

Part III

Predefined Classes

Table of Contents

1. Basic Data Structures	1-1
Object	1-5
Int	1-7
Var	1-8
Handle	1-10
basic Array methods - bArray,wArray,Array	1-12
Array	1-13
X-Array	1-14
Ordered-Col	1-15
Dictionary	1-17
LinkedList	1-19
Data Structure-Related Yerk Words	1-20
2. Strings	2-1
BasicStr	2-4
String	2-6
sArray	2-8
String-Related Yerk Words	2-9
3. Files	3-1
File	3-5
FileList	3-9
PathList	3-10
Fin	3-11
File-Related Yerk Words	3-12
4. Events	4-1
Event	4-5
Mouse	4-7
Timer	4-8
Event-Related Yerk Words	4-9
5. Windows	5-1

Window 5-4
CtlWind 5-9
Window-Related Yerk Words 5-9

6. Menus	6-1
Menu	6-4
AppleMenu	6-6
hMenu	6-7
pMenu	6-8
MBar	6-10
Menu-Related Yerk Words	6-11
7. Controls	7-1
Control	7-5
VScroll	7-8
Control-Related Yerk Words	7-9
8. Graphics	8-1
Point	8-3
Rect	8-5
RndRect	8-8
Oval	8-9
GrafPort	8-10
QDBitMap	8-11
Image	8-12
Picture	8-14
Icon	8-16
Graphics-Related Yerk Words	8-16
9. Dialogs	9-1
Dialog	9-4
UserItem	9-7
RadioSet	9-8
Alert	9-9
Dialog-Related Yerk Words	9-9
10. Drivers	10-1
PBDrvr	10-3
Port	10-5
Driver-Related Yerk Words	10-7
11. Floating Point	11-1
Float	11-4
fArray	11-6
FltHeap	11-7
Floating Point-Related Yerk Words	11-8

Chapter 1

Basic Data Structures

About This Chapter

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

Recommended Reading

IM - Memory Manager

IM - Programming in Assembly Language

Source Files

Object

Struct

Using the Basic Data Structures

This chapter will discuss the primitive classes that Yerk 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

Bytes

Int

Var

Handle

bArray

wArray

Array

X-Array

Ordered-col

WordCol

ByteCol

Dictionary

LinkedList

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: and Abs: return the relative and absolute base address of an object. In the case of a public object, this is usually not necessary because an

object returns its address when executed; Ivars, however, have no such behavior, and these methods are particularly useful when you need the address of an instance variable for a Toolbox call. Other methods in Object provide a hex dump of an object's data, access to an object's class pointer, and some general methods for indexed classes.

Bytes is not really a class, but rather is a Yerk 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 Yerk doesn't need direct access to. Remember, however, that Bytes is not an indexed type like barray -- it creates a single field.

Using the Scalar Classes

Classes Int, Var and Handle are called Scalar classes because they represent non-indexed objects that hold simple integer or pointer data. An Int can hold 16-bit, signed or unsigned integers. Since 68000 word fetches must be even-aligned, Ints are also used in Yerk for holding boolean values even though they theoretically could be stored in bytes. A Var can hold 32-bit signed values, and is also used to hold object addresses, heap block pointers, and cfas of Yerk words. A Handle is a subclass of Var that has methods added for allocating, sizing, locking and releasing relocatable blocks of 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 Yerk - 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).

You can find the address of an indexed cell of any width with the ^Elem: (pointer-to-element) method. This method is defined in class Object in a way that is independent of the actual width of the class's cells, because the width is looked up at runtime. You can inherit ^Elem: for use with any indexed class without redefining it because of this generality. Other methods in Object that are general enough to work with any indexed object are: 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; Clear:, which sets all of an array's cells to 0; and IXAddr:, which leaves the address of the 0th indexed cell. Yerk will issue a "Class is not indexed" error if you attempt to use any of the indexed methods on a non-indexed object.

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; Fill:, which fills an array with a value; Get:, which fetches all of an array's elements and places them on the stack (be careful with large arrays!), and Put:, which stores from the stack to each of an array's cells. This

group is shared by the three array classes that are predefined in Yerk, and is documented later in this section.

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. Arrays can be used to hold arrays of objects that you wish to refer to by index rather than by name. For instance, consider an application in which you need four windows accessible by index:

4 Array Windows

```
: fillWindows 4 0
  DO heap> window 1 to: windows
  LOOP ;

: killWindows 4 0
  DO 1 dispose: windows
  LOOP ;

\ resize window at index 2
300 100 size: [ 2 at: windows ]
```

In this example, the array `Windows` is used to hold pointers to objects of class `Window`. In `fillWindows`, the `heap>` prefix causes class `Window` to allocate a headerless object on the heap, leaving its address on the stack. This is then stored in a cell of the array `Windows`. You can then send late-bound messages to one of the windows by index, as in the `Size:` message at the bottom. Actually, the objects in `Windows` could be of any class that accepted a `Size:` message, due to the late binding. The word `killWindows` uses `Array's Dispose:` method to release the heap occupied by each of the window objects. You can exploit any kind of indexed structure - arrays, sequential and linked lists, to hold pointers to objects. As you become more experienced with programming in Yerk, you will find that this approach combined with late binding is one of the most powerful aspects of the language.

Class `X-Array` adds to the basic `Array` the ability to execute one of its indexed cells, assuming that it holds the cfa of a Yerk word. `X-Array` is a very important class in Yerk, because its behavior is used throughout the system itself to provide control dispatching by index, as in `Menu`, `Event` and `VScroll`. 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 Yerk words based on a series of contiguous indices.

Sequential and Linked Lists

Class `Ordered-Col` is another important class in Yerk. It adds to `Array` and `X-Array`, which comprise its inheritance path, 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.

As an example, Yerk uses an `Ordered-Col` to hold its symbol table of `Toolbox` call names. The

table is built by reading each name from a text file, generating a unique hash value for the name, and adding it to the end of the list. This is done at compile time, when the Tool module is being built. At runtime, the `indexOf:` method is used to search for a particular call, and the resulting index is used to generate the correct trap value. Ordered-Col can function in this way as a kind of primitive dictionary that uses sequential searches for its lookup. You

could define a subclass of O-C that has more sophisticated searching capabilities if you needed to implement a dictionary with superior performance.

Dictionary

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 a value. Dictionary is a subclass of Array, 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:

codePos token enter: symTab

(Note: token is an instance of class String) If it is in use, the routine will test to make certain that it is not already defined. If not, it enters it into the symbol table:

-1 token enter: symTab

At the completion of the first pass the symbol table is checked to see if it has any undefined labels, with the following code sequence:

```
: check-table { offset addr len -- }  
    offset -1 =  
    IF  
        print: token ." is an unresolved reference " cr  
    THEN ;
```

Data Structure Classes

class description: Object

class	Object
superclass	Meta
source file	Object
status	Core

description

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

instance variables

None

indexed data None

methods

non-indexed properties

addr: { -- addr }

Returns the relative base address of an object's data by copying it from the top of the methods stack to the parameter stack.

abs: { -- addr }

Returns the absolute base address of an object's data area. Usually used as an input to a Toolbox call.

class: { -- addr }

Returns a pointer to an object's class.

classinit:

This is a very special method -- whenever an object is created, Yerk 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. 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).

length: { -- #bytes }

Returns the length of an object's data area.

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.

indexed properties

^elem: { index -- addr }

Returns the relative address for the element at **index**.

limit: { index -- 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.

clear: { -- }

Sets each indexed element to 0.

system objects None

error messages "You can't use indexed methods on this class"

You tried to use an indexed method on a non-indexed class.

class description: Int

class Int
superclass Object
source file Struct
status Core

description

Int provides storage for signed or unsigned 16-bit quantities.

instance variables

2 Bytes data Room for 16 bits of data.

indexed data None

methods

accessing

get: { -- val }

Returns the value in the data area as a signed number. If bit 15 is on, this bit will be extended into the high-order word of the stack cell.

uget: { -- uval }

Returns the value in the data area as an unsigned number. The high-order word of the stack cell is always 0.

put: { val -- }

Stores a new value in the data area.

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.

=: { addr -- }

Left-to-right assignment of the contents of the data area to **addr**.

+: { val -- }

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

printing

print: { -- }

Prints the data in the current base on the screen.

system objects None

error messages None

class description: Var

class Var
superclass Object
source file Struct
status Core

description

Var provides storage for 32-bit quantities and pointers.

instance variables

4 Bytes data Allocates 32 bits of data.

indexed data None

methods**accessing**

get: { -- val }

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

obj: { -- addr }

A synonym for **get:** when the contents of the Var is an object pointer. Using the selector **obj:** serves to document this fact more clearly.

put: { val -- }

Stores a new value in the data area.

clear: { -- }

Stores 0 in the data area.

=: { addr -- }

Left-to-right assignment of the contents of the data area to **addr**.

+: { val -- }

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

dispose: { -- }

Assumes that the contents of the Var is a pointer to a block of non-relocatable heap. Calls DISPOSE to release the heap block, and stores 0 in the Var.

exec: { -- }

Assumes that the contents of the Var is the cfa of a Yerk word, and executes it.

printing

print: { -- }

Prints the data in the current base on the screen.

system objects None

error messages "I can't respond to **exec:** when my contents are 0"
An **exec:** was attempted on a Var holding 0.

class description: Handle

class Handle
superclass Var
source file Struct
status Core

description

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

instance variables

None (see Var)

indexed data

4-byte cells

methods**accessing**

ptr: { -- relPtr }

Returns a dereferenced, relative pointer from the handle.

release: { -- }

Releases the heap block pointed to by the handle and zeros the data.

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 the object's data.

lock: { -- }

Locks the block corresponding to the handle. This automatically performs a MoveHi prior to locking.

unlock: { -- }

Unlocks the block corresponding to the handle.

locked?: { -

bit version. All methods are identical to Ordered-Col except that exec: is not defined, since their classes do not inherit from X-Array.

class description: Dictionary (not included in Yerk.com or YerkFP.com)

class Dictionary
superclass Array
source file Dictionary
status Optional

description

Dictionary provides a hash-table entry and lookup data structure. It can be used to assign values to predefined string objects. Dictionary allows different strings to have the same value. String objects must be initialized by the **new:** method of the string class prior to their entry into the dictionary.

instance variables

None

indexed data 4-byte cells

methods**accessing**

enter: { val str -- }

Enters the value to be associated with the string **str**. If a value had been previously assigned to **str** it is changed to the new **val**.

query: { str -- val }

Retrieves the value associated with **str**. If no entry of **str** can be found in the dictionary a value of zero is returned.

manipulating

exec: { cfa -- }

Executes the yerk word whose **cfa** is on the stack for every element in the dictionary.

cleanup

dispose: { -- }

Disposes of the entire dictionary.

system objects None
error messages None

Examples:

```
0-> string test
0-> new: test
0-> string test2
0-> new: test2
0-> " hello" put: test
0-> " bye" put: test2
0-> 10 Dictionary Webster
0-> 456 test2 enter: Webster
0-> 123 test enter: Webster
0-> test2 query: Webster
1-> .
456 0-> : print type . ;
0-> 'c print exec: Webster
0-> bye 456 hello 123 0->
```

class description: LinkedList (not included in Yerk.com or YerkFP.com)

class LinkedList
superclass Object
source file LinkedList
status Optional

description

LinkedList provides an organized storage technique for ordered data.

instance variables

Int	front	index number of first element
Int	back	index number of last element
Int	current	current position of index
Int	size	# elements currently allocated to list

indexed data 4-byte cells

methods**accessing**

getCurrent: { -- idx }

Returns the current position index **idx**.

setCurrent: { idx -- }

Sets the current position index **idx**.

size: { -- size }

Returns the number of elements allocated to the list.

getData: { -- data }

Returns the data from the current position in the list.

setData: { data -- }

Assigns the data to the current position in the list.

front: { -- data }

Returns the data from the first position in the list and updates **idx**.

prev: { -- data }

Returns the data from the previous position before **idx** in the list and updates **idx**.

next: { -- data }

Returns the data from the next position after **idx** in the list and updates **idx**.

before: { data -- }

Inserts the data into the list in front of the current **idx** position. This increases the size of the list by one.

after: { data -- }

Inserts the data into the list after the current **idx** position. This increases the size of the list by one.

delete: { -- }

Deletes the current element from the list and makes it free. **idx** is updated to point to the next element. This decreases the size of the list by one.

display

print: { -- }

Lists the contents of all elements in the list, printing the **idx** number and the data in each corresponding element.

system objects	fEvent	Description of fEvent.
error messages	"My list is full"	Description of error message.

Data Structure-Related Yerk Words

@	!	W@	W!
C@	C!	+!	W+!
m@	m!	mw@	mw!
i->l	extend	bytes	limit
idxBase	(^elem)	at1	at4
to1	to4	?idx	?lxObj
++4	+range	-range	?range
copyM	+base	-base	newPtr
newHandle	killPtr	killHandle	growPtr
setHSize	getHSize	dispose	

2

Strings

About This Chapter

This chapter describes Yerk's string-handling classes. Strings are objects that contain variable-length sequences of text, with methods for printing, addition, deletion, insertion and pattern matching. Yerk's powerful string-handling facility provides an excellent base on which you can build parsers, natural language processors, and other text-based utilities.

Recommended Reading

IM - Toolbox Utilities

IM - OS Utilities

Yerk - II.4, "Using Strings in Yerk"

Source Files

BasicStr

String

Using Strings

Yerk strings are implemented as relocatable blocks of heap that can expand and contract as their contents change. The string object itself contains only a handle to the heap block that contains the string's data, and a current offset Ivar that maintains a current position within the string for searches, insertions and other operations.

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 textual data structure whose contents are likely to change in the course of your program's execution. Text constants can more efficiently be implemented as SCONs, TCONs, 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 when your program begins executing 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 Yerk. Both (BasicStr and String) are compiled into the

distributed Yerk.com image. BasicStr supports basic string operations, such as Get:, Put: , Insert: and Add:. Class String adds more esoteric methods, such as pattern matching, finding substrings, and buffered file I/O. 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 Yerk string handling.

Strings have a current length, which is the same as the length of the relocatable block of heap containing the string's data; a current offset, stored in an instance variable in the object; and the textual data itself. The offset can be adjusted by `moveTo:`, and determines at what position many of the string operations will occur. For instance, if the offset is 2 and you send an `insert:` message to a string, the replacement string that you provide will be inserted before character 2 in the string (numbering starts from 0). The offset will be left pointing to the first byte past the inserted string. A string will not allow you to explicitly move the offset past the end of the string, because Munger would behave strangely if that were permitted to happen. Note - If you wish a direct interface to the Munger function, the `Replace:` method in `BasicStr` calls `Munger` with the target and replacement strings that you specify.

You can set the offset implicitly by sending an `indexOf:` message, which will cause the string to search for the target string that you provide and set the offset to the first character at which a match was found, or -1 if no match was found. You should not attempt to perform any further operations with an offset of -1, or an error will occur.

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 uses late binding to facilitate communication with certain other objects. For instance, the `=:` method simplifies assignment of a string to another object that accepts { `addr len --` } as its `Put:` parameters; the phrase:

```
fred =: sam
```

assigns the string Sam to whatever type of object Fred is. The method results in a late-bound `Put:` message being sent to the receiver, so the receiver can be of any class that accepts a `Put:` with that format. Another way to accomplish the same thing, that takes more space but executes more quickly because it is early-bound is `get: sam put: fred`. The method `name=:` can be used in a similar manner to assign a string as the name of an object such as a file or window:

```
file myFcb  
myFcb name=: sam
```

String also has two methods that simplify its use as a file buffer. `Read:` and `ReadLine:` both accept a File object and a length, and will request that the File perform a read into the string, setting the length of the string to the number of bytes actually read. For example:

```
myFcb 80 readLine: sam  
print: sam  
A rose by any other name. 0->
```

reads the next line from the File object `myFcb`, and then prints the line that was read. This single method takes the place of several operations that normally would have been required for buffered

file I/O, and has the additional advantage that the file data is left in a String object instead of a "dead" buffer, and is therefore subject to all of the various manipulations that Strings 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. This is a convenient interface to the TEEdit TextBox routine.

justification =0	Left justification
1	Center justification
-1	Right justification

An subclass of class String is sArray, very similar to class ordered-col, except that it is an array of str255 format strings, with dynamically allocated storage. With the exception that this type of array must be sent the **new:** method prior to use, it acts identically as class ordered-col.

String-Related Classes

class description: BasicStr

class	BasicStr
superclass	Handle
source file	BasicStr
status	Core

description

BasicStr 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.

instance variables

Var	offset	Current offset for string operations.
-----	--------	---------------------------------------

indexed data	None
--------------	------

methods

accessing

replace: { addr1 len1 addr2 len2 -- }

Calls the Toolbox Munger function with specified target and replacement strings. If both strings are non-0, the target string will be searched for, starting at the current offset, and will be replaced by the replacement string (addr2 len2). Other functions described in Munger documentation.

put: { addr len -- }

Clears the string, replacing it with passed-in string.

get: { -- addr len }

Returns the total string, starting with byte 0. You may need to lock the string down if you want to ensure the text to stay at that location in memory.

ptr: { -- addr }

Returns the address of the first byte in the string. Remember, the string might be moved by the Memory Manager, so be sure to lock the string down if you intend to heavily write directly to its memory locations.

handle: { -- handle }

Returns a handle to the string data.

size: { -- len }

Returns the current length of the string.

clear: { -- }

Sets length and offset of string to 0.

moveTo: { offset -- }

Sets the current offset, which cannot be past the end of the string.

next: { chr t OR f }

Returns false boolean if at end of string, character and true boolean otherwise.

insert: { addr len -- }

Inserts the passed-in string at the current offset. Offset is left one byte past the end of the inserted string.

add: { addr len -- }

String argument is appended to the end of existing string's data. Offset is set equal to the length of the new string.

+: { char -- }

Appends a single character to the existing string.

uc: { -- }

Converts the entire string to upper case.

object creation

new: { -- }

Allocates room for the string's data on the heap. **new:** must be done before the string can be used.

release: { -- }

Disposes of the string's storage. Use **release:** before disposing of the string object.

display

print: { -- }

Uses TYPE to print the string on the primary output device.

system objects

parmStr Used for input from inDlg

error messages

"The offset from the last operation was negative".

You tried to perform a string operation when the offset was left negative. Use **moveTo:** to change the offset.

class description: String

class	String
superclass	BasicStr
sou	

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. PathList gives you automated access to HFS volumes. Fin gives you access to documents passed in to your application by the Finder.

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

PathList

fInfo

Using Files

All file access in Yerk is done through an object of class File. For instance, when you request that a source file be loaded, Yerk 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, 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
" demo.txt" 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 over 200 bytes in length, it is useful to be able to allocate them dynamically rather than have them locked into a static dictionary. Class FileList provides this function by maintaining a "stack" of pointers to file objects. Yerk 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 New: LoadFile. The message Last: loadFile returns the address of the last File object allocated, which is the "top" of the file stack. You could send explicitly late-bound messages to the File object on the top of the file stack, such as:

```
pad 100 read: [ last: loadFile ]  
bytesRead: [ last: loadFile ]
```

However, because the message Last: loadFile is used so often, we have defined a Vect, topFile, that contains the cfa of a Yerk word that sends this message. Thus, you can use phrases like:

```
open: topFile  
myBuf 100 read: topFile
```

that exploit Yerk's ability to automatically late-bind messages that have a Value or Vect as the receiver, making it unnecessary for you to enclose the receiver in brackets each time.

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

```
remove: loadFile
```

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

The Clear: method in FileList closes and removes any currently allocated loadFiles, and is called by Yerk's default Abort routine. Both File and FileList have Interpret: methods, that allow you to direct the Yerk interpreter to open and interpret from a file rather than the keyboard. The method in fileList passes the message through to the file object on top of the file stack. Interpretation will echo loaded text to the screen if the system Value dEcho is true, and will end immediately if there is an error. There is also an Expect: method in File that simulates a Yerk EXPECT, only from disk; this is different from a readLine: in that each line is echoed conditionally on dEcho, and a null is appended to the end of the text in the buffer so that Interpretation of the line can be properly terminated (see Glossary entry for INTERPRET). The Query: method in File is simply an Expect: to the TIB.

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 SFGGetFile, 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 using standard editing techniques, 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. The source file FrontEnd contains sample Stdget: and Stdput: messages that you might want to examine.

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).

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

PathList and getPtxt give you automated access to HFS volumes. Under the original file system, MFS (Macintosh File System), all files on a volume are accessible at a single level. In other words, you do not need to know which folder a file lives in to access it. Under the newer Hierarchical File System, files are kept in folders which act like individual volumes themselves. To access a file you must know which folder it lives in before you can open it. (Folders can also be nested within other folders.) This means a file name must be prefixed by a potentially complicated path specification before it can be opened. An easy way to cope with accessing files stored in multiple folders is to organize a *pathList* which will be searched any time you attempt to open a file. Yerk provides a pathlist mechanism which is integrated into the Open: method of class File. To use it, create a file which lists the paths to be searched in the order you would like them searched. The format of the path text file is:

```
(nothing in first line)
::System source:
::Module source:
::Toolbox Classes:
::Yerk folder:
```

Each line specifies the exact string which will in turn be prepended to the unqualified filename in the FCB in an attempt to find the file on the disk. The first line with nothing in it, (not even a blank), specifies the default folder; the folder from which the application started up, (the folder in which the YERK nucleus resides until the application is "installed"), so it will be the first folder searched.

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. If the disk name is not known but the intended disk drive # is, you may substitute &1 or &2 for the disk name which will reference what ever disk is mounted in the specified drive.

To load a pathlist file, type:

"nPath.txt" getPtxt

This loads the list from the file name nPath.txt into the PathList object which is automatically used 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. From the viewpoint of your application code, the file system is simplified to that of the original flat directory, except that the scope is narrowed to a special set of folders.

Finder Information Block

Fin gives you access to documents passed in to your application from the Finder. The Finder allows you to highlight one or more documents on the desktop and then choose either OPEN or PRINT from its File menu. The Finder then launches the application to which the documents belong passing in information about each file through a special information block. Using an object of class Fin provides easy access to these files. For example:

file myFcb	\ file object
fin myFin	\ finfo object
size: myFin 0	\ loop for each file passed in by the Finder
DO	\ (do nothing if no files passed)
i myFcb =: myFin	\ acquire control information for each file
open: myFcb	\ open the file
...	\ operate on the file, then close it
LOOP	

The =: method here is similar to the Stdget: & Stdput: methods of class file in that it automatically sets up the file object ready to be opened. Remember to check whether it was OPEN or PRINT that was originally chosen from the Finder:

print?: myFin

File-Related Classes

class description: File

class	File
superclass	Object
source file	Files
status	Core

description

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.

instance variables

144 Bytes	FCB	The File Manager's parameter block
64 Bytes	fileName	Room for the name of the currently open file.

indexed data None

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: { -- fCode }

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

getName: { -- addr len }

Returns the name of the current file.

getVref: { -- volRefNum }

Returns the volume reference number for the current file.

getDir: { -- DirID }

Returns the folder directory ID for the current file. For HFS volumes.

getType: { -- fType }

Returns the file type of the current file.

getFileInfo: { -- fCode }

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

print: { -- }

Prints the name of the current 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. #types must be between 0-4.

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 current file. If the name is given with its fully qualified path while running under MFS, the path specification will be disregarded.

setName: { -- }

Sets the name of the current file from the input stream.

reName: { addr len -- fCode }

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 current file.

setVRef: { volRefNum -- }

Sets the volume reference number for the current file.

setDir: { DirID -- }

Sets the folder directory ID for the current file. For HFS volumes.

file operations

create: { -- fCode }

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: { -- fCode }

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

openReadOnly: { -- fCode }

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

new: { -- fCode }

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

read: { addr len -- fCode }

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

write: { addr len -- fCode }

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

readLine: { addr len -- fCode }

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

moveTo: { pos -- fCode }

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

last: { -- }

Positions to the byte following the file's EOF.

close: { -- fCode }

Closes the currently open file.

delete: { -- fCode }

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

drive: { drive# -- fCode }

Changes the default drive to **drive#**. The file object must not contain an open parameter block, or its information will be lost.

parameter block access

fcB: { -- addr }

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

clear: { -- }

Clears the parameter block for a new Open.

interpretation

expect: { addr len -- eof }

Performs a Yerk EXPECT to the address provided, reading successive lines from the currently open file. Eof is true if the line is the last line of the file. Lines are echoed to the screen if dEcho is True.

query: { -- eof }

Performs a Yerk EXPECT to the TIB, reading successive lines from the currently open file. Eof is true if the line is the last line of the file. Lines are echoed to the screen if dEcho is True. The system Value IN is set to 0.

interpret: { -- }

Opens and loads the source file named in FileName by repeatedly calling **query: self**. The load is terminated if an error or EOF occurs.

system objects	fFcb	Used by Yerk for system file access.
error messages	None	- return codes from File Manager

class description: FileList

class	FileList
superclass	Ordered-Col
source file	Files
status	Core

description

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

instance variables None (see Ordered-Col)

indexed data 4-byte cells containing pointers to File objects

methods**accessing****new: { -- }**

Allocates an object of class File on the heap and adds its pointer to the end of the list. Error if list is full.

init: { -- }

Initializes list (to zeros). No files are closed or objects removed.

interpret: { -- }

Uses the last file object added to the list as the file from which to load.

remove: { -- }

Closes the top file in the list, disposes of the file object on the heap and removes the pointer from the end of the list.

clear: { -- }

Closes and removes all file objects currently in the list.

system objects

loadFile

6-element FileList used

for nested loads.

error messages

"My list is empty"

A

remove:

was

received with an empty list.

class description: SArray

class	sArray
superclass	String
source file	PathList
status	Core

description

Sarray provides the FileList for systematic access to HFS volumes through an automatic search mechanism. There is only one object, path, and it is late bound.

see discussion in Chapter III.2 under Strings

system objects	path	Late binding to heap
	object.	
error messages	"The offset from the last operation was negative" You tried to perform a subStr: operation when the offset was left negative. Use moveTo: to change the offset. "An index was out of range" "String elements must be < 255 characters long"	

class description: Fin (not included in yerk.com or yerkFP.com)

class	Fin
superclass	Object
source file	fInfo
status	Optional

description

Fin provides access to the Finder Information Block.

instance variables None

indexed data None

methods

accessing

print?: { -- bool }

Returns a true flag if the application was started with the Finder Print option; (menu item under FILE).

size: { -- #files }

Returns the number of files with which this application was started.

fRec: { idx -- addr }

Returns the address of the information block for this file.

fRef: { idx -- volRefNum }

Returns the volume reference number for this file.

fType: { idx -- fType }

Returns the file type for this file.

fVer: { idx -- fVer }

Returns the version number for this file, (typically 0).

filename: { idx -- addr len }

Returns the name of this file.

=: { idx fileObj -- }

Fills in the **fileObj** with the filename and VolRefNum of this file.

system objects None

error messages None

File-Related Yerk Words

(open)	(close)
(delete)	(fdos)
cleanUp	Sony
HFS?	getPtxt

(read)
+echo
External
fInfo

(write)
-echo
Profile

4

Events

About This Chapter

This chapter describes the Yerk 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. Yerk 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
- Yerk - Windows
- Yerk - Menus
- Yerk - Controls

Source Files

- Event
- Interval
- Tasks

Using Events

Class Event is the core of Yerk's event management. It instantiates a single object, fEvent, which resides in the nucleus portion of Yerk's dictionary. FEvent is functionally an X-Array of 23 elements, each of which contains the cfa of a Yerk 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.

Macintosh Pascal 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. Yerk 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. Yerk 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 Yerk's event management to be completely general and open-ended, because the programmer can always build more specific event responses into Window and Control subclasses. Yerk's 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:

<u>Type Code</u>	<u>Description</u>
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	OS Events - associated with multifinder events
13-15	* user-definable events
16-21	* not defined
22	*High level AppleEvents

Events marked with a * are events for which Yerk 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 Yerk events. FEvent also contains 23 indexed cells, one corresponding to each of the event types described above. Each of these cells contains the cfa 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 end of the source file Window, called SYSINIT, initializes fEvent with the correct cfas whenever Yerk starts up. This is accomplished by setting the System Vector OBJINIT to execute SYSINIT (as well as whatever else you want your program to do at startup).

Listening to Events

The chief means by which you can cause Yerk to listen to the event queue is by calling the Yerk 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.

A more general way to handle events is to use `next: fevent`. This will return a true if a keystroke has occurred (see Section II.5 - Waiting for Input). Using `next: fevent` will allow your program to continue looping, processing other tasks while waiting for events; it does not hang while waiting for a keystroke.

As pointed out in chapter II.4, you might need to use the Yerk 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 Yerk 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 Yerk, requiring only that you specify the key equivalents in your menu item text definitions. If the `Command` key was not held down, Yerk 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 attempts to mount the newly inserted disk. If it cannot, it calls the `formatter`.

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.

The OS-Event handler determines whether the user has clicked the mouse to suspend the current Yerk program and activate another program. If so, it executes either a suspend or resume word supplied by the user in two vectors, *suspend* and *resume*. It also will convert the application's clipboard to the system clipboard if the user supplies the appropriate word

in the vector *cvtClip*. Also, a global value is set true if the yerk program is in the foreground.

There are 3 user defineable events (see Inside Mac).

High Level events are handled by user supplied words to be placed in the vectors *AppleEvt* and *HLEvt*.. Yerk does not handle these events, but is setup for a stand-alone application to do so.

Event-Related Classes

class description: Event

class Event
superclass X-Array
source file Event
status Core

description

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

instance variables

Int	evt	The named lvars comprise an eventRecord.
Var	msg	
Var	time	
Var	loc	
Int	mods	
Int	mask	

indexed data None

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.

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.

system objects

fEvent

The system-wide Event object.

error messages

See messages for class X-Array.

class description: Mouse

class	Mouse
superclass	Object
source file	Event
status	Core

description

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.

instance variables

Var	last	Ticks value when the last click occurred.
Var	interval	Ticks between this click and the last one.

indexed data None

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 Yerk point.

click: { -- b }

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

put: { ticks -- }

Update the click interval with the current sysTicks value.

system objects	theMouse
error messages	None

class description: Timer

class Timer
superclass Object
source file Interval
status Optional
description

Timer defines a class of interval timer objects with a resolution of 1/60th of a second.

instance variables

Var	ticks	Current interval in 1/60ths of a second.
Int	on	Boolean is true if timer is counting.

indexed data None

methods

accessing

get: { -- ticks }

Returns the interval in ticks since the clock was started.

when: { -- ticks }

Update system event record and return its time.

control

start: { -- }

Begin timing an interval.

stop: { -- }

Stop the clock, and store the elapsed interval in ticks ivar.

clear: { -- }

Stops the clock, and resets internal counter to zero.

printing

print: { -- }

Print the current interval in seconds and hundredths.

system objects sysTimer
error messages None

Object defined in Interval source.

Event-Related Yerk Words

get-evt

desk

key-evt

key

addTask

?event

sys

disk-evt

rekey

killTask

stillDown?

null-evt

upd-evt

key!

taskList

waitClick

mouse-evt

actv-evt

dblTicks

5

Windows

About This Chapter

This chapter describes the Yerk classes and words that manage windows for the application. Windows are the central objects around which the rest of a Macintosh application is built. A window isolates a functional portion of your application, giving the user a concrete, visual framework for interaction. Yerk'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 Yerk's Window class, 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
- Yerk - Events
- Yerk - QuickDraw
- Yerk - Controls

Source Files

- Window
- CtlWind

Using Windows

Yerk provides two classes of window objects: class Window, built into the distributed Yerk.com image, provides behavior necessary for all windows, but does not include any management of controls. An optional class, CtlWind, adds the behavior necessary for a window with controls. 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 Yerk, while another part is known both to Yerk and to the Toolbox. From

the point of view of the Toolbox (and conventional languages like Pascal), a window is completely described by a window record. A Yerk 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 Yerk 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 the window object includes a window record as its private data, it passes the address of its own data to the Toolbox as the storage to use for the window record. Thus, any windows that you create will allocate all of their storage in the Yerk dictionary. If this seems undesirable, or if you have an application in which windows come and go dynamically, you can use the Heap> operator to create window objects whose data lives on the heap, and dispose of them when you are through (see the Advanced Yerk Concepts chapter for more detail on using Heap>).

The fact that an object allocates a window record as the first part of its data is important, because it simplifies the interaction between Yerk and the Toolbox. There are many cases in which Yerk 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, Yerk 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 Yerk 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 cfas of Yerk 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 cfa of NULL, except for the activate vector, which is set to the cfa of CLS.

Creating Windows

The steps involved in creating and using a window are as follows: First, instantiate a window class to create an object, and then initialize the action vectors of the window using the action 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
\ set the close, activate, draw and content vectors
4 'cfas null null cls null actions: myWind
```

The Activate vector is executed when the window becomes active, and the Close vector is executed when the user clicks the Close box. Typically, you will use both of these hooks to adjust items on the menu bar; enabling, disabling or changing the text in some cases. The Draw vector is executed when the window receives an update event, which is the Toolbox's way of telling the window to redraw itself. You should load this vector with the cfa of a routine that can reconstruct

the contents of your window; the Toolbox manipulates the visible region of the window's GrafPort so that only the area that actually should be redrawn actually is. If the window is of class CtlWind, any controls associated with the window will be redrawn automatically. Lastly, the Content vector must handle the specific processing that

the window requires when the user clicks the mouse in the window's content region; for instance, a graphics editor might use this vector to select and drag graphics objects.

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.

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.

Using action vectors will reduce the number of subclasses that you would have to define to customize windows for your application. You might still want to create a subclass if you have specific requirements for window deactivation, growing or other characteristics.

As with controls, you cannot define your windows as instance variables of another class, because late-bound messages are sent to any active window (see Chapter 7, Controls). Generally, this is appropriate, because windows tend to be the highest-level objects in an application. Each window encapsulates a major section of the application into a single unit, both from the user's and the programmer's perspective. You can create a background task for a window by redefining the `Idle:` method in class `Window`. This message is sent repeatedly to the front window while the Event object is listening to the event queue, and could be used to do things like call `TEIdle` in a text-edit window, or to update a clock display.

To get a feel for how Yerik's window objects can be used, it is most instructive to look at an existing application, such as `grDemo` or the `Exam` module. We have provided source for these utilities so that you can see how the various classes cooperate to build a real Yerik application. Much of the source, 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-Related Classes

class description: Window

class	Window
superclass	GrafPort
source file	Window - look at the source file for many more methods
status	Core

description

Window is the basic class of windows without controls.

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.
Int	scrollFlg	Flag to not update fpRect for scrolling
Var	zoom	cfa of user supplied zoombox handler

indexed data None

methods

setting characteristics

setLimits: { -- }

Sets both drag and grow limits based on multiple screen regions.

setView: { -- }

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

setGrow: { l t r b T or F -- }

Sets the window's grow limits. If boolean is true, the rectangle coordinates determine the minimum and maximum x and y values that the window can be grown. If false, the window will not be growable. You may use the Yerk word 'grayRgn' to set the l t r b points of the rectangle formed by all graphic screens the mac uses.

setDrag: { l t r b T or F -- }

Sets the window's drag limits. If boolean is true, the rectangle coordinates determine the minimum and maximum x and y values to which the window can be dragged. If false, the window will not be draggable.

setIdle: { cfa -- }

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.

moveTo: { x y -- }

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

show: { -- }

Calls ShowWindow to make the window visible.

hide: { -- }

Calls HideWindow to make the window invisible.

actions: { close enact draw content -- }

Sets action vectors with the cfas provided.

setAct: { enact deact -- }

Sets the activate and deactivate vectors with the cfas provided.

setDraw: { drawCfa -- }

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.

getVRect: { -- l t r b }

Returns the coordinates for the window's vertical scroll bar area.

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.

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.

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 addr len 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 set to non-growable, non-draggable windows with null action vectors except for Draw, which is set to the cfa of CLS.

example: { -- }

Creates an example object of class Window.

system objects
error messages

fWind
None

The Yerk system window.

class description: CtlWind

class CtlWind
superclass Window
source file CtlWind
status Optional

description

CtlWind adds behaviors necessary to support the use of Controls with windows.

instance variables None (inherited from superClass Window)

indexed data None

methods**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, and any controls associated with the window are redrawn.

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. Storage for the window's controls is disposed of.

content: { -- }

This method is executed when a mouse-down event occurs in the window's content region. If there is a hit in a control, the control is tracked and its action vector executed. Otherwise, the window's Content action vector is executed.

system objects None

error messages None

Window-Related Yerk Words

find-window	docWind	rndWind	dlgWind
initFont	savePort	restPort	thePort
theWindow	ctlHit?	g->l	

6

Menus

About This Chapter

This chapter describes the Yerk classes and words that allow you to build your application's menus easily and quickly. Yerk's special menu loader enables you to define all of the data relevant to your menus in a single, easily edited text file.

Recommended Reading

IM - Event Manager

IM - Menu Manager

Yerk - Events

Source Files

Menu

nmenu.txt

Using Menus

Yerk menus integrate the Toolbox concept of a menu within an object that stores Yerk words to be executed when the user makes a particular choice. The Yerk 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 Yerk 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 cfas 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 (excluding the Apple menu, which is predefined), 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.

After creating the menu objects in your code, you must either use the original Yerk method of creating a menu text file or create menu resources with ResEdit. Both ways are discussed below. (If you do use the original method, you must support it yourself, as after version 3.63, it will no longer be supported). If you intend on creating a stand-alone application, you should use the resource method.

The Old, non-supported way: Building the Menu Text File

You create a text file that contains the initialization text for each menu's title, items, key equivalents and handler words. This text file can be loaded when your application starts up

by executing the word GETMTXT, which will allocate the menus by calling the Toolbox and fill them with the data in the file.

The menu text file gathers all of the information about your application's menus into a single place. Yerk uses this information much as the Toolbox uses resources, to initialize the various data within the menu objects. Here is an example of a portion of an actual menu text file:

FILEMEN 256		\ name of object, resource ID
"File"		\ text string for title
"Launch Editor/E"	ed	\ text for 1st item, handler word
"Load.../L"	stdLoad	\ text for 2nd item, handler word
...		\ 1 line for each item
"\\\""		\ end the item list

for each menu in the application, you must have a series of lines in the menu text file in the above order. The first line is the dictionary name of the menu object, followed by a unique integer ID. In certain circumstances, you may decide you don't want to have the menu displayed in the menu bar just yet; in this case, you can add an 'x' after the ID (on the same line). The next line is the title of the menu as it is to appear on the menu bar, in quotes. The lines that follow each specify the text for an item, followed by the name of a Yerk word that will be executed if that item is chosen. Within the quotes, you can use any of the special characters listed in Inside Macintosh to indicate an icon, a key equivalent, or other format specification. The item list is terminated by a line with three backslashes in quotes.

Most applications will have an Apple menu with an About item and the desk accessories. You can simply copy the Apple menu specification that Yerk uses for its menus (available from source file nMenu.txt), and substitute your own information on the About line:

APPLEMEN 1		\ name of menu object and its ID
"\$14"		\ special character for apple symbol
"About Yerk..."	about	\ substitute your own text and handler
"(_____)"	null	\ a separator line
"RES" DRVR		\ load all resource names of type DRVR
"\\\""		

When your application starts up, it should call GETMTXT, passing it a string that is the filename of the menu text file. The file must be available on the current volume. For example:

```
" nMenu.txt" getMtxt
```

is how Yerk loads the menu bar with its own menus. GETMTXT first clears the menu bar of any existing menus, and then loads each menu in the order that it is found in the menu text file. Another word is GETPMTXT, which loads popUp and hierarchical menus without clearing the existing menuBar. Your program may have all menus (up to 24) in one file, read in with getMTxt, or separate files with popUps in another, read in when needed with getPMTxt.

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.

The supported way: Using Menus from a Resource File

You may use any resource editor (ResEdit) to create menu resources. Below is an example of YerK code to bring up the standard YerK menus, assuming the menus are defined in the 'yerK.rsrc' file with the resource ID's 1,2,3,4,5:

```
2 'cfas about null                                1 put: appleMen
5 'cfas stdLoad doSave stdSave Print bye          2 put: fileMen
8 'cfas null null sysCut sysCopy sysPaste sysClear null doEdit 3 put: editMen
9 'cfas doWords doOlist doClist hier exam doDe doGrep null install 4 put: utilMen
9 'cfas pEcho lEcho null .path .room doMlist purge null null      5 put: yerKMen

clear: menubar          \ clear the menubar at compile time
draw: menubar           \ now draw it as blank

: nmenu applemen fileMen editMen utilMen yerKMen 5 init: menubar ;
```

Executing the word 'nmenu' will load in the menus from the resource file, insert them into the menubar, and draw the menubar.

Manipulating Menus

After your menus are loaded, various methods are available to change their characteristics. Get: and Set: fetch and store the item string for a given item. The Menu Manager numbers items from 1 to N instead of starting from 0, so you should be aware that item number N is handled by cfa number N-1 in the menu's indexed data. 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 Enable: and Disable: 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. MBar's Enable: and Disable: methods affect the entire menuBar rather than individual menu items. Finally, Check: and UnCheck: (in Menu) control the display of a checkmark next to an item.

YerK 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-Related Classes

class description: Menu

class	Menu
superclass	X-Array
source file	Menu
status	Core

description

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

instance variables

Int	resID	Resource ID or menu ID.
Var	Mhndl	Handle to the menu's heap storage.

indexed data 4-byte cells (must be cfas of valid YerK words)

methods

creation

init: { resID -- }

Sets the resource ID of this menu.

ID: { -- resID }

Returns the resource ID of this menu.

new: { addr len -- }

Calls the toolbox to create a new menu using this object's data as the menu record.

insert: { -- }

Inserts this menu into the menu bar.

add: { addr len -- }

Appends this menu to the menu bar.

addRes: { type -- }

Adds all resources of a type to this menu. Use this to append fonts or desk accessories to a menu.

accessing

get: { item# -- addr len }
Returns the text string for an item.

set: { addr len item# -- }

Sets the text for an item. (NOTE: stack order is true after version 3.63)

normal: { -- }

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

enable: { item# -- }

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

disable: { item# -- }

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

check: { item# -- }

Shows a check mark by the item.

checked?: { item# -- b }

Returns true if the menu item has been checked.

unCheck: { item# -- }

Removes a check mark from an item.

exec: { item# -- }

Executes the handler for this item.

opendesk: { item# -- }

Runs the desk accessory named by the specified item.

system objects none

error messages "You must send me a **new:** message first"

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

class description: AppleMenu

class	AppleMenu
superclass	Menu
source file	Menu
status	Core

description

Class AppleMenu creates the menu object that displays the Desk Accessory List.

instance variables

none

indexed data 4-byte cells

methods

exec: { item# -- }

Executes the handler (**openDesk: super**). The first 2 items are executed as the superclass (for 'about')

system objects Apple menu usable by any application.

error messages "You must send me a **new:** message first"
An operation was attempted before the menu was created with the Toolbox.

class description: hMenu

class	hMenu
superclass	Menu
source file	Menu
status	Core

description

Class hMenu provides support for hierarchical menus.

instance variables

none

indexed data	4-byte cells (must be cfas of valid Yerk words)
--------------	---

methods

insert: { -- }

Inserts the hierarchical menu in the menubar.

system objects	none
----------------	------

error messages	"You must send me a new: message first" An operation was attempted before the menu was created with the Toolbox.
----------------	---

class description: pMenu

class	pMenu
superclass	hMenu
source file	Menu
status	Core

description

Class pMenu provides primitive support for popUp menus.

instance variables

int	type	Should the menu appear relative to cursor, or at an absolute position
point	offset	The offset of the pmenu from the cursor
int	lastPick	Stores the last item picked.

indexed data 4-byte cells (must be cfas of valid Yerk words)

methods**popUp: { -- }**

Brings the popUp Menu to life. It will appear with the x,y offset from the current mouse position.

getText: { item# -- addr len }

Same as **get:** for class Menu, but acts on the popUp item. **Get:** might be needed to get the text of a hierarchical menu attached to the popUp.

offset: { x y -- }

Set the offsets. If IVAR type is set to 0, this is the relative position that the mouse will appear from the upper left corner of the menu. If type is set to 1, then the stored offset is interpreted as an absolute location within the window's local coordinate system.

position: { x y -- }

Same as offset. Just an alias.

type: { n -- }

Sets the type of popUp. **n** can be 0 or 1. See **offset:**.

putItem: { lastPick -- }

Set the initial item to be hilited. The last item you pick is stored automatically.

getItem: { -- lastPick }

Get the item that was last selected.

getHText: { item -- addr len }

Get the text of the associated hierarchical menu, activated from a popup menu.

getName: { -- addr len }

Get the text of the last selected item.

system objects none

error messages "You must send me a **new:** message first"
An operation was attempted before the menu was created with the
Toolbox.

class description: Mbar

class	MBar
superclass	Object
source file	Menu
status	Core

description

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.

instance variables

24 WordCol	IDs	The list of menu IDs.
24 Array	Menus	An array of menu objects.

indexed data None

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: { nmen0 ... 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: { -- }

Monitors for a click in the menu bar, calls 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.

system objects	menuBar	System-wide menu bar (vectored).
error messages	None	

Menu-Related Yerk Words

?new	mSelect	doDsk	AppleMen
getMtxt	getPMtxt	theMenu	mItem
menuID			

7

Controls

About This Chapter

This chapter describes the Yerk classes and words that provide an interface to the Macintosh Control Manager. Controls are graphical objects that respond to mouse actions, and allow the user a visually direct means of controlling the behavior of the application. Yerk classes facilitate definition of all of the standard control types, and provide behavior appropriate to the Macintosh User Interface Guidelines.

Recommended Reading

IM - Window Manager
IM - Control Manager
Yerk - Windows

Source Files

Ctl
VScroll

Using Controls

Yerk groups Macintosh controls into two classes: class Control provides support for basic controls with a single part, and includes buttons, check boxes and radio buttons. Class VScroll adds the behavior necessary for scroll bars, which have 5 parts.

In order to use controls within your application, you must define a window that has the ability to recognize and manage controls. This behavior is provided by class CtlWind, whose Content: and Draw: methods include send appropriate messages to check for control hits with the mouse or redraw the window's control list. CtlWind contains whatever support can be included for a generic window with controls, and uses the calls that the Control Manager provides which traverse the window's control list rather than accepting handles to particular controls. You might wish to define a subclass of CtlWind to deal more directly with the specific controls associated with your window. For example, when a window becomes inactive, it should send Disable: messages to any scroll bars that it contains. This can only be sent to the individual controls, and not the entire control list.

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. Yerk, on the other hand, combines the control's Toolbox-related data with its own 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.

Controls store the cfas of Yerk 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 the content region of a window with controls, CtlWind executes the Yerk word CTLHIT? which calls the Toolbox routine FindControl to determine whether the mouse was inside of an active control when the button was depressed. If so, 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 the 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 Yerk word. Yerk contains a special compiler that packages Yerk 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 Yerk 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.

Creating Control Objects

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

```
Control saveBtn
```

You should then initialize the newly created object to assign it a particular control type and give it an action procedure. For example:

```
buttonID init: saveBtn  
'c doSave actions: saveBtn
```

This assigns saveBtn a control type of Button, and sets 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 your program executes, you must send a New: message to the control to cause it to create a Toolbox Control record on the heap and draw itself within its owning window. The New: message for an object of class Control requires as input a location, a title and the address of the owning window object. For instance:

```
100 250 " Save" myWind new: saveBtn
```

The control object determines the width of its rectangle automatically by determining the width in pixels of the title that you provide. The location that you specify is the TopLeft portion of the control's enclosing rectangle.

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: myCtl 1- put: myCtl -1 scroll: theText  ;
```

In this example, the word myCtl is actually a Vect that Yerk provides as a simple way for a control action word to derive its owning control object. Yerk automatically compiles a late-bound reference when a Vect or Value is used as the receiver of a message, and in this case myCtl is vectored to a word that derives the control object's address from the methods stack. 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 instance variables, because to do so would fail to provide a class pointer for the runtime method lookup. Late binding is necessary because there are cases in which the Toolbox returns the address of an object to the application, but it 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.

For this reason, you cannot define these types of objects (those whose address might be received from the Toolbox) as instance variables. We have found that it is generally advantageous to make controls and windows global objects anyway, because it avoids a lot of passthrough methods that take up space. However, you might want to define a new type of control window that could be instantiated dynamically, such as a text-edit window. It is most convenient in this case to bind the control to the window as an instance variable, because then both will be created together. The solution is to use a Var within the window to hold the address of the control object, and then instantiate the control on the heap when the window is created:

```
:M CLASSINIT:  heap> VScroll put: myScrollBar  ;M
```

This will create a scroll bar object on the heap with a valid class pointer, and place its address in the Var owned by the window. You can then send the scroll bar messages such as:

```
get: [ obj: myScrollBar ]
```

When the window is closed, you should dispose of the heap block allocated to the scroll bar by using the Dispose: method directly on the Var.

Dialogs

Yerk 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, Yerk 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. All dialogs in Yerk are currently modal - if you require non-modal dialog support, use a CtlWind with procID

dlgWind and control objects, and handle it via Yerk's normal window mechanism as opposed to the Toolbox Dialog Manager.

Control-Related Classes

class description: Control

class	Control
superclass	Object
source file	Ctl
status	Core

description

Control describes the class of basic, single-part controls, such as buttons, radio buttons and check boxes.

instance variables

Int	proclD	The Control definition ID for the Toolbox.
Handle	cltHndl	Handle to the control record.
Var	action	Cfa of action word to execute.
Int	myValue	Contains the numeric value of the control.
Var	ownWind	Pointer to window that owns this control.

indexed data None

methods

accessing

init: { proclD -- }

Sets the control definition procedure for the control.

actions: { cfa -- }

Sets the action word for the control.

get: { -- val }

Returns the value of the control.

put: { val -- }

Sets the value of the control.

setTitle: { addr len -- }

Sets title of control.

getTitle: { -- addr len }

Gets title of control.

getRect: { -- l t r b }

Returns the coordinates of the control rectangle.

handle: { -- ctlHandle }

Returns the value of the control handle.

exec: { part# -- }

Executes the control's action handler.

display

size: { w h -- }

Sets the width and height of the control rectangle.

moveTo: { x y -- }

Sets the topLeft point of the control rectangle.

hilite: { hiliteState -- }

Hilights, disables, or enables the entire control.

enable: { -- }

Enables the control.

disable: { -- }

Disables the control.

update: { -- }

Causes the control to be redrawn.

hide: { -- }

Calls Toolbox HideControl.

show: { -- }

Calls Toolbox ShowControl.

object creation

new: { x y addr len theWind -- }

Create a new control in the Toolbox for this control object. X and Y are the location of the upper left corner. Addr and Len specify the title string, and theWind is the address of the owning window object. The control will be created visible, with a control rectangle large enough to accommodate the title. Initial value is 0, and range is 0 to 1.

getnew: { id window -- }

Creates the control defined in a resource file. The resource file must first be opened, if not already.

classInit: { -- }

Sets default control to type Button with null action.

example: { -- }

Creates an example control of type Button.

system objects	None
error messages	None

class description: VScroll

class VScroll
superclass Control
source file VScroll
status Optional

description

VScroll provides the additional support required of scroll bars beyond the methods that its superclass provides for simple controls.

instance variables

5 bArray	parts	List of part numbers for table lookup.
5 X-Array	actions	Cfas corresponding to each part.

indexed data None

methods**accessing**

actions: { InUp InDn pgUp pgDn thumb -- }

Sets action handlers for the 5 parts.

putRange: { lo hi -- }

Set the lower and upper limits for the control's values.

object creation

new: { x y len theWind -- }

Create a new scroll bar in the Toolbox and save its handle. This method assumes a width of 16, for a vertical scroll bar.

classinit: { -- }

Sets action handlers to null and selects scrollbar type control.

example: { theWind -- }

Create an example vertical scroll bar and display in theWind.

display

disable: { -- }

Sets entire control to 255 hiliting.

enable: { -- }

Sets entire control to enabled hiliting.

system objects	None
error messages	None

Control-Related Yerk Words

myCtl

ctlProc

vsID

useWFont

theCtl

buttonID

get-ctl-obj

ctlHit?

checkID

set-ctl-obj

ctlExec

radioID

nullOSStr

8

Graphics

About This Chapter

This chapter describes the YerK 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 YerK 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
QD1

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. RndRect, an optional class, modifies Rect's behaviors for rounded-corner rectangles, and Oval for rectangle-delimited ovals. Class GrafPort is the superclass of window, and describes the graphics structure that QuickDraw uses as its foundation for windowed behavior.

QDBitmap describes a QuickDraw bitMap structure, which is a rectangular region of dots that are on or off, each dot corresponding to a bit in memory. QDBitmap isn't very useful in itself, but can provide a foundation for a more elaborate bitMap subclass. Class Image combines a bitmap's structure with data necessary to draw the bitmap on the screen via the QuickDraw CopyBits call. An Image is an indexed object whose indexed area is used to hold the source bitmap, and can be

moved to an arbitrary location on the screen. Image is useful for storing bit images that you might want to move around on the screen. You can initialize an Image's graphics data programmatically without having to go through a resource file, using any available methods to fill the Image's indexed data area. This might be useful for fast-moving images in an animated game.

Class Picture is an interface to QuickDraw's picture mechanism, storing its data in a resource file. Pictures can be created by drawing in MacPaint, cutting a section of the drawing, and pasting it into the Scrapbook; you must then use the resource editor to move the Picture from the scrapbook into your resource file. You can draw the Picture by sending a `getNew:` message followed by `goTo:` and `Draw:`. Class Icon also stores its data in a resource file, but in this case the data is limited to an icon's 32-by-32 bit array. Classes Picture and Icon rely on the Toolbox Utilities calls `GetPicture` and `GetIcon`.

The YerK 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`. YerK predefines four cursor images for you: `IBeamCurs`, `crossCurs`, `plusCurs`, and `watchCurs`.

YerK 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.

Graphics-Related Classes

class description: Point

class	Point
superclass	Object
source file	QD
status	Core

description

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 Yerk always represents Points on the stack as (X Y --).

instance variables

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

indexed data None

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.

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.

system objects	None
error messages	None

class description: Rect

class Rect
superclass Object
source file QD
status Core

description

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

instance variables

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

indexed data None

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 }

getTopX: { -- x }

getTopY: { -- y }

getBot: { -- 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 -- }

=: { rect -- }

Implements a right-to-left assignment of one rectangle to another.

size: { -- w h }

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

center: { -- 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 same center point by calling InsetRect. *Does not redraw the Rect.*

offset: { dx dy -- }

Causes the rectangle to be moved a specified distance X and Y. *Does not redraw the Rect.*

drawing

draw: { -- }

Draws the rectangle's frame.

print: { -- }

Draws the rectangle's frame.

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.

example: { -- }

Shows an example of a rect.

system objects	tempRect fpRect	A scratch rectangle. Used for scrolling within fWind.
error messages	None	

class description: RndRect

class	RndRect
superclass	Rect
source file	QD1
status	Optional

description

This class implements Rect's methods for rounded-corner rectangles.

instance variables

Point	ovalSize	Holds the X and Y radius of the corner curves.
-------	----------	--

indexed data	None
--------------	------

methods**accessing**

init: { x y -- }

Sets the radius in pixels of the rectangle's rounded corners.

(see methods in class Rect)

system objects	None
error messages	None

class description: Oval

class	Oval
superclass	Rect
source file	QD1
status	Optional

description

This class implements drawing methods for ovals, which are bounded by a rectangle.

instance variables none

indexed data None

methods

(see methods in class Rect)

system objects	None
error messages	None

class description: GrafPort

class	GrafPort
superclass	Object
source file	QD
status	Core

description

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.

instance variables

16 Bytes	graf1	Allocates the first 16 bytes of the GrafPort.
Rect	PortRect	The port rectangle defining the GrafPort's limits.
84 Bytes	graf2	Allocates the last 84 bytes of the GrafPort.

indexed data	None
--------------	------

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.

system objects	None
error messages	None

class description: QDBitmap

class	QDBitmap
superclass	Object
source file	QD1
status	Optional

description

This is a template class that simply maps the data structure of a QuickDraw bitmap. You can use it as a superclass for other classes that have drawing and bit manipulation methods.

instance variables

Var	baseAddr	The absolute base address of the bit image.
Int	rowBytes	How many bytes make up a horizontal row.
Rect	bndsRect	Sets the coordinate plane for the bitmap.

indexed data None

methods**accessing**

put: { addr width l t r b -- }

Stores new values for base address, rowBytes and bndsRect.

system objects None

error messages None

class description: mage

class	Image
superclass	Array
source file	QD1
status	Optional

description

Image is a sophisticated class that can be used to describe an arbitrary bit image. You provide the drawing mode, the rowBytes and coordinates of the source bitmap, and fill in the indexed area with the actual bit image for the object. The image can then be drawn at any location on the screen. See the IM - QuickDraw section on CopyBits for more information.

instance variables

Var	baseAddr	Base address of source bitmap (see QDBitMap).
Int	rowBytes	Row bytes for source bitmap.
Rect	bndsRect	Sets the coordinate plane for the source bitmap.
Var	destBits	Address of destination bit image.
Int	mode	Drawing mode for the image.
Rect	destRect	Rectangle in which image is to be drawn.

indexed data 4-byte cells

methods**accessing****getPort: { -- }**

Derives appropriate source and destination bit image pointers from the current GrafPort. Should be done once at runtime for each Image.

init: { mode rBytes x y -- }

Sets the drawing mode, row bytes of the source bitmap, and location of the upper left corner of bndsRect for the source bitmap.

goTo: { x y -- }

Moves the location of the destination image without actually drawing it.

drawing**draw: { -- }**

Draws the image at its current location.

moveTo: { x y -- }

Moves the location of the destination image and redraws it.

classInit: { -- }

Sets the drawing mode to Xor, row-bytes = 2 and the location of the destination image 0,0.

system objects None

error messages None

class description: Picture

class	Picture
superclass	Object
source file	QD1
status	Optional

description

This class provides an interface to the Toolbox Picture facility. This provides a convenient way to represent arbitrary graphics images as a series of calls to QuickDraw routines. Pictures can be created programmatically or by using MacPaint and cutting and pasting into the Scrapbook desk accessory.

instance variables

Handle	picHndl	Handle to an open Picture's data on the Heap.
Int	resID	Resource ID for the Picture data.
Rect	destRect	Boundaries for drawing the Picture.

indexed data None

methods**accessing**

init: { resID -- }

Sets the resource ID of the picture.

getNew: { -- }

Loads the Picture's data from a resource file using the resource ID set by **init:**. (You should not balance each **getNew:** with a **kill:**; **getNew:** can be used repeatedly and will automatically reuse its heap space.)

open: { l t r b -- }

Opens a new picture for programmatic creation using the coordinates provided for the picture's frame.

close: { -- }

Closes the currently open picture.

kill: { -- }

Disposes of the picture's heap storage.

size: { x y -- }

Sets the width and height of the picture's destination rectangle.

goTo: { x y -- }

Sets the location of the upper left corner of the picture. *Does not redraw the image.*

drawing

draw: { -- }

Draws the picture at its current location.

disp: { x y resID -- }

Combines the actions of **init:**, **getNew:**, **goTo:** and **draw:**. Use this method when displaying an image from a PICT resource, like those from the scrapbook.

system objects None

error messages None

class description: Icon

class Icon
superclass Object
source file QD1
status Optional

description

Icons are bit images confined to a 32-by-32 bit format. Class Icon allows you to define an icon's data within a resource file, and draw it from within your application by using only its resource ID.

instance variables

Handle	theHandle	Handle to the icon's heap data.
Int	resID	Resource ID for getNew :
Rect	theRect	Destination rectangle for drawing.

indexed data None

methods**accessing**

init: { resID -- }
Sets the object's resource ID.

getNew: { -- }
Loads the Icon's data from a resource file using the resource ID in **resID**.

goTo: { x y -- }
Sets the location of the upper left corner of the icon without drawing.

drawing

draw: { -- }
Draws the icon at its current location.

disp: { x y resID -- }
Combines the actions of **init:**, **goTo:** and **draw:**.

system objects None
error messages None

Graphics-Related YerK Words

Cursor	sysPat	g->l	line
scroll	>origin	gotoxy	@xy
iBeamCurs	plusCurs	crossCurs	watchCurs

srcCopy
restPort

srcOr
cls

srcXor

savePort

9

Dialogs

About This Chapter

This chapter describes the Yerk 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
Yerk - Windows
Yerk - Controls

Source Files

Alert
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 non-modal. Yerk's version 2.0 Dialog class supports only modal dialogs, which do not allow the user to activate a different window while the dialog is active. Non-modal dialogs can be implemented in Yerk as windows with procID dlgWind, and control objects as items.

Dialogs are heavily dependent upon resource definitions for the window and the item list. You can use Rmaker or the Resource Editor to create resources of type DLOG and DITL, and then store the resource ID in a Yerk object of class Dialog (see IM - Dialog Manager). The items in a dialog's item list are not defined as control objects in Yerk, because they can more easily be accessed via methods

in class Dialog itself. A Dialog object's indexed cells are loaded with the cfas of Yerk 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.

Each handler must end with one of two possible messages to the owning dialog. 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 reenter the dialog by sending a `Modal:` message to it. Under no circumstances must the handler simply return, because that will probably result in entering Yerk's event management, which will attempt to treat the Dialog as a subclass of Window, which it is not.

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.

To summarize, here are the steps involved in creating a Yerk dialog: first, you must create resource definitions for the dialog and its item list (for examples, see Yerk's resource definitions in Yerk.r). Then define a dialog object with as many indexed cells as there are items, and initialize it to the resource ID of the DLOG resource. For instance:

```
4 Dialog inDlg \ Yerk's general input dialog has 4 items
3 Init: inDlg \ the resource ID of the DLOG is 3
```

Then, you should set the item handlers of the dialog with the `Actions:` method:

```
4 'cfas getChoice noAction noAction getChoice Actions: inDlg
```

This causes the OK button and the text-edit field (items 1 and 4, indexed cells 0 and 3) of Yerk's general input dialog to execute the word `GETCHOICE` if either is selected. `GETCHOICE` sends a `GetText:` message to the text-edit item and places a true on the stack, telling the caller that a valid string was entered. The other handlers correspond to the dialog's Cancel button and a static text item, which will not return anyway because it is disabled in the Resource definition.

When it is time to execute the Dialog, ensure that the correct resource file is open, send a `GetNew:` message to load and display the dialog window, followed by a `Modal:` message to enter the Dialog Manager's event-handling loop. From here, everything will proceed automatically, presuming that your handlers were coded correctly. If you wish to modify the dialog's item list in memory, you can do so after sending the `GetNew:` message and before the `Modal:` message.

The `Modal:` method does not close the dialog before returning to the caller. This is necessary to support reentering the dialog from a control handler. The implication is that any handler that doesn't reenter the dialog must close the dialog via the `Close:` method. Here, then, is the structure required to use a Dialog box:

```
getNew: inDlg \ Loads the DLOG and DITL resources into memory
... \ Optional code; may pre-draw the dialog
Modal: inDlg \ Passes control to the Dialog Manager
```


... \ Optional code
Close: inDlg \ Dismisses the dialog box from the screen

In certain circumstances you may need to reenter the Dialog Manager's loop after having received control back from Modal: (for instance, a check box or radio button item). This can

be accomplished by using the word `ReturnToModal`. (Sending another `Modal:` message will seem to work but will pollute the stack -- **iterative use of Modal: will crash the system!**). The following is an example using `ReturnToModal` for Radio buttons:

```

3 Dialog Sam      \ This dialog has 2 radio buttons and an OK button
1000 init: Sam    \ assumes there is a dialog with resource ID of 1000

2 RadioSet Rset   \ a set of 2 radio buttons
Sam init: Rset    \ associate Radio set with the Dialog
1 add: Rset       \ add the item numbers to the list
2 add: Rset

\ ( -- ) Turn off any previously selected button; turn on button now selected
: Rbutton get: theItem select: Rset ReturnToModal ;

\ ( -- ) Report which button was ultimately selected
: Done ." You chose item #" get: Rset . ;

3 'cfas Rbutton Rbutton Done actions: Sam    \ set action vectors

getNew: Sam       \ load the dialog resource
1 select: Rset    \ default button setting
modal: Sam        \ pass control to the Dialog Manager
close: Sam        \ remove the dialog box from the screen

```

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. Use `Alert"` in place of `Abort"` in your application to comply with the "Mac Standard". See section in chapter II.6.

Dialog-Related Classes

class description: Dialog

class Dialog
superclass X-Array
source file Dialog
status Optional

description

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

instance variables

Int	resID	The resource ID of a DLOG resource.
Var	dialPtr	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 cfas)

methods

object creation

init: { resID -- }

Sets the resource ID of the dialog's DLOG resource.

hilite: { item# -- }

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

actions: { cfa0 ... cfaN -- }

Sets the dialog's item handlers.

setProc: { cfa -- }

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

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 Yerk 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.

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 close the dialog or use **ReturnToModal**.

close: { -- }

Closes the dialog window and disposes of the associated heap.

accessing items

get: { item# -- val }

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

put: { val item# -- }

Sets the value of a control item.

handle: { item# -- itemHandle }

Returns the handle to an item to allow direct manipulation via the Toolbox routines.

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.

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.

frame: { -- }

Draw the frame around the boldened OK button.

setItem: { userItem -- }

Tell the dialog to include this user item. See class UserItem.

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) holding parameters for selected items.

system objects
error messages

inDlg Yerk's general input dialog (module).
None

class description: UserItem

class UserItem
superclass Rect
source file Dialog
status Optional

description

UserItem is a class to support the use of user items in dialogs. A draw procedure (see IM) is necessary to tell the Dialog Manager how to draw the user item.

instance variables

Var	myProc	Drawing procedure.
Int	disabled	Is this item to be disabled?.
Int	itemNo	Item number in dialog.

methods

putItem: { item# -- }
Tell the userItem what item it is within the dialog.

disable: { -- }
Marks the user item as disabled.

enable: { -- }
Marks the user item as enabled.

setProc: { cfaProc -- }
Sets the procedure for drawing the user item.

system objects None
error messages None

class description: RadioSet

class RadioSet
superclass WordCol
source file Radio
status Optional

description

RadioSet provides support for radio button controls by organizing them into a group and handling the display function.

instance variables

Var theDialog Pointer to dialog that owns this control.

indexed data None

methods

accessing

init: { dialog -- }

Associates this dialog with the radio set.

select: { item# -- }

Select radio button for this item. If any other button had been previously selected, it will be deselected.

get: { -- item# }

Return the number of the item which has been selected.

system objects None

error messages None

class description: Alert

class Alert
superclass X-Array
source file Alert
status Optional

description

Alert loads and displays alerts defined in a resource file.

instance variables

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

indexed data 4-byte cells (must be cfas)

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:**.

system objects None

error messages None

Dialog-Related Yerk Words

theItem itemHandle itemType

10

Drivers

About This Chapter

This chapter describes the Yerk classes and words that allow your application to communicate with Macintosh device drivers such as the sound, printer or serial driver. All Macintosh device drivers communicate with the application via 6 basic calls: Open, Close, Read, Write, Control and Status. Yerk provides a general interface to the driver support upon which can be built classes tailored to specific drivers.

Recommended Reading

IM - Device Drivers

IM - Serial Driver

IM - Sound Driver

IM - Printer Driver

IM - OS

Source Files

Drv

Serial

Print

Sound

Using Drivers

Macintosh device drivers use Parameter Blocks to pass data to and from the application. Class PBDrv maps out the structure of a generalized Parameter Block object whose behavior is appropriate to any driver. You can then define subclasses of PBDrv, as we have in classes Port and Printer, that add specific behavior for the driver in question. All device drivers communicate with the application via 6 basic Toolbox calls: Open, Close, Read, Write and Status. PBDrv has methods that perform these calls, assuming that you have already set up the parameter block with the appropriate data. The Yerk word (FDOS) can also be used to make a Toolbox call involving a parameter block (see glossary).

PBDrv's data consists of a 12-byte area for the text name of the driver, followed by a set of instance variables that allocate a parameter block. Of the four variant records for parameter blocks, the two that are of most interest for non-block-structured devices are IOParam and

CtlParam. These two variants both begin at offset 26 in the parameter block (immediately following IORefNum). Yerk does not have innate support for variant records, so the instance variables within PBDrvIr reflect a hybrid parameter block that can be used for both I/O (Open, Close, Read, Write) and control (Control, Status) calls.

To make a Control or Status call, you should place the desired Control code in the Ivar csCode, followed by the call's parameters in csp1, csp2 and following fields if needed.

There might be calls for which the mapping of csp1 and csp2 as Ints is inconvenient, in which case you can simply do direct stores to the area starting at (addr: header + 28). Your individual driver classes should implement the commonly used Control calls by setting up the data as described and then sending the message Control: super or Status: super.

Read and Write calls can be performed in two modes: Wait (synchronous) or No-Wait (asynchronous). In the former, your code will hang until the I/O operation is complete, at which time control is returned in the normal manner. No-waited I/O requires that you pass the address of a Yerk :PROC word to be executed when the call completes. Control immediately returns to the place following the call, but your passed-in completion word will be executed as an interrupt handler whenever the I/O call completes. This would allow you, for instance, to write a terminal emulator in which you have simultaneously pending Read and a Write calls to the serial port. Whichever one occurred would be serviced by the appropriate completion word. Your completion routine should be very short, because certain critical Macintosh interrupts are disabled while your routine is executing. The best approach is to set flags within your completion routine, and do any lengthy operations after you have returned to the main wait loop by polling the state of the flags.

Class Port implements an interface to the Macintosh ROM serial driver. The sequence of messages in the following example is important - Open: and Reset: must come last:

```
Port Modem
0 1 Init: Modem          \ port=0 (modem port) direction=1 (output)
1 8 0 setConfig: Modem   \ 1 stop bit, 8 data bits, no parity
9600 baud: Modem
open: Modem
reset: Modem              \ Send the parameters to the port
```

The serial ports have separate IOREfNums assigned for their input and output sides; you assign one of the four possibilities to a given Port object by using the Init: method. The Config: method allows you to set the stop bits data bits and parity. The Baud: method accepts most of the standard baud rates to set line speed. The Cts: method (not shown in the example) allows you to turn hardware handshake on or off; both waited and nowait Read and Write are supported. You can then Open: the port and send a Reset: message to send the communications parameters to the driver.

Driver-Related Classes

class description: PBDrvr

class	PBDrvr
superclass	Object
source file	Drv
status	Optional

description

This class provides a general interface to any Macintosh device driver. It defines a parameter block with basic methods to manipulate its data, along with implementations of the 6 basic Device Driver Toolbox calls.

instance variables

12 Bytes	name	Text name of driver.
12 Bytes	header	Fields for internal use
Var	IOComp	Absolute address of IO completion PROC word.
Int	IOResult	Result code from last operation.
Var	IONamePtr	Abs address of driver name.
Int	Vref	Volume reference number
Int	IORefnum	I/O reference number identifies which driver.
Int	CSCode	Control call type code.
Int	csp1	Control call parameters
Int	csp2	
Var	IOBuffer	Abs address of the buffer for I/O
Var	IOReq	Number of bytes to read or write.
Var	IOAct	Number of bytes actually read or written.
Int	IOPosMode	Block device positioning mode.
Var	IOPosOffset	Block device offset position.

indexed data None

methods

accessing

fcb: { -- addr }

Returns the base address of the parameter block.

name: { addr len -- }

Sets the name of the driver.

bytesRead: { -- #bytes }

Returns the actual number of bytes read or written.

result: { -- code }

Returns the completion code from the last operation.

I/O operations

open: { -- fcode }

Attempts to open the driver with the given name and/or refNum, and returns the result code.

close: { -- fcode }

Attempts to close the driver with the given name and/or refNum, and returns the result code.

read: { addr len -- fcode }

Performs a waited read into the buffer at **addr** for **len** bytes.

write: { addr len -- fcode }

Performs a waited write from the buffer at **addr** for **len** bytes.

readNW: { cfa addr len -- fcode }

Performs a no-waited read into the buffer at **addr** for **len** bytes, and executes the proc specified by **cfa** (which must be a :PROC definition) when the I/O is complete.

writeNW: { cfa addr len -- fcode }

Performs a no-waited write from the buffer at **addr** for **len** bytes, and executes the proc specified by **cfa** (which must be a :PROC definition) when the I/O is complete.

kill: { -- fcode }

Kills any pending operation of the port.

system objects	None
error messages	None

class description: Port

class	Port
superclass	PBDrv
source file	Serial
status	Optional

description

Port is a subclass of PBDrv that provides an interface to the Macintosh ROM serial driver. It supports both waited and no-wait I/O with optional CTS hardware handshaking.

instance variables

Int	thePort	0=modem port, 1=printer port.
Int	direction	0=input, 1=output, 2=both.
Int	config	Configuration word for SerReset Control call.
Int	inRefNum	Input IOREfNum
Int	outRefNum	Output IOREfNum

indexed data None

methods**accessing**

init: { port# direction -- }

Sets the physical port and direction for this object. Port# can be 0 (modem) or 1 (printer); direction can be 0 (input), 1 (output), or 2 (both).

config: { stopBits dataBits parity-- }

Sets communication parms for the Port. Stop bits can be 1 or 2; data bits can be 7 or 8; and parity can be 1 (odd), 2 (even), or 0 (none).

setConfig: { configWord -- }

Allows you to set the config word directly.

baud: { baudRate -- }

Sets the data transfer rate for the port. Should be in multiples of 300. **Reset:** should be used after setting this way.

buffer: { addr len -- }

Increases the internal buffer size from its default of 64 bytes.

isOpen: { -- b }

True if the driver has been opened by any application.

I/O operations (also see PBDvr)

open: { -- fcode }

Opens the read driver and/or write driver for the port and sends it the current communications parameters.

control: { -- }

Performs a control call using the port's parameter block. See IM.

status: { -- }

Performs a status call using the port's parameter block. See IM.

reset: { -- }

Sends the communications parameters to the port. The port must be opened and parms set before sending the **reset**.

get: { -- char }

Reads a single character from the serial port (waited).

put: { char -- }

Writes a single character to the serial port (waited).

read: { addr len -- fcode }

Performs a waited read into the buffer at **addr** for **len** bytes.

write: { addr len - fcode }

Performs a waited write from the buffer at **addr** for **len** bytes.

readNW: { cfa addr len -- fcode }

Performs a no-waited read into the buffer at **addr** for **len** bytes, and executes the proc specified by **cfa** (which must be a :PROC definition) when the I/O is complete.

writeNW: { cfa addr len -- fcode }

Performs a no-waited write from the buffer at **addr** for **len** bytes, and executes the proc specified by **cfa** (which must be a :PROC definition) when the I/O is complete.

bytesRead: { -- #bytes }

Returns the actual number of bytes read or written.

bytesIn: { -- #bytes }

Returns the actual number of bytes received, but not read.

baudRate: { baudRate -- }

Sets the data transfer rate for the port. Should be in multiples of 300. This informs the Mac driver immediately, and you do not have to **Reset:**.

cts: { bool -- fcode }

Turns CTS handshaking on or off.

dtr: { bool -- fcode }

Turns DTR handshaking on or off.

xon: { bool -- fcode }

Turns Xon/Xoff handshaking on or off.

system objects None

error messages None

Driver-Related YerK Words

(open)

(read)

(write)

(close)

(fdos)

11

Floating Point

About This Chapter

This chapter describes the Yerk classes and words that manage floating point values for an application running on a Macintosh with 512K bytes of memory, or more. A floating point number is a 10 byte data value which is stored in a block in the heap and is accessed through its pointer.

Recommended Reading

SANE manual

Source Files

fltMem
fpCode
fargs
fpi/o
finterpret
fvalue
elCode
fpExtra
Float

Using Floating Point

Class Float is an interface to the *Standard Apple Numerics Environment*, (SANE). Floating point computations are implemented as direct calls to SANE routines. Many but not all SANE functions are implemented as methods of class Float. Other functions are available as Yerk words, (e.g., ln1, exp1, x**y, compound, annuity, etc...), See the Glossary in Part IV of the manual.

The Yerk floating point interpreter is contained in YerkFP.com, not Yerk.com. To use the floating point, you must open either YerkFP.com, or a saved dictionary image that originated with YerkFP.com.

It is important to note that Yerk inherits certain behaviors from FORTH 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 Yerk floating point interpreter the occurrence of a decimal point causes conversion to a 10 byte Float number 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.

A floating point number 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 two 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 your self 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
print: harold
2 at: harold e.

\ set 2nd element to 2.145
\ print all elements of harold
\ 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 Yerk 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. Yerk 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 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.

Floating Point - Related Classes

class description: Float

class	Float
superclass	Object
source file	Float
status	Core

description

Float provides the basic methods for manipulation of floating point numbers.

instance variables

10 Bytes	data	Floating point representation of the value.
----------	------	---

indexed data	None
--------------	------

methods

accessing

get: { -- fval }

Returns the value in the data area.

put: { fval -- }

Stores a new value in the data area.

=: { faddr -- }

Assigns this float's data to another object.

sign manipulation

absval: { -- |fval| }

Computes the absolute value of the Float and returns the result.

neg: { -- ±fval }

Changes the sign of the Float and returns the result.

arithmetic functions

+: { fval -- }

Adds **fval** to the contents of the Float's data.

-: { fval -- }

Subtracts **fval** from the contents of the Float's data.

***: { fval -- }**

Multiplies **fval** into the contents of the Float's data.

/: { fval -- }

Divides the contents of the Float's data by **fval**.

trigonometric functions

sin: { -- sin(x) }

Computes the sine of the value of the Float.

cos: { -- cos(x) }

Computes the cosine of the value of the Float.

tan: { -- tan(x) }

Computes the tangent of the value of the Float.

arctan: { -- tan⁻¹(x) }

Computes the arc tangent of the value of the Float.

transcendental functions

ln: { -- ln(x) }

Computes the natural logarithm of the value of the Float.

exp: { - -e^x }

Computes the exp of the value of the Float.

log: { -- log₁₀(x) }

Computes the log base 10 of the value of the Float.

antilog: { -- 10^x }

Computes the anti logarithm of the value of the Float.

conversions

deg: { -- degrees }

Converts radians to degrees and returns the result.

rad: { -- radians }

Converts degrees to radians and returns the result.

printing

print: { -- }

Prints the value in the data area of this Float.

system objects	None
error messages	None

class description: fArray

class	fArray
superclass	Object
source file	Float
status	Core

description

fArray provides an array construct for floating point numbers.

instance variables

None

indexed data	10-byte cells
--------------	---------------

methods**accessing**

at: { index -- fval }

Returns the data at a 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.

system objects	None
----------------	------

error messages	None
----------------	------

class description: FltHeap

class FltHeap
superclass Object
source file FltMem
status Core

description

 This class defines the floating heap.

instance variables

 Int FreeHead Offset of first free block.

indexed data 12-byte cells

methods**object creation**

init: { -- }

 Sets all blocks to free and links them together.

new: { -- fPtr }

 Returns a pointer to a new block.

dispose: { fPtr -- }

 Disposes of the block pointed at by **fPtr**.

room: { -- #free }

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

system objects

 fltMem The system supplied float
heap provides 100 blocks.

error messages

 None

Floating Point-Related Yerk Words

fDup	fDrop	fOver	fValue
fCon	fLit	fLiteral	e
pi	1.0	0.0	ln(10)
e.	e.r	f,	fnumber
f+	f-	f*	f/
float>	>float	f=	f<>
f<	f>	f<=	f>=
f0=	f0<	f0>	fAbs
fNegate	round	trunc	ln
ln1	log	log2	log21
exp	exp2	exp1	exp21
sin	cos	tan	arctan
cot	x**y	compound	annuity
deg2rad	rad2deg	sqrt	yerk>flt
yerk>int	f.r	atof	fmax
fmin			