Mops

Mike's Object-oriented Programming System

Version 2.6

Part III

Predefined Classes

Mops is an object-oriented programming system, derived from the Neon language developed by Charles Duff and sold by Kriya, Inc. Kriya have discontinued support for Neon, and have released all the source code into the public domain, retaining only the ownership of the name Neon.

Mops implemented by:	Michael Hore
Able assistance from:	Doug Hoffman Greg Haverkamp Xan Gregg
Documentation updated:	Version 2.6, June 1995
Documentation formatted by:	Craig Treleaven

Printing this document

This document is in Microsoft Word Version 5.1 format and uses the fonts Times, Courier, and Helvetica, only. It is formatted using the Laserwriter 7 driver for US Letter paper, portrait orientation, with fractional widths enabled. If you want to print any other way, you will probably need to repaginate and regenerate the table of contents and table of predefined classes and methods. See below.

Almost every paragraph in this document is formatted using a Word style. Formatting is consistent throughout and can be reformatted in moments this way.

Viewing on-line

Of course, you can read the whole manual on-screen. Word's Find… command can help to locate items of interest. One other technique is useful but not well known. Use the Outline View and click the "3" in the ruler at the top of the screen. Word will then show the headings down to a list of all the predefined classes. Clicking the "4" will show the groupings of each classes' methods (such as "accessing"). In either case, whichever line is at the top of the window in outline view will become the line at the top of the window when you switch back to Normal View. By scrolling in Outline View, you can quickly find the section of interest and position the window for reading in Normal View.

Two-sided printing

As shipped, this document is formattted for 2-sided printing to save paper. If you haven't printed two-sided documents with your printer before, you might want to practise with the first few pages before sending the whole thing. On most printers, you need to use Word's option to print first the odd numbered pages (in the Print… dialog), reload the paper and then print the even numbered pages.

Single-sided printing

If you don't want to bother with two-sided printing, use the Document dialog and make the Gutter margin zero. If you adjust the Left and Right margins so the printable width is still 6.5 inches, the page breaks should stay in the same places. Blank pages may pop out here and there as all chapters start on an odd-numbered page.

A4 Paper

If you select A4 paper in the Page Setup… dialog, the page breaks will change. Regenerate the Methods table and the table of contents, as below. As far as I can tell, the paragraph styles all do the right thing and ajust to the paper width. Well, all except one: the header on odd-numbered pages will extend a quarter inch into the margin because the tab stop is at 6.5 inches. Redefine the Header style to set it to 6.25, if you feel the need.

Table of Methods and Table of Contents

The tables at the beginning of this document employ a 'trick' in Word. Headings for the Table of Contents have Heading 1 and Heading 2 styles. The sections for the classes have Heading 3 style and groups of methods have Heading 4 style. Each method has Heading 5 style. We use the Table of Contents… dialog to collect headings from level 1 to level 2 for the TOC. Use the same command to collect headings from level 3 to 5 for the table of methods. Word finds the correct table and replaces it with the newly generated table.

About this chapter

This chapter describes the Mops classes and words providing you with the fundamental structures that are necessary for programming in Mops. Most of these correspond to well-established data structures that are familiar to computer scientists, and some of them are unique to Mops on the Macintosh.

Recommended reading

IM - Memory Manager
IM - Programming in Assembly Language

Source files

Class
Struct

Using the basic data structures

This chapter will discuss the primitive classes that Mops provides as building blocks out of which you can assemble the data structures necessary to build your application. These classes are useful both as instance variables of more complex classes and as general classes from which you can derive more specialized subclasses. The classes that will be covered here include:

Object	DicAddr	Ordered-Col
Bytes	X-Addr	WordCol
Int	Indexed-Obj	ByteCol
Uint	bArray	X-Col
Longword	wArray	Sequence
Var	Array	HandleArray
Handle	X-Array	HandleList
ObjHandle	Obj_array	Dic-Mark
Ptr	(Col)	Resource

Using Class Object and Bytes

The root of all classes is class Object. It has no data, but does have a set of behaviors that are generally applicable to any object, regardless of its format. A class that has no particular inheritance path should make Object its superclass, which will cause it to inherit the general properties that all objects should have. Addr: returns the base address of an object. This is less useful with Mops 2.5, however, since just naming an object without sending it a message will cause its address to be pushed, and this works for both public objects and ivars (it used to only work for public objects). Other methods in Object provide a hex dump of an object's data and access to an object's class pointer.

Bytes is not really a class, but rather is a Mops word that enables you to allocate a certain number of bytes as an instance variable within a class definition. Bytes is chiefly useful when mapping parts of Toolbox data structures that only need to be allocated but not accessed. Bytes actually creates an ivar of class Object, so you can use Object's methods, such as Addr:, on an ivar created with Bytes. As an example, class Window uses Bytes to allocate portions of the window record that Mops doesn't need direct access to. Remember, however, that Bytes is not an indexed type like barray—its an Object. If you send it a length: message you'll always get zero, which is the length of Object—this surely won't be what you want!

Using the scalar classes

Scalar classes represent non-indexed objects which hold simple integer or pointer data. A Byte, Int or Var can hold an 8,16, or 32-bit signed integer respectively. A UByte or Uint can hold an 8 or 16-bit unsigned integer. Bool is a 1-byte boolean class. A Ptr is a simple pointer or address. A DicAddr holds the address of a word in the dictionary. An X-addr holds the address of an executable word. A Handle has methods for allocating, sizing, locking and releasing relocatable blocks of heap. ObjHandle is a subclass intended for dealing with dynamic objects on the heap. For all of the scalar classes, Get: and Put: fetch and store the object's private data with an operation of the appropriate width.

Using the array classes

There are three basic array classes in Mops - bArray, wArray and Array, having 1, 2 and 4-byte indexed cells. We have defined a basic set of array methods that are shared by these classes, and must be redefined if you create array classes with different indexed widths. Most array messages require that an index be on the stack that reflects which cell of the array the operation refers to (indexes begin with 0).

We have defined a generic superclass for all arrays, called Indexed-Obj. This class defines some general methods which are independent of the indexed width. These are ^Elem:, which returns the address of an indexed cell of any width, using a run-time lookup, Limit:, which tells you the maximum number of elements allocated to an object; Width:, which tells you the width of an object's indexed cells; ClearX:, which sets all of an array's cells to 0; and IXAddr:, which leaves the address of the 0th indexed cell.

There is also a group of methods that must be redefined for each array class having a different width. These include: At:, which fetches the contents of the cell at an index; To: which stores to the indexed cell at an index, +To:, which increments an indexed cell by a value; -To:, which decrements an indexed cell, and Fill:, which fills an array with a value. This group is shared by the three array classes that are predefined in Mops, and is documented later in this section. We also override ^Elem: in these three classes to give greater speed, since we know the indexed width at compile time.

Because class Array has 4-byte cells, it can be used to hold pointers to various kinds of structures in a way that the other array classes cannot.

We have defined several classes which make it easy to handle groups of objects. By multiply inheriting Obj_array with any other class, you create an array of objects of that class. You then use the select: method to make one of those objects "current", and can then access the "current" object exactly as if it were a normal object, i.e. not part an array at all.

HandleArray and HandleList are both for sets of heap-based objects, accessed through ObjHandles. HandleArray uses ObjArray as one of its superclasses, so it uses the select: method to make a particular one of its ObjHandles "current".

Here's an example of how a HandleArray could be used to implement four windows accessible by index:

```
4 HandleArray Windows
: CreateWindows
       4 0 DO  i select: windows  ['] window  newObj: windows  LOOP  ;
\ Resize window at index 2:
       2 select: windows   300 100  size: [ obj: windows ]
```

Notice how once we have used select: to choose which ObjHandle in the HandleArray we are referring to, we can then send other methods to the HandleArray exactly as if it were a single ObjHandle. We can send late-bound messages to one of the windows, as in the size: message at the bottom, by using the obj: method defined for the ObjHandle class. Actually, the objects in Windows could be of any class that accepted a size: message, due to the late binding.

When you are finished with Windows, you can release all its heap storage simply by sending the message

```
release: windows
```

If you look at the source for the HandleArray class, you will see that release: causes each of the ObjHandles to be selected in turn, and releaseObj: sent to each one. If you now look at the source for ObjHandle, you will see that releaseObj: causes release: to be sent to the object pointed to by the handle, so that it will release any heap storage it has allocated, then finally release: super is called, which releases the heap block pointed to by the handle (that is, the object itself). Thus, by simply sending release: to a HandleArray, we are releasing all the heap storage it owns. Incidentally, if you want to just release one of the handles in a HandleArray, use select: followed by releaseObj:—this is the reason we have defined releaseObj: separately from release:.

Class X-Array adds to the basic Array the ability to execute one of its indexed cells, assuming that it holds the xt of a Mops word. X-Array is a very important class in Mops, because its behavior is used throughout the system itself to provide control dispatching by index, as in Menu and Event. The classinit: method in X-Array sets each indexed cell to Null so the object will behave gracefully if you fail to initialize it in your application. Use X-Array whenever you need to execute one of a group of Mops words based on a series of contiguous indices.

Using Collections

Class Ordered-Col is another important class in Mops. It is implemented by multiply inheriting the (Col) class with one of the array classes. It adds to the array class the concept of a current length and the ability to add to and remove from the list. This list also has many of the properties of a stack, which are exploited in such classes as FileList (see Chapter III.3). When you create an Ordered-Col (O-C), you must specify, as with all indexed classes, the number of elements to allocate in the dictionary (or the heap). O-C uses this as a maximum up to which its variable-length list will grow via the Add: method. The advantage of an O-C is that you can add values to the end of the list without maintaining the index yourself, only the sequence in which to add. You might want to utilize the O-C's properties only while initializing the object, after which it is simply used as an Array. WordCol is an Ordered-Col with 16-bit cells rather than 32-bit.

Using the Dictionary Class

Class Dictionary is a piece of code written for use with the assembler. It is used to hold the symbol table, and allows strings (labels in the case of the assembler) to be dynamically added as the program is running, and associated with any object. Dictionary is a subclass of HandleArray, and each element in the array can be the beginning of a chain of dictionary elements (instances of class DictElt). Class dictionary has two accessing methods, they are enter: and query:. When a label is encountered in the assembler, it is entered in the symbol table. If it is a definition, then the code position is stored with it, so:

```
: ENTERLABEL
      token  query: symtab
      nilP <> IF  253 asmError  THEN    \ Error if already defined
      ['] var  newObj: tempH
      codePos 2*  here +  obj: tempH  put: **
      tempH token  enter: symtab  unlock: tempH  ;
```

(Note: token is an instance of class String.)

Object

Object contains behavior appropriate to all objects in the system. Every superclass chain ultimately traces back to Object.

Superclass	Meta
Source file	Class
Status	Core
Instance variables	None
Indexed data	None
System objects	None

Meta

Methods

class: (-- addr)

Returns a pointer to the object's class.

.id (--)

Types the object's name.

.class (--)

Types the name of the object's class.

addr: (-- addr)

Returns the base address of an object's data.

length: (-- #bytes)

Returns the length of the object's ivar data area.

copyto: (^obj --)

^obj is a pointer to another object. This method copies that object's ivar data to this object. Be careful using this method—no check is done that the objects are of the same class. However this method can be very useful in some situations.

classinit: (--)

This a very special method—whenever an object is created, Mops sends it a classinit: message so that it will initialize itself to reasonable values, or whatever the programmer desires all objects of that class to do when created. This method corresponds to a constructor method in C++. In class Object, it is a do-nothing method, allowing any subclass to override it as appropriate. By convention, init: is used for explicit programmatic initialization and customization thereafter, and new: is used to set up the toolbox-interface portion of toolbox objects (such as making a window known to the Macintosh window manager).

release: (--)

This method does nothing in class Object itself. However, in general you should send release: to an object before you FORGET it or deallocate its memory. release: will cause an object to release any heap memory it has allocated and do any other cleaning up which may be necessary. This method corresponds to a destructor method in C++.

dump: (-- addr)

Dumps the dictionary entry for the object in a hex format.

print: (-- addr)

Dumps the dictionary entry for the object in a hex format. This provides a default print: method for objects that don't have a more sophisticated form of displaying their data.

Error messages

None

Longword

Longword provides storage for 32-bit quantities. It is not intended to be used directly, but is a generic superclass for Var , Handle, Ptr and DicAddr.

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
4 bytes	data Allocates 32 bits of data.
Indexed data	None
System objects	None

Object

Methods

accessing

get: (-- val)

Returns the value in the data area as a signed number.

put: (val --)

Stores a new value in the data area.

->: (^longword --)

Copies the passed-in Longword's data to this Longword.

clear: (--)

Stores 0 in the data area.

initialization

classinit: (--)

Calls clear:.

display

print: (--)

Prints the data in the current number base on the screen.

Error messages

None

Var

Var provides storage for 32-bit numeric quantities.

Superclass	Longword
Source file	Struct
Status	Core
Instance variables	None (see Longword)
Indexed data	None
System objects	None

Longword, Object

Methods

accessing

+: (val --)

Adds val to the contents of the Var's data area.

-: (val --)

Subtracts val from the contents of the Var's data area.

Error messages

None

Int, Uint

Provides storage for 16-bit quantities—signed (Int) and unsigned (Uint).

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
2 bytes	data Room for 16 bits of data.
Indexed data	None
System objects	None
Object	

Methods

accessing

get: (-- val)

Returns the value in the data area. For class Int, this is a signed number—if bit 15 is on, this bit will be extended into the high-order 16 bits of the stack cell. For class Uint, the number is unsigned—the high-order 16 bits of the stack cell will be set to zero.

put: (val --)

Stores a new value in the data area.

->: (^int --)

Copies the passed-in Int/Uint's value to this Int/Uint.

clear: (--)

Stores 0 in the data area.

int: (-- int)

Returns the value in the data area as a 16-bit stack cell. This is useful for Toolbox calls that require parameters of type Int.

+: (val --)

Adds val to the contents of the Int/Uint's data area.

-: (val --)

Subtracts val from the contents of the Int/Uint's data area.

display

print: (--)

Types the data in the current base on the screen.

Error messages

None

Byte, Ubyte

Provides storage for 8-bit quantities—signed (Byte) and unsigned (Ubyte).

Superclass	Object

Source file	Struct	
Status	Core	
Instance variables		
1 bytes	data	Room for 8 bits of data.
Indexed data	None	
System objects	None	
Object		

Methods

accessing

get: (-- val)

Returns the value in the data area. For class Byte, this is a signed number —if bit 7 is on, this bit will be extended into the high-order 24 bits of the stack cell. For class Ubyte, the number is unsigned—the high-order 24 bits of the stack cell will be set to zero.

put: (val --)

Stores a new value in the data area.

->: (^int --)

Copies the passed-in Byte/Ubyte's value to this Byte/Ubyte.

clear: (--)

Stores 0 in the data area.

display

print: (--)

Types the data in the current base on the screen.

Error messages

None

Bool

Provides storage for boolean values (true or false). These are stored in 8 bits.

Superclass	Byte
Source file	Struct
Status	Core
Instance variables	None (see Byte)
Indexed data	None
System objects	None

Byte, Object

Methods

accessing

put: (val --)

Stores a new value in the data byte. If val is non-zero, a "proper" true is stored (all ones). If val is zero, false (zero) is stored. Thus val isn't just copied, but is converted to a "proper" boolean value.

get: (-- true | -- false)

This method is inherited from Byte, but it is worth mentioning here that

since it sign-extends, and we have either all zeros or all ones in the data byte, get: returns a “proper” boolean flag value on the stack (all zeros or all ones).

set: (--)

Sets the data byte to true (all ones).

clear: (--)

Sets the data byte to false (zero).

display

print: (--)

Types the data as “true” or “false”.

Error messages

None

Handle

Handle adds to Longword methods useful for manipulating relocatable blocks of heap.

Superclass	Var
Source file	Struct
Status	Core
Instance variables	None (see Longword)
Indexed data	None
System objects	None

Var, Longword, Object

Methods

accessing

ptr: (-- ptr)

Returns a dereferenced pointer from the handle.

nptr: (-- ptr)

Returns a dereferenced pointer from the handle, and masks out any flag bits. If your Mac is running in 24-bit mode, handles may have flag information in the high-order byte. nptr: zeros these bits, so that you may compare or do arithmetic on the resulting pointer without problems. But if you just want to access the data via the pointer, use ptr: which is slightly faster.

manipulation

clear: (--)

Stores nilH in the handle.

setSize: (len --)

Sets a new size for the heap block corresponding to the handle.

size: (-- len)

Returns the current size of the handle.

new: (#bytes --)

Allocates a block of relocatable heap via the Memory Manager, and stores the handle in this object's data.

release: (--)

Releases the heap block pointed to by the handle and stores nilH in the handle. Does nothing if the handle already contains nilH.

lock: (--)

Locks the block corresponding to the handle.

unlock: (--)

Unlocks the block corresponding to the handle. Does nothing if the handle contains nilH.

locked?: (-- b)

Returns a boolean; true if the block is locked.

getState: (-- n)

Returns the state of the handle, via the system call _HGetState. Prior to locking a handle, it is best to get the state of the handle, perform the operation that needed the handle locked, then reset the state of the handle with setState:

setState: (n --)

Sets the state of the handle.

moveHi: (--)

Calls the system to move the heap block high in memory. It is generally a good idea to do this prior to locking the handle, to minimize heap fragmentation, unless you are going to unlock the handle again very quickly.

->: (^handle --)

Copies the heap data pointed to by the passed-in handle to this handle's heap block, and sets the size of this handle's block appropriately.

Error messages

"Set handle size failed"

Non-0 return from memory manager on a SetHSize system call, probably resulting from a setSize: or ->: call with insufficient memory available.

ObjHandle

ObjHandle adds to Handle methods for manipulating heap-based objects.

Superclass	Handle
Source file	Struct
Status	Core
Instance variables	None
Indexed data	None
System objects	Many and various

Handle, Var, Longword, Object

Methods

accessing

obj: (-- ^obj)

Locks the handle and returns a pointer to the addressed object. You should normally unlock: when you're through operating on the object.

manipulation

newObj: ([#elts] ^class --)

Creates a new object of the given class on the heap and sets the handle.

releaseObj: (--)

Releases the handle, first sending release: to the object it points to. Does nothing if the handle is nil.

display

print: (--)

Types both the handle and the object it points to, in the latter case sending a late-bound print: to the object.

dump: (--)

Likewise dumps both the handle and the object.

Error messages

None

Ptr

Ptr adds to Longword methods useful for manipulating non-relocatable blocks of heap. Note: it is normally better to use Handles rather than Ptrs, to avoid the heap becoming fragmented with blocks which cannot be moved.

Superclass	Longword
Source file	Struct
Status	Core
Instance variables	None (see Longword)
Indexed data	None
System objects	None

Longword, Object

Methods

manipulation

new: (len --)

Allocates a block of non-relocatable heap, and stores the pointer in the object's data.

release: (--)

Deallocates the memory pointed to by the Ptr, and stores nilP in the Ptr. Does nothing if the Ptr already contains nilP.

clear: (--)

Stores nilP in the Ptr.

nil?: (-- b)

Returns True if the Ptr contains nilP.

Error messages

"new: on a pointer couldn't get enough heap"

DicAddr

Dicaddr is used for storing the address of a location within the dictionary. If the dictionary is saved and reloaded in a subsequent run, the address will still be valid. This is accomplished by storing the address in a relocatable format. Don't depend on details of this format, in case it changes.

Superclass	Longword
Source file	Struct
Status	Core
Instance variables	None (see Longword)
Indexed data	None

System objects	None
Longword, Object	

Methods

access

get: (-- addr)

Overrides get: in Longword. Fetches the object's data (a relocatable address), converts it to absolute and returns it.

put: (addr --)

Stores the passed-in address in the object's data, using our relocatable format.

print: (--)

Types the word name corresponding to the stored address, or (no name) if the address isn't the address of a Mops word.

Error messages

"you can't store a module address outside the module"

You attempted to put: the address of a location in a module, into a DicAddr located outside the module. This is illegal, since the module may have moved or been purged from memory when the DicAddr is next accessed.

X-Addr

An X-Addr is almost the same as a DicAddr. The only difference is that it is intended for dictionary addresses which are the execution tokens of Mops words, and so may be executed. (Note that in Mops an execution token is an address of a word, whereas in other Forth systems it may not be an actual address.) Thus we again use our relocatable format. The only difference to a DicAddr is that there is an exec: method, and no get: method.

Superclass	Longword
Source file	Struct
Status	Core
Instance variables	None (see Longword)
Indexed data	None
System objects	None

Longword, Object

Methods

access

exec: (-- various)

Executes the word whose xt has been stored in the X-Addr.

put: (xt --)

Stores the xt in the object's data, using our relocatable format.

Error messages

"you can't store a module address outside the module"

See DicAddr.

Indexed-Obj

This class is the generic superclass for all arrays. It defines the general indexed methods, which apply regardless of indexed width.

Superclass	Object
Source file	Struct,Struct1
Status	Core
Instance variables	None

Indexed data	None (supplied by subclasses)
System objects	None
Object	

Methods

accessing

^elem: (index -- addr)

Returns the address for the element at index.

limit: (-- maxIndex+1)

Returns the allocated size of an indexed object. The maximum usable index for an indexed object is this value minus 1.

width: (-- #bytes)

Returns the width of each indexed element.

ixAddr: (-- addr)

Returns the relative address for the 0th element.

manipulation

clearX: (--)

Sets each indexed element to 0.

Error messages

"Index or value out of range"

One of the methods taking an index found the index to be out of range for this array.

Basic array classes—bArray, wArray, Array

These basic access methods are implemented for the three array classes predefined in Mops.

Superclass	Indexed-Obj
Source file	Struct,Struct1
Status	Core
Instance variables	None
Indexed data	1,2,4-byte cells
System objects	None
Indexed-Obj, Object	

Methods

accessing

at: (index -- val)

Returns the data at a given indexed cell.

to: (val index --)

Stores data at a given indexed cell.

+to: (increment index --)

Increments data at a given indexed cell.

-to: (decrement index --)

Decrements data at a given indexed cell.

fill: (val --)

Stores val in each cell of the array.

Error messages

"Index or value out of range"

As for Indexed-Obj.

X-Array

X-Array is an Array with the ability to execute its indexed data as xts of Mops words.

Superclass	Array
Source file	Struct
Status	Core
Instance variables	None
Indexed data	4-byte cells
System objects	???

Array, Indexed-Obj, Object

Methods

accessing

exec: (ind --)

Executes the xt in the indexed cell at ind.

put: (xt0 ... xt(N-1) N --)

Stores the N xts into the indexed elements of this object. xt0 goes into element 0, xt1 into element 1, and so on.

actions: (xt0 ... xt(N-1) N --)

A synonym for put:. A more appropriate name to use in subclasses such as dialogs.

display

print: (--)

Types the name of the word whose xt is in each element.

initialization

classinit: (--)

Sets all indexed elements to the null xt.

Error messages

"Wrong number of xts in list"

For put:, the value N did not match the number of indexed elements for this object.

Obj_array

This class is a generic superclass which makes it easy to generate an array of objects of a given class. Just define a new class which multiply inherits from the given class (or classes) and Obj_array (which must come last). This will add an indexed section to each object of the new class, with elements wide enough to contain objects of the original class(es). Then select: "switches in" the selected element to be the "current" element, and all the normal methods of the class(es) can then be used.

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
Int	current The number of the element currently “switched in”.
Indexed data	Any width—the actual width is determined by the other class(es).
System objects	None
Object	

Methods

select: (index --)

Makes the indexed element current.

current: (-- index)

Returns the index of the current element.

Error messages

"Index or value out of range"

An out-of-range index value was used for select:.

(Col), Ordered-Col, wordCol, byteCol

Collections are ordered lists with a current size, that can also behave like a stack. We implement them by multiply inheriting the generic (Col) class with an array class of the appropriate width. (Col) adds the concept of a current size to the array methods.

Note: class Ordered-Col, wordCol and ByteCol are 32, 16 and 8 bit collections respectively. All methods are identical to (Col).

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
int	Size # elements currently held in the list.
Indexed data	None (supplied by the array class)
System objects	None
Object	

Methods

accessing

size: (-- #elements)

Returns the number of elements currently held in the list. This must always be less than or equal to limit:.

add: (val --)

Appends value in the next available cell, and increments size by 1. An error occurs if size=limit before the operation (list full).

last: (-- val)

Fetches the contents of the cell last added to the list. Error if list is empty.

remove: (ind --)

Deletes the element at ind from the list, and reduces size by 1. Error if the list is empty.

clear: (--)

Sets list to empty.

indexOf: (val -- ind t OR -- f)

Searches for val within the current list, and returns the index and a True boolean if it was found, and False boolean if not found.

Error messages

"My list is empty"

A remove: or last: was attempted on an empty list.

"My list is full"

An add: was attempted with size=limit.

X-Col

This class is a collection of execution tokens. It adds one new method, and overrides one method of X-Array.

Superclass	(Col) X-Array
Source file	Struct
Status	Core
Instance variables	None
Indexed data	None (supplied by the X-Array)
System objects	None

(Col), X-Array, Array, Indexed-Obj, Object

Methods

removeXt: (xt --)

Removes the xt equal to the passed-in xt Does nothing if no match is found.

print: (--)

As for print: in class X-Array, but only types the xt names that are actually in the collection.

Error messages

As for (Col).

Sequence

Sequence is a generic superclass for classes which have multiple items which frequently need to be looked at in sequence. At present the main function of Sequence is to implement the each: method, which makes it very simple to deal with each element. The usage is

```
BEGIN  each: <obj>  WHILE  <do something to the element>   REPEAT
```

Sequence can be multiply inherited with any class which implements the first?: and next: methods. The actual implementation details are quite irrelevant, as long as these methods are supported.

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
Var	nxt_xt Saves the xt for the next: method of the other class.
Var	^self Saves the address of Self as required for BIND_WITH.
Indexed data	None
System objects	None

Object

Methods

each: ((varies) -- true OR -- false)

Initiates processing of a sequence as in the example above.

uneach: (--)

Terminates processing of a sequence before the normal end. Use prior to an EXIT out of an each: loop.

Error messages

None

HandleArray

HandleArray provides for an array of handles to heap-based objects. This array may be addressed as a stack, similarly to Ordered-Col.

Superclass	ObjHandle Array Obj_array
Source file	Struct
Status	Core
Instance variables	
Int	size number of elements currently held in the array.
Indexed data	4-byte cells (handles)
System objects	None

ObjHandle, Array, Obj-array, Handle, Var, Longword, Indexed-Obj, Object

Methods

accessing

size: (-- size)

Returns the current size.

setSize: (size --)

Sets the current size.

manipulation

release: (--)

Releases the current handle.

push: (handle --)

Stores the handle in the next free position in the array—the “top of the stack” of handles.

top: (--)

Makes the “top” handle current.

drop: (--)

Removes the “top” handle, releasing it, and selects the next handle.

pushNewObj: (^class --)

Creates a new object of the given class on the heap, and pushes the resulting handle on to the “top of the stack” of handles.

clearX: (--)

Sets all the handles to nilH.

Error messages

"My list is empty”

"My list is full"

See (Col).

HandleList

HandleList allows the implementation of a list of heap-based objects. Unlike HandleArray, the list can be of indefinite length. We use a heap block to store the handles to the objects contiguously, rather than have a separate block for each handle and link them together. This saves on memory overhead and reduces the number of Memory Manager calls. It also reflects the assumption that insertions and deletions into the middle of the list will be infrequent, as these could be more inefficient than with a linked scheme. We expect that elements will normally be added to the end, and probably not removed at all, or not very often.

Superclass	ObjHandle Sequence
Source file	Struct
Status	Core
Instance variables	
handle	TheList Points to the memory block containing the handles.
var	Size The current size of the block, in bytes.
var	Pos The (byte) offset in that block of the current handle.
Indexed data	None
System objects	None

ObjHandle, Sequence, Handle, Var, Longword, Object

Methods

accessing

select: (index --)

Makes the indexed handle current.

selectLast: (--)

Makes the last indexed handle current.

current: (-- index)

Returns the index of the current handle.

size: (-- size)

Returns the number of handles in the list. (This is in fact the ivar Size divided by 4).

setSize: (size --)

Sets the current size.

The next two methods are needed by each:, but may be called directly as well. Note that next: ASSUMES that the list is allocated in the heap and that a valid element is selected as the current element. each: ensures this, since if first?: returns false, next?: is never called. But if you call it directly, make sure this condition holds.

first?: (-- b)

If the list isn't empty, makes the first handle current and returns True. If the list is empty, returns False.

next?: (--)

If the current handle isn't the last one, makes the next handle current and

returns True. If the current handle is the last one, returns False.

manipulation

newObj: (^class --)

Creates a new object of the passed-in class on the heap, and adds its handle to the list.

releaseObj: (--)

Releases the current handle, first sending release: to the object it points to. Does nothing if the handle is nil.

removeObj: (--)

Releases the current handle as in releaseObj:, and removes it altogether from the list.

release: (--)

Releases the whole list. Every handle is released as in releaseObj:, and the whole block containing the handles is released as well.

display

dumpAll: (--)

Gives a dump of the whole list, including sending dump: to each of the objects.

printAll: (--)

Displays the whole list, including sending print: to each of the objects.

Error messages

None

PtrList

PtrList allows the implementation of a list of pointers which point to objects. The objects can be anywhere. Similarly to HandleList, we use a heap block to store the pointers.

Superclass	Ptr Sequence
Source file	Struct
Status	Core
Instance variables	
handle	TheList Points to the memory block containing the pointers.
var	Size The current size of the block, in bytes.
var	Pos The (byte) offset in that block of the current pointer.
Indexed data	None
System objects	None
Ptr, Sequence, Longword, Object	

Methods

accessing

select: (index --)

Makes the indexed pointer current.

selectLast: (--)

Makes the last pointer current.

current: (-- index)

Returns the index of the current pointer.

size: (-- size)

Returns the number of pointer in the list. (This is in fact the ivar Size divided by 4).

The next two methods are needed by each:, but may be called directly as well. Note that next: ASSUMES that

the list is allocated in the heap and that a valid element is selected as the current element. each: ensures this, since if first?: returns false, next?: is never called. But if you call it directly, make sure this condition holds.

first?: (-- b)

If the list isn't empty, makes the first pointer current and returns True. If the list is empty, returns False.

next?: (--)

If the current pointer isn't the last one, makes the next handle current and returns True. If the current pointer is the last one, returns False.

manipulation

add: (ptr --)

Adds the pointer to the end of the list.

remove: (--)

Removes the current pointer from the list.

display

dumpAll: (--)

Gives a dump of the whole list, including sending dump: to the objects pointed to by all the pointers.

printAll: (--)

Displays the whole list, including sending print: to the objects pointed to by all the pointers.

Error messages

None

Dic-Mark

Dic-Mark marks a dictionary position, and includes methods for traversing the dictionary.

Superclass	Object
Source file	Struct
Status	Core
Instance variables	
array	Links Stores the set of dictionary addresses which point to the various entries on the various threads corresponding to the current position.
int	Current The index in Links of the current position itself.
Indexed data	None
System objects	TheMark
Object	

Methods

accessing

current: (-- addr)

Returns the current position.

manipulation

set: (addr --)

Sets the current position to addr (setting the array Links appropriately).

setToTop: (--)

Sets the current position to the top of the dictionary.

next: (-- addr)

Moves the current position to the preceding dictionary word, and returns the address of the link field of that word. Returns zero if we were already at the base of the dictionary.

Error messages

None

Resource

Resource implements Macintosh Resources.

Superclass	Handle
Source file	Struct

Status	Core
Instance variables	
Var	Type — 4-byte code for the resource type of this resource.
Int	ID — The resource's ID.
Indexed data	None
System objects	Some
Handle, Var, Longword, Object	

Methods

accessing

set: (type ID --)

Stores the passed-in type and ID in this object's data.

getNew: (-- index)

Accesses the resource with the current type and ID via a GetResource call, and stores the handle in this object's data.

getXstr: (index -- addr length)

This resource object must have type STR# (string list). Accesses the indexed string and returns its address and length.

Error messages

"We couldn't find this resource"

A call to getnew: resulted in the Mac system not being able to locate a resource with the current type and ID. Possibly the type or ID are wrong, or the correct resource file isn't open.

Chapter 2—Strings

About this chapter

This chapter describes Mops's string-handling classes. Strings are objects that contain variable-length sequences of text, with methods for deletion, insertion etc. Mops' powerful string-handling facility provides an excellent base on which you can build various text-based utilities.
Recommended reading
IM - Toolbox Utilities
IM - OS Utilities
Mops - II.4, "Using Strings in Mops"

Source files

String
StrUtilities
String+

Using strings

Mops strings are implemented as relocatable blocks of heap that can expand and contract as their contents change. A string object itself contains a handle to the heap block that contains the string's data. It also contains three other ivars which we will describe below.

Strings can be useful for a wide variety of programming needs. They can serve as file buffers, staging areas for text to be printed on the screen, dictionaries, or vehicles for parsing user input. You should consider using strings for any run of bytes whose length and/or contents are likely to change in the course of your program's execution. Strings are not restricted to ASCII text, although that will probably be their most common use. Note, however, that text constants can more efficiently be implemented as SCONs or string literals (see II.4 for more information).

Using strings is somewhat like using files, in that you must open the string before you use it and close it when you're through. This is done by sending a New: message to each string before you use it, to allocate the string's heap storage, and then sending a Release: message when you no longer need the string. Release: is actually inherited from String's superclass, Handle, and calls the Toolbox routine DisposeHandle.

There are two classes of strings in Mops. String supports basic string operations, such as Get:, Put: , Insert: and Add:. Class String+, a subclass of String, adds more methods, such as searching. Both classes are in the precompiled Mops.dic, and are really only split into two classes since String+ has some code methods, which require the Assembler for compilation, whereas we do require some string operations at an earlier point in the building of the full system, before the Assembler is available. But for all practical purposes you can treat the two classes as a single class.

Many of the String methods are built around the Toolbox Utilities routine Munger, which is a general-purpose string-processing primitive. You might read the IM Toolbox Utilities section on Munger to gain a deeper understanding of what characteristics it contributes to Mops string handling.

Strings have a current size, which is the same as the length of the relocatable block of heap containing the string's data. Strings also have two offets into the string data, called POS and LIM. POS marks the "current" position, and LIM the "current" end. Most string operations operate on the substring delimited by POS and LIM, which we call the active part of the string, rather than the whole string. We also keep the size of the string (the real size, that is) in an ivar, so that we can get it quickly without a system call.

Communicating with other objects

While most of the method descriptions below should be self-explanatory, several are worth additional comment. One group of String+'s methods takes the address of another String or String+ object as one of its parameters, and accesses the active part of this second string.

String+ also has several methods that simplify its use as a file buffer. ReadN:, ReadRest:, ReadAll: and ReadLine?: all accept a File object as one of the parameters, and will request that the File perform a read into the string, setting the size of the string to the number of bytes actually read. Doing things this way is very convenient, especially as the file data is left in a String+ object, and is therefore subject to all of the various manipulations that String+ can perform.

Finally, String+'s Draw: method accepts a rectangle object and a justification parameter, and draws the contents of the string as justified text within the box specified by the rectangle.

String

String defines a variable-length string object with basic access methods whose data exists as a relocatable block of heap. Size is limited only by available memory.

Superclass	Handle
Source file	String
Status	Core
Instance variables	
Var	pos — Offset into the string of the beginning of the active part.
Var	lim — One plus the offset of the last char in the active part. Note that if pos = lim, the active part is empty. Some methods signal an error if pos > lim, or if either is negative or greater than the size of the string.
Var	size — The size of the heap block containing the string data.
Int	flags — Various flags are stored here.
Indexed data	None
System objects	???
Handle, Var, Longword, Object	

Methods

accessing

handle: (-- handle)

Returns the handle to the string—replaces get: in the superclass Handle, since we will be redefining get: here with a different meaning.

pos: (-- n)

Returns the value of Pos.

>pos: (n --)

Stores n in Pos.

lim: (-- n)

Returns the value of Lim.

>lim: (n --)

Stores n in Lim.

len: (-- n)

Returns the value of Lim - Pos, i.e. the length of the active part.

>len: (n --)

Adds n and Pos, and stores the result in Lim.

skip: (n --)

Adds n to Pos.

more: (n --)

Adds n to Lim.

start: (--)

Clears Pos, so that the active part now starts at the “real” start of the string.

begin: (--)

Clears both Pos and Lim. Useful for setting up for an iterative operation on the string.

end: (--)

Sets both Pos and Lim to the size (i.e. the end) of the string. Useful for setting up for an iterative operation which has to go backwards through the string.

nolim: (--)

Sets Lim to the end of the string.

reset: (--)

Clears Pos, and sets Lim to the end of the string. The active part will now be the whole string.

step: (--)

Steps forward in the string, setting Pos to Lim and then setting Lim to the end of the string.

<step: (--)

Steps backward in the string, setting Lim to Pos and then clearing Pos.

manipulation

new: (--)

Creates a heap block for the string’s data, and sets the handle. The initial size is zero. new: must be done before the string can be used.

?new: (--)

Ensures a heap block is allocated, by calling new: if necessary (indicated by the handle being nilH). If a block is already allocated, does nothing.

size: (-- n)

Returns the size of the (whole) string.

setSize: (n --)

Sets the size of the (whole) string to n, then does a reset:.

clear: (--)

Ensures a heap block is allocated, calling new: if necessary, then sets its size to zero.

get: (-- addr len)

Returns the address and length of the active part of the string.

1st: (-- c)

Returns the character at Pos.

^1st: (-- addr)

Returns the address of the character at Pos.

uc: (-- addr len)

Converts the active part to upper case and does a get:.

put: (addr len --)

Ensures a heap block is allocated, calling new: if necessary, then replaces it with passed-in string, and does reset: as well.

->: (str --)

Replaces the whole of this string (as in put:) with the active part of str, which may be a String or String+ (we use early binding, and assume the class).

insert: (addr len --)

Ensures a heap block is allocated, calling new: if necessary, then inserts the string given by (addr len) at Pos. Increments both Pos and Lim by len (thus the bytes at the Pos and Lim position will be the same as before, and the byte immediately preceding the Pos position will be the last of the inserted bytes).

$insert: (str --)

Inserts the active part of str, as for insert:.

add: (addr len)

Inserts (addr len) at the end of this string. Pos and Lim are then set to the (updated) end position.

$add (str --)

Inserts the active part of str at the end of this string.

+: (c --)

Appends the character c to the end of the string, and sets Pos and Lim to the (updated) end position.

fill: (c --)

Overwrites each character in the active part of the string with the character c.

search (addr len -- b)

Searches the active part of this string, starting from the left (i.e. the Pos position), for the string (addr len). If a match is found, Lim is set to indicate the first of the matching characters and true is returned. If no match is found, Lim is unchanged and false is returned.

Note 1: an improved version with case control is provided in String+.

Note 2: We use Lim rather than Pos, since it often happens after a search that some operation needs to be done on the part of the string preceding the matching substring. If this isn't needed, step: is convenient for updating Pos to the matching substring position and preparing for another search.

chsearch: (c -- b)

Searches the active part of this string for the character c. If it is found, Lim is set there and true is returned. If it isn't found, Lim is unchanged and false is returned.

object interaction

copyto: (^string-obj --)

Overrides copyto: in class Object. The only change is that we set a flag in this object, marking it as a copy. This will mean that any future operation which would change the size of this object will be blocked with an error message. You will be able to alter Pos and Lim freely, but not insert or delete. It is frequently useful to have several copies of the same string object, in order to manipulate several active parts at once. But I have found that it's important to keep one as the "original" object, and only insert/delete on this one. Failure to do this led to crashes.

mark_original: (--)

Overrides the above check, by clearing the flag, so that this string becomes "original". Only use this method if you're quite sure what you're doing. The idea of the long name is that you won't type it accidentally!

display

print: (--)

Displays the active part of the string, assuming it to be ASCII characters.

dump: (--)

Gives a dump of the string, displaying various useful quantities such as Pos and Lim, and displaying the contents of the string as ASCII characters and in hex.

rd: (--)

"Reset and dump". Does reset:. then dump:. Short to type when debugging!

Error messages

"String pointer(s) out of bounds"

Pos was found to be greater than Lim, or either was negative or greater than the size of the string. Pos and Lim are also displayed when this message is given. We check for this error condition whenever we access the actual characters of the string. Operations such as >pos: don't perform the check—this is for speed, and also because when we are doing manipulations on Pos and Lim we don't want to put any restriction on intermediate values.

"Can't do that on a string copy"

You attempted to insert, delete, or change the size of a string object which was flagged as a "copy". See above under copyto:.

TrTbl

Translate tables allow very fast searching of strings for specified sets of characters. In effect we are separating the specification of what we are searching for from the actual search operation itself. This allows an uncluttered and extremely fast search operation (the scan: and <scan: methods of class String+), and it also allows a very flexible (and easily extensible) choice of what to search for. The setup time for translate tables can generally be factored out of inner loops, or done at compile time, and is quite fast, anyway. We first define a class (trtbl) which is needed to define the table mapping lower case letters to upper case. This table is then used by some of the methods in the Trtbl class proper. However this is just an implementation convenience—these classes really should be thought of as one class, so we put all the methods together here.

Superclass	(TrTbl), whose superclass is Object
Source file	StrUtilities
Status	Core
Instance variables	
int	count Used internally in counting characters selected, so the table bytes can be set correctly.
256 bytes	TheTbl The table itself.
Indexed data	None
System objects	
UCtbl	A table which maps lower case letters to upper case, and leaves everything else unchanged.

Object

Methods

accessing

tbl: (-- addr)

Returns the address of TheTbl.

selection

clear: (--)

Clears all bytes of the table to zero.

put: (addr len --)

Copies the bytes given by (addr len) into the table. If len is greater than 256, only the first 256 bytes are copied.

selchars: (addr len --)

Selects each of the bytes given by (addr len). The table byte corresponding to each byte in the list will be set non-zero. The actual value used will be n, where this is the nth byte which has been selected since the last clear:. If two or more bytes in the list are the same (which means they select the same table position), the first will be used in determining the value of the table byte. The counting of n will nevertheless still continue for all the bytes in the passed-in list. Note that this rule only applies within one selchars: operation—if a character is selected by selchars: (or selchar: below) which has already been selected in a previous selection operation, and it is the nth character selected since the last clear:, the corresponding table byte will still be set to n even though it was already non-zero.

selchar: (c --)

Selects the single character c. The value of the table byte is determined as in selchars:.

selcharNC: (c --)

"Select char, no case". Selects a character, and if it is a letter, enters the same value in the lower case and upper case positions of the table, so that case will in effect be ignored when the table is used.

selRange: (lo hi --)

Selects all characters with values from lo to hi inclusive. The selected table bytes will all be set to 1—when a range is selected, there isn't usually a need to distinguish the individual characters. Does nothing if hi < lo.

invert: (--)

Reverses the current selection. All non-zero table bytes are cleared, and all zero bytes are set to -1. (There is no special significance in this value; it was just the simplest to do quickly, thanks to the SEQ machine instruction.)

>uc: (--)

Copies the 26 bytes corresponding to A-Z into the a-z positions. Subsequently any translate operation using this table object will give identical results for upper and lower case letters. Note the direction of the copy—you need to first set up the UPPER case letter positions, then use >uc:.

operations

transc: (c -- c')

Translates the single character c using the table, and returns the corresponding byte c' from the table.

All other translate table operations are methods of class String+.

Error messages

None

String+

String+ adds many useful methods to String.

Superclass	String
Source file	String+
Status	Core

Instance variables None (see String)
Indexed data None
String, Handle, Var, Longword, Object

Methods

accessing

swapPos: (n -- n')

Swaps Pos with the top of the stack.

save: (-- handle pos lim)

Saves the current string parameters.

restore: (handle pos lim --)

Restores the string parameters. Must match a save:.

character fetching

2nd: (-- c)

Returns the second char in the active part, or 0 if the active part's length is 1. Gives an error if the active part is empty.

last: (-- c)

Returns the last char in the active part. Gives an error if the active part is empty.

comparisons

compare: (addr len -- n)

Compares the string (addr len) with the active part of this string. Comparison is by CMPSTR, with the (addr len) string as the first operand. Case is significant if CASE? is set to true. Returns: -1 if the first string is low, 0 if strings are equal, 1 if the first string is high. We assume the lengths are both less than 64K.

?: (addr len -- n)

As for compare:, except that if the the (addr len) string is shorter than the active part of this string, only the first len chars in the active part are used. Note that this only makes a difference if an "equal" result is obtained.

=?: (addr len -- b)

Compares as for ?:, but only tests for equal/not equal. Returns true on equal.

ch=?: (c -- b)

Compares the given single character against the character at Pos. Returns true on equal. If the active part of the string is empty, always returns false.

searching

search: (addr len -- b)

Similar to search: in String, but has full case control, according to the setting of the value Case?. This also applies to all the following searching operations.

<search: (addr len -- b)

Backwards search. Searches the active part of this string, starting from the right (i.e. the Lim position), for the string (addr len). If a match is found, Pos is set to indicate the first (leftmost) of the matching characters and true is returned. If no match is found, Pos is unchanged and false is returned.

sch&skip: (addr len -- b)

Searches for the string (addr len) and if found, sets Pos to the character following the found substring. Leaves Lim unchanged.

chsearch: (c -- b)

Searches for the single character c. If found, returns true and leaves Lim pointing there. If not found returns false and leaves Lim unchanged.

<chsearch: (c -- b)

Backward search for the character c. If found, sets Pos.

chsch&skip: (c -- b)

What you'd expect. Searches as for chsearch:, and if the char is found, Pos is set pointing to the next character. Lim is unchanged.

chskip?: (c -- b)

Searches for the first character NOT equal to c. This method has a couple of differences to the other searching methods, dictated by what we normally need it for. If it suceeds, Pos (not Lim) is set to that position, and it is always case sensitive, regardless of CASE?.

chskip: (c --)

As for chskip?:, but returns no boolean result.

scanning

scan: (trtbl -- n)

Searches for a single character, using a translate table. "Success" is defined as a character which yields a non-zero value from the table. The return result is this non-zero value, or zero if none was found. On success, as usual, Lim is set to point to the found character.

<scan: (trtbl -- n)

Backward scan.

scax: (trtbl -- n)

"Scan excluding". As for scan:, but "success" is defined as a character which yields a zero value from the table. The return result is the last byte fetched from the table, which will be zero on success, or otherwise it will be whatever table byte corresponds to the last char in the active part of the string—something non-zero, in any case.

<scax: (trtbl -- n)

Backward scax.

translate: (trtbl --)

Translates the whole active part of the string, using the table. Replaces each byte in the string with the looked-up value from the table.

trans1st: (trtbl -- n)

Translates the first char in the active part of the string, and returns the looked-up value. The char in the string isn't changed. Returns zero if the active part is empty.

>uc: (--)

Converts any letters in the active part to upper case. This is done by

UCtbl translate: self

This is faster than UPPER, and not limited to 64K.

ch>uc: (--)

Converts the first char of the active part to upper case.

insertion, deletion, replacement

chinsert: (c --)

Inserts the char c at Pos. Pos and Lim are incremented by 1.

ovwr: (addr len --)

Overwrites the active part of this string with the string (addr len). Copying stops at the end of the active part, or when len characters have been transferred. Pos is incremented by the number of chars transferred. This operation is faster than normal replacement, as the length of this string cannot change, so we don't need to call Munger.

chovwr: (c --)

Overwrites the char at Pos with c.

$ovwr: (str --)

Overwrites the active part of this string with the active part of str.

repl: (addr len --)

Replaces the active part of this string with the string (addr len). Pos and Lim are both set pointing just past the newly inserted characters.

$repl: (str --)

Replaces the active part of this string with the active part of str.

sch&repl: (addr1 len1 addr2 len2 -- b)

Searches for the string (addr1 len1) in the active part of this string, using search:. If a match is found, the matching substring is replaced by the string (addr2 len2), Pos and Lim are both set pointing just past the newly inserted characters, and true is returned. If no match is found, Pos and Lim are unchanged and false is returned.

replAll: (addr1 len1 addr2 len2 --)

Replaces all occurrences of (addr1 len1) by (addr2 len2) in the WHOLE of this string (i.e. ignoring Pos and Lim). After the operation, a reset: is done.

delete: (--)

Deletes the active part. Lim is then set equal to Pos.

deleteN: (n --)

From Pos, deletes n characters or up to Lim, whichever comes first. Lim is reduced by the number of characters deleted.

line-oriented methods

line>: (--)

sets Lim to the end of the current line (i.e. starting from Pos, the next Return character or the end of the string). Pos is unchanged.

nextline?: (-- b)

Sets Pos and Lim to delimit the next line. This means Pos will point to the char after the Return character, and Lim to the next Return, or the end of the string. If Lim initially does not point to a Return character, the “next” line will actually be the rest of the current one, starting from where Lim pointed. This behaviour means that if Pos and Lim are initially zero, calling nextline?: will actually yield the first line. This can be useful. The returned boolean is true if we actually get another line, and false if we don’t, that is, if Lim was initially at the end of the string. Note that if the string ends with a Return character, and Lim points to this character when nextline?: is called, this is not the same as Lim pointing to the end of the string. Lim will actually be one less than its “end of string” value. Thus nextline?: will return true with an empty line. The next call will return false.

<nextline?: (-- b)

The backwards equivalent.

addline: (addr len --)

Adds the (addr len) string to this string as for add:. Also adds a Return at the end, if (addr len) doesn't already end with a Return.

$addline: (str --)

Adds the active part of str to this string, as for addline:.

I/O methods

readN: (file n --)

Reads n bytes using the passed-in file object. The file must already be open. The bytes read completely replace the WHOLE string (that is, Pos and Lim are ignored). A reset: is done at the end.

readLine?: (file n -- b)

Reads the next line up to a max of n chars into this string (as for readN:). Returns false if end of file. Reads a final Return character (if any) from the file, but doesn't include it in the bytes transferred to the string.

readRest: (file --)

Reads all the rest of the file from its current position into the string.

readAll: (file --)

Reads all the file into the string.

readTop: (--)

Reads all of Topfile into the string, then closes and drops Topfile (see class FileList). Topfile must already be open.

$write: (file --)

Writes the active part to the file.

send: (file --)

Writes the whole string object to the file. See under class File for a full description of the standard methods send: and bring:, which can be implemented by any classes which need them.

bring: (file --)

Reads back the string object from the file, assuming that it was written by send:.

display

draw: (theRect justification --)

Draws the active part in rect theRect, using the Toolbox TextBox routine. justification =

0	Left justification
1	Center justification
-1	Right justification

printAll: (--)

Displays the whole string via TYPE. Handles any embedded Return characters by starting a new line for each one.

Error messages

None

Chapter 3—Files

About this chapter

This chapter describes the Mops classes and words that provide an interface to the Macintosh file system. Class File combines a Toolbox parameter block with methods for reading, writing, interpreting and getting information about files; including Standard File I/O. FileList provides a mechanism for dynamic allocation of File objects instead of having to create them statically in the dictionary.

Recommended reading

IM - File Manager
IM - OS
IM - Device Manager
IM - Package Manager
IM - Standard File Package
IM - Structure of a Macintosh Application

Source files

Files
PathMod.txt

Using files

All file access in Mops is done through an object of class File. For instance, when you request that a source file be loaded, Mops creates a new File object, gives it a filename, opens it, and interprets from the file rather than from the keyboard. File has as part of its data a parameter block, also called a File Control Block or FCB, which holds the data about the file that is needed by the Device Manager and the File Manager. Appended to this is a 64-byte area that holds the name of the file that is associated with the File object. To create an access path to a file, you must first create an object of class File, give it a name, and open it:

```
File  myFile
" someFilename" name:  myFile
open: myFile  abort" open failed"
```

The Name: message first clears the parameter block so that fields won't be left over from a previous open. (This implies that you must set information other than the file name, like setVref:, after sending the Name: message.) When you open the file, a unique IORefNum is assigned to it and placed in the parameter block. You may then use any of the I/O methods to access the file, most of which return a code that reflects the result code from the Macintosh File Manager. If this code is non-0, it means that an error occurred during the I/O. You should check for EOF (-39) on reads, which should not always be treated as an error.

Because File objects are almost 150 bytes in length, it is useful to be able to allocate them dynamically rather than have them locked into a static dictionary. Class FileList, which is a subclass of HandleArray, provides this function by maintaining a "stack" of handles to file objects in the heap. Mops has a single 6-element object of class FileList, called LoadFile, that it uses internally to provide a nested load facility. You can request that LoadFile allocate a new temporary File with the message pushNew: LoadFile. The objPtr Topfile is maintained to always point to the last File object allocated, which is the "top" of the file stack. Thus, you can use phrases like:

```
open: topfile
myBuf 100  read: topfile
```

After you are through using a dynamically allocated File object, you must close it and remove it from the file stack:

```
drop: loadFile
```

Drop: automatically ensures that topFile is closed, but if you need to see the 'close' return code you will want to issue close: topfile before drop: loadfile.

The Clear: method in FileList closes and removes any currently allocated files in the list, and is called by Mops's default Abort routine.

There is a word, LOADTOP, which will open topfile, then invoke the Mops interpreter to interpret from that file rather than the keyboard, then close topfile when it reaches the end. Interpretation will echo loaded text to the screen if the system Value echo? is true, and will end immediately if there is an error. There is also an Accept: method in File that simulates a Mops ACCEPT, but reads from a file.

Standard File Package

The StdGet: and StdPut: methods give easy access to the Macintosh Standard File Package. This code is called by most applications when the user needs to select a file to open, or a "Save As" name. StdGet: and StdPut: set up and execute the various calls to the package manager. StdGet: calls SFGetFile, which displays the familiar scrollable list of files to open within a rectangle, and returns with a boolean on the top of the stack that tells you whether the user actually picked a file or hit the Cancel button. If the boolean is true, your file object will have been set up with the parameters obtained by SFGetFile.

StdPut: is used when you need to get a name from the user for a Save. You need to provide two strings—the first is a prompt, such as "Save file as:", and the second is the default filename that will appear within the text edit item of the dialog. The user is free to edit the text, and the method will return if the user hits Save, Cancel or the Return key. Again, a boolean is returned and if it is true, your file object will have been set up with the parameters obtained by SFPutFile.

With the StdGet: message, you provide a list of up to four file types to be filtered by SFGetFile. Only the file types that you have listed will be included in the list of files to select. For instance,

```
'type TEXT 1  stdGet: topfile
```

causes the Standard File Package to include only files of type "TEXT" in its list, (the 1 indicates the number of types specified). If you want all file types to be shown, do it thus:

```
-1  stdGet: topfile
```

Keep in mind that neither StdGet: nor StdPut: ever actually open the chosen file. They are identical in function to sending Name: & SetVref: to the file object. You must subsequently send a Create:, Open: or OpenReadOnly: before you can access the file.

Hierarchical File System

Mac folders are the equivalent of MS-DOS or Unix directories. This means that to find a file, the system needs not only its name but the names of all the nested folders in which it is located. The names of these folders, from the top level down, is called the path to the file. If you need to, you can in fact specify a file with a full pathname, which takes the form

```
volumeName:folder1:folder2:folder3:filename
```

This is not normally a good idea, at least not in an installed application, since a user might rename or move a folder at any time, which would render the full pathname invalid. Apple recommend that you use Standard File calls whenever possible to locate files.

However in some situations you may know that some files are always in particular places, and in these situations you may use a full pathname. Probably you will always keep your Mops source files in the same place, for example.

To make the management of full pathnames easier in such situations, we provide a mechanism which is

integrated into the Open: method of class File, whereby a set of possible pathnames can be prepended to the filename, one at a time, until the file is found. We use this system in the running of the Mops development system itself, so that Mops source files can be stored in a

number of different folders without requiring you to have to provide full pathnames or answer many Standard File dialogs. We call this set of pathnames a pathlist.

You specify a pathlist in an ordinary text file. The format is, for example,

```
::System source:
::Module source:
::Toolbox Classes:
::Mops folder:
```

Each line specifies the exact string which will in turn be prepended to the unqualified filename in the file object in an attempt to find the file on the disk. Note however, that whatever you specify in the pathlist, the first folder searched will be the "default folder", which is the folder from which the application started up, (the folder in which the Mops nucleus resides until the application is "installed"). If the file isn't found in the default folder, the path specified in the top line of the pathlist file will be used, then the second, and so on, until either the file is found or the list is exhausted. If the file still isn't found, a "file not found" error will be returned.

In this example all the paths start with two colons. This says to step out of the folder in which the application resides then step down into the specified folder. You may also specify one colon which says to step down into the specific folder immediately within the application folder; or you might use three colons which say to step out of two folder levels then step down. You may also begin with no colon which specifies a disk name.

To load a pathlist file, type e.g.:

```
" myPath" getPaths
```

This loads the list from the file named myPath into a string which is maintained by the PathsMod module, which is called by Open:. From then on any Open: will search this pathlist to find the file to be opened; unless the file name is already fully qualified. This technique gives you a degree of transparency since the specific code which issues the Open: never needs to know the particular paths which are being searched.

You may disable the use of any pathlist by setting the value use_paths? false. This is the initial default in installed applications. When you call getPaths, this value is set true, so you don't need to do it yourself.

File

File provides object-oriented access to the Macintosh File Manager. An object of class File should be created for each separate access path required in your application. File objects can be allocated dynamically by using a FileList, described below.

Superclass	Object
Source file	Files
Status	Core
Instance variables	
134 bytes	FCB max parameter block (108 but for hgetvinfo)
record	FSSpec
{	int FSvRefNum
	var FSDirID
	64 bytes FileName
}	
Indexed data	None
System objects	
fFcb	Used by Mops for system file access.
Object	

Methods

getting file information

size: (-- #bytes)

Returns the logical size in bytes of the currently open file.

bytesRead: (-- #bytes)

Returns the actual byte count from the last operation.

result: (-- rc)

Returns the File Manager's result code from the last operation.

getName: (-- addr len)

Returns the name of the file.

getFref: (-- fileRefNum)

Returns the file reference number.

getVref: (-- volRefNum)

Returns the volume reference number.

getDirID: (-- DirID)

Returns the folder directory ID.

getType: (-- fType)

Returns the file type.

getFileInfo: (-- rc)

Fills the parameter block with file info as outlined in the getFileInfo call in Inside Macintosh.

print: (--)

Prints the name of the file on the screen.

setting file characteristics

stdget: (type0 … typeN #types -- bool)

Calls the Standard Get file routine. If a valid file is chosen, places the information into the file object; ready for open: If you don't want to specify a type, set #types to 0 or -1.

stdput: (addr1 len1 addr2 len2 -- bool)

Calls the Standard Put file routine. If a valid file is chosen, places the information into the file object; ready for open:, or create:

name: (addr len --)

Sets the name of the file. Clears the rest of the FCB.

setName: (--)

Sets the name of the file from the input stream.

rename: (addr len -- rc)

Sets the name of file on disk. File does not have to be open.

mode: (mode --)

Sets the positioning mode for the currently open file.

set: (fType sig --)

Sets the file type and signature for the file.

setFref: (fileRefNum --)

Sets the file reference number.

setVRef: (volRefNum --)

Sets the volume reference number.

setDirID: (DirID --)

Sets the folder directory ID.

file operations

create: (-- rc)

Attempts to open the file whose name is in FileName for read/write access. If file is not found, creates it then opens it for read/write.

open: (-- rc)

Opens the file whose name is in FileName for read/write access.

openReadOnly: (-- rc)

Opens the file whose name is in FileName for read access.

new: (-- rc)

Creates the file whose name is in FileName with 0 length.

read: (addr len -- rc)

Reads len bytes into the buffer starting at addr (waited).

write: (addr len -- rc)

Writes len bytes from the buffer starting at addr (waited).

readLine: (addr len -- rc)

Reads len bytes into the buffer starting at addr (waited). The read will terminate if a CR is received ($0D).

moveTo: (pos -- rc)

Sets the current position pointer in the parameter block to pos relative to the beginning of the file.

last: (--)

Positions to the end of the file.

close: (-- rc)

Closes the currently open file.

delete: (-- rc)

Deletes the file whose name is in fileName from the disk. The file must not be open, or an error will result.

volume-level operations

flushVol: (-- rc)

The Mac system maintains a cache in RAM for each volume (usually a disk). When data is written to the volume, it may not be written immediately to the actual disk, but stored in the cache. flushVol: writes out any data which is still in the cache for the volume containing the current file. This avoids any risk of data loss if there is a loss of power or a serious system crash.

parameter block access

fcb: (-- addr)

Returns the address of the parameter block associated with this File object.

clear: (--)

Clears the parameter block for a new Open.

interpretation

accept: (addr len -- #chrs eof?)

Performs a Mops ACCEPT to the address provided, reading a line from the currently open file. eof? is true if the line is the last line of the file. #chrs is the actual number of characters read, excluding any terminating carriage return. Lines are echoed to the screen if ECHO? is true.

Error messages

None—return codes from File Manager

FileList

FileList is a HandleArray with specialized methods that assume the elements contain handles to File objects. It provides dynamic allocation of File objects, keeping the handles in what is effectively a file stack.

Superclass	HandleArray
Source file	Files
Status	Core
Instance variables	None (see HandleArray)
Indexed data	4-byte cells containing handles to File objects
System objects	
loadFile	6-element FileList used for nested loads.

HandleArray, ObjHandle, Array, Obj-array, Handle, Var, Longword, Indexed-Obj, Object

Methods

accessing

pushNew: (--)

Allocates an object of class File on the heap and adds its handle to the end of the list. Puts a pointer to the file object in Topfile (locking the handle). Error if the list is full.

drop: (--)

Closes the top file object if it was still open, then deletes it and removes it from the list. Topfile is set to point to the new top file. Error if the list is empty.

clear: (--)

Closes and removes all files from the list. Useful for cleaning up after an error.

Error messages

"My list is empty"

A remove: was received with an empty list.

Chapter 4—Events

About this chapter

This chapter describes the Mops classes and words that manage Macintosh events for the application. Macintosh applications are event-driven, meaning that the program must at all times be responsive to the various input devices available to the user, including the keyboard, the mouse and the disk. Mops has built-in support classes that make event handling virtually invisible to the application, enabling the programmer to focus on the problems that he or she is attempting to solve. Most of the time, you will not find it necessary to concern yourself with the Event classes, but this chapter will provide some orientation in case you would like to modify event handling to suit your specific needs.

Recommended reading

IM - Event Manager
IM - Window Manager
IM - Menu Manager
IM - Control Manager
Mops - Windows
Mops - Menus
Mops - Controls

Source files

Event

Using events

Class Event is the core of Mops's event management. It instantiates a single object, fEvent, which resides in the nucleus portion of Mops's dictionary. FEvent is functionally an X-Array of 24 elements, each of which contains the xt of a Mops word corresponding to a particular event type. The Macintosh OS maintains a first-in first-out queue of events received from various I/O devices, and the application can request that the next event be accepted from the queue at any time. If no 'real' events are outstanding, the Macintosh returns to the application with a Null event, which is simply a statement that nothing else happened so that the application can continue with its processing.

Non object-oriented Macintosh programs are usually designed with a huge case statement at the highest level that processes the various types of events that can occur. This results in a sort of inverted structure, in which the lowest-level processing is managed at the highest level of the code. Mops avoids this by handling as many conditions as it can behind the scenes (for instance, calling the Menu Manager when the user clicks the mouse in the menu bar) and using late binding to allow the application to provide specific processing where it is needed. For example, each window in an application might take a unique action when the user clicks the mouse in its content region. Mops simply sends a late-bound Content: message to the front window when a content click occurs, which results in the specific content method being executed that is appropriate for the window's class. Late binding allows Mops' event management to be completely general and open-ended, because the programmer can always build more specific event responses into Window and Control subclasses. Mops' basic Window and Control classes provide general behavior that will be acceptable for many situations.

Macintosh events are assigned a contiguous series of type codes:

Type Code	Description
0	* Null event - used to provide background processing
1	Mouse down - button was depressed

2 * Mouse up - button was released

3	Key down - key was depressed
4	* Key up - key was released
5	AutoKey - key is being held down
6	Update - a window must redraw a portion of its contents
7	Disk - a disk was inserted in a drive
8	Activate - a window became active or inactive
10	* Network - an AppleBus event occurred
11	* IODriver - a device driver event occurred
12-14	* user-definable events
15	OS events, such as Suspend and Resume
23	High-level events, including AppleEvents

Events marked with a * are events for which Mops executes its null-event code rather than code specific to the type of event. If your application assigns significance to these event types, you will have to install your own action word in the cell of fEvent corresponding to the event's type. You might also need to change fEvent's mask with the set: method to accommodate event types that are currently masked out.

Class Event contains a set of named ivars that allocate a Toolbox event record. Event's sole object, fEvent, passes its base address to the Event Manager as the event record to use for all Mops events. FEvent also contains 24 indexed cells, for the event types described above. Each of these cells contains the xt of a word that handles the specific event type; you will find the source for these event handlers at the end of source file Event. A word defined at the beginning of the source file ObjInit, called -MODELESS, initializes fEvent with the correct xts whenever Mops starts up. This is accomplished by setting the System Vector OBJINIT to execute a word SYSINIT, which calls -MODELESS. (The name, by the way, arises because Mops initially has no modeless dialogs. In the source file Dialog+ there is a matching word +MODELESS which installs the event handling that is required if there is a modeless dialog active.)

Listening to events

The chief means by which you can cause Mops to listen to the event queue is by calling the Mops word KEY. This causes class Event to enter a loop that requests the next event from the queue and executes the indexed cell corresponding to the event type. Each handler word is responsible for leaving a boolean on the stack that tells class Event whether to return to the caller; currently, only Key-down and AutoKey events will trigger a return. Other events are managed as they come, triggering menu choices, window activation or updating, and control selections. To the original caller of KEY, all of this activity is invisible, because it will not resume execution until a keystroke is received. Thus, the caller of KEY enters a sort of "suspended animation" while the Macintosh handles non-keystroke events. This serves to separate the bulk of event management from the traditional, keystroke-oriented parts of your application, and was designed to simplify Macintosh programming for those used to more conventional systems.

As pointed out in chapter II.4, you might need to use the Mops word BECOME if you nest calls to KEY within several layers of code, because a menu or control choice could cause a new portion of the application to begin executing, and ultimately cause the system to run out of return stack. An alternate structure is to do all keystroke processing via an infinite loop at the top of your application that calls KEY and executes the Key: method of the front window. While less familiar to most of us, this architecture will probably result in a simpler application in the long run.

Specific event handling

Null events (all event types with the * above) can be used to execute the Idle: method for the front window. The programmer should use a window's Idle: method to perform any background processing that is required for that

particular window (such as call TEIdle in a text edit window).

The Idle: method should execute quickly so as not to bog down the responsiveness of the system to input.

Mouse-down events are handled based on what window region the click occurred in (from FindWindow - see IM Window Manager). Of the seven possible regions, only two are of real concern to the programmer, because Mops can take appropriate action for the others. If the mouse is clicked in a close box, the window executes whatever action word you have installed in the window's CLOSE vector, just as a content region click will execute the window's CONTENT vector. The Actions: method allows you to customize these two aspects of a window's mouse click handling. You might also have to redefine the Grow: method for your windows if they require resizing of controls or other unique behavior; Grow: is executed in response to a grow-region click.

The Key-down handler fetches the value of the key entered from the event record's Message field, first checking to see if the Command key was held down simultaneously. If so, the Menu Manager is called to process a potential key-equivalent menu choice. Key equivalents are thus managed automatically by Mops, requiring only that you specify the key equivalents in your menu item text definitions. If the Command key was not held down, Mops returns to the word or method that called KEY with the value of the entered key on the stack.

The Update handler sends a late-bound Draw: message to the corresponding window object, causing it to redraw its contents.

The Disk-inserted handler takes the normal default action, which is to check if the system has already mounted the disk, which it will have attempted to do. If the mount was unsuccessful, the handler calls the system routine DIBadMount to display the usual error message "This disk is unreadable". If you provide your own handler, put its xt in cell 7 of fEvent to process disk-inserted events.

The Activate handler determines whether the event is an Activate or Deactivate, and sends the appropriate late-bound Enable: or Disable: message to the window involved.

Event

Event associates a Toolbox event record with a dispatcher that executes a Mops word for each type of event received.

Superclass	X-Array		
Source file	Event		
Status	Core		
Instance variables			
int	what	The named ivars comprise an eventRecord—see IM.	
var	msg		
var	time		
var	loc		
int	mods		
int	mask		
Indexed data	None		
System objects			
fEvent	The system-wide Event object.		

X-Array, Array, Indexed-Obj, Object

Methods

accessing

type: (-- evt)

Returns the type of the last event received.

mods: (-- mods)

Returns the value of the mods field.

msg: (-- msg)

Returns the value of the msg field.

where: (-- point)

Returns the position of the mouse as a global, packed Toolbox Point.

msgID: (-- msgID)

Returns the high-level message ID. This is actually the same as the Loc field, but the different usage is made clear by the different name.

when: (-- ticks)

Returns the number of ticks (1/60ths of a second) since system startup.

set: (mask --)

Sets the event mask.

polling

next: (-- … b)

Gets next event out of event queue and executes the appropriate action vector, which leaves a boolean on the stack. Some events (such as key events) may leave other information on the stack under the boolean, depending on the action handlers.

key: (-- key)

Loops and polls the event queue (via next:) until a keystroke is received. During this time, all other events will be handled automatically as they come.

Error messages

See messages for class X-Array.

Mouse

Mouse integrates various Toolbox calls, providing easy access to the mouse's position in local coordinates, the state of the mouse button, and whether a double-click has occurred.

Superclass	Object
Source file	Event
Status	Core
Instance variables	
var	last Ticks value when the last click occurred.
var	interval Ticks between this click and the last one.
Indexed data	None
System objects	
theMouse	???

Object

Methods

accessing

get: (-- x y but)

Returns the mouse's local position and a boolean reflecting the state of the button.

where: (-- x y)

Returns the mouse's current position as a local Mops point.

click: (-- b)

Returns 2 if last click was a double-click, 1 otherwise.

put: (ticks --)

Updates the click interval with the current sysTicks value.

Error messages

None

Chapter 5—Windows

About this chapter

This chapter describes the Mops classes and words that manage windows for the application. Mops's window classes take away much of the burden of window management, providing the basis upon which you can build more detailed behavior. Standard Macintosh window behavior, such as dragging, growing and updating, is handled automatically by Mops's Window and Window+ classes, freeing you to solve application-level problems instead of constantly having to rewrite system-level code for window management.

Recommended reading

IM - Event Manager
IM - Window Manager
IM - QuickDraw
IM - Control Manager
Mops - Events
Mops - QuickDraw
Mops - Views
Mops - Controls

Source files

Window
WindowMod.txt
Window+

Using windows

Mops provides two classes of window objects: class Window, built into the distributed Mops.dic image, provides basic behavior necessary for all windows, but does not include any management of views (or controls, which are a view subclass). Another class, Window+, adds the behavior necessary for windows with views. Unless your application is a particularly "quick and dirty" one, which just needs an extremely simple text-only output along the style of a dumb terminal, we recommend you use the Window+ class. Lessons 18 and 19 of the Tutorial have already given an introduction to views, and how these interact with the Window+ class.

Because a Macintosh window record incorporates a QuickDraw GrafPort as the first portion of its data, class Window is a subclass of class GrafPort, inheriting both the GrafPort data and three GrafPort-related methods (see the QuickDraw section of this manual).

Windows, like controls and certain other Toolbox objects, have a dual identity in that part of the object is known only to Mops, while another part is known both to Mops and to the Toolbox. From the point of view of the Toolbox (and conventional languages like Pascal or C), a window is completely described by a window record. A Mops window object packages the window record data within the larger context of the object's private data, adding ivars to support the additional level of management that a Window object provides. The result is that the programmer is confronted with a much simpler model using objects, because all of the "boilerplate" kinds of behavior, such as dragging, growing, closing, updating and activation are handled within the window object itself rather than being thrown in with the application code. That is how Mops is able to simplify the logical model of the Toolbox and elevate it to a higher level, while still giving you the freedom to change any of the default behavior that occurs in such basic classes as Window.

There are two ways to create a new window using the Toolbox: you can ask that the Toolbox allocate the

window record on the heap, or you can provide the data area yourself. Because a Mops window object includes a window record as its private data, it always uses the second of

these methods, passing the address of its own data to the Toolbox as the storage to use for the window record. Of course, if you have an application in which windows come and go dynamically, so that you wish to allocate them on the heap, you can use the Mops ObjHandle mechanism to do so.

The fact that an object allocates a window record as the first part of its data is important, because it simplifies the interaction between Mops and the Toolbox. There are many cases in which Mops must determine which window is involved in event processing by calling the Toolbox, which will return a pointer to the window record. If the window record were not part of the object, Mops would have to somehow derive the address of the object's data from the window record. As it is, the window record is synonymous with the object's base address, making communication with the Toolbox much simpler. Other Mops objects, such as Controls, do not have this luxury, and must take extra steps to derive the object address.

Window objects add to the window record data a group of instance variables that keep track of the window's drag and grow characteristics, a boolean that tells whether the window is currently "alive" with respect to the Toolbox, and a set of action "hooks" that allow you to customize a window's behavior without necessarily having to create a subclass. These action vectors hold the xts of Mops words to execute when the window is involved in a content click, an update event, an activate event, or selection of the close box. The ClassInit: method of Window initializes the vectors to the xt of NULL, except for the activate vector, which is set to the xt of CLS ("clear screen", which erases the viewing area of the window).

For the Window+ class, which you should normally be using, you should leave the click handler and the update handler set to NULL (which they will be initially anyway), since clicks and drawing are handled through our view mechanism. You may, however, have a good reason to customize the activate or close handlers—for example, you may need to change menu items depending on which windows are open or in front.

Creating windows

The steps involved in creating and using a window are as follows: First, instantiate a window class (i.e. create a window object), and then initialize the action vectors of the window using the actions: method. For windows whose data exists in the dictionary or a module, this can occur at compile time:

```
window  myWind                    \ create a new window object
xts{ doClose doActivate null null }  actions: myWind
       \  Set the close, activate, draw and content vectors
```

The Activate vector is executed when the window becomes active, and the Close vector is executed when the use clicks the Close box. Typically, you will use both of these hooks to adjust items in your menus.

The Draw vector is called when the window receives an update event, which is the Toolbox's way of telling the window to redraw itself. Note however that drawing should now be done through our view mechanism, and not by setting the window's draw handler. At the moment we are really only maintaining a draw handler for backward compatibility, and it will probably disappear in future.

If the window is of class Window+, any views associated with the window will be redrawn automatically, since the DRAW: method for Window+, among other things, calls DRAW: on the contView, which causes DRAW: to be sent to all the views.

Lastly, the Content vector is called when the user clicks the mouse in the window's content region. Here again, you should now normally handle content clicks through the view mechanism—the click handler may also disappear in future.

You can also set the window's drag and grow characteristics at compile time, if the ClassInit: defaults do not suit your needs. Each requires a boolean on the top of the stack reflecting

whether the window is growable or draggable, and the four coordinates of a rectangle underneath the boolean if it is true. For example:

```
10 10 500 300 true setDrag: myWind
false setGrow: myWind
```

causes myWind to be draggable, but not growable. But note, in class Window, we actually ignore the rectangle coordinates passed in for setDrag: and setGrow:, and use a default value based on the size of the screen at the time the drag or grow is actually done. This is probably more generally useful than using any fixed values for the drag or grow limits. So unless you actually want fixed values and override these methods, you can just pass any four dummy values.

When your application executes, you must send a New: message to the window to cause it to become active with the Toolbox and to draw itself on the screen. New: requires a rectangle holding the dimensions of the window's frame, a title, a procID for the window type, and booleans reflecting whether the window should be visible when created and whether it should have a close box. For instance:

```
10 10 300 200 put: tempRect
tempRect " A New Window" docWind true true new: myWind
```

would create a new document window using the dimensions stored in tempRect that would be visible and have a close box. If you would rather define your window's characteristics using resources, you can call the GetNew: method to open the window using a template from a resource file.

To get a feel for how Mops's window objects can be used, it is most instructive to look at an existing application, such as grDemo. Lessons 18, 19 and 20 of the Tutorial deal with grDemo, and lessons 18 and 19 in particular give a good introduction to the View and Window+ classes.

Much of the code, as you will see, is concerned with initializing the various objects properly; much of the actual work is accomplished internally to the methods already defined for those objects.

Window

Window is the basic class of windows without controls.

Superclass	GrafPort
Source file	WindowMod.txt
Status	Core

Instance variables

32 bytes	wind1	Unstructured data for the window record.
handle	CtlList	Windows control list
12 bytes	wind2	Unstructured data for the window record.
rect	contRect	The rectangle defining the content region.
rect	growRect	Contains the window's current grow limits.
rect	dragRect	Contains the window's current drag limits.
int	growFlg	True if the window is growable.
int	dragFlg	True if the window is draggable.
int	Alive	True if the window is alive in the Toolbox.
var	Idle	The window's idle event action vector.
var	Deact	The window's deactivate event action vector.
var	Content	The window's content click action vector.
var	Draw	The window's update event action vector.
var	Enact	The window's activate event action vector.
var	Close	The window's close box action vector.
int	ResID	Resource id for GetNewWindow.

Indexed data None

System objects

fWind — The Mops system window used by the nucleus.

GrafPort, Object

Methods

setting characteristics

setContRect: (--)

Sets the content rectangle after the window has been resized. Also sets Mops's scrolling rectangle, used by CR, equal to the content rectangle.

setGrow: (l t r b T or F --)

Sets the window's grow limits. The old action was that if the boolean was true, the rectangle coordinates determined the minimum and maximum x and y values that the window could be grown. However the current action is to ignore these coordinates and use a SCREENBITS call instead. If the boolean is false, the window will not be growable.

setDrag: (l t r b T or F --)

Sets the window's drag limits. The old action was that if the boolean was true, the rectangle coordinates determined the minimum and maximum x and y values that the window could be dragged. However the current action is to ignore these coordinates and use a SCREENBITS call instead. If the boolean is false, the window will not be draggable.

setScroll (b --)

The passed-in boolean indicates whether scrolling is enabled for this window or not. This is primarily intended for the Mops window fWind, which supports scrolling text, and uses the temporary rectangle fpRect for this purpose. If a window doesn't support scrolling, then fpRect won't be altered when that window is active, so you can use it for your own purposes without conflict.

Note however, that the "proper" way to support scrolling text is via a Scroller view within a Window+.

setIdle: (xt --)

Sets the word which will execute in response to idle messages to the window.

set: (--)

Sets the window's grafPort as the current grafPort, and calls setView to update the content rect.

select: (--)

Makes this window the frontmost, active window.

size: (w h --)

Sets the dimensions of the window to the given width and height, without moving the window's upper-left corner.

setSize: (w h --)

The same as size:—this is for naming consistency with Rects and Views.

move: (x y --)

Moves the upper-left corner of the window to global coordinates x and y without changing its size.

center: (--)

Centers the window on the screen.

show: (--)

Calls ShowWindow to make the window visible.

hide: (--)

Calls HideWindow to make the window invisible.

actions: (close enact draw content 4 --)

Sets action vectors with the xts provided. We require an xt count (4 in this case) as this is standard for all actions: methods.

setAct: (enact deact --)

Sets the activate and deactivate vectors with the xts provided.

setDraw: (drawXt --)

Sets only the Draw action vector.

title: (addr len --)

Sets the title of the window to the passed-in string.

name: (addr len --)

An alias for title: (above).

querying

getName: (-- addr len)

Returns the window's title string.

getRect: (-- l t r b)

Returns the window's port rectangle coordinates.

getVSRect: (-- l t r b)

Returns the window's default vertical scroll bar rectangle coordinates. (Does not require a scroll bar to be present, but if it were, this is where it would be).

getHSRect: (-- l t r b)

Returns the window's default horizontal scroll bar rectangle coordinates.

maxX: (-- x)

Returns the x coordinate value which the top left corner of the window would have if the window were to be moved all the way to the right of the current screen (so the window's right hand edge would coincide with the right of the screen). Thus it is the maximum x coordinate value which the window could have without being in any way obscured. Doesn't actually move the window.

maxY: (-- y)

Likewise, returns the y coordinate value which the top left corner of the window would have if the window were to be moved all the way to the bottom of the current screen.

active: (-- b)

Returns true if the window is currently active.

alive: (-- b)

Returns true if the window is currently alive in the Toolbox.

event handling

draw: (--)

This method is executed when an update event occurs for the window. If the window is growable, a grow icon is drawn with scroll bar delimiters. The window's Draw action vector is executed.

idle: (--)

This method may be used for background processing. Whenever fEvent gets a null event out of the event queue (for instance, while waiting for the user to type a character) a late-bound idle: message is sent to the front (active) window. That window's idle: method can then do any background processing necessary (such as updating a clock picture). The idle method defaults to a do-nothing method in class Window, and should be kept short enough to keep from bogging down responsiveness to user input.

enable: (--)

This method is executed when an activate event occurs for the window. The window's Enact action vector is executed.

disable: (--)

This method is executed when a deactivate event occurs for the window. Does nothing in class Window.

update: (--)

Forces an update event to occur that will redraw the entire window. The window will not actually be redrawn until KEY is called and event handling is active.

close: (--)

This method is executed when the user clicks the window's close box. The window's Close action vector is executed, and CloseWindow is called.

release: (--)

The same as close:—this is our standard destructor name.

drag: (--)

This method is executed when a mouse-down event occurs in the window's drag region. The Toolbox is called to pull a gray outline around with the mouse. If inactive, the window is made active after dragging.

zoom: (part --)

This method is executed in response to a click in the zoom box. part is supplied by the system when the Event code calls FindWindow—it will be 7 if the window is to be zoomed in, or 8 if it is to be zoomed out. Note that although this method is included in class Window, the remainder of zoomable window support is in class Window+, so a zoomable window should therefore be a Window+.

grow: (--)

This method is executed when a mouse-down event occurs in the window's grow region. The Toolbox is called to pull a gray outline around with the mouse. If inactive, the window is made active after growing.

content: (--)

This method is executed when a mouse-down event occurs in the window's content region. The window's Content action vector is executed.

key: (--)

This method can be used to provide window-specific keystroke handling. The key: method in class Window simply does a DROP. See Chapter II.4 for more information.

object creation

new: (^rect tAddr tLen procID visible goAway --
)

Calls the Toolbox to create a new window using this object's data as the window record. Parameters determine the window's bounds in global coordinates, the title, the type (procID -- see dlgWind, docWind, rndWind) of window, and whether it is visible and has a close box.

getNew: (resID --)

Same as new:, but uses the resource template with resource id resID.

classinit: (--)

All objects of class Window are initially set to non-growable, non-draggable windows with null action vectors except for Draw, which is set to the xt of CLS.

test: (--)

Creates a test object of class Window.

Error messages

None

Window+

Window+ adds support for views, and also zooming. Unless your window is to be very basic indeed, you should use this class.

Superclass	Window
Source file	Window+
Status	Optional
Instance variables	
ptr	^contView Pointer to the ContView— the view consisting of the whole contents of the window.
bool	zoomFlg True if this window is to be zoomable.
Indexed data	None
System objects	None
Window, Grafport, Object	

Methods

accessing

setZoom: (b --)

Passed-in boolean indicates if this window will be zoomable.

getView: (-- ^view)

Returns a pointer to the ContView.

setView: (^view --)

Sets the ContView.

event handling

grow: (--)

Handles a click in the grow box.

zoom: (--)

Handles a click in the zoom box.

enable: (--)

Handles an activate event. Sends enable: to the Contview, which causes enable: to be sent to all child views as well.

disable: (--)

Handles a deactivate event. Sends disable: to the Contview, which causes disable: to be sent to all child views.

content: (--)

Handles a click in the content region of the window. Checks for a hit on a control. If it wasn't a control hit, then click: is sent to the ContView. This causes click: to be sent to any child views, until the view is found which actually contains the clicked position. The Click handler of this view is

then executed.

draw: (--)

Draws the window with its controls. draw: is sent to the ContView, which causes draw: to be sent to all child views. Any custom drawing should be done via the Draw handlers in the views.

idle: (--)

As for idle: in class Window, but also sends an idle: method to the ConView, which causes idle: to be sent to all child views.

close: (--)

Handles a click in the close box. Sends release: to the ContView, which causes release: to be sent to all child views. The storage for any controls is released via a KillControls call. Finally the window is closed and its storage released.

object creation

new: (^rect tAddr tLen procID vis goAway
^view --)

As for new: on class Window, except there is one additional parameter, a View pointer. This View will become the ContView. init: is called on this ContView, setting its ViewRect to the contents rectangle of the window. new: is then called on the ContView, which causes new: to be called on all child views and controls.

getnew: (resID ^view --)

As for new:, but the window data comes from the resource given by resID.

test: (--)

Creates a test window.

Error messages

None

Chapter 6—Views and Controls

About This Chapter

This chapter describes views, which are also used by MacApp, TCL, PowerPlant, the Newton, and most other OOP development systems for GUI (graphical user interface) platforms. A view basically defines a rectangular area within a window, within which drawing may take place. For those familiar with MacApp or TCL, there is a small difference in that in those systems a window is itself a view. In Mops, a window isn't a view, but contains a view or views.

Our basic Window class doesn't support views at all; for this you will need the Window+ class. Although you may draw directly in a Window via its Draw handler, we are now trying to discourage this. We don't even guarantee that in future windows will continue to have their own Draw handlers—in fact they probably won't!

You should now use a view within a Window+, and draw either by overriding the DRAW: method for the view, or by using its draw handler. This will give you much greater flexibility as to what you can do within the window. This comes about because a view can contain any number of child views. A child view is another view, but is constrained to be drawn within its parent view.

Class Control, which in earlier versions of Mops was an independent class, is now a subclass of View.

Recommended reading

Mops - Windows

Source Files

View

Using Views

A view must have an owning window. A view has an associated rectangular area within which anything owned by a view is drawn, including any child views.

A Window+ contains one special view (the ContView) which covers the whole drawing area of the window (excluding any scroll bars). All other views within the Window+ must be child views of the ContView (or child views of those child views, etc.).

The rectangular area of a view is defined in an ivar which is a rect, ViewRect. To be precise, this rectangle defines the outer boundary of this view, relative to the current grafPort. This rect is used by its owning view to set the clip region and the coordinate origin before calling any method on the view.

However, as we saw in the Tutorial in lesson 18, you will normally specify the size and position of a view by setting the bounds and justification for the four sides. Please see that Tutorial lesson for a full rundown of the various options you have.

Using Controls

Mops defines a generic Control class, which is a subclass of View, and then has subclasses for the different kinds of standard control—Button, Checkbox, RadioButton, Vscroll and Hscroll.

Since a control is a view, in order to use controls within your application, you must use a Window+ or a subclass of Window+, since these are necessary to support views.

As is the case with other Toolbox objects (such as Windows) control objects have a dual identity. Part of the control's data is maintained by the Toolbox on the heap, and can be accessed by the application via a handle. If

you were writing in a conventional language, such as C or Pascal,

you would consider the handle to be the control, and you would have to build a lot of structure into your code to support the user's selection of the various parts of the control. Mops, on the other hand, combines the control's Toolbox-related data with its own View-related data to comprise a single object that contains all it needs to know about managing the various actions that can occur. You need only instantiate and initialize the object properly, and it takes care of the rest. Even much of the initialization is handled automatically via the View class.

Controls store the xts of Mops words as action vectors that will be executed when the various parts of the control are selected. Simple controls (class Control) have a single action vector, while scroll bars have 5. You can use these classes as a model for defining your own control classes if you wish to define new types.

When you click in a control, the control object receives a click: message as part of the normal view handling. The click: method in class Control then calls the Toolbox routine FindControl to identify which part of the control has been clicked. After that, two different things may happen, depending on the control type and part number affected.

For buttons, check boxes and scroll bar thumbs, the control is highlighted while the button remains depressed, but no other action is taken. The Toolbox routine TrackControl takes care of highlighting the correct control part while the mouse is in its proximity and the button is down. When the button is released, a late-bound exec: message is sent to the control object, causing it to execute its action handler for the correct part.

For the other parts of a scroll bar, however, it is desirable that a custom routine be executed while the button is held down in the part. For instance, while you hold down the button in the up arrow of a scroll bar, an editor should gradually scroll the document in small increments until the button is released. This can be accomplished by passing a procedural argument to the TrackControl routine, but the procedure must look like a Pascal procedure rather than a Mops word. Mops contains a special compiler that packages Mops words in a way that makes them look like Pascal procedures (:PROC ... ;PROC). We have created one of these procedures to execute the action vector of a control repeatedly while the mouse button is down, and Mops passes this procedure to TrackControl in the case of the non-thumb scroll bar parts. Whatever actions you have defined for these parts will be executed while the part is being selected.

We have also provided a subclass of View named Scroller, which provides for a vertical and horizontal scroll bar along the edges of the view, with action vectors already defined for you.

Creating control objects

Defining a control object requires three steps. First, instantiate the object with a phrase like:

```
button saveBtn
```

You should then initialize the newly created object to assign it a position and a title. For example:

```
100 250 " Save"  init: saveBtn
' doSave  actions: saveBtn
```

Here we define saveBtn as a Button, specify that its top left corner will be be at coordinates (100, 250) relative to the view that it will appear in, and give it a title. Then we set doSave as its action word. DoSave will be executed if the user releases the mouse button while the mouse is within saveBtn's control rectangle. Finally, when the program executes, we must use addView: to add the control to its parent view's list of child views. Then when we fire up the window with new: or getNew:, the control will automatically receive a new: message which will cause it to create a Toolbox Control record on the heap and draw itself.

Control action words often need a way to determine which control they have been dispatched from. For example, a common action taken in scroll bar arrows is to get the control's value, add some increment to it, and put the new value in the control. This could be done in the following manner:

```
: doUpArrow   get: thisCtl  1-  put: thisCtl  -1 scroll: theText ;
```

In this example, the word thisCtl is actually a Value that Mops provides as a simple way for a control action word to derive its owning control object. Mops automatically compiles a late-bound reference when a Value is used as the receiver of a message, and in this case thisCtl contains the control object's address. This allows you to write very general action words that can be assigned to several different control objects simultaneously.

Design issues

Because late-bound messages must be sent to controls and windows, these objects cannot be defined as normal named ivars, because to do so would fail to provide a class pointer for the runtime method lookup. If you wish to make a control or window an ivar, you will need to define a subclass with the General attribute, then use that class. However you don't need to do this with Vscroll or Hscroll, since these have already been defined as General.

Late binding is necessary because there are cases in which the Toolbox returns the address of an object to the application, but is is undesirable to make any assumptions about the actual class of the object. For instance, when you click the mouse button, the Toolbox call FindWindow tells you in which window the click occurred. This requires that a Content: message be sent to the object, but because the programmer is free to define subclasses of class Window, there is no way to know ahead of time what class the window object belongs to.

Dialogs

Mops implements controls in dialogs differently than in normal windows. Since dialogs rely heavily upon resource definitions and don't usually occasion much interaction with the items themselves other than getting or setting values, Mops does not build dialog control items as objects, but rather accesses them through methods in the Dialog class itself. This saves a lot of space, and actually simplifies the interface for the programmer.

View

description ???

Superclass	Object
Source file	View
Status	Optional

Instance variables

rect	viewRect	Bounding rectangle, rel to grafport.
rect	bounds	We use this to set the viewRect
ptr	^parent	Points to parent (containing) view
ptr	^myWind	Points to owning window
ptrList	children	List of child views
x-addr	draw	Draw handler
x-addr	clickHndlr	Click handler
bool	alive?	
bool	enabled?	
bool	wantsClicks?	True if we can accept clicks
bool	setClip?	True if we need to set the clip (default)
bool	measureFromMe?	True if other siblings are to use this view for sibling relative justification modes
byte	#updates	Counts number of pending updates
byte	Ljust	Left justification
byte	Tjust	Top
byte	Rjust	Right
byte	Bjust	Bottom

Indexed data	None

System objects None
Object

Methods

accessing

getViewRect: (-- l t r b)

Returns the viewRect coordinates.

^viewRect: (-- addr)

Returns the address of the viewRect object.

bounds: (-- l t r b)

Returns the bounds coordinates.

getBounds: (-- l t r b)

A synonym.

getJust: (-- lj tj rj bj)

Returns the current justification values for the four sides in the usual order.

enabled?: (-- b)

Returns true if this view is enabled.

window: (-- ^wind)

Returns the address of the owning window object.

setWindow: (^wind --)

Sets the owning window to the window whose address is passed in.

wantsClicks: (b --)

Sets the wantsClicks? ivar to the passed-in boolean value.

setClick: (xt --)

Sets the click action vector, and sets wantsClicks? to true.

setDraw: (??? --)

Sets the draw action vector.

setParent: (^view --)

Sets the parent view address (which is stored in the ivar ^parent).

update: (???) Generates an update event for the view.

clear: (???)

Clears the view.

View positioning

bounds>viewRect: (???)

Used internally by the view mechanism. Uses the current bounds and

justification values to set the viewRect.

childrenMoved: (???)

Used internally by the view mechanism. Signals this view that its children's positions have to be recomputed. Causes this view to send moved: to all its children (see below).

moved: (???)

Called when something has happened to change the position of this view (such as the parent view moving, or the bounds or justification parameters changing), requiring the viewRect to be recomputed. This method calls bounds>viewRect: self and childrenMoved: self.

object creation

Note that addview: must be called at run time, since a view's address is passed in. new: is called at run time, once this view's parent view already exists. However, new: is normally called automatically, when new: is sent to the owning Window+ object. The Window+ calls new: on its contView, and everything continues from there, since new: on a view calls new: on all its child views.

setJust: (lj tj rj bj --)

Sets the justification values for the four sides in the usual order.

setBounds: (l t r b --)

Sets the bounds for the four sides in the usual order.

measureFrom: (b --)

Sets the MeasureFromMe? ivar. If set True, this view will be used as a reference if any of its siblings use sibling-relative justification.

addview: (^view --)

Adds the passed-in view to this view's list of child views. This method can only be called at run time.

new: (^view --)

Fires up the view at run time, and calls new: on all its child views. The passed-in view is the containing view of this view, or nilP if there is none (i.e. this is the ContView of a window). Note that new: is normally called automatically via a new: on the owning Window+.

manipulation

release: (--)

Releases this view and all its child views. This too is normally called automatically via a release: on the owning Window+.

classinit: (--)

Sets the initial defaults. SetClip? and WantsClicks? are set True. The justification values are set to parLeft, parTop, parLeft, parTop.

display

draw: (--)

Draws the view (and all its child views). The setting up and winding up is done via the callFirst/callLast mechanism, and here in class View itself the only other action that draw: takes is to execute the Draw handler. Please see the comments in the source code for more detail.

event handling

idle: (--)

Called when idle: is sent to the owning window. Does nothing in class View itself, except to call idle: on all the child views.

click: (-- b)

Called when a click has occurred in the owning window's content area. Returns True if the click was actually in this view (or one of its child views), otherwise returns False. If the click was in this view itself (not in a child view), the Click handler is executed.

key: (c --)

Called when a key event has occured and this view's window is in front. Does nothing in class View itself, except to call key: on all the child views.

enable: (--)

Enables the view. Called from Window+ when the window is enabled.

disable: (--)

Disables the view. Called from Window+ when the window is disabled.

drawX: (???)

Can be useful in debugging, when you want to see the view but don't have "real" drawing code yet. It just draws a big X across the view area, joining the diagonally opposite corners.

Error messages

None

Control

Control is a generic superclass for controls.

Superclass	View
Source file	Ctl
Status	Optional
instance variables	
int	procID The control definition ID for the Toolbox.
int	resID The control's resource ID.
handle	CtlHndl Handle to the control record.
int	myValue Contains a copy of the numeric value of the control.
int	titleLen Length of the control's title.
32 bytes	title The text of the title.
Indexed data	None
System objects	None

View, Object

Methods

accessing

putResID: (resID --)

Stores the resource ID for the control. Used if the control is defined via a resource.

get: (-- val)

Returns the value of the control.

put: (val --)

Sets the value of the control.

handle: (-- hndl)

Returns the control handle.

setTitle: (addr len --)

Sets title of control.

getTitle: (-- addr len)

Gets title of control.

exec: (part# --)

Executes the control's action handler.

manipulation

moved: (--)

As for moved: on the superclass View, but also contains a call to MoveControl to inform the system of the control's new position.

hilite: (hiliteState --)

Hilights, disables, or enables the entire control.

enable: (--)

Enables the control.

disable: (--)

Disables the control.

hide: (--)

Calls Toolbox HideControl.

show: (--)

Calls Toolbox ShowControl.

click: (--)

As for click: on the superclass View, but contains some necessary toolbox calls as described above in the introduction to this class.

object creation

new: (--)

As for new: on the superclass View, but contains a call to NewControl to create a new control in the Toolbox for this control object. The initial value is 0, and the range is 0 to 1.

getnew: (--)

As for new:, but uses a resource. The resource ID must already have been set via putResID:.

release: (--)

Releases the control handle, then calls release: super to do the normal View releasing action.

classinit: (--)

Sets default control to type "button" with null click action, and a zero length title.

display

draw: (--)

As for draw: on the superclass View, but contains a call to Draw1Control so that the system will do the actual drawing of the control.

Error messages

None

TitledCtl

TitledCtl just adds a convenient init: method for setting up a control with a title, where the width of the control's rect is determined by what the title is. We assume the font will be Chicago and the height of the control is 20. This may be overridden in subclasses as necessary.

Superclass	Control
Source file	Ctl
Status	Optional
Instance variables	None (see Ctl)
Indexed data	None
System objects	None

Control, View, Object

Methods

object creation

init: (x y addr len --)

Initializes the control (may be done at compile time). Sets up the control with a title. x and y are the initial top left Bounds values (using whatever justification is in effect). (addr len) gives the title.

Error messages

None

Button, RadioButton, CheckBox

Button, RadioButton and CheckBox provide support for those types of control. The only change to TitledCtl is the customization of classinit: to set the appropriate proc ID so that the system will draw the right kind of control.

Superclass	TitledCtl
Source file	Ctl

Status	Optional
Instance variables	None
indexed data	None
TitledCtl, Control, View, Object	

Methods

classinit: (--)

Initializes the control with the appropriate proc ID.

Error messages

None

Vscroll, Hscroll

Vscroll and Hscroll provide support for vertical and horizontal scroll bars. Class Hscroll is set up as a subclass of Vscroll, and just overrides classinit: to set the ivar horiz? to true. It takes no other special action. Several methods interrogate this ivar and do the appropriate thing.

Superclass	Control
Source file	Ctl
Status	Optional
Instance variables	

int	min	The current minimum value for the control.
int	max	The current maximum value.
5 ordered-col	parts	Give the part IDs for the various parts of a scroll bar.
5 x-array	actions	The corresponding action handlers.
bool	horiz?	True if this is a horizontal scroll bar; false if vertical scroll bar.

Indexed data	None
System objects	None
Control, View, Object	

Methods

accessing

actions: (up dn pgUp pgDn thumb 5 --)

Sets up the action handlers for the 5 parts. up etc. are xts, and we require a 5 on top of the stack as an xt count, as for all actions: methods.

put: (val --)

Sets the value of the scroll bar. The value is coerced to not fall outside the current minimum and maximum.

putMax: (val --)

Sets the maximum value.

putMin: (val --)

Sets the minimum value.

putRange: (lo hi --)

Sets the minimum and maximum.

object creation

init: (top left len --)

Initializes the scroll bar (may be done at compile time).

new: (^view --)

Creates a new control in the Toolbox for this scroll bar object. ^view is the address of the owning view, within which the control will appear. new: will normally be called automatically when new: is done on the view.

getNew: (resID ^view --)

Creates a new control in the Toolbox for this scroll bar object, using a resource.

classinit: (--)

Sets action handlers to null and selects scrollbar type control.

Class Hscroll is set up as a subclass of Vscroll, and overrides classinit: to set the ivar horiz? to true. It takes no other special action.

display

disable: (--)

Sets entire control to 255 hiliting (disabled).

enable: (--)

Sets entire control to enabled hiliting.

execution

exec: (part# --)

Called from the event handling code when there is a mouse-down in the given part of the scroll bar. Executes the corresponding action handler.

Error messages

None

Scroller

Scroller is a view which has support for a vertical and horizontal scroll bar along the right hand and bottom edge respectively. We implement it with three child views: mainView, which is the display area, and the two scroll bars themselves. These child views are ivars of Scroller. MainView is an instance of a one-off class, Mview. This class has a rectangle, PanRect, which normally ought to enclose all the child views of the Mview. The usual scenario is that PanRect is larger than the viewRect, and scrolling amounts to shifting the child views (and PanRect) around within the viewRect—which, from another point of view, can be thought of as "panning" the viewRect over the PanRect area. Mview has appropriate methods for returning the distances by which PanRect falls outside the viewRect area, so that the parent Scroller can set the scroll bar values appropriately. One unusual thing we do here is to override addView: on Scroller so that it becomes an addView: on MainView, since this is usually what we really mean. In the case where you want to really addView: on the Scroller, such as to add another child view alongside one of the scroll bars, you should subclass Scroller with the extra views as ivars, and at run time do addView: super as we do for the scroll bars (see the new: method). Another approach we could have taken to implementing MainView would have been as a pointer, with late binding. That way MainView could have been any view subclass. That would have been more flexible, but possibly overkill for what we usually want to do—it would have required a more complex setting-up process, with the MainView address having to be passed in after new: has been done. But if you need the extra flexibility, feel free to clone Scroller and make the changes!

PanRect can obviously be very big, so we don't implement it as a regular rect, but define a new class, BigRect, which uses vars rather than ints for the coordinates.

Superclass	View
Source file	Scroller
Status	Optional
Instance variables	

Mview	mainView
vscroll	theVscroll
hscroll	theHscroll
bool	vscroll? True if v scroll bar to be used

bool	hscroll?	True if h scroll bar to be used
bool	usePanRect?	True if we're to use PanRect
var	Hpan	Horizontal panning range
var	Hpos	Current vertical posn
var	Vpan	Vertical ditto
var	Vpos	
int	Hunit	# pixels for one horizontal arrow click
int	Vunit	# pixels for one vertical arrow click

Indexed data None

System objects None

View, Object

Methods

accessing

Vscroll: (b --)

Passed-in boolean is true if this Scroller is to have a vertical scroll bar.

Hscroll: (b --)

Passed-in boolean is true if this Scroller is to have a horizontal scroll bar.

putPanRect: (l t r b --)

Sets the PanRect directly. The default, however, is to use the rectangle bounding all the child views.

>Hunit: (n --)

Sets the Hunit—the number of pixels for one horizontal arrow click.

>Vunit: (n --)

Sets the Vunit.

>Hrange: (lo hi --)

Sets the lower and upper limits for the horizontal scroll bar. This will normally be done automatically whenever the size of the PanRect is set.

>Vrange: (lo hi --)

Sets the lower and upper limits for the vertical scroll bar.

object creation

new: (--)

Fires up the Scroller at run time. Calls addView: self with mainView and the two scroll bars, since as they are child views as well as ivars. Note that new: is normally called automatically via a new: on the owning Window+.

manipulation

?Venable: (--)

Enables the vertical scroll bar, if there is one and there is a scrolling range (that is, if the PanRect extends beyond the ViewRect in the vertical direction).

?Henable: (--)

Enables the horizontal scroll bar, if there is one and there is a scrolling range.

enable: (--)

Enables the Scroller. The above two methods are called to set the enabled/disabled status of the scroll bars appropriately.

disable: (--)

Disables the Scroller.

moved: (left top --)

As for moved: on a View, but includes handling of the scroll bars.

pan: (dx dy --)

Pans the view over the child views by the given distance. The scroll bars aren't altered—use panRight: etc. for this, since they adjust the appropriate scroll bar and then call pan:.

Note that panning doesn't actually move the view within its parent view, but the panning is accomplised by shifting the child views in the opposite direction. Our convention is that positive dx and dy correspond to a pan to the right and down, which means that the child views are being shifted to the left and up, which is a "negative" shift.

panRight: (dx --)

Pans the view to the right by dx pixels, or until the scroll limit is reached. Adjusts the horizontal scroll bar appropriately.

panLeft: (dx --)

Pans the view to the left by dx pixels likewise.

panUp: (dy --)

Pans the view up by dy pixels likewise.

panDown (dy --)

Pans the view down by dy pixels likewise.

Hpage: (-- #pixels)

Returns the number of pixels corresponding to a "page" in the horizontal direction.

Vpage: (-- #pixels)

Returns the number of pixels corresponding to a "page" in the vertical direction.

1right: (--)

Handles a click in the right arrow of the horizontal scroll bar.

1left: (--)

Handles a click in the left arrow of the horizontal scroll bar.

1up: (--)

Handles a click in the up arrow of the vertical scroll bar.

1down: (--)

Handles a click in the down arrow of the vertical scroll bar.

pgRight: (--)

Handles a click in the "page right" region of the horizontal scroll bar.

pgLeft: (--)

Handles a click in the "page left" region of the horizontal scroll bar.

pgUp: (--)

Handles a click in the “page up” region of the vertical scroll bar.

pgDown: (???)

Handles a click in the “page down” region of the vertical scroll bar.

Hdrag: (--)

Handles a drag of the horizontal scroll bar thumb.

Vdrag: (--)

Handles a drag of the vertical scroll bar thumb.

classinit: (--)

Initializes the Scroller to the default configuration—with both scroll bars, and both Hunit and Vunit set to 4. The justifications of the scroll bars are set so that they will be placed along the right and bottom edge of the Scroller. The action handlers of both bars are set to call the appropriate methods above.

Error messages

None

TEScroller

TEScroller is a view which displays a TextEdit record in mainView (the main display area). All the customary text editing operations are supported, including (of course) scrolling. The main part of the Mops window (excluding the stack display) is a TEScroller. The TextEdit record is handled via an ivar theTE, which is an instance of the class TextEdit. Class TextEdit is probably not much use in isolation, (it would normally be used via this TEScroller class), so we won't document it in detail here in the manual. However the source code (in file TextEdit) should be reasonably self-explanatory.

Superclass	Scroller	
Source file	TEScroller	
Status	Optional	
Instance variables		
TextEdit	theTE	
rect	rPanRect	TE needs a rect, not a bigRect. So we mirror our PanRect here.
Indexed data	None	
Scroller, View, Object		

Methods

Note: Many of the methods are just overridden versions of the methods of the superclass Scroller, where there is an additional action to perform to the TextEdit object, but where the behavior of the method is really the same. We won't list these individually, but concentrate on the methods that are new in this class.

accessing

textHandle: (-- hndl)

Returns a handle to the text in the TextEdit object.

handle: (-- TEhndl)

Returns the TEHandle.

size: (-- n)

Returns the number of characters in the text.

getSelect: (-- start end)

Returns the selection range. Start and end are character offsets in the text.

selStart: (-- n)

Returns the offset of the start of the selected range.

selEnd: (-- n)

Returns the offset of the end of the selected range.

getLine: (-- start end)

Returns the offset of the start and end of the line with the cursor. No text need be selected.

lineEnd: (-- n)

Returns the offset of the end of the line with the cursor.

editing

?scroll: (x y --)

If necessary, does a scroll so that the point (x, y) is in view. X and y are relative to the window, not the view.

setSelect: (start end --)

Sets the selection range.

caretLoc: (-- x y)

Returns the (window-relative) coordinates of the caret position.

caretIntoView: (???)

Scrolls if necessary so that the caret is in view.

key: (c --)

Handles a typed key, by passing it to the TextEdit object which will call TEKey.

insert: (addr len --)

Inserts the passed-in string into the text.

$insert: (^str --)

Inserts the active part of the passed-in string+ object.

cut: (???)

Handles a cut operation. The TextEdit object will call TECut.

copy: (???)

Handles a copy operation.

paste: (???)

Handles a paste operation.

clear: (???)

Handles a clear operation.

Chapter 7—Menus

About This Chapter

This chapter describes the Mops classes and words that allow you to build your application's menus.

Recommended Reading

IM - Event Manager
IM - Menu Manager
Mops - Events

Source Files

Menu
MenuMod.txt

Using Menus

Mops menus integrate the Toolbox concept of a menu within an object that stores Mops words to be executed when the user makes a particular choice. The Mops event object, fEvent, takes care of actually pulling down the menus by tracking the mouse and calling the menu manager whenever there is a click in the menu bar. FEvent also handles key-equivalents for you automatically if you declare the key equivalents in your item text. If the user makes a choice, a message is sent to the Mops menuBar object telling it to find the menu affected and execute the cell indexed by the item number chosen. Menus are a subclass of X-Array, which provides the ability to execute one of a list of xts by index.

There are two steps to defining the menus for your application. You must first create an object of class Menu for each menu in the application, and allocate as many indexed cells to each menu object as there are items to be selected. For example:

```
7 menu  FileMen           \  FileMen can have at most 7 items.
```

However, if you have many menu items with similar behavior, such as in a font menu, you only need to provide one indexed cell for that behavior, provided it is the last. This is because if a menu item is selected whose number is beyond the last indexed cell of the menu object, no error occurs, but the last indexed cell is executed. This feature is useful for the Apple menu as well.

Then, in ResEdit, you create menu resources for your menus. This is quite easy to do. You will need to assign a resource ID for your menus in ResEdit—it is quite satisfactory to give the Apple menu ID 1, the Edit menu ID 2 and so on. Then when you initialize each menu object in Mops with the init: method, you will pass the corresponding resource ID, along with an xt list giving the word to be executed corresponding to each menu item. Then at run time you send getnew: to each menu object to cause the menu to appear.

Menu items which call hierarchical menus need to have a special character appended to the name of the menu item. See IM-IV for more info.

After your menus are set up, various methods are available to change their characteristics. getitem: and setitem: fetch and store the item string for a given item. Note that while the Menu Manager numbers items from 1 to N instead of starting from 0, in Mops we follow our normal convention of starting from 0. This means that the xt for the word to be executed when item N is selected (using our numbering) will be item number N in the xt list. The Toolbox automatically highlights (turns black) any menu title for which an item is chosen, and the normal: method can be used to unhighlight any menu. Class Menu automatically does a normal: after a handler returns, but some never return if they do a BECOME or an ABORT. Class Menu's enableItem: and disableItem: methods are useful during activate events, when you should ensure that only those menu items are enabled that

are appropriate for the current window and the current state. If

you want to enable or disable an entire menu, use enable: and disable:. To enable or disable the entire menu bar, use MBar's enable: and disable: methods. Finally, check: and unCheck: (in Menu) control the display of a checkmark next to an item.

Mops defines a single instance of class MBar, called menuBar. Because forward references to this object were necessary, menuBar is vectored through a Value. As a result, any messages that you send to the menuBar object will be late-bound. You should rarely need to communicate with MenuBar directly, because most of the methods important to an application are in class Menu.

Menu

Class menu creates objects that associate Mops words with each of the items in a Macintosh menu. The Mops word associated with an item is executed when that item is chosen by the user.

Superclass	X-Array
Source file	Menu, MenuMod.txt
Status	Core
Instance variables	
int	resID Resource ID of this menu
var	Mhndl \Handle to the menu's heap storage
Indexed data	4-byte cells (must be xts of valid Mops words)
System objects	None
X-Array, Array, Indexed-Obj, Object	

Methods

object creation/deletion

putResID: (resID --)

Sets the resource ID for this menu.

init: (xt1 ... xtN N resID --)

Sets the xt list and resource ID for this menu. This may be done at compile time.

new: (addr len --)

Calls the toolbox to create a new menu using this object's data as the menu record. Not resource based. The title is passed in.

getNew: (--)

Calls the toolbox to create a new resource-based menu. We recommend that you use init: and getNew: as the normal means of creating a menu.

insert: (--)

Inserts this menu into the menu bar.

addRes: (type --)

Adds all resources of a type to this menu. Use this to append fonts or Apple menu items to a menu.

release: (--)

Releases the menu's heap storage.

normal: (--)

Removes the highlighting from all titles across the menu bar (regardless of which menu is the subject of this selector).

operations on individual items

getItem: (item# -- addr len)

Returns the text string for an item.

putItem: (item# addr len --)

Replaces the text string for the item by the passed-in text. The "meta-characters" described in IM don't apply.

insertItem: (item# addr len --)

Inserts a new item after the item given by item#. The “meta-characters” described in IM apply—for example, if the text begins with a hyphen, the item will be a dividing line across the width of the menu.

deleteItem: (item# --)

Deletes the given item.

addItem: (addr len --)

Adds a new item to the end of this menu, with the passed-in text. The text may be blank but should not be the null string. The “meta-characters” described in IM apply.

add: (addr len --)

An alias for addItem:.

enableItem: (item# --)

Makes an item eligible for selection by the user (black).

disableItem: (item# --)

Makes an item ineligible for selection by the user (gray).

openDesk: (item# --)

Runs the desk accessory/Apple menu item named by the specified item.

exec: (item# --)

Executes the handler for this item.\ Menu handlers will have item# on the stack when they execute, and they should leave it there. This way, they can ignore it if they want to, which will be the most common situation. If the item# is too great for this menu, we actually execute the last item rather than give an error. This allows us to save memory when a menu may have dozens of identical items. But of course we don’t alter the item# on the stack.

check: (item# --)

Shows a check mark by the item.

unCheck: (item# --)

Removes a check mark from an item.

accessing

ID: (-- resID)

Returns the resource ID of this menu.

handle: (-- hndl)

Returns the menu handle.

Error messages

"You must send me a new: message first"

An operation was attempted before the menu was created with the Toolbox.

AppleMenu

Subclass AppleMenu facilitates standard Apple Menu support, by filling the menu with all the DAs/Apple Menu items at getNew: time.

Superclass	Menu
Source file	MenuMod.txt
Status	Core
Instance variables	none
Indexed data	4-byte cells (must be xts of valid Mops words)
System objects	
AppleMen	Apple menu usable by any application. The first item executes AboutVec, a vector which can be set to your "About..." handler word.

Menu, X-Array, Array, Indexed-Obj, Object

Methods

getNew: (item# --)

As for getNew: in Menu, but also adds all the DAs/ Apple Menu items via addRes: self.

Error messages

"You must send me a new: message first"

An operation was attempted before the menu was created with the Toolbox.

EditMenu

Subclass EditMenu facilitates standard DA support. The exec: method first calls SystemEdit so any active DA gets a go at it.

Superclass	Menu
Source file	MenuMod.txt
Status	Core
Instance variables	none
Indexed data	4-byte cells (must be xts of valid Mops words)
System objects	None

Menu, X-Array, Array, Indexed-Obj, Object

Methods

exec: (--)

As for exec: on Menu, but includes a SystemEdit call.

Error messages

"You must send me a new: message first"

An operation was attempted before the menu was created with the Toolbox.

PopupMenu

Class PopupMenu provides support for popUp menus in dialogs. (If you need one somewhere else, you will have to define a subclass with different init: and normal: methods.) The sequence for setting up a pop-up menu is first to initialize the menu and dialog objects via init: methods—this may be done at compile time. At run time, send getnew: to the dialog and menu, then send a link: to the menu, passing the address of the dialog object. See the example code at the end of the file PopupMenu. The handling of pop-up menus is notoriously tricky, but we handle most of the mundane details for you.

Superclass	Menu
Source file	PopupMenuMod.txt, PopupMenu
Status	Optional

Instance variables

string+	ITEMTEXT	Current text displayed in pop-up box

int	ITEM# Current item # displayed in pop-up box
int	BOX# Dialog item# of pop-up box
int	TITLE# Dialog item# of pop-up title
rect	TITLERECT
rect	POPUPBOX
dicAddr	^DLG Points to owning dialog

ptr	F-LINK Forward link for chain of pop-up menus belonging to the one dialog.
Indexed data	4-byte cells (must be xts of valid Mops words)
System objects	None
Menu, X-Array, Array, Indexed-Obj, Object	

Methods

accessing

getText: (-- addr len)

Returns the text to be displayed in the pop-up box.

putText: (addr len --)

Sets the text to be displayed in the pop-up box. Doesn't attempt to actually display it, since the dialog may not be initialized.

item#: (-- item#)

Returns the number of the menu item displayed in the pop-up box.

putItem#: (item# --)

Sets the number of the menu item to be displayed in the pop-up box.

putTitle#: (title# --)

Sets the dialog item number of the pop-up title (note—not the pop-up box!).

put^dlg: (^dlg --)

Stores the passed-in pointer to the owning dialog object.

f-link: (-- ptr)

Returns the address of the next popUpMenu object belonging to the same dialog. Returns nilP if none.

set-f-link: (ptr --)

Sets the link to the next popUpMenu object belonging to the same dialog. This link (a pointer to a popUpMenu object) is passed in.

box#: (-- n)

Returns the dialog item number of the pop-up box.

object creation

init: (xt-list resID box# title# --)

Initializes the menu.

getNew: (--)

As for getNew: on Menu.

link: (^dlg --)

Call this at run-time, after you have sent getnew: to both the dialog and the menu. This method handles all the housekeeping for associating the menu with the dialog. For example, it installs a userItem handler for the pop-up

box (PUBoxProc in file PopupMenu). This word is called from the dialog whenever the pop-up box needs to be redrawn. (Don't confuse this with the action handler for the pop-up box, which handles a click on the box, and which you have to specify when you send init: to the dialog.)

operations

normal: (--)

Unhilites the pop-up title.

drawText: (--)

Draws the text in the pop-up box.

drawBox: (--)

Draws the pop-up box, with the required drop shadow and down-pointing arrow symbol. This method will be called automatically when the dialog box is updated, via our userItem handler PUBoxProc. (The pop-up box is a userItem in the dialog, and at link: time we automatically install PUBoxProc as the userItem handler.)

hit: (--)

Handles a click on the pop-up box. Your action handler for the pop-up box (which is an item in the dialog) should include sending hit: to the menu. As well as handling details such as the hiliting of the pop-up box, hit: sends exec: super, to execute your handler for the selected menu item. When your handler gets called, the previously selected item# (whose text was in the pop-up box before) will still be in the item# ivar, while the new one will be on the stack. This will be useful if you need to know if the item# is being changed or not.

Error messages

"You must send me a new: message first"

An operation was attempted before the menu was created with the Toolbox.

Mbar

MBar is used to create a single system object, MenuBar. It maintains the list of menu objects and their IDs, and is chiefly useful at startup to build and draw the menu bar via messages sent by the menu text loader.

Superclass	Object
Source file	Menu
Status	Core
Instance variables	
24 WordCol	IDs The list of menu IDs.
24 Array	Menus An array of menu objects.
Indexed data	None
System objects	
menuBar	System-wide menu bar (vectored).

Object

Methods

accessing

clear: (--)

Clears all menus out of the menu bar.

add: (men0 … menN #menus --)

Specifies the menu objects to be added to the menu bar.

new: (--)

Calls the toolbox to insert each menu from add: into the menu bar and draws the menu bar.

init: (men0 … menN #menus --)

Combines the actions of clear:, add: and new:.

draw: (--)

Draws the menu bar. (Primarily used by new:).

enable: (--)

Enables all menus in menu bar.

disable: (--)

Disables all menus in menu bar.

exec: (item# menuID --)

Executes the handler for the given item in the given menu.

click: (--)

Handles a click in the menu bar, calling the toolbox to track menu selection until the mouse button is released. Then executes the handler for the selected item (if any).

key: (chr --)

Executes the handler of an item selected by a command key combination, if any.

Error messages

None

Chapter 8—Graphics

About This Chapter

This chapter describes the Mops classes and words that provide an interface to the Macintosh QuickDraw graphics package. This part of the Toolbox is responsible for all of the basic graphics management underlying windows, controls, dialogs and menus, and can also be called directly to accomplish various drawing tasks. Because QuickDraw is so pervasive in the Macintosh, you should read the Inside Macintosh chapter on QuickDraw as a first step in learning how to use the rest of the User Interface Toolbox. While Mops provides an easy interface to much of the QuickDraw package, it is very useful to try and get an understanding of the basic philosophy behind QuickDraw graphics by reading Inside Macintosh before you attempt to do any sophisticated graphics programming. Other Toolbox modules, such as the Window Manager and the Control Manager, rely heavily upon QuickDraw to do much of their actual work.

Recommended Reading

IM - QuickDraw
IM - Window Manager
IM - Toolbox Utilities

Source Files

QD

Using the Graphics Classes

Class Rect is the most widely used QuickDraw class, describing an extensive set of behaviors appropriate to basic rectangles. Rectangles are used for a variety of operations in the Toolbox, such as sizing windows, controls and dialogs, drawing rectangular frames, filling with a pattern, and clipping. Class GrafPort is the superclass of window, and describes the graphics structure that QuickDraw uses as its foundation for windowed behavior.

The Mops word Cursor serves as a defining word to associate names with cursor resource definitions. For instance:

```
1 cursor  IBeamCurs
```

associates the name IBeamCurs with the cursor data whose resource ID in the system is 1. This happens to be the cursor used for editing text. After this declaration, you can change the current cursor to this image by simply executing the word IBeamCurs. Mops predefines four cursor images for you: IBeamCurs, crossCurs, plusCurs, and watchCurs.

Mops provides access to the system pattern list via the word sysPat. For instance, the phrase:

```
3 SysPat
```

leaves a pointer to the system pattern with ID 3. This can be used to send a Fill: message to various graphics objects.

Point

Point provides a building block for Rect. A Rect is composed of two Points, each consisting of two Ints, Y and X. Point's methods are useful in providing more advantageous access to a rectangle's data. QuickDraw stores Y before X in Rectangles, but Mops always represents Points on the stack as (X Y --).

Superclass	Object
Source file	QD
Status	Core
Instance variables	

int	Y	The Point's Y coordinate.
int	X	The Point's X coordinate.

Indexed data	None
System objects	None
Object	

Methods

accessing

get: (-- x y)

Returns the values in Y and X.

getX: (-- x)

Returns the value in X.

getY: (-- y)

Returns the value in Y.

int: (-- x:y)

Returns a single longword with x and y packed into the high and low words, as required for a number of Toolbox calls.

put: (x y --)

Stores new values in Y and X.

putX: (x --)

Stores a new value in X.

putY: (y --)

Stores a new value in Y.

Error messages

None

Rect

Rect is a widely used class that describes various properties of Rectangles.

Superclass	Object
Source file	QD
Status	Core
Instance variables	

Point	TopL	Point describing top left corner.
Point	BotR	Point describing bottom right corner.

Indexed data	None

System objects

tempRect	A scratch rectangle.
fpRect	Used for scrolling within fWind.

Object

Methods

accessing

get: (-- l t r b)

Returns the values in the Rect's two Points in (X Y X Y) format. The first pair is the topLeft Point, and the pair on top of the stack is the bottomRight Point.

getTop: (-- x y)

topInt: (-- x:y)

Returns the point TopL with x and y packed into a longword.

getTopX: (-- x)

getTopY: (-- y)

getBot: (-- x y)

botInt: (-- x:y)

getBotX: (-- x)

getBotY: (-- y)

put: (l t r b --)

Stores new coordinates in the Rectangle.

putTop: (x y --)

putTopX: (x --)

putTopY: (y --)

putBot: (x y --)

putBotX: (x --)

putBotY: (y --)

size: (-- w h)

Returns the size of the rectangle in pixels as width and height.

setSize: (w h --)

Sets the width and height of the rectangle according to w and h, by setting the BotR point.

getCenter: (-- x y)

Returns the center point of the rectangle in the local coordinates of the current GrafPort.

inset: (dx dy --)

Causes the rectangle to be grown or shrunk around the center point. The left and right sides are moved in by the amount specified by dx; the top and bottom are moved towards the center by the amount specified by dy. If dh or dv is negative, the appropriate pair of sides is moved outwards instead of inwards. Does not redraw the Rect. If either the width or height would become negative, the rectangle is reset to the empty rectangle (0 0 0 0).

shift: (dx dy --)

Moves the rectangle the given distance. Does not redraw the Rect.

offset: (dx dy --)

A synonym for shift:.

stretch: (dx dy --)

Shifts the BotR point by the given distance.

->: (^rect --)

Copies the Rect whose address is passed in, to this Rect.

drawing

draw: (--)

Draws the rectangle's frame.

dropShadow: (--)

As for draw:, but adds a standard Mac drop shadow.

disp: (l t r b --)

Combines the actions of put: and draw:.

clear: (--)

Fills the rectangle's frame with the current GrafPort's background pattern.

paint: (--)

Fills the rectangle's frame with the current GrafPort's foreground pattern.

fill: (pattern --)

Given a pointer to a QuickDraw Pattern, fills the Rect's frame with the Pattern. For example, 3 sysPat fill: myRect would fill the rectangle with the system pattern #3.

invert: (--)

Inverts the pixels bounded by the rectangle.

clip: (--)

Clips all subsequent drawing to the area bounded by the rectangle.

update: (--)

Causes an update event to occur for the current window that will redraw everything inside the rectangle.

Error messages

None

GrafPort

This class maps the record structure for a GrafPort. A QuickDraw GrafPort defines a local drawing environment with its own coordinate system and pen characteristics, and provides the basic functionality necessary for windowing. A Window record incorporates a GrafPort as the first part of its data.

Superclass	Object
Source file	QD
Status	Core
Instance variables	

16 bytes	graf1	Allocates the first 16 bytes of the GrafPort.
rect	PortRect	The port rectangle defining the GrafPort's limits.
44 bytes	graf2	Allocates the last 84 bytes of the GrafPort.
var	Text1	font, face, mode, size
var	Text2	
32 bytes	graf3	

Indexed data	None
System objects	None

Object

Methods

accessing

getRect: (-- l t r b)

Returns the coordinates of the GrafPort's PortRect.

putRect: (l t r b --)

Sets the coordinates of the GrafPort's PortRect.

drawing

set: (--)

Causes this to be the current GrafPort for subsequent drawing.

Error messages

None

Chapter 9—Dialogs

About This Chapter

This chapter describes the Mops classes and words that manage dialogs and alerts for the application. Dialogs are special-purpose windows that allow the user to modify parameters that affect the operation of your program. Alerts tell the user of exception conditions, and verify that certain critical operations are appropriate before executing them.

Recommended Reading

IM - Window Manager
IM - Dialog Manager
IM - Control Manager
Mops - Windows
Mops - Controls

Source Files

Alert
Dialog
DialogMod.txt
Dialog+

Using Dialogs

Class Dialog is, like Menu, a subclass of X-Array. Dialogs are similar to menus in that they have items associated with them that can be selected by the user. Dialog items can be controls, static (non-editable) text, or editable text. A dialog item is selected by pressing the mouse button in the region assigned to the item; if an item is enabled, it will cause the Dialog Manager to return immediately to the caller. Text-edit items do not return immediately, but wait for keyboard input to be terminated by a carriage return.

Macintosh dialogs can be modal or modeless. Mops's basic Dialog class supports modal dialogs, which do not allow the user to activate a different window while the dialog is active. Subclass Dialog+ adds the necessary support for modeless dialogs, which allow the user to activate a different window, select a menu or do anything else while they are active.

Note that Apple recommend that you use modeless dialogs by preference, since they are more user-friendly. They recommend you reserve modal dialogs for unusual situations where a user response must be obtained before the program can proceed (although in this situation an Alert box may well be more appropriate).

Dialogs are heavily dependent upon resource definitions for the window and the item list. You can use ResEdit to create resources of type DLOG and DITL, and then store the resource ID in a Mops object of class Dialog (see IM - Dialog Manager). The items in a dialog's item list are not defined as control objects in Mops, because they can more easily be accessed via methods in class Dialog itself. A Dialog object's indexed cells are loaded with the xts of Mops words, one per item. For example, the handler for a Control of type CheckBox might be to get: the value of the control, XOR it with itself, and put: the result back in the control. This would result in toggling the value of the control between 0 and 1, turning the check mark off and on. The value of the checkBox could be used to set a program parameter when the user leaves the dialog.

For modal dialogs, once getNew: has been sent to fire up the dialog, you should send a modal: method to the dialog. This calls the Toolbox routine ModalDialog, which initiates the Dialog Manager's event loop, which handles events until the user clicks on a dialog item. Then, your handler for that item will be executed. Each

handler must end in one of two ways. If the item chosen indicates that the user wants to leave the dialog, and the data provided by the user is

acceptable, the handler should send a close: message to its dialog, which will make the window disappear and release the heap storage for the Toolbox data. Otherwise, the handler should use the word ReturnToModal before it returns. This will signal the modal: method, which is still active, to loop and call ModalDialog again. Otherwise the modal: method will return to its caller.

Modeless dialogs don't require modal:, of course. Before you send getNew: to a modeless dialog to fire it up, you should execute the word +MODELESS, which modifies Mops' event handling so that it recognizes events relating to modeless dialogs. When you are finished with all your modeless dialogs, you may execute -MODELESS to restore the original event handling, but there is really no need to do so as the additional overhead is minimal.

Item data can be accessed via get: and put: messages to the dialog, which assume that the item is a control; for text-edit items, use getText: and putText:. SetSelect: will set the selection range for a text-edit item. If you need to perform operations not provided in class Dialog, you can either get the handle for the item via the Handle: method, and call the Toolbox directly, or you can define a subclass of Dialog that has additional item methods.

Alerts are very similar to Dialogs in implementation, with the exception that loading the Alert and entering the modal event loop are accomplished in a single operation via the show: message. You must have a resource definition of type ALRT in your resource file with an associated item list. The init: method of class Alert accepts both a resource ID and a type code; the type determines which icon will be printed in the upper-left corner of the alert:

Type =	0	Caution Alert
	1	Note Alert
	2	Stop Alert
	>2	no icon

Because it is loaded and executed in a single operation, you cannot modify the item list for an Alert in memory. Therefore, you should create item lists containing the exact information for each alert in your application. You could, of course, also define a subclass of Alert that separates the loading and event-handling operations.

Also provided is Alert" which gives you a predefined alert box without having to define any resources. You can use Alert" in place of Abort" in your application to better comply with the Mac user interface guidelines. However, Alert" is still rather "quick and dirty"; it is best of all to create custom alert boxes.

Dialog

Dialog implements a modal dialog object by associating the Toolbox dialog data with an X-Array containing the dialog's item handlers.

Superclass	X-Array
Source file	Dialog, DialogMod.txt
Status	Optional
Instance variables	

int	resID	The resource ID of a DLOG resource.
ptr	dlgPtr	Pointer to the dialog's non-relocatable heap.
var	procPtr	Pointer to the PROC definition for ModalDialog.
int	boldItem	item# to be drawn with a bold outline.

Indexed data	4-byte cells (must be xts)
System objects	None

X-Array, Array, Indexed-Obj, Object

Methods

object creation

init: (<xt list> resID --)

Sets the dialog's item handlers and resource ID of the dialog's DLOG resource.

setBold: (item# --)

Sets the item to be boldly outlined when the dialog is drawn.

setProc: (xt --)

Sets the filterProc to be used by modal. This must be a :PROC definition.

putResID: (resID --)

Sets the resource ID for the dialog.

getNew: (--)

Loads the resource template for the dialog and displays its window as the frontmost window. No key event can be processed after getNew: is issued and before modal: is issued. (This means a getNew:, modal: sequence must be typed on one line if executed from the Mops prompt.)

show: (--)

If the dialog resource is set to be initially invisible, you will want to show: it when all the items are drawn. It makes for a nicer build creation appearance.

hide: (--)

Makes the dialog invisible.

modal: (--)

Enters the Dialog Manager's modal event handler, which will beep whenever the user tries to click the mouse outside of the dialog window. If a click occurs inside an enabled item, the dialog will execute the handler corresponding to that item, which should either send a close: to close the dialog or use ReturnToModal.

close: (--)

Closes the dialog window and disposes of the associated heap.

accessing items

getItem: (item# -- val)

Gets the value of a control item (numbered from 1 to N).

putItem: (val item# --)

Sets the value of a control item.

getText: (item# -- addr len)

Returns the text string for an item.

putText: (addr len item# --)

Sets the text string for an item.

setSelect: (start end item# --)

Sets the selection range for an editable text item.

dlgPtr: (-- ^dlg)

Returns the pointer to the dialog (after getNew: has been called).

itemHandle: (item# -- hndl)

Returns the handle to the given item. TempRect is also set to the item's rectangle.

open?: (-- b)

Returns true if the dialog is open.

drawing

draw: (--)

Forces drawing of the dialog without (or before) executing modal:. Useful in non-modal (or pre-modal) situations.

set: (--)

Same as set: for class Window. Any text will be output to the dialog object.

drawBold: (--)

Draw the frame around the boldened OK button. Normally called automatically—from getNew:, and also, if this is a modeless dialog, in response to an update event.

setUserProc: (^proc item# --)

Sets the :proc routine for a user item.

manipulating individual items

hideItem: (item# --)

Hides the item.

showItem: (item# --)

Shows the item.

disableItem: (item# --)

Disables the item. It will be dimmed and not respond to clicks.

enableItem: (item# --)

Enables the item.

hitBold: (--)

Acts as if the bold item has been clicked, and executes the corresponding item handler. Call this if Return or Enter is typed while the dialog is active. Does nothing if there isn't a bold item.

key: (-- b)

Called when a key down event occurs with this dialog's window active. Returns false if we've handled the key here, so no further action is required. Subclasses can have customized key: methods; here we just provide a hopefully sensible default action—namely, we treat a Return or Enter as a click on the bold item, and ignore all other keys.

manipulating the dialog's window

title: (addr len --)

Sets the title of the dialog.

maxX: (-- x)

Sends maxX: to the dialog's window.

maxY: (-- y)

move: (x y --)

Moves the dialog so that its top left corner is at the given coordinates.

center: (???)

Centers the dialog on the screen, by sending center: to the dialog's

window.

select: (--)

Makes this dialog the frontmost window.

global parameters

ParamText (addr0 len0 addr1 len1 addr2 len2 addr3 len3 --)

This is a word, not a method. Calls the Dialog Manager routine ParamText which defines dialog text substitution strings. All subsequent static or editable text in dialogs will automatically substitute strings 0 through 3 for occurrences of “^0”, “^1”, “^2”, and “^3”, respectively. NOTE: the combined length of the four strings must be less than or equal to 252 bytes.

theItem, itemHandle, itemType

objects (see source file Dialog) holding parameters for selected items.

Error messages

None

Dialog+

Dialog+ adds support for modeless dialogs and pop-up menu dialog items.

Superclass	Dialog
Source file	Dialog+
Status	Optional
Instance variables	
ptr	F-link Forward link in chain of dialogs
ptr	B-link Backward link ditto
bool	enabled?
ptr	PUM-link Link to any pop-up menus
Indexed data	4-byte cells (must be xts)
System objects	None

Dialog, X-Array, Array, Indexed-Obj, Object

Methods

object creation

getNew: (--)

Loads the resource template for the dialog and displays its window as the frontmost window. It will be a modeless dialog. Provided +MODELESS has been executed, all dialog events such as updates or clicks on the dialog's items will be handled automatically.

manipulation

close: (--)

Closes the dialog and releases its storage.

disable: (--)

Disables the dialog. It will still respond to update events, but not to clicks.

enable: (--)

Enables the dialog. It will respond to all events as normal.

exec: (item# --)

If the dialog is enabled, executes the handler for the given item. Otherwise does nothing.

enabled?: (-- b)

Returns true if the dialog is enabled.

Error messages

None

Alert

Alert loads and displays alerts defined in a resource file.

Superclass	X-Array
Source file	Alert
Status	Optional
Instance variables	

Int	resID	Resource ID of an ALRT resource.
Int	type	Icon type to display (see above).

Indexed data	4-byte cells (must be xts)
System objects	None

X-Array, Array, Indexed-Obj, Object

Methods

object creation

init: (resID type --)

Sets the icon type and resource ID of the Alert.

show: (--)

Loads the Alert from a resource file, and enters the Dialog Manager's modal event loop. Item handlers are executed as in Dialog, above.

disp: (resId type --)

Combines the actions of init: and show:.

Error messages

None

Chapter 10—Floating Point

About This Chapter

This chapter describes the Mops classes and words that manage floating point. A floating point number is a 12 byte data value which is stored in a block in the heap and is accessed through its pointer.

Recommended Reading

SANE manual

Source Files

Floatint point

Using Floating Point

Floating point (FP) operations are implemented in different ways on different Macs. Some have a coprocessor, in some cases FP operations are available on the main processor, and some Macs have no hardware FP at all, and must use the software package Standard Apple Numerics Environment, (SANE). FP computations are handled by Mops in the appropriate way for the machine Mops is running on. Many but not all FP functions are implemented as methods of class Float. Other functions are available as Mops words, (e.g., ln1, exp1, x**y, compound, annuity, etc…). See the Glossary for these words.

To access the Mops FP facilities, you must load the file “Floating point” and save the dictionary. Once “Floating point” is loaded, Mops uses a slightly modified interpreter which converts numbers with decimal points to floating point format.

It is important to note that Mops inherits certain behaviors from the Forth language concerning decimal points when running the standard integer interpreter. Particularly, the decimal point is ignored except that DPL marks its position during input. In the Mops FP interpreter the occurrence of a decimal point causes conversion to a 12 byte FP number (a “float") which is stored in a block of storage in the floating heap. The application retains a pointer to the storage and is responsible to use the prescribed operations so as not to drop the last copy of a pointer before freeing the float, nor to free the float while still using a working copy of the pointer.

So, a float is created when a decimal point is used on input:

```
1.2    \ this creates the float and puts its pointer on the stack
```

You can use the normal stack operations to alter the position of a float on the stack:

```
1.2  3.4      \ creates two floats
swap e. e.    \ swaps and prints them, (deallocating their blocks)
```

To dup or drop a float, however, you must use special operators that take the float heap into account:

```
1.2  3.4 fdup e.    \ creates 2 floats, dups the top one and prints it
```

There might be special cases in which you want to use regular DUP to make a copy of the pointer, which is faster than FDUP. There are two dangers to avoid: 1) leaving yourself with a pointer to a deallocated heap block, or 2) dropping all copies of a pointer before the corresponding block is deallocated. The following example is a safe use of DUP:

```
1.22  dup .h fdrop  \ creates a float, prints its pointer,
                    \ then drops it
```

The following are examples of floating arithmetic:

```
10.3  3.0  f*  e.          \ multiply two floats and print the product
3.4    5.5  f+  e.         \ add two floats and print the sum
1.0 sin  e.                \ compute the sin of the float and print the
                           \ result
```

fValue is the analog to Value for floating point numbers:

```
.211 fValue  stuart        \ create a float value with an initial
                                  \  value
stuart e.                         \ print the value
1.234 -> stuart  stuart e. \ store a new value and print it
2.2 ++> stuart  stuart e.  \ increment the value and print it
```

Class Float creates an object with a variety of computational methods:

```
Float fred                        \ create a float object
print: fred                       \ print its value
1.34  put: fred  print: fred      \ change its value and print it
3.4 +: fred  print: fred   \ increment the value and print it
sin: fred  e.              \ find the sine; doesn't change fred's
                                  \  value
```

Class fArray creates a floating point array:

```
3 fArray  harold                  \ create a 3 element float array
3.9 1  to: harold                 \ set 1st element to 3.9
2.145 2  to: harold        \ set 2nd element to 2.145
print: harold              \ print all elements of harold
2 at: harold  e.                  \ print 2nd element of harold
```

You can define a floating point local variable by preceding the name with '%'. The usual operations are used:

```
: tst { int1 %flt1 -- }    \ declare 1 integer and 1 float parameter
int1 .  %flt1 e.           \ print both values, each in their own format
3.445 ++> %flt1  %flt1 e.  \ increment the float and print it
1.22 -> %flt1  %flt1 e. ;  \ change the value of the float and
                                  \  print it
```

Conversion between Mops integers and floats are accomplished as follows:

```
123 >float  e.      \ convert  the integer to a float and print it
3.545 float> .      \ convert the float to an integer and print it
```

It should not be necessary to create objects of class FltHeap. Mops provides fltMem which will accommodate up to 100 Float numbers. In certain cases you may want to use method new:, which returns a pointer to a new float block in the heap, and dispose:, which frees a float block, although the printing words and methods automatically dispose of float heap blocks. Also, room: returns the number of free float blocks remaining in the float heap.

Here are some more technical details. You don't need to read this section to use floating point, but there are some details here which can help you to get the best performance out of your FP code.

The 12-byte FP format is the 96-bit "extended" format defined for the harware floatint point unit (FPU)—the MC68881 or MC68882, or the integrated FPU on the 68040 chip. SANE defines an 80-bit extended format which is basically the same, except that the FPU carries 16 unused bits (for quad-word alignment) which are not present in the SANE format. Mops uses the 96-bit format in order to make it easer to call the FPU directly, which Mops does if there is an FPU installed. The value FPU? is set non-zero by the startup code if an FPU is present, and this value is tested "on the fly" by the most common FP words. If the FPU is present, these FP words use it directly without calling SANE, which saves a lot of overhead. If the FPU isn't present, of course, SANE is called.

This method produces code which gives reasonably good performance, and will run on any Mac. For example it can be compiled on a Mac with an FPU, and run on one without. If, however, you know your code will always be running on a Mac with an FPU, you can set the value UseFPU? to true, and Mops will compile FPU instructions in line. Also, up to 6 FP parms/locals will be kept in the FPU's registers (which more than doubles the access speed), and some optimization is done on the code. This produces some very fast code indeed. To get the best results from this feature, it is best to use FP locals for temporary values, rather than the stack, which requires data to be moved over to the integer unit and back again.

The forthcoming PowerPC Macs have their own FP instructions, handled within the chip. Until Mops can compile native PPC code, it will have to run under emulation. However, the emulator does NOT emulate an MC68881/2. This means that for now, if you're running FP operations in Mops on a PowerPC, you'll have to use SANE. This should happen automatically, since FPU? will always be set to false on a PowerPC.

But it should now be clear that on a PowerPC, you should never set UseFPU? to true. This would cause Mops to generate 68881/2 instructions, which would be invalid. Later, when Mops can compile native PowerPC code, it will use the native PPC FP instructions, because they will always be present. In this case it will ignore UseFPU?. So the simple rule for PowerPC compatibility is, leave UseFPU? alone!

The file "FP test" in the "Float source" folder has a timing test and the results I obtained on my IIsi. To summarize briefly, the test has two floating adds, two multiplies and a divide. With UseFPU? false, that is compiling code which will run on all Macs, changing the FP words to use the FPU directly made the loop run about three times faster than when they had called SANE. Setting UseFPU? true made it run about 14 times faster than originally. The code ran at about 50% of the speed of the best I could do in hand-coded assembly. I doubt that many other languages for the Mac can do as well as this.

A note about accuracy. Apple have warned (Mac Tech Note # 146) that SANE is more accurate than the FPU in computing the functions Sin, Cos, ArcTan, Exp, Ln, Tan, Exp1, Exp2, Ln1 and Log2. Currently, in Mops, we use the FPU, if present, for Sin, Cos, Tan and ArcTan. We may extend use of the FPU to the other functions in future. I think the loss in accuracy is in the last 7 bits of the 64-bit mantissa. If you can live with this, the code will certainly be much faster than a SANE call. If you need the extra accuracy, you can either (i) call SANE yourself or (ii) before calling the Mops word for the function, put false -> FPU? in your code. This will tell Mops there is no FPU, and fool it into calling SANE. Remember to leave UseFPU? false, as otherwise inline FPU code will be generated that never looks at FPU?.

Finally, heed Apple's warning, and don't assume the machine you're running on will always have an FPU if it is a 68020 or better. The IIsi has a 68030 but the FPU is optional. If in doubt, your code should check the FPU? flag which we provide in Mops. It's always best to play safe.

fArray

fArray provides an array construct for floating point numbers.

Superclass	Indexed-obj
Source file	Floating point
Status	Optional
Instance variables	None
Indexed data	12-byte cells
System objects	None
Indexed-Obj, Object	

Methods

accessing

^elem: (index -- addr)

Returns the address of the indexed cell.

at: (index -- fval)

Returns the data at the given indexed cell.

to: (fval index --)

Stores data at a given indexed cell.

fill: (fval --)

Stores fval in each cell of the array.

print: (--)

Prints the elements of the array with their respective index number.

classinit: (--)

Stores the value “undefined” in each cell.

Error messages

None

FltHeap

This class defines the floating heap.

Superclass	Object
Source file	Floating point
Status	Optional
Instance variables	
int	FreeHead Offset of first free block
Indexed data	14-byte cells
System objects	
fltMem	The system supplied float heap provides 100 blocks.

Object

Methods

object creation

init: (--)

Sets all blocks to free and links them together.

new: (-- fPtr)

Returns a pointer to a new block.

release: (fPtr --)

Frees the block pointed at by fPtr.

room: (-- #free)

Returns the number of free float blocks remaining in float heap.

Error messages

None