# Mops

***Mike's Object-oriented Programming System***

## Version 2.6

# Part II

## General Reference

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

Mops implemented by: Michael Hore

Able assistance from: Doug Hoffman
Greg Haverkamp
Xan Gregg

Documentation updated: Version 2.6, June 1995

Documentation formatted by: Craig Treleaven

### Printing this document

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

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

### Viewing on-line

Of course, you can read the whole manual on-screen. Word's Find… command can help to locate items of interest. One other technique is useful but not well known. Use the Outline View and click the "2" in the ruler at the top of the screen. Word will then show the chapters and the sub-headings within. Whichever line is at the top of the window in outline view will become the line at the top of the window when you switch back to Normal View. By scrolling in Outline View, you can quickly find the section of interest and position the window for reading in Normal View.

### Two-sided printing

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

### Single-sided printing

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

### A4 Paper

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

### Table of Contents

Use the Table of Contents… dialog to collect headings from level 1 to level 3 for the Table of Contents. Figure captions have Heading 5 style, but I didn't see a reason to create a table of figures.

## The Mops Menu Bar

Mops has a simple set of menus, yet the features built into them make writing code, compiling, and debugging rather easy. When you start Mops.dic, a specially designed Mops "front end" brings in the basic Mops menus—Apple, File, Edit, Utilities, and Mops.

To help you understand the functions of each menu selection, we'll describe the action of each menu item. We'll also explain the built-in utilities, which can make you more productive in program creation and debugging. Also see Part II, Chapter 2 for more details on the operation of an Editor.

### The Apple menu

About Mops…

Displays the Mops version number and the date that version was released.

### The File menu

Load…

Allows you to load text source files on top of the Mops dictionary currently in memory (the same as issuing the "// filename" command from the Mops prompt). The standard file dialog box appears, from which you can select the file to load. As the source file loads, it is compiled by Mops. If your program requires the loading of several text files, the files must be loaded in the proper order (so that dependent words and classes are loaded after the words or classes they depend on). The word NEED which we described in the Tutorial makes it easy to ensure everything is loaded in the right order.

Save Dictionary

Copies to a disk a compiled image of a program you have in memory. It saves the image to a file with the same filename as is shown at the top of the Mops window. For this reason, Save should be used with care. If you have added code to Mops.dic and wish to save the image as a separate application, then use the Save As… selection, below; otherwise, your Mops.dic file will contain your additions.

Save Dictionary As…

Lets you copy to a disk a compiled image of a program you have in memory (you are prompted for a new filename, and you may save to a different disk, if you like). For example, when Mops.dic was originally built, its compiled image was saved with this command. To start a program saved in this manner, double-click the appropriate Mops document icon from the desktop, just as you start Mops.dic. It is recommended that you save programs in this manner only after their source code has been sufficiently debugged. Until then, you'll want to take advantage of the interactivity of the Mops interpreter while debugging source files by maintaining the code as source files and Loading them to test how well the program runs.

Quit

This is the equivalent of the Mops command, "bye." All files are immediately closed, and you are returned to the desktop.

### The Edit menu

Cut, Copy, Paste, Clear and Select All

These perform all the customary editing functions in the Mops window..

### The List menu

Words

Presents a running list of all words in the current dictionary, starting with the word most recently defined (i.e., highest in memory). The name field of each dictionary entry is displayed along with the hex address of the name field. To pause the list, press any key once. To restart the list, press the Spacebar; to cancel the list display, press any key other than the Spacebar.

Objects…

This is not yet implemented in Mops. Eventually it will present a dialog box from which you select the class in memory whose objects you wish to see listed.

Classes…

This is not yet implemented either. It will present a list of all classes defined in the current dictionary in memory. The classes are arranged hierarchically so you can see the inheritance chains of all classes in the dictionary.

### The Show menu

HFS Paths

Displays in the Mops window the current paths which will be searched when you open a file.

Free Space

Displays in the Mops window the amount of memory available for new dictionary entries, as well as the condition of the heap. The Total heap figure is the current available heap if you do nothing to purge modules from it. The Largest block figure represents the largest amount of heap available in a contiguous block if you purged all extraneous blocks from the heap. A typical listing is shown below:

```
Room in dictionary:                223926
Distance to top of hibase range:    61906
Total heap (no purge):             325888
Largest block (purge):             325990
```

Show Module Status

Lists all modules defined in the dictionary in memory, and indicates which one(s) are currently on the heap by printing their load addresses. Modules are locked while executing to prevent their being removed from the heap at an inopportune time. A typical listing is shown below:

## The Utilities menu

Echo During Load

Displays every line of text from a source file as it is being loaded and compiled into Mops. Use this feature in the early stages of program development to aid you in discovering exactly where your bugs are cropping up. By following the load, line-by-line, you can see exactly where Mops runs into trouble and stops the load. Once your code is sufficiently debugged, you can turn off echo to speed up loading. This selection is identical to the Mops command +echo. If you select Echo During Load, a check mark appears next to the menu listing. Selecting it again turns off the feature and removes the check mark. You can pause an echoed load by hitting a key, while quiet loads do not pause to permit type-ahead.

Clear Stack

Clears the stack. This command is also available in Quick Edit.

Clear Window

Clears all text from the Mops window. This command is also available in Quick Edit.

Install

The use of Install has been described in the Tutorial.

Purge Modules

Clears the heap of all modules loaded by your program.

## Communication with Quick Edit

Mops and Quick Edit send Apple events to each other, with a number of benefits.

Of course to use these features, both Mops and QE must be running. If this isn't so, or if you're running an earlier system which doesn't support Apple events, these commands will be ignored.

- If an error occurs during loading a file, Mops sends QE an Apple event asking it to open the source file at the error line.
- The Mops command "edit someFile" will ask QE to open the given file.
- The Mops command "openSource someWord" will ask QE to open the source file containing the definition of someWord. If there's a log file, QE will put the cursor to the start of the definition. If there isn't, QE will search to the first occurrence of "someWord" in the file (which will probably be at the start of the definition anyway, or maybe in a comment just before).

In QE, there is now a "Mops" menu with a number of commands:

- Load
- Save and load
- Save and reload (saves the file, then asks Mops to execute "rl")
- Clear Stack (command-0). Also now available in Mops as command-0.
- Open Source (command-1). You highlight a word, "someWord" say, then choose this command, and QE asks Mops to execute "OpenSource someWord" as described above.
- Clear Window (command-2). Clears the Mops window. Also available in Mops as command-2.

- Defined? (command-3). You highlight a word, then choose this command. Mops will display a message indicating if the word is defined or not.
- Forget (command-4). You highlight a word, then choose this command. Mops will forget down to this word.

# Classes and objects

Building a Mops program is largely a process of defining classes of objects—classes which are the "framework" and objects which are the "movers and doers". In this chapter, we provide you with details of the inner workings of classes and their components, with special emphasis on instance variables and methods. We'll also discuss several Mops words that may be particularly useful in building your own class definitions. You won't need to know everything in this chapter to be successful at building classes, but you should at least survey the information. It may come in handy later, as your programming skills grow.

## Planning your subclasses

Mops comes with many predefined classes—building blocks, which have been designed to be as general as possible. Your application will probably require more specific behavior than the predefined classes are capable of, in which case you will want to define one or more of your own subclasses of existing classes. Your program's unique operations and flavor will be the result of the behaviors you define in your subclasses.

## The class hierarchy

Determining the relationship between a new class and existing ones is an important step in designing a Mops program. The relationship should be guided by the way in which the new class is to rely on instance variables and methods of classes already defined.

A subclass can add new instance variables to those of its superclass, but it can never redefine the original ones. Therefore, a new class should be defined as a subclass of another only if the instance variables of the superclass are needed for objects of the new subclass. If you find that an object of a subclass is not using many of the superclass's ivars, then the subclass should probably be a subclass of a different class.

Methods, on the other hand, can be redefined in subclasses without hesitation. It is practical to carry through the methods of the superclass that apply to the subclass, and then redefine or add new ones where needed to give the subclass its unique properties. Of course, the more methods a subclass inherits, the more compact the code will be.

Multiple inheritance can be a powerful technique if used judiciously. If overdone it can cause unnecessary complexity, but used wisely it can simplify things considerably. Let's say a class MyClass has three superclasses, sup1, sup2 and sup3. In an object of MyClass, the ivars corresponding to sup1 will come first, then those of sup2, then those of sup3. In typical Forth laid-back manner, we don't check for any clashes in method names in the superclasses. When we look up a method name in MyClass, we look at the methods declared in MyClass itself first, then in sup1, then in sup2, then in sup3. If, say, that particular method name had been declared in both sup2 and sup3, the one in sup2 will be used, and the one in sup3 will never be accessible within MyClass.

If more than one of MyClass and any of the superclasses is indexed, we do a check that the specified indexed widths are the same. We give an error if this condition isn't met.

Of course this isn't the only possible approach we could have taken to multiple inheritance. For example we could have had the ivar regions overlay rather than concatenate. We could do this as an option later, if anyone wants it.

For a good example of the use of multiple inheritance to simplify code, look at the class Ordered-Col in file Struct, and its associated classes.

### Choosing between ivars and objects

In addition to designing the class inheritance of your application, you will have to decide what should be an instance variable and what should be a public object. Because an instance variable is invisible to objects other than its owning object, any communication between an ivar and other objects must be passed explicitly through the ivar's owning object. If you find yourself creating numerous "passthrough" methods that are only there to provide access to a single instance variable, you should reconsider your design. It probably indicates that the instance variable should more appropriately be a public object.

A good example of this kind of realization occurred when we wrote the grDemo source file. In an earlier version, the three scroll bars were designed as instance variables of the window. We found, however, that we were sending many messages to the scroll bars by way of the window object. By changing our strategy and making the scroll bars public objects, the program now has more direct communication to the scroll bars with the added bonus of shortening the source for grDemo by approximately 30 percent. Ideally, then, your objects should do most of their communication internally (i.e., sending messages to self, super, or ivars). Keep to a minimum the number of messages that are to be sent between objects. This minimizes inter-object coupling, makes objects more independent, and makes your application more maintainable.

### When to use ivars

But there are times when it makes sense to define instance variables, as we originally tried in grDemo. For example, whenever you find that one object communicates frequently with only one other object, it is likely that one of those objects should be an instance variable of the other. The same holds true when you find it necessary to create objects in pairs (or other multiples)—instead of creating two similar objects, consider creating a third object that consists of two instance variables. If the window in grDemo had been intended as a general-purpose class instead of a one-time application, it would have made sense to keep the scroll bars as ivars, because it would be easier to add the entire window to later applications.

Much of the work that goes into writing a Mops program should be devoted to designing object boundaries. A well-planned application will be much more understandable at the source code level. By clearly defining class functions, a better sense of structure will prevail. This, after all, is what object-oriented programming is all about, and you will probably need to work with objects for awhile before you really fine-tune your ability to create an optimal design. The best design is one in which inter-object communication is minimal and well-defined, reflecting clearly the structure of the problem being solved.

### Defining a class

Now, let's take a closer look at the mechanics of building a new class. A class definition has the following skeletal structure:

```
:class   ClassName   super{ super1 ... superN }  [ n  indexed ]  [ large ]
        [   instance variable names  ]
        [  method definitions ]
 ;class
```

In the above example, the brackets indicate optional sections of a class definition. If you build a class with one superclass that omits all of the optional sections, it will behave in exactly the same manner as its superclass, because you will not have added any ivars or methods to make the new

class any different. ClassName is the name that you assign to the new class. super1 etc. are the names of existing classes that are to be the basic models for the new class. The word indexed, when preceded by a number, defines the width in bytes of each indexed instance variable cell for the new class. An indexed width of 0 indicates that the class is not indexed; if this number is non-zero, the class will require that a number, indicating the number of elements, be on the stack when the class is instantiated (i.e., when you create an object of that class). Thus, the line of Mops code, 3 Array A1, builds an indexed object, called A1, of class Array that has 3 indexed elements.

If you include the word LARGE in your class definition, you are saying that this is an indexed class, and its objects may have more than 32k indexed elements. Indexing operations on LARGE classes will use 32-bit arithmetic, which will slow accesses very slightly (on 68000-based machines only—Plus, Classic, SE). No bounds checking is done, since the CHK machine instruction, which we use for bounds checking, uses only 16-bit index values. For these reasons, don't use LARGE unless you really have to.

Note that you can declare as LARGE a subclass of an indexed class which isn't LARGE. But this isn't really a good idea. Any methods inherited from the superclass won't know that the class is LARGE, so if they try to access the indexed area they won't do it properly. For example, they will execute a CHK instruction, checking the low 16 bits of the index against the low 16 bits of the limit! This for sure won't be what you want. So in this kind of situation, you had better know what you're doing, and only inherit methods which don't access the indexed area.

## Ivars

Next in a class definition come the instance variable declarations, which are simply statements of the form:

```
[ # of elements ]  ClassName  ivarName
```

ClassName here is the class that defines the characteristics of the ivar. Each ivar declaration statement creates an entry in the private instance variable dictionary of the class currently being defined (See Figure 2-1). The entry for each ivar contains fields for the header, data, and a pointer to the class specified by the ivar's ClassName. An instance variable definition is really just a template for the private data of the object. When an object is created, the object's data area is assembled (i.e., memory space is reserved) according to the specifications in the template.

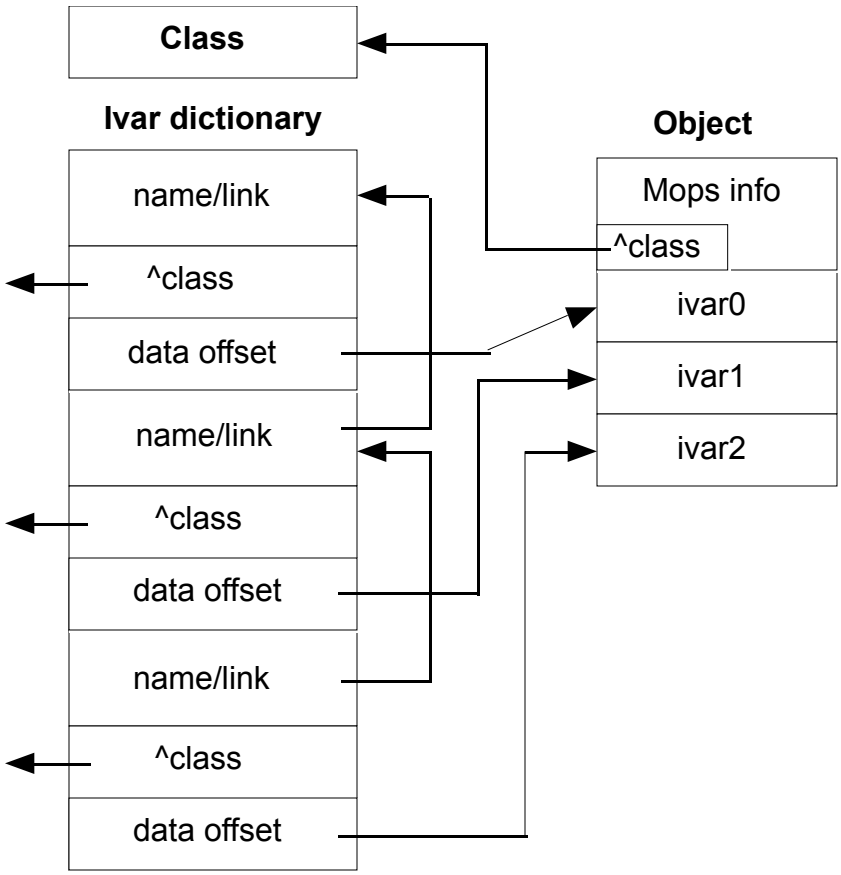



**Figure 2-1—Instance Variables and Objects**

Notice that the "object" in the above diagram has some "Mops info" at the start—as we mentioned in Lesson 5 of the Tutorial, Mops objects have 8 bytes of extra information at the start. Part of this extra information is a pointer to the class of the object, as we show in the diagram. (We 'll give the full details of this extra information in the technical section later.)

Then comes the first ivar, so this is the start of the actual data area of the object.

Instance variables are also objects. The main difference (other than the private/public distinction) is that they can be declared as part of a "record", and in this case their "Mops information" is omitted, as we discussed in Lesson 5.

In a practical example, a Toolbox Rectangle is stored as 4 consecutive 2-byte integers (the x and y coordinates for the TopLeft and BottomRight corners). To pass these parameters to the Mac Toolbox, it is most convenient to map this structure with 4 Int ivars (each 2 bytes wide), using the record {...} syntax, knowing that the 2 bytes of data for each Int will be adjacent to one another.

### ⇒ For Advanced Mops Programmers:

Indexed ivars also have, in addition to the Mops info, a 6-byte indexed header. The data in the indexed header consists of the number of indexed elements and the length (in bytes) of each element. This data is used for range checking.

### Ivars as Toolbox data structures

As you may have noticed in the sample applications in the tutorial, instance variables are very often used as representations of the data structures that the Macintosh Toolbox expects to see when Toolbox calls are made. In essence, the list of ivars creates a mapping between the data

fields in an object and a structure that the Toolbox recognizes. The Toolbox structure might only be a subset of the entire body of data in the object, as it is in the case of class Window.

In defining a new class that calls a Toolbox routine, you will often need to map the layout of the class's data structure to mirror a Toolbox data structure (Toolbox data structures are listed at the end of each section of Inside Macintosh). To map the data such that the Toolbox will be able to use it properly, define all of the Integer or Char fields as Int ivars, and all Long (32-bit) or Pointer fields as Var ivars, and remember to use record {...} around the whole group of ivars.

If there is a section of the data record that you will not need named access to (e.g., data that never changes in the course of a program, but must be in the object's data structure for the Toolbox call), you can save ivar dictionary space by using the BYTES pseudo-class to allocate a string of bytes with a single name. For instance, the following ivar declaration:

```
var   v1
20    bytes junk
var   v2
```

builds a data area that has two 4-byte Vars, v1 and v2, with 20 bytes of data, called junk, between them. The Toolbox will use this area, but the object will never need to access it directly. This is more space-efficient than assigning individual names to a lot of little fields—names that will never be used because the data placed there never is used by Mops. BYTES actually builds an ivar entry of class Object and then reserves a data length equal to the number that you declare. This means that, if necessary, you can get its address with the addr: method. But don't send it a length: message, since this will always return zero (the length of an Object). Note also that BYTES is *not* an indexed data type like bArray—it creates one named field, not an array of bytes.

### How ivars are linked

Figure 2-2 shows the format of an instance variable dictionary entry. It bears some similarity to a standard Mops dictionary entry, except that the ivar name is converted to a hashed value (a compacted form automatically derived from a complex math algorithm). All of the ivar entries for a given class form a linked list back to the root of the ivar chain (see the left column of Figure 2-1). This root is the pseudo-ivar, SUPER (SELF and SUPER exist as instance variables in class META—the superclass of the all-encompassing class OBJECT). The message compiler detects references to these two special ivars, and begins the method search in a place appropriate for each. Therefore, when a new class is being defined, the ^class field of SUPER is patched (directed) to the new class's superclass, and that of SELF to the new class itself. In this way, the search for a given method automatically begins in the proper place for SELF and SUPER references.

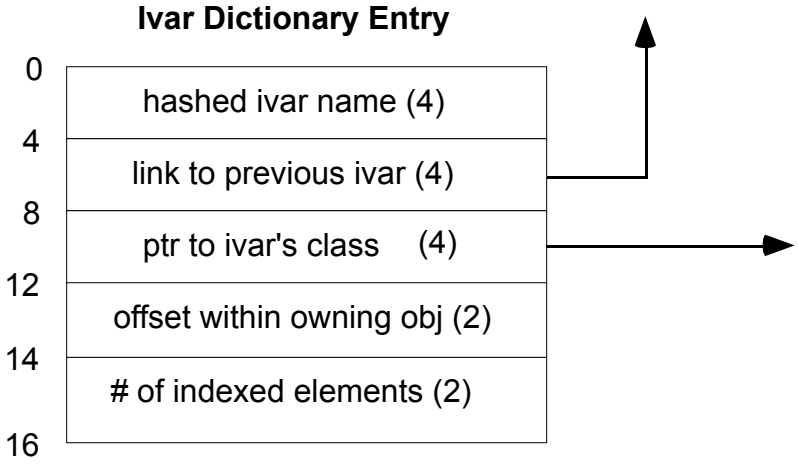



**Figure 2-2—Instance Variable Fields**

### Potential ivar errors

There is an extremely small possibility that you could get an ivar redefinition error when loading a new class, even if the names of the two ivars in question are different. Because the names of the ivars are hashed, two different names could conceivably generate the same hash value. This situation should be extremely rare, since we use a 4-byte hash value. But if it arises, try a different name for one of the ivars.

## Methods

After all of the instance variables are declared, you must write the methods for the new class. A method definition takes the form (here brackets denote optional sections):

```
:m  SELECTOR:  [ { named args \ local vars -- results } ]
        [ method code ]    ;m
```

A valid selector name (any alphanumerics ending in a colon) must follow the :m (separated by at least one space). The selector becomes the name of the new method in the dictionary. A class's method has access to all public objects, to all instance variables of its own class and all of its superclasses up the chain, and to SELF and SUPER. A method can use any previous methods already defined in the current class, and can recurse by simply using the name of the method being defined with SELF as the receiver. Be sure to use SUPER rather than SELF as the message receiver in a redefined method if you want to call the original method in the superclass.

### Special Mops words for primitive methods

You will find that the methods that you write for classes at the ends of long superclass chains consist primarily of messages to ivars or Self, instead of a lot of Mops words or primitive operations. You can see an example of this in the predefined class, Ordered-Col (see Basic Data Structures in Part III), which has several superclasses.

But you may find it necessary from time to time—especially after you've gained some experience defining simpler classes—to define a new class that uses primitive methods involving direct access to the class's data area. An example of this kind of object is class Var in the Struct source file, which manipulates its data directly. Several Mops words described below will come in handy for writing primitive methods like this.

### ^BASE and ^ELEM

You can use ^BASE (pointer to the base address of the current object) from within any method to place the base address of the current object onto the stack. ^BASE copies the machine register A2, which always contains the address of the currently executing object. Note that ^BASE leaves the same address as the phrase Addr: self, or that left by using the name of the object in another word or method. This address points to the data field of the object, which also happens to be the beginning of its named ivar data area.

Primitive methods generally have to get the base address of the data area, and then perform some kind of fetch or store operation at that address. The get: method for class LongWord, for instance, could have been defined in the following manner:

```
:m GET:  ^base @  ;m
```

which fetches the longword (32 bits wide) at the object's data area. For a LongWord this is the entire data area, defined as a single ivar: 4 Bytes Data. The actual definition of get: in class LongWord is somewhat different, since it compiles inline code for speed.

Because a primitive method will often be the code that gets executed after a long chain of nested messages, it pays to make it as efficient as possible. Mops has very fast code operations for its most often used primitive methods.

Indexed classes, for example, have an extensive set of primitives for their most often-used operations. LIMIT returns the maximum number of cells allocated to an indexed object. After the phrase 3 array a1 (creating an indexed object, a1, of class array, with 3 data cells), executing LIMIT within one of A1's methods would produce 3 on the stack. ^ELEM (pronounced "pointer-to-element") expects an index on the stack to begin with, and leaves on the stack the address of the corresponding indexed element; it will invoke an error routine if the class is not indexed. (Incidentally, ^ELEM performs range checking to make sure the index on the stack is within the range of the index.) Another fast primitive, IDXBASE, leaves a pointer to the 0th (i.e., the first element) element of an indexed object or ivar (equivalent to 0 ^ELEM).

Other optimized primitives you should be aware of are those that access 1, 2, and 4-byte arrays. Instead of using ^ELEM, it is faster to use ^ELEM1 for 1-byte elements, ^ELEM2 for 2-byte elements, or ^ELEM4 for 4-byte elements.

Finally, the message width: self will leave the width of an object's indexed elements on the stack.

# More about objects

## Binding

As you know, there are two principal states that the Mops system can be in. When you first start Mops, the system is waiting for your input, ready to interpret whatever you enter at the keyboard or from a load file on the disk. This is known as run state or interpret state, and when it is active, Mops immediately executes whatever word names you enter into the input stream. On the other hand, you may wish to create colon definitions or methods that compile code to be executed later. After the Mops interpreter sees a colon (or :m) and before it sees the next semicolon (or ;m), Mops will be in compile state, and rather than immediately execute the words that it sees, Mops will compile code to call those words into the dictionary. A colon definition is thus a list of calls to other words that will be executed at a later time, when the name of the word is seen in the input stream and the system is in interpret state.

When Mops sees the name of a word and is in interpret state, it attempts to find a string matching the name string of the desired word in the Mops dictionary. A very fast search word, called FIND, does this in the Mops system. When you send a message to an object, such as:

```
get: myint
```

the first thing that happens is that the selector, Get:, is converted into a unique 32-bit code known as a hash value. Then the object name, myint, must be looked up in the Mops dictionary and executed to determine the address of its data. The class of the object myInt is determined, and from that is derived the address in the Mops dictionary of the method that was defined last for that class. This process is known as the binding of a compiled method, and is necessary to determine what code will be processed for that particular message. Note that two different searches must occur before the method can be resolved: the object must be found as a word in the Mops dictionary, and then the compiled method must be found as an entry in the methods dictionary for the object's class.

### Early binding

If we were to enter the above message when Mops was in compile state, the search of both the object and the compiled method would occur at compile time—during the compilation of whatever word or method was being defined. This information would already have been determined by the time that the new definition was actually executed. This is the default manner in which Mops compiles messages, and is known as early binding. Because most of the work of searching is done at compile time, the execution of the message can be very efficient, because it was bound to the actual address of the compiled method in the dictionary.

### Late binding

There are occasions, however, when it is very desirable not to bind the compiled method address when the message is being compiled. Consider, for instance, a situation in which you have a collection of objects that you need to print on the screen. You might have rectangles, strings, bitmapped images, and other objects, all in the same collection. Each of the objects already knows how to print itself by means of a compiled method for Print: that exists somewhere in its

inheritance chain. Your program could be much simpler, however, if it didn't have to explicitly concern itself with the class of a particular object at compile time, but could just send the Print: method to the object and have normal method resolution occur at runtime. This would allow you could store the addresses of the various objects that need to be printed in an array or list without concern for the class of each one.

The technique just described is known as late binding, and is used by Smalltalk and other object-oriented languages as the only style of method resolution. While very powerful and elegant, late binding traditionally has serious performance disadvantages that make most of these languages poor candidates for production of commercial applications. Because Mops provides both late and early binding, you can tailor your application for speed or generality where needed. Even late-bound Mops messages are quite fast, thanks to a highly optimized search primitive. Late binding makes the various objects in your application highly independent of each other, leading to much easier program maintenance. And late binding can greatly simplify the control structure of your code, because much conditional processing can be handled via class differences.

### Late binding and Toolbox calls

Mops itself uses late binding within many of its Toolbox class methods. For example, when the fEvent object (a default even object that is predefined in Mops) receives a MouseDown event from the Toolbox, meaning that the user has clicked the mouse button, fEvent hands the processing of the Click over to the window (or menu bar) that was involved. If the Toolbox tells fEvent that a click was received in the Content region of a window, fEvent sends a Content: message to the window involved. This event must be processed differently according to whether the window has controls, editable text, graphics, and so on. In a conventional C or Pascal program, a large switch/case statement would be required that would handle clicks for different types of windows. In Mops, the differential processing is handled automatically by late binding of the Content: method, because the correct processing will occur for the class of the actual window involved. The programmer is then free to define new subclasses of Window with their own Content: methods, and fEvent can still do exactly the same thing.

### Early vs. late binding

You can cause late binding to occur in a particular message with a very simple modification of your source:

```
get:  myint           \ early binding
get: [ myint ]        \ late binding
```

In the first example, Mops would determine at compile time the class of the object myInt, and in the second example this resolution would happen at run time. If myInt is truly an object, using late binding would be a useless waste of time, because the class of myInt could not possibly change. However, the brackets can enclose any code sequence that generates an address of a valid object at runtime. This can be a single Mops word, a sequence of words, messages to other objects, or anything else. Some examples:

```
(A)   get: [ dup ]      \ message receiver is the object whose address
                        \ was duplicated on the data stack.

(B)   get: [ i at: myArray ]  \ receiver is the object whose address is
                              \ at element i in myArray.

(C)   0 value  theObject      \ create a Value to hold an object address
      myInt -> theObject      \ place the address of myInt in theObject
      get:  [ theObject ]     \ receiver is myInt via theObject

(D)   get:  [  ]              \ receiver is object whose addr is top
                              \ of stack
```

Since the normal use of brackets in Forth is to turn compilation off and on, this particular interpretation of brackets only applies immediately after a selector, and the regular Forth interpretation applies otherwise.

To help avoid (or maybe to add to) confusion, we have added two more ways to specify a late bind:

```
method: **
method: []
```

to bind to whatever is on the top of the stack at run time. method: [] is really the same as method: [ ] with a space between the brackets. It is the same syntax as used in JForth with ODE, for the Amiga.

There are situations where you may want to do a late bind in a loop, but where you know that the binding will actually be the same each time around. We have provided a means by which you can do the binding before entering the loop, and then use the resulting method address in the loop with almost the same speed as early binding. The syntax is

```
BIND_WITH ( ^obj --<selector> ^obj-modified  cfa )
```

If saveCfa and ^obj-mod are values or locals, the usage of this word is:

```
(get object's address)  bind_with someSelector:
-> saveCfa   -> ^obj-mod
(in the loop)  ^obj-mod  saveCfa  ex-method
```

The use of the modified object address is a bit obscure, and is related to multiple inheritance. The method you actually end up binding to may be in one of the superclasses, and the ivars for that superclass may not actually start at the beginning of the object. The modified object address gives the start of the ivars for the superclass, which is the address the method needs to execute. Anyway, if you just follow the above code, you shouldn't go wrong.

BIND_WITH can do a lot to remove the performance disadvantage of using late binding. See the file Struct for some more examples, in the (Col) class.

Another thing you will notice in that class is the usefulness of late binding to Self. This is particularly so with multiple inheritance. (Col) knows it will be implementing subclasses multiply inherited with some kind of array, but it doesn't need to worry about what kind of array. It can simply send messages such as AT: to Self, late-bound, and the right kind of array access will be done. We have even provided an extra syntax to make this operation look neater, e.g.

```
at: [self]
```

Thus the following are all equivalent:

```
aMethod: [self]
aMethod: [ self ]
self  aMethod: **
^base aMethod: **
```

You can take your pick. But in the case of late binding to Self, I think the first one looks the best.

## When to use late binding

You should be able to see that this is a very general and powerful technique. As you become more skilled in building object-based applications, you will find yourself using the power of late binding more and more. The following are some situations in which late binding is particularly useful:

## 1. Forward referencing

You may find it convenient to create code that sends messages to an object that won't be defined until later in the source code. For instance, two classes may need to send messages to each other, meaning that one of them will have to be referenced before it is defined. Cases like this can be easily solved by defining a Value that will hold the actual address of an object at runtime, and compiling late-bound messages using the Value rather than an object, as in example C above. You could also define a Vect which returns the address of an object when you call it. Because these usages are so common, Mops allows you to omit the brackets when using a Value or Vect as the receiver of a message, since in this situation only late binding would make sense. Thus, if we have

```
0 value  ObjAddr
```

the code

```
get: [ ObjAddr ]
```

may be written

```
get:  ObjAddr
```

with exactly the same meaning. Mops automatically compiles a late-bound reference whenever a Value or Vect is used in this manner.

## 2. Passing objects as arguments

Frequently, you will find it useful to pass an object as an argument to a Mops word or method. For instance, the following word computes the difference in the areas enclosed by two rectangles:

```
: ?netArea   { rect1 rect2 -- netArea }
      size: rect1  *  size: rect2  *  -  ;
```

In this example, two named input parameters, rect1 and rect2 are the addresses of objects rect1 and rect2, and are used as receivers of size: messages. This definition compiles exactly the same kind of late-bound reference as if a Value were used. The size: method is looked up and executed at runtime, yielding the dimensions of the rectangle. The area calculation proceeds easily with that information.

## 3. Dynamic objects

These will be discussed in the next section.

## 4. Algorithmic determination of message receivers

Because you can use any code sequence that results in an object address between brackets, you can algorithmically determine which object will be the receiver of a given message. This allows you to traverse a list of objects, sending the same message to each one; it also permits sending a message to an object whose address came from another source, such as a Toolbox call. It might be that the routine itself that generates the object address must be dynamically changed at runtime, in which case you could use a Vect as message receiver. This will compile a late-bound reference in which the Vect is executed first, which in turn executes the Mops word whose xt it holds; this places an object address on the stack that will be the actual receiver of the late-bound message. By changing the contents of the Vect, you can substitute a new algorithm to generate the object address.

# Dynamic (heap-based) objects

Many applications need to create objects dynamically rather than build them into the dictionary at compile time. For instance, an application that handles multiple windows cannot know in advance how many windows will be open at any one time. It would be clumsy to have to prede

fine a number of windows in the dictionary called Window1, Window2, and so on. The best approach in this situation is to create a list of window objects that can expand and contract dynamically. To avoid wasting storage, it is most appropriate to create the window objects on the heap when they are needed, and return the heap to the system when a window is closed.

Heap-based objects are actually based on a subclass of Handle, called ObjHandle. In this class we provide methods for creating and accessing heap objects. A heap object can be created thus:

```
ObjHandle  anObjHdl
' someClass  newObj: anObjHdl
```

Then, to access the object, the method obj: anObjHdl returns a pointer to the object, and also locks the handle so that the object won't be unceremoniously moved while we are doing things with it. Remember to unlock: anObjHdl when finished. So, using the above example, you can access the object thus:

```
mssg1: [ get: anObjHdl ]
       ...
       unlock: anObjHdl
```

When you are completely finished with the object, send release: anObjHdl. This will automatically cause a late-bound Release: to be sent to the object itself, before its storage is released, in case it has some heap storage of its own.

If you know that a dynamic object has one particular class, you can avoid the time penalty of late binding to it, as we'll now see.

## More ways of early binding

### Object pointers

The idea of an object pointer is to provide a convenient way of early binding to an object whose identity is determined at run time (for example, a heap-based object), but whose class we know at compile time. In cases like this we would like the efficiency of early binding to the object.

With an objPtr, the class is associated with the pointer at compile time, then whenever an object address is stored in the pointer there is a check that the class of the object matches. Then after that, sending a method to the pointer actually sends it to the object the pointer points to, with early binding (since we know the class at compile time). An object pointer is a "low-level" entity, rather like a Value. The syntax for object pointers is:

```
objPtr  anObjPtr   class_is  theClass
              ...
( get obj addr to the stack )  -> anObjPtr
              ...
aMethod: anObjPtr
```

Occasionally, the desired class for an object pointer may not be defined at the time the object pointer needs to be defined. In this case, use the syntax

```
objPtr  anObjPtr
```

then after the class is defined:

```
' anObjPtr  set_to_class  theClass
```

This, of course, must precede any code which sends a message to anObjPtr. See the file Dialog+ for some examples—there I had to implement methods manipulating a chain of pointers to objects of the same class as was being defined. For this purpose I put the set_to_class line straight after the :class line, but before the ivars and methods. This is quite allowable.

Note that the address you store in an objPtr must be an **object** address. If you use ' (tick) or ['] on an object in the dictionary, you will get the cfa of the object, which isn't the same. As we saw

earlier, an object has a pointer to its class at the start (it has some other information there as well). To get the object address, which is the address of its first ivar, you just use the name of the object without any selector. Alternatively, if you already have the cfa of the object, use the word >OBJ to convert it to the object address. So, either of the following will work:

```
anObj  -> anObjPtr
' anObj  >obj  -> anObjPtr
```

### class_as>

There's a way to force an early bind to an object, without having to set up an objPtr. The disadvantage is that it's less secure. With earlier versions of Mops you could say

```
( obj addr on stack )  aMethod: theClass
```

with the object's class being used as the "object" to which the method is sent. This syntax was available in Neon, but was undocumented (!). It was, and still is, available in Mops. It can be dangerous if you don't know what you're doing, since there can't be any check on the real class of the object (since it's not known at compile time), and there isn't even a check that what's on the stack is a legal address. If it isn't the address of an object of that particular class, an immediate crash is probably the best you can hope for. But if you know what you're doing this syntax can be very handy.

The only difficulty I have had with it is that in reading code it isn't glaringly obvious that you're not sending a normal message to an object. If your class names aren't well chosen, it might appear that the thing following the selector is an object name, not a class name, which would give that code quite a different meaning. (Of course I'm not talking to you, since you always name your classes in such an unambiguous manner that they just couldn't be anything but classes.)

Anyway for those who do sometimes give their classes less than ideal names, with version 2.6 we have a new syntax for the above operation:

```
<obj addr on stack>  msg: class_as> someClass
```

The old way will still work - I don't plan to delete it and maybe break existing code - but the new way reads less ambiguously (and compiles exactly the same code).

## Public and static ivars

From version 2.5.1, we have provided some extra features relating to ivars. Most simple programs won't need these features, but more experienced Mops users may well find them very useful. (If you're new to Mops, you might want to skip this section.)

Static (or class) ivars belong to the class rather than to an object. They're rather like globals except that they don't clutter the global namespace.

The syntax for accessing them is just the same as for normal ivars.

They're declared like this:

```
class myClass super{ mySuper }
      var   oneVar
static
{     var   someVar
      int   someInt
}
      var anotherVar
```

In this example, someVar and someInt are static, oneVar and anotherVar are normal ivars. In the methods of myClass, whenever you access someVar, you are accessing the SAME ivar, no matter what object you are in, and similarly for someInt.

Public ivars can be accessed from outside the class. They're declared this way:

```
class myClass super{ mySuper }
public
      var   aVar
      int   anInt
end_public
      var anotherVar
      int   anotherInt
```

They're accessed from outside the class via this syntax:

```
msg: ivar> anIvar IN someObject
```

(where someObject is an object of myClass, of course.)

I've considered adding this feature to Mops for some time,doing this for some time, with mixed feelings, but eventually decided it was worth it in some situations where otherwise I'd need to define dozens of pass-through methods. It may not be brilliantly elegant, but it's very practical.

This is really an extension of the public/private distinction which we already have for methods. The default is for ivars to be private, and ivars to be public. You can now use PUBLIC or PRIVATE anywhere in the ivar list or method declarations of a class to change this default. You can use END_PUBLIC or END_PRIVATE to restore the default.

If you combine these two new features, you can get a "public static" ivar. To access this from outside the class, you can't use the above syntax since there's no object to refer to. So the syntax is:

```
msg: ivar> aStaticIvar IN_CLASS myClass
```

# Putting together a Mops application

Once you have a blueprint for the class hierarchy of a program, you're ready to structure the program, actually write the source code, and then assemble the pieces into a self-running Macintosh application. In this chapter, we provide the details for the following steps:

- Structuring the program for keyboard and/or mouse input;
- Creating readable source code files;
- Compiling your code and predefined classes with load files;
- Debugging the program;
- Installing the program as a standalone application.

### Structure of a typical application

In most Mops programs for the Macintosh, a handful of classes will be the primary, high-level building blocks for your application. Into these blocks go the specific processing that make your program unique.

Windows tend to contain the major sections or "mini-applications" within your code. They will contain a number of Views, each of which will handle its own area of the window. The views will probably have a number of subclasses to handle the different kinds of items in the window. Controls are one such subclass of View, and usually determine control paths within a given part of the application, or can be used to provide a more convenient mechanism for setting options. Much of the important code in your application will probably be called via the DRAW: and CLICK: methods of these various View classes.

Menus give the user a means to choose another part of the application or to alter an option setting.

And Dialogs are special-purpose windows that focus the user's attention on a specific choice or set of choices.

## Bringing objects to life

All of the above classes create objects that are recognized by both Mops and the Toolbox.  When your app

got all the parameters and return the appropriate error if not. If True is returned, we assume that the event has been fully handled within the vector routine, and so we return straight to the caller - the code is the result code that gets passed back. The default for these vectors assumes that the Mops development environment is running, and does the appropriate things.

The current Mops setting for some of these vectors may be adequate for your application. As set up in Mops, OpenAppVec simply returns False and does nothing else.

OpenDocVec finds the number of files in the given list of files, and puts the number in the value #DocsToOpen. It then handles each file in the given list of files by opening it using the default file object fFcb, then calling the vector Read1DocVec to read and close it. If this is sufficient for your application, you will just have to redirect Read1DocVec appropriately, as we describe in the next paragraph. As set up, Read1DocVec assumes the file is a Mops dictionary.

Read1DocVec must point to a routine which will read the file designated by the file object fFcb, close it, and return a result on the stack. True indicates the OpenDocVec loop can continue. False means the loop must terminate. You could return False, for example, on an error, or simply if your application can't accept more than one document open at a time.

PrintDocVec is set to the same routine as OpenDocVec, since it doesn't make sense to try to print a Mops dictionary. Your application really ought to do something different.

QuitAppVec is the exception, in that it is NOT called from the QuitApplication Apple event handler. This is because if you try to quit to the Finder from inside an Apple event handler, you'll crash! You MUST return in the normal way from an Apple event handler, or the system won't be pleased with you at all. So what our QuitApplication handler does is set a flag QuitApp? and return. Then back in normal Mops execution, after handling an Apple event, we check if QuitApp? is True, and if so, we execute QuitAppVec. This code is in the source file Event. The current setting for QuitAppVec is simply to call BYE. Your application will probably want to do something a bit more intelligent.

Well, sorry about that terribly long-winded dissertation on Apple events. They do represent a major new addition to Apple's system, and are decidedly non-trivial to handle and describe! I have tried to make the relevant comments in the nucleus source code reasonably full. This is in the file Nuc1.asm.

## Compiling your source

As you write portions of your program, you can load them into Mops (they compile while loading) to let the compiler search the code for errors and to let you fully test how well the code executes. You won't necessarily save the compiled program until a logical section is completed and debugged—once you save a compiled chunk of code, you will no longer be able to edit what is saved. Instead, while you're reworking a section, you should maintain your program as text files and load them into Mops each time you want to test the code.

When you load a typical program, you will be doing so on top of Mops.dic (or MopsFP.dic), which contains a number of—but not all of—Mops's predefined classes already compiled. It is important to understand how source files for your program and the optional predefined classes should be loaded onto Mops.dic. When you loaded the grDemo in the Tutorial several predefined classes were automatically loaded before loading in the grDemo code. This was done by the NEED command at the start of the grDemo file. The sequence of loading is important, but can easily be handled by NEED.

### Switching between compiler and editor

When you compile a source file the first time, you may discover that an error crops up, at which point, the compiler displays a message directing you to the problem area and stops loading. You'll then want to go back into the source file to remedy the problem. This can be done as simply as switching windows to your editor. If you're using Quick Edit, it will have already scrolled your source file to the right place.

Your process of program building will take the following steps:

- Using the editor, load an existing source file or create a blank page for new work.
- After entering a few definitions, you may want to test them. Save your work in the editor (an easy step to forget!), switch to the Mops window and type L (for load), or choose Load... from the File menu, or type Command-L. A dialog box will come up. Select

your source file, and it will load. Alternatively you may type // followed by the name of your source file, if you don't want to have to reply to a dialog box.

- After doing this once, you can reload the same file (having made changes or additions) by typing RL. Mops remembers which source file you are using, via a special dictionary entry which is added automatically when a file begins loading. When you type RL, Mops first does a FORGET back to that point in the dictionary, then loads the file. If you just want to FORGET back to that point but not load the file, type FM (Forget to Mark).
- In the event that you made a mistake in your coding, Mops will report an error of some sort. Switch to the editor window, edit your source file, save, switch back, and RL.

If you have been accustomed to working with a compiled language like C or Pascal, you might be somewhat startled by the immediacy of Mops while you are in the editor. It can speed your development time tremendously to be able to interact with the language as you write, and you should learn to do this often (sometimes it's easy to forget). Frequently, in the time that would be taken to remember something while editing, you could have gotten an answer directly by using the full power of Mops's interpreter.

### Saving compiled programs

You can Save an image of the dictionary at any point during compilation of your source (this is different from installing a finished application, as described later), by selecting Save as… from the File menu. This creates a binary image on disk of that portion of the dictionary from the top of the nucleus up to the last word compiled. Save your work often, because you can always use FORGET or FM to remove any part of the dictionary other than the nucleus. It is good to do a Save just before loading any file that is in a questionable state or in the process of being debugged. Then, if the machine crashes, you need only double-click on the saved image's icon to get right back to where you were.

You can have several saved images on a disk, without causing any problem.

Incidentally, Mops has a powerful file stack facility that allows you to nest loads up to six deep. Thus, a file loaded by NEED can also NEED other files. When this happens, Mops stacks the currently open file (i.e., temporarily interrupt loading of one file) and begins loading the new file. When the second file load is complete, the load of the original, stacked file resumes on the line following the NEED statement.

### Other compiling tips

When loading a file that has never before been compiled, select Echo During Load from the Mops menu to cause each line of the file to be echoed to the screen as it is loaded. If an error occurs while you're watching a file load, you'll have a much better idea of where the problem is.

For files that you know well, disable Echo During Load for a much faster load, but you won't get as detailed messages if an error occurs during compilation. You might then use WORDS to determine the last name loaded into the dictionary—this should be the name of the word containing the error.

After an error, Mops prints the contents of the file stack—the file at the top of the stack is this file containing the error. You can pause an echoed load at any time by hitting the space bar. You can then either continue (by pressing the space bar again), or abort the load (by pressing a different key).

## Debugging your code

You should begin debugging as soon as you have a small section of code that compiles successfully. Start testing the lowest-level words or methods first, so that you can establish a firm base of code that you have confidence in. Call these words interactively (i.e., from the Mops prompt), setting up reasonable parameters on the stack, and then using the .S stack dump to determine if

the results are correct. You can also use the Debugger utility to step through a definition instruction by instruction. We will describe this utility shortly.

### Evaluating Mops error messages

It's quite possible that in the early stages of program development, you'll generate a Mops error during execution or compilation of a word or method. If this is the case, find the error in the Error Handling section of this manual, and try to determine the precise cause in your code.

Frequently, Mops might catch an error that is actually an indirect result of another problem which Mops did not catch. For example, if your code accidentally overwrites the header of a previously defined array, upon execution, the error will point to the array, when, in actuality, the problem is with the errant code. Another example would be a number accidentally left on the stack that doesn't interfere with execution until much later in the program. In cases like these, you must work backwards, tracing the origins of each value on the stack, and seeing if it makes sense. Eventually, you will find the word that is producing an incorrect result, and make a change in the source code accordingly. It can be very helpful to place statements in your code that print out key data values.

### System errors

Sometimes, your code will produce an error that is caught by the Macintosh system before Mops becomes aware of it. In these cases, unless you have Macsbug installed, you will get the "bomb box". The most common system error codes are 2 (when the CPU tries to access an illegal address) and 3 (when the CPU attempts to execute data as code). You'll probably have to reset your Mac if you're not using Macsbug, but if you are using Macsbug (always a good idea when testing code) you may be able to resume by typing:

```
a5=currenta5
g 1e4
```

Sometimes, however, enough "damage" will be done to the heap or 68000 register contents to necessitate a restart, even with Macsbug installed. Therefore, be sure you save your source code files. This will provide you with a safe record of the code that caused the errors.

If you tried to execute a @ or W@ operation on an odd address, such as 2001, you would generate a System Error 2 (only on 68000 CPUs). The 68000 processor has a set of instructions that are optimized for even addresses, and some Mops words use these instructions in their code. Odd address errors can be caused directly in the manner described, but are more likely to result from a different problem that just happens to generate an odd number which then gets used as an address.

Macs with 68030 or 68040 CPUs won't generate odd address errors, but will give a "bus error" if something is used as an address which is outside the range of legal addresses for that machine. As with odd addresses, these are most likely to arise from some other problem which leads to something being used as an address which isn't really an address. Note that we store an illegal address value, which is also odd, in nil handles and pointers. Thus any attempt to use a nil handle or pointer will give either an odd address error or a bus error depending on the CPU.

Any fatal system error is best tracked down by first finding the precise location where the error occurs. Do this by testing words interactively, and then reasoning out why the offender isn't working properly.

A System Error 28 means that the system stack (the Mops data stack) has grown down into the top of the heap. Because Toolbox routines use the system stack for their data storage, stack overflow can occur if a deeply nested Mops word calls a Toolbox routine that uses a lot of stack. You can use the Install utility (described in the Tutorial) to adjust the proportions of heap available for the stack and the dynamic heap.

Some errors may cause the machine to lock up, make strange sounds, or break up the video. In these cases, the code has destroyed something essential to the operating system before either Mops or the Macintosh Operating System could detect it. The only choice here is to reset the Mac and try to determine where the code is going wrong. You might want to scatter ." messages through your code, which can print values and strings to keep you posted on where the code is executing at a given moment. This will help you narrow down the location of a problem fairly quickly.

## Your application's icons

We have already described the use of the Install utility in the last Tutorial lesson. Here we will discuss how to give your application and its documents their own icons.

There is a complex interplay of resources within a Macintosh application that describe an application and its icons to the Finder. You may want to read 'Structure of a Macintosh Application' in Inside Macintosh before proceeding, especially if your application manipulates its own document files. We will describe here only the steps that relate to Mops and your application.

When you first use Install to create your application, it has no icons. You can use ResEdit, Icon Edit or various other utilities to create icons. If you give them the expected resource IDs, you can paste them straight into your application using ResEdit. The expected IDs are 128 for the application itself, and 129 to 132 for the various document types which your application uses. Your icon editing utility will have a way of specifying the resource IDs of the icons.

Assuming you have created a resource file using one of these utilities, and that you have given the icons the right IDs, do the following:

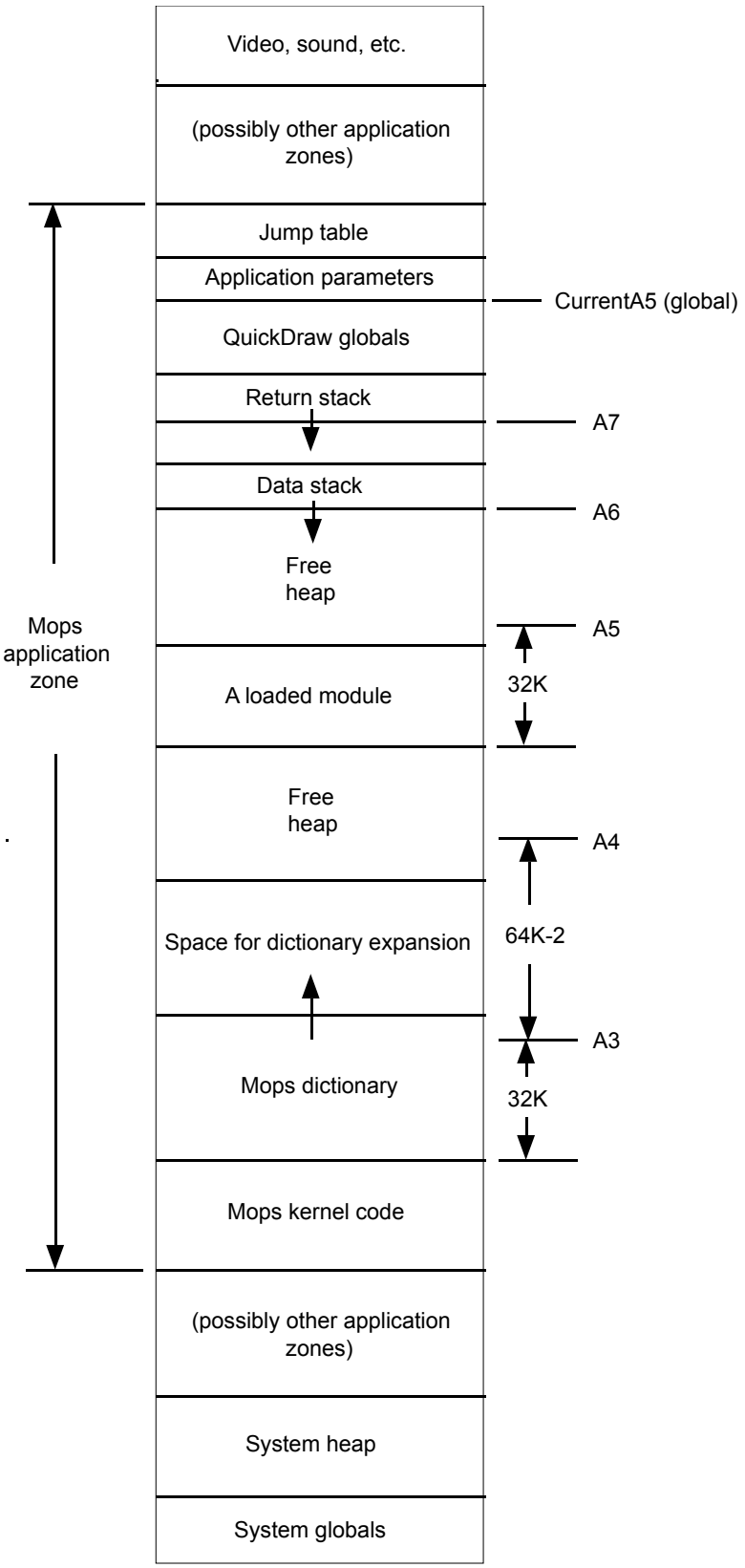
1. Start ResEdit, then open both your new application and your icon resource file.
2. Select your icon file, and do Select All (Command-A), then Copy.
3. Select your application.
4. Do Paste, then Save.
5. Quit ResEdit.

Your application may not appear with its new icon immediately, since the Finder keeps icon information in its own "desktop file". If you close and then open the window containing your application, this may cause the Finder to recognize the new icon.

# Memory Organization

We will now move on to some of Mops's more advanced capabilities. Some of the ideas and terms we'll be describing are more fully explained in Inside Macintosh, and we'll direct you to the appropriate IM sections when necessary. Finally, if you wish to gain deeper understanding of the specialized compiling words that operate inside Mops (derived from the Forth language), we suggest you read one of several commercially available Forth texts listed in the references at the end of this chapter.

Because of the way the Macintosh manages memory, Mops has several distinct areas in which it stores data. The following diagram gives a typical picture of the Mac's memory while a Mops program is running. If you're new to Mac programming, don't worry about what everything in the diagram means at this stage—we'll describe the main things you need to know as we go along.




# Figure 5-1—Mops Memory map

## The Mops Dictionary

Near the beginning of this manual we said a Mops program builds a dictionary of words. Each word and its definition occupies a portion of the computer's memory. A Mops definition consists of several parts, including information like whether a word is a value or a colon definition, the numbers or other data associated with the word, and machine instructions which carry out the operations specified for that word. In Mops, the areas reserved for those parts of a definition are called fields.

Most Mops words have four fields that you should be aware of:

```
link field
name field
handler field
code field
```

and they look like this in memory:

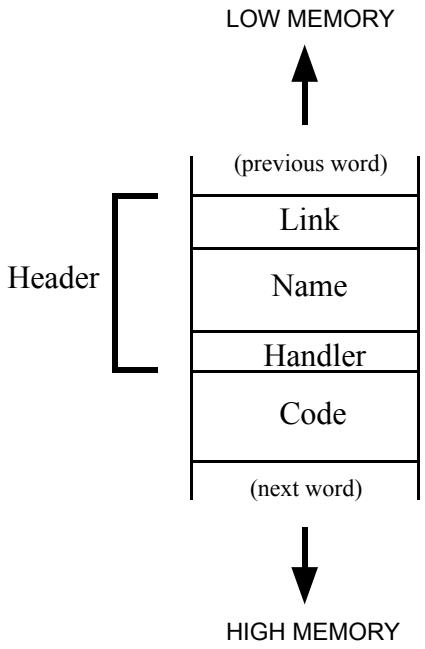



**Figure 5-2—Fields in a Dictionary Entry**

The link, name and handler fields are usually grouped together as the header field. The name field holds the actual name of the word. Its length varies with the length of the name.

The link field helps Mops programs compile quickly. In the link field is the address of an earlier word in the dictionary. This facilitates the search through the dictionary each time you type a previously defined word. The search starts at the most recently defined word (the word nearest high memory). If there is no match in the first word, the search looks to the link field for the address of the next word on which to attempt a match, and so on backward through the dictionary. The length of the name and parameter fields can change from definition to definition so there is not a fixed memory interval between words.

The content of the handler field specifies whether the word is a colon definition, a value, and so on. The handler field usually contains a negative integer, unique for each word type. When the word is compiled, the handler field is used to transfer control to the right part of the Mops compiler to compile words of this type.

For normal colon definitions, the code field contains executable code—the code that was compiled when the definition of that word was read by Mops. If you later type the word name at the keyboard, Mops looks at the handler field, sees that it is a colon definition, and transfers control

to the beginning of the code field. Your compiled code is then executed. At the end of the definition, where you had put the semicolon, there is a machine language "return" (RTS) instruction. This causes control to return to the Mops interpreter.

If, instead of typing your word at the keyboard, you put it into another definition, the process is rather similar, except that instead of calling your definition directly, Mops compiles a machine language call instruction at that point, to do the calling. Then, when the later definition executes, and that call instruction is reached, your earlier definition is called.

For Mops word types other than colon definitions, the code field doesn't necessarily contain code. For example, for values and constants it contains the actual 4-byte value. In this case we call this field the "data field", since it contains data. But it is really the same field by a different name.

## The kernel or nucleus

In the diagram, you'll see that above some memory areas used by the Mac system is the Mops kernel code. The kernel (also known as nucleus) is the lowest, most elemental part of the Mops dictionary—that part of Mops without any predefined classes. It is what is loaded into memory when you double-click the Mops application icon itself. Saved Mops images, such as Mops.dic, appear to the Finder as documents with Mops as their owner. Therefore, when you open a saved Mops image, the Finder starts the Mops kernel as the application, and passes the name of the saved image file to the Mops kernel as a parameter (see Inside Macintosh for more detail on this). The Mops kernel then determines whether the file is a valid Mops image file, and, if so, loads it in an area of memory above the kernel. (This way of doing things is very efficient for development, since the image saved is the document you have been working on. You may say many variations of this document without changing the application itself)

The memory area dedicated to the Mops kernel and other specifics of your program is known as the application heap, or simply the heap.

## The heap

The heap is a region of memory that can be divided into smaller sections, called blocks. When a program needs some memory temporarily, it can ask the Macintosh Memory Manager for a block of heap, and later give it back when it is done. This is what the Mops kernel does at startup in order to acquire memory for future expansion of the Mops dictionary. Mops requests a block of heap that will be large enough to allow for expansion of the user dictionary, but will leave enough room for both the system and Mops to use on a temporary basis (dynamic heap) as the program executes. Exactly how much memory is to be devoted to the dynamic heap can be altered by the Mops Install utility, which was described in lesson 21 of the Tutorial.

Many parts of Mops and the Mac Operating System rely upon dynamic heap. For instance, when your application requires a resource, such as a font, to be loaded from disk, the Font Manager places the font in the dynamic heap. Mops modules (described below) are loaded into the dynamic heap, and any class can be told to create an object whose data exists on the dynamic heap instead of in the dictionary. Whenever you see a System Error 25, it usually means that heap has become used up or fragmented (see the Memory Manager section of Inside Macintosh). You can remedy this situation either by leaving more dynamic heap with the Install utility or by being more careful to release the heap used by your application's modules or heap objects once they are no longer needed.

The Mops dictionary can grow until it exhausts its allotted block of heap. The Mops word ROOM will return the amount in bytes of available dictionary space at any time.

## Mops stacks

The two Mops stacks—data (parameter) and return stacks— are allocated above the application heap. Both grow downward: the return stack grows towards the base of the data stack. The

Mops kernel allocates room for 1500 32-bit cells on the return stack. The data stack is limited only by the maximum address to which the heap is permitted to grow, and is allocated 50000 bytes in the distributed Mops system. Macintosh Toolbox routines allocate their local variables on the stack, which accounts for the relatively large size of the data stack.

Various system errors can be caused by one of the stacks growing beyond its bounds. This type of error can be difficult to detect, but you should be particularly alert for return stack problems when writing recursive routines. The return stack also gets heavy use when you use a lot of named parameters and/or local variables in your methods or Mops words, since the previous values of these quantities are saved on the return stack.

The data stack is used not only by Mops, but also by the Toolbox when you invoke one of its routines. Toolbox routines allocate their local data and do parameter passing on the system stack, which is actually the same as the Mops data stack. This makes the Mops/Toolbox interface fairly easy. A Mops word need only place parameters on the data stack, just as if it were about to execute another word or method (also see the later chapter "Calling the Toolbox"). If the Toolbox routine calls many other routines, the system stack could potentially grow into the application heap. The Macintosh might catch this problem with its "stack sniffer" routine, in which case you will see the dreaded System Error 28. Alternatively, the Macintosh could begin producing a bizarre sequence of sights and sounds caused by overwriting of some heap data before the stack sniffer could catch it. In cases like this, you either must give Mops more stack using the Install utility, or adjust your algorithm to nest less deeply on the data stack.

## Addresses—relocatable and absolute

For speed, we normally hold all addresses in the normal Mac form (which we'll call absolute since its the actual address which is directly used by the hardware). This does mean, however, that we have to do some juggling to handle addresses that are stored in the dictionary and then saved in a dictionary image which is reloaded later. In general, these addresses won't be valid any longer, since the program may well be located at a different place in memory.

For this kind of operation we have defined a relocatable address format, and the words @abs and reloc! to respectively fetch and store a relocatable address with conversion to/from absolute. We have also provided two classes, DicAddr and X-Addr which use the relocatable format internally. (DicAddr is for addresses of data, and X-Addr for executable word addresses.) These have access methods that incorporate the conversion. **You should always use one of these mechanisms for accessing relocatable addresses.** Don't rely on any details of the relocatable address itself, as this may change at any stage in the future, and probably will. It will stay at the same size as a stack cell, but we make no other guarantees.

Note that relocatable conversion does not need to be done nearly as often as @, w@ or c@, so that we really do gain by standardizing on absolute addresses.

## Handles and pointers

When you allocate a block of memory in the heap, you can ask for either a Handle or a Pointer. A pointer points directly at the memory you allocated, while a handle points at what is called a Master Pointer, which is what actually points to the memory. See the Memory Manager chapter of Inside Macintosh for more details on handles and pointers. But briefly, using a handle allows the Memory Manager to move the block of memory around, and thus allow memory to be used more efficiently. When the Memory Manager moves a block of memory, it updates the master pointer so that programs will still know where the block of memory is. However, blocks accessed via a pointer can't be moved, since the Memory Manager has no way of knowing where the pointer (or any copies of it) are located—the program may have put them anywhere.

In Mops, we encourage the use of handles rather than pointers. (Apple thinks this is a good idea too.) Both handles and pointers are objects, with appropriate methods defined for them. If you want to do a number of operations quickly on a block of memory allocated via a handle, you may

lock it in memory so the Memory Manager won't move it while you are accessing it. You may then retrieve the actual address of the block, and know that it will remain valid until you unlock the block. The methods lock: and unlock: of class Handle perform this function.

An accidental clobbering of a handle or pointer can produce a bug that can have very nasty and generally unrepeatable results, and be very hard to track down. Making them objects helps discourage doing dangerous things with them, and also allows a degree of error checking. An unallocated handle or pointer object is given a value which should always cause a trap if it is used as an address. (The actual values we use are $ FFA00101 for unallocated handles, and $ FFA00103 for unallocated pointers. These two values are defined as constants with the names NilH and NilP respectively.) These are illegal addresses on all Macs, as far as I am aware, and give a Bus Error if used.

You may define objects which are allocated on the heap, with a handle pointing to them. See the section "Dynamic Objects" below.

# Strings in Mops

## String types

There are three main ways of representing strings in Mops: as STR255 strings, as addr-len format strings, or as objects of class String or String+.

When WORD parses a string, it places at HERE (the next available memory location above the dictionary) a byte representing the length of the string, followed by the text of the string. This is the same representation as that used by the Toolbox for the str255 data type (see Inside Macintosh). When passing strings as parameters on the stack and manipulating them, however, it is usually most convenient to use two cells to represent the string as ( addr len -- ). The word COUNT accepts an address of a str255-format string and returns its ( addr len ) representation. Conversely, the word STR255 converts addr len to str255 format, returning the address of the length byte to facilitate Toolbox calls.

Mops preserves a special 256-byte buffer expressly for conversion of addr len to str255 format, and this buffer's address is left on the stack by the word BUF255. You can use this area occasionally as a temporary workspace for other operations, provided you don't interfere with routine string processing. Note that STR255 allows you to have only one string at a time. For Toolbox calls that require multiple strings, you will have to use the word >STR255, which accepts an arbitrary address for setup of the string.

## String literals and constants

A Mops string literal is a quote followed by one space and the text of the string, immediately followed by another quote:

```
" Harold"          \ leaves ( -- addr len ) of string
```

Thus, " Harold" TYPE would print the word Harold on the screen. You should use the string literal whenever you have a single occurrence of a string.

String constants (SCON), on the other hand, are useful when you need to use a string several times in your code. You assign a name to the string, and then use the string name for operations with that string, as follows:

```
scon  harry "Harold"          \ assign name harry to string
harry  type                   \ prints "Harold" on the screen
```

**Note that in this case you don't put a space between the " and the first character of the string.** In the first example, " was a word, and so had to have a space after it. In the second example, " isn't a word, but just a character being read by the word SCON. In fact you can use any character as a delimiter, which is useful if you need to include a " within the string itself. The first non-blank character after the name of the scon is the delimiter, and the string consists of all the characters up till the next occurrence of the delimiter. Thus you can have:

```
scon  3quotes  /"""/
```

Using the name of the scon in your program leaves ( addr len ) on the stack, and compiles into a single xt of the dictionary entry for the constant name. Both string literals and constants are fixed strings that once defined, cannot be changed.

For heavily text-oriented applications, a more suitable approach would be to define the text strings as Resources (see section on Using Resources in Mops).

## Other string techniques

For string variables, use an object of classes String or String+, discussed in Part III of this manual. These classes are useful for building strings dynamically, finding substrings, and searching within strings.

In the Tutorial we described the word & which compiles a single literal ASCII character. This can be used either inside or outside a definition. There are also the ANSI standard words CHAR and [CHAR] which perform this function. Outside a definition, the phrase:

```
char A
```

places the ASCII code for 'A' (65) on the stack. Inside a definition, the phrase

```
[char] A
```

compiles a literal for 'A', and places the value on the stack at runtime. And as we've seen, you can use

```
& A
```

either inside or outside a definition, which is probably more convenient (although not ANSI).

There are several primitives for getting a string from the input stream (either keyboard or disk) into the dictionary. WORD gets the next word, without embedded blanks, and moves it to HERE, then returns that address. @WORD is a useful word that takes the place of the frequent phrase BL WORD. For strings, WORD" reads a quote-delimited string from the input stream, moves it to HERE, and returns that address. MWORD ("Mops word") executes BL WORD, and maps the word to upper case. It is used by Mops itself when interpreting or compiling.

# Calling the Toolbox

Because Mops' data stack is the Macintosh system stack, calling stack-based Toolbox routines is fairly easy. You should read the Inside Macintosh section on Using QuickDraw from Assembly Language to understand how Toolbox routines use the stack, but we will give you a brief overview here.

### Toolbox data cells

Toolbox routines expect parameters of two types on the stack: 16-bit and 32-bit. Since all Mops parameters are 32-bit, you might occasionally have to convert a Mops cell to a 16-bit Toolbox cell. As we saw earlier, both Mops stacks grow towards low memory. Moreover, the stack pointer always points to the high-order word (a word is 16 bits) of the top 32-bit element. To convert a 32-bit value to a 16-bit value, we need only add 2 to the stack pointer, which will advance it to the low-order word. There is a Mops word called MAKEINT that does this for you. It gets its name from the fact that the Toolbox Integer data type is 16 bits long. Conversely, you can convert a 16-bit element to a 32-bit cell in two ways. For unsigned integer values, you can simply use WORD0 to push 16 bits of 0 to the stack, creating a 32-bit cell with its high-order word 0. For signed integers, use I->L, which converts a signed integer to a signed long (32-bit) value. This will produce a high-order word of $ FFFF if the integer is negative. The INT: method for Mops INT objects will return their value as 16 bits.

### Toolbox data types

Three principal Toolbox data types use a 16-bit representation: Integer, Boolean, and Char. Other data types are either stored in a 32-bit cell, or you pass a 32-bit pointer to a longer data structure. Any composite structure longer than 32 bits uses a pointer when passing it as a parameter.

When a Toolbox routine requires a VAR parameter, this is a call by address, and must use a pointer to the actual data structure, even if it is an Integer or a Long. This is because the Toolbox will actually change the value of the parameter, and needs its location to do so.

Certain calls may require two 32-bit cells to be packed into two 16-bit cells. For example, to convert a Mops Point to a Toolbox Point, you use the Mops word PACK. UNPACK does the opposite operation, converting two 16-bit values to signed 32-bit values.

### Procedure and function calls

Toolbox routines are for historical reasons defined in terms of the Pascal language. They can be either

Procedures or Functions, depending on how the Toolbox responds to their calls. In C terminology, we would say that they either return a result or don't.

Procedures don't return any result on the stack. To set them up, you just push your parameters on the stack in the order that they are listed. Functions, on the other hand do return a value of either 16 or 32 bits on the stack. They are like primitive Mops words that can only return a single cell. To set up a function, you must first make room for the value returned, using WORD0 for a 16-bit value or 0 for a 32-bit value. Then push the parameters onto the stack as you would for a

Procedure. Finally, after all the parameters are set up, you can use CALL followed by the name of the routine. You can find the name from the list of Toolbox calls in this manual or from Inside Macintosh—the names are exactly the same. CALL, when compiled, invokes a module that contains the hashed values of all the Toolbox routine names, and compiles a trap number that does the actual work of the call. Here are examples of typical Procedure and Function calls, first with the (Pascal) definition as in Inside Macintosh, then with the equivalent Mops code:

```
PROCEDURE InvertRndRect (r: Rect; ovalWidth, ovalHeight: INTEGER);

     rect  myRect            \ create a rectangle object

     addr: myRect            \ get a pointer to the rectangle
     rWidth rHeight  pack    \ pack two 32-bit values into two integers
     call  InvertRoundRect   \ call the Toolbox routine



FUNCTION  GetNewWindow (windowID: INTEGER; wStorage: Ptr;
      behind: WindowPtr) : WindowPtr;

     0                       \ make room for returned WindowPtr
     int: resID              \ get this window's resource ID
     ^base                   \ addr of this object for window
     -1                      \ in front of all other windows
     call GetNewWindow       \ call the Toolbox routine
     put:  windowPtr         \ save the returned window ptr
```

## Accessing system variables and constants

You may sometimes need to access system variables, which exist in particular memory locations. These are called "low memory globals", because they are in low memory, and they are accessible to all programs. These are all given particular names by Apple. Likewise there are many system constants which Apple define by name.

There is a Mops module that looks up these names for you, so that you can put the name in your program without having to worry about the actual value. You can access globals thus:

```
global <name>
```

and the address of the appropriate global will be pushed at run time. Thus,

```
global Ticks @
```

will yield the current number of ticks.

You can access system constants thus:

```
konst <name>
```

and the value corresponding to <name> will be pushed at run time.

# Modules

Mops provides a facility for creating separately compilable, relocatable modules. Modules encourage you to separate your program into well-defined, independent units, which makes your code easier to understand and maintain. You must explicitly state which definitions from a given module will be available (exported) to callers outside the module. Any other definitions become unavailable to the rest of the program after the module is compiled. Modules are loaded automatically on the heap whenever one of the exported definitions is referenced. The application must manage and release any modules that are no longer needed.

### Module guidelines

Certain guidelines must be observed when dividing your application into modules.

Exported names are taken to be executable words. Therefore you mustn't export other things such as objects or Values. If you need to export an object, you will need to define a word in the module which gets the address of the object (just naming it will do that), then export that word name. You can then use late binding to send a message to the object.

You may export class names. Thus, you can define a class in a module, include the class name in the imports list, and instantiate objects of that class anywhere. In fact in the Mops system itself, we handle windows, menus and dialogs this way. All the methods of the class are in effect exported along with the class name; you don't have to take any special action, apart from putting the class name in the imports list, and taking care with action handlers (see below). Whenever you send a message to an exported class, the module will be invoked automatically.

Naturally this has some performance implications. There is a fair amount of overhead involved in invoking a module. You can reduce this to some extent by locking the module over a number of calls (this is true for ordinary exported words as well). But if message execution for a particular class is really time-critical, it would really best to leave the class in the main dictionary. Where exported classes are most useful is for those classes that depend heavily on Toolbox calls for most of their methods (such as windows, menus and dialogs). Toolbox calls are generally much slower than Mops module invocations, so the extra time penalty of putting the class into a module won't be significant.

In Mops you can enter modules only through the exported words (otherwise a machine base address register will not be set correctly, and you'll crash). Therefore a module should not store the address of one of its internal words in a vector in the main dictionary. Even if the module is locked in memory, you would not be able to execute this vector. We have included a check on stores to vectors, that a module address isn't being stored outside the module, so as to give an error message if this is attempted. This check can be turned off, but don't do it unless you are absolutely sure you know what you are doing!

## How to use modules

Each module has a corresponding object in the main dictionary, which is used for interaction with the module, and which conceptually *is* the module. These module objects have appropriate methods to load them, purge them, query their status etc.

## Creating modules in Mops is a three-stage process:

1. Create a definition for the module and the entry points that are to be available to the rest of the application. This must exist in the resident portion of the application, and has the following format:

```
FROM MyMod IMPORT{ word1 word2 word3 }
```

This statement declares a module, MyMod, from which will be imported three definitions, word1, word2 and word3. These two names will exist only in the disk image of the module until one of them is referenced, at which time the entire module will be loaded into the heap. On the disk, the binary image of the module will have the name MyMod.bin.

2. Write the source code for the module in a separate file, called whatever is the name of your module, followed by ".txt". Thus in the above example the source file would be called MyMod.txt.

3. The module must be compiled and saved in its binary format before it will be available to callers. To compile a module, you must send a compile: message to the module, e.g.

```
compile: MyMod
```

When the module has been compled, a message will appear stating that the module has been saved. You must have room in your dictionary to load the module source file.

When we compile a module, we temporarily hide all of the main dictionary above the module object itself. This means you don't have to worry about what you might have loaded in the dictionary when you want to recompile a module.

If you need to call a module several times in succession, you can save some run-time overhead by locking it in memory while you are making the calls. Do it thus:

```
lock: MyMod
```

You will also need to lock the module if you get the address of somewhere in the module—an object, say—and need to use the address again later. If the module isn't locked, it may move in the heap in the meantime or be removed altogether, so that the address won't be valid any longer. When you are finished with the module, remember to unlock it thus:

```
unlock: MyMod
```

When you are completely finished with a module, you can release its heap memory thus:

```
release: MyMod
```

You do not have to do this, however, since loaded modules which are not actually being executed will be released automatically if more heap space is needed. If you want a module not to be released from memory, you can send the message

```
keep: MyMod
```

This will flag a module as needing to be kept in memory. Unlike the situation with lock:, the module may be moved within the heap by the Memory Manager, if it isn't actually being executed. To undo a keep:, send

```
drop: MyMod
```

This will not release the module from memory, but will once again allow it to be released if more heap space is needed for something else.

To include a module in an installed application, mark it as installable any time before calling install. Do it like this:

```
true  setInstall: MyMod
```

# Miscellaneous Topics

## ANSI standard

Mops is fairly close to the ANSI Forth standard. If you load the file ANSI, the remaining differences are dealt with, and you should, we hope, have a conforming ANSI Forth system, implementing the CORE word set, the ERROR and ERROR EXT words, and most of the CORE EXT words. Thus ANSI standard Forth programs using these word sets should be able to run under Mops.

Note that Mops selectors are not consistent with the standard, since under the standard, word names ending with colon must be normal Forth words. We have therefore provided a value SLCTRS? which if set to false will disable selectors. The file ANSI sets this flag false. Set it back to true if you need to.

## Local sections and temporary objects

Local sections are an extension to the named parameter/local variable scheme. Local variables are so useful, that there could be a tendency to make definitions too long, simply because you need to keep a number of local variables around. Local sections remove this problem, since they allow local variables to have a scope which extends over several definitions. Within a local section, all words have access to the named parameters and locals.

You begin a local section with the word LOCAL. The syntax is e.g.:

```
LOCAL  LocName  { parm1 parm2 \ loc1 loc2 loc3 -- }
```

The word LocName is the "main" word of the local section, i.e. the one which takes the named parameters and whose entry causes any locals to be allocated. It must be the last word defined in the local section, using :LOC and ;LOC in place of : and ;. LOCAL doesn't start the definition of this word; it functions like a forward definition. It is really a means of making the names of the parms and locals available to the compiler at the start of the local section. All the words defined between LOCAL and the ;LOC at the end of the main word, can access the parameters and locals. These words can be called freely from other definitions within the local section, but must not be called from outside, for obvious reasons.

We also have local sections for methods, which work in a similar manner. You declare a local section for methods with MLOCAL instead of LOCAL. :MLOC commences the definition of the "main" method, and ;MLOC ends the definition of that method, and ends the local section.

Temporary objects are rather like local variables. They could also be called "local objects" but the word "local" is being used for enough things already, so we're calling them "temporary". It's the same thing, though. They are normal objects in all respects, except that they only exist within one definition (or local section), and have no storage allocated otherwise. The syntax is as in this example:

```
: SomeWord
temp{ int         anInt
      var         aVar
```

```
      string      aString  }

      123 put: anInt
      " hello"  put: aString
      ...
;
```

You can also use the syntax

```
      temp { ... }
```

with "temp" and "{" as separate words, if you prefer. I tend to use both, depending on how many temporary objects I'm declaring, and how I want to format the declaration.

As you can see, within the definition you can use the temporary objects in exactly the same way as normal objects. They are actually allocated in a frame on the return stack. However you can use >R etc freely in the definition, since I keep a separate frame pointer. Of course, the temporary objects get a classInit: message automatically when the definition is entered and their space is allocated. They also get a release: message automatically when the definition exits (either at the semicolon or via EXIT). Thus if you use a temporary string as in the above example, you don't have to worry about sending it release: to get rid of its heap storage at the end of the definition.

As with local variables, if you call a definition recursively, you will get a fresh copy of any temporary objects.

The syntax for temporary objects within a local section is exactly as you would expect:

```
      LOCAL  localName  { parm1 parm2 \ loc1 loc2 }
      temp{  int1
             var1 }
```

The local variables here are entirely optional. A local section can have either local variables, or temporary objects, or both. (Not much point in having neither!!)

## Case statements

We provide no less than three different flavors of case statement in Mops; each of these is most suitable in a different situation.

The first is the Eaker model, and was described in the Tutorial. We'll use expr1, exprn2 and so on to mean any code that leaves one result on the stack.

```
expr1
CASE  expr2  OF  some code    ENDOF
      expr3  OF  some more code     ENDOF
      default code comes here
ENDCASE
```

This form of case construct compiles directly to a set of equivalent simpler operations:

```
expr1 expr2  OVER =
IF    some code
ELSE  expr3  OVER =
      IF    some more code
      ELSE  default code
      THEN
THEN
DROP
```

As you can see from the equivalent Forth code, If any of expr2, expr3... matches expr1, the associated code is executed and then control passes to after the whole case construct. If none match, the default code is executed. Note also that right at the end, a DROP is done to get rid of

the expr1 value—this means that the default code can be left out completely without leaving a spurious value on the stack. But if you consume the expr1 value in the default code, remember to put a dummy value on the stack to be consumed by the DROP. Here's a (rather useless) example:

```
CASE  10    OF          ." ten"                            ENDOF
      12    OF          ." twelve"                         ENDOF
      13    OF          ." thirteen or sixteen"            ENDOF
      16    OF          ." thirteen or sixteen"            ENDOF
      20 30 RANGEOF     ." twenty to thirty inclusive"     ENDOF
      ( default )       ." something else, namely "  .
      0     ( to be consumed )
ENDCASE
```

Notice that the ENDCASE consumes one value off the stack, and also that there's no easy way of handling different values which lead to the same action (as in 13 and 16 above).

The second type of CASE we have in Mops is a keyed case, in which a test value is compared to succesive values in a linear list. Here's the equivalent of the above example:

```
CASE[  10  ]=>          ." ten"
   [  12  ]=>           ." twelve"
   [  13  ], [ 16 ]=>   ." thirteen or sixteen"
   [ 20  30  RANGE]=>   ." twenty to thirty inclusive"
   DEFAULT=>            ." something else, namely "  .
]CASE
```

This format will compile to more compact code than the former example, and should execute significantly faster. The former CASE syntax has the advantage that the test values are computed each time, so can be different on different executions, if this is what you want. If, however, your test values don't change, which is more likely, the latter CASE[ syntax is better to use, since the test values are obtained at compile time and compiled once and for all into the code. As you may gather from the syntax, compilation is turned off and on when obtaining the test values, so that for example you could put `CASE[ value1 value2 + ]=>` etc. The arithmetic will take place at compile time.

You can also see that there's a straightforward way of handling different values giving the same action. You can also use `RANGE]`, with the expected meaning.

The third type of case we have is an indexed case. In this kind of case, a direct table lookup is done to determine the outcome. Here's our example again, with one change:

```
SELECT[ 10 ]=>          ." ten"
     [ 12 ]=>           ." twelve"
     [ 13 ], [ 16 ]=>   ." thirteen or sixteen"
      DEFAULT=>         ." something else, namely "  .
]SELECT
```

Notice that the syntax is almost the same as for the keyed case, but that there's no equivalent of the range test. This is precisely because it uses a direct table lookup rather than a series of comparisons. Later we might implement the filling in of a range of entries in the table, in which case we could implement the range test. But we haven't done this yet.

This code is the fastest of all to execute, since one direct lookup is done on the table. However in some situations this construction may take up too much space. The table which is generated will contain two bytes for every value in the range between the highest and lowest test values. Thus in this example the table will have an entry for every integer between 10 and 16 inclusive, i.e. 7 entries, which will take 14 bytes. This would be fine, but in many situations the table would be

enormous. Mops will assume that an attempt to build a table of more than 500 entries is an error, and give a message.

## Recursion and forward referencing

You may occasionally wish to call a word from within its own definition (this is called recursion). At first glance you may think that the logical thing to do would be to simply use the word's own name. (This was actually the way things were done in Neon.) However you may often want to call an earlier word with the same name, in the situation where you are redefining the word to have similar but slightly changed behavior. For this reason it is now standard Forth practice to "hide" the name of the current definition while it's being compiled, so that a dictionary search won't find it, but will instead find an earlier word with the same name, if there is one. This is what Mops does as well. We therefore need another way of specifying recursion, and this is done, logically enough, with the word RECURSE. Using RECURSE means that you are calling the current definition.

If a situation arises in which you need to reference a Mops word before it has been defined, you can use Mops's forward reference facility. Before the word can be referenced the first time, you must declare it as forward in the following manner:

```
forward newWord
```

This declares newWord as a forward referenced Mops word. Later, when you are able to define newWord, you must do so in the following manner:

```
:f  newWord  ...  ;f
```

:f is a special colon compiling word that resolves forward referenced definitions. It creates a headerless entry for the new word in the dictionary, and then patches the previous entry, built by FORWARD, to point to the new definition. This will cause all compiled references to the FORWARD definition to actually execute the later definition. If you forget to resolve a forward reference with :f ... ;f, you will see a message at runtime informing you of the exact nature of the unresolved forward reference.

As a related issue, the Mops word PATCH can be used to patch any given word's references to another word. PATCH is used in the following manner:

```
patch oldWord newWord
```

Both :f and PATCH make oldWord behave like a colon definition, which means that they cannot be used for specialized definitions such as classes, objects, data structures, etc.

## Using resources in Mops

A resource on the Macintosh is essentially a structured database into which you can store initialization information for Toolbox objects and other data items, such as strings. Resources can improve the maintainability of your program: if you store all of the textual information for your application in a resource file, for example, it becomes very easy to convert your application to another language or change the wording of a given string. Another benefit of resources is that they shift the burden of storage for initialization values from your resident code to the dynamic heap, so your application takes up less memory space.

### Toolbox resources

There are several ways in which you can use resources with Mops. For example, Toolbox objects such as Windows generally have two methods you can summon for bringing a window alive. NEW: relies upon values passed to the method via the stack, and is independent of resource files. GETNEW: uses only a resource ID to find the template for the object in the currently open resource files. For instance, a Window would have a resource of type 'WIND' from which it would get its size, visible and goAway values, etc. Note that resource templates such as those of type 'WIND' only contain information relating to the portion of the object that the

Toolbox knows about - in the case of a Window, the window record. Other parts of the object, such as the window actions, must still be initialized by the application's code. Certain objects, such as those of class Icon, get all of their data from a resource item, and simply read the resource data whenever they are called upon to do anything.

Another way to use resources is for non-Toolbox objects that have no predefined template type. For these objects, you will need to use an existing type such as STR, or define your own types using ResEdit (see Putting Together a Mops Application).

### Defining and using resources

Mops provides an easy way to define a resource item from within your application. For instance:

```
resource myWind
'Type WIND  256  set: myWind
```

defines a resource called myWind that has type WIND and a resource ID of 256. When you need to access this resource, sent the message

```
getnew: myWind
```

Resource is a subclass of Handle, so you can now obtain a pointer to the resource data with

```
ptr: myWind
```

Note that if you do anything that might cause a heap compaction, this pointer will be wrong; but you can avoid this with the lock: method of class Handle, thus:

```
lock: myWind
```

You can open a new resource file in the following manner:

```
" myFile.rsrc"  openResFile
```

This opens the resource file named " myFile.rsrc" and make it the first file in the search order. All open resource files are closed automatically when your application terminates. Mops uses the file mops.rsrc for its resources during normal operation, and you can add your own resources to this file with ResEdit.

The source file QD1 includes support for cursors, icons and QuickDraw pictures via the resource interface. This makes it very easy for you to dress up your application with fancy graphics that you can create with a graphics application, and then add them to a resource file.

## Clearing nested stacks—Become

In a non-hierarchical, non-modal environment such as the Macintosh, the user is generally free to select another menu choice or open a different window at any time. This could happen while your code is nested several levels down, listening to events. If the user selects a new menu option that leads to an entirely new part of the program and your code is already several words deep on the return stack, the routine dispatched by the menu will nest several levels more. This could continue indefinitely until your application runs out of return stack, at which point it will bomb.

You have two ways to avoid this situation. One is to create an inverted architecture for your program, such that its event loop is at the highest level, and the code always returns to that level before listening to the event queue. This implies that you can never use KEY from within a called word, but only from the highest level. This may often be a good solution, but in other situations it may not lead to the easiest or the clearest implementation of a particular problem.

Mops gives you an alternative method by providing you with a Mops word called BECOME. BECOME causes Mops to erase everything that currently is on the stacks, and resets them to their normal empty values. Mops then executes the word whose name follows BECOME in the input stream. This automatically makes that last word the highest-level word in the application. In this manner you can actually have several mini-applications within one, each callable from the

other. At the point that BECOME is executed, you can rest assured that the stacks are empty and the application is essentially at ground zero.

## System vectors

Mops uses a powerful technique called vectoring to provide maximum flexibility for the programmer. Vectoring is the name given to the process of using a global or local variable to hold the address of a Mops word. For example, let's say that you would like Mops to interpret a file from disk just as though you were typing it at the keyboard. A built-in Mops system vector, named KEYVEC, always holds the address of the word that Mops normally uses to acquire keyboard input. By changing the contents of KEYVEC to point to a special word you define—a word that reads a single character from disk—all Mops words that accept keyboard input will then take their input from disk, instead of from the keyboard. For example:

```
: diskKey  Here 1 read: ffcb  drop  \ get 1 character from disk
      here c@  ;                    \ place it on the stack

" sam" name: ffcb
open: ffcb  .
' diskKey -> keyVec     \ set KEYVEC to get chars from disk file Sam
```

Of course, in a real example you would have to restore the proper KEYVEC value when EOF (end of file condition) was reached.

Mops has a full set of vectors for all critical I/O and compilation routines, allowing you to tailor the behavior of the Mops environment very easily. These vectors cannot reside in the Kernel, since that would preclude having several saved images that used different vectors. Thus, each saved image has its own set of system vectors, located near the start of the dictionary.

System vectors are slightly different to normal vectors, in that a 0 value may be stored in a system vector. This means that the default action is to be taken. Each system vector has its own predefined default word (which cannot be altered).

Here are the main system vectors and their required behaviors—there are others that are used internally in the Mops system and should not normally be changed, unless you really know what you are doing. Also there are some system vectors relating to Apple events which are discussed separately later.

KEY ( -- char)

gets keyboard input.

EMITVEC ( char -- )

sends one character to the primary output device.

PEMITVEC ( char --)

sends one character to the secondary output device.

CRVEC ( -- )

sends a carriage return to the primary output device.

PCRVEC ( -- )

sends a carriage return to the secondary output device.

TYPEVEC ( addr len -- )

sends a string to the primary output device.

PTYPEVEC ( addr len -- )

sends a string to the secondary output device.

ECHOVEC ( char -- )

handles echoing to the output device of the keys being input by ACCEPT.

ABORTVEC ( -- )

cleans up the stacks and notifies the user of an error. The Mops word CL3 ("clean-up 3") is normally executed by this vector, and your error word should call CL3 if it is to return to the Mops

interpreter.

QUITVEC ( -- )

this word will be executed before the interpreter enters its main loop. It should be the startup word for an installed application.

UFIND ( -- xt true OR -- false)

is a special purpose variant of FIND (this vector is actually called by FIND before FIND searches the Mops dictionary for an occurrence of a particular name at the top of the dictionary -- HERE. You won't have to worry about this vector unless you plan to write new compiling words for Mops.

OBJINIT ( -- )

initializes certain areas of the kernel at Mops startup. It normally contains the xt of SYSINIT. Should not be altered by the user.

HEADER ( -- )

lays down a dictionary header.

## Mops defining and compiling words

Much of the Mops language is, itself, written in Mops. This seemingly unlikely loop is possible because Mops is an extensible language - meaning that you can write new words in Mops that modify or extend the basic behavior of the language itself. In a sense, every word that you write extends Mops, because it adds to the same dictionary used by the Mops system. There are three layers of extensibility in Mops:

### 1. Vocabulary extensions.

Whenever you write a new word, instantiate a new object, create a new Value, and so on, you are extending the vocabulary of words that succeeding words can use. This obviously adds more power and function to the language.

### 2. Class extensions.

When you define a new class of objects, you are extending Mops in a somewhat more profound way than in a vocabulary extension. Creating a new class creates a new template for building other objects. These templates are known as defining words, because they can, themselves, define new dictionary entries. This is a powerful technique, because there is a good chance that you will be able to reuse defining words in your other applications. Eventually, you'll develop a large library of Classes, which should make your future application development much easier.

### 3. Compiler extensions.

The deepest layer of extensibility is concerned with constructing the tools that create defining words. Words such as :CLASS or :M are specialized compiling words that can truly extend the language syntax and add entirely new features. Compilers are the inner soul of the Mops language. They create control structures (such as IF, BEGIN and DO), Mops's class compilation facility, message processing, prefix operators...even colon itself is an example of a Mops compiler word. Mops's compiler words come largely from Forth.

You can use Mops for years without ever writing a compiling word, because Mops's class/object facility provides a very complete environment for programming. But if you are an advanced programmer and are interested in writing compiling words, the best source of ideas and learning is in the Forth literature, particularly the FORML and Rochester conference proceedings, available from the Forth Interest Group.

## Error handling

Mops' error handling is based upon the ANSI Forth Standard, which uses the two words CATCH and THROW. CATCH is used as follows:

```
['] someWord  catch
```

(of course, if interpreting rather than compiling, put ' instead of ['']). The action which takes place is to execute SomeWord, and if no error occurs, push a zero on top of whatever SomeWord may have put on the stack.

By saying "if no error occurs", we really mean that THROW did not take an error exit. THROW pops the top item on the stack, and if it is non-zero, it restores the stacks to where they were when the current CATCH was called (assuming that at least one CATCH is in effect). Then the non-zero value, assumed to be an error code, is pushed onto the stack and execution continues straight after the CATCH. If THROW finds a zero value on top of the stack it means no error, and execution continues normally.

From this you can see that CATCH and THROW provide a flexible means of unwinding out of deeply nested code if an error occurs. If CATCH catches an error, but doesn't want to recognize that particular error code, it can simply re-THROW it.

If no CATCH is in effect when an error occurs, Mops takes its default error action, which is to display an error message and execute ABORT. ABORT sets all stacks to their empty state, and initializes other Mops system variables to a suitable value before returning.

Before it executes, ABORT executes the System Vector ABORTVEC, to give extra flexibility in error recovery. Mops normally installs its own error handler word in ABORTVEC, called CL3, which prints the current stack of load files and clears this stack. You should call CL3 if you install an error routine for use during development, although the use of CATCH and THROW would probably be better. For your final installed application, since much of the Mops system won't be present, the Install routine asks you to specify an error word, which will be placed in ABORTVEC. You should provide a routine which tells the user what is happening and a suitable action to take (most likely in an Alert or Dialog box).

There are two error routines that indirectly call ABORT—ABORT" and ?ERROR. ABORT" must be followed by a space, and then a string terminated by a quote. Its action at runtime is as follows: if the top of the data stack is true (non-zero), it will print the string between the quotes and then execute abort. If false, ABORT" returns without doing anything. For instance, the phrase

```
read: theFile  abort" File read failed"
```

would check the return code from a disk read operation, and abort if it indicated an error. You can force an ABORT" to occur with the statement TRUE ABORT" ..." .

Embedding a lot of error strings in your code can take up unnecessary memory space, and it also makes the messages difficult to change. ?ERROR allows you to specify the actual text for your error strings in a resource file, and takes the resource ID number of the string to print, assumed to be a resource of type "STR " (note the space at the end). It works conditionally in the same way as ABORT". For instance,

```
find not ?error -13
```

prints the string with resource ID -13 if the word in the input stream is not found, and then aborts. Mops uses ?ERROR for most of its error messages. If you want, you can just print a resource string without executing ABORT by using the word MSG#. It takes a resource ID just as ?ERROR does. All of the error words function in compilation state only. To get a list of all messages and their numbers, type

```
.msgs
```

If you want to add a new message, do it this way:

```
<msg number> " the text of your message"  addMsg
```

If you want to change an existing message, you can't just use AddMsg as above or you'll get an error—this is just as a safety check. You have to remove the existing message first, thus:

```
<msg number>  removeMsg
```

If adding your own messages, please use numbers above 200, so as not to clash with future error messages we may want to add to the Mops system.

For an error while loading with echo off, the last word in the dictionary will usually be the word that experienced an error.

The file Mops.rsrc is a resource file containing all of Mops's error messages. Whenever one of the error words executes, it checks that Mops.rsrc is open. Of course, you must have Mops.rsrc within the folder Mops *f*, or Mops won't be able to find it.

## Inline definitions

You may specify that a definition or method is to be compiled inline whenever it is used. This allows faster execution. The syntax is:

```
: XXX  inline{ <some code> }  ;
```

The code <some code> is stored as a string, and whenever xxx is compiled into a definition, the string is compiled using EVALUATE. We actually store the source text for <some code> as a string, and EVALUATE it. This can give very good compiled code due to our optimization, which is why we took this approach. This syntax is really equivalent to

```
: XXX  " <some code>"  evaluate  ;   immediate
```

but the syntax is probably clearer. It also has advantages when used in methods. The syntax for an inline method is

```
:m YYY:
      inline{ <some code> }
      <some more code>  ;m
```

The reason there are two pieces of code is that we can't do compilation at run time, since the Handlers segment isn't always present then, but late binding does occur at run time. Therefore we need some pre-compiled code which is the equivalent of the inline code, but which can be called in the conventional manner on a late bind. Thus the two pieces of code above ought to be the same. There is, however, one difference. We assume that inline code chunks will be fairly short, and are to be optimized for speed. Therefore, when compiling the inline code (on an early bind), we do not change the machine's address register which normally points to the current object's base address. A normal method entry involves saving the previous base address on the return stack and setting up the new value, then popping the previous value when we exit the method. But for inline methods we bypass this procedure. Therefore if you want to refer to the current object's base address in some inline code, you can't use ^BASE. Instead, use the word OBJ. This is an immediate word which will generate the most efficient code to reference the object's base address, whatever it is, and allow our normal optimization to take effect. (Thus if we early-bind the same inline method to different objects, and there is a reference to OBJ, we will not necessarily get identical code compiled. It will depend on whether the object being bound to is a dictionary object, an ivar, or whatever.) Likewise, if you need to make an indexed reference in an inline method, instead of ^elem, use IX. All this sounds complicated, but it is actually fairly simple to use. The file Struct has many methods which use inline code, so if you look there you will see plenty of examples of how to do it.

# Utilitiy Modules

The Mops system contains several modules providing general functions that you might want to use in your application. You can do this either by referencing the imported names from the compiled modules as they are distributed in the Mops Folder, or by altering the source in the Modules folder and recompiling it to tailor the modules for your own use. Modules are advantageous in that they take up very little room in the resident dictionary, and load themselves only when needed on the application heap.

### The Alert box

This module gives you a predefined alert box without having to define any resources. Use Alert" within a word or method to produce an informatory message. Example:

```
readLine: myFcb  2 alert" A read error has occurred"
```

If the readLine is successful no alert will appear, but if readLine returns a non-zero result, an alert box with the given message will be displayed along with the specific error number. You may use Alert" in place of Abort" in your final application to comply with the "Mac Standard". Although customized alert boxes are better, this gives you a "quick and dirty" means of producing an alert box.

To use Alert", you must first include the source file AlertQ in your application. You may use need AlertQ to accomplish this.

You may modify the sizes and positions of any of the items, by using ResEdit to modify the resources ALRT 900 and DITL 900 in your application. These resources come initially from Mops.rsrc, and are copied into installed applications by Install, if the file AlertQ has been included.

### The Decompiler and Debugger

These are integrated these into a single module, DebugMod, since the job of displaying compiled code is very similar whether you're decompiling or debugging. To decompile a word, type

```
see  aWord
```

or to debug a word, type

```
debug  aWord
```

then keep typing spaces to step through each instruction. Of course, if you're debugging, nothing will happen until aWord gets executed.

The display gives a kind of pseudo-assembler, since we're compiling native code. But if you've loaded the source file through the usual load mechanism, the source will appear in a window at the top of the screen. If you loaded with logging on (turn it on with +LOG and off with -LOG) then a "log file" will have been generated, which will allow the Decompiler/Debugger to find the right place in the source corresponding to whatever compiled code you're looking at. The source window will be scrolled to the right place and an underscore will appear more or less under the right source text. This will move along as you step through the compiled code.

If you're running under System 7 and also have Quick Edit running, then instead of Mops producing its own (rather primitive) source code display, an Apple event will be sent to Quick Edit asking it to open the source file.

If you type <return> instead of space, then if you are looking at a call to another word (JSR or BSR), you will go down into that definition. If you type U, you will come back up again. Q will quit decompiling/debugging.

If you are debugging, G does a "go" (return to normal execution) and N is like G except that you will drop into the Debugger again the Next time you execute the word you wanted to debug. Typing R gives you a dump of the machine registers. Typing F (for Forth) gets you to the Forth/Mops interpreter so you can inspect values etc. Typing Resume gets you back into the debugger with everything as you left it.

Typing the arrow keys forces scrolling of the source text window. Keypad-7 goes to the top of the source, keypad-1 goes to the end, keypad-3 goes forward to the next colon, and keypad-9 goes back to the previous colon. (These keys are used by Microsoft Word to mean Home, End, Page down and Page up, respectively, which is why I chose them.)

To decompile/debug in a module, type e.g.

```
in aMod  see aWord
```

To decompile/debug a method, type:

```
[in aMod]  see aMeth: aClass
```

The debugger works fine on both my Mac Plus and IIsi, and even under System 7. Since I change two interrupt vectors (T-bit and TRAP #0), it may not always work on all Macs or systems. It may, and hopefully it will, but I may not get a chance to try it out. I will be very surprised indeed if it works on PowerPC Macs.

Be careful if you are using Macsbug—the debugger replaces the vectors when it terminates normally, but if the program crashes or something, the vectors may be left changed, and Macsbug won't then work properly. Rebooting is the only way out.

If you're running under System 7 and have Quick Edit running as well as Mops, then as we mentioned above, Debug and See will send Quick Edit an Apple event asking it to open the source file. You can use this feature directly, by typing

```
edit someFile
```

Then assuming that someFile can be found via the normal Mops file lookup, the Apple event is sent to Quick Edit so that it will open the file. If the file can't be found, or Quick Edit isn't running, you'll get an error message.

## The Profiler

The purpose of a profiler is to give statistics on time spent in various lines of code, and also the number of times they have been executed. Because of the hierarchical nature of the language, it seemed to make the most sense to base profiling on a given word, whose definition is profiled. This way bottlenecks can be tracked down interactively, and you can zero in on the places of interest, rather than have to wade through a mountain of useless information. Anyway, it was easier to implement this way.

To use this feature, the file containing the definition to be profiled must be loaded with logging on, since the profiler needs to know the correspondence between the source and compiled code, and this is recorded in the log file. Once this is done, just type

```
profile aWord
```

then after aWord has been run, more than once if need be, type

```
showP
```

and the source code of the definition of aWord will appear on the screen, with the statistics on the left side of each line. Turn printing on first for a hard copy.

Profile and showP are in DebugMod. Profiling works by putting a breakpoint on the first instruction corresponding to each source line. There is thus a few instructions' overhead on each line, which will slightly skew the timing results for lines such as aValue IF which only contain one instruction anyway. We try to minimize this effect by not counting time while the profiling code itself is running, but the breakpoint trap and the return instruction (RTE) do take a number of cycles. Use your common sense. Also, for processor independence, we use Mac ticks (1/60 secs) to count time. Therefore the word being profiled ought to accumulate at least several seconds' execution time, for the results to be very meaningful. Call it repeatedly in a loop, if necessary. The longer the execution time, the more accurate the results. If you get a lot of lines apparently taking zero time, this probably means you need to run the word more times. (But remember, lines consisting of just THEN, say, don't actually compile any code. So of course they won't take any time.)

If you're only interested in the execution counts, you don't need to bother about the length of the run. These counts ought to be right, no matter what.

Finally, this is a new feature, and it hasn't been very extensively tested, so please exercise caution (i.e. don't blame me if you crash without saving).

## Run-time initialization

We have a feature to make it easy to do special-purpose run-time initialization. Some files such as LongMath need an initialization call on startup (in the case of LongMath, this is to check if the processor we're running on has 32-bit multiply and divide instructions). Using several such packages together could cause problems, as each could have redirected ObjInit without being aware of the others. And yet we want to be able to have standard packages such as LongMath which don't have to be aware of what other packages may or may not be present. To avoid this problem, we have an x-array of words to execute on startup, called INIT_ACTIONS, and these are all executed right after ObjInit is called. This way, ObjInit can be restricted to just initializing the standard Mops objects, and any extra initialization can be done by adding a cfa to init_actions. If you write a package that needs special startup action, make the startup action into a word, let's call it MyStartupWord, then at the end of your code put

```
' MyStartupWord  add: init_actions
```

and that's it. If you do a Forget or RL (reload) there's no problem, since the Mops loading code starts by purging init_actions of any xts above the current top of the dictionary. This prevents init_actions from getting invalid xts in it.

## PowerPC assembler and disassembler

Mops can't generate native PPC code yet—maybe we'll get there for the next major release—but in the meantime we do have a PPC assembler and disassembler, thanks to Xan Gregg. Xan has written this assembler for the PPC version of MacForth, and he's kindly allowing me to distribute it with Mops in exchange for a program I've written to parse Apple's new Universal Headers.

I've adapted the assembler to the Mops environment, so it's a module, pasmMod (with the source file pasmMod.txt). You write a PPC code definition thus:

```
:ppc_code  someName

      <ppc instructions>

;ppc_code
```

This will create a normal header for someName in the dictionary, then align the DP to a 4-byte boundary (as required for PPC code), then compile the code. Note that if you tick someName, the resulting "cfa" mightn't be on a 4-byte boundary since our cfa's are only 2-byte aligned for the 680x0. So the "cfa" should be rounded up to the next 4-byte boundary to get the address of the first PPC instruction in the definition. Possibly when we have a full native PPC version of Mops we might do something a bit more sensible here.

There's no manual for the assembler or disassembler yet—you're on your own! But the comments in the source files are fairly extensive, so if you're the type of person who might want to write PPC assembly, you'll probably be able to figure out what to do! — especially as the file "test pasm" in the System Source folder has a definition containing all PPC instructions. Note: it's a Forth-style postfix assembler.

The PPC assembler-related files are:

| | | |
|---|---|---|
| pasmMod.txt | in "Module Source" | the assembler. |
| disasm | in "Module Source" | the disassembler (loaded by pasmMod.txt). |
| test pasm | in "System Source" | a big test definition with all the PPCinstructions. |

# Technical Section

This chapter is intended for hackers, or people who want to understand the nucleus source code better, or for those who are just insatiably curious. You may use Mops with great success without reading this section, but if you run into some obscure problem which defies analysis, or if you're writing a lot of assembly code definitions, you might find something useful here.

### Mops run-time environment

In compiling code to reference the dictionary, we have the problem that native 68000 code only allows a 16-bit displacement for addressing, in the normal An plus displacement addressing mode. That is, unless we resort to extra steps in every memory addressing operation, every part of the dictionary must be within 32K of where an A register is pointing. The 68020/030 chips have a 32-bit displacement mode, but the 68000 doesn't. Mops has to run on 68000 machines (Plus, SE, Classic, PowerBook 100), of course, so I was forced to find a solution to this problem. The eventual solution was to dedicate three A registers for addressing. Mops uses A3 and A4 to address the main dictionary—these registers are set up with A4 pointing 64K higher than A3, and these two registers never change during execution. This way we can address a dictionary up to 128K in size. A dictionary can be larger than this, but memory references above the 128K range may be slightly less efficient. (We give details on this in the later section "Dictionary size".)

We call A3 "lobase" and A4 "hibase". Modules are addressed using A5 ("modbase"). Since Neon never allowed a module to be accessed from outside except through an exported word, we maintain this rule in Mops, and thus we can ensure that A5 is always valid when a module is in execution.

We use A6 as the data stack pointer, and A7 as the return stack pointer. This is the other way around to Neon and Yerk. The advantage for Mops is that a Mops word can be called with a simple BSR or JSR instruction, and nothing further needs to be done with the return address, since it is already on the top of the return stack. All the words which call the system (such as TRAP) exchange A6 and A7 before the system trap, and then exchange them back on return. This means that Mops code can push parameters onto the data stack and take results from the data stack, exactly as in Neon and Yerk.

In Mops we have been able to eliminate the methods stack (which Neon had). We have the same functionality, but we don't need actually need a separate stack. This comes about because we use A2 for the base address of the current object in methods, and save nested object addresses on the return stack. Named parameters and local variables used to use the methods stack in Neon; we use D4-D7 and an overflow memory area, and save

whatever is necessary on the return stack during nested definitions.

So to summarize the register usage:

```
D0-2  scratch
D3    loop counter I
D4-7  named parms/locals
```

```
A0-1  scratch
A3    lobase
A4    hibase
A5    modbase
A6    data stack pointer
A7    return stack pointer
```

## Dictionary header format

This has changed somewhat from Neon. Most significantly, we have an 8-way threaded structure, for speed in lookup. To decide which thread to use for a given word, we simply use the length of the word's name, modulo 8.

In a dictionary header, first there is a 4-byte link field at the lfa (link field address), which gives the displacement from the lfa itself to the previous header on the same thread. Then comes the name field, which consists of a length byte followed by the name itself, with a zero byte added if necessary so that the total name field is even in length. Next comes the 2-byte handler code (see later under "compilation and optimization". That is the end of the header. We say that the following byte is at the cfa, even though we don't have a code field as such. For most definitions, executable code starts at the cfa, and for constants, variables and values the data comes there.

In earlier Mops versions we used a 2-byte link field. The change to 4 bytes was made to allow very large dictionaries.

## Compilation and optimization

Compilation in a STC/native code system is necessarily a bit more involved than in traditional Forth-based systems. Optimization adds another level of complexity. Most user coding can be done exactly as in Neon, but the underlying compilation process is a bit different, and so there are a few pitfalls to avoid.

Firstly, let's look at plain ordinary colon definitions. When a reference to one of these is compiled, we compile a BSR.S if we can, otherwise a JSR. Such words are EXECUTEd simply by calling them. So far so good.

We also have inline definitions. For example, it would have been silly to handle DUP by JSRring to the following code:

```
MOVE.L      (A6),-(A6)
RTS
```

when we could simply have compiled the MOVE right in line. Two bytes instead of 4 for the JSR, and much faster. So what we do is mark certain definitions as inline, and when we compile them, we just move the code in. In previous versions of Mops you couldn't EXECUTE such definitions, but as from v. 1.6 you can, since we include an RTS instruction at the end of the original copy of the inline code (which starts at the cfa of the definition). This RTS isn't copied over when the code is inlined in another definition, but is there in the original copy so that it can be called by EXECUTE.

Other dictionary items, such as objects, require other special actions to take place when a reference to them is compiled. So what we have tried to do is to take quite a general approach to the problem. Each dictionary header contains a 2-byte field we call the "handler code". This determines exactly what should happen when a reference to this dictionary item is to be compiled. For inline definitions, the handler code is positive (but not zero). It is interpreted as a byte count of the number of bytes of inline code, and this code immediately follows the count. For most other dictionary items, the handler code is negative and even. It is used as an opcode for a call to Handlers, which is a separate segment (CODE segment 3). There are quite a number of different opcodes, and this gives us great flexibility. In installed applications, the Handlers segment is not included, and this saves about 16K of code space, and also makes it completely impossible to do any Mops interpretation from installed applications.

Our method of code optimization should be transparent to the user, but the method used has some interesting features. We don't attempt the complexities of looking ahead in the input stream to find sequences of words that could be optimized. Rather, we check what we have already compiled to see if it can be improved. Whenever the Handlers module compiles code that may have optimization potential, it pushes a descriptor onto a stack. Then when we are checking for optimization possibilities, we check downwards on this stack and take appropriate action. Each descriptor contains a copy of the DP value for the start of the corresponding code sequence, so whenever we find that we can optimize the previously-compiled code we can easily reset the DP to the right place, then emit the improved code. This technique allows us to get some surprisingly good results. For example, if vv is a variable, the code

```
0  vv  i +  c!
```

will compile to

```
LEA   xx(A3),A1          ; Address of vv to A1
CLR.B 0(A1,D3.L)         ; Index on D3 (loop counter i)
```

This technique also means that we never have to backtrack if we tried an optimization strategy which failed. The unoptimized code is always compiled first, and is only recompiled if we know we can legitimately improve it. This optimization technique has worked well in practice, and generally gives around a 15% improvement in execution speed and a 10% reduction in code size. Of course, some common code sequences are improved much more than this.

Another optimization involves conditionals. If, for example, we have the code

```
aConst  IF  <some code>  ELSE  <some more code>  THEN
```

where the conditional is testing a constant, the condition will be evaluated at compile time, and either <some code> or <some more code> will be compiled, but not both. We therefore have a conditional compilation capability, without introducing any new syntax at all.

Finally, it is worth noting that our optimization technique is reasonably efficient, so that compilation speed does not appear to have been degraded significantly at all. It is certainly much faster than Neon.

## Object format

Here is our format for normal objects (remember that ivars declared within record {...} won't have a header—that is, there will be nothing before the object's actual data). I'll take this from the top. For dictionary objects, this will be the cfa. For heap objects, this will be the beginning of the heap block.

2 bytes

offset to the indexed area. For non-indexed objects, this will be 6. See below for the subtle reason.

4 bytes

Class pointer (relocatable)

2 bytes

Offset from the data start to the class pointer. -6 for simple objects.

(object's data starts here)

For indexed objects, the indexed area (after the ivars) is preceded by a descriptor with the format:

2 bytes

Width of indexed elements (in bytes)

4 bytes

Number of elements minus 1 (i.e. LIMIT-1)

The last field was a 2-byte field in Neon, and also it held Limit rather than Limit-1. The expansion to 4 bytes allows the number of elements to be limited only by available memory. The

change to Limit-1 is so that we can do a bounds check on an index with a single CHK machine instruction.

The reason we need the offset from the data to the class pointer relates to multiple inheritance. In Neon, the class pointer immediately preceded the object's data. But with multiple inheritance, we will have the ivars for the various superclasses following each other. If a method in one of the superclasses is called, and it needs the class pointer, it can't look just before its first ivar, since that isn't actually the start of the object's data any more. So what we do is put a 2-byte offset to the class pointer before each group of ivars belonging to each superclass.

Now for the reason we use 6 as the offset to the indexed area header for non-indexed objects. When doing indexing, we use the indexed area header or descriptor (I sometimes abbreviate this as "xdesc") to check for the indexed access being in bounds. We actually do the check with a CHK instruction, so it is a very fast check. Now, for non-indexed objects, an offset of 6 won't get us to any xdesc, since there isn't one, but will get us to the other offset - the offset to the class pointer. Now this offset is always negative. When the CHK instruction is executed, it will see a negative number as the "number of elements", and always fail. I don't normally resort to clever tricks like this, but efficiency considerations were paramount here.

## Dictionary size

The Mops main dictionary may be very large (up to 8 megabytes, in fact). We had to find a way to solve the problem of addressability for the part of the dictionary that is above the hibase (A4) range.

What we do, if we have to compile a reference to this area, is to use a PC-relative mode, if the instruction making the reference is within 32K bytes of the target location. This will probably be so for the majority of references. If we're outside the PC-relative distance, we compile the sequence

```
MOVE  A3,A0
ADD.L #offset,A0
<op>  (A0)
```

This involves some overhead, but it does the job. Another problem is that if the operation we are compiling alters memory, we can't use the PC-relative mode, even if we are within 32K of the target location (not on a 68000, anyway). So in this case we compile

```
LEA   <addr>,A0
<op>  (A0)
```

Hardened hackers will be quick to point out that we can't do a TST in PC-relative mode either. So now we always compile a MOVE to D0 instead of a TST. This sets the condition code exactly as for a TST, and takes the same number of cycles.

To get some idea as to the real-world space/time penalty for compiling these code sequences, I loaded Mops up to Files, then did a big ALLOT to get the DP above the hibase range, then loaded the rest of the system. The resulting load was less than 10% larger than normal. The speed seemed unaffected, but I didn't do detailed measurements. I expect it was impacted to about the same extent as the space. These results suggest that for most applications, going over the hibase range won't make a lot of difference.

## CODE resources

In the original Neon implementation, the CODE 0 resource was the jump table (as in all Mac applications), and CODE 1 was the dictionary. In installed applications, each module became another code segment, but in "development mode" these were the only CODE segments there were. On startup, CODE 1 expanded itself (via _SetHandleSize) to the required dictionary size, and the selected dictionary image was then read in above the nucleus.

The Mops scheme is a slight development of this. Under System 7 I found the system was loading CODE 1 high, because it was marked "locked". It then couldn't expand itself, since it couldn't relocate while running, and there was no room above it. I found I could get around this problem by not marking it locked in the resource file, but felt that maybe I was heading for trouble later on. I thought that possibly having a CODE segment expand itself, while actually running, might not be such a marvellous idea. So now we have another CODE segment. The main dictionary is now CODE 2, and CODE 1 is a small segment that does most of the initialization stuff. It also gets a handle to CODE 2, unlocks it (just in case), expands it as needed, then moves it high, locks it and jumps to it. Once in CODE 2, we unload CODE 1, which has done its job.

CODE 3 is the Handlers segment. It has only one entry point, and an opcode is passed to it to tell it what to do. The comments in Handlers have all the details.

Installed applications have CODE 0, CODE 1 and CODE 2 segments. Handlers is not present, and is not needed. The CODE 2 segment doesn't need to be expanded, since all the needed dictionary is already there. CODE 1 still does all the initialization, but skips the part where it expands CODE 2. As in Neon, any needed modules become additional CODE segments. Their resource numbers will be strange numbers obtained by calling UniqueID, but the names are attached to the resources so you can tell which is which in ResEdit, if you need to.

The jump table is very short, since it only needs to reference 4 CODE segments. Modules are accessed through the corresponding module object in the dictionary, not through the jump table. Other object-oriented systems on the Mac have used the jump table for implementing dynamic objects, and so have run into problems with the original 32K limit on the size of the jump table. This limit has now been removed by Apple with its new "32-bit everything" addressing scheme, but in Mops we never had the problem anyway.

## Relocatable address format

This section is included for information only, and to help debugging. A I said at the beginning, please don't rely on these details remaining accurate. I have no plans to change at the moment, but you never know.

Our relocatable address format was determined largely by the need to be able to address anywhere within a very large dictionary, and also to fit in 4 bytes. The top byte is 5, 6 or 7.

It is 5 if the address is relative to the main dictionary. The low 3 bytes give the offset from lobase (A3).

The top byte is 6 if it is a module address. The low three bytes give the offset from modbase (A5).

The top byte is 7 if it is a self-relative address. We currently use this for a module address which is stored in the same module. A situation can arise for which A5 isn't valid—an action handler in an object whose class is implemented in a different module, may well be accessed when A5 is set up for that other module. This relocatable format will work in this situation since it doesn't depend on A5.

It only takes around 10 machine instructions to decode a relocatable address, and we also pick up the case where the top byte isn't 5, 6 or 7, and give the "Not a relocatable address" error. (We used 5, 6 and 7, rather than 0, 1 and 2, to give a higher rate of error detection—if an erroneous longword is used as a relocatable address, its probablilty of having a high byte of zero would be greater than for most other values, I would expect.)