Part II
Using Yerk

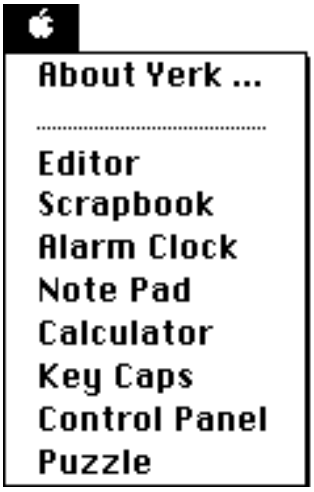
Table of Contents

1
The Yerk Menu Bar

Yerk has a simple set of menus, yet the features built into them make writing code, compiling, and debugging rather easy. When you start yerk.com, a specially designed Yerk "front end" brings in the basic Yerk menus -- Apple, File, Edit, Utilities, and Yerk. Additionally, when you start a desk accessory Editor on top of yerk.com, an additional menu may appear on the menu bar.

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 Yerk... About Yerk... produces an overlaying window that displays the version number and author credits for Yerk.

The rest of the selections are the desk accessories, which operate just as they would from the Finder and in all Macintosh applications as well as from the Editor window. You may have also installed your own.

THE FILE MENU

Load... — Load... allows you to load text source files atop the Yerk dictionary currently in memory (the same as issuing the "// filename" command from the Yerk prompt). The standard file dialog box appears, from which you can select the file to load, from any active disk on your system. As the source file loads, it is compiled by Yerk. If your program requires the loading of several text files, the files must be loaded in the proper order (so that superclasses are defined before their respective subclasses). As a shortcut, you can build a separate "loader" file, which consists of a list of commands that load all the desired source files in order. See the text file "demo.load" on the Yerk disk as an example of how grDemo was loaded into Yerk by a loader file. The Load dialog does not change the default drive that Yerk uses to find its resources and other essential files.

Save — Save 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 Yerk window. For this reason, Save should be used with care (the destination folder is the Yerk folder). If you have added code to Yerk.com and wish to save the image as a separate application, then use the Save As... selection, below; otherwise, your Yerk.com file will contain your additions.

Save As... — Save 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 Yerk.com was originally built, its compiled image was saved with this command. To start a program saved in this manner, double-click the appropriate Yerk document icon from the desktop, just as you start Yerk.com. 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 Yerk interpreter while debugging source files by maintaining the code as source files and Loading them to test how well the program runs.

Print... Print... produces a standard file dialog box, which allows you to select a text file to send to the printer. You can perform this operation independently of an Editor, which has another print command. Yerk's Print produces a formatted listing on an ImageWriter printer only. At present, it does not list out to a laser writer.

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

THE EDIT MENU

Undo Undo is deactivated.

Cut In the Yerk window, Cut is deactivated.

In an Editor window, Cut deletes selected text from the displayed body of text. The cut text is placed in the Clipboard until another Cut or Copy operation is performed. If you wish to move a block of text inside an Editor document, you can Cut it from one location, move the text pointer to the target location, and select Paste from the Edit menu.

Copy In the Yerk window, Copy is deactivated.

In an Editor window, Copy copies selected text and places it in the Clipboard until another Cut or Copy operation is performed. Place the

text pointer in the target location for the duplicate copy, and select Paste from the Edit menu. Copy also lets you select a portion of a source file and Paste it into the Yerk interpreter on the Yerk window to test your code. See Part II, chapter 2 for more details.

Paste

In the Yerk window, Paste inserts text Copied from a text file at the Yerk prompt. If, for example, you write a Yerk definition in an Editor window, you can Copy it from the Editor, activate the Yerk window, and select Paste from the File menu. The text of the definition is quickly loaded into memory, as if you had typed it at the Yerk prompt. If the code you Paste into the Yerk window is executable code, Yerk will respond just as if you typed the code at the Yerk prompt yourself. See Part II chapter 2 for more details.

In an Editor window, Paste inserts text Cut or Copied from the same or different text file at the location of the text insertion pointer.

Clear

In the Yerk window, Clear is inactive.

In an Editor window, Clear erases the selection range without putting it into the Clipboard.

Editor

In the Yerk window, Editor brings up McSink (a shareware desk accessory editor), if you have it installed. The name of the desk accessory editor is stored in a STR resource (ID=99) in the yerk.rsrc file. You may change this to the name of your desk accessory editor name. Notice that two leading zeros are required for system 7, and one leading zero is required for systems 6.08 and earlier.

THE UTILITIES MENU

List Words

List 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 return to the Yerk prompt, press any key other than the Spacebar. Holding the option key down while selecting this item will allow you to select the word to start listing from; holding the shift key down will allow you to select the start address.

List Words should be used when the Yerk window is active.

List Objects... List Objects presents a dialog box from which you select the class in memory whose objects you wish to see listed. Type "do' classname" to bypass the dialog box input.

List Objects should be used when the Yerk window is active.

List Classes List Classes presents a list of all classes and objects 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. Object addresses are shown in hex, with the object names in lower case. Issuing this command on YerkFP.com with tool classes loaded using the load-list source 'addl.ld' results in something similar to the following:

```
OBJECT                  Dlen: 0   Width: 0
    PICTURE                       Dlen: 14    Width: 0
    CONTROL                       Dlen: 18    Width: 0
    VSCROLL                       Dlen: 76    Width: 0
        VSCTL                     Dlen: 86    Width: 0
    PBDRVR                        Dlen: 62    Width: 0
    PORT                          Dlen: 72    Width: 0
        13E4A  pwout
        13DF4  iwout
        READPORT                  Dlen: 88    Width: 0
          13F06 pwin
          13EA0 iwin
    DATETIME                      Dlen: 36    Width: 0
     13002  sysdate
    TIMER                         Dlen: 6     Width: 0
     12D02  ptimer
     12CD0  systimer
    FARRAY                Dlen: 0   Width: 10
    FLOAT                         Dlen: 10    Width: 0
    FPI/O                         Dlen: 68    Width: 0
     10E8C floati/o
    FLTHEAP                       Dlen: 2     Width: 12
     F062        fltmem
    MBAR                          Dlen: 176   Width: 0
    BARRAY                Dlen: 0   Width: 1
    BYTECOL                       Dlen: 2     Width: 1
        RADIOSET                  Dlen: 8     Width: 1
    WARRAY                        Dlen: 0     Width: 2
```

```
WORDCOL                         Dlen: 2     Width: 2
MOUSE                           Dlen: 8     Width: 0
 C470    themouse
GRAFPORT                        Dlen: 108   Width: 0
WINDOW                          Dlen: 218   Width: 0
```

```
        44E fwind
        RSRCWIND                    Dlen: 220   Width: 0
        CTLWIND                     Dlen: 220   Width: 0
      PADDLE            Dlen: 220   Width: 0
    RECT                            Dlen: 8     Width: 0
     14DBA theclip
     BEC8       temprect
     6AE        fprect
    SCROLLRECT                      Dlen: 12    Width: 0
    BOXWORD                         Dlen: 8     Width: 0
    USERITEM                        Dlen: 16    Width: 0
    POINT                           Dlen: 4     Width: 0
    BITMAP                  Dlen: 0 Width: 1
    FILE                            Dlen: 234   Width: 0
     5C0    ffcb
    ARRAY                           Dlen: 0     Width: 4
    X-ARRAY                         Dlen: 0     Width: 4
        B8DA   aact
        DIALOG              Dlen: 12 Width: 4
        MENU                        Dlen: 6     Width: 4
         12116  yerkmen
         120D8  utilmen
         1209E  editmen
         12070  filemen
        HMENU                       Dlen: 6     Width: 4
            PMENU                   Dlen: 14    Width: 4
        APPLEMENU                   Dlen: 6     Width: 4
            E8EE    applemen
        EVENT                       Dlen: 26    Width: 4
         536    fevent
        ORDERED-COL                 Dlen: 2     Width: 4
        RADIO                       Dlen: 4     Width: 4
        FILELIST                    Dlen: 2     Width: 4
            9A7E    loadfile
    VAR                             Dlen: 4     Width: 0
     15008  itemhandle
     12390  theoffset
    HANDLE                          Dlen: 4     Width: 0
        A404    mhndl
        BASICSTR                    Dlen: 8     Width: 0
         12154 imagename
        STRING              Dlen: 8 Width: 0
            1216E  parmstr
            SARRAY          Dlen: 14 Width: 0
    INT                     Dlen: 2 Width: 0
     15020  itemtype
     14FF6  theitem

number of classes is    49
number of objects is    30
```

List Chain

List Chain brings up a dialog box in which you may enter a class name. This class hierarchy from class Object to that class is listed in the Yerk window. You may also type hc' in the Yerk window followed by a class

name.

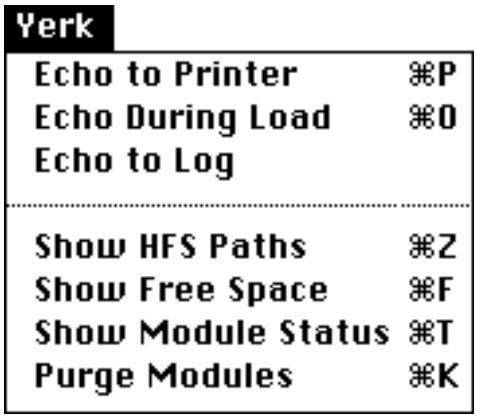
List Chain should be used when the Yerk window is active.

Examine Memory — Examine Memory presents a byte-by-byte memory dump of the Yerk memory heap. This entire memory area is scrollable. Additionally, you can have the utility search for the location of a particular Yerk word, a particular address, or the location of HERE by clicking the appropriate buttons in the Examine Memory dialog box. You also have your choice of viewing the memory locations as relative (Yerk) or absolute (Macintosh) hexadecimal addresses.

Grep... — Grep... presents a dialog box that lets you specify a text string for which all text files on the disk will be searched. Results of the search -- a listing of the file name, the line number of the file with the string, and a display of the text line -- will be displayed on the Yerk window or can be printed out by selecting Echo to Printer on the Yerk menu.

Install — In the Yerk window, Install lets you adjust the relative sizes of the stack, heap, and dictionary in memory. As you change the size of the stack or dictionary area, the heap is affected. Therefore, you can adjust the stack and dictionary sizes directly using the utility's adjustment controls to the right of the digital readouts. See Part II, Chapter 5 for more details on the Install utility.

THE YERK MENU



Echo to Printer — Echo to Printer duplicates line-by-line, all text that appears in the Yerk

window as it is either typed or displayed by your program. It lets you keep a running record of your input and your program's output during a session. This is particularly helpful because text in the Yerk window may be overwritten by dialog and other windows, and is not updated when the Yerk window is reactivated. If you select Echo to Printer, a check mark appears next to the

menu listing. Selecting it again turns off the feature and removes the check mark.

Echo During Load

Echo During Load displays every line of text from a source file as it is being loaded and compiled into Yerk. 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 Yerk 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 Yerk 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.

Echo to Log

Echo to Log will echo all text that would normally print to the Yerk window to a disk file. The default filename is 'Logfile', located in the Yerk folder, will be appended to each time you select it. If you wish to name your own file and destination, hold the option key down when selecting this item. The selection is identical to the Yerk commands +file and -file. If you select Echo To Log, a check mark appears next to the menu listing. Selecting it again turns off the feature and removes the check mark.

Show Free Space

Show Free Space displays in the Yerk 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 on a 512K Mac is shown below:

```
Room in Dictionary:    227765
Total Heap (no purge): 142062
Largest Block (purge): 146358
```

Show Module Status

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:

```
TOOL1          0
ASMMOD                 0
DOCMOD                 0
KONSTANTMOD    0
ENV            0
UTIL           4DDFC   ***Locked***
```

```
TOOL        0
SORTMOD     0
QPMOD       0
PRINTMOD    0
LOGMOD              506A4
INDMOD      0
IMOD        0
GREPMOD     0
EXAMMOD     4EBEC  ***Locked***
DEMOD       0
ALERTMOD    0
ABOUTMOD    0
```

Purge Modules Purge Modules clears the heap of all modules loaded by your program.

2
Using an Editor

If you have ever used an editor with other programming languages, you probably realized that a lot of time was wasted while switching between the Editor and the language, especially when fine tuning a nearly completed program. In Yerk, however, the Editor and the language can run almost simultaneously. In fact, if the Editor functions as a desk accessory, it is ready to be summoned without having to quit the language. An Editor window and the Yerk window can both be on the screen at once (although only one can be active at a time), and switching from one to the other is as easy as clicking the appropriate window when you need it. If you are running multifinder or system 7, you may use an editor or word processor application.

Organizing Your Editor Window

Because both an Editor and the Yerk window are draggable and growable, you can shrink and move them whichever way proves to be most productive for you. The precise positioning you choose will probably depend on the kind of work you're doing at any moment. Here are some examples.

If you feel most comfortable seeing full windows for both Yerk and the Editor, then you can leave them both almost as they are originally created. You should, however, adjust one of the windows so it is shorter and wider than the other. In other words, you want at least some sliver of each window to be visible when the other window is active. In that way, you can switch instantly between Editor and Yerk windows by clicking the mouse pointer on that sliver of "buried" window showing from underneath.

In other instances, you may want to have as much of both windows visible as possible at one time. You can essentially divide your Macintosh screen into two halves. Close up the Yerk window by dragging the grow box up to about midway on the screen. Then start the Editor. Drag its grow box to about the same spot. Then place the pointer on the title bar and drag the entire Editor window to the bottom of the screen (see Figure II.2-1).

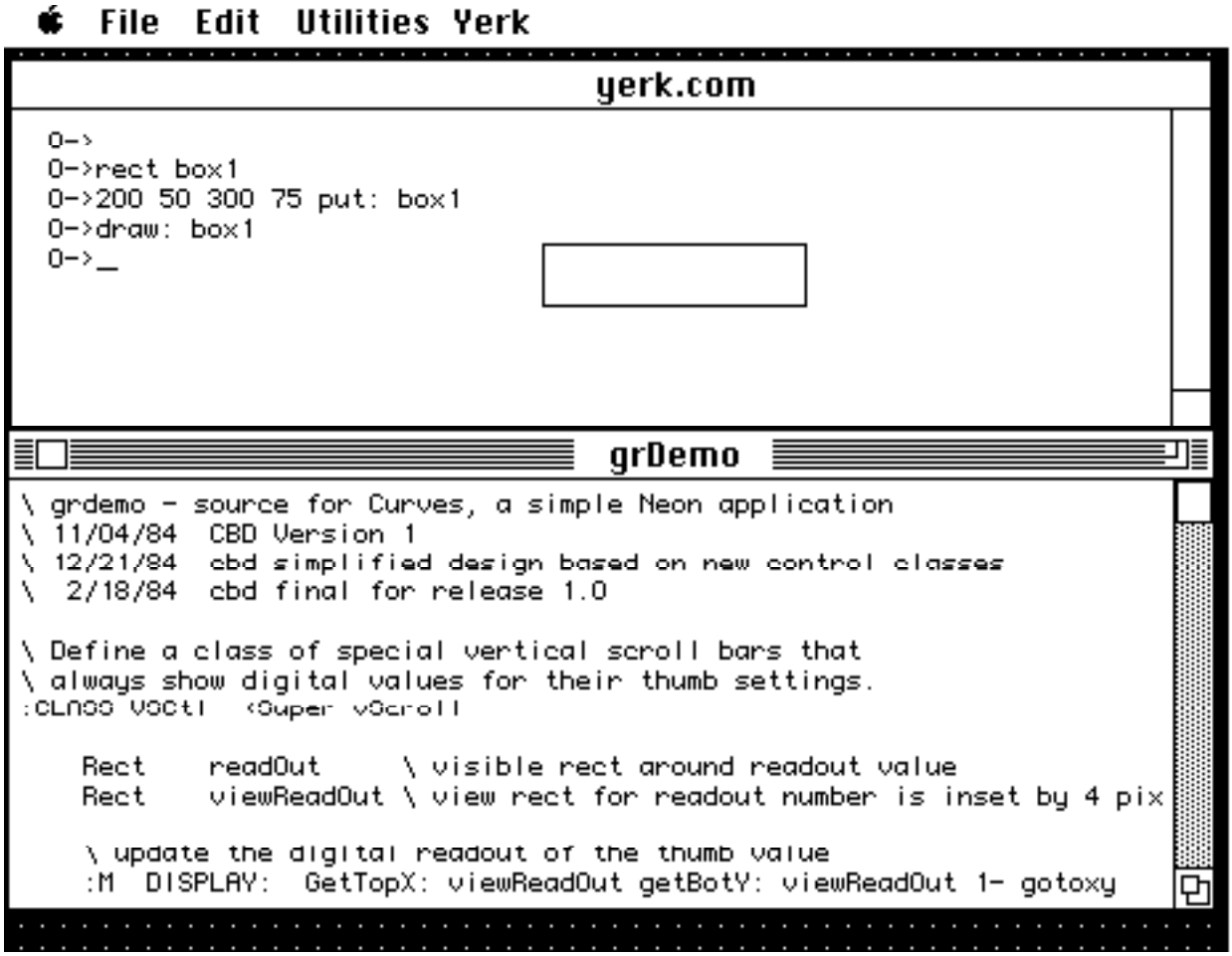


Figure II.2-1

This latter arrangement will be particularly useful for those times when you want to test a part of your code written in the Editor. You'll be able to select and copy a section of text from the Editor window (after saving your text, of course), select the Yerk window, and then Paste the text into the Yerk interpreter. Yerk definitions will be compiled as if you were typing the text in by hand. If the results are not what you expected (and the error wasn't severe enough to "bomb" the system), you can immediately switch back to the Editor window, modify your code, and try again.

Saving Files

Pre-existing files that you have loaded into the Editor should be saved periodically during editing sessions. This guards against lost source code in the event of a power failure. The Save option from the Editor menu performs the save operation to a disk file with the same name as the file you're working on.

As further protection for yourself, we recommend that you perform a Save before each time you switch to the Yerk window for any reason, whether it be running one of the utilities from the Utilities or Yerk menus, or, especially, when copying Editor text and pasting it into

the Yerk interpreter. The primary reason for this extra safety is that the Mac could, for any number of reasons, find itself in some unrecoverable error condition, forcing a complete Reset. If you have not saved your text file, you won't have a record of the errant command that got you into trouble, and you may also lose other work not saved prior to the window switch.

For a new text file created from a Blank Page, save the file with the Save As... selection from the Editor menu. You will be prompted for the name you wish to give to the file. You will also be able to direct the file saving to occur on another disk in the same or different disk drive. If you are creating several different source files, you might consider saving them on a disk other than your Yerk working disk. Fortunately, Yerk's Load command (from the File menu) lets you easily load files from other disks.

It is also good practice to save backup copies of your source files. The Save As... option makes this an easy operation. If you are finished editing a file for the day, you can save it to the current disk by simply selecting Save from the Editor menu. But by then selecting Save As..., you will also be able to save it to another disk by either clicking the Drive button (if you have a storage disk in another drive) or ejecting the current disk, inserting a new disk, and then clicking the Save button in the dialog box. The current file name is pre-typed into the filename box for you, simplifying this second save to your backup disk.

Editor File Size

We recommend that you keep your source code files to as small a size as possible, dividing your programs up into easily digestible and debuggable portions. When you have debugged fundamental parts of your program, you can work more confidently on later, more complex parts of the program. The procession of Sin, Turtle, and grDemo files in the Tutorial show you one approach. The largest file in the series, grDemo, is only 8K.

3
Writing your own Classes

Building a Yerk program is largely a process of defining classes of objects -- classes which are the "frame work" 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 Yerk 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

Yerk 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 Yerk 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.

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 Yerk 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.

Class Structures

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 SuperClassName  [ n <Indexed ]
   [   instance variable names  ]
   [  method definitions ]
;CLASS
```

In the above example, the brackets indicate optional sections of a class definition. If you build a class 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; by convention, class names are always capitalized. SuperClassName is the name of an existing class that is to be the basic model 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 Yerk code, 3 Array A1, builds an indexed object, called A1, of class Array that has 3 indexed elements.

How Instance Variables Work

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 II.3-1). 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.

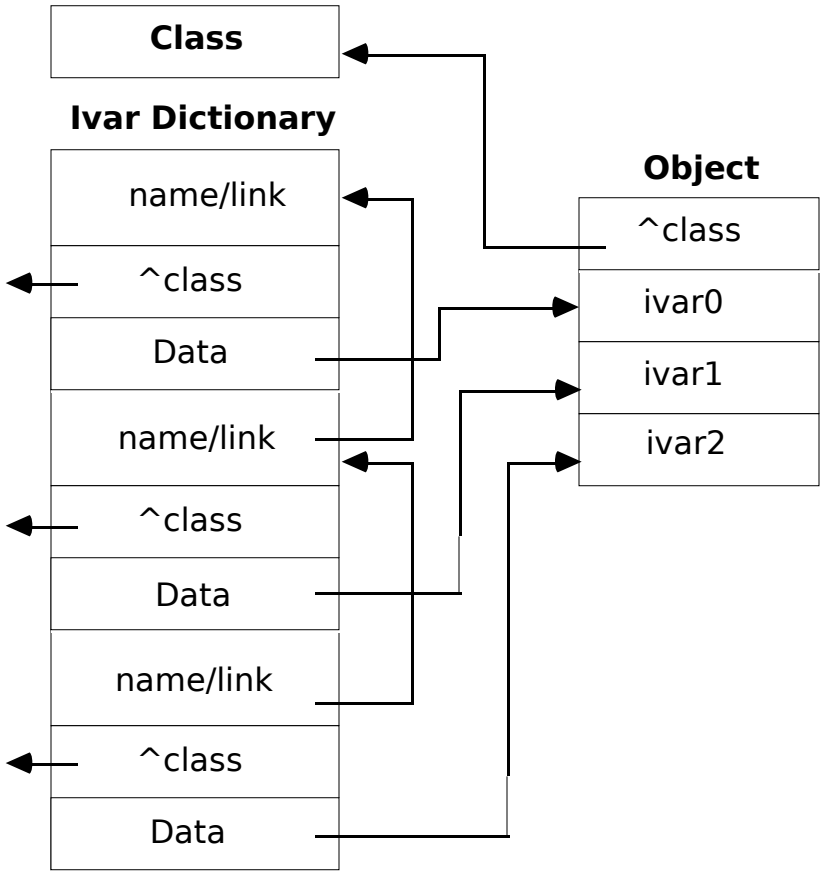


FIgure II.3.1

Figure II.3-1

Instance variables behave as if they were created as objects, but with one major difference (other than the private/public distinction). Figure II.3-1 shows that an object has a pointer to its class (^class -- "pointer to class") in the field that corresponds to the cfa of a standard Yerk word. This pointer actually doubles as a cfa, because the class contains a short sequence of executable code at the location pointed to by objects of that class. An instance

variable attached to an object, on the other hand, has no class pointer, because that would interfere with the mapping of contiguous data cells by named ivars. Notice in Figure II.3-1 how the fields for ivars 0, 1, and 2 in the object are strictly data, without any class pointers.

In a practical example, a Toolbox Rectangle is stored as 4 consecutive 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), knowing that the 2 bytes of data for each integer will be adjacent to one another. If instance variables stored class pointers, your program would have to fish out the integers from the mess of data and pointers before calling the Toolbox routine.

⇒ For Advanced Yerk Programmers:

One possible drawback to an ivar's structure is that only early-bound references may be compiled for most instance variables (see Part II, Chapter 4, for further discussion on early and late binding). Because the class of an instance variable is declared when the class is defined, late binding would be useless anyway; if a loosely bound ivar is desired, it can be accomplished by declaring a Var instance variable that will contain the address of an object, and using references of the form:

```
selector: [ obj: myVar ]
```

The brackets in this example force the message to be late-bound, which allows the effective instance variable's class to be resolved at runtime. The obj: method of class Var is equivalent to the get: method, but makes explicit the fact that the Var is returning an object address.

A minor difference between objects and ivars is that while objects are executable entries in the Yerk dictionary, ivars are not available as Yerk words. That is to say, if you use a Yerk word (e.g., type it at the Yerk prompt), it puts its base address on the stack; but using the name of an ivar will not place its address on the stack. It is rare that the address of an ivar is required, but should it be necessary, your program can send an addr: message to the ivar.

An indexed ivar, however, does have an adjacent class pointer. This is because at runtime, the program can access a specific indexed ivar (i.e., one element of the array holding the indexed data) only with the help of information that is derived from the class's data (this all occurs behind the scenes). Indexed ivars also have, in addition to the class pointer, a 4-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 (see Part II, Chapter 4 for further details).

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.

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 Yerk. BYTES actually builds an ivar entry of class Object and a data length equal to the number that you declare. This means that, if necessary, you can use some of the primitive methods defined in class Object on a BYTES definition (for instance, getting its address with the addr: method). Note 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 II.3-2 shows the format of an instance variable dictionary entry. It bears some similarity to a standard Yerk 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 II.3-1). 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 defined and includes messages to SUPER and SELF, 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.

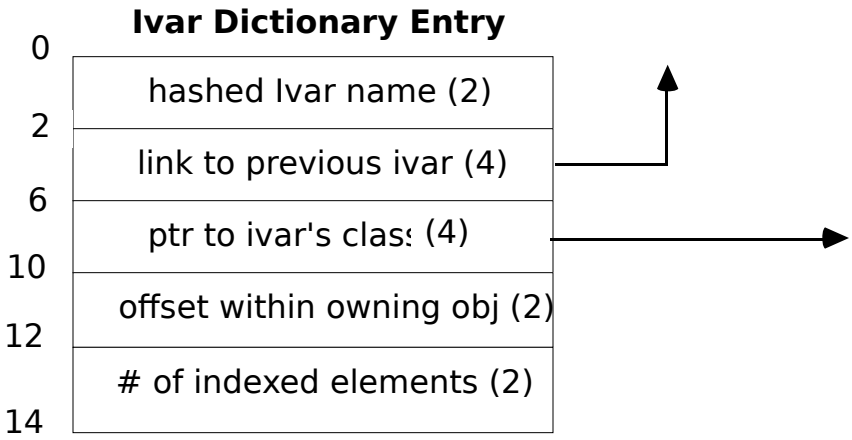

Ivar Dictionary Entry
0
hashed Ivar name (2)
2
link to previous ivar (4)
6
ptr to ivar's class (4)
10
offset within owning obj (2)
12
# of indexed elements (2)
14

Figure II.3-2

Potential Ivar Errors

There is a 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, but if it arises, try a different name for one of the ivars.

Methods Structure

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.

Checking Selectors for Uniqueness

Selectors are hashed just like Ivars, which means that occasionally you could have two selectors with different names but the same hash value. This could produce an inappropriate redefinition of a previous method, and cause incorrect execution of your code. When you first compile the selectors in a new class, watch closely for Yerk's "Method Redefined" messages. If any seem inappropriate, you might have a collision, in which case the only solution is to rename one of the selectors. You can test the hash value for a name with the following word:

```
: .Hash   @word hash .h  ;
```

This word will print in hexadecimal the hash value for a name entered in the input stream (as in .HASH GET:).

Special Yerk 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 Yerk 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 three 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 Object source file, which manipulates its data directly. Several Yerk words described below will come in handy for writing primitive methods like this.

Pointer-to-Base -- ^BASE

You can use ^BASE (pointer to the base address of the current object) from within any method to place the relative base address of the current object onto the stack. ^BASE copies the top of the methods stack, which always contains the address of the currently executing object; ^BASE is also equivalent to COPYM (see Glossary). 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 pfa of the object, which also happens to be the beginning of its named Ivar data area. Use (ABS) or ABS: Self to get the absolute base address for passing to a Toolbox routine.

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 Var, 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 Var this is the entire data area, defined as a single Ivar: 4 Bytes Data. The actual definition of get: in class Var is somewhat different, because it makes use of a more efficient Yerk primitive, M@ (see Glossary). This word combines into a single operation of machine code the operations of copying the top of the methods stack and fetching data from the resulting address. The following words all work in a similar manner for long and word-length operations:

```
M@          ( fetch 32 bit data from addr on the methods stack )
M!          ( store 32 bit data to  the addr on the methods stack )
MW@         ( fetch 16 bit data from the methods stack )
MW!         ( store 32 bit data from the methods stack )
```

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. Yerk has very fast code operations for its most often used primitive methods.

LIMIT and Pointer-to-Element -- ^ELEM

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. After range checking the index value, ^ELEM calls a fast code primitive, called (^ELEM), pronounced "paren pointer to element," which does not range checking.) 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).

Range checking is beneficial for trapping problems while writing a program, but should be turned off when the program is close to completion to speed up compiling and runtime execution. Yerk range checking is controlled by the flags +RANGE and -RANGE. +RANGE turns on runtime range checking, and -RANGE turns off runtime range checking. +RANGE is the system default, and is executed at system startup time.

Other optimized primitives you should be aware of are those that access 1, 2, and 4-byte arrays. Instead of having to use ^ELEM @ for an array fetch, for example, a single code word, AT4, combines both operations. TO4, AT2, TO2, AT1 and TO1 work in a similar manner.

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

The Methods Stack

Because the methods stack plays such an intimate role in supporting messages, you must be very careful if you wish to use the methods stack for your own purposes. You might, for example, want to save a value with PUSHM and later retrieve it with POPM; if you do this inside of a method, and attempt to access an instance variable or SELF while the extra cell is pushed on the methods stack, the wrong address will be on the top of the methods stack. To be safe, it is best to use local variables to save temporary values during the execution of a method.

We suggest that from here you familiarize yourself with the advanced Yerk concepts detailed in Chapter 4.

4
Advanced Yerk Concepts and Techniques

This chapter will acquaint you with some of Yerk's more advanced capabilities -- things that you might need to know if you will be using Yerk to develop commercial applications. Some of the ideas and terms presented in this chapter 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 compilers that operate inside Yerk (these compilers were 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.

MEMORY ORGANIZATION

Because of the way the Macintosh manages memory, Yerk has several distinct areas in which it stores data.

The Kernel

Immediately above the memory area used by the Mac system is the Yerk kernel (also known as the Yerk nucleus). The Yerk kernel (represented on the DeskTop by the "Yerk" icon on your disk) is an image of the lowest, most elemental part of the Yerk dictionary -- that part of Yerk without any predefined classes. It has been designed in such a way that the Finder is able to start it up as an application. To the Finder, the Yerk kernel looks like a very simple Pascal program, one with a single CODE segment. Saved Yerk images, such as Yerk.com, appear to the Finder as documents with Yerk as their owner. Therefore, when you open a saved Yerk image, the Finder starts the Yerk kernel as the application, and passes the name of the saved image file as a parameter in the Finder Information area (see Inside Macintosh for more detail on this). Yerk then determines whether the file is a valid Yerk 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 save many variations of this document without changing the application itself). The memory area dedicated to the Yerk kernel and other specifics of your program is known as the application heap, or simply the heap (information about the System Vector Table, SVT, comes later in this chapter).

Yerk keeps track of information in the heap according to a relative address, with the Yerk kernel starting at relative address 0. Most Toolbox calls, however, require an absolute address, which is easily obtained in a program by sending an abs: message.

Dynamic 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 Yerk

kernel does at startup in order to acquire memory for future expansion of the Yerk dictionary. Yerk 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 Yerk 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 Yerk Install utility, which described in Chapter II.5.

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

Yerk Stacks

The three Yerk stacks -- Return, Methods, and Data (Parameter) stacks -- are allocated above the application heap. All grow downward: the Return stack grows into the base of the Methods stack, and the Methods stack, in turn, grows into the base of the Data stack. The Yerk kernel allocates room for 100 32-bit cells on the Return stack, and 300 cells on the Methods stack. The Data stack is limited only by the maximum address to which the Heap is permitted to grow, and is allocated 9000 bytes in the distributed Yerk 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 Methods stack gets particularly heavy use when you use a lot of named parameters and/or local variables in your methods or Yerk words. When a message is sent to an object and a method executes, the Methods stack (mstack) is set up as in Figure II.4-1. Thus, a Yerk word or method uses (n+2) methods stack cells when it executes, where n is the total number of named parameters plus local variables.

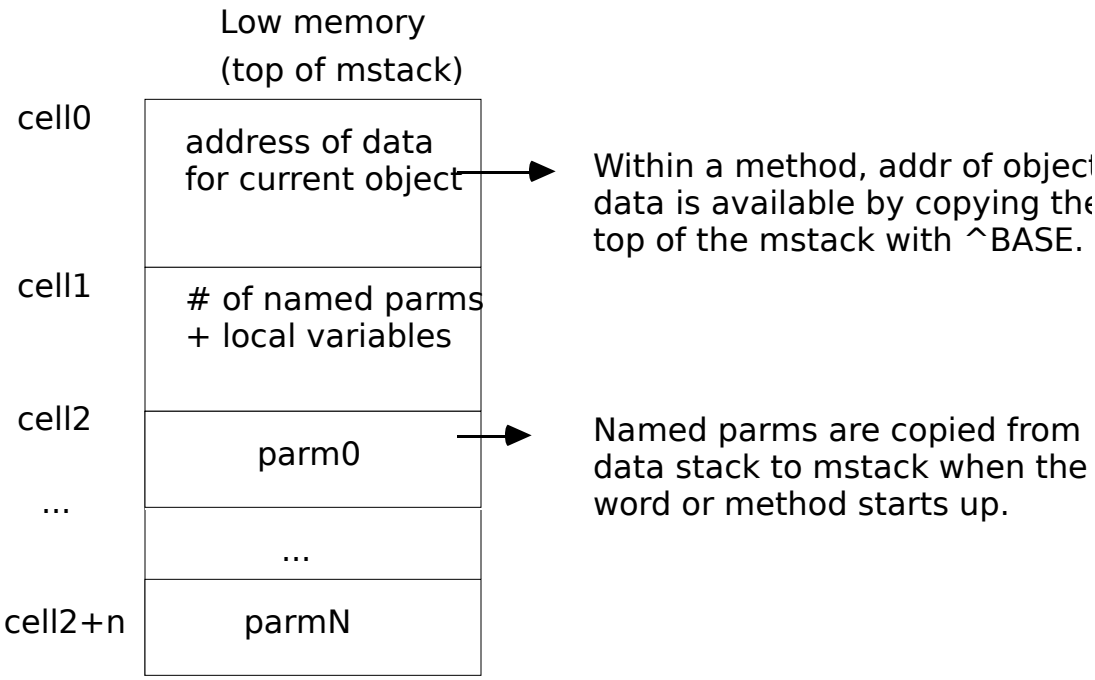


Figure II.4-1

The Data stack is used not only by Yerk, 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 makes the Yerk/Toolbox interface fairly easy. A Yerk word need only place parameters on the data stack, just as if it were about to execute another word or method. 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 a fatal 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 Yerk more stack (using the Install utility, described in Chapter II.5), or adjust your algorithm to nest less deeply on the data stack. (Again, I have never had stack problems).

YERK STRINGS

You may often need to define literal expressions that contain strings or cfas, particularly when you initialize objects. Yerk has several compiler words that might be useful in this respect.

Two String Types

There are two ways of representing strings in Yerk: as STR255 strings or as addr-len format strings. 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, it is therefore 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 absolute address of the length byte to facilitate Toolbox calls.

Yerk 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 Yerk 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
```

Using the name of the scon in your program leaves (addr len) on the stack, and compiles into a single cfa 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 Yerk).

Other String Techniques

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

The word ASCII functions as a literal compiler for a single ASCII character. For instance, the

phrase:

```
Ascii A
```

compiles a literal containing the Ascii code for 'A' (65) in the dictionary, and places the value on the stack at runtime.

There are two primitives for getting a string from the input stream (either keyboard or disk) into the dictionary. WORD gets the next word, without embedded blanks, maps it to upper case and moves it to HERE. Yerk uses WORD to parse the names of Yerk words while

interpreting or compiling. @WORD is a useful macro that takes the place of the frequent phrase BL WORD HERE. For strings, WORD" reads a quote-delimited string from the input stream and moves it to HERE, without mapping to upper case. It is the equivalent of a lower-case version of 34 WORD HERE.

CALLING THE TOOLBOX

Because Yerk's 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 Yerk parameters are 32-bit, you might occasionally have to convert a Yerk cell to a 16-bit Toolbox cell. As you saw in Figure 11.4-1, all Yerk 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 Yerk 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 Yerk 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. Certain calls may require two 32-bit cells to be packed into two 16-bit cells. For example, to convert a Yerk Point to a Toolbox Point, you use the Yerk word PACK. UNPACK does the opposite operation, converting two 16-bit values to signed 32-bit values.

Converting to Absolute Addresses

When the Toolbox requires an address to be passed on the stack, you will generally have to convert Yerk's relocatable addresses to their absolute representation. +BASE converts relative to absolute, and -BASE absolute to relative. The Abs: method will provide the absolute address of any object, and (ABS) provides the absolute address of SELF from within a method. When an object stores a Handle, the value is usually left as an absolute address, and can be passed back to other Toolbox routines directly. >PTR will generate a relative, dereferenced pointer from a

Toolbox Handle. 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.

Procedure and Function Calls

Toolbox routines can be either Procedures or Functions, depending on how the Toolbox responds to their calls. Procedures don't return any values 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 Yerk 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 corresponding Function calls:

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

  Rect  myRect                \ create a rectangle object

  abs: myRect                 \ get an absolute pointer to a rectangle
  rWidth rHeight  Pack        \ pack two 32-bit values into two Integers
  call  InvertRoundRect       \ call the Toolbox routine.

PROCEDURE StdBits (VAR srcBits: BitMap; VAR srcRect,dstRect: Rect;
      mode: INTEGER; maskRgn: RgnHandle);

  QDBitMap  myBits            \ create a bitmap
   Value  maskRgn             \ a place to store a region handle

   abs: myBits                \ pass addr of bitmap for VAR parameter
   abs: myRect                \ pass addrs of two VAR rectangle parameters
   abs: tempRect
   3 makeint                  \ Integer for mode
   maskRgn                    \ get the region handle.
   call StdBits               \ call the Toolbox routine.

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

   :M GETNEW:
        0                     \ make room for returned WindowPtr
        int: resID            \ get this window's resource ID
        abs: self             \ pass absolute 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
  ;M
```

Register-Based Toolbox Routines

Toolbox routines that are not stack-based, but are register based, are a little more difficult to call; but Yerk provides you with several aids that make them much simpler. It is necessary to get the arguments from the Yerk data stack into the appropriate registers, and upon exit from the routine to take the results from the registers and place them on the stack. All toolbox register routines use the registers A0,D0,A1, and D1 for their arguments. Yerk words, such

as popD0, which pops a value from the stack and puts it into the register D0, and pushD0, which takes a value from the register D0 and pushes it onto the stack. The words popD0, popA0, pushD0, and pushA0 are compiled into the system, and the words popA1, popD1, pushA1, and pushD1 are defined in the optional file pops.

These words should only be used in the context of a Yerk code word, that is a word that is actually assembly language code, instead of a high-level word, which consists of cfas that are pointers to other words in the system. All the low-level words in the system are implemented as code words, for example, DUP and SWAP. To create your own code word, begin with the word CREATE followed by the name of your word, and end it with NEXT, which is the equivalent of ; in a code word. For instance a code word to call the toolbox routine FlushEvents would look like this:

```
Create   (FlushEvents)
      popD0                       \ Move parms from stack to D0
      " FlushEvents" asmCall      \ Compile trap number
      pushD0                      \ Return result code to stack
next,
```

Note the use of asmCall to have Yerk look up the trap number for you. This word takes one value from the stack, the combined EventMask and StopMask and returns a result code. If you've forgotten what FlushEvents does, it eliminates all the events from the event queue of the types indicated by the EventMask, until it comes to the first event of a type in the StopMask. It returns the event code of the event that was triggered by stop mask or a 0 if all events were removed. If you were to actually use this word you would probably want to create a higher-level word to call it, like this:

```
:  FlushEvents  { eventMask stopMask -- resCode }
        eventMask stopMask pack (flushEvents) ;
```

Accessing System Variables

In the above instance, what events are allowed on the event queue is controlled by a system variable, the SysEvtMask. If you want to change this variable you may reference it by its location in the parameter RAM ($144 in this case). This is an absolute memory address, so a word to access this can be defined in the following manner:

```
: SetEventMask ( eventMask -- )
        $ 144 -base w! ;
```

Note the use of -base to convert from an absolute address to a Yerk relative address.

If you do not happen to have the variable's memory address on the tip of your tongue, you may use

a Yerk module that looks up global mac locations for you. The above definition could also be defined as:

```
: SetEventMask ( eventMask --)
        global SysEvtMask w! ;
```

The word 'global' looks up the absolute memory location of the Macintosh global 'SysEvtMask' and compiles the code to be identical to the first definition.

SYSTEM SWITCHES

The following switches alter system behavior and will be useful to you during development:

+ECHO/-ECHO	turns screen echo on/off during loading.
+RANGE/-RANGE	turns runtime indexed object range checking on/off.
+CURS/-CURS	turns keyboard cursor on/off

EARLY AND LATE BINDING

As you know, there are two principal states that the Yerk system can be in. When you first start Yerk, 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, Yerk 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 Yerk interpreter sees a colon (or :M) and before it sees the next semicolon (or ;M), Yerk will be in Compile State, and rather than immediately execute the words that it sees, Yerk will compile the CFAs of those words into successive locations in 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.

Binding

When Yerk 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 Yerk dictionary. A very fast search word, called FIND, does this in the Yerk 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 16-bit code known as a hash value. Then the object name, myInt, must be looked up in the Yerk 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 Yerk 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 Yerk 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 Yerk 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 Yerk 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 Yerk provides both late and early binding, you can tailor your application for speed or generality where needed. Even late-bound Yerk 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

Yerk 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 Yerk) 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 Lisa Pascal program, a large Case statement would be required that would handle clicks for different types of windows. In Yerk, 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, Yerk 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 Yerk 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
```

Because an object will leave the address of the start of its Instance Variable space on the stack, any address may be treated as if it is an object of any class. You may get unpredicatable results if the address points to random memory. But suppose you know a particular bit of memory has stored in it the data of a toolkit rectangle. You may put the address on the stack, and use the following generic syntax to retrieve the rectangle's Ivars as if you had instantiated it as an object of class Rect (in this case, 'theClass' would be replaced by 'Rect'):

```
(E) addr get: theClass       \ the address on the stack will be treated as if
                             \ it is an instance of class 'theClass'
```

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 or Vect structure rather than an object, as in example C above. A Yerk data word used in this manner is known as an Object Vector. Because object vectors have so many uses, Yerk allows you to omit the brackets when using a Value or Vect as the receiver of a message:

```
0 Vect theObject
get: [ theObject ]    is equivalent to:
get:  theObject
```

Yerk automatically compiles a late-bound reference whenever a Value or Vect is used in this manner. The resulting code first executes the data word to generate an object address on the stack, and then looks up the compiled method in the object's class. Method lookup can only occur if the object contains a pointer to a valid class in its cfa field.

The procedure, then, to forward reference an object is as follows: first, define a Value or Vect that will be responsible for generating the actual address of the object at runtime. Any references in subsequently compiled code to the object vector thus defined will be late-bound. When it becomes possible to define the class of the new object, do so, and then instantiate the new class using the name of your previously defined vector as an object name. When a class detects that a name used for object creation has already been defined as a vector, it creates a headerless object in the Yerk dictionary, placing its address in the vector. This avoids wasting two name/link fields for a single object. The code for this procedure would look like this:

```
0 Value theObject
    ( ...compiled code references the object vector, as in )
    (     get:  theObject  )

 :CLASS  newClass  <super ...    \ define the class of the object
   ...
 ;CLASS

  newClass theObject        \  using the name theObject causes newClass
                            \ to create  a headerless instance of itself,
                            \ leaving the address in theObject.
```

2. Passing Objects as Arguments

Frequently, you will find it useful to pass an object as an argument to a Yerk 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 or Vect were used, which implies that named parameters can function as object vectors. 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

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 predefine 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. Yerk allows you to tell a class to create an instance of itself in the heap rather than in the dictionary by using the heap> prefix before the name of the class. For example:

```
 6 Ordered-Col  myWindows   \ define a list for the window objects

 \ create a new window object on the heap and add to the list
 : newWind   heap> window   add: myWindows   ;

 \ send a message to the last window in the list
: openIt   tempRect " Title" docWind true true
    new:  [  last:  myWindows ]  ;

\ close the last window and release its heap
: closeIt   close: [  last: myWindows ]                \ close the last window
     size: myWindows  1- dispose: myWindows    \ dispose of its heap space
     size: myWindows 1- remove: myWindows ;    \ and remove entry from list
```

In the closeIt word, the window is sent a late-bound Close: message, and then its heap is released using the Dispose: method of a superclass of Ordered-Col, Array. If an object vector were used instead of an array object, the heap for the object could have been released by using the dispose> prefix.

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 Yerk word whose cfa 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.

USING MODULES

Yerk provides a facility for creating separately compilable, relocatable modules that allow you to build an application that is larger than the memory space of the Macintosh. Modules also 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.

Class names should not be exported, because a class compiles objects that include a pointer to the class as part of their data. This pointer would only be correct if the module were loaded at the same place every time -- something that cannot be relied upon.

In general, compiler words like classes should not be exported from modules because of this hidden behavior. Classes can instantiate objects within a module, however, with no fear of incorrect pointers, because those addresses are relocated along with the module.

Yerk will prevent you from compiling any early bound messages to objects that cross module boundaries, because the names of those objects will look like normal Yerk words to the rest of the application. You must therefore write messages that cross modules in the following manner:

```
get: [ myObj ]   \ myObj exists in a different module
```

Because modules cannot be relocated once they loaded are onto the heap, the heap could become fragmented, or divided into many small blocks that prevent the acquisition of any single, large block. You can minimize this by making your modules relatively small (less than 4k bytes) and releasing them when they are not needed. A Vect called GrowZone holds the cfa of a word that is responsible for releasing unneeded heap when a heap request fails. This Vect normally holds the cfa of RELEASE, which disposes of the heap for all unlocked modules. Your growZone should call RELEASE before doing further application-specific processing.

Because all exported definitions exist in the resident dictionary, modules can conceivably reference definitions from other modules. This could lead to a situation, however, in which memory requirements for the simultaneous loading of several modules exceeds the capacity of the heap.

Finally, the relocation algorithm used by Yerk relies upon all addresses being 32 bits long. If you explicitly store 16-bit addresses in a module, it will not relocate correctly. Use of the Yerk word ' (tick) inside of a module colon definition is dangerous because it could lead to a word literal if the pfa of its target happens to be below the 64K boundary. 'C avoids this problem, because it always compiles long literals.

Another thing to be careful of is to always use the word 'reserve' instead of 'allot' if you need such an allocation. 'Reserve' allots the required memory and clears it to zero. This is necessary since the module loader is a two pass loader, and does a compare of the two loadings to determine what is code. If 'allot' is used, the two memory areas will be equivalent, confusing the loader.

How to Use Modules

Creating modules in Yerk 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 Util1 IMPORT{ Dump Words }
```

This statement declares a module, Util1, from which will be imported two definitions, Dump and Words. 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 first available block of heap. On the disk, the binary image of the module will have the name Util1.bin.

2. Write the source code for the module in a separate file, enclosing the code between the two statements :MODULE and ;MODULE.

```
:MODULE  Util1
  ...      ( source statements )
;MODULE
```

The word following the :MODULE statement must be the name associated with the module's exported definitions in the resident dictionary.

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 execute in the Yerk window a statement of the form:

```
Module fileName
```

FileName is the name of the source file for the module. This invokes a special loader that will generate a relocatable binary image for the module definition specified in the source file. You will see several messages stating that words have been redefined, and then a message stating that the module has been successfully saved. An error in your source file could cause an abort, in which case you should type Forget Task to clear the dictionary of the partial compilation. A successful compilation clears the dictionary automatically. You must have room in your dictionary to load the module source file twice, since that is necessary to generate the relocatable image.

FORWARD REFERENCING

If a situation arises in which you need to reference a Yerk word before it has been defined, you can use Yerk'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 Yerk 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 compiler 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 Yerk 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 YERK

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 Yerk. 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 the GNRL facility of Rmaker (see Putting Together a Yerk Application).

Defining and Using Resources

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

```
'Type WIND 256  rsrc myWind
```

defines a resource called myWind that has type WIND and a resource ID of 256. If you later use the name myWind in your code, it will load its corresponding resource, and return a relative pointer to the resource data. Note that if you do anything that might cause a heap compaction, this pointer will be wrong; you should therefore do any processing that could disturb the heap before getting the resource. The word MSG#, described in the Error Handling section of this chapter, provides an easy interface to resource strings for printing on the screen.

Yerk menus do not have to use the Macintosh resource mechanism for initialization. Instead, Yerk has its own Menu text loader that allows you to place all menu information, including the item action handlers, in the same text file. See the class reference chapter on Menus in Part III for more information. (However, you are perfectly free to use the traditional

resource menu mechanism - the Yerk menu loader was developed before a usable resource editor was available).

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, or ABORT is executed. Yerk uses the file yerk.rsrc for its resources during normal operation, and you can add your own resources to this file with the Apple Resource Editor (ResEdit) or Rmaker.

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 other applications, such as the Icon Editor or MacPaint, 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, executing KEY, and 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. While most Pascal programs are written in this manner, it doesn't necessarily lead to the easiest or the clearest implementation of a particular problem.

Clear the Stacks

Yerk gives you an alternative method by providing you with a Yerk word called BECOME. BECOME causes Yerk to erase everything that currently is on the three stacks, and resets them to their normal empty values. Yerk 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. You will probably find this very convenient for

certain applications, particularly those in which keyboard input is very important. Your application can call KEY from within a nested routine, and then execute BECOME whenever a menu item is chosen that goes into a new state -- the program doesn't have to return to the original caller of KEY after performing the operation selected from the menu.

THE SYSTEM VECTOR TABLE

Yerk 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 Yerk word. For example, let's say that you would like Yerk to interpret a file from disk just as though you were typing it at the keyboard. A built-in Yerk system vector, named KEYVEC, always holds the address of the word that Yerk 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 Yerk 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  .
 'c 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.

Yerk has a full set of vectors for all critical I/O and compilation routines, allowing you to tailor the behavior of the Yerk environment very easily. These vectors cannot reside in the Kernel, since that would preclude having several saved images that used different vectors on the same disk. Thus, each saved image has its own System Vector Table (SVT), just below the start of its dictionary. The kernel routines can then look in the SVT to find the current value for a given system vector. If a 0 value is found, each system vector also has a default word that it will execute. These are the system vectors and their required behaviors:

KEYVEC	(-- char) get keyboard input.
EMITVEC	(char --) send one character to the primary output device.
PEMITVEC	(char --) send one character to the secondary output device.
TYPEVEC	(addr len --) send a string to the primary output device.
PTYPEVEC	(addr len --) send a string to the secondary output device.
EXPVEC	(addr len --) allows a user-installable EXPECT routine.
ECHOVEC	(char --) handles echoing to the output device of the keys being input by EXPECT.
ABORTVEC	(--) clean up the stacks and notify the user of an error. The Yerk word CLEAN2 is normally executed by this vector, and your error word should call CLEAN2 if it is to return to the Yerk 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	(-- pfa 0 t OR f) is a special purpose variant of FIND (this vector

is actually called by FIND before FIND searches the Yerk 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 compilers for Yerk -- a job for only the most advanced programmers.

OBJINIT	(--) initializes certain areas of the kernel at Yerk startup. It normally contains the cfa of SYSINIT. Should not be altered by the user.
PCRVEC	(--) send carriage return to the secondary output device.
BLDVEC	(--) holds the cfa of the word that builds objects from classes. Should not be altered by the application.
CREATE	(--) user-installable CREATE routine. Normally this is set to (CREATE) and you should never have to reset this. For experts only to fiddle with.
INTERPRET	(--) interprets the code in the TIB buffer.
CRVEC	(--) sends a carriage return to the primary output device.

MULTIPLE-CODEFIELD WORDS

Yerk contains two principal facilities for building new data structures: the class/object compilers and multiple-codefield (mcfa) compilers.

Multifunctional Mcfa Words

Mcfa words are like primitive, very efficient objects -- they can have multiple behaviors, but they cannot provide some of the most powerful features of objects, such as nested definitions, inheritance and late binding of methods (described below). Mcfa words are used in Yerk chiefly to build a layer of support for certain areas of a program for which the complexity of the class/object compilers is unnecessary. They can be useful for constructing new global data types, especially constants.

The sample demo file, Tcon, contains an example of using the mcfa compiler to create a text constant called TCON. This word stores text with its various attributes, such as font, size, and so on. When the interpreter executes the name of the TCON, it prints itself in the proper format on the screen.

Colon vs. Code Words

Mcfa words require a little explanation of how Yerk normally executes a word. Figure II.4-2 shows the format of two basic types of Yerk word: Code words and Colon words. Any executable Yerk word must have at least one field containing the address of actual machine code that can be executed by the 68000. This field is called the CFA (for code field address) or Compilation Address. In the case of a Code word such as DUP, its cfa points to its own parameter field, which is the field immediately following the cfa. Thus, a Code word contains machine code in its data area. Code words are the workhorses of the Yerk system, and the actual work that occurs when an application is running is largely the result of executing Code words that have been called by other

words. They are the primitive operations that exist in the Yerk kernel. You can define your own Code words with the Yerk Assembler.

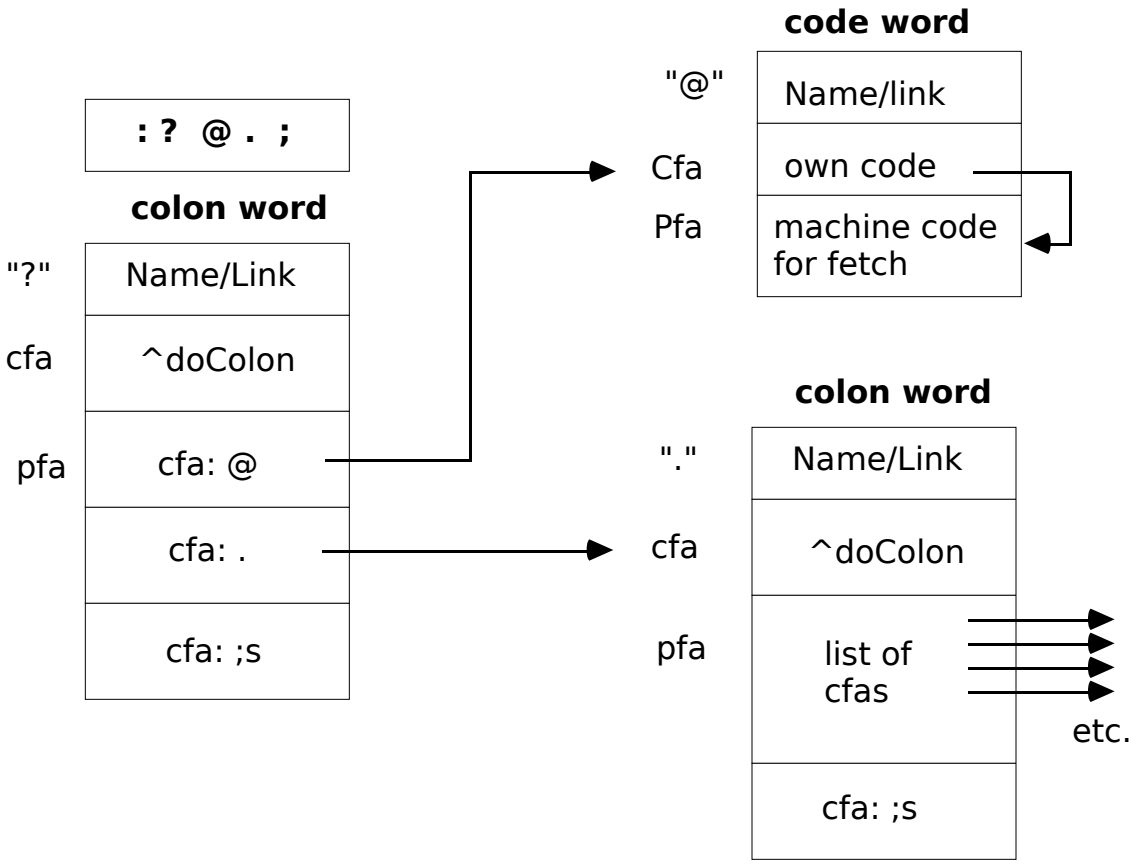


Figure II.4-2

Yerk Definition Dictionary Entries

A word defined in the manner **: wordName ...** ; is called a Colon word. When you define a new word in Yerk using Colon, what gets created is a dictionary entry whose cfa points to a short sequence of code called DoColon. This is not a word, but just a machine code sequence that exists somewhere in the Yerk kernel. The pfa field of a Colon word contains a string of pointers ("threads") to cfas of other words in the system. In the example, the ? word calls two other words, @ (fetch) and . (dot). Thus, the dictionary definition of ? contains at its pfa pointers to the cfas of Fetch and Dot. Any Colon word also ends with a pointer to the cfa of ;S, which is needed to leave the current Colon word and return to its caller.

Yerk executes a colon word by starting at its pfa and going down the list of pointers to codefields of other words one by one, executing each by extracting the contents of its cfa to reach machine code. Since each called word can itself be a Colon word, Yerk could "nest" several levels before

it actually gets to code that does something useful. While this may seem slow, Yerk is extremely efficient at traversing these "threads". Colon words are the essence of a threaded language, because they provide the ability to assemble more powerful words out of existing ones. They make it possible to build an application gradually, aggregating functions at each level into a small, easily understood word. The current version of

Yerk uses a model from Forth called an "indirect-threaded inner interpreter", and you can learn more about this architecture by reading the Forth literature.

Mcfa Word Dictionary Entries

Figure II.4-3 shows the Forth structure Variable, and Figure II.4-4 show its more powerful Yerk successor Value, which has a mcfa structure. A Forth Variable is a data structure that can be created with a phrase like 10 Variable XX (where 10 is the number to be stored, and XX is the name you give to the variable), and consists of a header (the name/link field), a single cfa and a cell of data. Its codefield points to a sequence of machine code called doVariable, which puts the address of the variable's pfa field on the stack. You then must do a @ (fetch) to retrieve the contents of the variable's data. Variables are typical of the kind of data structures available in Forth, and are characterized by the fact that they have a single behavior, indicated by the single codefield. Because this one behavior must handle various operations, such as fetching, storing and incrementing, the best thing to do is simply to put the address of the data on the stack. This leads to pairs of operations such as varName @, varName !, and so on. To do any of the basic operations on a variable thus requires compilation of two cfas: the cfa of the variable itself, and then the cfa of an operator (@, !, +!, etc.)

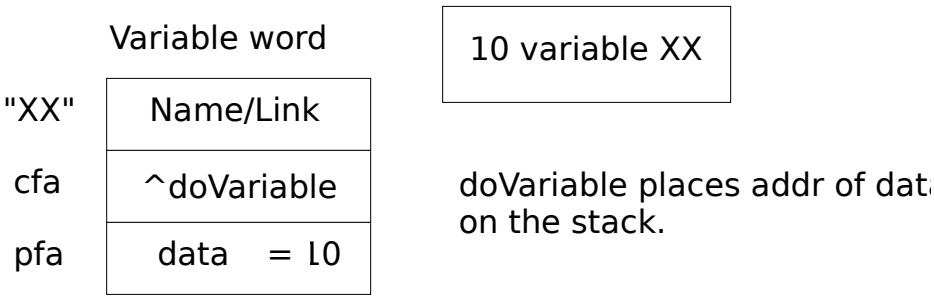


Figure II.4-3

This situation is less than ideal from several standpoints, including readability, space efficiency and processing time efficiency. It certainly contributes much to Forth's reputation as a hard-to-read language. The Forth programmer must be careful to match the proper operator with the proper data type, since the system cannot do any error checking.

In designing Yerk, however, we felt it essential that data structures have multiple behaviors, and that the system should determine appropriate behaviors for a given data type/operator pair rather than the programmer. This led us primarily to the class/object facility in Yerk, but produced mcfa data types as a useful, intermediate step.

Mcfa Words -- A Real-Life Example

Figure II.4-4 shows the internal representation of a Value data type. It has three codefields instead of one, allowing it to have three behaviors specific to its particular data format. The normal codefield (0cfa -- "zero cfa") performs a fetch operation. This is the codefield that would be compiled by Yerk if you simply entered the name of the Value without a prefix. If you used the -> prefix, however, the 2cfa of the Value would be compiled, which produces a store operation. The 1cfa of a Value, invoked with the ++> prefix, produces an increment

operation. Each of these operations compiles to a single cell in the dictionary, as opposed to two each for a Variable operation. In other words, a Value's three cfas share a single data cell, whereas the fetching, incrementing, and storing operations of a Forth Variable must have their own data cells. Thus, Value structures provide both space and time efficiency, and result in more readable source code.

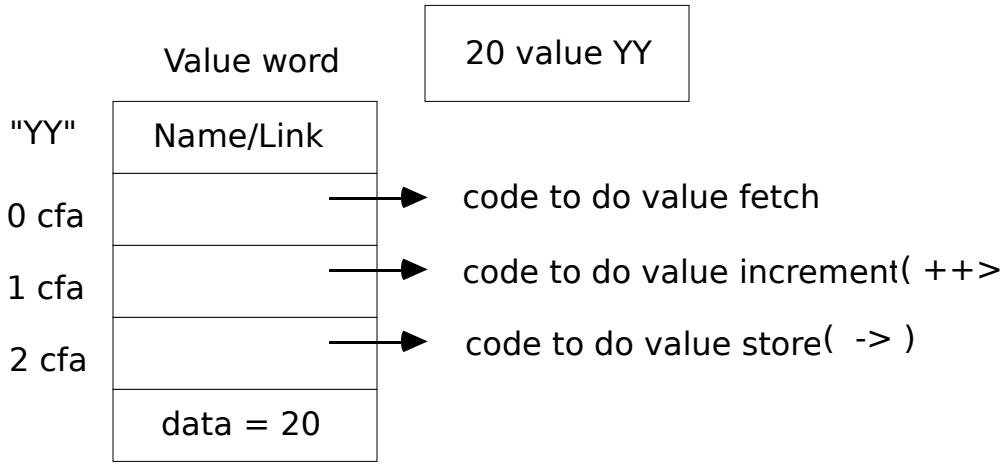


Figure II.4-4

YERK DEFINING AND COMPILING WORDS

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

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 or create a new mcfa data type, you are extending Yerk in a somewhat more profound way than in a vocabulary extension. Creating a new class or mcfa word 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 compilers that can truly extend the

language syntax and add entirely new features. Compilers are the inner soul of the Yerk language. They create control structures (such as IF, BEGIN and DO), Yerk's class compilation facility, message processing, prefix operators...even colon itself is an example of a Yerk compiler word. Yerk's compiler words are borrowed largely from Forth. Some of the most prominent are:

```
Compile
[compile]
,
w,
c,
@pfa
branch
create
s,
str,
```

For Advanced Programmers -- Writing Compilers

You can use Yerk for years without ever writing a compiler word, because Yerk's class/object facility provides a very complete environment for programming. But if you are an advanced programmer and are interested in writing compiler 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.

DEFINING YOUR OWN DEFINING WORDS

Value is an example of a defining word - that is, a word that can itself produce new entries in the Yerk dictionary (e.g., the word YY in Figure II.4-4). We have predefined Value for you, and you can use it to create new instances of the Value type. This is entirely analogous to the manner in which classes define a certain type of object, and can produce new instances of themselves.

How to Create a New Defining Word

Yerk provides a general facility for defining new mcfa data types, which we shall explain by using an implementation of Value as an example (the real implementation is much more efficient).

```
2 Prefix ->                 \ define the new prefix operators.
 1 Prefix ++>

 3 codeFields               \ Values will have 3 codeFields.

 ' ->     Do..   !     ..End  \ 2cfa is the store operation
 ' ++>    Do..   +!    ..End  \ 1cfa is the increment operation
```

```
        Do..    @   ..End   \ 0cfa is the fetch operation.

: Value  Build ,   ..End   \ define the compilation behavior of value
```

```
10 value xx
100 -> xx
xx . 100
```

In this example, we first define the two prefix operators to be used with Value, -> and ++>. This is of the form n Prefix prfName, where n is the codefield number that this prefix will compile (as in 0cfa, 1cfa, 2cfa, etc.). Once a prefix is defined in the Yerk dictionary, it never need be defined again, that is, a prefix definition is global. Note, too, that a given prefix will compile the same codefield number in all of the data types to which it is applied. We then declare how many codefields the Value data type will have. This is followed by a declaration of the behavior for each codefield, starting with the highest one (2cfa in this case). Each behavior is preceded by an optional ' ("tick") prfName, which leaves the pfa of a prefix operator; this will allow the specified prefix to be applied to all instances of this data type. The behavior is then defined between the control structures DO.. and ..END. These behaviors can rely upon the data address of the data structure to be on top of the data stack at runtime, with any parameters supplied by the caller immediately below the top of the stack. Assembler users can define code sequences for the behaviors instead of using high-level Yerk words.

After the behaviors of each cfa are defined, we must give the defining word for the data type a name. In this case the name is Value. The word BUILD must immediately follow the name. It creates the three codefields whenever a new instance is being compiled (i.e., when a new value name is created). We then specify the code that creates the data area for a Value. In our example, we use the simple Comma, which compiles a cell containing the number placed on the stack by the user (nn Value Fred). We are then free to define instances of the Value data type (10 value XX), and all instances will behave in the manner specified.

Prefix Operators

An operator defined using the **Prefix** defining word can only be used with mcfa data structures, and only on a particular codeField. Yerk includes many different kinds of data structures: local variables, named parameters, Values, and so on. For this reason, Yerk contains certain prefix operators that have the ability to work on a variety of different data types. They are:

->	Store into a data structure.
++>	Increment the contents of a data structure.
exec>	Execute the contents of a datum as a CFA.
dispose>	Treat the datum as a heap pointer and release the block.

These prefix operators work on any predefined data type for which they would be appropriate; for instance, the -> operator can be used with Values, named parms, Vects or SysVects.

ERROR HANDLING

The Yerk word ABORT is executed whenever an error occurs in compilation, range checking or runtime processing. ABORT sets all stacks to their empty state, and initializes other Yerk system variables to a suitable value before returning.

Before it executes, ABORT executes the System Vector ABORTVEC to provide a hook for user-specified error recovery. Yerk normally installs its own error handler word in ABORTVEC, called CLEAN2, which prints the current stack of load files, clears this stack, and frees heap that might have been used by Yerk system modules. You should call CLEAN2 if you install an error routine for use during development. For your final application, you should install 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 three error routines that indirectly call ABORT - ABORT", ?ERROR and CLASSERR". 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. It works conditionally in the same way as ABORT". For instance,

```
find not ?error 141
```

prints the string with resource ID 141 if the word in the input stream is not found, and then aborts. Yerk 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.

From within a method, it is desirable to have a more complete statement of where the system was when the error occurred. Methods generally use error reporting of the form:

```
read: theFile  classErr" 150
```

which will give a message like the following:

```
Error from class  ARRAY :: BED0  File read failed
```

In this message, the first part tells you exactly where the error came from; first the class of the object whose method experienced the error, and then the hex address of the object or instance variable. The object's address and not its name is reported because most instance variables and some objects have no name field. Generally, WORDS will tell you the name of the object in

which the error occurred.

For an error while loading with echo off, the last word in the dictionary will usually be the word that experienced an error. The number 150 in the example is a resource ID for the string to print, which in this case was **File read failed**.

The file yerk.rsrc is a resource file containing all of Yerk's error messages. Whenever one of the error words executes, it opens yerk.rsrc to ensure that the text of the message will be found. Of course, you must have yerk.rsrc on the disk that you will be using for

development. You can add your own error messages in a number of ways. First, you can append more strings to the TYPE STR resources using ResEdit to generate a new yerk.rsrc file; or, you can create your own resource file containing all messages for your own application. If you do this, you must ensure that your resource file is open whenever a message is needed. You should use resource ID numbers greater than 500 for your application, because IDs lower than 500 may be used by Yerk or the Macintosh system.

Do not attempt to change any of the resources in the file Yerk, as this will alter the resource map, which could prevent yerk.com and your application from loading. If the resource map is changed, you must restore the Yerk file from a backup disk for the system to work properly.

5
Putting Together a Yerk 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 Yerk 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. The bulk of the important code in your application will probably be in the DRAW and CONTENT handlers of the window objects (these action handlers are explained in Chapter II.3). Menus give the user a means to choose another part of the application or to alter an option setting. Controls usually determine control paths within a given part of the application, or can be used to provide a more convenient mechanism for setting options. Alerts allow the user to respond to error conditions. 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 Yerk and the Toolbox. When your application starts up, it generally must send New: or GetNew: messages to all of these dual role objects (Yerk and Toolbox object) that are needed immediately. Such messages cause the objects to make themselves known to the Toolbox, and to allocate any

heap data that the Toolbox needs to keep track of the objects' states. Then the application will begin listening to events -- thereby becoming sensitive to the user's keyboard and mouse input.

Waiting for Input

There are two ways in which you can structure your Yerk application with respect to keyboard and mouse input. One is to call KEY at those points in your code where you expect keyboard input. In this scenario, KEY will cause Yerk to track and dispatch processing for any other kind of event (predominantly mouse-down events) that comes along until a keystroke is received, at which point KEY gives its results to the object that called it in the first place. Thus, while your application is waiting for keystrokes, the user can use the mouse to select a menu item and cause a change in the control processing of the application, or activate a different window -- anything that performs a different "task" as defined by the window's action handlers. In this kind of structure, however, program execution could revert to the first window (where KEY was called) if there is a keyboard event. This, in effect, constitutes a "mode" that is generally avoided in an event-driven environment, such as the Macintosh. Fortunately, Yerk provides a mechanism for avoiding modal behavior with the word BECOME, which allows you to suddenly "pop out" of a nested call to KEY and begin executing another part of the code (see section II.4). If you have calls to KEY within low-level code, you should execute BECOME whenever a major change in processing occurs, such as an activate event or certain menu choices.

Your application can be somewhat simpler, however, if you design it without nested calls to KEY. Each window object has a KEY: method built in that can handle keystrokes in an object-specific way. You can then make the highest-level code in your application a simple loop of the form:

```
: Listen BEGIN key key: [ frontWind ] AGAIN ;
```

where [frontWind] is a late bound object that calls the FrontWindow Toolbox call, which automatically tracks the current window object (see Early and Late Binding, Chapter II.4). LISTEN is an infinite loop that handles events until there is a keyboard event, at which time it will pass the key's value to the front window and use the Key: method in that window's class. For instance, if your code contained the LISTEN line above, and you defined a subclass of window that had the following Key: method,

```
:M KEY: emit ;M
```

any keystrokes that you typed would be echoed to the screen in the window that the Toolbox detects is the current window.

If, on the other hand, your application only uses the mouse for input -- keystrokes are irrelevant -- you can use the Key: method in the predefined Window class, which does a DROP. Any press of a keyboard key will be ignored.

With this LISTEN kind of structure, all of your code is automatically at the top level any time event handling is active. In a multiple-window environment, you will find that this is the most effective way to design your application, because it eliminates modal behavior and requires less code.

Most conventional computers depend entirely upon keystrokes and command parsing to dispatch control within the program. As a result, a design philosophy revolving around the

keyboard has prevailed over the years. The Macintosh requires a new approach -- one that is best learned by example. The grDemo source file on the distribution disk (and explained in the Tutorial, Part I) provides a simple example of an event-driven Macintosh application written in Yerk, and is worth examining rather closely for the manner in which the program and user communicate with each other.

Using KEY will cause your Yerk application to spin in a BEGIN UNTIL loop, waiting for a keystroke. This is fine if all you want to do is process events and handle keystrokes. However, if you want to perform other tasks in your main event loop, then another way to handle events is to build a LISTEN structure as in the following example:

```
: Listen    BEGIN next: fevent IF drop $ FF and key: [ frontWind ] THEN
                    doSomething
                    doAnotherThing
            AGAIN ;
```

In this example, the program will process events (next: fevent). A keystroke is the only Macintosh event that will leave a true on the stack, so process that event by dropping the modifiers word and using only the low byte of the event message, send the key: message to the frontWindow. All other events will leave a false on the stack, so that part of the code will not be executed (See Part III.4, Events). Other things may then be executed in your event loop. You might want things to happen every second or so, in which case you may set up an interval timer with one of the toolkit classes (TIMER in source INTERVAL). You may also want to use a VBL task or a Time Manager task for such purposes.

CREATING SOURCE FILES

You can use an editor or word processor to create source files for your application. If you use MacWrite or Microsoft Word, you must be sure to save your work as Text Only, because Yerk will be confused by the formatting information that these programs store in standard documents. In most cases, however, it will be most convenient to do all of your editing in Yerk, itself, because you can test parts of your code as you work on the program.

Source File Structure

Source files should begin with a comment block that specifies the following information.

```
\ file Name - explanation of this file's contents
\ 12/24/84 cbd  Version 1
\ 12/25/84 cbd  the froomich no longer twiddles the upper gryphon.
\
\  ... comments for each group of modifications
```

```
\

Decimal       \ just for safety's sake
```

You will find it most convenient to divide your source up into several small files, instead of as one large one. With smaller files, you can load and test isolated pieces of code, and not have to wait for a large file to recompile each time you make a change. Be sure to end each source file with a carriage return at the end of the line so that the first word of the succeeding file will start on a new line.

Modules

Sections of your application that are needed only occasionally can be made into modules. These modular source files have a special format and carry some restrictions as to their use (see section II.4). Modules have the advantage of being separately compiled from the rest of the application, taking up space only when needed, and being in a separate dictionary whose names are unavailable to the rest of the application. Judicious use of modules can cut down on the overall memory requirements of your application. Modules also divide the code into clearly separate pieces, which may help you in organizing your program.

Printing Source Files

You can print your text files to an ImageWriter from within Yerk by using the Print option on the File menu. This produces a formatted listing with line numbers, file name, modification date and time to aid you in keeping track of various versions of your software. If you want to print to a laser Writer, you can use the editor or word processor for printing files.

COMPILING YOUR SOURCE

As you write portions of your program, you can load them into Yerk (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 Yerk each time you want to test the code.

When you load a typical program, you will be doing so on top of Yerk.com (or YerkFP.com), which contains a number of -- but not all of -- Yerk'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 Yerk.com. When you loaded the grDemo in the Tutorial you used a file called demo.load that loaded in several predefined classes before loading in the grDemo code. These classes are available to you as a Yerk user, but need to be loaded before being used. The sequence of loading is also important. In the example, Class ctl was loaded before Class VScroll, because ctl is a superclass of VScroll. You should use the Editor to browse through the optional predefined classes, and familiarize yourself with their capabilities. There is a file in the System Source folder (addl.ld) which is a load file showing a typical load order for additional toolbox support.

Shifting 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. 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.

- As you edit, you might want to test small pieces of edited code to see how they behave. To do this, Copy the fragment into the Clipboard while in the editor, shift to

the Yerk window, and Paste. Yerk will interpret the fragment just as though you had just entered it at the keyboard.

- After you have done a significant amount of work, you will want to reload the source files that you are working with. Save your work in the editor, switch to the Yerk window and use FORGET wordName to erase that portion of the dictionary that contains your compiled words if you had loaded them previously.

- Then you can load your work, either directly with // or by using the Load menu option. The latter allows you to change the drive on which Yerk looks for a source file without changing the default drive.

- In the unlikely event that you made a mistake in your coding, Yerk will report an error of some sort and Abort. To repair the problem, simply switch windows and start the process over.

If you have been accustomed to working with a compiled language like C or Pascal, you might be somewhat startled by the immediacy of Yerk 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 Yerk'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 in this section), by selecting SAVE DICTIONARY AS... option 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 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 may also select the SAVE DICTIONARY… menu item, which will save an image of the current dictionary using the current name of your document and placing it in the Yerk Folder. The Save As item allows you to change the name and specify where you want the image saved.

There is a Yerk word, 'saver', which will save the document you are working on to the folder of the orignally launched document, using the Yerk window name as the filename. If a file by that name already exists in the folder, it will be replaced by the newly saved image. No warning will be given. This is useful to make quick saves of the same document...but be careful not to delete the originally launched document (by first using SAVE AS... menu item to change the working document's name).

You can have several saved images on a disk, all using the same nucleus. Because each saved image has a separate System Vector Table, each can constitute an entirely different application. Be sure, however, to leave enough room on the disk for further additions and changes to your program.

Speedy Recompiling

In all likelihood, you will have several source text files under construction at a time -- or at least in such a fluid state that saving them as binary images would be unwise. And as you

debug a file, you'll want to re-load all those files atop Yerk.com or whatever binary image you've saved up till then. While it may seem that you'll be issuing streams of load commands (either the // command from the Yerk window or Load... from the File menu), you can actually use Yerk to write an easy shortcut.

Phrases like **// Filename** can be embedded within source files to perform what amounts to batch processing of a series of load commands. You can build a load file that contains load statements for each of the source files in your application -- a procedure that documents the load order and makes it easy to reconstruct the application from the bottom up. Another use for this facility is to print the amount of space occupied by each file as it is loaded, using a word such as ?BYTES:

```
 0 value oldRoom
 : ?Bytes  oldRoom  room - .  ."  bytes used"    room -> oldRoom  ;
 // file1
 ?Bytes  \  prints the amount of space consumed by file 1
 // file2
 ?Bytes  \  prints the amount of space consumed by file 2
 ...
```

Simply load this load file, and the individual source code files will be automatically loaded in the proper order every time you need them. And since this file does not define any new Yerk words, it takes up no memory space once the loads are finished.

Incidentally, Yerk has a powerful file stack facility that allows you to nest loads up to six deep. By embedding a load command inside a file you are loading, the Yerk loader stacks the currently open file (i.e., temporarily interrupt loading of one file) and begin loading the new file. When the second file load is complete, the load of the original, stacked file resumes on the line following the // statement.

Other Compiling Tips

When loading a file that has never before been compiled, select Echo During Load from the Yerk menu to cause each line of the file to be echoed (printed) 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, Yerk prints the contents of the file stack -- the file at the top of the stack is this file containing the error. You can pause a 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).

As mentioned earlier, the Yerk.com file, as distributed, contains support for the utilities in the Yerk menu bar (from the source file FrontEnd). Since you will not need this support in your final application, you could dispose of the FrontEnd support with the statement FORGET FRONTEND, before compiling your dictionary in the final version. In the meantime, it serves a useful purpose in helping you debug your program and generally manage the Yerk programming environment.

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 Yerk prompt), setting up reasonable parameters on the stack, and then using the .S stack dump to determine if the results are correct. Another good tool for examining data is EXAMINE MEMORY on the Utilities menu, which provides a hex dump of the entire memory. From within that utility, you can also scroll or search for the memory dump of a particular word or address. Also, you might need to use the Echo to Log (+file) menu option if you want a record of any error messages.

Evaluating Yerk Error Messages

It's quite possible that in the early stages of program development, you'll generate a Yerk 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, Yerk might catch an error that is actually an indirect result of another problem which Yerk 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 systems software before Yerk becomes aware of it. In these cases, you will see an Alert box with a bomb icon and an error code displayed -- the familiar "Bomb Box" (unless you have the debugger installed). The most common system error codes are 2 (when the 68000 tries to perform a word or long operation on an odd address) and 3 (when the CPU attempts to execute data as code). Most of the time, you can recover from these errors by clicking the Resume button in the Bomb box and begin debugging right away (see the next section if you are using Macsbug). Occasionally, however, enough "damage" will be done to the heap or 68000 register contents to necessitate a Restart, which you can perform either by clicking the Bomb box's Restart button, or pressing the Reset button on the programmer's switch. In either case, you will be re-booting the system. Therefore, be sure you save your source code files before loading them or before pasting selected text from

the Editor window into the Yerk window. 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 CPU's). The 68000 processor has a set of instructions that are optimized for even addresses, and some Yerk 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 address. 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 Yerk 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 Yerk word calls a Toolbox routine that uses a lot of stack. You can use the Install utility (described below) 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 Yerk or the Macintosh Operating System could detect it. The only choice here is to Restart (Reset) 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.

Macsbug

We strongly recommend using the Apple Macsbug utility for debugging. It is available from many bulletin boards, users group, and ftp.apple.com on Internet. If you bomb, you wind up in the debugger, and by after you examine the stacks (A6 dump is helpful to show you the return stack chain that got you into your problems), you may type 'g applscratch' or 'g A78' to return to Yerk. Sometimes your error may have wiped out part of the Mac system, or part of the Yerk dictionary, in which case you may have to type 'es' (escape to shell) to get back to the finder, or even 'rb' (reboot) to restart the Mac.

THE INSTALL UTILITY

The Install utility, available from the Utilities menu or by typing INSTALL, enables you to tailor Yerk's startup memory allocation to fit the requirements of your application.

Assigning Memory

When Yerk starts up, three things are competing for the available memory: the data (system) stack, the dictionary, and the dynamic heap. The system stack must be given enough room to grow with both Yerk's parameters and the data for Toolbox routines that Yerk calls. If the stack runs out of room, a fatal System Error 28 will result. Yerk must also acquire enough memory from the applications heap for its dictionary when it starts up. After both the system stack and the dictionary are allocated, what's left over is given to the application heap for dynamic use by resources, modules, heap objects and so on.

You must balance the use of these three memory areas in your own application. Moving code from the resident dictionary to modules transfers memory usage from the dictionary to the dynamic heap, allowing you to turn otherwise unwieldy programs on a 128K Mac. Objects can be allocated in either the dictionary or the heap. You don't have a lot of control over how much room the stack will need while the program is executing, because the deepest intrusions of the data stack into the heap take place in the course of Toolbox routines, which run autonomously once they are called. Each application you write will have a somewhat different memory usage profile, so we've provided the Install utility to allow you to set the quantity of memory allocated for each region.

Install Choices

If you choose Install from the Utilities menu (or just type in the word Install), you will see a Dialog box that has boxes showing the current values for stack, heap and dictionary headRoom (room remaining) allocations. The value for the stack describes the area that Yerk will reserve exclusively for the system (data) stack at cold start. This value also sets the maximum address that can be used by the heap.

At the bottom of the Dialog box, the dictionary value is actually the amount of room remaining between HERE and the end of the dictionary block. For this value to be useful to you at this point, you should have already built up the dictionary with all of the code for your application. The controls to the right of the stack and dictionary boxes allow you to alter the amount of stack space and dictionary room. As you adjust either value, you will notice that the heap value changes accordingly. Increasing either stack or dictionary takes away from the heap, and vice-versa.

Each memory area has a minimum value you can specify. For the stack, the minimum is 3000 bytes, a number that we have determined by experiment. Many applications will do perfectly well with 3000 bytes of stack space. If you begin experiencing System Error 28, however, you will need more space. The heap cannot be smaller than 8000 bytes -- the minimum at which desk accessories can be functional -- and should handle minimum resource requirements. 22K is more realistic for interactive use of Yerk with its modules. Dictionary room cannot go below 128 bytes, which protects your application in case it executes code that moves a string to HERE. If you know that your application uses more room at HERE during its execution than 128 bytes, this figure will have to be increased accordingly.

Yerk uses a memory allocation strategy that attempts to optimize its use on both large and small address-space computers. When Yerk starts up, the Nucleus contains a minimum heap value (set by the middle box in the Install Utility). This value is the minimum amount that Yerk will give to the system for general use. On a small Macintosh, this value will guarantee that the system has enough to function effectively, and the rest will be taken for dictionary space. On a large Macintosh, it is desirable to give more space to the system because this means less disk access and therefore faster response times during development. To accomplish this, Yerk has in its nucleus a maximum dictionary size value that will limit how large the dictionary can be on a large Mac. This will guarantee that the system will have more heap to play with if lots of heap is available. Yerk is distributed with a maximum dictionary size of 300,000 bytes, but you can alter this value with the word MAXDICT, for example:

```
350000 maxDict
```

After setting maxDict, you should use the Save button in the Install utility to save the nucleus to disk. Dynamic heap is set in the 'Get Info' box from the finder.

Using Install

In general, your Install strategy should be as follows. Execute Install and adjust the dictionary room figure to the minimum that you will require for your application at runtime. This will give any extra room in the dictionary to the heap. In most cases you may simply click on the "Max Heap" button. Then adjust the stack figure to the minimum usable value that you are confident will not produce stack overflow. If you don't know, you can start

with the default value of 3000 bytes and increase this number if you get Error 28 during testing. This will provide the maximum amount of memory to the heap, which is really the goal of this procedure.

After you set the quantities for both dictionary and stack, you can use the Save button in the Install Dialog box to actually store these parameters in the Nucleus (which, at this point, must still be named Yerk on the current disk). Save will cause Yerk to terminate and return to the Finder. When you start Yerk again, it will be with the new values.

Remember that if you have set the dictionary room to 128 bytes, you can't start up Yerk and start compiling, because you'll immediately run out of room. The dictionary should therefore be set to its minimum only after your application is completely compiled, debugged and ready to roll. You might want to use Install during development to adjust the amount of stack space, but in this case you should leave only enough dictionary space to keep the system usable interactively, while your dictionary grows.

You might get yourself into a situation where the values that you've set for the Nucleus on your disk make it unusable, and you can't even start Install to change them. In that case, just copy the Yerk file from your archived Yerk System disk to your working disk, and you should be able to run Install again.

After you have values that work for your application and everything else is ready to go, use the Install button instead of the Save button. Install is just like Save except that it will allow you to create a stand-alone application rather than just saving the Yerk nucleus with different stack, dictionary and heap values.

INSTALLING YOUR APPLICATION

Installing your application means setting it up so that your application is a stand-alone, double-clickable application on the desktop. The application runs immediately upon startup, which prevents the user from being able to access the Yerk interpreter. You may, if you choose to do so, keep a trapdoor in your code to let you (or anyone else who knows how) to reach the interpreter and use it to compile new words.

You must understand the difference between a stand-alone application and a development document. You have to think about how the application is going to work as opposed to how it worked as a development environment. Eliminate all openrsrc calls (I patch the old ones [to null]), use menus in resources (see Section III.6), don't read in a path file, and get rid of some of the unnecessary words in your development startup word. Then run the install module, setting the various words appropriately. Certain modules will load as Code resources, and you may delete those you don't need with ResEdit. Add any other resources you need from the development

resource files, and you are ready to go.

The key is to think of how a real application will work as opposed to the development document.

Taking the original startup word, yerk, from the FrontEnd source (with line numbers):

```
\ system startup word

 1 : yerk
```

```
2       sysInit \ Initialize nucleus objects - fFcb, fEvent, fpRect, fWind
3       " fpInit" sFind IF drop cfa execute THEN     \ Initialize FP system
4       0 ?event drop abs: fWind call BeginUpdate
5       getVrect: fWind 14 + put: tempRect update: tempRect
6       abs: fWind call EndUpdate
7       initNewWindow: fwind show: fwind
8       <[ 2 ]> 'cfas enfW disfW setAct: fWind       \ fWind activate activities
9       OpenNR
10      new: imageName  new: parmStr
11      nPath
12      nMenu                                       \ get Yerk menu bar
13      initProcs                                   \ loads all proc words with a5,a3
14      myDoc 2dup put: imageName title: fWind      \ fWind title bar
15      ?yerkFlgs release ;
16
17 'c yerk -> objInit
```

Here are the changes that should be made to change this word to the cold start word, if you wanted the application to run the Yerk interpreter on startup:

1. Eliminate line 2, since sysInit has already done its job and the application will store the nucleus as it is. It won't hurt to leave it in, but you don't need it.

2. If you use floating point, substitute line 3 with "fpInit"..just execute it, since you know it's there.

3. Drop the update stuff in lines 4-6 (It doesn't hurt to leave them in, but it is just unnecessary).

4. Eliminate lines 7-8 for the same reason as point 1.

5. Eliminate line 9, since your resource file is now the resource fork of the application. If your program includes references to openNR, or your own word to open your development resource file, patch them as:

   ```
   patch openNR null
   patch myRsrcOpen null
   ```

   It is better to not use any resource opening words within code of your application. Do them at compile time. If you leave them in your code, the Mac Resource Manager may not be able to find a resource you want..particularly if the file is not distributeded with your application. I use resedit to copy all resources I need in my application to the resource fork. This can mean the STR rsrcs from yerk.rsrc also.

6. Eliminate line 11, unless you want to include the npath.txt file in the yerk folder and have the yerk folder around..If you distribute the application, you don't want to do this. In fact, a path file

for an application is unnecessary. The user will always use the std file dialogs.

7. Redefine 'nmenu' in line 12 to read menus from the resource file (if you use the old, non-supported way of using menus).

8. Eliminate line 14 since there is no longer a document. The name of the fwind is in the resource file.

So, the resultant startup word should look like:

```
\ system startup word
1 : yerk
2
3       fpInit \ Initialize FP system
4
5
6
7       initNewWindow: fwind show: fwind
8
9
10      new: imageName  new: parmStr
11
12      OpenMyRsrcMenus
13      initProcs
14
15      ?yerkFlgs release ;
16
17 'c yerk -> objInit
```

Installing Your Program As a Finished Application

Here are the steps that you should take to install your application once it is complete and debugged:

1. Enter the Yerk interpreter and compile your entire application up to the top.

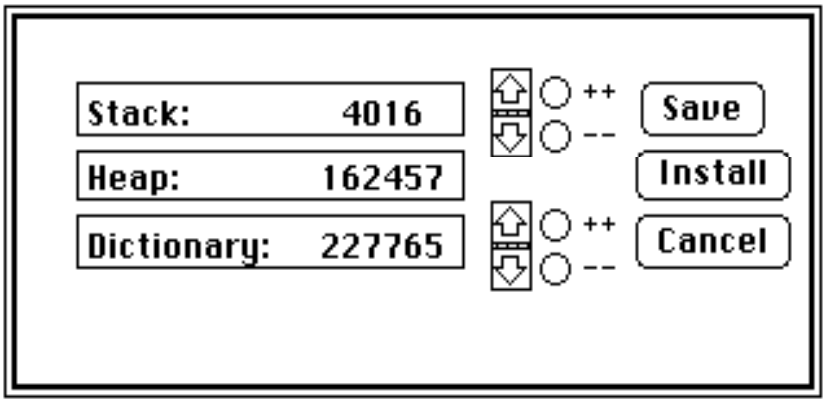
2. Define three Yerk words (colon definitions), one that initializes the application (cold start word - see above example), one that runs the program itself (more like a warm start word), and another that enters an appropriate error recovery routine in case Yerk aborts.

3. Save the resulting dictionary to a new file.

4. Compile your modules, and perform any special data creation you require.

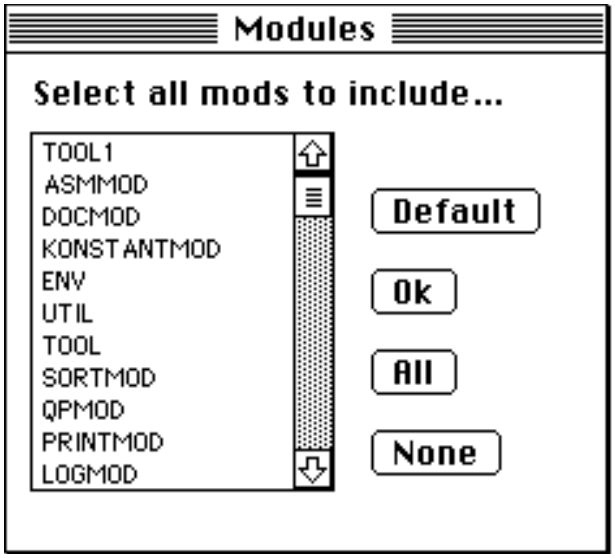
5. Make a duplicate of your disk, and use the duplicate for the following steps. Keep the original handy for future debugging sessions.

6. Double click your saved application dictionary and run the Install utility. A dialog will appear that will let you select the stack and dictionary values. Usually you would set these values to minimum values that you know will work with your application (these values may



already be set from your debugging procedure). Click the Install button.

7. A window will appear that will allow you to select those modules which you need to have in your application. Any module that your program calls should be selected. If you use no modules, don't select any. The modules will be compiled into the application as CODE resources. When you have selected what you need, press Ok.



8. Another dialog will appear after clicking Ok in the Modules Window. Set your unique application file signature and document file types in the dialog box. (Contact Apple to register your unique signatures). Enter the version string for your application; usually the product name and version number. Provide the name of your application; this will be the name of the application file on the disk. Use a name the user will recognize as being related to the installed application. Provide the three words which you defined in Step 2. The cfa of the cold start word will be placed in the system vector OBJINIT; the warm start word's cfa will be placed in the system vector

QUITVEC; and the error word's cfa will be placed in ABORTVEC. Click on the fWind visible check box if your application depends on Yerk's window to be visible when it runs. Click on delete dictionary and/or delete modules if your install disk is a floppy and does not have room for two copies of the dictionary and/or modules. (You will not want these files on the final disk anyway). When you click on the OK button your application will install and you will eventually be back in the Finder.

9. You now have created an application which is a resource file. But you must still add all the resources your program needs that may be in the yerk.rsrc file or your own development resource file. Use a resource editor like ResEdit to copy those resources to your application. You may also want to change the SIZE resource to reflect your application's needs, rather than Yerk's needs. You will also want to replace Yerk's Icon resources (see next section) with your own.

9. After you are back in the Finder, you must rebuild the desktop. Put the disk in the trash then reinsert it while holding down the option and command keys simultaneously. This will recreate the DeskTop file so that it has the correct resource information about your application (and your own icons will appear on the desktop if you created them (see next section). You may now move your application to other disk volumes.

YOUR APPLICATION'S 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 Yerk and your application.

You will want to create your own icons for the Yerk nucleus (which, to the user and the Finder, is your application) and documents that will be owned by the application. You can do this most easily by using ResEdit. From ResEdit, open YERK, click on the ICN# resource, do a Copy and close YERK. Create a new file, for example "my icons", and do a Paste. Double click on ICN# to view the existing Yerk icons. To design your application icon, double click on Yerk's icon, the last one in the set, and work on the "fat bits" image transforming it into you custom application icon. Similarly, if your application has its own documents you may transform any of Yerk's document icons into your custom icons.

Later, after you have installed your application, edit the "my icon" file with ResEdit; choose the ICN# resource and do a Copy; then open your application file, click on its ICN# resource and do a Paste. This replaces the Yerk icon set with your application icon set. When you reset your disk as in step 9 above, the standard Yerk icons will be replaced by your customized set.

6
Utility Modules

The Yerk Disk 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 Yerk 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 Input Dialog

The Module InDlg provides a general-purpose input dialog that displays a prompt which you provide, and accepts the user's input from a text-edit item. The text is left as an **addr len** on the stack.

DoInDlg has the following stack context: (addr len -- b). The addr len cannot be entered as a quote string. As input, doInDlg accepts a string that will be the prompt presented to the user. It returns a boolean denoting whether the user hit OK or Cancel (false if Cancel). You should only process the input if the boolean is true. Several examples of calling this and other utility modules are contained in FrontEnd and Exam.

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. Use Alert" in place of Abort" in your final application to comply with the "Mac Standard". You may modify the sizes and positions of any of the items in the module source file "AlertQ", but do not add or delete

items or change the length of the item data. The alert DITL has three items: 1) the OK button, 2) the error number, and 3) the message.

The action taken after the alert box is dismissed depends upon the contents of X-Array "Aact". Aact has four elements; one for each stage of the alert: 0=2 and >2. By default, stage zero will execute abort, while stages 1, 2, & >2 will do nothing allowing the program to continue after the point where the alert occurred. You may modify these vectors. Examples:

```
'c bye 0 to: Aact                    \ StopAlert exits to the Finder
```

```
4 'cfas null null null null put: Aact    \ all alerts do nothing
'c abort fill: Aact                      \ all alerts abort, (executing the error word)
```

You may add alerts to your resource file by using ResEdit. Then call the alert by using ID# N ALERT!, where ID# is the resource ID of the alert, and N is a number 0 through 2 (depending on the system ICON you wish to display). You may have up to 4 control buttons in your alert, but most have only one. When you click on a button, its number will be returned on the stack (0-3).

The Print Module

This Module is invoked whenever you wish to send output to the printer. There are words for both direct output (pEmit, pType, pCR) and vectored output (+print). +Print installs the printer emit, type, and CR words in Yerk's secondary output vectors, and -print causes these vectors to revert back to null behavior. Only character I/O using the printer's native font is supported.

The Sort Module

Sort is a general purpose shell sort which operates on 4-byte ordered lists. This can be an Array, an Ordered-Col or simply an allocated section of memory. To use Sort, provide the address of the 0th element of the list, the number of elements in the list and the CFA of a comparison word to be used by the sort algorithm. The comparison word must take two values off the stack and return a result of: -1 -- val1 < val2; 0 -- val1 = val2; 1 -- val1 > val2.

To sort data other than integers, store pointers to your data in the list and write your comparison word so that it operates on the referenced data rather than on the values themselves. To sort strings, for example, you would construct your list so that each element would be a pointer to a string then design the comparison word to prepare the stack and use $= to perform an ASCII comparison. By appropriate design your comparison word can produce an ascending or descending result. In general your list can represent anything you need it to; integers, pointers to str255 strings, pointers to fixed records, handles to blocks of storage in the heap, etc.. See "sort" in the glossary.

The Decompiler

This utility decompiles Yerk words, objects, classes, and other Yerk definitions. To decompile a Yerk word pull down the Utilities menu and select Decompile, or type:

```
de' wordname
```

if the subject is a method, type:

```
de' methodName: className
```

if the word exists within a module, type:

```
de' wordName|modName
```

The decompiler has several options in its dialog box. The default option setting will display Yerk colon definitions nearly as they might have appeared in the original source and objects showing their data (with nested data constructs and indexed data). Options to display

absolute, relative or offset positions within definitions are available for words, objects and classes and an option to display superclass data for objects is also available.

Another aid to browsing through code is the 'DocMod', a module that among other things will bring up a window to display the sourcefile containing a given word, class, or object. There are eight import words from this module: see marks srcName fm rl /// mforget findfmark. These handy development words are defined in the Yerk Glossary. Two other words are +docs and -docs, switches to turn documentation loading during compile on and off. If documentation is on, then every word compiled into the Yerk dictionary will have 2 extra bytes in the header (prior to the name field) and each file loaded will have its name also compiled into the dictionary. Using the imported word 'see' you can bring up a scrollable window of the source file that contains the word after 'see'. During development, you might decide to have +docs turned on, but when you finally create an application, you would recompile with documentation turned off to save space.

The global value 'docs' is defined in the nucleus, and is set initially to false. If you turn documentation on by typing '+docs', this value is set to true. However, because it resides in the nucleus, 'docs' will not be permanently set when you save your development document. When you launch the document, you will have to type '+docs' again to turn documentation on. To save it permanently, use the import word 'SaveNuc' which will save the nucleus as is.