by you, and should be a named 'cicn' resource that has the same name as the one you defined MComponentName with.

### Editing Dialog ('DLOG')

If your modifier supplies an editing window, then you will need to provide a dialog (DLOG) resource in your component and name it with the same name as the one you defined MComponentName with.

### String List ('STR#')

Your component will need to supply a 'STR#' resource, named with the same name that you defined MComponentName with. The first string should be the full name of the component. This string will be displayed to the user when she holds down the Control key and moves the mouse pointer over the modifier icon on the kit's palette.

## THE MODIFIER USER INTERFACE

When a user double-clicks a modifier, they are presented with a dialog that allows them to customize the settings of the modifier to perform a certain function.

Details on presenting a user interface and handling user input are covered in "Editing" on page 3.56.

You should also read "Editor Construction Guide" on page 2.42, which illustrates the necessary resources you need to include in your component to define the look and feel of your editing window.

For guidelines on providing an mFactory-standard interface, you should read Chapter 5, "User Interface Guidelines".

Appendix C contains a list of codes that can be used in an editor dialog.

## DEFINING THE COMPONENT AND OVERRIDING THE METHODS

Modifiers typically use the kMPaletteBased flag in their declaration:

```
MDefineComponent(MyComp,kMPaletteBased);
```

### Handling the Basic Required Methods

As with any MOM component, your modifier must override the following four basic methods to work properly:

- MCompConstructor
- MCompCopyConstructor
- MCompCopy
- MCompDestructor.

You should use the constructor functions to allocate and initialize any instance-specific data. The MCompCopy routine is called when an alias of your component is being updated. You should deallocate any instance-specific data in the MCompDestructor method. The sample components illustrate how to override these functions.

### Other Mandatory Overrides

If your component makes use of events or keeps references to other objects, then it must also override MCompEnvironmentChanged(), which will be called when the editor environment has changed in such a way as to affect your component's data. Currently, there are two instances when it is called: your component has been dragged to another project, or objects in the project have been duplicated. When this method is called, you must respond by calling the MAdaptData() method with your event data, object references, and references to sound file markers.

The MCompEnvironmentChanged() and MAdaptData() combination looks like this:

```
static MErr
MCompEnvironmentChanged(MSelf *self, MEnvChangeReason changeReason, void
*changeData)
{
  MAdaptData(reason, changeData, (MDataType *)&self->f_executeEvent);
  return kMNoCompErr;
}
```

The “changeReason” and “changeData” fields are opaque data structures that you need to pass to MAdaptData(). The last parameter is the MDataType that needs to be “adapted” to the environment change; upon return it will be modified as necessary to reflect the change in the environment.

This needs to be done for several reasons. For example, when a modifier is dragged to a new project, it may need to update its message ID, because certain message types (like author messages) are project specific. Also, if an object that your component keeps track of is duplicated, you may need to update the object reference. See the sample code projects for examples of how to override MCompEnvironmentChanged and how to call MAdaptData.

### Messaging

During a title's execution, your modifier may receive messages sent to it from other components. If your modifier responds to messages, then it needs to see if a message that has been received is one that it responds to, and take the necessary action.

Different types of modifiers will respond to different types of messages. Messenger modifiers, for example, will listen for a specific trigger event, then send a given message along to a given destination. Other types of modifiers will have separate events that are used as “triggers” that cause the modifier to be enabled or disabled.

Your component will use the MDetectMessage() macro to determine if it has received a message that it is configured to respond to. This call typically looks like this:

```
if (MDetectMessage(message, self->f_executeEvent)) {
  // process the message by performing your component’s duties
  ...
  return kMNoCompErr;
}
else
  return kMUnableToComplyCompErr;
```

The “MoveToPt” example modifier contains a good example of message handling.

### Attributes

As a general rule, your modifier should make any settings that it allows users to modify available as attributes that can be set and queried by other components.

Attributes are provided mainly for interfacing with Miniscript, the mTropolis scripting component and language. They are also useful for components to communicate with each other when they do not know beforehand what methods are implemented by each other.

You should give your attributes lowercase names, because Miniscript is case-insensitive and uses lowercase letters in its operations.

See “Component Attributes” on page 3.53 for an in-depth discussion of handling attributes. You should also review the modifier code examples, which show how to use the routines MCompGetAttribute and MCompSetAttribute.

### Persistence

Your modifier needs to write its data (like the events it responds to, etc.) into the project stream when the user builds a title or saves the project so that it may reload its settings when the title runs or the project is reopened. Your component accomplishes this by using the provided persistence methods.

See “Component Persistence” on page 3.55 for a detailed discussion of these methods.

Using the provided methods and data structures, your component can provide revision information and learn specific information about the save/restore process, such as whether a title is being built or the project is just being saved, what platform it is being saved to, etc.

Most of the example modifiers make use of the Persistence methods, and are a good place to get starter code from.

### Adding Your Own Methods to a Component

One of the cornerstones of MOM is the ability to descend MOM classes from other MOM classes at runtime. To provide this functionality in your components, you will need to create function call macros to handle the different methods that subclasses will need to call.

When your component introduces a new method, you should also supply an "_Execute" macro for that method so that subclasses of the component can call superclass methods. For example, if the Messenger modifier defines the following method:

```
MCompStdFuncDcl MCompSendMessengerMessage( MsgrComp *self )
{
...
}
```

the following method macro should be defined:

```
#ifdef __cplusplus
 extern "C" {
#endif

 typedef MErr ( COMPCALLTYPE *SendMessMessageType )( MSelf *self );
 MCompStdFuncDcl MCompSendMessengerMessage( MsgrComp *self );
 #define MSendMessengerMessage( self ) \
     _Execute0( self, kMsgrSlot, kMsgrSendProc, SendMessMessageType )

#ifdef __cplusplus
 }
#endif
```

The "_ExecuteN" macro has the syntax:

```
_ExecuteN( self, kMyCompSlot, kMyMethodIndex, MyMethodPrototype )
```

Where "N" is the number of arguments the macro accepts, up to 9. Parameters are:

| Parameter | Purpose |
|---|---|
| self | The "self" pointer of the component. |
| kMyCompSlot | The class "slot" of the component. |
| kMyMethodIndex | Index of this method in the class. |
| MyMethodPrototype | Prototype of the method to call. |

In MOM, just about all component methods are called in this manner. You can look in the header files for examples of _Execute macros for the methods of each class. By convention, method macros do not have “Comp” in their name. The “Comp” designation is reserved for the method that is overridden by the component; to call the method, you drop the “Comp”.

In the above example, the Messenger modifier defines the method MCompSendMessengerMessage, which may be overridden by a subclass component. To actually call the method from the subclass, however, you would call it by using the macro MSendMessengerMessage.

The example modifier code contains the Timer Messenger and regular Messenger, which provide an example of how to descend a MOM class from another MOM class.

## IMPLEMENTING SERVICES

A MOM "Service" is a MOM Component that provides a particular service to a set of clients. Services can be instantiated in different ways. They may be automatically instantiated when mTropolis launches, they may be instantiated by drag-and-drop from a floating palette, or they may be instantiated automatically when a client requests a connection.

For example, a String Service may provide clients with the ability to manipulate string data, or a Gravity service may be called upon to provide gravitational information about a certain scene or section.

Like modifiers, MOM Services may have user interfaces associated with them, which can be accessed by double-clicking on an instantiated service icon.

Services may be registered at the Global level or at the Project level. Global services are available to any client in any project that requests them. Project services are available only to clients within that project.

### Related Information

If you have not already done so, read "Implementing MOM Modifiers" on page 2.33, because services are a superset of modifiers. You should also become familiar with the earlier sections of this chapter which cover topics that are common among MOM Components.

If your service will have a user interface associated with it, then be sure to read Chapter 5, "User Interface Guidelines".

Details on presenting a user interface and handling user input are covered in "Editing" on page 3.56.

You should also read "Editor Construction Guide" on page 2.42, which illustrates the necessary resources you need to include in your component to define the look and feel of your editing window.

You should also become familiar with Chapter 3, "Basic MOM Technologies", especially "Component Attributes" on page 3.53 and "Messaging" on page 3.58.

## BUILDING A SERVICE

Services are constructed just like any other MOM Component. In fact, there really isn't anything structurally different about them. Their main difference is that the work they perform is usually for the benefit of other components.

Services may be a part of a component kit just like any other MOM components. If they are defined with the kMPaletteBased flag, then they will appear in the kit's palette.

- *Note: Services that are defined using the kMPaletteBased flag cannot use the kMAutoSpawn flag, because palette-based components are instantiated by dragging them to a project, which is mutually exclusive from automatic instantiation.*

## REGISTERING THE SERVICE

When you want a service to become available to potential clients, it needs to be registered at either the project level or the global level.

Registration is accomplished using the MRegisterGService and MRegisterPService methods. When the service becomes unavailable, you should call MUnregisterGService (or MUnregisterPService) as necessary. You will need to unregister your service when it is destroyed or becomes disabled. For example:

```
// Register the service at the global level
  MRegisterGService( self, self, kMGeneralServiceKey );
...
// Unregister the Service
  MUnregisterGService( self, self );
```

## CLIENT CONNECTIONS

When a client wishes to connect to a MOM Service, it first calls MGetGService() or MGetPService() (for project-level services) to see if there is an instance of that service available. There is an option in the method call to instantiate the service if it has not been. This method returns a "self" pointer to the service.

Clients connect themselves to a Service by calling MAddClient(), which adds the "self" pointer of the client to the list of clients maintained by the service. When a client wishes to remove itself from the list, it calls MDeleteClient(). The service does not receive any type of notification when its client list changes; the MGetNumClients() method must be called before servicing clients to ensure that there are in fact clients to service. For example:

```
  MService*gravServ;

  MGetGService(self,kGravServID,kMAnyInstance,kMSpawnIfNotFound,&gravServ );
  if ( MError() == kMNoCompErr )
  {
       long  globalTime;
       doubleradians, cosVal, sinVal;

       MAddClient( gravServ, self );
  }
```

- *Note: One way to work around this, if necessary, is to require clients to call MSetAttribute on your service with any information that it needs to know at client registration time.*

Clients do not necessarily need to register themselves with a service in order to take advantage of a service's offerings. This capability is provided for the convenience of services that may need to periodically notify clients of interesting events related to the service. For example, the above mentioned String Service does not need to maintain a client list, nor do clients need to register themselves with the service (though they do need to call MGetGService or MGetPService to make sure the service is really there). They may simply call predefined methods in the service to perform the string manipulation. More advanced services, however, may need to keep a client list for notification purposes.

## SERVICING CLIENTS

Assuming that your service keeps a client list, there will come a time when you need to service the clients in the list. Your service should first call MGetNumClients() to see how many clients it has to service. You should then loop through each client using MGetNthClient() to retrieve the self pointer to the given client. You can then either set attributes in the client or call methods in the client directly. For example:

```
MGetNumClients( self, &numClients );
if ( numClients >= 1 )
{
     for ( i = 0; i < numClients; i++ )
     {
        MGetNthClient( self, i, &client1 );
        if ( ( retValue = MError() ) != kMNoCompErr )
           break;

        // Do some work with the client
     }
}
```

## SERVICE METHOD REFERENCE

The following methods, documented in aChapter 4, “Service Reference”, are used in creating MOM services:

- “MGetGService, MGetPService” on page 4.85
- “MRegisterGService, MRegisterPService” on page 4.87
- “MUnregisterGService, MUnregisterPService” on page 4.86
- “MAddClient” on page 4.151
- “MDeleteClient” on page 4.161
- “MGetNumClients” on page 4.185
- “MGetNthClient” on page 4.184

## EDITOR CONSTRUCTION GUIDE

Some MOM components—notably modifiers—have a user interface associated with them. This user interface is commonly referred to as the "editor dialog" or "configuration dialog". It is a dialog that opens when the user double-clicks the icon of an instantiated modifier.

The editor dialog is where the user can interactively set the parameters of the modifier (as opposed to setting them programmatically, via Miniscript). Editor dialogs can have many different types of user interface elements, or "widgets," contained within them. To present a consistent interface to the user, you should use the editor dialog items that mFactory provides, and respond to them in the prescribed way. More information about user interface issues can be found in Chapter 5, "User Interface Guidelines".

Most of the code that is needed to handle the user's interaction with the interface elements is provided for you by the base editor class, which your component will automatically inherit. Your component only needs to provide support for the Editing methods that it needs (see "Editing" on page 3.56). What you need to do as a developer is lay out the editor dialog and save it as a named 'DLOG' resource, using the same name as you defined MComponentName with. The Core will open the dialog for you when the user double-clicks the component. Details on the layout of the editor dialog are provided in the following sections.

## CONSTRUCTING AN EDITOR DIALOG

Editor dialogs are constructed using a resource editing program, such as ResEdit or Resorcerer. At this time, mTropolis does not have a built-in dialog editor, so constructing a dialog requires placing static text descriptors on the dialog template directly. Keywords are used to specify the types of interface elements that will occupy the rectangles defined by these static text items during edit time.

mFactory will eventually provide a graphical dialog editor. Until then, this is the recommended method for constructing editors. This is the same method mFactory engineers use in-house—they share your pain.

### Components of an Editor Dialog

Figure 2.1 is an illustration of a typical editor window as it appears in Resorcerer.

Each object in the dialog is represented by a static text item which defines the item's boundary. The text within the item serves as a keyword specifier for the type of interface element that it represents, a list of which follows at the end of this section.

This particular editor has two buttons located in the lower right corner—the Accept and Decline buttons, labeled "OK" and "Cancel", respectively. The "Accept" and "Decline" labels attached to the resource specifiers indicate which method will be called in the component in response to the user selecting

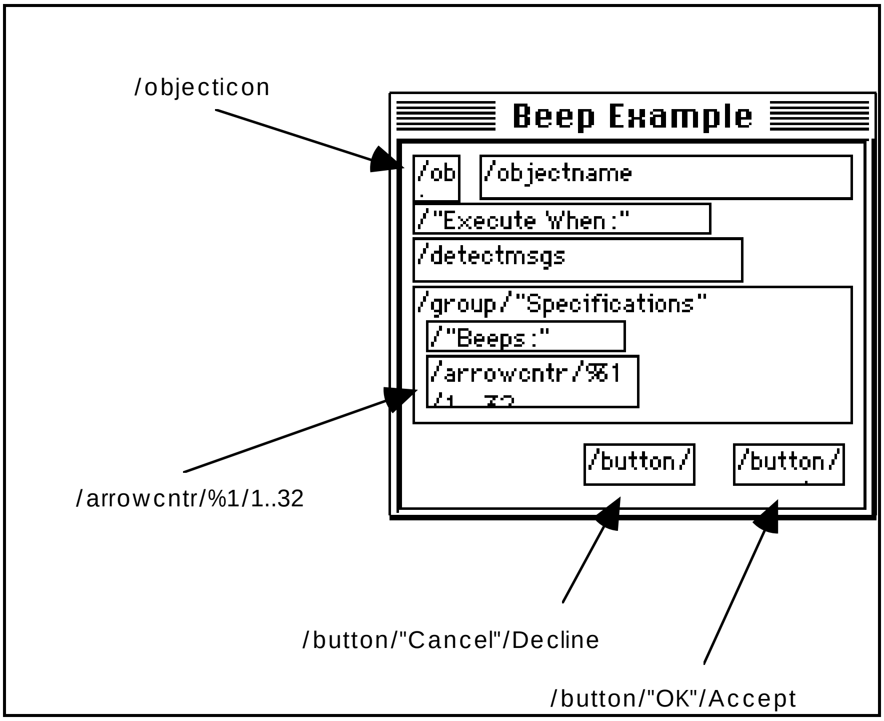


*Figure 2.1 A typical editor window shown here for the "Beep Example" modifier*

one of these items: MCompEditorAccept or MCompEditorDecline.

The editor also has an object icon specifier in the upper left, next to which is the object name. During runtime, these specifiers will be replaced with the icon and name of the object.

Next is an "Execute When" text label, under which is a "detectmsgs" specifier, indicating the presence of a special popup menu that lists the available messages that this component can respond to.

Underneath these items is a Group box specifier, which encloses two more objects—a text label and an "Arrow Counter", which is an element that has a text field and an up/down arrow control. This controller is used to manipulate numbers that have a "step" value associated with the arrow controls. The specifier includes the text "%1" and "1...32". The "%1" specifier indicates the step increment for the arrow controls. The "1...32" specifier indicates the range of allowed values in the item.

### More Interface Examples

The “Interface” example modifier contains a dialog that uses one of each of the interface elements described below, and is a good reference from which to copy and paste the widgets that you need. The source code for the interface example also illustrates how to set and retrieve data from these interface widgets.

Just about all of the other modifier examples also use dialog editors, and have simpler dialogs. The “Beep” example is a good place to start.

## EDITOR DIALOG KEYWORD REFERENCE

Below is a list of editor keywords, and samples of the interface elements they produce when the dialog is active.

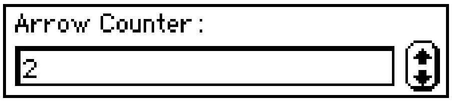
Also included for each keyword are additional parameter descriptions unique to that element, as well as the related MOM data type that corresponds to each type of element.

The argument specifiers described in Table 2.1 indicate special arguments that are included along with each keyword.

| Specifier | Purpose |
|---|---|
| `""` | Display enclosed text in the dialog as a static text item. |
| `# <command>` | |
| `%<arg1>` | Step value for fields that increment/ decrement in value |
| `<arg2>...<arg3>` | Field has an allowable numerical range from arg2 to arg3 |
| `$<f_value>` | Initial value for given field |
| `@<resource id>` | Resource ID for this field |
| `*<arg4>` | |
| `!<type specifier>` | |

*Table 2.1: Argument Specifiers*

### Arrow Counter



To create an arrow counter, define a static text item and set its text to the following:

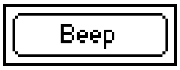
```
/arrowcntr/$2/0...100/%2
```

where $2 specifies an initial value of 2, 0...100 specifies a range between 0 and 100, and %2 specifies a step of 2.

MOM Data Type: MInteger

- *Note: You can get the same functionality from the “integer” item specifier.*

### Buttons



To create a button, define a static text item and set its text to the following:

```
/button/"Beep"
```

where "Beep" is the name of the button.

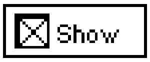
The keywords “accept”, “decline”, and “default” are used in conjunction with button specifiers to indicate buttons that act as the okay, cancel, and default buttons.

For example, to specify that the “Beep” button is to be the default button and should act as the dialog’s cancel button, use the following syntax:

```
/button/"Beep"/decline/default
```

See “Interface” example.

### Checkboxes



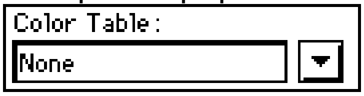
To create a checkbox, define a static text item and set its text to the following:

```
/checkbox/"Show"
```

where "Show" is the title given to the checkbox.

See “Interface” example.

### Color Look-Up Table Pop-Up



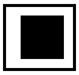
To create a CLUT file selection pop-up, define a static text item and set its text to the following:

```
/clutfiles
```

MOM Data Type: MSymbol

- *Note: MInitSymbol( self->f_clut, -1, 2 );   (2 is an identifier for CLUT files)*

### Color Swatch

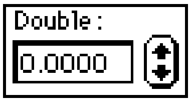


To create a color selection swatch, define a static text item and set its text to the following:

```
/colorspot
```

MOM Data Type: MRGBColor

### Double-Precision Fields



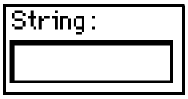
To create a double field, define a static text item and set its text to the following:

```
/double/0...999/%0.5
```

where 0...999 is the valid range of doubles and %0.5 specifies the arrow increment/ decrement step.

MOM Data Type: MDouble

### Edit Text



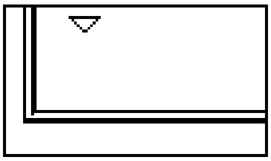
To create a string field, define a static text item and set its text to the following:

```
/edittext
```

MOM Data Type: MString

- *Note: The max length is specified when the string is initialized in the modifier code.*

**"Expando"**



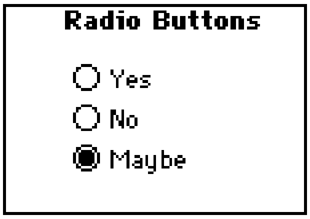
To create an expando, define a static text item and set its text to the following:

```
/expando
```

- *Note: All of the items enclosed by the expando's rectangle will be shown (or hidden) by clicking on the expando.*

Set "Interface" example.

**Grouped Items**

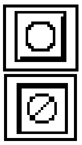


To create a group of items, define a static text item and set its text to the following:

/group/"Radio Buttons"

- *Note: The string that follows the "group" keyword is the title given to the group. For example, if the "Radio Button" group is to have the name "Are You Sure" the static text item should be set to /group/"Are You Sure". Radio button groupings are handled automatically by mTropolis when specified within a group item.*

**Icon Buttons**



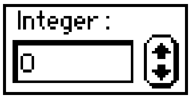
To create an icon button, define a static text item and set its text to the following:

```
/iconbutton/@1310/@1311
```

where 1310 is the "unhighlighted" state of the button icon and 1311 is the "highlighted" state of the button icon.

See "Interface" example.

**Integer Fields**



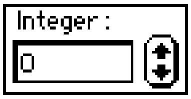
To create an integer field, define a static text item and set its text to the following:

```
/integer/-999...999/%1
```

where -999...999 is the valid range of integers and %1 specifies the arrow increment/ decrement step. If the step is not specified, the arrow counter is not displayed.

MOM Data Type: MInteger

**Integer Step Field**



To create an integer step field, define a static text item and set its text to the following:

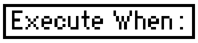
/integer/-999...999/%1

where -999...999 is the valid range of integers and %1 specifies the arrow increment/

decrement step. If the step is not specified, the arrow counter is not displayed.

MOM Data Type: MInteger

**Labels**



To create a label, define a static text item and set its text to the following:

```
/"Execute When:"
```

where the string in quotes that follows the / is the label string.

If you want to make the label bold, you should do the following:

```
/"Execute When:"/bold
```

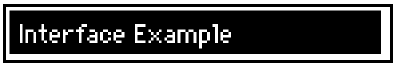
**Modifier icon**

To create a modifier icon item, define a static text item and set its text to the following:

```
/objecticon
```

The cicn resource with the resource name corresponding to the mod gets linked to this item.

**Modifier Name**

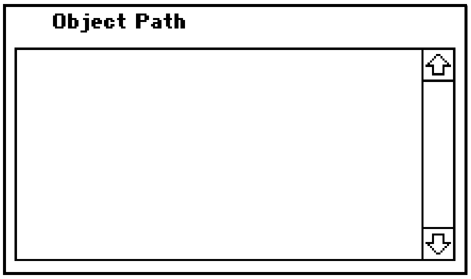


To create a modifier name edit text item, define a static text item and set its text to the following:

```
/objectname
```

The first string in the STR# resource corresponding to the mod gets inserted into the edit text box the first time the dialog is opened.

**File Path Edit Text**



To create a path item, define a static text item and set its text to the following:

```
/pathtext
```

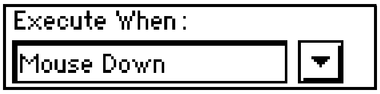
PathText items cause spaces and ':' to act as work breaks.

MOM Data Type: MString

## Pop-Up Items

Pop-ups have an identifier keyword associated with each type of pop-up so mTropolis know which type of list to display in the popup.

**Activate When**

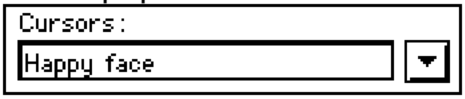


To create a pop up that displays the list of activate messages, define a static text item and set its text to the following:

```
/detectmsgs
```

MOM Data Type: MEvent

**Cursors Pop-Up**



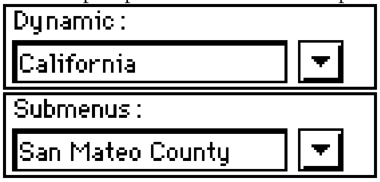
To create a cursor popup, define a static text item and set its text to the following:

`/cursorspopup`

MOM Data Type: MSymbol

- *Note: MInitSymbol( self->f_clut, 10001, 0 ); (This defaults your cursor to the first one in the list).*

Dynamic Pop-Ups and Submenu Pop-Ups



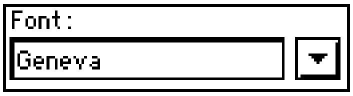
To create a pop up that displays a dynamic list of strings or a pop-up with submenus, define a static text item and set its text to the following:

`/popup`

See the "Interface" example to see how to initialize dynamic pop-ups and pop-ups with submenus.

MOM Data Type: MSymbol

**Fonts**

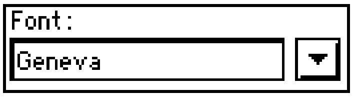


To create a pop up that displays a list of available fonts, define a static text item and set its text to the following:

`/fonts`

MOM Data Type: MSymbol

**Font Size**

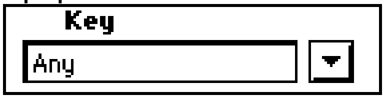


To create a pop up that displays a list of available font sizes, define a static text item and set its text to the following:

`/fontsize`

MOM Data Type: MSymbol

**Key Pop-up**

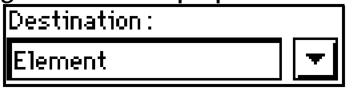


To create a pop-up that displays a list of keyboard chars, define a static text item and set its text to the following:

`/keys`

MOM Data Type: MSymbol

**Message Destination Pop-Up**



To create a pop up that displays a message destination list, define a static text item and set its text to the following:

`/targets`

MOM Data Type: MTargetType

**Message/Command Pop-Ups**

To create a pop-up that displays a list of messages and commands to send, define a static text item and set its text to the following:

```
/sendmsgs
```

MOM Data Type: MEvent

**Sound Pop-Ups**

To create a pop up that displays a list of sounds and lets you link to sound files, define a static text item and set its text to the following:

```
/soundfiles
```

MOM Data Type: MSymbol

- *Note: MInitSymbol( self->f_sound, -1,0 ); (This defaults your sound to the first one in the list)*

**Scene, Subsection, and Section Pop-Ups**

To create a pop up that displays a list of scenes, define a static text item and set its text to the following:

```
/scenes
```

MOM Data Type: MIDPath

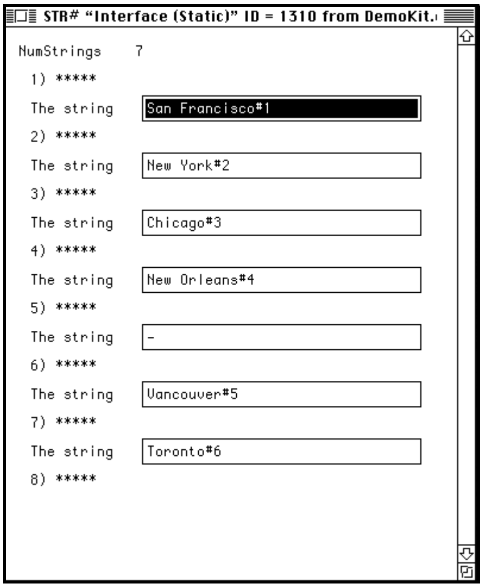
Sections and subsections are also setup with the /sections and /subsections keywords. See the Change Scene modifier example to see how section, subsection, and scene pop-ups are setup and updated during runtime.

**Static Pop-Ups**

To create a pop up that displays a list of static strings, define a static text item and set its text to the following:

```
/popup/@1310
```

where 1310 is the id of the STR# resource to display in the static popup. Each string in the STR# resource can be followed by an identifier to associate unique ids with each string. For example, the following STR# resource:



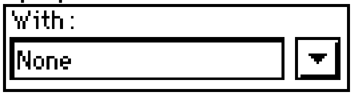
STR# "Interface (Static)" ID = 1310 from DemoKit.

```
NumStrings    7
  1) *****
  The string   San Francisco#1
  2) *****
  The string   New York#2
  3) *****
  The string   Chicago#3
  4) *****
  The string   New Orleans#4
  5) *****
  The string   -
  6) *****
  The string   Vancouver#5
  7) *****
  The string   Toronto#6
  8) *****
```

associates the ID 4 with the string "New Orleans".

MOM Data Type: MSymbol

**With Pop-Ups**

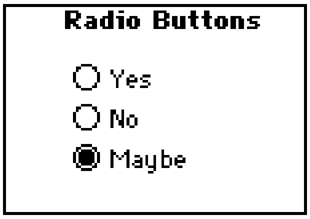


To create a pop up that displays a list of data to send, define a static text item and set its text to the following:

```
/withdata
```

MOM Data Type: MDataType

**Radio Buttons**



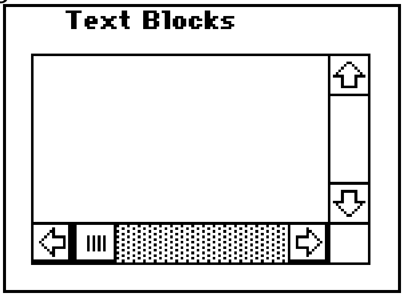
To create a radio button, define a static text item and set its text to the following:

```
/radio/"Maybe"
```

where "Maybe" is the title given to the radio button.

- *Note: Radio button groupings are handled automatically by mTropolis when radio buttons are defined within a "radiogroup" or "group" item.*

**Large Text Blocks**

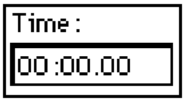


To create a text block field, define a static text item and set its text to the following:

```
/text
```

MOM Data Type: MString

**Time Fields**



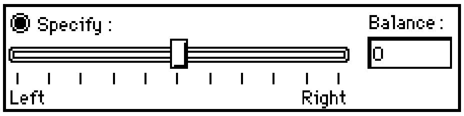
To create a time field, define a static text item and set its text to the following:

```
/timefield/$300
```

where $300 is supposed to initialize the time value.

MOM Data Type: MTime

**Sliders**



To create a value slider like the one in the Sound Panning modifier, define a static text item and set its text to the following:

```
/valslider/-100...100/$0
```

where -100...100/$0 specifies the range of valid slider values and the slider's initial value.

You set up a value slider by associating an MInteger field with the slider dialog item.

In MCompConstructor, initialize the slider value as follows:

```
MInitInteger(self->f_sliderValue, 0 );
```

In MCompEditorOpen you'd do something like this:

```
MSetEditorItem( editor, kSliderValue,
  ( MDataType * )
   &self->f_sliderValue );
MSetEditorItem( editor, kSlider, ( MDataType *) &self->f_sliderValue );
```

where kSlider represents the slider dialog item.

In MCompEditorItemChanged you have to keep the integer value and the slider in sync as follows:

```
 else if( item == kSliderValue ) {
      MGetEditorItem( editor, kSliderValue, ( MDataType * ) &self->f_sliderValue
);
      MSetEditorItem( editor, kSlider, ( MDataType *) &self->f_sliderValue );
 }
 else if( item == kSlider) {
      MGetEditorItem( editor, kSlider, ( MDataType * ) &self->f_sliderValue );
      if( self->f_sliderValue.f_value > kMaxSliderValue ) {
         self->f_sliderValue.f_value = kMaxSliderValue;
         MSetEditorItem( editor, kSlider, ( MDataType *) &self->f_sliderValue );
      }
      if( self->f_sliderValue.f_value < kMinSliderValue ) {
         self->f_sliderValue.f_value = kMinSliderValue;
         MSetEditorItem( editor, kSlider, ( MDataType *) &self->f_sliderValue );
      }
      MSetEditorItem( editor, kSliderValue, ( MDataType *) &self->f_sliderValue
);
 }
```

MOM Data Type: an MInteger that gets linked to your slider.

## IMPLEMENTING mTOOLS

This section discusses the creation of mTools. mTools are different from other MOM components in that they affect the mTropolis editor itself, not titles, like other conventional MOM components (e.g., modifiers). Using mTools, you can enhance the capabilities of the editor by adding new features or modifying existing features.

“MOM mTool Services” on page 4.256 contains information about the various methods your component can call and override to extend the functionality of the mTropolis editor.

## WHAT ARE mTOOLS?

mTools are MOM components that perform their work during edit time (as opposed to title runtime). These components can add menu items, get information about the current project, and otherwise manipulate the editing environment.

mTools are constructed just like any other MOM component, and have a set of methods that need to be overridden to perform their jobs. mTools are instantiated when the user chooses the menu item associated with that mTool. Generally speaking, you’ll want to have a separate mTool for each menu item you install. When the editor starts up, your component’s MClassStaticInit() function will be called; this is a good point to add your mTool menu entries to mTropolis.

## WHAT KINDS OF THINGS CAN mTOOLS DO?

In the current release of MOM, mTools can add items to mTropolis menus, invoke other mTropolis menu items, and retrieve a list of the currently selected objects in the Layout window. The selection list cannot currently be changed by mTools; they may only query the selection and act upon the selected objects.

However, you can perform just about any kind of action on the selected objects using attributes, as you would at runtime. You can also affect the project itself in the same way.

In future releases, mTools will have many more capabilities.

## mTOOL USER INTERFACE

Currently, mTools can add menu items to the mTropolis menus, and may display modal dialogs. In the future, mTools will have full user interface parity with the host editor. They will be able to have floating palettes, add their own menus to the menu bar (along with menu items in the standard mTropolis menus), and display different types of windows.

See the mTool examples and “MOM mTool Services” on page 4.256 for information on how to build editor-based tools.

# Chapter 3. Basic MOM Technologies

This chapter describes the basic technologies used by the mFactory Object Model.

## COMPONENT ATTRIBUTES

Component attributes can be thought of as “member” variables of an object, or “properties” of an object. For example, a component that beeps may define as its attributes the volume of the beep, what the beep sound is, how long it plays, and how many times it beeps. Each of these attributes can be queried and set by the user (using Miniscript or the modifier’s editing window) or by other components (by making direct attribute calls).

Attributes are defined as unique lowercase 16-byte sequences called MOM IDs (see “The MOM ID” on page 2.17). Internally, MOM IDs are case-sensitive, and you may mix cases for your attribute names. However, since the Miniscript modifier uses only lowercase, you should be sure that any attributes that the user can access via miniscript are defined to be lowercase.

### How are Attributes Used?

Users can write Miniscript code that accesses the attributes of an object. They can also directly manipulate attributes using a modifier’s editing dialog. This is how users select the initial properties of your component, and how they change them during runtime.

Other components may also want to know about your component’s attributes. For example, the Behavior modifier needs to know what events a component responds to and sends so that it may correctly draw the “event lines” in its window.

- *Note: Attributes are provided mainly as an interface for Miniscript and for dynamic resolution of component-to-component communication. It is much faster for components to directly call each others’ methods, though this carries the restriction of knowing ahead of time what methods a component supports.*

Some attributes are common and described throughout the MOM SDK in the various *Attrb.h files. Your component can use these attributes to request data from other components and the elements to which they are attached.

- *Note: Modifier MOM components must override MCompGetAttribute if they support an execution or termination event, so that the Behavior modifier will be able to correctly draw the event lines in its window.*

Your component can easily implement these behaviors using only five method overrides. Component creators are strongly encouraged to support all of the attribute methods so that other components can learn about the attributes supported by your components.

**Related Information**
Refer to the sample code to see how attributes are retrieved and set.

### Methods to Override

Your component should override the following methods.

- "MCompGetAttribute" on page 4.176: return the requested attribute
- "MCompSetAttribute" on page 4.195: set your internal data to the given data
- "MCompGetAttributeCount" on page 4.178: return the number of attributes your component understands
- "MCompGetNthAttributeName" on page 4.183: get the name of your component's *n*th attribute
- "MCompGetAttributeMaxIndex" on page 4.179: return the maximum index of the requested attribute

### Methods You Can Call

The following methods can be called to retrieve or set attributes:

- "MGetElementAttribute" on page 4.71
- "MSetElementAttribute" on page 4.72
- MGetAttribute(): callable version of MCompGetAttribute
- MSetAttribute(): callable version of MCompSetAttribute

## COMPONENT PERSISTENCE

This section describes how to save and restore your component's data using the project stream.

MOM Components can save information into and restore information from the project file using the Component Persistence methods. For example, the settings that the user enters into your component's editing window (if it is a modifier) need to be saved along with the project so that it will behave properly at runtime and so that the author can modify settings in the future.

Implementing persistence is not difficult. You need to override three methods, and the methods that you use to read and write data automatically handle the necessary size calculations (as long as you use MDataTypes).

#### Related Information

Many of the example MOM Component kits use these methods.

### Methods To Override

Your component needs to override these three methods in order to implement persistence:

- "MCompGetSaveInfo" on page 4.186
- "MCompSaveComponent" on page 4.194
- "MCompRestoreComponent" on page 4.192

### Methods You Will Need To Call

Your component will need to call these methods to read, write, and calculate the size of its data:

- "MSizeOfValue" on page 4.111
- "MWriteValue" on page 4.115
- "MReadValue" on page 4.110
- "MWriteBytes" on page 4.112
- "MReadBytes" on page 4.107
- "MWriteInt16" on page 4.113
- "MReadInt16" on page 4.108
- "MWriteInt32" on page 4.114
- "MReadInt32" on page 4.109

## EDITING

This section describes the steps needed to implement an Editing dialog in your MOM Component. Most likely, an Editing window will be used by a modifier or service (if it has a user interface).

You should read this section if you want users to be able to manipulate your component's settings using a standard mFactory editing dialog.

### How Users Interact with your Component

Some MOM Components, such as modifiers and services that have user interfaces associated with them, need to provide a way for users to manipulate their settings. For example, the Graphic modifier needs a way to allow the user to set the color, ink type, "Apply When" and "Remove When" events, etc.

Modifiers can be double-clicked by the user, which displays an editing window in which they can manipulate the component's settings. These windows can contain edit fields, popup menus, "expando" controls (which grow and shrink the window), and other types of controls.

Much of the code to handle all of this is already implemented for you, such as displaying a dialog, operating the controls, and handling user input. Your component basically needs to handle setting up and retrieving information from the dialog, and validating the user input.

#### Related Information

You should read Chapter 5, "User Interface Guidelines" for a more detailed discussion of user interface issues involved with creating components. You should also read "Editor Construction Guide" on page 2.42, which provides details on creating editor dialogs and the different types of controls that they can contain.

The "Interface" example modifier provides a dialog example that contains one of each of the different kinds of interface elements. The sample code also provides examples of how to use each of the methods listed in this reference.

### Methods To Override

There are five methods that your component can override in order to implement the editing window interface. You may not need to override all of them; many components can function simply by overriding the first two.

- "MCompEditorOpen" on page 4.171: The editor window is being opened. Set the contents now.
- "MCompEditorAccept" on page 4.165: The user clicked the OK button.
- "MCompEditorDecline" on page 4.168e: The user clicked the Cancel button
- "MCompEditorItemChanged" on page 4.169: An editor dialog item has changed somehow.

- "MCompEditorPrevalidateItem" on page 4.172: An item needs to have its contents validated.

### Utility Methods

There are several methods that you can use to access information in the dialog.

- "MGetEditorItem" on page 4.94: retrieve data from a dialog item
- "MSetEditorItem" on page 4.99: set a dialog item with the given data
- "MAddSymbol" on page 4.91: add an item to a hierarchical menu
- "MSetEditorItemState" on page 4.100: enable or disable a dialog item
- 
- MSetEditorSubItemState: enable or disable a hierarchical menu item
- 
- "MSetItemTitle" on page 4.103: set a dialog item title
- "MGetItemTitle" on page 4.96: get a dialog item title
- "MSetItemSpec" on page 4.102: set information related to a dialog control
- "MGetItemSpec" on page 4.95: get information related to a dialog control
- "MForgetSymbols" on page 4.93: tell a hierarchical menu to release its symbol information
- "MGetSymbolName" on page 4.97: get the name of a given symbol
- "MSetSymbolName" on page 4.104: set the name of a given symbol
- "MGetSymbolSelection" on page 4.98: get the current symbol selection
- "MAddSubSymbolItem" on page 4.92: add a symbol to a submenu
- "MForceDialogUpdate" on page 4.105: force the editor dialog to update its appearance

#### Related Information

Several of the sample components, such as the "Interface" example, implement editors.

### Where To Go From Here

Read "Editor Construction Guide" on page 2.42, which illustrates the different types of controls that an editor dialog can contain and discusses the technical implementation details of editor dialogs.

## MESSAGING

In its simplest form, messaging is the ability of one component to communicate with another one by sending it a "message," which may or may not be associated with additional data.

It is this ability which gives mTropolis much of its power, since objects do not need to be aware of how other objects behave. They can simply send them a message just like any other object, and if the target is prepared to deal with the message, then it can take its action.

There are several different types of messages, from simple mouse events to author-defined messages to project-level messages such as "Project Started."

Messaging in mTropolis is not a complicated proposition. Your component only needs to override one method to process incoming messages, and sending messages to other objects usually requires only a few method calls.

### Specifying a Message Target

In edit mode, the user specifies a target for a message by choosing it from a pop-up menu in a modifier's editing window. When the user closes the window, the modifier executes its MCompEditorAccept function, and calls MGetEditorItem() for the item of the popup menu. The data returned for a target pop-up menu is usually an MTargetType (a form of MDataType).

### Creating the Message

Messages are most often created using the MInitMessage() method, which has this syntax:

```
MInitMessage(MSelf *self,
  MTargetType *target, Mevent *event,
  MDataType *withData,
  long messageInfo, Mmessage *msg);
```

You must supply the target and event parameters. The withData parameter is optional, set it to NULL if you do not send any data with the event. The messageInfo field is a set of bit flags that hold information about the message. You can use the predefined constants `kMImmediateMessageMask`, `kMCascadeMessageMask`, and `kMAbsorbMessageMask`. These options are used to implement the three check boxes found at the bottom of all mTropolis messengers. The last parameter is a pointer to a message structure that is filled in for you by the Core. You can then pass that structure to MSendMessage.

### Sending the Message

The MSendMessage method is used to dispatch the message that has been set up via MInitMessage. It has this syntax:

```
MSendMessage(Mmessage *msg);
```

Typical usage is to first call MInitMessage, then call MSendMessage with the result as follows:

```
MMessagemessage;
if ( self->f_msgCmdEvent.f_event != kMNone )
{
MInitMessage( self, &self->f_target, &self->f_msgCmdEvent,
&self->f_withData, self->f_msgFlags.f_value, &message );
  MSendMessage( &message );
  return MError();
}
return kMNoCompErr;
```

### How Do I Process an Incoming Message?

If your component can respond to messages, it should override the MCompProcessMessage method. Some components respond to more than one message. They may have an event on which they enable themselves and another on which they disable themselves.

Your component should call MDetectMessage() to see if the incoming message is one that your component responds to, as shown below:

```
// if you respond to more than one message, then call MDetectMessage
// for each one.
if ( MDetectMessage( message, self->f_executeEvent ) )
{
  MSendMessengerMessage( self );
  return MError();
}
else return kMUnableToComplyCompErr;
```

### Methods To Override

The following method should be overridden to provide messaging support:

- “MCompProcessMessage” on page 4.188: this method is called when your component receives a message, either in runtime or in edit mode.

### Methods To Call

The following methods can be called to process and dispatch messages:

- MDetectMessage
- “MInitMessage” on page 4.76: call to initialize a message structure with a target, event, data, and sending information.
- “MSendMessage” on page 4.78: call to actually send a message.

### Where to Go From Here

The Messenger sample code included with the MOM SDK demonstrates an example of a typical messenger modifier which uses the

MInitMessage and MSendMessage to send messages to other objects, and uses the MCompProcessMessage method to process incoming messages.

Component Attributes

## mFUNKS

This section describes mFunks, which are special types of callback functions that allow you to “splice” methods from one class onto another. This is not multiple inheritance; it is simply a way for a class to provide access to its internals to other “helper” methods.

MFunks can be turned on and off at will by your component. When an mFunk is registered, it can be called by the mTropolis core to perform its work. To turn it off, your component simply unregisters the mFunk. MFunks are maintained in lists, and their effects can be cumulative. Any given object may have one or more mFunks associated with it.

MFunks are also aware of inheritance. If a base class “A” defines a method called “MyFunkMethod”, and descendant class “B” overrides “MyFunkMethod”, then class B’s MyFunkMethod will be called. Class B’s method can then call the superclass’ method to augment its original behavior.

### Defining and Using mFunks

This section discusses mFunk definition and usage, and introduces the methods and data structures necessary to handle mTropolis’ built-in “buffered drawing” mFunks.

To define an mFunk, your component needs to provide a method to handle the mFunk, and then call MInitFunk() to set up the MFunk data structure. The MFunk data structure and MInitFunk macro are defined as follows:

```
typedef struct MFunk {// That MFunky Data Structure!
  shortf_code;  // type of MFunk you are registering
  void*f_comp;  // self pointer of your component
  shortf_slot;  // your component’s slot
  short f_index;// index of method to handle the MFunk
  long f_ID;       // ID number of this particular Funk
  long f_data;  // User-defined refcon for this Funk
  long f_peckingOrder;// Where this Funk stands relative to its peers
} MFunk ;
MFunk theFunk;
MInitFunk (MFunk theFunk, short funkType, MSelf *self, short compSlot, short
funkMethod, long funkID, long funkData, long peckingOrder);
```

For the funkType parameter, you should pass `kMPostDrawBufferedFunk`, `kMPreDrawBufferedFunk`, or `kMDrawBufferedFunk`, depending on the mFunk type you want to handle. The self parameter should be your component’s MSelf pointer. The compSlot and funkMethod parameters denote your component’s slot and the index of the method that will handle the mFunk call. Your component will typically pass 0 as the funk ID. MOM will fill this value in for you when MRegisterFunk() is called;

you need to pass it to MUnregisterFunkByID. The f_data field you can use to store any special data you wish. The f_peckingOrder field indicates where you want to MFunk to be registered in the list as it currently stands (note that subsequent MFunk registrations will alter the list).

- *Note: The f_peckingOrder field of MFunk data structure is currently unused, but will be supported in future releases of MOM.*

The “DrawFunk” example modifier provides an illustration of implementing all three mFunks.

### Registering the mFunk

Your component should call MRegisterFunk() when it wants the mFunk to become active and able to respond to the callback:

```
MRegisterFunk(MObjectRef *element,
  MFunk *theFunk,
  MDataType *initialData);
```

The “element” parameter should be a pointer to the object to register the funk with. This will typically be obtained from MGetElement(). The “theFunk” parameter should be a pointer to the MFunk structure that was initialized with MInitFunk(). The initialData parameter is any initial data you need to initialize the mFunk with, if the mFunk you are registering supports initial data (the drawing mFunks currently do not).

Your component will typically call this method when it receives a message that it knows how to process (that is, MDetectMessage() has returned TRUE).

### Unregistering the mFunk

To unregister the mFunk, your component calls MUnregisterFunkByID():

```
MUnregisterFunkByID(MSelf *self,
  short funkType, long funkID);
```

The “self” parameter should be your component’s self pointer. The “funkType” parameter should be the type of mFunk that you are unregistering (`kMPostDrawBufferedFunk`, `kMPreDrawBufferedFunk`, or `kMDrawBufferedFunk`). The funkID parameter should be the ID of the mFunk that you are unregistering, which was passed back in the MFunk structure when MRegisterFunk() was called.

Your component will typically call this method when it is disabled (MCompDisabled() method is called), when it is destroyed (MCompDestructor() method is called), or when it is disconnected from its parent (MCompDisconnect() method is called). You will also typically unregister an mFunk when your component detects a disable message in it MCompProcessMessage() method is called and the MDetectMessage() macro returns TRUE when called with your disable event (or MDetectDisableMessage() returns TRUE).

### Currently Defined mFunks

Currently, there are three defined mFunks for use by MOM components (more will be exposed in the future), and all three are related to drawing elements: the PreDrawBuffered, PostDrawBuffered, and DrawBuffered mFunks. Each of these mFunks is discussed in detail below.

**Buffered Drawing**

When mTropolis draws an element on the screen, your component may want to modify the draw behavior to suit its needs. For example, you may want to add some type of graphic adornment to an element after it has been rendered. You might also want to handle the drawing entirely yourself. The buffered drawing mFunks make this possible. PICT, QuickTime, and mToon elements all support drawing mFunks.

The three mFunks denote the times at which your component may affect the drawing process. For example, the "Pre-" drawing mFunk is called before any drawing actually takes place. The "Draw" mFunk is called when an element would normally be drawn into the offscreen buffer by the Core. Your component may draw its own contents into the offscreen buffer, or it may affect the transfer of the asset's pixel data into the offscreen buffer. The "Post" drawing mFunk is called after the element has been drawn into the offscreen. At this point, your mFunk may do any additional drawing it wants to.

- *Note: mToon elements that use QuickTime compression codecs do not support the MCompDrawBuffered mFunk. Also, these mFunks do not work for direct-to-screen elements.*

## MCompPreDrawBuffered

```
MCompPreDrawBuffered(MSelf
   *self,PLUpdateSpec*
   drawSpec, PLOffScreenSpec
   *gworldSpec);
```

This method is called when your component has registered a drawing mFunk of type kMPreDrawBufferedFunk, and the element that the mFunk is registered with is about to begin drawing.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| drawSpec | Platform-specific drawing geometry information |
| gworldSpec | Platform-specific drawing port information |

When this method is called, your component will be passed two pieces of platform-specific information: a PLUpdateSpec and a PLOffScreenSpec. These structures are defined in MTYPE.H. The drawSpec will be set up with geometry information including the rectangle that defines the element. The gworldSpec will contain information about the graphics port that is being drawn into. This is the pixel map that will be transferred to the screen when all the drawing is done.

## MCompDrawBuffered

```
MCompDrawBuffered(MSelf *self,
   PLUpdateSpec *drawSpec,
   PLOffscreenSpec
   *gworldSpec, PLPixMapSpec
   *pixSpec);
```

This method is called when your component has registered a drawing mFunk of type kMDrawBufferedFunk, and the element that the mFunk is registered with would normally do its own drawing. Your component can either draw the contents of the element itself, or use the pixel data provided in the “pixSpec” parameter to draw the asset its own way.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| drawSpec | Platform-specific drawing geometry information |
| gworldSpec | Platform-specific drawing port information |
| pixSpec | Platform-specific image information about the asset’s pixel data |

When this method is called, your component will be passed three pieces of platform-specific information: a PLUpdateSpec, a PLOffScreenSpec, and a PLPixMapSpec. These structures are defined in MTYPE.H. The drawSpec will be set up with geometry information including the rectangle that defines the element. The gworldSpec will contain information about the graphics port that is being drawn into. This is the pixel map that will be transferred to the screen when all the drawing is done. The pixSpec will contain platform-specific data about the pixels of the asset associated with the element.

## MCompPostDrawBuffered

```
MCompPostDrawBuffered(MSelf
   *self,PLUpdateSpec*
   drawSpec, PLOffScreenSpec
   *gworldSpec);
```

This method is called when your component has registered a drawing mFunk of type kMPostDrawBufferedFunk, and the element that the mFunk is registered with has finished its drawing. At this point, your mFunk can do any additional drawing that it needs to do.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| drawSpec | Platform-specific drawing geometry information |
| gworldSpec | Platform-specific drawing port information |

## SUPPORTING MINISCRIPT

Miniscript is mFactory's scripting language. Its purpose is not to act as a full-blown scripting language, because the real power of mTropolis lies in its drag-and-drop objects. Miniscript is mainly intended to be a command language, in which several commands can be easily implemented in script. It also provides ready access to the attributes of a given object, which otherwise may be too numerous to list in a dialog.

### It's Automatic!

The user can create Miniscript components and use them to access the attributes in your component. If you have already implemented attributes in your component via the MCompSetAttribute and MCompGetAttribute methods (see "Component Attributes" on page 3.53), then you do not need to do anything further to support Miniscript—mTropolis takes care of this for you.

There are just a few things to consider when supporting attributes to make Miniscript work.

#### Use Lowercase Attribute Names

Miniscript processes text in all lowercase, and since attributes are case-sensitive, they need to be in lowercase in order for them to work with Miniscript.

#### Use Descriptive Attribute Names

Attribute names should be descriptive enough that the user will remember them. Names should also correspond to their related controls in your component's editing window, if it has one.

#### Provide Full Attribute Support

Except for attributes that you wish to keep private from users, you should expose all of your attribute names so that users can access them.

Component Attributes

# Chapter 4. Service Reference

This chapter contains reference documentation for all available MOM services. A master list of all documented MOM routines can be found in Chapter 8, “Alphabetical List of MOM Routines”.

Documented services include:

## MOM COERCION SERVICE

The MOM Coercion Service provides a method to convert from any MOM data type to another MOM data type.

## MCoerceType

```
MCoerceType( MSelf *self,
   MDataType *intype, MDataType
   *outtype )
```

MCoerceType coerces the MOM data type supplied in intype to the MOM data type given in outtype.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| intype | the MOM data type that needs to be coerced |
| outtype | the new data type |

## MOM COMPONENT SERVICES

The MOM Component Service provides a variety of services to components. The following table lists the Component Service methods for each functional area.

| Functional Area | Methods |
|---|---|
| Elements | MSetElementAttribute, MGetElementAttribute |
| mFunks | MRegisterFunk, MUnregisterFunkByID |
| Messaging | MInitSimpleMessage, MInitMessage, MSendMessage |
| MDataType Handling | MCopyData, MDisposeData, MInitData2Type |
| Resource Handling | MGetKitRef |
| Class Utilities | MCountClasses, MGetClassInfoByIndex |
| Services | MRegisterGService, MUnregisterGService, MGetGService, MRegisterPService, MUnregisterPService, MGetPService. |

## MGetElementAttribute

```
MGetElementAttribute(MObjectRef
    *elemRef, MomID attribName,
    MDataType *selector, MDataType
    *dataValue )
```

MGetElementAttribute retrieves an attribute of an element.

**Parameters**

| | |
|---|---|
| elemref | an element |
| attribName | the attribute name |
| selector | additional info to select the attribute |
| dataValue | the attribute value to return |

**Example**

```
static void getElementAttrib(YourComp *self)
{
 MObjectRef   *element;
 MPoint2D     elemsize;

 MInitPoint2D( elemsize, 0, 0 );
 MGetElement( &element );
 MGetElementAttribute( element, kMSizeAttrib, NULL, (MDataType *)&elemsize );
}
```

**See Also**

“MGetElement” on page 4.180

## MSetElementAttribute

```
MSetElementAttribute( MObjectRef
    *elemRef, MomID attribName,
    MDataType *selector, MDataType
    *dataValue )
```

MSetElementAttribute sets an attribute of an element.

**Parameters**

| | |
|---|---|
| elemref | the element |
| attribName | the attribute name |
| selector | additional information in selecting attribute |
| dataValue | attribute value to return |

**Example**

```
static void getElementAttrib(YourComp *self)
{
 MObjectRef   element;
 MPoint2D     elemsize;

 MInitPoint2D( elemsize, 128, 128 );
 MGetElement( &element );
 MSetElementAttribute( element, kMSizeAttrib, NULL, (MDataType *)&elemsize );
}
```

**See Also**

"MGetElementAttribute" on page 4.71

## MRegisterFunk

```
MRegisterFunk( MObjectRef
    *objRef, MFunk* theFunk, void
    *initialData )
```

MRegisterFunk registers an mFunk with an object.

**Parameters**

| | |
|---|---|
| objRef | the object to register the mFunk with |
| theFunk | the initialized funk to be registered to the element |
| initialData | any initial data required by the object |

**Example**

```
if ( MDetectMessage(message, self->f_applyEvent) )
{
      MFunk funk;
      MObjectRef*element;

      MInitFunk(funk,kMDrawBufferedFunk,self,kDrawBitsSlot,MDrawBuffered);
      MGetElement(&element);
      MRegisterFunk(element, &funk, NULL);
      return kMNoCompErr;
}
```

**See Also**

"MUnregisterFunkByID" on page 4.74

## MUnregisterFunkByID

```
MUnregisterFunkByID( MObjectRef
   *objRef, short funkCode, long
   funkID )
```

MUnregisterFunkByID unregisters an mFunk having a given type and ID number from the object.

**Parameters**

| | |
|---|---|
| elemRef | the element to which the funk has been registered |
| funkCode | the type of the mFunk. A list of these can be found in MPlayer.h |
| funkID | ID number of the funk. This is stored in the Mfunk data structure. |

**Example**

```
MFunk funk;

if (MDetectMessage(message, self->f_removeEvent||
MDetectDisableMessage(message) )
{
      MObjectRef*element;

      MGetElement(&element);
      MUnregisterFunk(element, kMBufferedDrawFunk, funk.ID);
      return kMNoCompErr;
}
```

**See Also**
“MRegisterFunk” on page 4.73

## MInitSimpleMessage

```
MInitSimpleMessage( MSelf *self,
   MObjectRef *target, MEvent
   *event, MMessage *message)
```

MInitSimpleMessage constructs a mTropolis message structure using the given target as the recipient of the message, event as the message to be sent.

**Parameters**

| | |
|---|---|
| self | your component |
| target | the recipient of the message |
| event | the message or event id and info |
| message | mTropolis message structure that is used to send a message |

**Example**

```
static MErr MCompProcessMessage(MomentumComp *self, MMessagePtr message)
{
 if ( MDetectMessage(message, self->f_enableEvent) )
 {
      // send kMHide to my element
      MErr     ret;
      MEvente;
      MObjectRef*element = nil;
      MMessagem;

      MInitEvent(e, kMHide, 0);
      MGetElement(&element);
      if (MError() != kMNoCompErr || !element)
         return kMUnableToComplyCompErr;
      MInitSimpleMessage(self, element, &e, &m);
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      MSendMessage(&m);
      return MError();
 }
 // ...
}
```

**See Also**

"MInitMessage" on page 4.76

"MSendMessage" on page 4.78

## MInitMessage

```
MInitMessage( MSelf *self,
   MObjectRef *target, MEvent
   *event, MDataType *data, long
   options, MMessage *message)
```

Like MInitSimpleMessage, MInitMessage constructs a mTropolis message structure using the given target as the recipient of the message, event as the message to be sent, data as the message “with” data, and options as the messaging sending options.

**Parameters**

| | |
|---|---|
| self | your component |
| target | the recipient of the message |
| event | the message or event id and info |
| data | the message with |
| options | the messaging options<br>default is 0, defined masks in mevent.h are:<br>kMImmediateMessageMask<br>kMCascadeMessageMask<br>kMRelayMessageMask |
| message | mTropolis message structure that is used to send a message |

**Example**

```
static MErr MCompProcessMessage(MomentumComp *self, MMessagePtr message)
{
  if ( MDetectMessage(message, self->f_enableEvent) )
  {
      // send kMHide to my element
      MErr     ret;
      MEvent   e;
      MObjectRef*element = nil;
      MDataTyped;
      MMessage m;

      MInitEvent(e, kMHide, 0);
      MGetElement(&element);
      if (MError() != kMNoCompErr || !element)
         return kMUnableToComplyCompErr;
      MInitInteger(d.f_integer, 2345);
      MInitMessage(self, element, &e, &d, 0, &m);
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      MSendMessage(&m);
      return MError();
  }
  // ...
}
```

**See Also**

“MInitSimpleMessage” on page 4.75

“MSendMessage” on page 4.78

## MSendMessage

`MSendMessage(MMessage *message)`

MSendMessage sends a message constructed by either MInitMessage or MInitSimpleMessage.

**Parameters**

| | |
|---|---|
| message | the message constructed by MInitSimpleMessage or MInitMessage |

**Example**

```
static MErr MCompProcessMessage(MomentumComp *self, MMessagePtr message)
{
  if ( MDetectMessage(message, self->f_enableEvent) )
  {
        // send kMHide to my element
        MErr     ret;
        MEvent   e;
        MObjectRef*element = nil;
        MMessage m;

        MInitEvent(e, kMHide, 0);
        MGetElement(&element);
        if (MError() != kMNoCompErr || !element)
           return kMUnableToComplyCompErr;
        MInitSimpleMessage(self, element, &e, &m);
        if ((ret = MError()) != kMNoCompErr)
           return ret;
        MSendMessage(&m);
        return MError();
  }
  // ...
}
```

**See Also**

"MInitMessage" on page 4.76
"MInitSimpleMessage" on page 4.75

## MCopyData

```
MCopyData(MDataType *dest,
   MDataType *source)
```

MCopyData is used to copy an arbitrary MDataType.

**Parameters**

| | |
|---|---|
| dest | destination MDataType |
| source | source MDataType |

## MDisposeData

```
MDisposeData(MSelf *self,
   MDataType *data)
```

MDisposeData is used to dispose an arbitrary MDataType.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| data | the MDataType to dispose |

## MInitData2Type

```
MInitData2Type(MDataType *data,
   short thetype)
```

MInitData2Type will initialize an arbitrary MDataType to a given type and zero out its fields.

**Parameters**

| | |
|---|---|
| data | MDataType to initialize |
| thetype | type to initialize the MDatatype with |

## MGetKitRef

```
MGetKitRef(CompID *theID,
   MFFileRef *resRef)
```

On Macintosh systems, returns the resource file number of the component's kit. On Windows, returns the HINSTANCE of the component DLL.

**Parameters**

| | |
|---|---|
| theID | CompID of the component |
| resRef | Mac: a short, representing the resource file number. Windows: HINSTANCE of the DLL. |

## MCountClasses

```
MCountClasses(short *count, short
   *numInternal)
```

MCountClasses counts the number of MOM classes currently loaded in mTropolis.

**Parameters:**

| | |
|---|---|
| count | total number of MOM classes |
| numInternal | number of MOM classes which are internally defined |

**See Also**

"MGetClassInfoByIndex" on page 4.84

## MGetClassInfoByIndex

```
MGetClassInfoByIndex(short indx,
   MFCompClassInfo *classInfo)
```

Retrieves the class information for the class at a given index.

**Parameters:**

| | |
|---|---|
| indx | index of the MOM class to retrieve data for |
| classInfo | record containing information about the MOM class |

**See Also**

"MCountClasses" on page 4.83

## MGetGService, MGetPService

```
MGetGService( MSelf *self, CompID
    *theID, long instkey, long
    flags, MService **theservice )
```

MGetGService and MGetPService retrieve global- and project-level services by service identifier, respectively.

**Parameters**

| | |
|---|---|
| self | your component |
| theID | service id |
| instkey | service instance information, currently unused |
| flags | additional information, currently unused |
| theService | the service to return |

**Example**

```
static MErr CompProcessMessage(MomentumComp *self, MMessagePtr message)
{
 if ( MDetectMessage(message, self->f_enableEvent) )
 {
      MService*gravServ;

      MGetService(self, kGravityService, kMAnyInstance, kMSpawnIfNotFound,
&gravServ);
      if ( MError() == kMNoCompErr )
      {
         MAddClient(gravServ, self);
         // ...
      }
      return kMNoCompErr;
 }
 // ...
}
```

**See Also**

“MRegisterGService, MRegisterPService” on page 4.87

“MUnregisterGService, MUnregisterPService” on page 4.86

## MUnregisterGService, MUnregisterPService

```
MUnregisterGService( MSelf *self,
    MSelf *theservice )
```

MUnregisterGService and MUnregisterPService unregister global- and project-level services.

**Parameters**

| | |
|---|---|
| self | your component |
| theService | the service to be unregistered |

**Example**

```
static MErr MCompProcessMessage(GravitySrvComp *self, MMessagePtr message)
{
  if ( MDetectMessage(message, self->f_enableEvent) )
  {
       MRegisterGService(self, self, kMGeneralServiceKey);
       return kMNoCompErr;
  }
  else if ( MDetectMessage(message, self->f_disableEvent) ||
MDetectDisableMessage(message) )
  {
       MUnregisterGService(self, self);
       return kMNoCompErr;
  }
  else
       return kMUnableToComplyCompErr;
}
```

**See Also**

"MRegisterGService, MRegisterPService" on page 4.87

## MRegisterGService, MRegisterPService

```
MRegisterGService( MSelf *self,
   MSelf *theservice, long
   instkey )
```

MRegisterGService and MRegisterPService register a component as a global- or project-level service, making it available to clients.

**Parameters**

| | |
|---|---|
| self | your component |
| theservice | the service component to be registered |
| instkey | use kMGeneralServiceKey |

### Example

```
static MErr MCompProcessMessage(GravitySrvComp *self, MMessagePtr message)
{
 if ( MDetectMessage(message, self->f_enableEvent) )
 {
      MRegisterGService(self, self, kMGeneralServiceKey);
      return kMNoCompErr;
 }
 else if ( MDetectMessage(message, self->f_disableEvent) ||
            MDetectDisableMessage(message) )
 {
      MUnregisterGService(self, self);
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

### See Also

"MUnregisterGService, MUnregisterPService" on page 4.86

## MOM DEBUG SERVICE

The MOM Debug Service provides a method to post debug messages into the mTropolis message log window.

## MPostDebugMessage

```
MPostDebugMessage(MSelf *self,
   short severity, char
   *theMessage)
```

**Parameters**

| | |
|---|---|
| self | your component |
| severity | level of importance of the message, currently unused |
| theMessage | a null-terminated message string |

**Example**

```
MPostDebugMessage( self, nil, “A message for the message log” );
```

## MOM DIALOG EDITOR SERVICE

The MOM Editor service provides a set of functions to handle a component's dialog controls at the authoring time. Currently implemented functions are listed in the following table.

| Editor Functional Area | Methods |
|---|---|
| Editor Item Manipulation | MSetEditorItem, MGetEditorItem, MSetItemTitle, MGetItemTitle, MSetItemSpec, MGetItemSpec, MSetEditorItemState, MSetEditorSubItemState |
| Dialog Handling | MForceDialogUpdate |
| Symbol Manipulation (for pop-ups) | MAddSymbol, MForgetSymbols, MGetSymbolName, MSetSymbolName, MGetSymbolSelection, MAddSubSymbolItem |

## MAddSymbol

```
MAddSymbol( void *editor, short
   item, char *symName, long
   flags, long symVal, long
   symInfo )
```

MAddSymbol adds a symbol to a dynamic popup menu. A symbol is fully identifiable by its value, symbol info and name.

**Parameters**

| | |
|---|---|
| editor | your component's editor object |
| item | the item number of the dynamic popup menu item in the dialog |
| symName | a null terminated character string containing the symbol's name |
| flags | the default state of the item in the popup, currently always set to enabled (kMEnabledItem, kMDisabledItem) |
| symVal | the value part of the symbol of type MSymbol |
| symInfo | the info part of the symbol of type MSymbol |

**Example**

```
#define kDynamicPopupItem 17

static MErr MCompEditorOpen(PopupComp *self, void *editor, short editorType)
{
 if ( editorType == kMDialogEditorType )
 {
      MAddSymbol(editor, kDynamicPopupItem, "Dynamic 1", kMEnabledItem, 1, 0);
      MAddSymbol(editor, kDynamicPopupItem, "-", kMDisabledItem, 0, 0);
      MAddSymbol(editor, kDynamicPopupItem, "Dynamic 2", kMEnabledItem, 2, 0);
      MAddSymbol(editor, kDynamicPopupItem, "Dynamic 3", kMEnabledItem, 3, 0);
      MSetEditorItem(editor, kStaticPopupItem, (MDataType *)
            &self->f_staticPopup);
      MSetEditorItem(editor, kDynamicPopupItem, (MDataType *)
            &self->f_dynamicPopup);
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

"MSetEditorItem" on page 4.99

## MAddSubSymbolItem

```
MAddSubSymbolItem( void *editor,
   short item, long parcode, long
   parinfo, char *parname, long
   subcode, long subinfo, char
   *subname )
```

MAddSubSymbolItem adds a symbol to a sub menu of a dynamic popup menu.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the dynamic popup menu item in the dialog |
| parcode | symbol value of the parent menu |
| parinfo | symbol info of the parent menu |
| parname | symbol name of the parent menu |
| subcode | symbol value of the sub popup menu item |
| subinfo | symbol info of the sub popup menu item |
| subname | symbol name of the sub popup menu item |

**Example**

```
#define kDynamicPopupItem 17

static MErr MCompEditorOpen(PopupComp *self, void *editor, short editorType)
{
  if ( editorType == kMDialogEditorType )
  {
      MAddSymbol(editor, kDynamicPopupItem, "a parent menu", kMEnabledItem,
                  1, 0);
      MAddSubSymbolItem(editor, kDynamicPopupItem, 1, 0, "a parent menu", 11, 0,
                  "the first child");
      MAddSubSymbolItem(editor, kDynamicPopupItem,1,0,"a parent menu",12,0,
                  "the second child");
      MAddSubSymbolItem(editor, kDynamicPopupitem,1,0,"a parent menu",13,0,
                  "the third child")
      return kMNoCompErr;
  }
  else
      return kMUnableToComplyCompErr;
}
```

**See Also**

“MAddSymbol” on page 4.91

## MForgetSymbols

```
MForgetSymbols( void*editor,
   short item)
```

MForgetSymbols releases all symbols added to the dynamic popup menu item.

**Parameters**

| | |
|---|---|
| editor | the editor object of your component |
| item | the item number of the dynamic popup menu item in the dialog |

**See Also**

"MAddSymbol" on page 4.91

## MGetEditorItem

```
MGetEditorItem( void*editor,
    short item, MDataType *thedata
    )
```

MGetEditorItem retrieves the data from the a dialog item.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the dialog item whose value is to be retrieved |
| thedata | the return value |

**Example**

```
#define kValueItem 34

static MErr MCompEditorAccept( YourOwnComp *self, void *editor)
{
 MGetEditorItem(editor, kValueItem, (MDataType *) &self->f_myMString);
 return kMNoCompErr;
}
```

**See Also**

"MSetEditorItem" on page 4.99

## MGetItemSpec

```
MGetItemSpec(void *editor, short
    item, short expectedItemType,
    void* itemSpec )
```

MGetItemSpec retrieves the specification of the given item. The data returned is typically of type MControl.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose spec is to be retrieved |
| expectedItemType | the data type expected returned in itemSpec (use kMInteger). |
| itemSpec | the return value |

**Example**

```
#definekPathTextItem7// a text field in the dialog

static MErr CompEditorAccept(MyComp *self, void *editor)
{
  MErr       ret;
  MControl   itemSpec;
  long       len;

  MGetItemSpec(editor, kPathTextItem, kMInteger, &itemSpec);
  if ((ret = MError()) != kMNoCompErr)
       return ret;

  //get the length of text in the text field
  len = (long)itemSpec.f_value+1;//null terminator
  //...
}
```

**See Also**

"MSetItemSpec" on page 4.102
"MGetEditorItem" on page 4.94

## MGetItemTitle

```
MGetItemTitle(void *editor, short
   item, char *itemTitle )
```

MGetItemTitle retrieves the title of the given item in the editor.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose title is to be retrieved |
| itemTitle | the return value, memory should be allocated by the caller |

**See Also**

"MSetItemTitle" on page 4.103

## MGetSymbolName

```
MGetSymbolName(void *editor,
   short item, long itemcode, long
   iteminfo, char *itemname )
```

MGetSymbolName retrieves the name of the symbol with the given symbol value and symbol info at the given popup menu item in the dialog.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the popup menu item |
| itemcode | the symbol value |
| iteminfo | the symbol info |
| itemname | the null terminated name of the symbol, the memory should be provided by the called |

**Example**

```
#define kDynamicPopupItem 17

static MErr MCompEditorOpen(PopupComp *self, void *editor, short editorType)
{
 if ( editorType == kMDialogEditorType )
 {
      char*itemname[256];

      MAddSymbol(editor, kDynamicPopupItem, "the first menu", kMEnabledItem,
                  111, 0);
      MAddSymbol(editor, kDynamicPopupItem, "the second menu", kMEnabledItem,
                  222, 0);
      MGetSymbolName(editor,kDynamicPopupItem,222,0,itemname);
      //itemname is set to "the second menu
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

“MSetSymbolName” on page 4.104
“MAddSymbol” on page 4.91
“MAddSubSymbolItem” on page 4.92

## MGetSymbolSelection

```
MGetSymbolSelection(void
   *editor, short item, char
   *itemname )
```

MGetSymbolSelection retrieves the name of the symbol that is currently selected.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the popup menu item |
| itemname | the null terminated name of the symbol. The space for the name should be provided by the caller |

**See Also**

“MGetSymbolName” on page 4.97

## MSetEditorItem

```
MSetEditorItem( void*editor,
    short item, MDataType *thedata
    )
```

MSetEditorItem sets the value of a dialog item.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose is to be set |
| thedata | the data used in setting the given item |

**Example**

```
#define kValueItem 34

static MErr MCompEditorOpen( YourOwnComp *self, void *editor, short editorType)
{
 char * retstring;

 if ( editorType == kMDialogEditorType )
 {
      MSetEditorItem(editor, kValueItem, (MDataType *) &self->f_myMString);
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

"MGetEditorItem" on page 4.94

## MSetEditorItemState

```
MSetEditorItemState(void
    *editor, short item, short
    itemOption )
```

MSetEditorItemState enables or disables the state of a dialog item.

### Parameters

| | |
|---|---|
| editor | the editor of your component |
| item | the item number |
| itemOption | The state the item is to be set, can be one of the following : kMDisabledItem kMEnabledItem |

### Example

```
#define kCheckbox 14

static MErr MCompEditorItemChanged( YourOwnComp *self, void *editor,
            short editorType, short item, short part )
{
 MErr   retval = kMNoCompErr;
 MBoolean status;

 switch( item )
 {
     case kCheckbox :
        MInitBoolean( status, false);
        MGetEditorItem( editor, kCheckbox, (MDataType *)&status )
        if( status->f_value )
        {
           MSetEditorItemState( editor, kInfoRadioButton, kMEnabledItem );
        }
        else
        {
           MSetEditorItemState( editor, kInfoRadioButton, kMDisabledItem );
        }
        break;
     }
 return retval;
}
```

### See Also

"MSetEditorItem" on page 4.99
"MSetEditorSubItemState" on page 4.101

## MSetEditorSubItemState

```
MSetEditorSubItemState(void*
   editor, short item, short
   subItyem, short itemOption);
```

MSetEditorSubItemState enables or disables a submenu item.

**Parameters**

| | |
|---|---|
| editor | the editor object |
| item | pop-up menu item number |
| subItem | ID of the submenu item |
| itemOption | The state the item is to be set, can be one of the following : kMDisabledItem kMEnabledItem |

**See Also**

“MSetEditorItemState” on page 4.100

## MSetItemSpec

```
MSetItemSpec(void *editor, short
   item, short expectedItemType,
   void* itemSpec )
```

MSetItemSpec sets the specification of the given item.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose spec is to be set |
| expectedItemType | the data type expected returned in itemSpec (use kMInteger) |
| itemSpec | the returned spec |

**See Also**

“MGetItemSpec” on page 4.95

## MSetItemTitle

```
MSetItemSpec(void *editor, short
   item, char *newtitle )
```

MSetItemTitle sets the title of the given item.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose spec is to be set |
| newtitle | the title of the item |

**See Also**

"MGetItemTitle" on page 4.96

## MSetSymbolName

```
MSetSymboleName(void *editor,
   short item, long itemcode, long
   iteminfo, char *newname )
```

MSetSymbolName sets the name of the given popup item with the given symbol value and symbol info.

**Parameters**

| | |
|---|---|
| editor | the editor of your component |
| item | the item number of the item whose spec is to be set |
| itemcode | the given symbol value |
| iteminfo | the given symbol info |
| itemname | the new item name for the item with given symbol value and symbol info |

**See Also**

“MGetSymbolName” on page 4.97

## MForceDialogUpdate

```
MForceDialogUpdate( void
   *editorObj);
```

MforceDialogUpdate forces the editor dialog to update itself.

**Parameters**

| | |
|---|---|
| editorObj | component’s editor object |

## MOM FILE SERVICES

The MOM File Service provides a set of functions to save and restore MOM component information to a file or stream. MSizeofValue, MReadValue and MWriteValue provide convenient and efficient streaming of MDataType variables. MReadBytes and MWriteBytes are for non-MDataType variable streaming.

| **File Functionality** | **Methods** |
|---|---|
| MOM Data Types | MSizeOfValue, MWriteValue, MReadValue |
| Arbitrary Data | MWriteBytes, MReadBytes |
| Integer Values | MWriteInt16, MReadInt16, MWriteInt32, MReadInt32 |

## MReadBytes

```
MReadBytes( MStream *theFile, void
    *theData, long bytesToRead )
```

MReadBytes reads a number of bytes of binary data from the given stream.

**Parameters**

| | |
|---|---|
| theFile | the stream to read from |
| theData | a pointer to the data buffer, memory should be allocated by the caller |
| bytesToRead | the number of bytes to read into the buffer. The number of bytes read must be exactly the same amount as was written in order to maintain the integrity of the stream. |

**Example**

```
static MErr MCompRestoreComponent(MSelf *self, MStream *file, long saveInfo,
            short rev)
{
 MErr ret;
 size_t  valueSize;
 long    ltemp;
 char    localfooString[128];

 ltemp = sizeof( localfooString );
 if ( rev == kStringRev )
 {
      MReadBytes( file, (void *)localfooString, ltemp);
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      MReadValue(file, &self->f_value);
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

“MWriteBytes” on page 4.112

## MReadInt16

```
MReadInt16(MStream *theFile, void
   *theData, long nInt16 )
```

MReadInt16 reads a number of 16-bit integers from the file.

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theData | a pointer to the data buffer, memory should be allocated by the caller |
| nInt16 | the number of 16-bit integer numbers to read from the file, the total bytes to read is `nInt16*sizeof(short)` |

**See Also**

"MWriteInt16" on page 4.113

## MReadInt32

```
MReadInt32(MStream *theFile, void
   *theData, long nInt32 )
```

MReadInt32 reads a number of 32-bit integers from the file.

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theData | a pointer to the data buffer, memory should be allocated by the caller |
| nInt32 | the number of 32-bit integer numbers to read from the file, the total bytes to read is `nInt32*sizeof(long)` |

**See Also**

"MWriteInt32" on page 4.114

## MReadValue

```
MReadValue(MStream *theFile,
   MDataType *theValue )
```

MReadValue is used to extract MDataType structures from the stream. Buffer space is allocated by MReadValue for pointer-based MDataType structures (i.e., MString, MPointer, etc.). Use this function to read MDataType structures that have been written to the stream with MWriteValue. MReadValue keeps the persistence of what's written with MWriteValue.

If you pass an MString into MReadValue, the MString's f_ptr should be disposed of before calling MReadValue:

```
MDisposeString( self->f_string );
MInitSizedString( self->f_string, 0 );
MReadValue( file, &self->f_string );
```

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theValue | a pointer to the MDataType structure |

**Example**

```
static MErr MCompRestoreComponent(MSelf *self, MStream *file, long saveInfo,
            short rev)
{
 MErr    ret;
 size_t  valueSize;
 long    ltemp;
 char    localfooString[128];

 ltemp = sizeof( localfooString );
 if ( rev == kStringRev )
 {
      MReadBytes( file, (void *)localfooString, ltemp );
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      MReadValue(file, &self->f_value);
      if ((ret = MError()) != kMNoCompErr)
         return ret;
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

"MWriteValue" on page 4.115

## MSizeOfValue

```
MSizeOfValue(MStream *theFile,
    MDataType *theValue, long
    *length)
```

MSizeOfValue calculates the number of bytes of storage space required in the stream by the MDataType structure. When calculating the size of pointer based MDataType structures, MOM ignores and stores the buffer length and uses the data length (i.e., strlen for MString including the null terminator) in determining the storage requirements of the MDataType. If MWriteValue and MReadValue are used in conjunction, the MDataTypes are restored to their original state.

**Parameters**

| | |
|---|---|
| theFile | the file or the stream |
| theValue | a pointer to the MDataType structure |
| length | the return value |

**Example**

```
static MErr MCompGetSaveInfo(MSelf *self, MStream *file, long saveInfo,
            short *rev, long *len)
{
  size_t  valueSize;
  long    ltemp;

  MStrlen( &valueSize, self->somefoostring );
  *len = valueSize;
  MSizeOfValue( file, &self->f_value, &ltemp );
  *len += ltemp;
  *rev = kStringRev;
  return kMNoCompErr;
}
```

**See Also**

"MReadValue" on page 4.110
"MWriteValue" on page 4.115

## MWriteBytes

```
MWriteBytes(MStream *theFile,
   MDataType *theData, long
   bytesToWrite )
```

MWriteBytes writes a number of bytes of binary number into the file or stream.

**Parameters**

| | |
|---|---|
| theFile | the file or the stream. |
| theData | a pointer to the data being written |
| bytesToWrite | the number of bytes to write to the file |

**Example**

```
static MErr MCompSaveComponent(MSelf *self, MStream *file, long saveInfo)
{
 size_t  valueSize;
 long    ltemp = saveInfo;

 MStrlen( &valueSize,   foostring );
 ltemp = valueSize;
 MWriteBytes( file, (void *)foostring, ltemp );
 MWriteValue(file, &self->f_value);
 return kMNoCompErr;
}
```

**See Also**

"MReadBytes" on page 4.107

## MWriteInt16

```
MWriteInt16(MStream *theFile,
   void *theData, long nInt16 )
```

MWriteInt16 writes a number of 16-bit integers to the file.

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theData | a pointer to the data buffer |
| nInt16 | the number of 16-bit integer numbers to write to the file, the total bytes to write is `nInt16*sizeof(short)` |

**See Also**

"MReadInt16" on page 4.108

## MWriteInt32

```
MWriteInt32(MStream *theFile,
   void *theData, long nInt32 )
```

MWriteInt32 writes a number of 32-bit integers to the file.

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theData | a pointer to the data buffer |
| nInt32 | the number of 32-bit integer numbers to write to the file, the total bytes to write is `nInt32*sizeof(long)` |

**See Also**

“MReadInt32” on page 4.109

## MWriteValue

```
MWriteValue(MStream *theFile,
   MDataType *theValue )
```

MWriteValue is used during component storage to save an MDataType structure into the storage stream. MDataTypes written to stream using MWriteValue should use MReadValue to read them from the stream.

**Parameters**

| | |
|---|---|
| theFile | the file or stream |
| theValue | a pointer to the MDataType structure being stored |

**Example**

```
static MErr MCompSaveComponent(StringComp *self, MStream *file, long saveInfo)
{
 size_t  valueSize;
 long    ltemp;

 MStrlen( &valueSize,   self->somefoostring );
 ltemp = valueSize;
 MWriteBytes( file, (void *)self->somefoostring, ltemp ;
 MWriteValue(file, &self->f_value);
 return kMNoCompErr;
}
```

**See Also**

"MReadValue" on page 4.110

## MOM LIST SERVICE

The MOM list service provides methods to handle the creation, manipulation, and disposal of dynamic array data structures organized as lists. The size of each entry in the list is determined when the list is created. The list is always contiguous, and is resized whenever items are added or removed, preventing holes from appearing in the list. Lists use a 1-based index.

| List Function | Methods |
|---|---|
| List Creation and Destruction | MNewList, MDisposeList, MForgetItems |
| List Information | MGetNumItems |
| Item Manipulation | MPushItem, MPopItem, MInsertItem, MRemoveItem, MGetNthItem, MSetNthItem |

## MDisposeList

```
MDisposeList( MObjectRef
   *listRef)
```

MDisposeList disposes of all the items in the list and the list itself (as opposed to MForgetItems, which only disposes the items in the list).

**Parameters**

| | |
|---|---|
| listRef | the list object to be disposed |

**Example**

```
static void ListTest(MSelf *self)
{
 MObjectRef *listRef = nil;

 MNewList(sizeof(MInteger), &listRef);
 if (MError() == kMNoCompErr && listRef)
 {
     MDisposeList(listRef);
 }
}
```

**See Also**

“MNewList” on page 4.122
“MForgetItems” on page 4.118

## MForgetItems

```
MForgetItems( MObjectRef
    *listRef)
```

MForgetItems releases or frees all the items currently in the list, and sets the length of the list to 0. The list itself is not freed. Use MDisposeList to free the list's items and the list itself.

**Parameters**

| | |
|---|---|
| listRef | the list object |

**Example**

```
MErr copylist ( MObjectRef *destRef, MObjectRef *srcRef )
{
  UInt32 i,srccount;
  char *databuffer[1024];

  //assuming that items of both lists are of the same size and less than 1024
  MForgetItems(destList);
  MGetNumItems( srcRef, &srccount );
  for (i=1; i<=srccount; i++)
  {
       MGetNthItem( srcRef, i, (void *)databuffer );
       MSetNthItem( destRef, i, (void *)databuffer );
  }
  return kMNoCompErr;
}
```

**See Also**

"MDisposeList" on page 4.117
"MNewList" on page 4.122

## MGetNthItem

```
MGetNthItem( MObjectRef *listRef,
   UInt32 itemIndex, void
   *dataValue )
```

MGetNthItem retrieves a copy of an item in the list at a given index. Space for the retrieved item should be provided by the caller.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| itemIndex | the index of the item in the list, the index starts at 1 |
| dataValue | the buffer that stores the retrieved item, the size of the buffer must be no less than the item size of the list |

**Example**

```
MErr copylist ( MObjectRef *destRef, MObjectRef *srcRef )
{
 UInt32 i,srccount;
 char *databuffer[1024];

 //assuming that items of both lists are of the same size
 MForgetItems(destList);
 MGetNumItems( srcRef, &srccount );
 for (i=1; i<=srccount; i++)
 {
      MGetNthItem( srcRef, i, (void *)databuffer );
      MSetNthItem( destRef, i, (void *)databuffer );
 }
 return kMNoCompErr;
}
```

**See Also**

"MSetNthItem" on page 4.126

"MForgetItems" on page 4.118

## MGetNumItems

```
MGetNumItems( MObjectRef
   *listRef, long *numItems )
```

MGetNumItems returns the number of the items in the list.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| numItems | where the return value goes |

**Example**

```
MErr copylist ( MObjectRef *destRef, MObjectRef *srcRef )
{
 UInt32 i,srccount;
 char *databuffer[1024];

 //assuming that items of both lists are of the same size and less than 1024
 MForgetItems(destList);
 MGetNumItems( srcRef, &srccount );
 for (i=1; i<=srccount; i++)
 {
     MGetNthItem( srcRef, i, (void *)databuffer );
     MSetNthItem( destRef, i, (void *)databuffer );
 }
 return kMNoCompErr;
}
```

**See Also**

"MInsertItem" on page 4.121
"MPushItem" on page 4.124
"MPopItem" on page 4.123

## MInsertItem

```
MInserttem( MObjectRef *listRef,
   UInt32 itemIndex, void
   *dataValue )
```

MInsertItem inserts an item into the list at a given index, increasing the indices of the following items by 1. For example, assume a list that contains 3 items: {x, y, z}. After inserting a new item w at index of 2, the list will be {x, w, y, z}. This is different than MSetNthItem, which replaces the item at the given index. For example, in the above list, using MSetNthItem to set a new item w at the index of 2, the list would become {x, w, z}.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| itemIndex | the inserting position, starting at 1 |
| dataValue | the item |

**See Also**

"MSetNthItem" on page 4.126

## MNewList

```
MNewList( UInt32 theSize,
   MObjectRef **listRef )
```

MNewList creates a list of a given item size.

**Parameters**

| | |
|---|---|
| listRef | where the returned new list goes |
| theSize | item size of the list, must be >0 |

**Example**

```
static void ListTest(MSelf *self)
{
 MObjectRef *ListRef = nil;
 MIntegeritem;

 MNewList(sizeof(item), &listRef);
 if (MError() == kMNoCompErr && listRef)
 {
     MDisposeList(listRef);
 }
}
```

**See Also**

"MDisposeList" on page 4.117

## MPopItem

```
MPopItem( MObjectRef *listRef,
   void *dataValue )
```

MPopItem retrieves the last item in the list and deletes it from the list.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| dataValue | where the retuned item goes |

**Example**

```
MErr reverseList (MSelf *self, MObjectRef *list)
{
 UInt32 count;
 char *databuffer[1024];

 //assuming that list item size is less than 1024
 MGetNumItems( list, &count);
 for (i=1; i<=count; i++)
 {
      MPopItem(list, (void *)databuffer );
      MSetNthItem( list, i, (void *)databuffer );
 }
 return kMNoCompErr;
}
```

**See Also**

"MPushItem" on page 4.124

## MPushItem

```
MPushItem( MObjectRef *listRef,
   MomDataType *dataValue )
```

MPushItem appends an item to the end of the list.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| dataValue | the item data |

### Example

```
MErr appendList (MSelf *self, MObjectRef *destList, MObjectRef *srcList)
{
 UInt32 count;
 char *databuffer[1024];

 MGetNumItems(srcList, &count);
 for (i=1; i<=count; i++)
 {
     MGetNthItem(srcList, i, (void *)databuffer );
     MPushItem(destList, (void *)databuffer);
 }
 return kMNoCompErr;
}
```

### See Also

“MPopItem” on page 4.123

## MRemoveItem

```
MRemoveItem( MObjectRef *listRef,
   UInt32 itemIndex )
```

MRemoveItem removes an item from the list at a specific position, and decreases the items whose indices are greater than the given index by 1. For example, if a list contains {x, y, z}, and we use MRemoveItem to remove the 2nd item, then the list becomes {x, z}.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| itemIndex | the index of item in list, should be > 0 and not exceed the number of items in the list |

**See Also**

“MInsertItem” on page 4.121

## MSetNthItem

```
MSetNthItem( MObjectRef *listRef,
   UInt32 itemIndex, void
   *dataValue )
```

MSetNthItem replaces an item in the list at a given index with new data.

**Parameters**

| | |
|---|---|
| listRef | the list object |
| itemIndex | the index of item to be modified. Should be >0 and not exceed the number of items in the list. |
| dataValue | new item value |

### Example

```
MErr copylist ( MObjectRef *destRef, MObjectRef *srcRef )
{
 UInt32 i,srccount;
 char *databuffer[1024];

 MForgetItems(destList);
 MGetNumItems( srcRef, &srccount );
 for (i=1; i<=srccount; i++)
 {
      MGetNthItem( srcRef, i, (void *)databuffer );
      MSetNthItem( destRef, i, (void *)databuffer );
 }
 return kMNoCompErr;
}
```

### See Also

"MGetNthItem" on page 4.119
"MForgetItems" on page 4.118

## MOM MATH SERVICE

The MOM math service provides wrappers for the standard runtime math functions. The primary purpose of this service is to assist MOM developers in managing their component size. By making these standard math functions available through a MOM service, the ANSI library containing the math functions themselves does not have to be included in the component itself. If a math error occurs, the return value will reflect if it was a domain or a range error. This behavior is based upon the ANSI-C standard.

## MACos

```
MACos (double *retval, double
   inval)
```

This function computes the arccosine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MASin

```
MASin( double *retval, double
   inval)
```

This function computes the arcsine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MATan

```
MATan (double *retval, double
   inval)
```

This function computes the arctangent of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MATan2

```
MATan2 ( double *retval , double
   invalX, double invalY)
```

This function computes the arctangent of invalY/invalX and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| invalX | The value for the math function to act upon |
| invalY | The value for the math function to act upon |

## MCeil

```
MCeil ( double *retval , double
   inval)
```

This function computes the smallest integer that is greater than or equal to inval and returns the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MCos

```
MCos ( double *retval , double
   inval)
```

This function computes the cosine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MCosH

```
MCosH ( double *retval, double
   inval)
```

This function computes the hyperbolic cosine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MExp

```
MExp ( double *retval, double
   inval)
```

This function computes the exponential function of the floating point argument, inval, and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MFAbs

```
MFAbs (double *retval, double
   inval)
```

This function computes the absolute value of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MFloor

```
MFloor ( double *retval , double
   inval)
```

This function places the smallest integer, that is less than or equal to inval, in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MFMod

```
MFMod ( double *retval, double
   invalX, double invalY)
```

This function calculates the remainder of invalX/invalY, such that *invalX* = *i* * *invalY* + *remainder*. The *remainder* is returned in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| invalX | The value for the math function to act upon |
| invalY | The value for the math function to act upon |

## MFrExp

```
MFrExp ( double *retval, double
   inval, int *exp)
```

This function breaks down the floating-point value (inval) into a mantissa (m) and an exponent (exp), such that the absolute value of m is greater than or equal to 0.5 and less than 1.0, and $inval = m * 2^{exp}$. The mantissa (*m*) is returned in result.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |
| exp | the returned exponent |

## MLdExp

```
MLdExp ( double *retval, double
   inval, double exp )
```

This function computes a real number from the mantissa (inval) and exponent (exp), calculating the value of $inval * 2^{exp}$ and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |
| exp | The exponential value for 2 |

## MLog

```
MLog ( double *retval , double
   inval)
```

This function computes the natural logarithm of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MLog10

```
MLog10 ( double *retval, double
   inval)
```

This function computes the base-10 logarithm of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MModF

```
MModF ( double *retval , double
   inval, double *integralpart)
```

This function breaks down inval into fractional and integral parts, each of which has the same sign as inval. The signed fractional portion of inval is returned in retval and the signed integral portion is returned in integralpart.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |
| integralpart | The value for the math function to act upon |

## MPow

```
MPow ( double *result , double
   inval, double exp)
```

This function computes inval, raised to the power of exp, and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |
| exp | The exponential value for the math function to act upon |

## MSin

```
MSin ( double *retval, double
   inval)
```

This function computes the sine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MSinH

```
MSinH ( double *retval, double
   inval)
```

This function computes the hyperbolic sine of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MSqrt

```
MSqrt ( double *retval , double
   inval )
```

This function computes the square root of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MTan

```
MTan ( double *retval, double
   inval)
```

This function computes the tangent of inval and places the result in retval.

**Parameters**

| | |
|---|---|
| retval | The return value of the math function |
| inval | The value for the math function to act upon |

## MTanH

```
MTanH( double *retval, double
   inval);
```

This function computes the hyperbolic tangent of inval and returns the result in retval.

**Parameters**

| | |
|---|---|
| retval | result of function |
| inval | argument |

## MCOMPONENT BASE CLASS METHODS

MComponent is the base class from which MOM components derive themselves. It provides numerous methods to customize the component, including construction, destruction, and assignment. It also provides methods for handling modifier editor dialogs, persistence, runtime, attribute handling, client-server relationships between components, messaging and other notification methods.

| MComponent Functional Area | Methods |
|---|---|
| Basic | MCompConstructor, MCompCopyConstructor, MCompDestructor, MCompCopy |
| Dialog Editors | MCompEditorOpen, MCompEditorAccept, MCompEditorDecline, MCompEditorItemChanged, MCompEditorCustomItemMouseEvent, MCompEditorCustomItemDraw, MCompEditorPrevalidateItem, MCompApplyEditorEffect |
| Persistence | MCompGetSaveInfo, MCompSaveComponent, MCompRestoreComponent |
| Attribute Handling | MCompGetAttributeCount, MCompGetAttributeMaxIndex, MCompGetNthAttributeName, MCompGetAttribute, MCompSetAttribute |
| Messaging | MCompProcessMessage |
| Client-Server | MAddClient, MDeleteClient, MGetNumClients, MGetNthClient |
| Misc Notification | MCompEnabled, MCompDisabled, MCompConnect, MCompDisconnect, MCompReset, MCompFreeCache, MCompEnvironmentChanged |
| Low-Level | MCompExecutePrimitive, MRegisterMouseFeedback, MDeregisterMouseFeedback |
| Class Utilities | MClassStaticInit, MClassStaticDestruct |
| Misc Utility | MGetElement, MInvalidateElement, MCompGetCompID |

## MAddClient

```
MAddClient( MService *service,
   MSelf *self )
```

MAddClient creates a client-server inter-component relation, attaching your component as a client of the service. For example, a momentum modifier calls MGetGService to locate the gravity service, then calls MAddClient to register itself with that service.

**Parameters**

| | |
|---|---|
| service | the service of which you want to be a client. This is usually obtained by calling MGetGService or MGetPService |
| self | your component |

**Example**
For the momentum modifier mentioned above, it may register with the gravity service as follows:

```
static MErr MCompProcessMessage(MomentumComp *self, MMessagePtr message)
{
 if (MDetectMessage(message, self->f_enableEvent))
 {
     MService*gravServ;

     MGetService(self, kGravityService, kMAnyInstance, kMSpawnIfNotFound,
             &gravServ);
     if (MError() == kMNoCompErr)
     {
        MAddClient(gravServ, self);
        // ...
     }
     return kMNoCompErr;
 }
}
```

**See Also**
“MDeleteClient” on page 4.161
“MGetGService, MGetPService” on page 4.85

## MClassStaticInit

```
MClassStaticInit(short initCode)
```

MClassStaticInit is called once per class initialization. This provides an opportunity for components to do any one-time initialization (like checking for specific OS services, etc.).

**Parameters**

| | |
|---|---|
| initCode | initialization code. Currently kAppInit. |

## MClassStaticDestruct

```
MClassStaticDestruct(short
   destCode)
```

MClassStaticDestruct is called once per class initialization. This provides an opportunity for components to do any cleanup of data that was allocated at class initialization time.

**Parameters**

| | |
|---|---|
| destCode | destruction code. Currently kAppShutDown. |

## MCompApplyEditorEffect

```
MCompApplyEditorEffect( MSelf
    *self )
```

MCompApplyEditorEffect is called in edit mode for components that want to provide an edit-mode "preview" of their effects. For example, a graphic modifier wants its element to show a color (e.g., blue) in edit mode, not just in runtime when it receives its activation message. Usually, you only want to provide the effect if the user has selected either "Scene Started" or "Parent Enabled" as the activation message (this is how the graphic, gradient, and image effect modifiers all behave). This is accomplished by calling the MDefaultMessage() macro (defined in MEvent.h) to determine if your component's trigger event matches one of these events.

**Parameters**

| | |
|---|---|
| self | your component |

**Example**
If the following draw method was defined by your component in MCompMainName:

```
MDefineMethod(kDrawSlot, kDrawSlot, kDrawMethod, DrawMethod);
```

for a component whose data structure is:

```
typedef struct DrawComp
{
 MEventf_applyEvent;
} DrawComp;
```

then its MCompApplyEditorEffect method could be:

```
MErr MCompApplyEditorEffect(DrawComp *self)
{
 if ( MDefaultMessage(self->f_applyEvent) )
 {
      MFunk    funk;
      MObjectRef*element;

      MInitFunk(funk, kMBufferedDrawFunk, self, kDrawSlot, kDrawMethod);
      MGetElement(&element);
      MRegisterFunk(element, &funk);
 }
}
```

**See Also**
MDefaultMessage, defined in MEvent.h

## MCompConnect

```
MCompConnect( MSelf *self, short
    connectCode )
```

The MCompConnect method is called whenever your component becomes associated with a parent. For modifier-style components, this will happen in the following situations:

- when the component is first dragged into the mTropolis project,
- when the component is moved to a new parent (e.g., from one element to another), and
- whenever the component's project is opened.

For service-style components, depending upon the options passed to MDefineComponent, this will happen whenever the service gets instantiated, which can happen:

- when the service first gets initialized or,
- when a client first connects to the service.

MCompConnect is the first method called after your component has been entirely "hooked up" to its parent. Thus, any actions you need to perform that depend upon your component being fully initialized (e.g., registering your service by calling MRegisterGService) should be handled here (as opposed to in MCompConstructor).

**Parameters**

| | |
|---|---|
| self | A pointer to your component's primary data structure. |
| connectCode | Not currently used. In the future, it will pass information relating to your new parent. |

**Example**

For a service-type class that wants to register its availability as soon as it's connected, then its MCompConnect method could be:

```
MErr MCompConnect(SampleComp *self, short connectCode)
{
 MRegisterGService(self, self, kMGeneralServiceKey);
 MPostCTimeTask(self, nil, kUpdatePeriod, kSampleSlot, kSampleTask);
 return kMNoCompErr;
}
```

**See Also**

"MCompDisconnect" on page 4.164

## MCompConstructor

```
MCompConstructor( MSelf *self,
   MObjectRef *mRef )
```

The MCompConstructor method is called when your component is first created. Depending upon which options you specified when calling MDefineComponent, this may occur in the following situations:

- when a modifier-style component is dragged off the kit palette into a project,
- when the project containing the component is opened,
- when a service-style component is first connected to by a client, or
- when mTropolis first launches and processes the kits.

The component may also be created via duplication of an existing component. In this case, MCompCopyConstructor will be called instead of MCompConstructor.

MCompConstructor allocates memory and initializes the fields of your component's primary data structure. MOM data type macros can be used to initialize fields of MOM data type. For example, use MInitPoint2D() for fields of type MPoint2D.

Note that MCompConstructor is the first method in your component that will be called. At the time that it is called, the component is not fully initialized. Any actions that depend upon your component being fully initialized (querying your parent, etc.) should be handled in your MCompConnect method. MCompConstructor is one of four methods that your class must override, along with MCompCopyConstructor, MCompCopy, and MCompDestructor.

**Parameters**

| | |
|---|---|
| self | A pointer to your component's primary data structure. Note that the memory for this structure has already been allocated by MOM prior to it calling MCompConstructor. |
| mRef | Unused. |

**Example**

```
typedef struct SampleComp
{
 MPoint2Df_pointOne;
 short   f_other;
} SampleComp;

MErr MConstructor(SampleComp *self, MObjectRef *mRef)
{
 MInitPoint2D(self->f_pointOne, 0, 0);
 self->f_other = 0;
```

```
  return kMNoCompErr;
}
```

**See Also**

MDefineComponent
“MCompCopy” on page 4.158
“MCompCopyConstructor” on page 4.159
“MCompDestructor” on page 4.160

## MCompCopy

```
MCompCopy( MSelf *self, MSelf
   *previousSelf)
```

The MCompCopy method is called when an alias to your component is being updated. MCompCopy copies the contents of the data structure from the previous (i.e., the original) component to your component, similar to the tasks performed in MCompCopyConstructor().

MCompCopy is one of four methods that your class must override, along with MCompConstructor , MCompDestructor and MCompCopyConstructor.

**Parameters**

| | |
|---|---|
| self | A pointer to your component's primary data structure. |
| previousSelf | A pointer to the previous (i.e., the original) component's primary data structure. |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp
{
 MInteger   f_integer;
 long       f_other;
} SampleComp;
```

then the MCompCopy method could be:

```
MErr MCompCopy(SampleComp *self, SampleComp *previousSelf)
{
 MCopyInteger(self->f_integer, previousSelf->f_integer);
 self->f_other = previousSelf->f_other;
 return kMNoCompErr;
}
```

**See Also**

"MCompConstructor" on page 4.156
"MCompCopyConstructor" on page 4.159
"MCompDestructor" on page 4.160

## MCompCopyConstructor

```
MCompCopyConstructor( MSelf
    *self, MSelf *previousSelf,
    MObjectRef *mRef )
```

The MCompCopyConstructor method is called when your component is instantiated by duplication. For example, this occurs when the author selects a component in mTropolis and chooses “Copy” or “Duplicate” from the ‘Edit’ menu.

MCompCopyConstructor copies the contents of the data structure from the previous (i.e., the original) component to your component. MOM data types macros starting with MCopy…() can be used to copy fields of MOM data types, for example, use MCopyInteger() for MOM data fields of type MInteger.

MCompCopyConstructor is one of four methods that your class must override, along with MCompConstructor, MCompDestructor and MCompCopy.

**Parameters**

| | |
|---|---|
| self | A pointer to your component’s primary data structure. |
| previousSelf | A pointer to the previous (i.e., the original) component’s primary data structure. |
| mRef | Unused. |

**Example**
For a class whose data structure is:

```
typedef struct SampleComp
{
 MInteger   f_integer;
 long       f_other;
} SampleComp;
```

then the ‘MCopyConstructor’ method could be:

```
MErr MCopyConstructor(SampleComp *self, SampleComp *previousSelf,
      MObjectRef *mRef)
{
 MCopyInteger(self->f_integer, previousSelf->f_integer);
 self->f_other = previousSelf->f_other;
 return kMNoCompErr;
}
```

**See Also**
“MCompConstructor” on page 4.156
“MCompCopy” on page 4.158
“MCompDestructor” on page 4.160

## MCompDestructor

```
MCompDestructor(MSelf *self )
```

The MCompDestructor method is called when your component is in the process of being deleted. In MCompDestructor, your component should dispose of any memory allocated in MCompConstructor() and the fields of your component's primary data structure. MOM data type macros starting with MDispose…() can be used to dispose fields of MOM data types. For example, use MDisposePtr() to dispose fields of type MPointer.

MCompDestructor is one of four methods that your class must override, along with MCompConstructor, MCompCopyConstructor and MCompCopy.

**Parameters**

| | |
|---|---|
| self | Your component |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MPointer   f_ptrOne;
} SampleComp;
```

then the MCompDestructor method could be:

```
MErr MCompDestructor(SampleComp *self)
{
 MDisposePtr(self->f_ptrOne);
 return kNoCompErr;
}
```

**See Also**

"MCompConstructor" on page 4.156
"MCompCopyConstructor" on page 4.159
"MCompCopy" on page 4.158

## MDeleteClient

```
MDeleteClient( MSelf *service,
   MSelf *self )
```

Whereas MAddClient registers your component to a service as a client, MDeleteClient deregisters your component from the service. MAddClient and MDeleteClient complement each other in the client-server inter-component relationship.

**Parameters**

| | |
|---|---|
| service | The service from which you want to deregister. This is usually obtained by calling MGetGService or MGetPService |
| self | Your component |

**Example**

For a momentum modifier to deregister from the gravity service:

```
static MErr MCompProcessMessage(MomentumComp *self, MMessagePtr message)
{
 if ( MDetectMessage(message, self->f_disableEvent) ||
        MDetectDisableMessage(message) )
 {
      MService*gravServ;

      MGetGService(self, kGravityService, kMAnyInstance,
                kMSpawnIfNotFound, &gravServ);
      if ( MError() == kMNoCompErr )
      {
         MDeleteClient(gravServ, self);
         // ...
      }
      return kMNoCompErr;
 }
}
```

**See Also**

“MAddClient” on page 4.151

“MGetGService, MGetPService” on page 4.85

## MDeregisterMouseFeedBack

```
MDeregisterMouseFeedBack( MSelf
    *self, void* primData )
```

Whereas MRegisterMouseFeedBack registers your component to mTropolis to receive mouse input events, MDeregisterMouseFeedBack causes the mod to stop receiving mouse input events.

**Parameters**

| | |
|---|---|
| self | your component |
| primData | the event that has been registered to receive |

**Example**

```
static MErr MCompDisabled( YourOwnComp *self )
{
  MDeregisterMouseFeedBack( self,&self->f_activateEvent );
  return kMNoCompErr;
}
```

**See Also**

“MRegisterMouseFeedBack” on page 4.190

## MCompDisabled

```
MCompDisabled( MSelf *self )
```

As a complement to MCompEnabled, the MCompDisabled method is called whenever your component is disabled during runtime. Service-style components will get this method called as soon as runtime is exited. For modifier-style component, this may happen in one of two ways:

- if the component isn't in a switchable behavior, then it will get disabled when its scene ends, or
- if the component is in a switchable behavior, then it will get disabled whenever that behavior switches off.

This method will get called after any external event being sent (e.g., "Parent Disabled").

One example of what you may want to do in your MCompDisabled method is to deregister mTropolis' automatic mouse feedback (see MCompEnabled for a description). This is accomplished by calling the MDeregisterMouseFeedback function from within your MCompDisabled method.

**Parameters**

| | |
|---|---|
| self | A pointer to your component's primary data structure. |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_activateEvent;
} SampleComp;
```

then the MCompDisabled method could be:

```
MErr MCompDisabled(SampleComp *self)
{
 MDeregisterMouseFeedback(self, &self->f_activateEvent);
 return kMNoCompErr;
}
```

**See Also**

"MCompEnabled" on page 4.173

"MDeregisterMouseFeedBack" on page 4.162

## MCompDisconnect

`MCompDisconnect( MSelf *self )`

As a complement to MCompConnect, the MCompDisconnect method is called whenever your component is about to be disconnected from its parent. For modifier-style components, this will happen both when you delete the component, and when the component is being moved to a new parent (e.g., from one element to another).

MCompDisconnect is the last method called prior to your component being disconnected from its parent. Note that your component is still fully "hooked up" to its parent at this time. Thus, any final actions your need to perform that depend upon your component being disconnected to its current parent (e.g., deregistering your service by calling MUnregisterGService) should be handled here (as opposed to in MCompDestructor).

**Parameters**

| | |
|---|---|
| self | A pointer to your component's primary data structure. |

**Example**

For a service-type class that wants to deregister its availability as soon as it's about to be disconnected, then its MCompDisconnect method could be:

```
MErr MCompDisconnect(SampleComp *self)
{
  MUnregisterGService(self, self);
  MKillCTimeTask(self, nil);
  return kMNoCompErr;
}
```

**See Also**

"MCompConnect" on page 4.155
"MUnregisterGService, MUnregisterPService" on page 4.86

## MCompEditorAccept

```
MCompEditorAccept( MSelf *self,
    void *editor, short editorType
    )
```

The MCompEditorAccept method is called when the user confirms the settings in the dialog. For example, this occurs when the user hits the “OK” button in your component’s dialog. MCompEditorAccept provides an opportunity for your component to get the settings or values from the dialog and update your component’s data structure.

The MOM editor service provides functions such as MGetEditorItem() to get the values from the items on the dialog.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is “kDialogEditorType” |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_activateEvent;
} SampleComp;
```

then the MCompEditorAccept method could be:

```
MErr MCompEditorAccept(SampleComp *self, void *editor, short editorType)
{
 if ( editorType == kMDialogEditorType )
 {
     MGetEditorItem(editor, kActivatePopupItem, (MDataType *)
&self->f_activateEvent);
     return kMNoCompErr;
 }
 else
     return kMUnableToComplyCompErr;
}
```

where kActivatePopupItem is the item number of the pop-up in the class’s “DITL” resource.

**See Also**

“MCompEditorDecline” on page 4.168
“MCompEditorItemChanged” on page 4.169
“MCompEditorOpen” on page 4.171
“MSetEditorItem” on page 4.99
“Editor Construction Guide” on page 2.42
“Editing” on page 3.56
Chapter 5, “User Interface Guidelines”

## MCompEditorCustomItemDraw

```
MCompEditorCustomItemDraw (MSelf
   *self, void *editor, short
   editorType, short item, PLRect
   itemRect);
```

MCompEditorCustomItemDraw is called when a custom item in the editor dialog needs updating.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is “kDialogEditorType” |
| item | the item number of the custom item |
| ItemRect | the area of the custom item |

## MCompEditorCustomItemMouseEvent

```
MCompEditorCustomItemMouseEvent(M
   Self *self, void *editor, short
   editorType, short item, short
   eventType, PLPoint
   theMousePosition)
```

MCompEditorCustomItemMouseEvent is called to handle mouse events for custom items in the component's editor dialog.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is "kDialogEditorType" |
| item | the item number of the custom item |
| eventType | the event type |
| theMousePosition | the position of the mouse in the event |

## MCompEditorDecline

```
MCompEditorDecline( MSelf *self,
   void *editor )
```

The MCompEditorDecline method is called when the user clicks the “Cancel” button in the dialog.

Generally, you won’t override this method. However, you may override it if, for example, you want to cancel an asynchronous task that was started while your component’s dialog was open.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |

**Example**

If the following asynchronous task was started while your component’s dialog was open:

```
MPostCTimeTask(self, nil, kTimePeriod, kSampleSlot, kSampleTask);
self->f_taskActive = TRUE;
```

then the MCompEditorDecline method could be:

```
MErr MCompEditorDecline(SampleComp *self, void *editor, short editorType)
{
 if ( self->f_taskActive ) {
      MKillCTimeTask(self, nil);
      self->f_taskActive = FALSE;
 }
}
```

Note that you would also probably want to kill this asynchronous task from within your MCompEditorAccept method.

**See Also**

“MCompEditorAccept” on page 4.165
“MCompEditorItemChanged” on page 4.169
“MCompEditorOpen” on page 4.171
“MGetEditorItem” on page 4.94

## MCompEditorItemChanged

```
MCompEditorItemChanged( MSelf
   *self, void *editor, short
   editorType, short item, short
   part )
```

The MCompEditorItemChanged method is called when an item relating to your component has been changed in the editor. For example, this occurs when the user changes an item (e.g., clicks on a button or types in a field) in your component's dialog.

You may want to do any number of things when this method is called. For example if you have a button within your component dialog, MCompEditorItemChanged is called when the user clicks on the button. Also, if you have a pop-up menu that is only enabled when a certain checkbox is checked, then MCompEditorItemChanged will inform you when the checkbox changes state. You can then proceed to enable or disable the pop-up (with MSetEditorItemState) from within this method.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is "kDialogEditorType" |
| item | The item number of the item that changed (specified in the class's "DITL" resource). |
| part | The part number of certain multi-part dialog items, such as slider bars. |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MBoolean   f_checkbox;
 MPopup     f_popup;
} SampleComp;
```

then the MCompEditorItemChanged method could be:

```
MErr MCompEditorItemChanged(SampleComp *self, void *editor, short editorType,
      short item, short part)
{
 if ( item == kCheckboxItem ) {
      MBooleanchecked;

      MGetEditorItem(editor, kCheckboxItem, (MDataType *) &checked);
      MSetEditorItemState(editor, kPopupItem,
         checked ? kMItemEnable : kMItemDisable);
 }
 return kMNoCompErr;
}
```

**See Also**

"MGetEditorItem" on page 4.94
"MSetEditorItem" on page 4.99
"MSetEditorItemState" on page 4.100

## MCompEditorOpen

```
MCompEditorOpen( MSelf *self, void
    *editor, short editorType )
```

The MCompEditorOpen method is called when the mTropolis is about to show the editor dialog of your component. This occurs when the user double-clicks on a component in mTropolis. MCompEditorOpen provides your component with an opportunity to set up the editor with your data structure field values that can be seen and updated by the user. Use the MOM editor service functions such as MSetEditorItem() function to set up each field in the editor.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is “kDialogEditorType” |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_activateEvent;
} SampleComp;
```

then the MCompEditorOpen method could be:

```
MErr MCompEditorOpen(SampleComp *self, void *editor, short editorType)
{
 if ( editorType == kMDialogEditorType ) {
      MSetEditorItem(editor, kActivatePopupItem,
            (MDataType *) &self->f_activateEvent);
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

where “kActivatePopupItem” is the item number of the pop-up in the class’s “DITL” resource.

**See Also**

“MCompEditorAccept” on page 4.165
“MCompEditorDecline” on page 4.168
“MCompEditorItemChanged” on page 4.169
“MSetEditorItem” on page 4.99

## MCompEditorPrevalidateItem

```
MCompEditorPrevalidateItem(MSelf
   *self, void *editor, short
   editorType, short item,
   Boolean *bIsValid, char
   *errorstr);
```

The MCompEditorPrevalidateItem method is called to prevalidate any changes made to the item by the user.

**Parameters**

| | |
|---|---|
| self | your component |
| editor | the editor of your component |
| editorType | the editor type, currently supported is “kDialogEditorType” |
| short | the item number of the item to which changes have to made |
| bIsValid | set to true when the change is valid otherwise false |
| errorstr | the error message you'd like to display if the change is invalid, memory has been allocated for up to 255 char long null-terminated string |

**See Also**

“MCompEditorAccept” on page 4.165
“MCompEditorDecline” on page 4.168
“MCompEditorItemChanged” on page 4.169

## MCompEnabled

```
MCompEnabled( MSelf *self )
```

The MCompEnabled method is called whenever your component is enabled during runtime. Service-style components will get this method called as soon as runtime is entered. For modifier-style component, this may happen in one of two ways:

- if the component isn't in a switchable behavior, then it will get enabled when its scene starts; or
- if the component is in a switchable behavior, then it will get enabled whenever that behavior switches on.

This method will get called prior to any external event being sent (e.g., "Parent Enabled").

One example of what you may want to do in your MCompEnabled method is to register mTropolis' automatic mouse feedback. You'll notice that, in runtime, a modifier that is "listening" for a mouse event will change the default cursor when the mouse is over its element. This is accomplished by calling the MRegisterMouseFeedback function from within your MCompEnabled method.

**Parameters**

| | |
|---|---|
| self | your component |

**Example**
For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_activateEvent;
} SampleComp;
```

then the MCompEnabled method could be:

```
MErr MCompEnabled(SampleComp *self)
{
 MRegisterMouseFeedback(self, &self->f_activateEvent);
 return kMNoCompErr;
}
```

**See Also**
"MCompDisabled" on page 4.163
"MRegisterMouseFeedBack" on page 4.190

## MCompEnvironmentChanged

```
MCompEnvironmentChanged( MSelf
    *self, MEnvChangeReason
    reason, void *changeData )
```

Your component's MCompEnvironmentChanged method will be called when the component's surroundings have somehow changed. Currently, there are two cases when this will happen:

- your component is dragged into a different project, and
- when objects within the project have been duplicated.

Your component should call the MAdaptData function in response to this method being called. Your component should pass any events that it responds to in the datum field of MAdaptData. Your component should also pass any object IDs that it keeps track of. MAdaptData will take care of updating the given data item to reflect the changed environment.

**Parameters**

| | |
|---|---|
| self | pointer to your component |
| reason | reason for environment change. Pass this to MAdaptData() |
| changeData | data associated with the environment change. Pass this to MAdaptData() |

**Example**

```
static MErr MCompEnvironmentChanged(MSelf *self, MEnvChangeReason reason, void
*changeData)
{
 MAdaptData(reason, changeData, (MDataType *)&self->f_executeEvent);
 return kMNoCompErr;
}
```

## MCompFreeCache

```
MFreeCache( MSelf *self, long
   bytesneeded, short urgency )
```

MCompFreeCache is called when the system needs more memory and asks your component to free any cache it has. This method is currently unused.

**Parameters**

| | |
|---|---|
| self | your component |
| bytesneeded | number of bytes to free |
| urgency | the importance of the request |

## MCompGetAttribute

```
MCompGetAttribute(MSelf *self,
   MomID attribName, MDataType
   *selector , MDataType
   *dataValue)
```

The MCompGetAttribute method is called whenever an attribute of your component is requested. In your MCompGetAttribute method your component should provide the requested attribute in the dataValue parameter. All components using events should override this method and support the "events" attribute, which is used by the behavior modifier to graphically demonstrate what events a component responds to.

**Parameters**

| | |
|---|---|
| self | your component |
| attribName | The MOM ID of the attribute. Some standard MOM IDs are defined in the "MAttrib.h" file. |
| selector | Additional information used in selection process of requested attribute information. |
| dataValue | where the attribute is stored |

**Example**

For a component with the following data structure:

```
typedef struct PointVarComp {
 MPoint2D   f_point;
 MEvent     f_executeEvent;
} PointVarComp;
```

then its MCompGetAttribute could be:

```
static MErr MCompGetAttribute(GravitySrvComp *self, MomID attribName,
            MDataType *selector, MDataType *dataValue)
{
 //support for mTropolis behavior window by handling 'events' attribute
 if ( MCmpMomID(attribName, kMEventsAttrib) )
 {
      switch (selector->f_integer.f_value)
      {
         case 1:
            MCopyEvent(dataValue->f_event, self->f_enableEvent);
            return kMNoCompErr;
            break;
      default:
         return kMUnableToComplyCompErr;
         break;
      }
 }
else
     //example of support for kMValueAttrib
     if ( MCmpMomID(attribName, kMValueAttrib) )
     {
        dataValue->f_point.f_type = kMPoint2D;
        dataValue->f_point.f_value = self->f_point.f_value;
        return kMNoCompErr;
     }
     else
        return kMUnableToComplyCompErr;
  }
```

**See Also**

"MCompSetAttribute" on page 4.195

## MCompGetAttributeCount

```
MCompGetAttributeCount( MSelf
   *self, short *count )
```

MCompGetAttributeCount returns the number of attributes that the component supports.

**Parameters**

| | |
|---|---|
| self | your component |
| count | a pointer to the variable receiving the attribute count |

**See Also**

"MCompGetAttribute" on page 4.176
"MCompGetNthAttributeName" on page 4.183

## MCompGetAttributeMaxIndex

```
MCompGetAttributeMaxIndex( MSelf
   *self, MomID attribName, long
   *maxIndex )
```

MCompGetAttributeMaxIndex returns the highest index for the given attribute, which is of list or array type.

**Parameters**

| | |
|---|---|
| self | your component |
| attribName | the requested attribute |
| maxIndex | a pointer to the variable receiving the max index |

**See Also**

"MCompGetAttribute" on page 4.176
"MCompGetNthAttributeName" on page 4.183

## MGetElement

```
MGetElement( MObjectRef **elemRef
    )
```

MGetElement retrieves element containing the component.

**Parameters**

| | |
|---|---|
| elemRef | a pointer to a MObjectRef * variable that will be set to the element |

### Example

```
MObjectRef  *element;
MInteger    elementid;

MGetElement( &element );
MInitInteger(elementid, 0);
MGetElementAttribute( element, kMObjectIDAttrib, NULL, (MDataType *)&elementid);
```

### See Also

"MGetElementAttribute" on page 4.71

## MInvalidateElement

```
MInvlidateElement( Boolean
   immediateRedraw );
```

Components may call this method to invalidate the element that they are currently attached to, and may optionally have the element redrawn immediately.

**Parameters**

immediateRedaw
: set to TRUE to have the element redrawn immediately

## MCompGetCompID

```
MCompGetCompID(MSelf *self,
   CompID *theID)
```

Retrieves the Component ID of the given component.

**Parameters**

| | |
|---|---|
| self | pointer to component to retrieve CompID for |
| theID | CompID returned here |

## MCompGetNthAttributeName

```
MCompGetNthAttributeName( MSelf
    *self, short num, MomID
    attribName )
```

MCompGetNthAttributeName retrieves the attribute name from the given index.

**Parameters**

| | |
|---|---|
| self | your component |
| num | the attribute index value |
| attribName | the attribute name to return |

**Example**
The following function determines whether a component supports a specific attribute:

```
static Boolean MFindAttribute(SmapleComp *self, MomID attrib)
{
 short   count, i;
 MomIDname;

 MGetAttributeMaxIndex(self, *count);
        if (MError() != kMNoCompErr)
     return false;
 for (i=0; i<count; i++)
 {
     MGetNthAttributeName(self, name, i);
     if (MError() != kMNoCompErr)
        return false;
     if (MCmpMomID(name, attrib))
        return true;
 }
 return false;
}
```

**See Also**
“MCompGetAttributeMaxIndex” on page 4.179

## MGetNthClient

```
MGetNthClient( MSelf *service,
   short nthClient, MSelf
   **client )
```

A service calls MGetNthClient to get a pointer to a registered client.

**Parameters**

| | |
|---|---|
| service | the service |
| nthClient | an index between 0 and the return value of “MGetNumClients()”. |
| client | the client to return |

**Example**

For a gravity service that continually adjusts the velocity of its clients, it may communicate with them as follows:

```
static MErr GravityTask(GravityComp *self, MTInfo *tinfo)
{
 MErr   ret;
 short  i, numClients;
 MSelf  *client;

 MGetNumClients(self, &numClients);
 if ((ret = MError()) != kMNoCompErr)
      return ret;
 for (i=0; i<numClients; i++)
 {
      MGetNthClient(self, i, &client);
      // ...
 }
}
```

**See Also**

“MGetNumClients” on page 4.185

## MGetNumClients

```
MGetNumClients( MSelf *service,
    short *numClients )
```

A service calls MGetNumClients to get the number of currently registered clients, for example, a service may want to communicate with each of its clients. To accomplish this, it would first call MGetNumClients, then call MGetNthClient for each client.

**Parameters**

| | |
|---|---|
| service | the service |
| numClients | the number of clients to return |

**Example**

For a gravity service that continually adjusts the velocity of its clients, it may communicate with them as follows:

```
static MErr GravityTask(GravityComp *self, MTInfo *tinfo)
{
 short  i, numClients;
 MSelf  client;
 MErr   ret;

 MGetNumClients(self, &numClients);
 if ((ret = MError()) != kMNoCompErr)
      return ret;
 for (i=0; i<numClients; i++)
 {
      MGetNthClient(self, i, &client);
      // ...
 }
}
```

**See Also**

"MGetNthClient" on page 4.184

## MCompGetSaveInfo

```
MCompGetSaveInfo( MSelf *self,
    MStream *file, long saveInfo,
    short *rev, long *len )
```

The MCompGetSaveInfo method is called whenever your component is about to be saved.

Your MCompGetSaveInfo method should provide the editor with the aggregate length of all the fields from your data structure that you wish to save (MSizeOfValue() can used called for each field to be saved), and to provide a current revision number associated with your data structure. MOM requires this information prior to calling your MCompSaveComponent method.

**Parameters**

| | |
|---|---|
| self | your component |
| file | the file service that saves your component |
| saveInfo | the mask for saving options, currently defined are: kMSavingProjectMask kMSavingTitleMask |
| rev | the revision number of your component |
| len | the total length of your component to be saved |

**Example**

For a class whose data structure and revision number are:

```
typedef struct SampleComp {
  MEvent     f_event;
  MInteger   f_integer;
} SampleComp;

const short kSampleRev = 0;

MErr MCompGetSaveInfo(MSelf *self, MStream *file, long saveInfo, short *rev, long
*len)
{
  long   valueSize;

  MSizeOfValue(file, &self->f_event, &valueSize);
  *len = valueSize;
  MSizeOfValue(file, &self->f_integer, &valueSize);
  *len += valueSize;
  *rev = kSampleRev;
  return kMNoCompErr;
}
```

**See Also**

"MCompRestoreComponent" on page 4.192
"MCompSaveComponent" on page 4.194
"MSizeOfValue" on page 4.111

## MCompProcessMessage

```
MCompProcessMessage( MSelf *self,
   MMessagePtr message )
```

The MCompProcessMessage method is called when your component receives a message, either in runtime or in edit mode.

For many components (e.g., modifiers), this method is where most of the work will be performed (or at least initiated), since users configure them to act upon the response of an event. Your component must check to see if the incoming message contains the event chosen as a "trigger" by the user, which is accomplished by calling the MDetectMessage() macro.

Many components are also configured to deactivate upon the response of an event. After checking to make sure that the incoming message contains the deactivate event chosen by the user, you will want to cancel any actions, motions, etc., that are still in effect. For example, a graphic modifier has both an "Apply When" and a "Remove When" event pop-up, and it removes the graphic effect when the "Remove When" event is detected.

Note that in addition to deactivating your component upon receiving a user-specified event, you will also likely want to deactivate it then the component becomes disabled. For example, consider a graphic modifier with its "Remove When" pop-up set to "Mouse Down" in a behavior. When the behavior switches off, you'll want to remove the graphic effect, even though "Mouse Down" hasn't actually been received by the graphic modifier.

To handle this situation, you need to detect when your component gets disabled, which can be determined in two ways. First, note that your MCompDisabled method will be called whenever your component disables in runtime. Thus, the graphic modifier's code to remove an effect could be placed in this method.

But since this code already exists in MCompProcessMessage (to handle removing the graphic effect when the user-specified event is received), it would be more convenient to know within MCompProcessMessage when the component is about to be disabled. This is accomplished by calling the MDetectDisableMessage() macro in addition to MDetectMessage() when testing to see if you should remove the graphic effect, etc. See the example below.

If your CompProcessMessage responded in some way to the message, then "kMNoCompErr" should be returned; otherwise, "kMUnableToComplyCompErr" should be returned. These return values are monitored by the message log mechanism.

**Parameters**

| | |
|---|---|
| self | your component |
| message | the message data structure |

**Example**

For a simple graphic modifier whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_applyEvent;
 MEvent  f_removeEvent;
} SampleComp;
```

then the MCompProcessMessage method could be:

```
MErr MCompProcessMessage(SampleComp *self, MMessagePtr message)
{
 if ( MDetectMessage(message, self->f_applyEvent) )
 {
      // code to apply the graphic effect...
      return kMNoCompErr;
 }
 else
 if ( MDetectMessage(message, self->f_removeEvent) ||
MDetectDisableMessage(message) )
 {
      // code to remove the graphic effect...
      return kMNoCompErr;
 }
      return kMUnableToComplyCompErr;
}
```

**See Also**

MDetectMessage defined in MEvent.h
MDetectDisableMessage defined in MEvent.h

## MRegisterMouseFeedBack

```
MRegisterMouseFeedBack( MSelf
   *self, void* primData )
```

This function registers your component to receive mouse input events with the messaging system.

**Parameters**

| | |
|---|---|
| self | your component |
| primData | the event to be registered to receive |

**Example**

```
static MErr MCompEnabled(YourOwnComp *self )
{
 MRegisterMouseFeedBack( self, &self->f_activateEvent );
 return kMNoCompErr;
}
```

**See Also**

"MDeregisterMouseFeedBack" on page 4.162

## MCompReset

```
MCompReset( MSelf *self )
```

This function resets the component. It is currently unused.

**Parameters**

| | |
|---|---|
| self | your component |

## MCompRestoreComponent

```
MCompRestoreComponent( MSelf
   *self, MStream *file, long
   saveInfo, short rev )
```

The MCompRestoreComponent method is called when your component is getting restored from a file that was previously saved. Your MCompRestoreComponent method should read in your component's various field values that were previously saved. This is accomplished by calling the MReadValue() function for each field to be restored.

**Parameters**

| | |
|---|---|
| self | your component |
| file | the file service from which your component is read and restored |
| saveInfo | the options used in saving the component. The saveInfo parameter contains information about the project that is being saved. Each bit in the 32-bit value represents particular information about the save process. These constants are defined in MPersist.h. Brief descriptions are provided in the tables below. Table 4.1 described project-related save constants. Platform-related save information is shown in Table 4.2. These values are stored in the high byte of the saveInfo parameter. You use the provided constants kTargetPlatformMask and kPlatformShiftBits to access these values. |

| Constant | Description |
|---|---|
| kSwapBytesMask | bytes need to be swapped for save. Note: The kSwapBytesMask is informational only. If you use the provided mFactory save and restore methods, then byte swapping is done for you. |
| kSaveNamesMask | Names of objects are being saved |
| kSaveThumbsMask | Thumbnails are being saved with the project |
| kSavingLibraryMask | Project being saved is a library |
| kSavingProjectMask | Project being saved is an editor project |
| kSavingTitleMask | Project is being built into a title |
| kSavingPersistentMask | The Save and Restore modifier is being used |
| kUsingAVIMask | AVI data is being saved |
| kUsingExternalAssetMask | Assets are saved externally |

*Table 4.1: Project-Related Save Constants*

| Constant |
|---|
| MacSysPlatform |
| Window3xPlatform |
| Windows32Platorm |
| Windows95Platform |
| WindowsNTx86Platform |
| Mac68KPlatform |
| MacPPCPlatform |

*Table 4.2: Platform-Related Save Information*

| | |
|---|---|
| rev | the revision number of your component that was saved |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent  f_event;
 MIntegerf_integer;
} SampleComp;

const short kSampleRev = 0;
```

then the MCompRestoreComponent method could be:

```
MErr MCompRestoreComponent(MSelf *self, MStream *file, long saveInfo)
{
 if ( rev == kSampleRev )
 {
      MReadValue(file, &self->f_event);
      MReadValue(file, &self->f_integer);
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

**See Also**

"MCompGetSaveInfo" on page 4.186
"MCompSaveComponent" on page 4.194
"MReadValue" on page 4.110

## MCompSaveComponent

```
MCompSaveComponent( MSelf *self,
   MStream *file, long saveInfo )
```

The MCompSaveComponent method is called when your component is to be saved to a file.

Your MCompSaveComponent method should write out your component’s various field values. This is accomplished by calling the MWriteValue() function for each field to be saved. Note that, prior to saving your component, you must provide certain information to MOM via the MCompGetSaveInfo method.

**Parameters**

| | |
|---|---|
| self | your component |
| file | the file service to which your component is written and saved |
| saveInfo | the options for saving the component. Refer to Table 4.1 and Table 4.2 on page 193 for a complete list of options. |

**Example**

For a class whose data structure is:

```
typedef struct SampleComp {
 MEvent    f_event;
 MInteger  f_integer;
} SampleComp;
```

then the MCompSaveComponent method could be:

```
MErr MCompSaveComponent(MSelf *self, MStream *file, long saveInfo)
{
 MWriteValue(file, &self->f_event);
 MWriteValue(file, &self->f_integer);
 return kMNoCompErr;
}
```

**See Also**

“MCompGetSaveInfo” on page 4.186
“MCompRestoreComponent” on page 4.192
“MWriteValue” on page 4.115

## MCompSetAttribute

```
MCompSetAttribute(MSelf *self,
   MomID attribName, MDataType
   *selector, MDataType
   *dataValue)
```

The MCompSetAttribute method is called whenever a component is changing an attribute associated with your component. For example, a gravity service may get the mass and position of your component via MCompGetAttribute, then calculate its new velocity and set it via MCompSetAttribute (note that this is just an example, and not the recommended way for components to communicate with each other). Your MCompSetAttribute method should update your data structure with the new attribute value.

**Parameters**

| | |
|---|---|
| self | your component |
| attribName | The MOM ID of the attribute. Some standard MOM IDs are defined in the “MAttrib.h” MOM header file; others are defined by the actual component. |
| selector | additional information of various uses |
| dataValue | the new attribute value |

**Example**

```
typedef struct PointVarComp {
 MPoint2Df_point;
} PointVarComp;

MErr MCompSetAttribute(PointVarComp *self, MomID attribName, MDataType selector,
      MDataType *dataValue)
{
 if ( MCmpMomID(attribName, kMValueAttrib) &&
      (dataValue->f_type.f_type == kMPoint2D) )
 {
      self->f_point = dataValue->f_point;
      return kMNoCompErr;
 }
 else if ( MCmpMomID(attribName, kXAttrib) &&
      (dataValue->f_type.f_type == kMInteger) )
 {
      self->f_point.f_value.PNTX = dataValue->f_point.f_value.PNTX;
      return kMNoCompErr;
 }
 else if ( MCmpMomID(attribName, kYAttrib) &&
      (dataValue->f_type.f_type == kMInteger) )
 {
      self->f_point.f_value.PNTY = dataValue->f_point.f_value.PNTY;
      return kMNoCompErr;
```

```
  }
  else
       return kMUnableToComplyCompErr;
}
```

**See Also**

"MCompGetAttribute" on page 4.176
"MCompSetAttribute" on page 4.195

## MOM MEMORY SERVICE

The MOM memory service provides functions of pointer-based and handle-based memory management. MOM pointer-based memory is not resizable. Developers are encouraged to use these memory management routines. In the future, mFactory will enhance these routines to provide memory tracking utilities. Note that some of these routines use PL-types, indicating that at compile time these types will be resolved to platform-specific definitions.

| Memory Function | Methods |
|---|---|
| Pointer-based routines | MNewPtr, MDisposePtr, MRegisterForDeleteNotify |
| Handle-based routines | MNewHandle, MDisposeHandle, MGetHandleSize, MSetHandleSize, MLockHandle, MUnlockHandle |

## MDisposeHandle

```
MDisposeHandle( PLHugeHandle
   theHandle )
```

MDisposeHandle disposes of a memory handle previously allocated by MNewHandle.

**Parameters**

| | |
|---|---|
| theHandle | a previously allocated PLHugeHandle |

**Example**

```
static void handleMemoryTest(YourComp *self)
{
  PLHugeHandleh=nil;
  char         *ptr=nil;

  theSize = 4096;
  MNewHandle( theSize, &h );
  if (MError() != kMNoCompErr || !h)
       return;

  MGetHandleSize( hndl, &theSize );
  if (MError() != kMNoCompErr)
       return;
  theSize *= 2;
  MSetHandleSize( h, &theSize );
  if (MError() != kMNoCompErr)
       return;

  MLockHandle( h, &ptr);
  MUnlockHandle( h );
  MDisposeHandle( &h );
}
```

**See Also**

"MNewHandle" on page 4.202

## MDisposePtr

```
MDisposePtr( void* thePtr )
```

MDisposePtr disposes a pointer previously allocated by MNewPtr.

**Parameters**

| | |
|---|---|
| thePtr | the pointer to be freed |

**Example**

```
static void PointerMemoryTest(YourComp *self)
{
 char    *ptr = nil;
 UInt32  theSize = 1024;

 MNewPtr(theSize, &ptr );
 if (MError() != kMNoCompErr || !ptr)
      return;
 MDisposePtr(ptr );
}
```

**See Also**

“MNewPtr” on page 4.203

## MGetHandleSize

```
MGetHandleSize( PLHugeHandle
   theHandle, UInt32 *theSize )
```

MGetHandleSize returns the size of the memory handle in bytes.

**Parameters**

| | |
|---|---|
| theHandle | the memory handle |
| theSize | where the returned size goes |

**Example**

```
static void handleMemoryTest(YourComp *self)
 {
      PLHugeHandleh=nil;
      char        *ptr=nil;

      theSize = 409600;
      MNewHandle( theSize, &h );
      if (MError() != kMNoCompErr || !h)
         return;

      MGetHandleSize( hndl, &theSize );
      if (MError() != kMNoCompErr)
         return;
      theSize *= 2;
      MSetHandleSize( h, &theSize );
      if (MError() != kMNoCompErr)
         return;

      MLockHandle( h, &ptr);
      MUnlockHandle( h );
      MDisposeHandle( &h );
 }
```

**See Also**

"MSetHandleSize" on page 4.204
"MNewHandle" on page 4.202

## MLockHandle

```
MLockHandle( PLHugeHandle
   theHandle, void **thePtr )
```

MLockHandle locks the memory handle and returns the locked memory pointer.

**Parameters**

| | |
|---|---|
| theHandle | the memory handle |
| thePtr | the returned locked memory pointer |

**Example**

```
static void handleMemoryTest(YourComp *self)
 {
      PLHugeHandleh=nil;
      char        *ptr=nil;

      theSize = 409600;
      MNewHandle( theSize, &h );
      if (MError() != kMNoCompErr || !h)
         return;

      MGetHandleSize( hndl, &theSize );
      if (MError() != kMNoCompErr)
         return;
      theSize *= 2;
      MSetHandleSize( h, &theSize );
      if (MError() != kMNoCompErr)
         return;

      MLockHandle( h, &ptr);
      MUnlockHandle( h );
      MDisposeHandle( &h );
 }
```

**See Also**

“MUnlockHandle” on page 4.205

## MNewHandle

```
MNewHandle( UInt32 theSize,
   PLHugeHandle *theHandle )
```

MNewHandle allocates a memory handle of a given size.

**Parameters**

| | |
|---|---|
| theSize | the requested size |
| theHandle | the returned allocated memory handle |

**Example**

```
static void handleMemoryTest(YourComp *self)
 {
      PLHugeHandleh=nil;
      char        *ptr=nil;

      theSize = 409600;
      MNewHandle( theSize, &h );
      if (MError() != kMNoCompErr || !h)
         return;

      MGetHandleSize( hndl, &theSize );
      if (MError() != kMNoCompErr)
         return;
      theSize *= 2;
      MSetHandleSize( h, &theSize );
      if (MError() != kMNoCompErr)
         return;

      MLockHandle( h, &ptr);
      MUnlockHandle( h );
      MDisposeHandle( &h );
   }
```

**See Also**

“MDisposeHandle” on page 4.198

## MNewPtr

```
MNewPtr( UInt32 theSize, void
    **thePtr )
```

MNewPtr allocates a memory pointer of the given size.

**Parameters**

| | |
|---|---|
| theSize | the requested memory size |
| thePtr | the returned allocated memory pointer |

### Example

```
static void PointerMemoryTest(YourComp *self)
{
 char    *ptr = nil;
 UInt32  theSize = 1024;

 MNewPtr(theSize, &ptr );
 if (MError() != kMNoCompErr || !ptr)
      return;
 MDisposePtr(ptr );
}
```

### See Also

"MDisposePtr" on page 4.199

## MSetHandleSize

```
MSetHandleSize( PLHugeHandle
   theHandle, UInt32 theSize )
```

MSetHandleSize resizes the memory handle to a new size.

**Parameters**

| | |
|---|---|
| theHandle | the memory handle |
| theSize | the new size for the memory handle |

**Example**

```
static void handleMemoryTest(YourComp *self)
 {
     PLHugeHandleh=nil;
     char        *ptr=nil;

     theSize = 409600;
     MNewHandle( theSize, &h );
     if (MError() != kMNoCompErr || !h)
        return;

     MGetHandleSize( hndl, &theSize );
     if (MError() != kMNoCompErr)
        return;
     theSize *= 2;
     MSetHandleSize( h, &theSize );
     if (MError() != kMNoCompErr)
        return;

     MLockHandle( h, &ptr);
     MUnlockHandle( h );
     MDisposeHandle( &h );
   }
```

**See Also**

"MGetHandleSize" on page 4.200

## MUnlockHandle

```
MUnlockHandle( PLHugeHandle
   theHandle )
```

MUnlockHandle unlocks a memory handle, and invalidates the locked pointer returned from the previous MLockHandle.

**Parameters**

| | |
|---|---|
| theHandle | the memory handle |

**Example**

```
static void handleMemoryTest(YourComp *self)
 {
     PLHugeHandleh=nil;
     char        *ptr=nil;

     theSize = 409600;
     MNewHandle( theSize, &h );
     if (MError() != kMNoCompErr || !h)
        return;

     MGetHandleSize( hndl, &theSize );
     if (MError() != kMNoCompErr)
        return;
     theSize *= 2;
     MSetHandleSize( h, &theSize );
     if (MError() != kMNoCompErr)
        return;

     MLockHandle( h, &ptr);
     MUnlockHandle( h );
     MDisposeHandle( &h );
   }
```

**See Also**

“MLockHandle” on page 4.201

## MRegisterForDeleteNotify

```
MRegisterForDeleteNotify(MObjectP
   tr *theObjPtr);
```

MRegisterForDeleteNotify is used register a pointer to an object with the object that it points to. These pointers are maintained in a list by the object. When the object is deleted, it resets all of the pointers in the list to NULL.

For example, suppose a messenger modifier maintains a pointer to an object that is its target. If the target is deleted, then the pointer maintained by the messenger is now invalid. By registering the pointer with object, it will automatically be set to NULL when the object is deleted. The messenger can then check the pointer before using it to make sure that it is not NULL.

**Parameters**

| | |
|---|---|
| theObjPtr | Pointer to an object |

## MOM MISCELLANEOUS SERVICE

The MOM Miscellaneous Service provides methods which perform several useful functions that components may take advantage of.

| Miscellaneous Functionality | Methods |
|---|---|
| Time Utilities | MGetGlobalTime |
| Random Numbers | MGetRandomValue, MGetRandomRange |
| Cursor Management | MSetCursor |
| Adapting Data to Environment Changes | MAdaptData |
| Mouse Handling | MGetMouse, MStillDown |
| Keyboard Handling | MKeyIsDown |
| Screen Functions | MGetScreenRect |
| Selection Handling | MGetSelection |
| Alerts | MStopAlert |
| Path Keyword Handling | MParsePathFromString |
| Project Utilities | MOpenProject |

## MGetGlobalTime

`MGetGlobalTime( long *theTime )`

MGetGlobalTime returns the current system time (in ticks).

**Parameters**

| | |
|---|---|
| theTime | a pointer to a long receiving the current time |

**Example**

```
static void getTime(YourComp *self)
{
  long time;
  MGetGlobalTime( &time );
}
```

## MGetRandomValue

```
MGetRandomValue( long
   *randomValue )
```

MGetRandomValue returns a random number ranging from 0 to 32767.

**Parameters**

randomValuea pointer to a long receiving the random number

**Example**

```
static void getRandomNumber(YourComp *self)
{
  long num;
  MGetRandomValue( &num);
}
```

**See Also**

“MGetRandomRange” on page 4.210

## MGetRandomRange

```
MGetRandomRange( long range1, long
   range2, long *randomRange )
```

MGetRandomRange returns a random number ranging from range1 to range2.

**Parameters**

| | |
|---|---|
| range1 | one end of the range for the randomRange to fall within |
| range2 | another end of the range for the randomRange to fall within |
| randomRange | a pointer to the buffer receiving the randomRange |

**Example**

```
static void getRandomRange(YourComp *self)
{
 long num, r1 = 4567, r2=6789;
 MGetRandomRange(r1, r2, &num);
}
```

**See Also**

“MGetRandomValue” on page 4.209

## MSetCursor

```
MSetCursor( MSelf *self, long
   cursorID )
```

MSetCursor sets the current cursor to the one with the given cursor ID.

**Parameters**

| | |
|---|---|
| self | your component |
| cursorID | the cursor ID of the new cursor |

## MAdaptData

```
MAdaptData(MEnvChangeReason
   reason, void *changeData,
   MDataType *datum)
```

MAdaptData should be called by components in their MCompEnvironmentChanged method. Typically, this will need to be done by modifiers.

The MCompEnvironmentChanged method gets called when the component's environment has changed in certain ways. Currently, this includes copying the component to another project, and duplicating objects that the component may point to.

The MAdaptData routine should be called with the component's event data (the events that the component activate and deactivate on). It should also be called with any object IDs that the component keeps track of.

**Parameters:**

| | |
|---|---|
| reason | the reason the environment is changing, passed in by MCompEnvironmentChanged |
| changeData | data associated with the change, passed in by MCompEnvironmentChanged |
| datum | the MDataType to update in response to the change |

### Example

```
static MErr MCompEnvironmentChanged(MSelf *self, MEnvChangeReason reason,
            void *changeData)
{
 MAdaptData(reason, changeData, (MDataType *)&self->f_executeEvent);
 return kMNoCompErr;
}
```

## MGetMouse

```
MGetMouse(PLPoint *mousePt)
```

MGetMouse retrieves the current location of the mouse as a point.

**Parameters**

| | |
|---|---|
| mousePt | returns point location of mouse |

## MStillDown

```
MStillDown(short *stillDown)
```

MStillDown returns TRUE if the mouse is still down from its original click.

**Parameters**

| | |
|---|---|
| stillDown | TRUE if the mouse is still down from its click |

## MKeyIsDown

```
MKeyIsDown(short keyID, short
   *isDown)
```

MKeyIsDown returns TRUE if a key with the given ID is down. Key codes are listed in the “MKeyCode.h” header file.

**Parameters**

| | |
|---|---|
| keyID | ID of key in question. A list of key IDs is in MKeyCode.h. |
| isDown | TRUE if the key is down |

## MGetScreenRect

```
MGetScreenRect( PLRect
   *screenRect )
```

MGetScreenRect returns a rectangle which determines the dimensions of the title's screen.

**Parameters**

| | |
|---|---|
| screenRect | returns rectangle which defines current screen |

## MGetSelection

```
MGetSelection( MObjectRef
   **selList, Boolean
   deselectChildren )
```

MGetSelection returns a list of the currently-selected objects in the mTropolis editor environment, with an option to deselect all objects' children before running the list.

**Parameters**

| | |
|---|---|
| selList | returns list of selected objects |
| deselectChildren | set to TRUE if child objects should be deselected |

## MStopAlert

```
MStopAlert(char *msg)
```

MStopAlert displays a platform-specific “Stop Alert” dialog containing a text message.

**Parameters**

| | |
|---|---|
| msg | text message to display in the dialog |

## MParsePathFromString

```
MParsePathFromString(MSelf
   *self, char *inPath, char
   *outPath)
```

MParsePathFromString accepts a string containing path keywords and resolves it into an actual filepath. For example, the string "<Preferences Folder On Macintosh>:MyFolder" will resolve into "HD:System Folder:Preferences:MyFolder:"

**Parameters:**

| | |
|---|---|
| self | pointer to component |
| inPath | path string containing keywords |
| outPath | string containing resolved path |

## MOpenProject

```
MOpenProject(MSelf *self,
   PLFileSpec *projFile, long
   loadFlags)
```

MOpenProject launches a project file specified by projFile using the options specified by loadFlags.

**Parameters**

| | |
|---|---|
| self | pointer to component |
| projFile | file specification of project to launch |
| loadFlags | options to launch project with. Currently supported options are “kMReplaceProject” and “kMAddToReturnList”. |

**Example**

```
MCompStdFuncDcl MCompProcessMessage(OpenTitleComp *self, MMessagePtr message)
{
 if (MDetectMessage(message, self->f_executeEvent))
 {
      if (self->f_validpath)
      {
         PLFileSpecprojectSpec;

         projectSpec.f_pathspec.f_path = self->f_filepath.f_ptr;
         projectSpec.f_pathspec.f_type = 0;
         MOpenProject(self, &projectSpec, self->f_options.f_value);
      }
      return kMNoCompErr;
 }
 else
      return kMUnableToComplyCompErr;
}
```

## MOM STRING SERVICE

The MOM string service provides MOM wrappers for the standard C runtime string functions, as well as other string-related routines. The primary purpose of this service is to assist MOM developers in managing their component size. By making these standard string functions available through a MOM service, the string functions themselves do not have to be included in the actual runtime component, thus reducing runtime size requirements.

| String Functionality | Methods |
|---|---|
| String Creation and Destruction | MNewString, MFreeString, MDupeString |
| Memory Manipulation | MMemset, MMemcpy, MMemcmp, MMemmove, MMemchr |
| Time String Utilities | MLongTimeFromString, MTimeStringFromLongTime |
| ANSI String Functions | MSubstituteString, MStrcpy, MStrncpy, MStrcat, MStrncat, MStrcmp, MStrncmp, MStrchr, MStrrchr, MStrspn, MStrcspn, MStrpbrk, MStrstr, MStrlen, MStrtok, MStrcoll, MStrerror, MStrxfrm |

## MDupeString

```
MDupeString(char **destination,
    const char *source )
```

This function duplicates the source string to the destination string by allocating memory for the destination string and copying the content of the source into the destination.

**Parameters**

| | |
|---|---|
| destination | the returned destination string |
| source | the null-terminated string to be duplicated |

**Example**

```
static void stringTest(MSelf *self)
{
  char *astring = nil;

  MDupeString(&astring, "this is a source string");
  if (MError() != kMNoCompErr || !astring)
       return;
  //...
    MFreeString(astring);
}
```

**See Also**

"MFreeString" on page 4.223
"MNewString" on page 4.224

## MFreeString

`MFreeString(char *str)`

This function frees the string created by either MNewString or MDupeString.

**Parameters**

| | |
|---|---|
| str | the string to be freed |

**Example**

```
static void stringTest(MSelf *self)
{
  char *astring = nil;

  MDupeString(&astring, "this is a source string");
  if (MError() != kMNoCompErr || !astring)
        return;
  //...
    MFreeString(astring);
}
```

**See Also**

"MNewString" on page 4.224
"MDupeString" on page 4.222

## MNewString

```
MNewString(char **destination,
    size_t size)
```

This function creates a string of the given length, the created string should be freed by MDisposeString.

**Parameters**

| | |
|---|---|
| destination | the returned destination string |
| size | the size of the string, excluding the null terminator |

**Example**

```
static void stringTest(MSelf *self)
{
 char *astring = nil;

 MNewString(&astring, (size_t)256);
 if (MError() != kMNoCompErr || !astring)
      return;
 //...
  MFreeString(astring);
}
```

**See Also**

"MFreeString" on page 4.223
"MDupeString" on page 4.222

## MLongTimeFromString

```
MLongTimeFromString(char *s, long
   *timeValue)
```

This function accepts an 8-character string (representing time) in the form “00:00.00” (minutes:seconds.hundredths) and converts it into a long time value. If the passed string is not 8 characters long, “kMUnableToComplyCompErr” is returned.

**Parameters**

| | |
|---|---|
| s | string to convert to long time |
| timeValue | long time value returned here |

## MMemchr

```
MMemchr(void **result, const void
   *source, int c, size_t length)
```

MMemchr finds a character in a block of memory.

**Parameters**

| | |
|---|---|
| result | result of the call. If the character is found, source is returned, otherwise NULL |
| source | pointer to memory to look in |
| c | character to look for |
| length | length of memory block pointed to by source |

## MMemcmp

```
MMemcmp(int *result, const void
    *source1, const void *source2,
    size_t length)
```

Compare two blocks of memory.

**Parameters**

| | |
|---|---|
| result | result of the function. If source1 > source2, returns a positive integer. If source1 = source2, returns 0. If source1 < source2, returns a negative integer |
| source1 | pointer to first block of memory |
| source2 | pointer to second block of memory |
| length | number of bytes to compare |

## MMemcpy

```
MMemcpy(void *destination, const
   void *source, size_t length)
```

This routine copies memory from one location to another. Do not use this routine if the two blocks may overlap. Use MMemmove instead.

**Parameters**

| | |
|---|---|
| destination | destination to copy bytes to |
| source | source of bytes to copy |
| length | number of bytes to copy |

## MMemmove

```
MMemmove(void *destination, const
   void *source, size_t length)
```

This routine copies bytes from one location to another, even if they overlap.

**Parameters**

| | |
|---|---|
| destination | destination to copy bytes to |
| source | source of bytes to copy |
| length | number of bytes to copy |

## MMemset

```
MMemset(void *destination, int c,
   size_t length)
```

This routine initializes a block of memory with a given value.

**Parameters**

| | |
|---|---|
| destination | pointer to block of memory to initialize |
| c | data to initialize destination with |
| length | number of bytes to initialize |

## MStrcat

```
MStrcat(char *destination, const
   char *source )
```

This function appends the source string to the destination string.

**Parameters**

| | |
|---|---|
| destination | the destination string |
| source | the source string |

## MStrchr

```
MStrchr( char **retcharpos, const
   char *str, int ch )
```

This function finds the first occurrence of the character “ch” in the string “str” and places the position in “retcharpos”. If the character is not found, the position value is set NULL.

**Parameters**

| | |
|---|---|
| retcharpos | the returned position of the found character |
| str | the string to be searched upon |
| ch | the character to be located |

## MStrcmp

```
MStrcmp( int *retval, const char
   *str1, const char *str2 )
```

This function lexigraphically compares str2 to str1.

**Parameters**

| | |
|---|---|
| retval | a pointer to the result of the comparison with the following values and meanings:<br>< 0 str1 is less than str2<br>= 0 str1 is equal to str2<br>> 0 str1 is greater than str2 |
| str1 | a pointer to a string |
| str2 | a pointer to the comparison string |

## MStrcoll

```
MStrcoll(int *result, const void
   *source1, const void *source2)
```

This routine compares two strings, interpreted according to the current locale. This is useful only on Windows, on the Macintosh, only the current locale is supported (the Mac provides its own functions for locale-based string comparison).

**Parameters**

| | |
|---|---|
| result | returns result. If source1 > source2, returns positive. If source1 = source2, returns 0. If source1 < source2, returns negative. |
| source1 | pointer to first string |
| source2 | pointer to second string |

## MStrcpy

```
MStrcpy(char *destination, const
   char *source )
```

This function copies the source string to the destination string .

**Parameters**

| | |
|---|---|
| destination | the destination string |
| source | the source string |

## MStrcspn

```
MStrcspn( size_t *index, const
   char *str1, const char *str2 )
```

This function returns the index of the first occurrence of the character in `str1` that does belong to the set of characters specified by str2.

**Parameters**

| | |
|---|---|
| index | the returned index |
| str1 | the search string |
| str2 | the character set |

## MStrerror

```
MStrerror(char **result, int err)
```

This routine returns an error message appropriate for the error code in err.

**Parameters**

| | |
|---|---|
| result | resultant error message for err |
| err | error code |

## MStrlen

```
MStrlen( size_t *length, const
   char *str )
```

This function finds the length of the string, str. The terminating null character is not counted.

**Parameters**

| | |
|---|---|
| length | a pointer for the length of the string |
| str | a pointer to string whose length is being found |

## MStrncat

```
MStrncat( char *destination, const
   char *source, size_t count )
```

This function appends “count” bytes from the source string to the destination string.

**Parameters**

| | |
|---|---|
| destination | a pointer to the null-terminated buffer for the appending string |
| source | a pointer to the appending string |
| count | the number of bytes to be copied |

## MStrncmp

```
MStrncmp( int *retval, const char
   *str1, const char *str2, size_t
   count )
```

This function lexigraphically compares "count" bytes of str2 to str1.

**Parameters**

| | |
|---|---|
| retval | a pointer to the result of the comparison with the following values and meanings:<br>< 0 str1 is less than str2<br>= 0 str1 is equal to str2<br>> 0 str1 is greater than str2 |
| str1 | a pointer to a string |
| str2 | a pointer to the comparison string |
| count | the number of bytes to compare |

## MStrncpy

```
MStrncpy(char *destination, const
   char *source, size_t count )
```

This function copies “count” number of bytes from the source string to the destination string.

**Parameters**

| | |
|---|---|
| destination | a pointer to the buffer for the resultant string |
| source | a pointer to the origination string |
| count | the number of bytes to copy |

## MStrpbrk

```
MStrpbrk( char **retcharpos, const
   char *str1, const char *str2 )
```

This function scans str1 for any character found in the str2 character set.

**Parameters**

| | |
|---|---|
| retcharpos | a pointer to the first occurrence in str1 of any character found in the str2 character set |
| str1 | a pointer to the search string |
| str2 | a pointer to the character set |

## MStrrchr

```
MStrrchr( char **retcharpos, const
   char *str, int ch )
```

This function finds the last occurrence of the character “ch” in the string “str” and places the address of the found location in “retcharpos”. If the character is not found, the position value is set NULL.

**Parameters**

| | |
|---|---|
| retcharpos | a pointer to the first occurrence of ch |
| str | a pointer to the string being searched |
| ch | the character to be located |

## MStrspn

```
MStrspn( size_t *index, const char
   *str1, const char *str2 )
```

This function returns the index of the first occurrence of the character in `str1` that does not belong to the set of characters specified by str2.

**Parameters**

| | |
|---|---|
| index | a pointer to the resultant index value |
| str1 | a pointer to the searched string |
| str2 | a pointer to the character set |

## MStrstr

```
MStrstr( char **retstring, const
   char *str1, const char *str2 )
```

This function scans str1 for the substring str2.

**Parameters**

| | |
|---|---|
| retstring | a pointer to the first occurrence of str2 in str1, NULL if not found |
| str1 | a pointer to the scanned string |
| str2 | a pointer to search string |

## MStrtok

```
MStrtok( char **retstring, char
   *str1, const char *str2 )
```

This function finds the next token in str1, from any token found in str2, and places the address of the found token in retstring. To find the *first* token in a string, str1 must point to the search string. For finding subsequent tokens, str1 should be NULL.

**Parameters**

| | |
|---|---|
| retstring | a pointer to the resultant string |
| str1 | a pointer to the search string |
| str2 | a pointer to token string |

## MStrxfrm

```
MStrxfrm(size_t *result, char
   *dest, const char *source,
   size_t length)
```

This function transforms and copies bytes from one string to another. It is provided only for ANSI compatibility under Windows. The Macintosh uses its own utilities for localizing string data. Use the International Utilities found in Inside Macintosh.

**Parameters**

| | |
|---|---|
| result | result of function, length of source |
| dest | destination string |
| source | source string |
| length | length of source string |

## MSubstituteString

```
MSubstituteString(char *dest,
   char *template, char *s)
```

This routine provides the functional equivalent of the sprintf() function.

**Parameters**

dest

template

s

## MTimeStringFromLongTime

```
MTimeStringFromLongTime(long
   timeVal, char *s)
```

This routine returns a string in the format of "00:00.00" representing the long time value. Space for the string must be provided by the caller.

**Parameters**

| | |
|---|---|
| timeVal | time value to convert |
| s | string to hold converted time |

## MOM TASK SERVICES

The MOM Task Service provides methods which allow MOM components to perform periodic tasks. Your component should define a method that will be called when then timer task needs to be serviced.

| Task Functionality | Methods |
|---|---|
| Task Creation | MPostCTimeTask |
| Task Destruction | MKillCTimeTask |
| Task Manipulation | MGetCTimeTaskTime, MSetCTimeTaskTime |

## MGetCTimeTaskTime

```
MGetCTimeTaskTime( MSelf *self,
   MTInfo *tinfo, long *period )
```

MGetCTimeTaskTime retrieves the period of the posted periodic timer task.

### Parameters

| | |
|---|---|
| self | your component |
| tinfo | information associated with the timer task when it was registered |
| period | the returned period of the timer task |

### Example

```
static void updateTaskTimer( MSelf *self)
{
  long period;

  MGetCTimeTaskTime(self, &self->f_taskInfo, &period);
  if (MError() != kMNoCompErr)
       return;
  period = period/ 2;
  MSetCTimeTaskTime(self, &self->f_ taskInfo, &period);
  if (MError() != kMNoCompErr)
       return;
  //...
  return;
}
```

### See Also

“MSetCTimeTaskTime” on page 4.255

## MKillCTimeTask

```
MKillCTimeTask( MSelf *self,
   MTInfo *tinfo)
```

MKillCTimeTask stops a registered timer task which is identified by the timer info.

**Parameters**

| | |
|---|---|
| self | your component |
| tinfo | information associated with the timer task when it was registered |

**Example**

```
#define kUpdatePeriod 100

static MErr MCompProcessMessage(MSelf *self, MMessagePtr message )
{
  if(MDetectMessage(message, self->f_enableEvent );
  {
      MPostCTimeTaskj( self, nil, kUpdatePeriod, kMYourOwnSlot,
         kMYourOwnMethodSlot );
  }
  else if( MDetectMessage(message, self->f_disableEvent) ||
            MDetectDisableMessage(message) )
  {
      MKillCTimeTask( self, nil );
  }
}
```

**See Also**

“MPostCTimeTask” on page 4.253

## MPostCTimeTask

```
MPostCTimeTask(MSelf *self,
    MTInfo *tinfo, long period,
    short slot, short funcindex)
```

PostCTimeTask starts a periodic timer task. Your component should define a method that will be called when the task needs servicing (as shown in the example below).

**Parameters**

| | |
|---|---|
| self | your component |
| tinfo | additional information associated with the timer task to be registered. |
| period | the period of the timer task |
| slot | you component's slot |
| funcindex | method's slot that gets executed as the timer task |

**Example**

```
// Previously defined in MCompMainName
...
MDefineMethod(kMyCompSlot, kMBaseCompSlot, kMyTaskMethod, MyTastMethod);
...

// Definition of task method. The info parameter will be the same data that you
// registered the task with using MPostCTimeTask.
MCompStdFuncDcl MyTaskMethod(MSelf *self, MTInfo *info)
{
  // service the task
  return kMNoCompErr;
}

#define kUpdatePeriod 100

static MErr MCompProcessMessage(MSelf *self, MMessagePtr message )
{
  if(MDetectMessage(message, self->f_enableEvent );
  {
      MPostCTimeTaskj( self, nil, kUpdatePeriod, kMyCompSlot, kMyTaskMethod);
  }
  else if( MDetectMessage(message, self->f_disableEvent) ||
          MDetectDisableMessage(message) )
  {
      MKillCTimeTask( self, nil );
  }
}
```

**See Also**

"MKillCTimeTask" on page 4.252

## MSetCTimeTaskTime

```
MSetCTimeTaskTime( MSelf *self,
   MTInfo *tinfo, long period )
```

MSetCTimeTaskTime sets the period of the posted periodic timer task.

**Parameters**

| | |
|---|---|
| self | your component |
| tinfo | information associated with the timer task when it was registered |
| period | the new period of the timer task |

### Example

```
static void updateTaskTimer(MSelf *self)
{
  long period;

  MGetCTimeTaskTime(self, &self->f_info, &period);
  if (MError() != kMNoCompErr)
       return;
  period /= 2;
  MSetCTimeTaskTime(self, &self->f_info, &period);
  if (MError() != kMNoCompErr)
       return;
  //...
  return;
}
```

### See Also

“MGetCTimeTaskTime” on page 4.251

## MOM MTOOL SERVICES

The MOM Tool Service provides methods which allow editor-based mTools to perform their functions. MTools are intended to work at edit time rather than runtime, and therefore have different requirements. MTools need to be able to add entries to the mTropolis menus, be informed of user actions, and otherwise manipulate mTropolis at the project level.

## MAddToolMenuItem

```
MAddToolMenuItem(MSelf *self,
   CompID *toolID, short
   whichMenu, char *name);
```

This routine adds an mTool to the currently installed tools, and add the given string to the specified menu. Currently, mTools may add their menu items to the Tools menu, which is specified by the kToolsMenu constant.

If you want your component to be dynamically constructed when its menu item is chosen, then pass NULL for the “self” parameter (use your component’s self for auto-spawned components). Pass your component’s MOM ID as the “toolID” parameter.

## MReplaceToolMenuItem

```
MReplaceToolMenuItem(CompID
   *toolID, char *newStr);
```

If you need to change the menu string associated with your component, call MReplaceToolMenuItem. Pass your component's MOM ID as the "toolID" parameter, and the new menu string in "newStr." You may need to change your tool's menu string depending on the current state of the project, such as the number of items selected, etc.

## MDoMenuCommand

```
MDoMenuCommand(long cmdID);
```

Call this method when you want to execute one of mTropolis' menu commands directly. The "cmdID" should be set to the ID of the menu command you want to execute.

## MCompToolDo

```
MCompToolDo(MSelf *self);
```

Your component needs to override this method if it adds an mTool to the mTropolis editor menus. This method will be called in your component when the user selects your menu command from the menu. You should perform your work during this call.

## MCompToolUndo

```
MCompToolUndo(MSelf *self);
```

If your component was able to successfully carry out is work during the call to MCompToolDo(), then the user may choose to undo his action by choosing MCompToolUndo. If you override this method, then it will be called when the user wants to undo the action that your tool just performed.

## MCompToolRedo

```
MCompToolRedo(MSelf *self);
```

If your component was able to successfully carry out is work during the call to MCompToolUndo(), then the user may choose to redo his action by choosing MCompToolRedo. If you override this method, then it will be called when the user wants to redo the action that your tool just undid.

## MCompToolCommit

```
MCompToolCommit(MSelf *self);
```

Your component's MCompToolCommit method will be called when a tool's action is about to be committed to the project, that is, it will no longer be undoable (thus, you can get rid of any undo information). This happens when the application has been notified that another task has been performed.

## MOM QUERY SERVICE

The MOM Query Service provides methods that allow MOM components to perform object query operations on a mTropolis project. Components may need to do this in order to track down objects that may have been moved from one location to another, changed parents, been unloaded due to memory constraints, etc. Components use certain MDataTypes to keep track of their objects, and use the methods described in this service when an object needs to be accessed.

## MObjectHasPath

```
MObjectHasPath(MObjectRef
   *objPtr, Boolean *hasPath)
```

This routine returns TRUE if the given object can have path data associated with it. Currently, the only types of objects that can have paths associated with them are elements and modifiers (not assets). If this function returns TRUE, then you can call the MObjectGetPath() function.

## MObjectGetPath

```
MObjectGetPath(MObjectRef
   *objPtr, char ps, char
   *objectPath)
```

This routine retrieves the path of the given object. The character passed in via the "ps" parameter will be used as a path delimiter. The path of the object will be returned in objectPath, including the name of the object.

- *Note: Make sure to call MObjectHasPath() before making this call.*

## MFindObjectByPath

```
MFindObjectByPath(MSelf *self,
    char ps, char *objectPath,
    MObjectRef **objectList)
```

This routine returns a list of objects that match a given a path. Pass in the path delimiter and path text that you received from MObjectGetPath().

- *Note: The list will be created for you. Pass in a pointer to an uninitialized list object.*

## MFindFirstObjectByPath

```
MFindFirstObjectByPath(MSelf
    *self, char ps, char
    *objectPath, MObjectRef
    **objectPtr)
```

This routine returns the first object that matches a given path. Pass in the path delimiter and path text that you received from MObjectGetPath().

## MObjectGetPathLength

```
MObjectGetPathLength(MObjectPtr
   *objPtr, long *length)
```

This routine retrieves the length of the path string associated with the given object. You can use this value to allocate a buffer to hold the path data.

## MFindEventByName

```
MFindEventByName(char
   *eventName, MEvent *eventData)
```

Given an author event name, this function will find the MEvent data associated with it.

- *Note: Initialize the event structure before passing it in. Check the return value for this function using MError() to see if the event was found.*

## MEventGetName

```
MEventGetName(MEvent *eventData,
   char *name)
```

This routine retrieves the name of the author event specified by the MEvent data. You must supply an allocated string for the event name.

## MGetObjectPartialIDPath

```
MGetObjectPartialIDPath(MObjectRe
   f *objPtr, MPartialIDPath
   *idPath)
```

This routine retrieves the partial ID path data associated with an object.

## MFindObjectByPartialIDPath

```
MFindObjectByPartialIDPath(MObjec
   tRef *objPtr, MPartialIDPath
   *idPath)
```

This routine finds the object specified by the partial ID path data.

# Chapter 5. User Interface Guidelines

Users play a key role in product design at mFactory. They are consulted at each stage of the design process and are encouraged to collaborate fully in the creation of design solutions. This approach allows mFactory to create products that meet the needs of authoring professionals and to raise the expectations for software design.

In order to create a system that facilitates smooth work flow and rapid learning, mFactory has implemented a consistent interface across all product components. This includes the use of language, the placement of controls and settings, the grouping of modifiers with similar functionality into icon families, and the behavior of the interface.

In addition to consistency, a clean and simple approach to aesthetics distinguishes mFactory products. Reducing each design to its essence has allowed for elegant interface solutions.

## USING THESE GUIDELINES

The mFactory guidelines provide straightforward instructions for implementing MOM components that adhere to mFactory standards. The guidelines describe specific design methods that mFactory uses when creating interface solutions.

The guidelines can be used at different stages of the design process. When brainstorming with users and members of the design team, the document can be used as a common point of reference. During implementation, the guidelines can assist your team in making interface decisions. Towards completion, the checklist can be used to evaluate your design and to avoid common errors.

While the guidelines will help with the implementation of MOM components, they do not contain a fully inclusive description of the interface design process. A list of suggested reading (page 5.283) has been provided to assist you in researching design methods.

## INTERFACE ELEMENTS

During the creation of a project, a user may need to interact with dozens of modifiers. As a result, it is critical for the dialogs to follow a consistent approach to layout. A typical mTropolis modifier dialog is shown in Figure 5.1. The same dialog, as seen in ResEdit is shown in Figure 5.2.

- The icon should be placed in the upper left corner alongside of the name field.
- If there is a message that causes the modifier to execute or terminate its settings, this field or fields should be

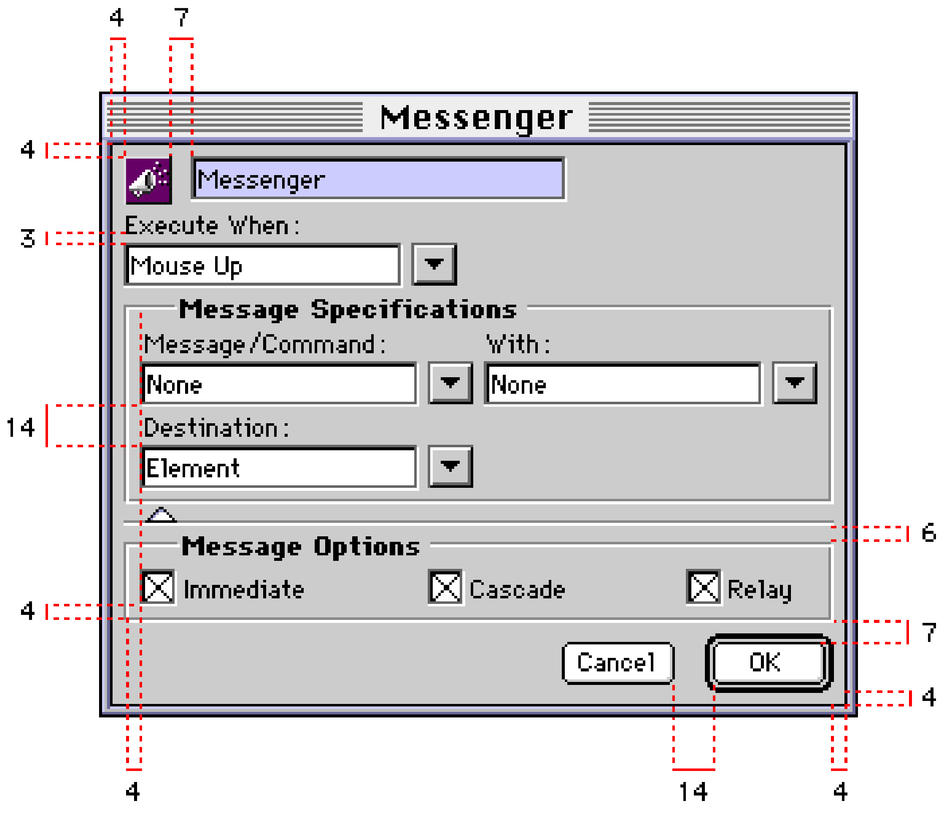


*Figure 5.1 Messenger as seen in mTropolis. Dimensions given in pixels.*

positioned below and aligned with the icon and name field.

- An enclosure should be used to group the settings that will be executed when the message is received.
- A drop down may be used to incrementally introduce complexity to the user by hiding less frequently used settings. These settings should also be grouped in an enclosure.
- Pop-up menu and edit field labels should be positioned above the control.
- When displayed horizontally, controls should be distributed evenly in the space allowed. When displayed vertically, they should be aligned flush left.
- Labels should be placed to the right of checkboxes and radio buttons and aligned flush left when stacked vertically.

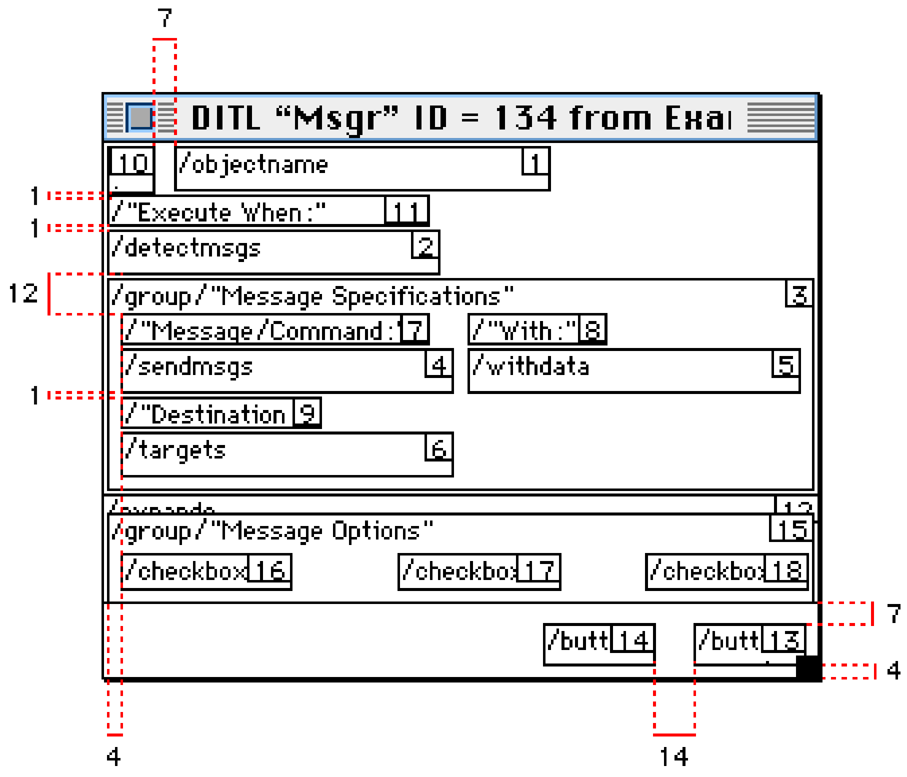


*Figure 5.2 Messenger as seen in ResEdit. Dimensions given in pixels.*

- The OK and Cancel buttons should be positioned to the right and aligned with the right most object.

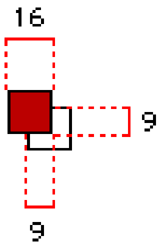
## Services

When a dialog is required for a service, the same approach to layout that is used for Modifiers should be followed.

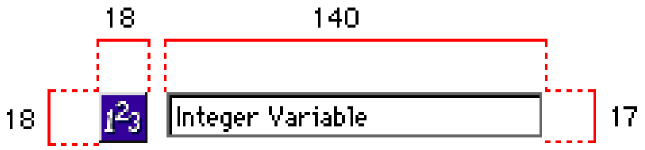
## Controls

The modifier kit containing MOM examples that accompanies mTropolis includes a modifier, Interface Example which demonstrates each of the dialog controls which can be created in MOM. The controls below are displayed as they are seen in mTropolis. Their dimensions are given in pixels and are the same dimensions as seen in ResEdit.

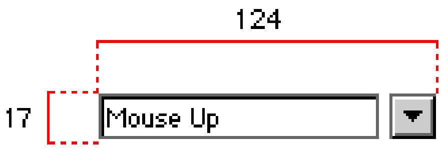
### Color Spot



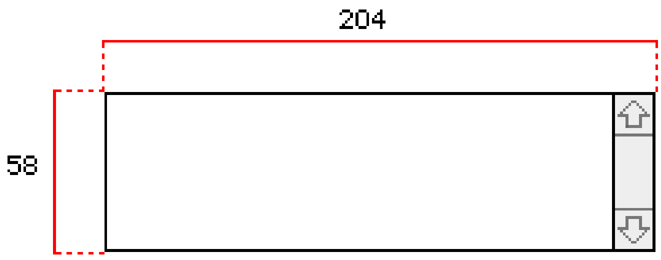
### Icon and Name Field



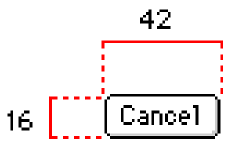
### Pop-Up



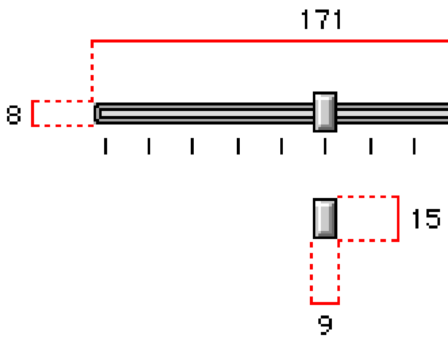
### Path Text Field



### Cancel/OK Button



**Value Slider**

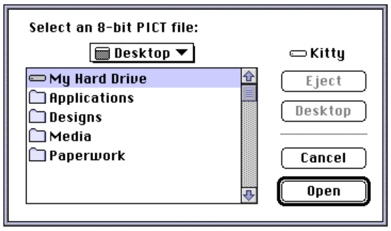


## Language

Language plays an important role in the mTropolis interface. Whether it is in the form of labels for controls or warning messages, it is best to use terms that will be easily understood by users.

- Assign names to modifiers that convey their intended usage and yet are not overly descriptive or lengthy. For example, Change Scene Modifier, Simple Motion Modifier and Scene Transition Modifier all communicate their purposes clearly and simply.
- For modifiers that are Messengers, Variables or Services, do not include the word modifier in the name.
- If there is a message that causes the modifier to execute or terminate its settings, this field or fields should be labeled with either "Execute When" and "Terminate When" or "Apply When" and "Remove When". The choice depends on the nature of the mod and what sounds appropriate.
- The labels "Enable When" and "Disable When" should only be used if the messages simply cause the modifier to begin listening for the message that will execute it. An example of this would be the Collision Messenger.
- For tools, use a name that is descriptive and if appropriate, conveys an action. The sample tool (in an experimental release) that accompanies mTropolis is called "Extract Color Table..." This name communicates clearly to the user the purpose of this feature.
- Descriptive phrases can be used in the standard file dialog to assist users when selecting a file (as shown in the figure below). This can be helpful in reminding

the user of what they are searching for, should they be interrupted.



## ICONS

Design considerations for modifier icons are described below.

### Modifiers

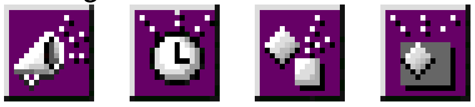
The modifier icons in mTropolis have been designed as a coordinated set. Not only does this facilitate learning and recognition but it allows for future growth. Icons can be more easily generated given a formal icon system.

- Create icons in grayscale unless color is absolutely essential to convey a concept.
- Adhere to existing modifier group colors where possible. If a new group must be created, select a background color that is subdued and that provides sufficient contrast for the icon.
- Reduce the design to its fewest, essential components. Remove any extraneous information that is not contributing to the recognition of the icon.
- Use a consistent "light source" from the upper left where possible.
- Avoid designing an icon in isolation from the rest of the family. Instead, place the icon that you are creating alongside of existing modifier icons in mTropolis to check for consistency.

### Groups

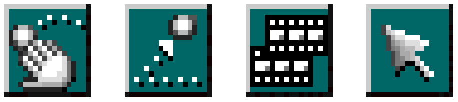
The RGB values of the color for each group has been given for the benefit of individuals working in paint applications. When working with a resource editor, you may open one of the modifier resource kits that accompany mTropolis to view and select a color of a specific group.

### Messengers



Messengers are characterized by their ability to send messages. In this icon group, dotted lines radiate from the objects indicating communication. This group's color is: R: 102, G: 0, B: 102.

### Effects



This group of modifiers generally produces a particular effect. That can include a gradient, a cursor or a color change. This group's color is: R: 0, G: 102, B: 102.

### Variables

Variables are used to store values of different data types such as text or integers. Variables are characterized by their numeric symbols and letters. This group's color is: R: 51, G: 0, B: 153.

### Tasks

This group is comprised of several mods used for navigation and several which handle file I/O. This group's color is: R: 51, G: 51, B: 102.

### Services

This group is characterized by its ability to enable certain areas of functionality. This group is new and yet already distinct. This group's color is: R: 102, G: 102, B: 153.

### Miniscript and Behavior

Miniscript and Behavior do not fit into the existing modifier groups. They each define their own group. The color for Miniscript is: R: 102, G: 153, B: 153. The color for Behavior is: R: 153, G: 102, B: 255.

### Qualifiers—Indicating Mac Only and Experimental Status

To indicate that a modifier will only have an effect in a Macintosh title, a yellow, 4 x 4 pixel dot is placed in the upper left corner. Yellow is generally used to mean caution. To indicate that a modifier is an experimental release, a red, 4 x 4 pixel dot is placed in the upper right corner. Red is generally used to indicate a warning or potential danger.

## MINISCRIPT INTERFACE

Design considerations for the Miniscript interface to components are described below.

### Attributes

Besides the visual interface of your component's dialogs, you can also provide access to your component through Miniscript, the mTropolis scripting language. Objects in mTropolis can have *attributes* that contain values which describe their current state. Some attributes are readable and can be used in Miniscript expressions. Others are writable and can be used as targets for Miniscript set statements. Attributes are accessed with the standard Miniscript dot notation, as in "set element.width to 20."

The *mTropolis Reference Guide* contains an entire chapter on element attributes, but modifiers and other components can also have attributes. When deciding which portions of your MOM component to expose as attributes, consider those that the user is likely to want to change dynamically from within a script. For example, if your component operates on a time-based medium, the playback speed might be a good candidate for an attribute. On the other hand, you would not want to provide an attribute for setting a modifier's "Enable When" message. Setting this message dynamically in a script can make debugging a project's logic much harder. Also note that an alternative to attributes is allowing a variable to be entered wherever a constant might be used. That way the user can change the variable at runtime to affect a change in the modifier's behavior.

## Functions

Currently it is not possible to add MOM functions to Miniscript. This limitation can be overcome to some extent by substituting attributes with side-effects for functions. For example, the list variable has an attribute called "sortAscend". Simply referencing this attribute immediately causes the list to be sorted in ascending order, as in "set elementCount to list.sortAscend." In this case, the attribute is read-only and returns the number of elements in the list as its result.

Another option is to create an attribute that operates as soon as it is "touched," as in "set worldManager.ejectCD to true," which ejects the CD, if one is present in the drive. Usually the value to which these write-only attributes are set is irrelevant, so it is best to simply require a boolean.

There are two possible ways to pass parameters to a MOM "function." One is to use the square bracket notation, as in:

```
set bitDepthOkay to \
  system.supportsBitDepth[32]
```

This method is limited to a single integer parameter, but it does enable you to return a result.

The other way is to use one or more attributes to hold the parameters and another which returns the result. For example, there is a built-in System attribute called *volumeIsMounted* that evaluates to true if a volume with the same name as the *volumeName* attribute is currently mounted. To use this "function," the user would first set *volumeName* to an appropriate string, such as:

```
set system.volumeName to "My CD"
```

and then would get the result by referencing the *volumeIsMounted* attribute, as in:

```
if system.volumeIsMounted then ...
```

Although passing "parameters" in this way can be awkward, it does work.

### Naming Conventions

- Attribute names must be 16 characters or less.
- Miniscript is case-insensitive in that all names are converted to lower-case before being evaluated. This means that in your MOM code, the attribute must be specified in all lower-case.
- If an attribute is used as a function, give it a name that implies an action, such as "ejectCD."
- If an attribute corresponds directly to an option in the modifier's dialog box, use the same name for the attribute.

## DESIGN CHECKLIST

### General Considerations

- Have you consulted users about what they are attempting to do and how they would like to use this new feature?
- Have you built on the existing mTropolis interface where possible?
- Have you avoided assigning new behaviors to existing interface objects?

### Icons

- Is color used only to distinguish modifier groups and not for the icon itself (except when essential to convey a concept)?
- Are your designs simple and clear?
- Have you used familiar, real-world objects where possible?
- Are your icons distinct from others?

### Dialogs

- Have you reduced extraneous space in dialogs according to guidelines?
- Are there warnings about risky actions?
- For a modifier in its default state, is the name that appears in the title bar the same as the name that appears in the name field?

### Language

- Is the language used in dialogs clear and concise?

### Miniscript

- Are your attributes specified in lower-case?

## SUGGESTED READING

**Apple Computer, Inc. (1992)**
*Macintosh Human Interface Guidelines,* Reading, MA: Addison-Wesley.

**Borenstein, Nathaniel S. (1991)**
*Programming As If People Mattered*, Princeton, NJ: Princeton University Press

**Horton, William (1994)**
*The Icon Book*, New York, NY: John Wiley and Sons, Inc.

**Laurel, Brenda (ed.) (1990)**
*The Art of Human-Computer Interface Design*, Reading, MA: Addison-Wesley

**Mullet, Kevin and Sano, Darrell (1995)**
*Designing Visual Interfaces*, Mountain View, CA: SunSoft Press

**Norman, Donald A. (1988)**
*The Psychology of Everyday Things*, New York: Basic Books (paperback edition titled, *The Design of Everyday Things*).

**Rubin, Jeffrey (1994)**
*Usability Testing*, New York, NY: John Wiley & Sons, Inc.

**Woolsey, Kristina Hooper (1996)**
*VizAbility*, Boston, MA: PWS Publishing Company

# Chapter 6. MOM FAQ

This chapter contains answers to frequently asked questions about MOM components and technology

**Q: How do I use resources in my MOM component?**
A: You may need to do this in order to retrieve strings from your resource data, or access a special dialog, etc. On the Windows side, you do not need to make any special modifications to your component to use resources that reside in the component. Simply call MGetKitRef, which will return an HINSTANCE that you can use to access your resources. On the Macintosh side, resource files are organized into a "search chain" by the system, so you need to save aside the current resource file number, get your component's resource file number from MGetKitRef, and make it the current resource file by calling UseResFile on the data it passes back. You must reset the old resource file when you are finished. An example of this is:

```
MFFileRef resfile,oldresfile;
MGetKitRef(myKitsMomID, &resfile); // Windows: HINSTANCE, Mac: short

#ifdef _Macintosh_
oldresfile = CurResFile();
UseResFile(resfile);
#endif

// Do stuff with the resources

#ifdef _Macintosh_
  UseResFile(oldresfile);
#endif
```

**Q: My component needs to do some one-time initialization. Where should I do it?**
A: MOM provides two static class methods: `MClassStaticInit`, and `MClassStaticDestruct`. These are called once for each class (at startup and shutdown, respectively), so you should perform any one-time stuff (like MIDI initialization or checking for AppleScript or other installed software and such) in these methods.

**Q: How come my modifier loses track of some types of events, like author messages, when the user drags it to another project? How do I account for this?**
A: Your component needs to override the MCompEnvironmentChanged method, and call MAdaptData with its Event member

variables in order to adapt them to the new project. Author messages are project-specific, and thus need to be added to the new project when a modifier that uses them is added to the project.

**Q: I have a messenger that keeps track of its destination target, and both the messenger and the target are inside a behavior. Trouble is, when I duplicate the behavior, the messenger loses its target.**
A: Your component needs to override the MCompEnvironmentChanged method, and call MAdaptData with the target member variable. Any component that keeps a reference to an Object ID needs to override this method, because when objects are duplicated they are assigned new IDs, and your component will need to update the ID of the object it is keeping a reference to.

**Q: Can I add my own function calls to Miniscript?**
A: Well, yes and no. Since your component's GetAttribute method will be called when the user executes a miniscript that asks for an attribute, you can use this as a trigger to perform a "function" when invoked. For example, a component that puts up a dialog can trigger on a Miniscript like:

```
set result to dialogmod.askuser
```

When the GetAttribute method is called with "askuser" as the attribute, you can then put up your dialog. What you can't currently do is modify Miniscript's vocabulary through MOM, nor can you have a function call that accepts parameters. mFactory will address this restriction in a future release.

# Chapter 7. Appendices

This chapter contains additional MOM reference material including:

## MOM EVENT CONSTANTS

The following event constants are defined in the “MConst.h” file and can be passed to functions such as MInitEvent() and MDetectMessage().

They are equivalent in spelling to the strings displayed in the “/detectmsgs” and “/sendmsgs” pop-ups, with the addition of the “kM” prefix.

The events are organized below by their positions in the “/detectmsgs” and “/sendmsgs” pop-ups. The one item that appears in both these pop-ups but is never detected nor sent is “None”. Its constant is “kMNone”.

| Submenu | Detect Events | Send Events |
|---|---|---|
| Mouse | | |
| | kMMouseDown | same |
| | kMMouseUp | same |
| | kMMouseUpInside | same |
| | kMMouseUpOutside | same |
| | kMMouseOver | same |
| | kMMouseOutside | same |
| | kMMouseTracking | same |
| | kMTrackedMouseOutside | same |
| | kMTrackedMouseBackInside | same |
| Element | | |
| | kMShown | na |
| | kMHidden | na |
| | kMSelected | na |
| | kMDeselected | na |
| | kMEditDone | na |
| | na | kMShow |
| | na | kMHide |
| | na | kMSelect |
| | na | kMDeselect |
| | na | kMToggleSelect |
| | na | kMEditElement |
| | na | kMUpdateCalculatedFields |

*Table 7.1: MOM Event Constants*

| Submenu | Detect Events | Send Events |
|---|---|---|
| | na | kMScrollUp |
| | na | kMScrollDown |
| | na | kMScrollLeft |
| | na | kMScrollRight |
| | na | kMPreloadMedia |
| Play Control | | |
| | kMPlayed | na |
| | kMStopped | na |
| | kMPaused | na |
| | kMUnpaused | na |
| | kMAtFirstCel | same |
| | kMAtLastCel | same |
| | na | kMPlay |
| | na | kMStop |
| | na | kMPause |
| | na | kMUnpause |
| | na | kMTogglePause |
| | na | kMPlayForward |
| | na | kMPlayBackward |
| Motion | | |
| | kMMotionStarted | same |
| | kMMotionEnded | same |
| Transition | | |
| | kMTransitionStarted | same |
| | kMTransitionEnded | same |
| Parent | | |
| | kMParentEnabled | same |
| | kMParentDisabled | same |
| Scene | | |
| | kMSceneStarted | same |
| | kMSceneEnded | same |
| | kMSceneDeactivated | same |

*Table 7.1: MOM Event Constants*

| Submenu | Detect Events | Send Events |
|---|---|---|
| | kMSceneReactivated | same |
| Shared Scene | | |
| | kMReturnedToScene | same |
| | kMSceneChanged | same |
| | kMNoNextScene | same |
| | kMNoPreviousScene | same |
| Project | | |
| | kMUserTimeout | same |
| | kMProjectStarted | same |
| | KMProjectEnded | same |
| | na | kMCloseTitle |

*Table 7.1: MOM Event Constants*

## MDATATYPE STRUCTURES

The following table describes the MOM data structures and provides a list of there associated macros.

| Data Type | Variable Name | Type | Description | Initialization, Copy, and Disposal Macros |
|---|---|---|---|---|
| MBoolean | f_type<br>f_value | short<br>short | Type identification<br>The boolean value | MInitBoolean, MCopyBoolean, MDisposeBoolean |
| MControl | f_type<br>f_minVal<br>f_maxVal<br>f_stepSmall<br>f_stepSmall<br>f_value | short<br>double<br>double<br>double<br>double<br>double | Type identification | MInitControl, MCopyControl, MDisposeControl |
| MDouble | f_type<br>f_value | short<br>double | Type identification<br>The double value | MInitDouble, MCopyDouble, MDisposeDouble |
| MEvent | f_type<br>f_event<br>f_eventinfo | short<br>long<br>long | Type identification<br>The event code<br>the additional event info | MInitEvent, MCopyEvent, MDisposeEvent |
| MIDPath | f_type<br>f_nItems<br>f_pathIDs | short<br>short<br>long* | Type identification<br>number of ID items in f_pathIDs<br>ID list | MInitIDPath, MCopyIDPath, MDisposeIDPath |
| MInteger | f_type<br>f_value | short<br>long | Type identification<br>The integer value. | MInitInteger, MCopyInteger, MDisposeInteger |
| MObjectPtr | f_type<br>f_previous<br>f_next<br>f_obj<br>f_partialIDPath | short<br>MObjectPtr<br>MObjectPtr<br>MObjectRef<br>MPartialIDPath | Type identification<br>init to nil, used internally<br>init to nil, used internally<br>the object reference<br>init to nil, used internally | MInitObjectPtr, MCopyObjectPtr, MDisposeObjectPtr |
| MPoint2D | f_type<br>f_value | short<br>PLPoint | Type identification<br>The point value. | MInitPoint2D, MCopyPoint2D, MDisposePoint2D |

*Table 7.2: MDataType Structures*

| Data Type | Variable Name | Type | Description | Initialization, Copy, and Disposal Macros |
|---|---|---|---|---|
| MPointer | f_type<br>f_owner<br>f_bufSize<br>f_dataLen<br>f_ptr | short<br>void *<br>long<br>long<br>void * | Type identification<br>who allocated memory<br>Length of allocated buffer<br>length of data in buffer<br>Pointer to allocated buffer | MInitPtr, MCopyPtr, MDisposePtr |
| MRange | f_type<br>f_value1<br>f_value2 | short<br>long<br>long | Type identification<br>the range-from value<br>the range-to value | MInitRange, MCopyRange, MDisposeRange |
| MRect2D | f_type<br>f_value | short<br>PLRect | Type identification<br>The rect value | MInitRect2D, MCopyRect2D, MDisposeRect2D |
| MRGBColor | f_type<br>f_value | short<br>PLRGBColor | Type identification<br>The rgb color value | MInitRGBColor, MCopyRGBColor, MDisposeRGBColor |
| MString | f_type<br>f_owner<br>f_bufSize<br>f_strLen<br>f_ptr | short<br>void *<br>long<br>long<br>char* | Type identification<br>who allocated memory<br>Length of allocated buffer<br>length of string in buffer<br>the allocated buffer | MInitString, MInitSizedString, MCopyString, MDisposeString |
| MSymbol | f_type<br>f_value<br>f_info | short<br>long<br>long | Type identification<br>the symbol value<br>additional info | MInitSymbol, MCopySymbol, MDisposeSymbol |
| MTime | f_type<br>f_value<br>f_scale<br>f_base | short<br>long<br>long<br>long | Type identification | MInitTime, MCopyTime, MDisposeTime |
| MTimeRange | f_type<br>f_value1<br>f_scale1<br>f_base1<br>f_value2<br>f_scale2<br>f_base2 | short<br>long<br>long<br>long<br>long<br>long<br>long | Type identification | MInitTimeRange, MCopyTimeRange, MDisposeTimeRange |

*Table 7.2: MDataType Structures*

| Data Type | Variable Name | Type | Description | Initialization, Copy, and Disposal Macros |
|---|---|---|---|---|
| MVector2D | f_type<br>f_angle<br>f_magnitude<br>f_timeUnits | short<br>double<br>double<br>double | Type identification<br>in radians<br>in units | MInitVector, MCopyVector, MDisposeVector |

*Table 7.2: MDataType Structures*

### MDataType Type Constants

The MDataType data structures use f_type to identify themselves. The following tables list and briefly describe the current data identification and action type constants available for use in MOM.

| Data Identification Constant | Description |
|---|---|
| kMBoolean | Boolean |
| kMControl | Control object |
| kMDouble | Real number |
| kMEventID | Event specification |
| kMIDPath | Object ID Path |
| kMInteger | Four byte integer |
| kMNoValueType | Type has no value |
| kMObjectPtr | MOM Object pointer |
| kMPoint2D | Two dimensional space coordinate |
| kMPointer | Memory pointer |
| kMRect2D | Two dimensional space rectangle |
| kMRGBColor | RGB color specification |
| kMStringPtr | Memory pointer to sized null terminated string data |
| kMSymbol | Symbol specification |
| kMUnivrslTime | Time specification |
| kMUnivrslTimeRange | Tiume range specification |
| kMVector2D | Two dimension vector |

*Table 7.3: MDataType Type Constants*

The one action type currently available is kMPreferredCopy. The appropriate response for this action type is to provide a copy of the data, including any required allocation of memory on behalf of the requestor. It is assumed that the passed data structure has no allocated buffer associated with it.

| Action Constant for f_type | Type |
|---|---|
| kMPreferredCopy | specialized, request for copy |

*Table 7.4: Constants for Action Types*

## MOM DIALOG RESOURCE HANDLING

Dialogs are an important MOM component mechanism for edit-time attribute setting. MOM provides its own set of controls for use in these dialogs. This section describes how MOM handles resources compiled into the component and also describes what controls are available for use in MOM components.

### Dialog Resources

On the Mac platform, a MOM dialog resource consists of a “DITL” (Dialog ITem List) and a “DLOG” (DiaLOG box). The DITL is comprised of a collection of standard static text controls (those provided by ResEdit) filled with MOM keywords (see table below) describing the actual runtime control to be used and any other appropriate information for that control (i.e., ranges for sliders). The tab ordering doubles as the identification value for the control. The DLOG resource is used in sizing the dialog. These resources should have the same identifying name and numeric ID. You don’t normally create a “DITL” resource; the resource editor will do this for you as you edit the corresponding “DLOG” resource.

Refer to “Editor Construction Guide” on page 2.42 for more information about using the static text tags in your component’s editing dialog.

### MOM Dialog Resource Keywords and Symbols

The following table lists the available keyword strings that can be used in the standard static text controls of the resource editor.

| Keyword or Symbol | Function | Example | Associated MDataType |
|---|---|---|---|
| # | Specifies an ID value | /scenes/#25 | N/A |
| $ | Initial value | /arrowcnt/0...1000/%10/$100 | N/A |
| % | Step value | /arrowcnt/0...1000/%10/$100 | N/A |
| * | Decimal places | /double/*8 | N/A |
| @ | Resource ID | /iconbutton/@2525/@2526 | N/A |
| accept | Results in call of MCompEditorAccept | /button/"OK"/accept | N/A |
| arrowcntr | Provides a numeric spinner, with range, stepsize and initial value | /arrowcnt/0...1000/%10/$100 | MInteger |
| base | Draws a line the length of the control | /base | N/A |

*Table 7.5: Resource Keyword Strings and Symbols*

| Keyword or Symbol | Function | Example | Associated MDataType |
|---|---|---|---|
| button | standard button | /button/"Button Title" | N/A |
| checkbox | standard checkbox | /checkbox/"CheckBox 1" | MBoolean |
| clutfiles | CLUT popup list | /clutfiles | |
| colorspot | Color selection list | /colorspot | MRGBColor |
| decline | Results in call of MCompEditorDecline | /button/"Cancel"/decline | N/A |
| default | Sets the default control | /button/"OK"/accept/default | N/A |
| detectmsgs | Detection messages popup | /detectmsgs | MEvent |
| double | Control accepting only numerics | /double | MDouble |
| edittext | Standard edit control | /edittext | MString |
| expando | Demarks an area of the dialog box that is initially hidden and will open up when the triangular tab is clicked. | /expando | N/A |
| filepath | | /filepath | MString |
| fonts | Provides a font list popup | /fonts | MString |
| fontsize | Provides a font size list popup | /fontsize | MInteger |
| growbox | Indicates if the dialog should have a maximize button in the title bar | /growbox | N/A |
| iconbutton | Button using two user provided icons for visual effects | /iconbutton/@2525/@2526 | N/A |
| infogroup | | | N/A |
| integer | Control accepting only integer numerics | /integer | MInteger |
| keys | Keyboard popup list | /keys | MEvent |
| noshow | | | N/A |
| objecticon | Provide the position and sizeof the components icon | /objecticon | N/A |
| objectname | Provides an edit control for the components name | /objectname | N/A |
| panel | | | N/A |

*Table 7.5: Resource Keyword Strings and Symbols*

| Keyword or Symbol | Function | Example | Associated MDataType |
|---|---|---|---|
| panelpopup | | | MInteger |
| popup | Popup using string resource list for contents | | MInteger |
| radio | Standard radio button | | MInteger |
| radiogroup | Grouping for Radio Buttons (size of static text control should encompass all appropriate radio buttons) | /radiogroup | N/A |
| scenes | Scenes popup list | /scenes | MString |
| scrollinglist | | | MInteger |
| sections | Sections popup list | /sections | MString |
| sendmsgs | Send messages popup list | /sendmsgs | MEvent |
| slider | | /slider | MInteger |
| soundfiles | Sound files popup list | /soundfiles | MString |
| standard | | | N/A |
| subsections | Subsections popup list | /subsections | MString |
| targets | Targets popup list | /targets | MString |
| text | Provides a scrolling edit field | /text/"initial text" | MString |
| valslider | Demarked, button informing slider | /valslider | MInteger |
| withdata | 'With Data' popup list | /withdata | MObjectPtr |

*Table 7.5: Resource Keyword Strings and Symbols*

# Chapter 8. Alphabetical List of MOM Routines