

Design Master™

C. K. Haun

Byte Works, Inc.
4700 Irving Blvd. NW, Suite 207
Albuquerque, NM 87114
(505) 898-8183

Credits

Design Master Program

C. K. Haun, RavenWare Software

Documentation

Barbara Allred

Cover Art

Sam Day

Sample Programs

Barbara Allred

Product Manager

Barbara Allred

Copyright 1990
By The Byte Works, Inc.
All Rights Reserved

Copyright 1986, 1987
Apple Computer, Inc.
All Rights Reserved

Limited Warranty - Subject to the below stated limitations, Byte Works, Inc. hereby warrants that the programs contained in this unit will load and run on the standard manufacturer's configuration for the computer listed for a period of ninety (90) days from date of purchase. Except for such warranty, this product is supplied on an "as is" basis without warranty as to merchantability or its fitness for any particular purpose. The limits of warranty extend only to the original purchaser.

Neither Byte Works, Inc. nor the authors of this program are liable or responsible to the purchaser and/or user for loss or damage caused, or alleged to be caused, directly or indirectly by this software and its attendant documentation, including (but not limited to) interruption of service, loss of business, or anticipatory profits.

To obtain the warranty offered, the enclosed purchaser registration card must be completed and returned to the Byte Works, Inc. within ten (10) days of purchase.

Important Notice - This is a fully copyrighted work and as such is protected under copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to \$50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a \$10,000 fine.

This product is sold for use on a *single computer* at a *single location*. Contact the publisher for information regarding licensing for use at multiple-workstation or multiple-computer installations.

Code Generated by Design Master - The Byte Works Inc. makes no claim whatsoever to files created with Design Master.

Apple Computer, Inc. MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

GS/OS is a copyrighted program of Apple Computer, Inc. licensed to Byte Works, Inc. to distribute for use only in combination with Design Master. Apple software shall not be copied onto another diskette (except for archive purpose) or into memory unless as part of the execution of Design Master. When Design Master has completed execution Apple Software shall not be used by any other program.

Apple is a registered trademark of Apple Computer, Inc.

A word from the author...

A program as involved as Design Master could not have been created in a vacuum, and I'd like to acknowledge some of the folks who have helped along the way.

All of the folks on GENie who saw it from its raw beginnings and added encouragement and comments as it grew.

The folks in System Engineering at Apple who wrote the toolbox and keep improving it so programs like Design Master are possible, and can get better.

Lora, for putting up with nearly a year of, "No, I can't go out this weekend, I've got to work on Design Master."

Mike Westerfield and Barbara Allred at Byte Works for publishing it and doing the really hard stuff like writing this manual. If it had been left up to me, you'd be looking at three pages of dot matrix print.

Design Master was written in 65816 assembly, and required 4380 cigarettes, 68.2 gallons of coffee and 600 hours of Lou Reed to produce.

C. K. Haun

Table of Contents

Chapter 1: Introduction	1
Who Should Use Design Master	1
What You Should Have Received	2
System Requirements	2
Required Hardware	2
Required Software	2
About This Manual	3
Visual Cues	3
Additional Reading And Reference	3
Chapter 2: "Drawing" A Program	7
Getting Started	7
<i>Back-up Design Master First!</i>	7
Starting Design Master	7
Preparing to Create the Text Editor	8
Design Masters Menu Bar;	8
Creating A Menu Bar	9
Creating A Dialog Box	18
Creating An Alert Box	22
Creating A Second Alert Box	24
Creating A Window	25
Chapter 3: Writing A Text Editor In A Few Hours	31
Designing The Text Editor	31
The Preliminary Design	31
The Detailed Design	32
The MiniWord Program	34
The Main Program Module	35
The Main Program	35
The Init Function	35
The InitMenus Procedure	40
The EventLoop Procedure	46
The MyWindow Function	52
The ShutDown Procedure	54
The HandleUpdate Procedure	55
The HandleSpecial Procedure	55
The HandleMenu Procedure	56
The DoQuit Procedure	58
The Error Unit	58
The InitError Procedure	58
The HandleError Procedure	61

Design Master	
The Cmds1 Unit	63
The InitCmds1 Function	63
The DoAbout Procedure	67
The DoClose Procedure	68
The WantToSave Procedure	70
The DoPSetUp Procedure	71
The DoPrint Procedure	74
The Cmds2 Unit	83
The InitCmds2 Function	83
The DoNew Procedure	89
The InitWindow Procedure	89
The GetUntitledName Procedure	90
The CreateWindow Function	91
The FiniWindow Procedure	92
The DoOpen Procedure	93
The GetOpenName Procedure	98
The DoSave Procedure	99
The GetText Function	102
The DoSaveAs Procedure	104
Creating The Final MiniWord Program	109
Embellishing The MiniWord Program	111
Chapter 4: Using Design Master to Create Resources	113
Using Resource Forks With MiniWord	114
Changing The Globals Unit	114
Changing The Main Module	115
Changing The Error Unit	118
Changing The Cmds2 Unit	119
Creating The MiniWord Program	125
Chapter 5: Command Reference	129
Coloring" With Design Master	129
Accessing Disk Files	130
The Apple Menu	131
About Design Master	131
Help	131
The File Menu	131
New Window...	132
New Dialog...	135
New MenuBar...	137
Open DM file... (⌘O)	141
Save... (⌘S)	141
Save As...	141

Table of Contents

Append...	143
Close (⌘W)	143
Delete file...	144
View file...	144
Quit (⌘Q)	145
The Edit Menu	146
Undo (⌘Z)	146
Cut (⌘X)	147
Copy (⌘C)	147
Paste (⌘V)	147
Clear	147
Test run (⌘R)	147
Turn Grid On/Off (⌘G)	148
Set Grid...	148
Show/Hide Coordinates	149
Source ID chars...	149
Switch to 320/640 mode	149
Preferences...	149
Show/Hide Clipboard	151
The Structures Menu	151
Frame...	152
Title...	152
Colors...	152
The Controls Menu	153
Radio button... (⌘1)	153
Push Button... (⌘2)	156
Check Box... (⌘3)	159
Edit Line... (⌘4)	161
Static text... (⌘5)	162
Long text... (⌘6)	163
StatText... (⌘7)	163
PopUp... (⌘8)	164
List... (⌘9)	166
Picture... (⌘0)	167
Text Edit... (⌘=)	171
Scroll... (⌘I)	173
Grow box... (⌘J)	175
Appendix A: Complete MiniWord Program	177
MAIN.PAS	177
CMDS1.PAS	190

Design Master	
CMDS2.PAS	200
ERROR.PAS	217
GLOBALS.PAS	221
Appendix B: The Boot Disk and How To Use It	223
Introduction To The Finder	223
Opening Disks, Folders, And Files	224
Initializing Disks	224
Copying Disks, Folders, and Files	225
Creating Folders	226
Renaming Disks, Folders, and Files	226
Deleting Folders and Files	226
Index	227

Chapter 1

Introduction

Welcome to Design Master! This sophisticated programming tool is as easy and fun to use as a paint program, allowing you to "draw" windows, menu bars, and dialog boxes. You can add radio buttons, simple buttons, check boxes, edit-line items, and text to your dialogs and windows, and paint them any color you like. You can also add super controls, including pop-up menus, icons, pictures, and lists, to your windows. Design Master includes an icon editor that lets you create and save icons, as well as a picture cropper to open picture files and paste part or all of a picture into your window. Creating menus is fast, simple, and painless.

Built into Design Master is a "test run" facility that lets you see exactly how your creation will look and operate in a program. You can save your work at any time, in a variety of formats, and add, remove, or change controls and menu items, or simply delete the current structure and start over.

Design Master can generate source code in the C, Pascal, ORCA/APW assembly, and Merlin 816 assembly languages. It can also generate source code for Apple's new Rez® compiler, and create resource forks. Finally, you can save your masterpieces to disk in binary files that use Design Master's internal format, and so can be loaded into Design Master and further changed and tested.

Who Should Use Design Master

Design Master is a programming tool; its purpose is to help programmers write desktop applications quickly. To make effective use of Design Master, you should first be an experienced programmer in any of the languages Pascal, C, or assembly language. You need not be an advanced programmer, but you should be familiar enough with one of these languages that you can write simple text programs containing more than one subroutine, and that you are familiar with all of the language's basic data types.

Second, you should own the Apple IIGS reference manuals needed to program the tools and the operating system. The toolbox is a large, powerful set of routines. The original toolbox reference manual set comprised over 1,400 pages in two volumes. The toolbox documentation has grown since then; some of the newer tools and changes to the old tools are described in loose-leaf form, and are available from Apple Computer, Inc.® It is not our intent to reproduce this voluminous amount of material in this manual. The reading list at the end of this chapter gives the manuals you should have at hand and how they can be purchased.

Finally, it is expected that before you begin Chapter 2, you will have read the *Apple IIGS Owner's Guide* manual that came with your computer.

Design Master

What You Should Have Received

The Design Master package consists of one 3.5-inch disk labeled Boot.Disk; one 3.5-inch disk labeled Design.Master; a product owner's registration card; and this manual. Be sure to return the registration card; it entitles you to receive software update information, as well as special product offers from the Byte Works. The registration card is also your proof of purchase. We cannot replace defective product components unless we receive it.

The Boot Disk volume contains version 5.0.2 or later of the GS/OS operating system and toolbox. Consult the Release.Notes file on the Design Master disk for the latest version numbers of the package's software. Note that Design Master requires version 5.0 or greater of GS/OS in order to run; the boot disk, therefore, has the version of the operating system and tools that it needs.

The Design Master disk contains the Design Master program and sample programs. Many of the files will be used in the tutorials in the next few chapters.

System Requirements

In order to use Design Master, you must have the minimum hardware and software specified in the next two sections. Recommended hardware is also mentioned.

Required Hardware

- An Apple IIGS computer, or an Apple //e computer with an installed Apple IIGS upgrade. Your machine must have a ROM 01 upgrade installed, since this is required by GS/OS 5.0.
- A minimum of 768K bytes of RAM. The 768KB is the total amount of memory that you must have installed in your machine. For example, if your computer is of the type that has 256KB on the mother board, then you'll need an additional 512KB of installed RAM.
- At least one 3.5 inch disk drive.

The following hardware is highly recommended, especially if you intend to do multiple-language development or to develop large programs:

- One megabyte of RAM.
- A second disk drive. While a hard disk is ideal for serious programming, a 5.25-inch disk to hold files while running Design Master on your 3.5-inch disk drive will work, too.
- A printer.

Required Software

All required software is found on the two disks that comprise the Design Master package. Software requirements can be found in the section "What You Should Have Received," appearing above in this chapter.

About This Manual

The manual provides three chapters of tutorial information followed by one chapter containing the complete Design Master reference. In the tutorial, a simple text editor is created using Design Master. Chapter 2 walks through the creation of the desktop objects that will be manipulated in the editor, providing a general overview of Design Master. Chapter 3 builds the complete program, discussing the source code in detail. It contains a thorough introduction to toolbox programming, including using the latest 5.0 calls and structures, such as super controls. Chapter 4 alters the source code in order to use resource forks. A broad introduction to the Resource Manager, resource forks, and the use of resources is given.

Visual Cues

We have used a variety of visual cues in the Design Master manual to help you distinguish between information, text displayed by the computer, and anything you have to type. When you need to enter characters, the type face

`looks like this.`

Design Master's menu names, commands, and buttons and options within its dialog boxes and alerts

`look like this.`

Additional Reading And Reference

The reading list below is split into two parts. The first list contains manuals that can be purchased from both A.P.D.A. (Apple Programmers' and Developers' Association), Apple Computer's retail outlet; and your local bookstore, from the Addison-Wesley publishing house. The second list covers documentation available only from A.P.D.A. For your convenience, A.P.D.A.'s address is:

A.P.D.A.
Apple Computer, Inc.
20525 Mariani Ave.
MS: 33G
Cupertino, CA 95014

The following books are in the Apple IIGS Technical Manual Suite; they are available through A.P.D.A. and your local bookstore:

Design Master

Apple IIGS Toolbox Reference Volume I

Apple IIGS Toolbox Reference Volume II

These volumes provide essential information on how tools work; they are absolutely essential in mastering the source code presented in this manual, and for optimum use of Design Master.

Human Interface Guidelines: The Apple Desktop Interface

This book gives a complete reference for writing standard desktop applications on the Apple IIGS and Macintosh computers. It also contains important information for the users of desktop programs.

Apple IIGS Hardware Reference

Apple IIGS Firmware Reference

These manuals provide information on how the Apple IIGS works.

Programmer's Introduction to the Apple IIGS

Provides programming concepts for the Apple IIGS.

Technical Introduction to the Apple IIGS

A good basic reference source for the Apple IIGS.

Apple Numerics Manual, Second Edition

This book describes SANE, Apple's floating-point tools. You will need it if you will be calling SANE directly. In compiled languages, all of the details of dealing with SANE are handled for you by the compiler.

Apple IIGS ProDOS 8 Reference

Apple IIGS ProDOS 16 Reference

These manuals document the operating systems which preceded GS/OS.

The following are available only from A.P.D.A:

Apple IIGS Toolbox Reference Update

Documents new tools and changes to the tools with the release of version 5.0 of GS/OS. This document is absolutely essential in mastering the source code presented in this manual, and for optimum use of Design Master.

GS/OS Reference, Volume 1, Beta Draft

Documents new tools and changes to the tools with the release of version 5.0 of GS/OS. This document is absolutely essential in mastering the source code presented in this manual, and for understanding GS/OS, the latest operating system on the Apple IIGS computer.

GS/OS Reference, Volume 2, Beta Draft

Chapter 1: Introduction

For those who write code on the bare metal. Covers writing device drivers, interrupt handlers, signal processors.

Chapter 2

"Drawing" A Program

Getting Started

In this chapter, we'll cover the basics of using Design Master. In the process, we'll create a simple text editor from the source code provided on the Design Master disk and from the source code generated by Design Master.

Back-up Design Master First!

Whenever you purchase new software, one of the first things you should do is to make back-up copies of the original disks. (Of course, sometimes this isn't possible when the disks are copy-protected.) Design Master is not copy-protected, so let's make back-ups of the disks. You can use whatever disk-copying utility you prefer to create the back-ups. You can also use the boot disk that came with Design Master to perform the back-up. Refer to Appendix B for instructions on using the boot disk.

Starting Design Master

The Design Master package contains two disks, one labeled Boot.Disk and the other named Design.Master. Boot.Disk is used to boot your computer; it boots into the Finder, Apple's program launcher. Since the Finder is a desktop program, you can use the mouse to do just about everything: pull down menus, select menu items, open files and folders. You can use the Finder to initialize, rename, and copy disks; create folders; delete files; and copy files and folders. In the instructions that follow, it is taken for granted that you know how to initialize disks, create folders, and copy files. If you do not know how to perform these tasks, refer to Appendix B for further information.

The Design Master disk contains the DesignMaster program, any data files the program needs, and a folder labeled Samples. The examples presented in this and the next two chapters will use source code provided in the Samples folder.

Start the Design Master program by booting your computer. Place the Design Master disk on-line, then double-click the Design Master disk to open it. (Don't start Design Master yet – we need to set up our program. Of course, if you're anxious to see what you just bought, double-click the Design Master icon to launch it. Select the Quit command, from the File menu, to return to the Finder from Design Master.

Preparing to Create the Text Editor

Place a formatted, empty disk on-line, and name it MiniWord. Create these folders on the MiniWord disk: Source and Binary.

Place the Design Master disk on-line and open the Samples folder. The Samples folder contains the folders ASM.Samples, Pascal.Samples, and C.Samples. The examples in the manual use the Pascal programming language, so we'll now open the Pascal.Samples folder. You should open the folder corresponding to your preferred language, either ASM.Samples for assembly-language, or C.Samples for the C programming language. Now open the Source folder and copy all of the files in it to the MiniWord disk's Source folder.

It's finally time to play with Design Master. We won't need the source code for a while, so this disk can be taken off-line if you have only one 3.5-inch disk drive. Close all open folders, place the Design Master disk on-line, then launch Design Master by double-clicking on its icon.

Design Masters Menu Bar;

Design Master's menu bar contains the Apple, File, Edit, Structures, and Controls menus. The Apple menu has an About item and a Help! command, and provides access to any new desk accessories (NDAs) that are installed in the SYSTEM/Desk.Accs folder of the boot disk. The About item shows the version number of Design Master. Our customer service department will need to know this version number to help you with problems or questions about Design Master.

Pull down the Apple menu again and notice the Help! item. Next to Help! are the characters ⌘H. The ⌘H is called a keyboard equivalent or a "hot key." You can issue the Help! command by selecting it from the Apple menu, or by holding down the ⌘ key while pressing the H key. (You can also use the h key.)

Help! brings up a window like this one:



The window contains brief descriptions of several of Design Master's commands. In cases where you need more detailed information, you should refer to Chapter 5 of this manual. You can

Chapter 2: "Drawing" A Program

close the window by clicking in its close box, located in the upper left corner of the window, or by selecting the `Close` command from the `File` menu.

Now pull down the `File` menu. You will see that several commands end with three dots. The dots indicate that selecting the command will bring up a window where you make choices and enter information. Commands appearing without the dots are those that require no further information from the user.

The first three commands in the `File` menu, `New Window`, `New Dialog`, and `New MenuBar`, are used to create structures. Only one structure may be created at a time.

The `Close` command is used to close the currently active text window or structure, while the `Quit` command exits the Design Master program. The other commands in this menu provide access to disk files.

The next menu in the menu bar, `Edit`, allows you to undo operations; cut, copy, paste, and clear controls; test the current structure; set up a grid system on the desktop; toggle the screen between 640 and 320 modes; and customize the program.

The `Structures` menu lets you change different aspects of the window you are currently creating.

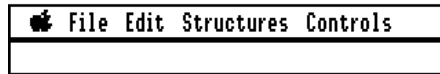
The final menu, `Controls`, provides commands to create dialog items and window controls. Notice that this menu is divided in half. Some controls can only be used in a dialog, while others can only be used in a window. The ones in the top half of the menu will work in dialogs. The controls in the bottom half of the menu will only work in a window. In addition, some of the controls in the top half of the menu can be used in a window. Design Master will highlight the controls you can use at any particular time.

Creating A Menu Bar

With the preliminaries out of the way, let's use Design Master to create a simple menu bar. If you haven't done so already, boot your computer and then launch Design Master.

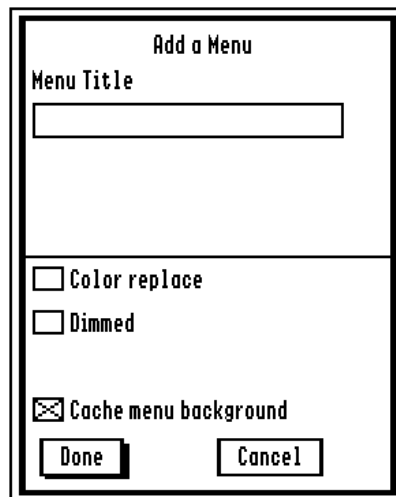
Now pull down the `File` menu and select `New MenuBar`. The desktop will fill with a work area for creating a menu bar:

Design Master



The bar below Design Master's menu bar displays our menu bar as it is created. The menus we want to build include the Apple menu, a File menu, an Edit menu, and a Search menu.

To create the Apple menu, click on the Add menu button. A dialog like this appears:



At the top of the dialog is the message Menu Title, with a rectangular box beneath it. Inside the box is a flashing vertical bar. The box is called an edit-line item, so named because editable text can be typed into the box. The flashing bar is called the insertion point; it marks where the next character will appear. An edit-line item supports a number of editing capabilities, such as inserting text, deleting text, moving through the text with the arrow keys, and cut, copy,

Chapter 2: "Drawing" A Program

and paste. Its editing facilities are described on pages 10-1 and 10-2, in the *Apple IIGS Toolbox Reference: Volume 1* manual.

The Apple menu is created by entering the single character @ (the at sign) in the `Menu Title` box. Use the keyboard to type an @ in the box, being sure to not place any spaces either before or after the @.

Beneath the title box are three square boxes with text to the right of each. These items are called check boxes. A check box is selected by clicking in the box with the mouse. When you click on the name of a menu to pull the menu down, the Menu Manager highlights the menu name. Normally, it does this by reversing the bits in the menu name, making all white bits black and all black bits white. The `Color replace` option tells the Menu Manager to use a form of highlighting called color replace highlighting, which only changes the white areas of the menu bar. You should select this option for the Apple menu; it will make the Apple menu look normal when it is highlighted. Selecting the `Dimmed` check box will create a menu whose initial state is unselectable, and whose appearance in the menu bar will be dimmed to show that it is currently not available. The `Cache menu background` check box, which is selected, tells the Menu Manager to remember what the menu looks like, so it can draw it faster if the menu is pulled down again. The Menu Manager is smart enough to know when you have changed something; in that case, it redraws the menu the slow way. There is really no penalty for caching the menu, so you should always leave this option on.

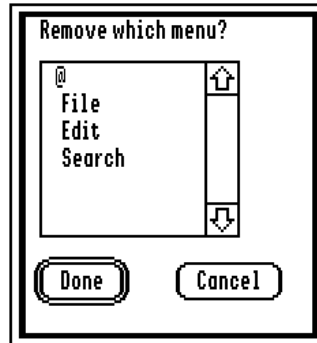
Click the `Done` button (or simply press the `RETURN` key since `Done` is the default button) to exit the `Add menu` dialog and add the Apple menu to our menu bar.

Now let's add the File menu to our menu bar. Again click the `Add menu` button. In order to separate menu names on the menu bar, we need to add spaces to the name. In the `Menu Title` box, enter **File** (type a space before and after the name). If you prefer, you can surround the name with more than single spaces. The important point is that you be consistent, because when the menu is selected, the spaces are selected, too. Click the `Done` button to add the File menu to our menu bar. Notice that we didn't select the `Color replace` check box this time.

Repeat the process two more times to add the Edit menu and the Search menu to our menu bar.

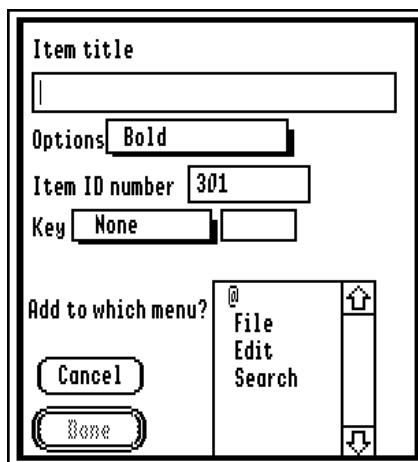
If you don't like the way a menu looks, or you'd like to delete a menu, you can select the `Remove menu` button. It brings up a dialog similar to this:

Design Master



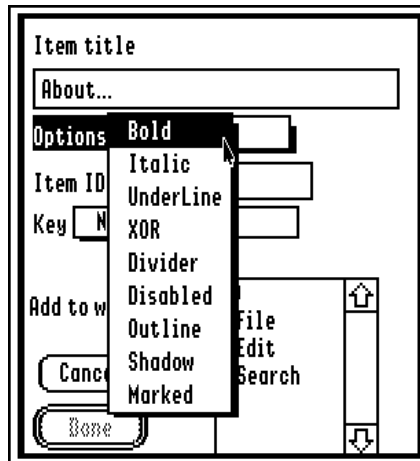
In the center of the dialog is a list of the menus you've created so far. (The box is called a list control. It has a vertical scroll bar that allows you to scroll the list, in the same way that a window is scrolled.) Use the mouse to click on the menu you'd like to remove, and then click the **Done** button. The menu will disappear from the menu bar, and the menus will be adjusted along the bar so that a "hole" isn't left after removing the menu. If you decide not to remove a menu, just click the **Cancel** button.

The next step is to add menu items to our menus. First let's add an **About** item to our Apple menu. Select the **Add menu item** button. It brings up a dialog like this:



In the edit-line box labeled **Item title**, type **About....** Below the title box is an **Options** item, with a rectangle to the right labeled **Bold**. This is a pop-up menu, so called because pressing on its title causes the menu to "pop up." The rectangle next to the title is called the pop-up's display rectangle. You can select items in a pop-up menu in the same way as from a regular menu, by releasing the mouse button when the desired item is highlighted.

Press on **Options** with the mouse button to access its menu:



The items in this menu are used to further define the current menu item. You can change the item's style to have it drawn in a bold, italic, outlined, or shadowed type face; have it underlined; have a check mark placed next to it to show that its default value is selected; define a keyboard equivalent for it; use XOR highlighting; or disable the item. You can select as many of these options as you want. Notice that if you select the `Divider` item the menu item will be drawn with a line beneath it. A divider is used to divide the items into logical groups, based on their function. We'll use dividers in our File menu to separate the file commands (such as new, open, and save) from our print commands.

If you accidentally select an option that you don't want, you can deselect it by selecting it a second time. Selected items are indicated with a check mark that appears to the left of the item's name.

Following Design Master's convention for its `About` item, let's select `Bold` from the Options pop-up.

Beneath the Options pop-up is an edit-line box labeled `Item ID` number. This is the menu item number that we'll use in our program to check if the user has selected this item. Enter **256** in the box.

On the lower right side of the dialog is a list control labeled `Add to which menu?` Select the `@` menu from the list (this is our Apple menu), and then click the `Done` button. If you pull down our Apple menu, you'll see an **About** item.

Now let's add a `New` item to our File menu. Once again, select the `Add menu item` button. Enter **New** in the `Item title` box. Notice that Design Master has automatically incremented the ID number to 257. This is the ID we'll use in our program, so we leave the box as is.

In many desktop programs, the `New` item has a keyboard equivalent of `N`, so let's follow suit and do the same. Below the `Item ID` option is a pop-up menu labeled `Key`. Select the `Character` item from this menu. In the edit box next to the pop-up, type the character `N`. (The

Design Master

ASCII code item in the pop-up lets you enter a number corresponding to the ASCII value of the key equivalent.)

Now select File from the menu list, and then click on Done.

If you now pull down our File menu, you'll see the New item with the characters ⌘N next to the right edge of the menu.

Repeat the process to add these items to our File menu:

<u>Item Name</u>	<u>Key Equiv.</u>	<u>ID Number</u>
Open...	⌘O	258
Close	⌘W	255
Save	⌘S	259

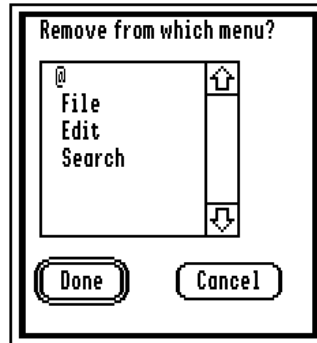
Notice that the ellipses (the ... marks) are a part of the name for the Open command. Note, too, that the Close item has an ID number that is out of sequence. This is because we can use reserved item IDs for certain commands in a desktop program, and have the tools take special actions for the commands. In our program, we'll use the special menu items of Close, Cut, Copy, and Paste. Doing so will free us from a lot of drudgery, as we'll see in the next chapter.

We now want to add a dividing line to our File menu, to separate the file operations from the print operations. To create the divider, select Add menu item. In the Add menu item dialog, type **Save as...** in the Item title box, and enter **260** in the Item ID box. Now select Divider from the Options pop-up. Select File from the menu list, and then click Done to add the Save as item, followed by a divider, to our File menu. You won't be able to see the divider until you add the rest of the menu items, but it is there.

Repeat the processes outlined above to add the following items to our menu bar:

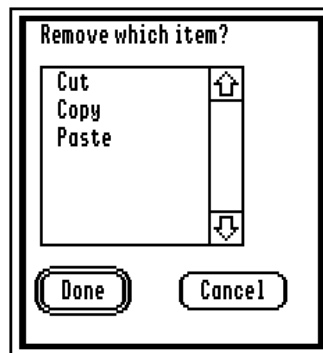
<u>Menu</u>	<u>Item Name</u>	<u>Key Equiv.</u>	<u>ID Number</u>	<u>Notes</u>
File	Page setup...		261	
File	Print...	⌘P	262	divider
File	Quit	⌘Q	263	
Edit	Cut	⌘X	251	
Edit	Copy	⌘C	252	
Edit	Paste	⌘V	253	
Search	Find...	⌘F	264	

If you make a mistake when creating a menu item, you can delete the item from its menu with the Remove menu item button. It brings up a dialog like this:



In the center of the dialog is a list of menus that have been created thus far. Use the mouse to select the menu from which an item is to be removed, then click the **Done** button. (Menus for which items have not been defined will be dimmed and unselectable.)

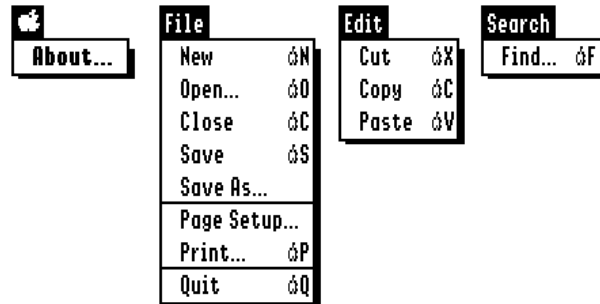
After selecting **Done**, a second dialog appears:



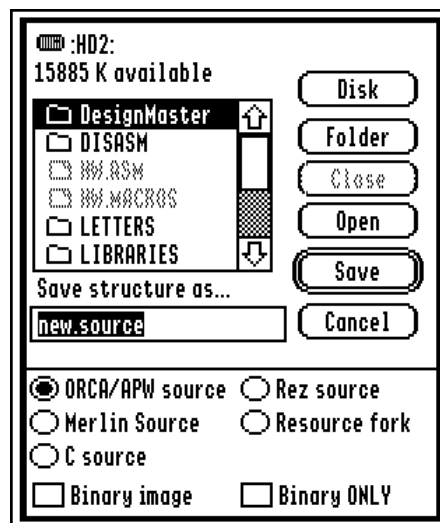
Use the mouse button to select the item that is to be removed, then click **Done**. The menu creation area will reappear on the desktop. If you pull down the menu from which an item was just deleted, you'll see that the item is now gone, without leaving any "holes" in the menu.

The menus in our final menu bar look like this:

Design Master



At this point, let's save our work so that we can use it in the next chapter, when we create a simple text editor. Select the `Save As` command from the `File` menu. It brings up a dialog like this:



At the top of the dialog is an icon depicting the current prefix, followed by the name of the prefix. For example, in the dialog above, there is a disk icon showing that the current prefix is the root volume (i.e. we haven't opened a folder), and ending with the name of the disk, which is `HD2`. Below the current prefix display is a box containing the files and folders within the current prefix. Folders are always selectable since they can be opened. Files are dimmed and unselectable, to prevent you from accidentally over-writing their contents.

The `Disk` button is used to move through the disks currently on-line. You can also use the `TAB` key in lieu of the `Disk` button.

Chapter 2: "Drawing" A Program

The `Folder` button is used to create a new folder. You first enter the name of the new folder in the edit-line box labeled `Save structure as...`, and then click the `Folder` button. The new folder will then appear in the file list, and it will be selected.

The `Close` button is used to close folders. This button will be dimmed and unselectable if the current prefix is not within a folder. You can also close the current folder by clicking on the prefix display above the file list.

The `Open` button is used to open folders. You can open a folder by clicking on its name in the file list, and then clicking on the `Open` button, or by double-clicking its name.

The `Save` button causes the current structure to be saved to disk, in the format selected.

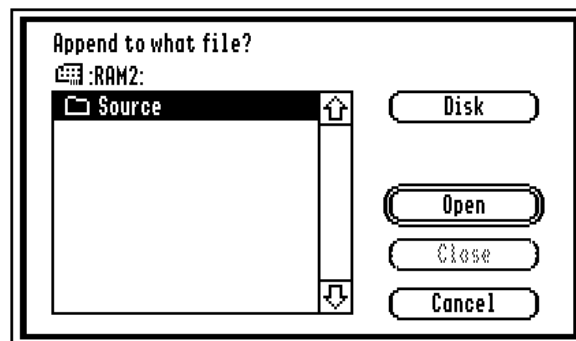
The `Cancel` button exits the dialog without saving a file to disk. Note: the current prefix will be set to the disk/folder selected. The `Cancel` button will not cause the current prefix to be reset to its original setting.

At the bottom of the dialog are several radio buttons that correspond to the file formats that Design Master can use to save our menu bar. We want to append the source code definition of our menu bar to the end of one of the source files that we placed on our data disk. We'll use Design Master's `Append` command to do this, rather than `Save as`, so we'll just ignore the radio buttons for now.

There are two check boxes at the very bottom of the dialog labeled `Binary image` and `Binary ONLY`. The binary format is Design Master's internal format; you can load a file in this format directly into Design Master. Select the `Binary ONLY` option.

Place your MiniWord disk on-line, and use the `Disk` and `Open` buttons to move to its `Binary` folder. Now enter the filename **menu** in the edit-line box below the file list, then click the `Save` button. A second save dialog appears, which is nearly identical with the first one. This dialog is used to save Design Master binary files. Notice that Design Master has appended the suffix `.bin` to the filename, creating the filename `menu.bin`. Click `Save` to complete the save operation.

We're now ready to append the source code definition of our menu bar to our program's source code. Pull down the `File` menu and select the `Append` command. Its dialog is very similar to the `Save as` dialog; its buttons operate exactly as they do in the `Save as` dialog:



Design Master

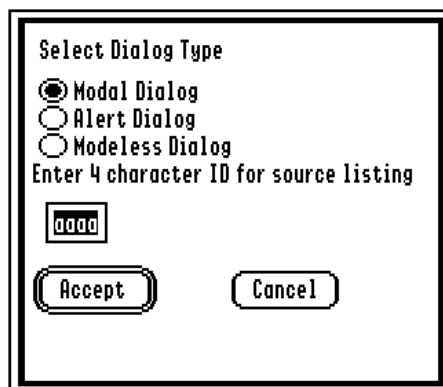
Use the `Close` button to close the Binary folder, then open the Source folder. Select the file named `main.pas` from the file list, then click the `Open` button. (For assembly language, select the file named `miniword2.asm`; for the C language, select the file named `main.cc`.)

After saving the file, select `Close` from the `File` menu. The menu bar we've created will disappear, and the desktop will return to its original state.

If you'd like to see the source file that Design Master has created, you can select the `View file` command from the `File` menu. The file will be displayed in a window that you can move, size, and scroll, but you will not be able to edit its contents. We'll put off discussion of the source code until Chapter 3. You can close the source code window by clicking in its close box, or with the `Close` command.

Creating A Dialog Box

Let's walk through the creation of a simple dialog box. We'll need one for our About item in the next chapter, so that's the item we'll design it around. Select the `New Dialog` command from the `File` menu. It brings up a dialog like this:



At the top of the dialog are three radio buttons that allow you to select the type of dialog box to be created. The `Modal Dialog` button is the default button, so it's already selected for us. Modal dialogs are used to present and obtain information. They stay on the screen, preventing you from doing anything else, until you are finished with the dialog. The dialog you are looking at is an example of a modal dialog. Alerts are used to get the user's attention; the only input they typically require is a simple "Yes" or "No." Alerts have a special icon in the upper left corner that depicts the type of alert, either standard, caution, note, or stop. Modeless dialogs present information, like a modal dialog, but also let you do other things in other windows while they are on the screen.

Beneath the radio buttons is an edit-line box labeled `Enter 4 character ID for source listing`. Since a desktop program can (and usually does) have many different dialogs,

Chapter 2: "Drawing" A Program

we need some way to easily distinguish one from another. Type **Abt** (for About) in the box. Now click **Accept** to complete the definition.

Design Master will draw an empty dialog box on the desktop. You can click on the dialog with the mouse, and, keeping the mouse button depressed, drag the dialog around on the desktop. In the lower right corner of the dialog is a red box, used to size the dialog. You can grow or shrink the dialog by clicking in the box, and, keeping the mouse button depressed, drag the box. The final size and position of the dialog will be recorded in the definition Design Master creates, so move the dialog now to the center of the desktop, and then size it so that it looks similar to this:



The first thing we'll want to put in our About dialog is a title. Pull down the **Controls** menu, and note that we can select any of the items in the top half of the menu. Select **Static text** from this menu. It brings up a simple dialog. Let's put a simple message in our About dialog regarding MiniWord. Type the following text into the large edit box, and then click **Done**. Where you see <your name>, enter your own name:

**A simple text editor written by
<your name> and Design Master.**

The static text item will appear in our dialog box, outlined in red. You can move the item within the dialog box, or use the box in its lower right corner to size it. The outlining indicates that the item is currently selected. (Items can also be selected by clicking on them.) Use the same idea to add the name of the program, MiniWord, to your dialog. When you are finished, the dialog will look something like this:



The **Edit** menu provides a number of operations that can be performed on a selected item. Issue the **Copy** command from this menu. This command causes a duplicate of the item to be

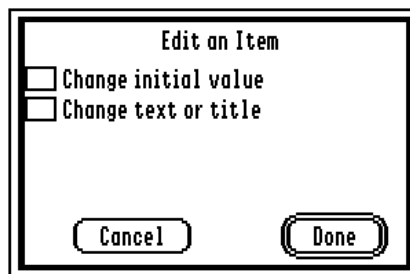
Design Master

placed in a special file called the Clipboard. Select the `Show Clipboard` command from the `Edit` menu. It brings up a window that shows the current item it contains. If you issue the `Paste` command, a second copy of the item will be pasted into our About dialog.

You can remove a selected item by selecting the `Clear` command from the `Edit` menu. You can also remove it with the `Cut` command; this command moves the item to the Clipboard.

You can close the Clipboard window by clicking in its close box, or by issuing the `Close` command from the `File` menu.

Another way to change an item, but without removing it, is to double-click on the item. Doing so brings up this dialog:



You can select one or both of the check boxes in this dialog. Clicking `Done` will continue with the editing process, while clicking `Cancel` will exit the dialog, leaving the selected item in its original form. The `Change initial value` option allows you to assign a value to the item that differs from the one derived by Design Master. See Chapter 6 of the *Apple IIGS Toolbox Reference: Volume 1* manual for details about item values. The `Change text or title` option, as implied, lets you relabel buttons or redefine static and edit text items.

Getting back to business, let's add one more item to our About dialog, a button that says "OK." Select `Push button` from the `Controls` menu. It brings up this dialog:

Enter additional parameters for Push Button.

Default text for item:

☒ Simple Round Button
☐ Simple Square Button
☐ Bold Round Button
☐ Bold Square Button

Item ID number

☐ Inactive ☐ Custom Colors

Type **OK** in the edit-line box labeled `Default text for item:` We want the OK button to be the default button (of course, it's the *only* button!), so let's select the `Bold Round Button` option. The default button in a dialog has an item ID of 1, so change the 4 in the `Item ID number` box to 1. We want the button to be active, so we skip the `Inactive` check box. Let's do something exciting like coloring our button, so select the `Custom Colors` check box. Click `Done` when you're finished making your choices.

A new dialog appears, used to color the push button we're creating:

Custom Colors

☒ Outline color
☐ Interior color when not hilited
☐ Interior color when hilited
☐ Text background when not hilited
☐ Text background when hilited
☐ Text foreground when not hilited
☐ Text foreground when hilited

Design Master

The dialog consists of a series of radio buttons, each with a box to the left of the button. At the bottom of the dialog is an array of colors, called the palette. The `OK` button is used to signal that coloring is complete. The `Cancel` button is used to exit the `Custom Colors` dialog without coloring the push button. The two buttons labeled `Title` show how the button will look as you paint it, in both its active and inactive states. To color an area in the button, click on the radio button for the desired part, then click on the color you want. The square next to the selected radio button will fill with the specified color, and the `Title` buttons will change to match your current color choices. We won't presume to tell you how to color your button.

After the button is created, it will appear in the upper left corner of our `About` dialog. Use the mouse to drag it to the bottom of the dialog and center it.

To see how the button will look when we click on it with the mouse, select the `Test run` command from the `Edit` menu. Try holding the mouse button down on the button, and then letting up on it. When you're satisfied that you've seen enough, select `End test run` from the `Edit` menu.

As with the static text item that we created, you can remove the push button by first selecting it and then using the `Clear` or `Cut` commands. You can change its text or value by double-clicking on it to bring up the `Edit an Item` dialog. Finally, you can place a copy of it in the Clipboard file with the `Copy` or `Cut` commands, and then use the `Paste` command to duplicate the item in this dialog or in another dialog that may be created after this one.

Let's save our work, now that our `About` dialog is finished. Select `Save as` from the `File` menu. It brings up the standard save file dialog that we looked at when saving our menu bar to disk. Be sure to put your MiniWord disk on-line, then use the buttons to reach the disk and `Binary` folder on the disk. This time enter **about** in the filename edit-line box. Be sure to check the `Binary ONLY` box before clicking the `Save` button to save the dialog definition in binary format.

Note that the save dialog for a dialog box does not have radio buttons for resource file formats. This is because the Resource Manager does not support dialogs.

Now select the `Append` command from the `File` menu. Use the buttons to move to the `Source` folder on your MiniWord disk. This time select the file named `cmds1.pas` from the file list, then click the `Open` button. (For assembly language, select the file named `miniword2.asm`; for the C language, select the file named `cmds1.cc`.)

When you're finished creating and saving our `About` dialog, select `Close` from the `File` menu to remove the dialog from the desktop.

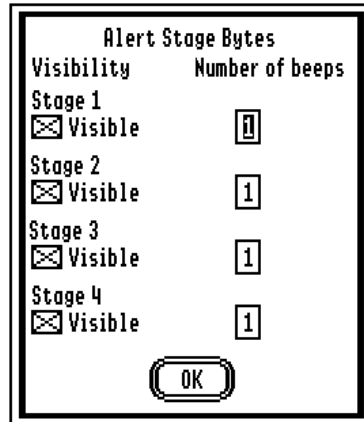
As with the menu bar we created, you can use the `View file` command to look at the source code that Design Master has created.

Creating An Alert Box

Now let's create an alert box for error messages. Pull down the `File` menu and select the `New Dialog` command. This time, though, click on the `Alert Dialog` radio button, and then enter **Err** in the 4-character `ID` edit line box. Click `Accept` when you've completed these steps.

Chapter 2: "Drawing" A Program

A second dialog box appears, which allows you to define what should happen at each stage of an alert.



The dialog box is titled "Alert Stage Bytes". It contains a table with two columns: "Visibility" and "Number of beeps". There are four rows, labeled "Stage 1" through "Stage 4". Each row has a checkbox in the "Visibility" column and a text input field in the "Number of beeps" column. All checkboxes are checked, and all input fields contain the number "1". An "OK" button is at the bottom.

Visibility	Number of beeps
Stage 1 <input checked="" type="checkbox"/> Visible	<input type="text" value="1"/>
Stage 2 <input checked="" type="checkbox"/> Visible	<input type="text" value="1"/>
Stage 3 <input checked="" type="checkbox"/> Visible	<input type="text" value="1"/>
Stage 4 <input checked="" type="checkbox"/> Visible	<input type="text" value="1"/>

OK

Alerts have several stages, each corresponding to how many times the alert has been used. This lets you change the alert if it has been used several times. You might use this ability so your program doesn't annoy the user by repeatedly reminding her about something. You could also use this ability to show the program's annoyance at a user who repeatedly tries to do something that is not possible. This dialog lets you change the behavior of the alert. Stage 1 corresponds to the first time the dialog is shown, stage 2 to the second time, stage 3 to the third time, while stage 4 is used any time after the first three times. At each stage, you can select whether the dialog is visible, as well as how many times the system beeps. In most programs, the default is a good choice.

When you've finished making your choices in this dialog, click the Done button to complete the alert box definition. An alert box like the one below will appear in the upper left corner of the desktop.



Make this alert about a quarter of the size of the screen, and center it.

In the upper left corner of the alert is a red box containing the message Alert Icon Goes Here. The actual icon that will appear in the box is determined by the alert function called in a program. The Dialog Manager defines four types of alerts, and fills in the icon box for three of the types. The alerts containing icons are stop, note, and caution. The alert box without an icon is called a standard alert. Examples of the different types of alerts are shown in the *Apple IIGS Toolbox Reference* manual.

Design Master

There are three controls we need to add to our error alert: two static text items, and an OK button. Pull down the `Controls` menu, and select `Static text`. Enter this static text message:

Here is a message that is fifty characters long!!!

in the edit-line box, then click on the `Done` button. We will use a subroutine in our program to change the text as needed; for now, we want to have a 50-character control defined in our alert. The static text item will appear in the upper left corner of the alert box. Use the mouse to drag it beneath the icon box. As with the other controls and structures created by Design Master, there is a grow box in the lower right corner of the static text control. Use the mouse to grow the message box so that it covers most of the lower portion of the alert, but leave room for another message and an OK button at the bottom.

Now select the `Static text` item from the `Controls` menu again. This time enter **\$0000** in the edit-line box, then click `Done`. We'll use this message to report the error number, as a hexadecimal value, returned by the tools to our program.

Select `Push button` from the `Controls` menu, and create an OK button, just like the one you created for your about dialog. Use the mouse to drag the button to the bottom center of the alert. Your finished alert should look something like this one.



Let's save our work, now that our error alert box is finished. Place your MiniWord disk online, then select the `Save as` command from the `File` menu. Move to the `Binary` folder on your data disk, check the `Binary ONLY` option, enter the filename **error** in the edit-line box, and click the `Save` button.

To save the source code definition, select the `Append` command from the `File` menu. Move to the `Source` folder on your data disk, select the file named `error.pas`, then click the `Append` button. (For assembly language, select the file named `miniword2.asm`; for the C language, select the file `error.cc`.)

As with the menu bar we created, you can use the `View file` command to look at the source code that Design Master has created.

Creating A Second Alert Box

Let's create one more alert box, this time to ask if the user wants to save a document to disk before closing its window. Pull down the `File` menu and select the `New Dialog` command.

Chapter 2: "Drawing" A Program

Click on the `Alert Dialog` radio button, and then enter **Save** in the `4-character ID` edit-line box. Click `Accept` when you've completed these steps. Set your alert stages the way you want them. Select `Static text` from the `Controls` menu, and enter this message into the edit-line box: **Save changes before closing?** Click `Done` when you've entered the message, and then center the message below the icon box in the alert. Be sure to leave room for two push buttons at the bottom of the alert.

Our save alert will have two push buttons, one labeled `OK` and the other labeled `Cancel`. Select `Push button` from the `Controls` menu. Enter **OK** in the `Default text` box, then select one of the `Bold button` options since this will be our default button. Change the `Item ID` to 1, and select `Custom Colors` if you want to color the button.

Select `Push button` from the `Controls` menu again. Enter **Cancel** in the `Default text` box, then select one of the non-bold button options since `Cancel` is not our default button. (If you used a square button for `OK`, you should use a square button for `Cancel`.) Change the `Item ID` to 2, and select `Custom Colors` if you want to color the button.

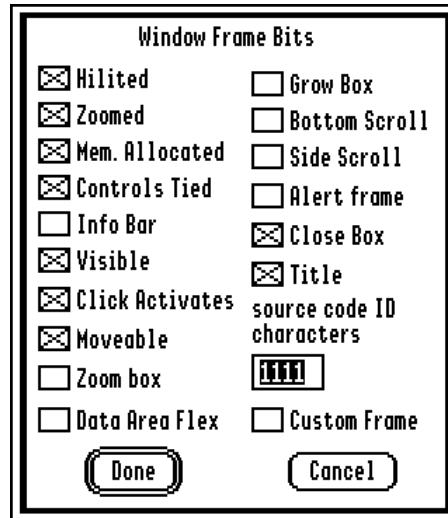
Once again, we'll want to save our work. Use the `Save as` command in the `File` menu to create a binary file named **save** in the `Binary` folder of your `MiniWord` disk. Use the `Append` command to append the source code to the end of the file named `cmds1.pas`, found in the `Source` folder of your data disk. (For assembly language, append the source code to the file `miniword2.asm`; for the `C` language, append the source code to the file `cmds1.cc`.)

Creating A Window

In our last example in this chapter, we'll create a "generic" window that we'll use in chapter 3 to contain our text editor's documents.

Select `New Window` from the `File` menu. It brings up this dialog:

Design Master



Almost all of the check boxes, with the single exception of `Custom Frame`, correspond to bits that are either set or cleared in the `wFrameBits` field of the parameter list record that is passed to the Window Manager's `NewWindow` call, as described in the *Apple IIGS Toolbox Reference: Volume 2* manual. We won't discuss each check box in detail here; we'll just briefly mention why we'll select the options that we do.

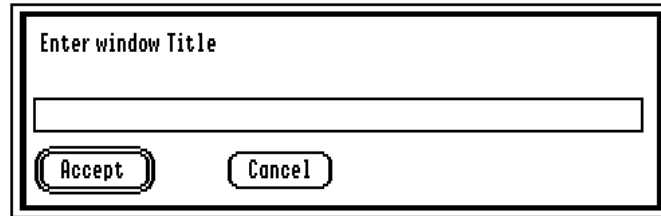
Make sure the following check boxes are selected: `Hilited`, `Zoomed`, `Mem. Allocated`, `Controls Tied`, `Visible`, `Click Activates`, `Movable`, `Zoom box`, `Close Box`, and `Title`. `Hilited` indicates that the frame is to appear highlighted. `Zoomed` means that the window will be drawn in its zoomed state. `Controls tied` means that the state of the window's controls are tied to the state of the window. `Visible` indicates that the window will be visible when it's drawn. `Click Activates` causes the window to become the active window when the mouse is clicked in its content region. `Movable` indicates that the window can be dragged by its title bar. `Zoom Box` and `Close box` mean that the window's frame will contain these boxes. `Title` indicates that the window will have a title bar. We don't want to add scroll bars or a grow box to the window because we'll add these controls to our text edit control, which we define next.

We'll only have one document window type, so we don't really need anything in the `source code ID characters` edit box. You can put `Wind` in, though, just in case you add another window to your program later.

We need to further define the window frame, so check the `Custom Frame` box.

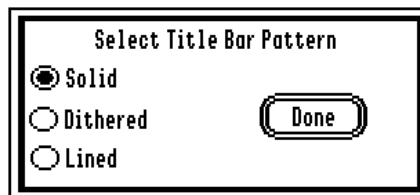
After all selections have been made, click the `Done` button.

A second dialog appears. It allows us to give the window a title.



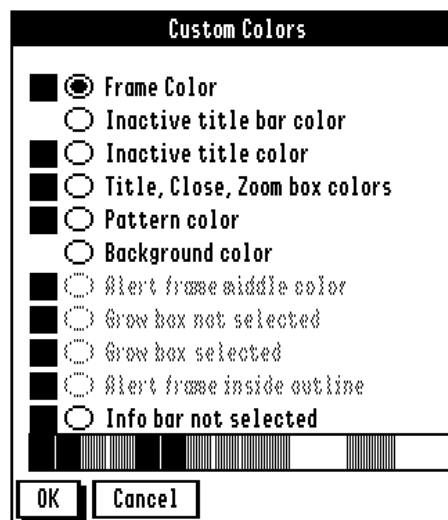
Since we're defining our "generic" window, simply click the `Accept` button.

Selecting the `Custom Frame` option in the first dialog box causes this third dialog to appear:



Standard document windows have horizontal lines in the title bar, so click the `Lined` radio button, then click `Done`.

One last dialog appears. It lets us color our window.



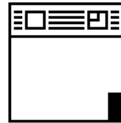
Design Master

This dialog functions just like the `Custom Colors` dialog described in the last section for the `Push button` command. As with that description, we won't presume to tell you how to color your window – we'll just discuss what area of the window is referred to by each of the radio buttons.

`Frame color` is the color of the outline of window. `Inactive title bar color` is the color that the title bar will have, not including the title, when the window is inactive. `Inactive title color` is the color of the window's title when the window is inactive. `Title, close, zoom box colors` is the color of these items in an active window. `Pattern color` is the color of the horizontal lines in the title bar. `Background color` is the color beneath the lines in the title bar. The `Alert frame` choices are used for windows defined with an alert-type frame, rather than a document frame, as in our window. `Grow box not selected` refers to the outline of the grow box, while `Grow box selected` refers to the inner rectangles of the control when the grow box is being dragged. `Info bar not selected` is the color of the info bar when it is not being selected; our window does not have an info bar.

After all selections have been made, click `Done` to complete the window definition, or click `Cancel` to abort the coloring operation.

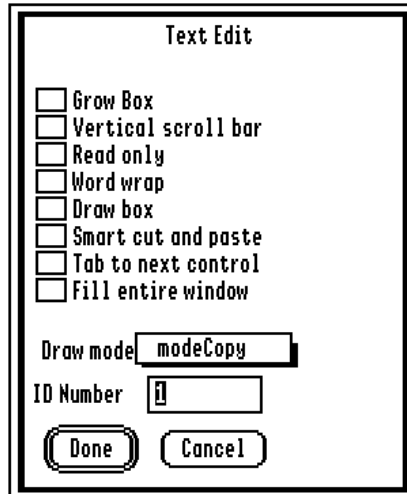
A new window will appear on the desktop, similar to this:



Use the grow box at the bottom of the window to size it, and drag on the title bar to move it, in order to set its initial zoomed size and position for the program we'll write in the next chapter. For our program, the window should nearly fill the desktop, and be centered.

If you decide that you want to change the window, you can use the commands in the `Structures` menu to alter the original definition without having to remove the window and start all over. The `Frame` command brings up a dialog similar to the initial `Window Frame Bits` dialog, except that the choices we've made for our current window will be checked, rather than the default values. The `Title` command brings up `New Window's title` dialog, and the `Colors` command brings up the `Custom Frame colors` dialog; both dialogs appear with our current choices shown.

Now pull down the `Controls` menu and select the `TextEdit` command in order to define a `Text Edit` control for our window. It brings up this dialog:



You can select any combination of the check box items for the control. Let's first discuss the items we'll select from this dialog, and why, and then discuss those items we won't select, and why. We want to select `Vertical scroll bar`, so that the Text Edit tool set will automatically handle scrolling the window as text is entered. We will select `Grow Box`, since we want to let the user change the size of the window. We will select `Word wrap`, since we want to allow wrapping of the text (soft carriage returns) at the end of lines during text input. We'll choose `Smart cut and paste` to enable support of "intelligent" cut and paste operations that take into account spacing around words. (Intelligent cut/paste is described in Chapter 25 of the *Toolbox Reference Manuals Update* document.) `Fill entire window` sets the size of the box to the size of the window, so we'll choose this to make our text look natural in its window, and not look like a special control.

We don't want to select `Read only`, since this would prohibit editing of the text within the control. We needn't select `Draw box`, which causes the text of the control to appear with a border marking the edges of its bounding rectangle, because we want the text to look natural within the window. We don't need `Tab to next control`, which allows the user to press the `TAB` key to move to the next control in the edit text window, since the other controls are defined on the frame, and we want the user to only use the mouse to operate them.

The `Draw mode` option provides a pop-up menu that allows you to specify the mode in which Quick Draw II will paint the text into the control's window. `modeCopy` copies the pixels directly into the window, ignoring the window's background pixels. This is the way text is normally drawn, and the one we'll choose.

Change the `ID Number` value to 5, since this is the value we'll use in our program in chapter 4.

Click the `Done` button when finished making the selections. The window now has a vertical scroll bar and a size box.

Design Master

To see how easy it is to use the text edit control, select `Test run` from the `Edit` menu. You can click in the window and enter text, select text, and even use the `Cut`, `Copy`, and `Paste` commands. When you're finished testing the window, select the `End test run` command from the `Edit` menu.

Be sure to delete any text you've added to the window, and then save the current definition in binary format with the `Save as` command in the `File` menu. As with our other definitions, place your MiniWord disk on-line, then use the buttons to move to the `Binary` folder on the disk. Create a binary file named **window.bin**. Now use the `Append` command in the `File` menu to append the window's source code definition to the file named `cmds2.pas` in the `Source` folder on your disk. (For assembly language, append the source to the file `miniword2.asm`; for the C language, append the source to the file named `cmd2.cc`.)

Chapter 3

Writing A Text Editor In A Few Hours

In this chapter, we will walk through all aspects of creating a complete desktop program using Design Master. The program we will build is a simple text editor; we created its menu bar, windows, and all of its dialogs in the last chapter.

The source code presented is written in the Pascal programming language. Pascal is often used in programming examples. This is because the language is similar enough to English that programs can be easily understood by most programmers, including those who are not familiar with Pascal.

The source code used in this chapter is written for the ORCA/Pascal compiler, with Pascal definitions generated by Design Master. The complete program listing is given in Appendix A. If you worked through the examples in Chapter 2, then you should have the source code stored on your MiniWord disk, in a folder named Source. The source code is also stored on the Design Master disk in the Samples/Pascal.Samples/Source folder. Within Samples are separate folders for each of the languages Pascal, C, and assembly. If you do not know Pascal, or you prefer to program in one of the other languages, you can refer to the samples on the disk as you read this chapter. You may want to print a listing of the source code for your language before we begin.

Designing The Text Editor

The first step in writing a program is to develop an overall design. The preliminary design includes what functions the program will perform, and its expected input and output. From the general design, the functions are broken down into separate modules, and details of each module are filled in. This design method is called "top down" programming: you start at the top, developing a general picture of what the program will do, and proceed down to the lowest levels, the actual routines.

The Preliminary Design

The broad view of our program is that it's a modest text editor. We'd like it to provide the abilities to create new documents, edit existing ones, print documents, and save them to disk. Our user interface will conform to Apple's Human Interface Guidelines. Each document will appear in its own window, and we'll have a menu bar across the top of the desktop containing menus of the available commands.

Design Master

We need to decide what types of files we'll allow to be edited. Files are one type of input we must process. Another type is keyboard input, when the user enters commands, responds to dialogs, and performs editing.

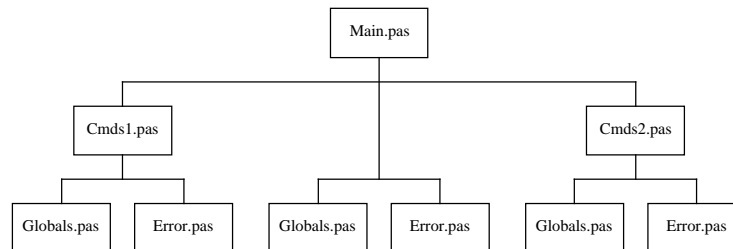
Our expected output will be modified files, new files, and print-outs of files.

The commands we need to process are specified in the program's menu items. If you worked through the examples in Chapter 2, you might recall that the menu commands we defined include About, New, Open, Close, Save, Save as, Page setup, Print, Quit, Cut, Copy, and Paste.

One other very important task we'll need to perform is the correct detection and handling of errors, both those returned by the tools (e.g. out of memory) and those generated by the user (e.g. trying to save a file to a write-protected disk). This is not a trivial job! As you'll see when we discuss the MiniWord program, a great deal of the source code will be devoted to error handling.

We're in luck that we're designing a desktop program on the Apple IIGS, since there's a standard way in which desktop programs are written. The program contains an initialization procedure, to create the desktop; a main loop, which detects and processes the user's commands; and a shut down procedure to exit the program.

Our program is beginning to take shape. We'll have a main module, which calls our initialization procedures, executes the main loop, and calls our shut down routine. We'll also have two modules to handle the commands, one for commands that interface with the file system, and one for the other commands. We'll use an error module that all of the other modules can call, and a global data module to hold the data structures that are common to all of the modules. A graphic representation of our preliminary design is given in the modules chart below.



The Detailed Design

As we flesh out the details of our program design, we need to answer the questions we raised about handling input and output. We'll see shortly that a basic understanding of how the tools work will provide the answers we seek.

A desktop program is driven by its main loop, which is only exited when the user chooses to quit the program or a catastrophic error occurs. At the top of the loop, the Event Manager is called to obtain the next event. Within the body of the loop, the appropriate routine to handle the event is dispatched. This type of loop is called an event loop.

An "event" is an action taken by the user. The tools define a number of events, summarized in the table below:

Chapter 3: Writing A Text Editor In A Few Hours

<u>Event Name</u>	<u>Event Description</u>
inButtDwn	button down event: user pressed mouse button
mouseUpEvt	button up event: user released mouse button
inKey	keystroke event: key was pressed
autoKeyEvt	auto key event: key held down by user
inUpdate	update event: a previously invisible portion of a window has now been exposed and needs to be repainted
activateEvt	activate event: previously inactive window needs to be made active
switchEvt	switch event: user wants to switch to another application
deskAccEvt	desk accessory event: event involves a desk accessory, not one of our windows
driverEvt	driver event
app1Evt	application 1 event
app2Evt	application 2 event
app3Evt	application 3 event
app4Evt	application 4 event
wInDesk	mouse clicked on the desktop, but not in a window or on the menu bar
wInMenuBar	user has selected a non-special menu item
wClickCalled	the Desk Manager's SystemClick routine was called to handle mouse-down events in a system window
wInContent	user clicked mouse in the content region of a window
wInDrag	user clicked mouse on title bar of window
wInGrow	user clicked mouse in grow box of an active window
wInGoAway	user clicked mouse in close box of an active window
wInZoom	user clicked mouse in zoom box of an active window
wInInfo	user clicked mouse in info bar region of window
wInSpecial	item ID selected was 250 - 255: user selected "special" menu command of undo, close, clear, cut, copy, or paste
wInDeskItem	item ID selected was 1 - 249: user selected a new desk accessory to be run
wInFrame	in Frame, but not on anything else: user clicked on frame of a window, but not in any of the frame's controls
wInactMenu	inactive menu item was "selected"
wClosedNDA	desk accessory was closed
wCalledSysEdit	the Desk Manager's SystemEdit routine was called to handle undo, cut, copy, paste, or clear command for a system window
wTrackZoom	zoom box clicked, but not selected
wHitFrame	user clicked on window's frame, making the window active
wInControl	mouse click or keystroke in control of the active window
wInSysWindow	mouse clicked in system a window

Design Master

We'll be using TaskMaster, built into the Window Manager, to control our main event loop. TaskMaster will greatly simplify our program, because it calls the Event Manager for us, and automatically handles a number of events, including using new desk accessories, activating/deactivating windows, and pulling down menus. It returns an event code that describes the type of event that has occurred, and other information needed to handle the event.

This solves the problem of obtaining menu commands from the user. Now we need to decide how we're going to handle reading files into document windows. The Standard File Operations tool set comes to mind, to handle getting the pathname of the file to open. We can then call GS/OS to open, read, and close the file, and the Text Edit tool set to paint the file's contents into a window.

There's one last type of input that needs to be considered: how do we edit the text in a window? Once again, the tools come to our rescue! We can define one text edit super control for each window we open on our desktop. Text Edit, in conjunction with TaskMaster and the Control Manager, will handle all of the editing chores for us, even Cut, Copy, and Paste!

When we're saving files to disk, we can call the Standard File Operations tool set to handle disk access and to obtain the pathname of the file to save. We can then call GS/OS to write the file to disk.

We can use the Print Manager to handle printing files.

One last detail requires our attention before we move on to a discussion of the MiniWord program. We need to define our initial data structures before we can begin to write the program. A method we'll employ to keep our text editor as uncomplicated as possible is to restrict the number of windows open at any one time to four. This allows us to use arrays to keep track of things, and avoids the headaches we might encounter with dynamic records, such as traversing linked lists and running out of memory.

Our main variables are those that keep track of window information. We'll also need variables to interface with the tools and GS/OS. Our other primary data structures are in the forms of records and templates generated by Design Master.

Since our MiniWord program is being designed around the concept of modules, we'll implement the Pascal program as a series of units, with one unit defining one logical module. We can now divide our data structures into those needed by all of the units (our global data), those needed throughout a unit but not by other units, and those used locally in individual subroutines. For example, the window-tracking information will be placed in the global data unit, the records needed to make GS/OS calls will be put in the unit that handles file-access commands, and the event record used by TaskMaster will be placed in the procedure containing the event loop.

The MiniWord Program

In the source code that follows, the MiniWord program will be put together unit by unit. We'll start with the main program, then look at the error unit, and end with the cmds1 and cmds2 units. Each subroutine will be discussed in detail. Global data accessed by the routines will be shown with the code to enhance its readability.

Chapter 3: Writing A Text Editor In A Few Hours

When we reach parts of the program that have been generated by Design Master, you will be instructed to replace the code in your MiniWord source file with the code you created in Chapter 2. If you worked through the examples in Chapter 2, your code will be found at the end of the source files. You should cut the appropriate code from the end of the file and paste it over the code it is to replace. The line numbers of the pasted-over code will be given. It is expected that you have an editor that can perform cut and paste operations, and that you know how to use your editor.

Our goals are to learn how to write a program which uses Design Master, and to learn the basics of desktop programming on the Apple IIGS computer. Armed with these noble aspirations, let's dive into the program.

The Main Program Module

The main programming unit contains the main program and the subroutines it calls, Init, EventLoop, and ShutDown. It also contains the top-level event handlers, HandleMenu, HandleSpecial, and HandleUpdate, called by the main event loop. The main module requires access to all of the other units, since the command handlers are dispatched from here.

The Main Program

The main program is very simple; it simply calls the Init function to initialize the program, then the EventLoop routine if Init successfully started the program, and ends with a call to the ShutDown procedure to wrap things up:

```
{ Main.pas, line 704 }
( ***** )
*
* Main program
*
***** )

begin

if Init then                                {start tools, bring up menu bar}
    EventLoop;                             {execute main event loop      }
ShutDown;                                  {unload tools                }

end.
```

The Init Function

The Init function loads the tools we'll need, creates our menu bar, and initializes our data structures. Any errors detected in Init are considered catastrophic, so we define Init as a function returning a boolean value. If Init returns true, then it was able to initialize the program; if it returns false, we skip calling the event loop and simply shut down the program.

Design Master

Init's first task is to record our user ID, passed to the program by the system loader. We'll need this user ID to pass to the Memory Manager when allocating memory.

There are a number of different methods used to obtain a user ID. Some programmers prefer to start the Memory Manager by issuing the MMStartUp call, which returns a user ID. (This is the same ID passed by the loader to your program, in the accumulator register.) For programs written in assembly language, the program typically stores the accumulator to a two-byte variable as one of its first instructions. For programs written in high-level languages, the compiler will usually take care of storing the user ID; the programmer can then access the ID from a compiler-defined location or by calling a function provided by the compiler. ORCA/Pascal uses the built-in function `userID` to return the ID to the programmer.

One advantage to using the master ID is that all memory belonging to the ID is automatically purged by the operating system when your program ends. The user ID that the compiler returns has also been modified slightly so you can use a `DisposeAll` call to dispose of any handles you allocate in the program.

A second way of obtaining a user ID for allocating private memory for your program is to take the master ID and change its `auxID` field. This is the method recommended by Apple, and the one that we use in our MiniWord program. The `auxID` field is contained in bits 11 through 8 of the master ID. You would generally use an OR operation to change the bits. For example, in our program, we give our user ID an `auxID` of 2 with the instruction:

```
myID := userID | 2;      {'|' = OR operation in ORCA/Pascal}
```

Another method of obtaining a user ID is to call the Miscellaneous tool set's `GetNewID` function. If you use this method, you must be sure to deallocate all memory you obtain with the ID, and to discard the user ID with the Miscellaneous tool set's `DeleteID` routine. This method is generally not recommended, since, if you forget to dispose of all of the memory you allocate, the program launcher cannot do it for you.

User IDs are explained in detail on pages 12-10 through 12-11 of volume 1 of the *Apple IIGs Toolbox Reference* manual.

```
{ Globals.pas, line 52 }
```

```
myID: integer;                                {MiniWord's user ID}
```

```
{ Main.pas, line 74 }
```

```
masterID: integer;                            {user ID passed by loader}
```

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Main.pas, line 311 }
procedure InitMenus; forward;
(* ***** *)
*
* Init - Initialize all global variables, start
*       the tools we need, create our menu bar.
*
* Output:
*       Returns true if everything started OK,
*       and false otherwise.
*
* ***** *)

function Init: boolean;

label 99;

var
    ok: boolean;

begin
    { Initialize all global variables. }

    Init      := true;                {assume all is well to start with }
    masterID  := userID;              {get user ID passed by loader }
    myID      := masterID | $0200;    {alter for our purposes, so we can do}
                                         { a simple DisposeAll at the end }
end;
```

Init next starts the tools required by MiniWord. The StartUpTools call is new with GS/OS version 5.0. We pass it a list of tools to be started, and it handles allocating direct page memory and making the startup call for each tool in the list. The call is described in detail in chapter 27 of the *Toolbox Update Notes*, which explains the changes made to the Tool Locator for version 5.0.

Let's look at the parameters that we pass to StartUpTools. The first parameter is our master ID. StartUpTools will use the ID to allocate all of the direct page memory required by the tools. The second parameter is called a "verb." These are also new in 5.0. A verb describes the type of the parameter which immediately follows it, either a pointer, a handle, or a resource ID. The verbs were introduced as a means of interfacing with the Resource Manager, while giving the application the choice of when to use resource forks, if at all. We are passing a pointer to the record containing the tool list, so the verb we use is the pointerVerb constant. (The verb constants are defined in the Common interface file for ORCA/Pascal.)

The last parameter we pass is a pointer to a start/stop record. The record is defined in the *Toolbox Update Notes*, on page 27-2:

```
(* Table of tools to load from the TOOLS directory in the SYSTEM folder *)
toolSpec = record
    toolNumber: integer;
    minVersion: integer;
end;

(* Change array size for your application. *)
ttArray = array [1..20] of toolSpec;
```

Design Master

```
toolTable = record
    numToolsRequired:    integer;
    tool:                ttArray;
end;

startStopRecord = record
    flags:               integer;
    videoMode:           integer;
    resFileID:           integer;
    DPageHandle:         handle;
    numTools:            integer;
    toolArray:           ttArray;
end;
```

The first field in the record is a flags field; at the time of publication, this integer must be set to zero. The second field specifies the screen mode that your program will use. Our program is in 640 mode, so we set this field to \$80. If you prefer 320 mode, you can set this field to zero. The next two fields are set by the StartUpTools call. If your program has a resource fork, the resource file ID assigned by the Resource Manager is returned in the resFileID field. The DPageHandle field contains a handle to the direct page area used by the tools. The handle should be regarded as "read-only," that is, do not disturb the tools' direct page area. The next field in the record is an integer that specifies the number of tools to be started. We're starting 20 tools. The final field is an array containing the tools we want to start. The tools will be started in the order specified in the array.

How do we know which tools to start? At the beginning of each chapter of the *Toolbox Reference* manuals in which a tool set is described, the other tools required by the tool set are listed. The start-up order for the tools is given in Appendix C of volume 2 of the *Toolbox Reference*, and is further explained in Tech Note #12 of the *Apple II Tech Notes*. To build the tool list, we start with the tools we know are needed to create a desktop program: the Event Manager, the Window Manager, the Control Manager, the Miscellaneous Tool Set, Quick Draw II, the Dialog Manager, and the Menu Manager. We then refer to our manuals to find out which tools these tools need, adding the new ones to our list. Finally, as we develop the program, we add new tools to perform required tasks, such as the Print Manager and the Text Edit Tool Set.

The StartUpTools call makes some changes to our record; it returns a pointer to the changed record. When it's time to unload the tools, we'll pass the pointer it returned to us. If, for some reason, the tools can't be started, we exit Init.

```
{ Main.pas, line 75 }

startStopAddr: longint;
startStopRec:  startStopRecord;           {GS/OS 5.0 1-stop load tools call}

{ Main.pas, line 338 }
{ Start tools we need: Use GS/OS 5.0 one-call startup mechanism. }
with startStopRec do begin
    flags      := 0;                      {flags must be zero }
    videoMode  := $80;                    {640 mode          }
```


Chapter 3: Writing A Text Editor In A Few Hours

```
numTools := 20;                                {we'll start 20 tools}
toolArray [1].toolNumber := $03;
toolArray [1].minVersion := $0300;             {Miscellaneous Toolset}
toolArray [2].toolNumber := $04;
toolArray [2].minVersion := $0300;             {Quick Draw II}
toolArray [3].toolNumber := $06;
toolArray [3].minVersion := $0300;             {Event Manager}
toolArray [4].toolNumber := $0E;
toolArray [4].minVersion := $0300;             {Window Manager}
toolArray [5].toolNumber := $10;
toolArray [5].minVersion := $0300;             {Control Manager}
toolArray [6].toolNumber := $0F;
toolArray [6].minVersion := $0300;             {Menu Manager}
toolArray [7].toolNumber := $14;
toolArray [7].minVersion := $0100;             {LineEdit Toolset}
toolArray [8].toolNumber := $15;
toolArray [8].minVersion := $0100;             {Dialog Manager}
toolArray [9].toolNumber := $08;
toolArray [9].minVersion := $0100;             {Sound Manager}
toolArray [10].toolNumber := $17;
toolArray [10].minVersion := $0100;            {Standard File Operations Toolset}
toolArray [11].toolNumber := $16;
toolArray [11].minVersion := $0104;            {Scrap Manager}
toolArray [12].toolNumber := $09;
toolArray [12].minVersion := $0100;            {Apple Desktop Bus Toolset}
toolArray [13].toolNumber := $05;
toolArray [13].minVersion := $0100;            {Desk Manager}
toolArray [14].toolNumber := $1C;
toolArray [14].minVersion := $0100;            {List Manager}
toolArray [15].toolNumber := $1B;
toolArray [15].minVersion := $0204;            {Font Manager}
toolArray [16].toolNumber := $13;
toolArray [16].minVersion := $0100;            {Print Manager}
toolArray [17].toolNumber := $12;
toolArray [17].minVersion := $0206;            {Quick Draw II Auxiliary}
toolArray [18].toolNumber := $0A;
toolArray [18].minVersion := $0100;            {SANE Toolset}
toolArray [19].toolNumber := $0B;
toolArray [19].minVersion := $0100;            {Integer Math Toolset}
toolArray [20].toolNumber := $22;
toolArray [20].minVersion := $0100;            {Text Edit Toolset}
end;
startStopAddr := StartUpTools (masterID, pointerVerb, ord4 (@startStopRec));
errNum := ToolError;
if errNum <> 0 then begin
  Init := false;
  goto 99;
end;
```

Notice that the button titles 'OK' and 'Cancel' are set in Init. This simplifies the program, since we can use the same strings in all of our dialogs and alerts that use OK and Cancel buttons. We'll look at the changes we need to make to the source code generated by Design Master to use these titles when we discuss the Error unit, later.

Design Master

```
{ Globals.pas, line 57 }
  okTitle:      packed array [0..2] of char;           {common button titles}
  cancelTitle: packed array [0..6] of char;

{ Main.pas, line 392 }
{ Initialize our units. }

okTitle      := 'OK';
cancelTitle  := 'Cancel';
```

Once the tools are loaded, Init calls the routines InitError, InitCmds1, and InitCmds2 to initialize the other units. If any of the units cannot be initialized, the Init function is exited, returning false. We'll look at the initialization routines for these units when we begin discussing them. InitMenus is a local procedure of the main module. We'll look at it after we finish discussing Init.

```
{ Main.pas, line 396 }
InitError;
if not (InitCmds1) then begin
  Init := false;
  goto 99;
end;
if not (InitCmds2) then begin
  Init := false;
  goto 99;
end;
InitMenus;
```

Init's final task is call Quick Draw II to change the cursor from a watch (set up by the StartUpTools call) to a standard arrow pointer. This is accomplished with the InitCursor call.

```
{ Main.pas, line 407 }
99:
InitCursor;                                {StartUpTools brings up the watch cursor}
                                           { so change it to the arrow          }
end;
```

The InitMenus Procedure

Let's see how MiniWord's menu bar is created. Take a close look at the menu definitions generated by Design Master. (If you worked through the examples in Chapter 2, you can use your editor to replace the lines below with the menu definitions that you generated. Recall that your definitions were appended to the end of the main module, main.pas for Pascal, main.cc for C, and miniword2.asm for assembly language.) Each menu has been placed in its own menu template, named menuxx, where the xx is a number, starting at 01 and increasing by one for each menu defined. Immediately following each menu template is a series of item templates, one for each

Chapter 3: Writing A Text Editor In A Few Hours

item in the menu. The menu items are named `menuxxitemyy`, where the `xx` identifies the item's menu, and the `yy` is a number, starting at 00, which uniquely identifies the item.

```
{ Main.pas, line 23 }
var
  { *** DATA STRUCTURES GENERATED BY DESIGN MASTER, and altered by B.A. *** }
  {
  { Note: Design Master will generate the type "pString" instead of
  { "packed array of char." These have been changed to use
  { less space.
  {
  dropmenutitle01: packed array [0..20] of char;           {menu titles}
  dropmenutitle02: packed array [0..20] of char;
  dropmenutitle03: packed array [0..20] of char;
  dropmenutitle04: packed array [0..20] of char;

  menu01itemtitle00: packed array [0..20] of char;         {menu item titles}

  menu02itemtitle00: packed array [0..20] of char;
  menu02itemtitle01: packed array [0..20] of char;
  menu02itemtitle02: packed array [0..20] of char;
  menu02itemtitle03: packed array [0..20] of char;
  menu02itemtitle04: packed array [0..20] of char;
  menu02itemtitle05: packed array [0..20] of char;
  menu02itemtitle06: packed array [0..20] of char;
  menu02itemtitle07: packed array [0..20] of char;

  menu03itemtitle00: packed array [0..20] of char;
  menu03itemtitle01: packed array [0..20] of char;
  menu03itemtitle02: packed array [0..20] of char;

  menu04itemtitle00: packed array [0..20] of char;

  menu01: menuTemplate;
  menu02: menuTemplate;
  menu03: menuTemplate;
  menu04: menuTemplate;

  menu01item00: menuItemTemplate;
  menu02item00: menuItemTemplate;
  menu02item01: menuItemTemplate;
  menu02item02: menuItemTemplate;
  menu02item03: menuItemTemplate;
  menu02item04: menuItemTemplate;
  menu02item05: menuItemTemplate;
  menu02item06: menuItemTemplate;
  menu02item07: menuItemTemplate;
  menu03item00: menuItemTemplate;
  menu03item01: menuItemTemplate;
  menu03item02: menuItemTemplate;
  menu04item00: menuItemTemplate;
```

Design Master

The first part of the InitMenus procedure has been generated by Design Master. It simply fills in all of the templates with values calculated from the choices we made when creating our menu bar.

If you are working through the examples, you will want to paste your menu initialization code over the lines below.

```
{ Main.pas, line 412 }
( ***** )
*
* InitMenus - Initialize the menu bar.
*
( ***** )

procedure InitMenus;

var
    tmp: longint;
    i: integer;

begin
    { *** GENERATED BY DESIGN MASTER, with comments provided by B.A. *** }

    dropmenutitle01 := '@';                                {Apple menu}
    with menu01 do begin
        version      := 0;
        menuID       := $0001;
        menuFlag     := $0008;                            {cache menu; will pass ptr to title}
        menuTitleRef := ord4 (@dropMenuTitle01);           {pointer to menu's title}
        itemRefs [1] := ord4 (@menu01item00);              {About item reference }
        itemRefs [2] := 0;                                  {null terminator     }
    end;

    menu01itemtitle00 := 'About';
    with menu01item00 do begin
        version      := 0;
        itemID       := 256;                                {About ID           }
        itemChar     := $00;                                {shortcut characters}
        itemAltChar  := $00;
        itemCheck    := $0000;
        itemFlag     := $0041;                              {bold, divider beneath, will}
                                                { pass pointer to title  }
        itemTitleRef := ord4 (@menu01itemtitle00);         {pointer to item's name}
    end;
```

Chapter 3: Writing A Text Editor In A Few Hours

```
dropmenutitle02 := ' File ';
with menu02 do begin
    version      := 0;
    menuID       := $0002;
    menuFlag     := $0008;           {cache menu; will pass ptr to title}
    menuTitleRef := ord4 (@dropMenuTitle02); {pointer to menu's title }
    itemRefs [1] := ord4 (@menu02item00);   {item reference: New      }
    itemRefs [2] := ord4 (@menu02item01);   {item reference: Open    }
    itemRefs [3] := ord4 (@menu02item02);   {item reference: Close   }
    itemRefs [4] := ord4 (@menu02item03);   {item reference: Save    }
    itemRefs [5] := ord4 (@menu02item04);   {item reference: Save as }
    itemRefs [6] := ord4 (@menu02item05);   {item reference: Page setup}
    itemRefs [7] := ord4 (@menu02item06);   {item reference: Print   }
    itemRefs [8] := ord4 (@menu02item07);   {item reference: Quit    }
    itemRefs [9] := 0;                     {null terminator        }
end;

menu02itemtitle00 := 'New';
with menu02item00 do begin
    version      := 0;
    itemID       := 257;               {new ID                }
    itemChar     := $4E;               {shortcut characters    }
    itemAltChar  := $6E;
    itemCheck    := $0000;
    itemFlag     := $0000;             {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle00); {pointer to item's name }
end;

menu02itemtitle01 := 'Open...';
with menu02item01 do begin
    version      := 0;
    itemID       := 258;               {open ID                }
    itemChar     := $4F;               {shortcut characters    }
    itemAltChar  := $6F;
    itemCheck    := $0000;
    itemFlag     := $0000;             {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle01); {pointer to item's name }
end;

menu02itemtitle02 := 'Close';
with menu02item02 do begin
    version      := 0;
    itemID       := 255;               {close ID               }
    itemChar     := $57;               {shortcut characters    }
    itemAltChar  := $77;
    itemCheck    := $0000;
    itemFlag     := $0000;             {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle02); {pointer to item's name }
end;
```

Design Master

```
menu02itemtitle03 := 'Save';
with menu02item03 do begin
    version      := 0;
    itemID       := 259;                      {save ID          }
    itemChar     := $53;                      {shortcut characters}
    itemAltChar  := $73;
    itemCheck    := $0000;
    itemFlag     := $0000;                    {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle03); {pointer to item's name }
end;

menu02itemtitle04 := 'Save as...';
with menu02item04 do begin
    version      := 0;
    itemID       := 260;                      {save as ID       }
    itemChar     := $00;                      {shortcut characters}
    itemAltChar  := $00;
    itemCheck    := $0000;
    itemFlag     := $0040;                    {will pass pointer to title}
                                           { draw divider beneath }
    itemTitleRef := ord4 (@menu02itemtitle04); {pointer to item's name }
end;

menu02itemtitle05 := 'Page setup...';
with menu02item05 do begin
    version      := 0;
    itemID       := 261;                      {page setup ID    }
    itemChar     := $00;                      {shortcut characters}
    itemAltChar  := $00;
    itemCheck    := $0000;
    itemFlag     := $0000;                    {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle05); {pointer to item's name }
end;

menu02itemtitle06 := 'Print...';
with menu02item06 do begin
    version      := 0;
    itemID       := 262;                      {print ID         }
    itemChar     := $50;                      {shortcut characters}
    itemAltChar  := $70;
    itemCheck    := $0000;
    itemFlag     := $0040;                    {will pass pointer to title}
                                           { divider beneath      }
    itemTitleRef := ord4 (@menu02itemtitle06); {pointer to item's name }
end;

menu02itemtitle07 := 'Quit';
with menu02item07 do begin
    version      := 0;
    itemID       := 263;                      {quit ID          }
    itemChar     := $51;                      {shortcut characters}
    itemAltChar  := $71;
    itemCheck    := $0000;
    itemFlag     := $0000;                    {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle07); {pointer to item's name }
```

Chapter 3: Writing A Text Editor In A Few Hours

```
end;

dropmenutitle03 := ' Edit ';
with menu03 do begin
    version      := 0;
    menuID       := $0003;
    menuFlag     := $0008;           {cache menu; will pass ptr to title}
    menuTitleRef := ord4 (@dropmenutitle03); {pointer to menu's title }
    itemRefs [1] := ord4 (@menu03item00);    {item reference: Cut      }
    itemRefs [2] := ord4 (@menu03item01);    {item reference: Copy    }
    itemRefs [3] := ord4 (@menu03item02);    {item reference: Paste   }
    itemRefs [4] := 0;                      {null terminator        }
end;

menu03itemtitle00 := 'Cut';
with menu03item00 do begin
    version      := 0;
    itemID       := 251;              {cut ID                  }
    itemChar     := $58;              {shortcut characters     }
    itemAltChar  := $78;
    itemCheck    := $0000;
    itemFlag     := $0000;            {will pass pointer to title}
    itemTitleRef := ord4 (@menu03itemtitle00); {pointer to item's name  }
end;

menu03itemtitle01 := 'Copy';
with menu03item01 do begin
    version      := 0;
    itemID       := 252;              {copy ID                 }
    itemChar     := $43;              {shortcut characters     }
    itemAltChar  := $63;
    itemCheck    := $0000;
    itemFlag     := $0000;            {will pass pointer to title}
    itemTitleRef := ord4 (@menu03itemtitle01); {pointer to item's name  }
end;

menu03itemtitle02 := 'Paste';
with menu03item02 do begin
    version      := 0;
    itemID       := 253;              {paste ID                }
    itemChar     := $56;              {shortcut characters     }
    itemAltChar  := $76;
    itemCheck    := $0000;
    itemFlag     := $0000;            {will pass pointer to title}
    itemTitleRef := ord4 (@menu03itemtitle02); {pointer to item's name  }
end;
```

After the menu templates have been initialized, `InitMenus` creates the menu bar. The first Menu Manager call that it makes is `NewMenu2`, which allocates space for a menu and its items. `NewMenu2` is a new call provided with GS/OS 5.0. It works just like the old `NewMenu` call, but a verb parameter has been added that is used to tell the Menu Manager the type of menu template that will be passed, either a pointer, handle, or resource ID. In our case, we're passing a pointer to the template. `NewMenu2` returns a handle to the menu just created.

Design Master

The next call we need to make is `InsertMenu`, which will insert the menu we just created with the `NewMenu2` call into our menu bar. We pass it the menu handle returned by the `NewMenu2` call.

We need to make a series of `NewMenu2` and `InsertMenu` calls, one for each menu in our menu bar. We build the menu bar from the last menu to the first, because the `insertAfter` parameter passed to the `InsertMenu` call is zero, which tells the Menu Manager that the menu is to be inserted at the front of the current menu list. Notice that neither call returns an error; thus, there's no code required to check for errors.

```
{ Main.pas, line 622 }
{ The rest of the code is this procedure is ours. }
{ Create the menu bar. Start at the last menu, since we're inserting each new }
{ menu at the front of the current menu list. }

tmp := ord4 (@menu04); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu03); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu02); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu01); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
```

After we've completely defined the menu bar, we call the Desk Manager's `FixAppleMenu` routine, passing it the menu ID of our Apple menu. The Desk Manager will then add all of the new desk accessories it finds in the boot disk's `Desk.Accs` folder of the `System` folder to our Apple menu. This call returns no errors.

The last two Menu Manager calls we make are `FixMenuBar`, which calculates the size of the menu bar, and `DrawMenuBar`, which paints our menu bar on our desktop. Neither call returns errors.

```
{ Main.pas, line 631 }
FixAppleMenu (1);           {add desk accessories to Apple menu }
i := FixMenuBar;           {compute standard sizes for menu bar and menus}
                           { throw away returned height }

DrawMenuBar;

end; {InitMenus}
```

The EventLoop Procedure

Chapter 3: Writing A Text Editor In A Few Hours

The EventLoop procedure, shown next, drives the MiniWord program. EventLoop is responsible for several important variables. The done flag drives its main loop. Done is a global variable because it can be set by our error handler, in case we detect an error from which the program cannot recover. EventLoop also maintains the window-tracking variables numWindows, the number of currently open windows; index, the index into the arrays for the currently active window; and currWindow, the grafPort pointer for the currently active window.

```
{ Globals.pas, line 35 }
type
    windowType = (noWindow, fromFile, fromNew);           {Window types}

{ Globals.pas, line 50 }
var
    done:          boolean;           {true if user is ready to exit MiniWord }
    {...}
    { Window tracking information -- We're allowing only 4 windows to be opened }
    { on the desktop. }

    windowOpen:    array [0..3] of windowType;           {array of open window flags }

{ Main.pas, line 98 }
{ Forward declarations of local subroutines that EventLoop calls. }

procedure HandleMenu (menuData: longint; var index, numWindows: integer;
                     var currWindow: grafPortPtr; var done: boolean); forward;

procedure HandleSpecial (menuData: longint; var index, numWindows: integer;
                        var currWindow: grafPortPtr); forward;

procedure HandleUpdate (theWindow: grafPortPtr); forward;

(*****
*
* EventLoop - Get next event, then dispatch the
*             appropriate routine to handle it.
*
*****)

procedure EventLoop;

{const}                                {Event codes returned by TaskMaster}

{...}
```

Design Master

```
var
  numWindows: integer;           {# of currently open windows      }
  index:      integer;           {window arrays index, active window }
  currWindow: grafPortPtr;       {current active window             }

  taskRecord: wmTaskRec;         {used to communicate with TaskMaster}
  eventCode:  integer;           {returned by TaskMaster             }
  i:          integer;

begin
  done := false;                 {we ain't done yet  }
  for i := 0 to 3 do             {no window open yet }
    windowOpen [i] := noWindow;
  numWindows := 0;
  index      := 0;
```

The first action taken in the event loop is to call the Window Manager's TaskMaster function. We pass it a bit flag that describes the types of events in which we're interested, and a pointer to an extended task record. It returns an event code. The event code tells our program what happened; for example, if the user pulls down the File menu and selects New, the Event Manager returns a code telling us that a menu command was selected, along with other values telling what menu and menu item were selected. The program uses the values returned by the event manager to call the appropriate event handler.

The task record has been extended for GS/OS version 5.0. You can find the general structure for an event in the Apple IIGS Toolbox Reference Manual, or you can look at the source code for the tool interfaces used with your compiler. In Pascal, the event record looks like this:

```
(* Event record *)
eventRecord = record
  eventWhat:      integer;
  eventMessage:   longint;
  eventWhen:      longint;
  eventWhere:     point;
  eventModifiers: integer;
  taskData:       longint;
  taskMask:       longint;
  lastClickTick:  longint;
  ClickCount:     integer;
  TaskData2:      longint;
  TaskData3:      longint;
  TaskData4:      longint;
  lastClickPt:    point;
end;
wmTaskRec = eventRecord;
```

Chapter 3: Writing A Text Editor In A Few Hours

The only information we'll need is the `wmTaskData` field. The other fields will be used by the tools to help us handle events.

The event mask we pass to TaskMaster is described on page 7-11 of volume 1 of the *Apple IIGS Toolbox Reference* manual. The event mask we use is `$076E`. This signals that we're interested in almost all event types detected by the Event Manager, including those involving desk accessories, application switching, window activation/deactivation, window updating, keyboard activities, and mouse button activities. The only events we don't want to know about are application-defined events (since we have none), and device driver events (since MiniWord is not a device driver).

We tell TaskMaster the events we want it to handle for us in the `wmTaskMask` field of the task record we pass it. The task mask is introduced on page 25-14 of volume 2 of the *Apple IIGS Toolbox Reference* manual, and is further described in chapter 28 of the *Toolbox Update* notes. The task mask we use in the MiniWord program, `$001FBFFF`, allows a number of events to be returned. The events that TaskMaster can return are explained in the following table:

<u>Bit</u>	<u>Flag name</u>	<u>Use with TaskMaster, when bit is set</u>
31-21	N/A	Reserved - must be set to zero.
20	<code>tmIdleEvents</code>	Passes idle events to the active controls of the active window. Enabling this allows Text Edit to draw a flashing insertion point in the active window while awaiting other chores to perform.
19	<code>tmMultiClick</code>	Puts multi-click information in the task record. Enabling this allows Text Edit to detect and handle single, double, and triple clicks in a text edit control.
18	<code>tmControlMenu</code>	Passes menu events to controls in the active window. Enabling this allows Text Edit to detect and handle the Cut, Copy, and Paste menu selections.
17	<code>tmControlKey</code>	Passes keyboard events to controls in the active window. Enabling this allows Text Edit to obtain and handle keyboard input.
16	<code>tmContentControls</code>	Calls the Control Manager's routines <code>FindControl</code> and <code>TrackControl</code> when the mouse button is clicked in the content region of an active window. Enabling this bit allows Text Edit to handle text selection for us.
15	<code>tmInfo</code>	A window won't be activated if the user clicks in its info bar. We set this since we're not using info bars.

Design Master

14	tmInactive	Returns the wInactMenu event code when the user selects an inactive menu item. We're not setting this bit in MiniWord because we're ignoring selection of inactive menu items.
13	tmCRedraw	Redraws a window's controls when a window activate event is detected. This frees us from having to handle activate events.
12	tmSpecial	TaskMaster will handle selection of special menu items, those with an ID less than 256. This lets the tools automatically handle Cut, Copy, and Paste, as well as desk accessories. It returns an event code of wInSpecial, with the menu item ID in the low word of wmTaskData.
11	tmScroll	The tools will handle scrolling the active window, and will activate an inactive window when the user clicks on its scroll bar.
10	tmGrow	The tools will automatically handle sizing windows for us.
9	tmZoom	The tools will automatically handle zooming windows for us.
8	tmClose	Calls the Window Manager's TrackGoAway function, returning wInGoAway if the user clicked on the window's close box. We can then call our window close routine.
7	tmContent	The tools will automatically activate an inactive window if the user clicks in its content region.
6	tmDragW	The tools will automatically handle dragging a window around on the desktop.
5	tmSysClick	Calls the Desk Manager's SystemClick routine, which handles mouse-down events in a desk accessory.
4	tmOpenNDA	Calls the Desk Manager's OpenNDA routine, which opens a new desk accessory selected by the user from the Apple menu.
3	tmMenuSel	Calls the Menu Manager's MenuSelect routine when the user selects a menu item. TaskMaster returns an event code of wInMenu and the menu item ID in the low word of wmTaskData.
2	tmFindW	Calls the Window Manager's FindWindow routine when a mouse-down event occurs, returning the appropriate event code.

Chapter 3: Writing A Text Editor In A Few Hours

1	tmUpdate	Since the window content procedure field in our window parameter list is nil, TaskMaster returns an event code of wInUpdate and the grafPort pointer of the window needing updating in wmTaskData.
0	tmMenuKey	Calls the Menu Manager's MenuKey and MenuSelect routines for us when the user selects a menu item with a key equivalent. It returns the event code wInMenu and the menu item ID in the low word of wmTaskData.

As you can see from the table, as well as the event loop code, the only events we need to handle are clicking in the close box of an open window, window updates, and selection of a menu item.

```
{ Main.pas, line 157 }
  taskRecord: wmTaskRec;           {used to communicate with TaskMaster}
  eventCode: integer;             {returned by TaskMaster}

taskRecord.taskMask := $001FBFFF; {let TaskMaster do almost everything}

{...}

while not (done) do begin          {execute event loop 'til we're done}

  { Call TaskMaster to get next event we need to handle. }

  eventCode := TaskMaster ($076E,   {event mask = just about everything}
                           taskRecord); {pointer to extended task record }

  errNum := ToolError;
  if errNum <> 0 then                {only error possible}
    HandleError (errNum, fatalErr, stopAlertTyp) { is messing up }
                                                { wmTaskMask field }

  { Window update event? }

  else if eventCode = inUpdate then
    HandleUpdate (grafPortPtr (taskRecord.taskData))

  { Cut, copy, paste, or close command? }

  else if eventCode = wInSpecial then
    HandleSpecial (taskRecord.taskData, index, numWindows, currWindow)
```

Design Master

```
{ Did user click in close box? }

else if eventCode = wInGoAway then begin
  if MyWindow (index, currWindow, numWindows) then
    DoClose (index, currWindow, numWindows)
  end

{ Non-special command? }

else if eventCode = wInMenuBar then
  HandleMenu (taskRecord.taskData, index, numWindows, currWindow, done)

end {while not done}

end; {EventLoop}
```

The MyWindow Function

The MyWindow function is called when the event loop detects some sort of window event; it returns true if the active window belongs to MiniWord, and false otherwise. If the active window is ours, it also sets the passed variable index to the index in the window arrays that corresponds to the active window, and sets the passed variable currWindow to the grafPort pointer of the active window.

MyWindow's first action is to check the value of the numWindows variable, which contains the number of MiniWord windows currently open. If the value is zero, then no application windows are up, and MyWindow returns false.

```
{ Main.pas, line 638 }
(*****
*
* MyWindow - Checks if front window is one of ours.
*
* Output:
*   true if it's one of ours; false otherwise
*
*****)

function MyWindow (* var index: integer; var currWindow: grafPortPtr;
  numWindows: integer): boolean *);

label 99;

var
  systemWind: boolean;           {true if the front window is a system window}
  tmp:        grafPortPtr;
  tmp2:       longint;

begin
  MyWindow := true;              {assume the front window is ours}
```

Chapter 3: Writing A Text Editor In A Few Hours

```
{ First check if any windows are open. }

if numWindows = 0 then begin
  MyWindow := false;
  goto 99;
end;
```

If numWindows is not zero, then MyWindow calls the Window Manager's FrontWindow routine to obtain the grafPort pointer of the active window. It saves the pointer in the local tmp variable, then passes tmp to the Window Manager's GetSysWFlag function. GetSysWFlag returns true if the window is a system window (i.e. belongs to a desk accessory), and false otherwise.

```
{ Main.pas, line 667 }
{ Now check if the window is one of ours. }

tmp      := FrontWindow;           {get grafPort pointer of active window}
systemWind := GetSysWFlag (tmp);   {active window belong to desk }
                                           { accessory? }

if systemWind then begin           {Yes - MyWindow is false, exit}
  MyWindow := false;
  goto 99;
end;
```

If the system window flag is false, MyWindow calls the Window Manager's GetWRefCon function to obtain the window's index into the window tracking arrays, and returns the value true to indicate that the active window is ours.

Notice the use of type casting with ORCA/Pascal in line 681:

```
index := convert (tmp2) .lsw;
```

We are telling the compiler to take the four bytes contained in tmp2, and treat them as being of type convert. Treating tmp2 as a convert record allows us to break the longint value into two integers. Index, an integer, can then be set to the least significant word of the four-byte value. We know that the high word will always be zero since we're only allowing four open windows on the desktop at once.

```
{ Globals.pas, line 44 }
convert = record           {for choosing whether to handle 4 bytes}
  case boolean of          { as 1 longint or 2 integers }
    true: (long: longint);
    false: (lsw, msw: integer);
  end;
```

Design Master

```
{ Main.pas, line 678 }
{ It's one of ours, so get index into window arrays from wRefCon field. }

tmp2      := GetWRefCon (tmp);
index     := convert (tmp2) .lsw;
currWindow := tmp;

99:
end;
```

The ShutDown Procedure

Our final top-level routine is named ShutDown. As the name implies, it shuts down our MiniWord program. Its first action is to dispose of all memory that we've allocated. Notice how easy it is to release all of our memory blocks with the DisposeAll call, using our altered user ID. ShutDown next shuts down all the tools we started, with the ShutDownTools call. As mentioned in the description of the Init call, we pass the routine the pointer to the startStop record returned by the StartUpTools call.

```
{ Main.pas, line 687 }
( *****
*
* ShutDown - Unload the tools we started.
*
***** )

procedure ShutDown;

begin
DisposeAll (myID);                {dispose of all memory we allocated}
ShutDownTools (pointerVerb, startStopAddr);    {shut down tools we started}
errNum := ToolError;
if errNum <> 0 then
    HandleError (errNum, fatalErr, stopAlertTyp);
end;
```


The HandleUpdate Procedure

The next level of routines we'll look at are those called directly by our top-level routines, the event handlers. The first subroutine we'll look at is `HandleUpdate`, which handles updating a window. Update events are generated when a portion of the window that previously wasn't visible is now visible, so that its content region needs to be redrawn. This can happen when windows are moved around on the desktop, or when a new window becomes the active window.

As you can see, the routine is extremely simple. We call the Window Manager's `BeginUpdate` routine to set up the window's `grafPort`, then call the Control Manager's `DrawControls` routine to redraw all of the window's controls (including the text edit control, which contains the window's entire content region), and finally call the Window Manager's `EndUpdate` routine to complete the update. TaskMaster has passed us the `grafPort` pointer for the window requiring updating in the `wmTaskData` field of its task record, which the event loop passes in the `theWindow` variable.

```
{ Main.pas, line 296 }
( ***** )
*
* HandleUpdate - Handle update event for active
*               window.
*
***** )

procedure HandleUpdate (* theWindow: grafPortPtr *);

begin
  BeginUpdate (theWindow);
  DrawControls (theWindow);
  EndUpdate   (theWindow);
end;
```

The HandleSpecial Procedure

The next event-handling routine we'll consider is `HandleSpecial`, which takes care of selection of "special" menu items, those whose item IDs range from 250 to 255. As we've already mentioned, Cut, Copy, and Paste are automatically handled for us, so that the only special item we need to take care of is Close.

We first obtain the menu item number from the low-order word of the `wmTaskData` field, passed to the event loop by TaskMaster, and passed to us in the `menuData` variable. If the menu item is the Close item, we call our close window routine, `DoClose`.

We then call the Menu Manager's `HiliteMenu` routine to unhighlight the menu that the user has pulled down, and exit `HandleSpecial`. We pass `HiliteMenu` a flag of false to unhighlight, and the menu ID of the menu to unhighlight. The menu ID has been passed to the event loop by TaskMaster, in the high-order word of `wmTaskData`. It is passed to us in the high-order word of the `menuData` variable.

Design Master

{ Globals.pas, line 20 }

```
cutID      = 251;
copyID     = 252;
pasteID    = 253;
closeID    = 255;
aboutID    = 256;
newID      = 257;
openID     = 258;
saveID     = 259;
saveAsID   = 260;
pSetUpID   = 261;
printID    = 262;
quitID     = 263;
findID     = 264;
```

{Menu item IDs}

{ Main.pas, line 264 }

```
(*****
*
* HandleSpecial - Handle special menu commands,
*                 menu IDs 250 - 255.
*
*****)

procedure HandleSpecial (* menuData: longint; var index, numWindows: integer;
                        var currWindow: grafPortPtr *);

var
    theItem: integer;
    menuNum: integer;

begin
    { Menu item ID is in low-order word of the taskData field of our task record. }
    { Menu ID is in the high-order word of the field. }

    theItem := convert (menuData) .lsb;
    menuNum := convert (menuData) .msb;

    { Close command selected? All other special items handled by TaskMaster. }
    if theItem = closeID then
        if MyWindow (index, currWindow, numWindows) then
            DoClose (index, currWindow, numWindows);

    { Unhighlight the menu they just pulled down. }

    HiliteMenu (false, menuNum);
end; {HandleSpecial}
```

The HandleMenu Procedure

Our next event handler is HandleMenu, which determines the menu command selected by the user and then dispatches the appropriate routine. The menu item ID is passed to the event loop

Chapter 3: Writing A Text Editor In A Few Hours

by TaskMaster, in the low-order word of the wmTaskData field of its task record. The event loop passes this field to us in the menuData variable.

As in the HandleSpecial routine, we need to unhighlight the menu that the user has pulled down after returning from the menu item routine.

```
{ Main.pas, line 209 }
(*****)
*
* HandleMenu - Handle menu selections, menu IDs
*             256 ->.
*
*****)

procedure HandleMenu (* menuData: longint; var index, numWindows: integer;
                      var currWindow: grafPortPtr; var done: boolean *);

type
  menuIDs = 256..264;

var
  theItem: menuIDs;
  menuNum: integer;

begin
  { Menu item ID is in low-order word of the taskData field of our task record. }
  { Menu ID is in the high-order word of the field. }

  theItem := convert (menuData) .lsw;
  menuNum := convert (menuData) .msw;

  { Dispatch the appropriate routine, based on menu item selected. }

  case theItem of
    aboutID:    DoAbout;

    newID:      DoNew (index, numWindows);

    openID:     DoOpen (index, numWindows);

    saveID:     if MyWindow (index, currWindow, numWindows) then
                  DoSave (index, currWindow);

    saveAsID:   if MyWindow (index, currWindow, numWindows) then
                  DoSaveAs (index, currWindow);

    pSetUpID:   DoPSetUp;

    printID:    if MyWindow (index, currWindow, numWindows) then
                  DoPrint (index, currWindow);

    quitID:     DoQuit (done, index, numWindows, currWindow);

    findID:     DoFind;
  end;
```

Design Master

```
{ Unhighlight the menu they just pulled down. }

HiliteMenu (false, menuNum);
end;
```

The DoQuit Procedure

The last subroutine we need to examine is DoQuit, called when the user selects the Quit command from the File menu. DoQuit begins with a loop, which is repeated until all of MiniWord's windows have been closed. It continually calls the MyWindow function until the function returns false, at which time we know that all document windows have been closed. Within the body of the loop, the DoClose routine is called to close the currently active window. Once all windows are closed, DoQuit sets the passed flag done to true to indicate the user is finished running MiniWord.

```
{ Main.pas, line 82 }
( ***** )
*
* DoQuit - Handle Quit command.
*
( ***** )

procedure DoQuit (var done: boolean; var index, numWindows: integer;
                  var currWindow: grafPortPtr);
begin
  while (MyWindow (index, currWindow, numWindows)) do      {close all open windows}
    DoClose (index, currWindow, numWindows);

done := true;                                              {set the done flag}
end;
```

The Error Unit

The Error unit is responsible for reporting errors detected while MiniWord is running. It divides errors into two main categories: those from which the program can recover, and those which are so severe that we simply abort the program.

The InitError Procedure

The Error unit has an initialization procedure, InitError, called during start up of the MiniWord program. Its first action is to initialize the unit's array of error messages, errMsg.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Globals.pas, line 41 }
  pString50 = packed array [0..50] of char;

{ Error.pas, line 23 }

                                {Error numbers}

errType = (OOM, fatalPrintErr, memoryErr, openErr, readErr, printErr,
           writeErr, fileErr, createErr, deleteErr, getTextErr, getFileErr,
           getTextInfoErr, fatalErr);

{ Error.pas, line 33 }
  errMsg: array [errType] of pString50;           {error messages      }

{ Error.pas, line 106 }
(*****
*
* InitError - Initializes the Error unit.
*
*****)

procedure InitError;

begin
errMsg [OOM]           := 'Out of memory.  Aborting MiniWord.';
errMsg [fatalPrintErr] := 'Fatal error reported by Print Manager.';
errMsg [memoryErr]     := 'Memory error:  Unable to perform operation.';
errMsg [openErr]       := 'Error returned by GS/OS when opening file.';
errMsg [readErr]       := 'Error returned by GS/OS when reading file.';
errMsg [printErr]      := 'Error:  Aborting printing.';
errMsg [writeErr]      := 'Error returned by GS/OS when writing file.';
errMsg [fileErr]       := 'Error when accessing file.';
errMsg [createErr]     := 'Error returned by GS/OS when creating file.';
errMsg [deleteErr]     := 'Error returned by GS/OS when deleting file.';
errMsg [getTextErr]    := 'Error returned by TextEdit when reading file.';
errMsg [getFileErr]    := 'Error returned by SFO when accessing file.';
errMsg [getTextInfoErr] := 'Error returned by Text Edit when getting info.';
errMsg [fatalErr]      := 'Fatal error:  Cannot recover.';
```

The second task that the InitError procedure must perform has been generated by Design Master. It simply fills in all of the error dialog templates with values calculated from the choices made when creating the dialog.

If you are working through the examples, you will first want to paste your error dialog variables over those at the top of the error.pas file, and then paste your error dialog initialization code over those in the InitError procedure.

Design Master

```
{ Error.pas, line 47 }
var
  { *** GENERATED BY DESIGN MASTER *** }

  errAlert:      alertTemplate;
  ITEM00Err1:    itemTemplate;
  ITEM01Err1:    itemTemplate;
  ITEM02Err1:    itemTemplate;
  item00pointerErr1: packed array [0..60] of char;
  item01pointerErr1: packed array [0..5] of char;

{ Error.pas, line 130 }
{ *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
{ Initialize the error alert. }

with errAlert do begin
  with atBoundsRect do begin
    v1 := $002A;
    h1 := $004C;
    v2 := $0082;
    h2 := $01F9;
  end;
  atAlertID := 1;
  atStage1 := $81;
  atStage2 := $81;
  atStage3 := $81;
  atStage4 := $81;
  atItemList [1] := @item00Err1;
  atItemList [2] := @item01Err1;
  atItemList [3] := @item02Err1;
  atItemList [4] := nil;
end;

with ITEM00Err1 do begin
  itemID := $0064;
  with itemRect do begin
    v1 := 30;
    h1 := 10;
    v2 := 45;
    h2 := 409;
  end;
  itemType := $800F;
  itemDescr := @item00pointerErr1;
  itemValue := 50;
  itemFlag := 0;
  itemColor := nil;
end;
item00pointerErr1 := 'Here is a message that is fifty characters long!!!';

{Error-handling alert template }
{enclosing rectangle for alert }
{alert ID number }
{stage 1: draw alert, emit 1 beep}
{stage 2: draw alert, emit 1 beep}
{stage 3: draw alert, emit 1 beep}
{stage 4: draw alert, emit 1 beep}
{item pointer: Error message }
{item pointer: Error number }
{item pointer: OK button }
{null terminator }
{Error message item template }
{item ID number }
{bounding rectangle }
{static text + item disable }
{pointer to error message }
{length of static text to display }
{default flag }
{no color table }
```

Chapter 3: Writing A Text Editor In A Few Hours

```
with ITEM01Err1 do begin
    itemID := $0065;
    with itemRect do begin
        v1 := 50;
        h1 := 150;
        v2 := 65;
        h2 := 200;
    end;
    itemType := $800F;
    itemDescr := @item01pointerErr1;
    itemValue := 5;
    itemFlag := 0;
    itemColor := nil;
end;
item01pointerErr1 := '$0000';

with ITEM02Err1 do begin
    itemID := 1;
    with itemRect do begin
        v1 := 70;
        h1 := 200;
        v2 := 85;
        h2 := 230;
    end;
    itemType := $000A;
    itemDescr := @okTitle;
    itemValue := 0;
    itemFlag := $0001;
    itemColor := nil;
end;

end; {InitError}

end. {Error unit}
```

Notice the line:

```
    itemDescr := @okTitle;           {pointer to button's title }
```

Wherever any of our dialog item templates describe a pointer to a commonly-used title string, we can substitute the address (the @ operator in ORCA/Pascal) of the global string for the address of the string generated by Design Master. Of course, we should delete the string generated by Design Master. Using the same strings where we can will make our program smaller and more efficient.

The HandleError Procedure

The HandleError subroutine uses the error alert that we created in the last chapter. There are two changes to the alert's item templates that we'll need to make every time HandleError is called.

Design Master

The first change we make is storing a pointer to the correct error message to be displayed. This is accomplished by using the `whichErr` variable, passed as a parameter to the routine, as an index into the `errMsg` array. The address of the appropriate error message is obtained from the array and then stored into the `itemDescr` field of the `item00Err1` record.

```
{ Globals.pas, line 38 }
                                {Alert types}
    alertType = (stdAlertTyp, stopAlertTyp, noteAlertTyp, cautionAlertTyp);

{ Error.pas, line 32 }
    errNum: integer;                {error # returned by tools}

{ Error.pas, line 57 }
( *****
*
* HandleError - Report errors detected in
*               MiniWord.
*
* Input:
*     error      - error number returned by tool
*     whichErr   - error message #
*     whichAlert - alert type
*
***** )

procedure HandleError (* error: integer; whichErr: errType;
                      whichAlert: alertType *) ;

var
    tmp: longint;
    junk: integer;

begin
    { Get error message to display. }

    item00err1.itemDescr := @errMsg [whichErr];
```

The second change we need to make is to transform the `errNum` integer passed to the subroutine into a hexadecimal value, which is the form used in the *Toolbox Reference* manuals for error numbers. We call the Integer Math tool set's `Int2Hex` procedure to obtain a 4-digit hexadecimal number, passing it the integer to be converted, the address of the field in which to store the result, and the length of the field.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Error.pas, line 82 }
{ Convert integer error number to hex string in order to display error number }
{ in the same format as used by GS/OS and the tools. }

tmp := ord4 (item01err1.itemDescr) + 2;      {adjust address obtained from }
                                           { alert item template to point}
                                           { beyond length byte and '$' }
Int2Hex (error, tmp, 4);      {call Integer Math toolset to perform conversion}
```

After setting up the fields for the alert, we then call the appropriate Dialog Manager alert function, based on the whichAlert parameter passed to the HandleError procedure. There is only item in the alert that has been enabled, the OK button, so we simply discard the item hit integer returned by the alert. Note that if the alert type is stopAlert, we set the program's done flag; this is one of our methods of handling severe error conditions.

```
{ Error.pas, line 90 }
{ Bring up alert. }

case whichAlert of
  stdAlertTyp:      junk := Alert (errAlert, nil);

  noteAlertTyp:     junk := NoteAlert (errAlert, nil);

  cautionAlertTyp:  junk := CautionAlert (errAlert, nil);

  stopAlertTyp:     begin
                    junk := StopAlert (errAlert, nil);
                    done := true;
                    end;

end;
end;
```

The Cmds1 Unit

The Cmds1 unit handles the menu commands that do not deal directly with disk files, including About, Close, Page setup, and Print. The subroutines in this unit are called from Main.pas.

The InitCmds1 Function

The Cmds1 unit contains a routine to initialize the unit, the InitCmds1 function. The first item InitCmds1 needs to handle is allocating memory for a print record, and then calling the Print Manager to initialize the record. Memory is obtained on the Apple IIGS by calling the Memory Manager's NewHandle function. Of special importance is the memory attributes bit-flag parameter, described on page 12-37 of volume 1 of the *Apple IIGS Toolbox Reference* manual. The memory handle for the print record needs to be locked (i.e. it cannot be moved or purged). If the request for memory cannot be satisfied, we report the error and exit InitCmds1. If we're successful in obtaining memory for the print record, we then call the Print Manager's PrDefault

Design Master

routine to initialize the print record. If an error is returned, we handle it in the same way as the other errors detected in MiniWord.

```
{ Cmds1.pas, line 66 }
{ Our data structures, global to the Cmds1 unit. }

printHandle: prHandle;           {print record handle           }

{ Cmds1.pas, line 367 }
( *****
*
* InitCmds1 - Initialize the Cmds1 unit's data
*             structures.
*
***** )

function InitCmds1 (* : boolean *);

label 99;

begin
InitCmds1 := true;

{ Create print record. First allocate memory to obtain handle to record, then }
{ call the Print Manager to initialize it. Attributes are locked, don't purge, }
{ don't move. }

printHandle := prHandle (NewHandle (140, myID, $C010, nil));
errNum      := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, memoryErr, stopAlertTyp);
    InitCmds1 := false;
    goto 99;
end;

PrDefault (printHandle);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, fatalPrintErr, stopAlertTyp);
    InitCmds1 := false;
    goto 99;
end;
```

InitCmds1 next initializes the About and Want-to-Save dialogs. The variables and initialization code have been generated by Design Master. If you are working through the examples, you can paste the source code you generated over the template definitions at the top of the cmdsl.pas file, and also paste your initialization code over the appropriate lines in the InitCmds1 function.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds1.pas, line 50}
var
  { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }

  aboutDlg:      dialogTemplate;           {About dialog box}
  ITEM00Abt1:    itemTemplate;
  ITEM01Abt1:    itemTemplate;
  item00pointerAbt1: packed array [0..100] of char;
  item01colorsAbt1: colorTable;

  saveAlert:     alertTemplate;           {WantToSave alert box}
  ITEM00Save1:   itemTemplate;
  ITEM01Save1:   itemTemplate;
  ITEM02Save1:   itemTemplate;
  item00pointerSave1: packed array [0..30] of char;

{ Cmds1.pas, line 402 }
{ *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }

with aboutDlg do begin
  with dtBoundsRect do begin           {Enclosing rectangle}
    v1 := $002B;
    h1 := $00C4;
    v2 := $009C;
    h2 := $01B9;
  end;
  dtVisible      := true;              {Visiblilty flag      }
  dtRefCon       := 0;                {RefCon, for application use}
  dtItemList [1] := @item00Abt1;      {item pointer:  message  }
  dtItemList [2] := @item01Abt1;      {item pointer:  OK button }
  dtItemList [3] := nil;              {null terminator        }
end;

with ITEM00Abt1 do begin               {About dialog's static text item}
  itemID := $0064;                    {Item ID number        }
  with itemRect do begin              {bounding rectangle      }
    v1 := 0004;
    h1 := 0008;
    v2 := 0072;
    h2 := 0241;
  end;
  itemType := $800F;                 {static text + disable }
  itemDescr := @item00pointerAbt1;    {pointer to static text}
  itemValue := 0096;                 {length of text        }
  itemFlag := 0;                     {default flag          }
  itemColor := nil;                  {pointer to color table}
end;

item00pointerAbt1 := '                MiniWord';
item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), chr ($0D));
item00pointerAbt1 := concat (item00pointerAbt1, 'A simple word processor written');
item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), 'by Barbara Allred and ');
item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), 'Design Master', chr ($0D));
```

Design Master

```

with ITEM01Abt1 do begin
    itemID := $0001;
    with itemRect do begin
        v1 := 0089;
        h1 := 0094;
        v2 := 0102;
        h2 := 0149;
    end;
    itemType := $000A;
    itemDescr := @okTitle;
    itemValue := 0;
    itemFlag := $0001;
    itemColor := @item01colorsAbt1;
end;

item01colorsAbt1 [0] := $0010;
item01colorsAbt1 [1] := $00D0;
item01colorsAbt1 [2] := $0070;
item01colorsAbt1 [3] := $00E8;
item01colorsAbt1 [4] := $00B9;

with saveAlert do begin
    with atBoundsRect do begin
        v1 := $0028;
        h1 := $009C;
        v2 := $007A;
        h2 := $017B;
    end;
    atAlertID := 2;
    atStage1 := $81;
    atStage2 := $81;
    atStage3 := $81;
    atStage4 := $81;
    atItemList [1] := @item00Savel;
    atItemList [2] := @item01Savel;
    atItemList [3] := @item02Savel;
    atItemList [4] := nil;
end;

with ITEM00Savel do begin
    itemID := $0064;
    with itemRect do begin
        v1 := 37;
        h1 := 8;
        v2 := 47;
        h2 := 217;
    end;
    itemType := $800F;
    itemDescr := @item00pointerSavel;
    itemValue := 28;
    itemFlag := 0;
    itemColor := nil;
end;

item00pointerSavel := 'Save changes before closing?';

```

{About dialog's OK button}

{bounding rectangle }

{simple button }

{pointer to button's title}

{bold, round-cornered }

{ptr to button's color tbl}

{button outline color }

{interior color when not highlighted}

{interior color when highlighted }

{text color when not highlighted }

{text color when highlighted }

{WantToSave alert template}

{bounding rectangle }

{Alert ID number }

{stage 1: draw alert, 1 beep }

{stage 2: draw alert, 1 beep }

{stage 3: draw alert, 1 beep }

{stage 4: draw alert, 1 beep }

{item pointer: Save message }

{item pointer: OK button }

{item pointer: Cancel button}

{null terminator }

{save alert's message template}

{bounding rectangle }

{static text + item disable}

{pointer to text }

{length of text }

{default flag }

{no color table }

Chapter 3: Writing A Text Editor In A Few Hours

```
with ITEM01Save1 do begin
    itemID := 1;
    with itemRect do begin
        v1 := 62;
        h1 := 12;
        v2 := 75;
        h2 := 67;
    end;
    itemType := $000A;
    itemDescr := @okTitle;
    itemValue := 0;
    itemFlag := $0003;
    itemColor := nil;
end;

with ITEM02Save1 do begin
    itemID := 2;
    with itemRect do begin
        v1 := 62;
        h1 := 115;
        v2 := 75;
        h2 := 201;
    end;
    itemType := $000A;
    itemDescr := @cancelTitle;
    itemValue := 0;
    itemFlag := $0002;
    itemColor := nil;
end;

99:
end; {InitCmds1}
```

```
{save alert's OK button template}

{bounding rectangle          }

{simple button                }
{pointer to button's title   }

{bold, square-cornered button}
{no color table              }
```

```
{save alert's Cancel button template}

{bounding rectangle          }

{simple button                }
{pointer to button's title   }

{plain, square-cornered button}
{no color table              }
```

The DoAbout Procedure

The DoAbout procedure is called when the user selects the About item from the Apple menu. The source code to handle our About item is very simple – all we need to do is to make three Dialog Manager calls. We first call the GetNewModalDialog routine to create our dialog on the desktop, passing it a pointer to the template created by Design Master. It returns a pointer to the dialog's grafPort. Next we call ModalDialog, which detects and handles events in the currently active modal dialog box. This call returns the item ID of the item selected by the user. Our final call is to CloseDialog, which shuts down the About dialog. We pass it the grafPort pointer passed to us by the GetNewModalDialog call.

Design Master

```
{ Cmds1.pas, line 76 }
(*****
*
* DoAbout - Handle About command.
*
*****)

procedure DoAbout;

var
  theDialog: grafPortPtr;           {pointer to About dialog's grafPort}
  junk:      integer;               {item hit returned by ModalDialog }

begin
  theDialog := GetNewModalDialog (aboutDlg);           {create modal dialog}
  errNum    := ToolError;
  if errNum <> 0 then
    HandleError (errNum, memoryErr, cautionAlertTyp)

  else begin
    junk := ModalDialog (nil);           {call Dialog Mgr to detect}
                                           { user clicking OK button}
                                           { use default filter proc}

    errNum := ToolError;
    if errNum <> 0 then
      HandleError (errNum, fatalErr, stopAlertTyp) {only error is front window}
                                           { not modal dialog! }
    else
      CloseDialog (theDialog);
    end;
  end;

end; {DoAbout}
```

The DoClose Procedure

The DoClose procedure is called when the user selects the Close command from the File menu, when the close box of a window is clicked, and by the DoQuit routine before exiting MiniWord.

DoClose first enables the New and Open commands in the File menu, since we're now assured of having open slots in our window tracking arrays. This is a short-cut; we could go to the trouble of first checking whether they were disabled, and then enable them, but it would take just as long to see if the items were enabled as it would to enable them, so we just skip the check and enable them on each call.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds1.pas, line 109 }
(*****)
*
* DoClose - Handle Close command.
*
*****)

procedure DoClose (* index: integer; currWindow: grafPortPtr;
                  var numWindows: integer *);

var
    tmp:   ctlRecHndl;
    tePtr: teRecPtr;
    flag:  integer;

begin
    EnableMItem (newID);
    EnableMItem (openID);
```

```
{pointer to textEdit control's}
{ record for front window    }
{flag containing dirty bit in }
{ teRecord for front window  }
```

```
{can now create new window}
{can now open a file      }
```

DoClose next checks the window's dirty bit. If the contents have changed, we need to give the user a chance to save the window's contents before closing the window. In order to determine whether the file has changed, the ctrlFlag field in the text edit control's record is read. If the dirty bit (bit 6) in the flag is set, then the file has changed. (Text Edit automatically sets this bit when the control's contents change.)

To access the dirty bit, we first dereference the handle for the text edit control, returned by NewControl2 when we created the window. We then access the ctrlFlag field from the record pointer, ANDing the value with \$0040 to mask out all bits but the one in which we're interested. If the value is not zero, then Text Edit has set the dirty bit to indicate that the window's contents have changed, and we call the WantToSave routine. If the value is zero, no change has occurred, and we skip asking if they'd like to save the file.

```
{ Globals.pas, line 12 }
const
    isDirty    = $0040;
    notDirty   = $FFBF;

{ Cmds1.pas, line 118 }
var
    tmp:   ctlRecHndl;
    tePtr: teRecPtr;
    flag:  integer;
```

```
{mask to check dirty bit      }
{mask to clear dirty bit after saving }
{ file to disk                }
```

```
{pointer to textEdit control's}
{ record for front window    }
{flag containing dirty bit in }
{ teRecord for front window  }
```

Design Master

```
{ Cmds1.pas, line 128 }
{ Check if they want to save data before closing window. Dereference      }
{ textEdit control's handle in order to check if the dirty bit has been set }
{ by Text Edit. If it has, give user chance to save window before closing }
{ it.                                                                    }

tmp    := textEdHandle [index];
tePtr  := teRecPtr (tmp^);

if (tePtr^.ctrlFlag & isDirty) <> 0 then
    WantToSave (index, currWindow);
```

The window is now closed by calling the Window Manager's CloseWindow routine. We pass CloseWindow the active window's grafPort pointer, contained in the passed variable currWindow. At the same time, we decrement the window counter, the passed variable numWindows, since there's one less window open on the desktop.

```
{ Cmds1.pas, line 138 }
CloseWindow (currWindow);

numWindows := numWindows - 1;           {one less window open on desktop}
```

We next need to check whether the window we just closed was associated with a disk file, and if so, to dispose of its pathname handle. (The path handle was allocated by the Standard File Operations tool set when the file was opened. It is our responsibility to dispose of the handle when we're finished with it. This does *not* dispose of the disk file!) We make the check by consulting the windowOpen array for this window. If the element for this window contains the enumeration constant fromFile, a disk file is tied to the window and we proceed to dispose of the pathHandle.

DoClose's last task is to place a noWindow constant in the windowOpen array for this window, to indicate that the slot is open for the next window.

```
{ Cmds1.pas, line 142 }
{ If window allocated by Open command, free memory used by it. }

if windowOpen [index] = fromFile then
    DisposeHandle (pathHandle [index]);

windowOpen [index] := noWindow;           {free up slot in window tracking array}

end; {DoClose}
```

The WantToSave Procedure

The WantToSave procedure is called when the program needs to see if the user wants to save the contents of a window that is about to be closed. It brings up the save alert we created in the last chapter by calling the Dialog Manager's NoteAlert routine. NoteAlert returns the item ID

Chapter 3: Writing A Text Editor In A Few Hours

of the item selected, either a 1 for the OK button, or a 2 for the Cancel button. If the user selects the OK button, we call our DoSave procedure, described later in this chapter.

```
{ Cmds1.pas, line 527 }
(*****
*
* WantToSave - Ask user if they'd like to save
*               a file before closing its window.
*
*****)

procedure WantToSave (* index: integer, currWindow: grafPortPtr *);

var
    result: integer;

begin
    { Bring up want-to-save alert.  If user selects OK button, call DoSave to }
    { save the window to disk.                                           }

    if (NoteAlert (saveAlert, nil)) = 1 then
        DoSave (index, currWindow);

end; {WantToSave}
```

The DoPSetup Procedure

Printing documents is handled in our MiniWord program in the two subroutines DoPSetup and DoPrint. DoPSetup is called when the user selects the Page setup command from the File menu, while DoPrint is called when the Print command is selected.

The DoPSetup subroutine is one of the simplest in the program. It calls the Print Manager's PrStlDialog routine to bring up the Page Setup dialog, passing it the handle to the program's print record. The dialog returns an integer result, which we discard since the PrStlDialog call does all the work. Some of the fields in the print record are set as a result of this dialog; we'll point out each field as we discuss the Page Setup dialog.

The print record is described on pages 15-10 through 15-14 in volume 1 of the *Toolbox Reference* manual. Its form is:

```
(* Printer information subrecord *)
prInfoRec = record
    iDev:    integer;
    iVRes:   integer;
    iHRes:   integer;
    rPage:   rect;
end;
```

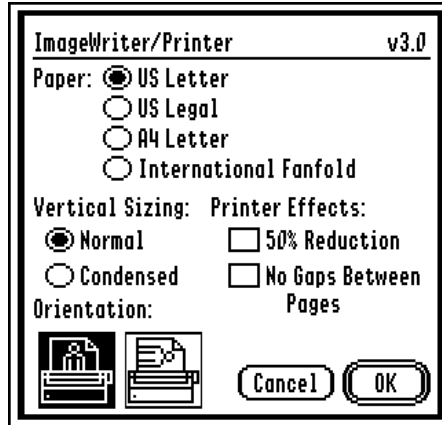
Design Master

```
(* Printer style subrecord *)
prStyleRec = record
    wDev:          integer;
    internA:       array [0..2] of integer;
    feed:          integer;
    paperType:     integer;
    case boolean of
        true:      (crWidth:    integer;);
        false:     (vSizing:    integer;
                    reduction:  integer;
                    internB:    integer;);
    end;

(* Job information subrecord *)
prJobRec = record
    iFstPage:      integer;
    iLstPage:      integer;
    iCopies:        integer;
    bJDocLoop:     byte;
    fFromUser:     byte;
    pIdleProc:     procPtr;
    pFileName:     pathPtr;
    iFileVol:      integer;
    bFileVers:     byte;
    bJobX:         byte;
    end;

(* Print record *)
PrRec = record
    prVersion:     integer;
    prInfo:        prInfoRec;
    rPaper:        rect;
    prStl:         prStyleRec;
    prInfoPT:      array [0..13] of byte;
    prXInfo:       array [0..23] of byte;
    prJob:         prJobRec;
    printX:        array [0..37] of byte;
    iReserved:     integer;
    end;
```

The Page Setup dialog for an ImageWriter® printer, looks like this:



At the top of the dialog is the name of the printer attached to your computer and the current version of the printer driver that MiniWord found in Drivers folder of your System folder. The version number is given in the prVersion field of the print record; it was placed there when we issued the PrDefault call in our InitCmds1 function. The name of the printer is derived from the Control Panel new desk accessory, available in our Apple menu. It is found in the print record in the iDev field of the prInfo subrecord.

Beneath the header are four radio buttons. These are used to tell the Print Manager what size of paper is in the printer. The various paper sizes are described on page 15-5 of volume 1 of the *Apple IIGS Toolbox Reference* manual:

<u>Paper type</u>	<u>Dimensions</u>
US Letter	8.5 by 11.0 inches
US Legal	8.5 by 14.0 inches
A4 Letter	210 by 297 millimeters
B5 Letter	176 by 250 millimeters
International fanfold	210 millimeters by 12 inches

The paper type selected is placed into the paperType field of the style subrecord of the print record.

The Vertical sizing buttons are used to select the height of the text on the printer. For the ImageWriter, appropriate bits are set in the wDev field of the style subrecord. On the LaserWriter, the vSizing field of the style subrecord is filled in.

The Printer Effects buttons, on the ImageWriter® printer, allow the user to print half-height text and to completely cover the entire printed page, leaving no gaps between the printed pages. On the LaserWriter® printer, you can specify the amount to reduce the text size, as a percentage, whether you want smoothing (to smooth out jagged pixels in the picture being printed), and if font substitution should occur, in the event the text's font is not available in the Fonts folder of the

Design Master

System folder. For both printers, the Print Manager will set the appropriate bits in the wDev field of the style subrecord.

The user can select the orientation of the page, up and down or sideways, for either printer. The choices made are reflected in the appropriate bits in the wDev field of the style subrecord.

```
{ Cmds1.pas, line 345 }
( ***** )
*
* DoPSetUp - Handle Page setUp command.
*
***** )

procedure DoPSetUp;

var
    junk: boolean;

begin
    { Bring up Page Setup dialog; throw away result since Print Manager handles }
    { everything for us. }

    junk := PrStlDialog (printHandle);
    errNum := ToolError;
    if errNum <> 0 then
        HandleError (errNum, memoryErr, cautionAlertTyp);

end; {DoPSetUp}
```

The DoPrint Procedure

The DoPrint procedure is one of the more complicated subroutines in our program. The Print Manager is not terribly difficult to use, but it does require a lot of time reading the Print Manager's chapter in the *Toolbox Reference* manual, and quite a bit of patience until you succeed in correctly printing a document. As with most of the work we've done in MiniWord, the Text Edit tool set comes to our aid by providing a special call used just for printing.

DoPrint first brings up the Print Job dialog by calling the Print Manager's PrJobDialog function. We pass the call a handle to the print record, the variable prHandle, that we created and initialized in our InitCmds1 procedure. If PrJobDialog returns an error, we report the error and exit the DoPrint subroutine. The dialog returns true if the user wants to print the document, and false otherwise. It also fills in some of the fields in the print record, based on the user's selections.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds1.pas, line 69 }
prRect:      Rect;           {TEPaintText rect to draw into }
prStatus:    prStatusRec;    {reports info to user during spooled printing}

{ Cmds1.pas, line 162 }
(*****
*
* DoPrint - Handle Print command.
*
*****)

procedure DoPrint (* index: integer; currWindow: grafPortPtr *);

label 99;                                {error label}

const
    thruPrinting = $2209;                {err code returned by TEPaintText when}
                                         { starting line # exceeds last line #}

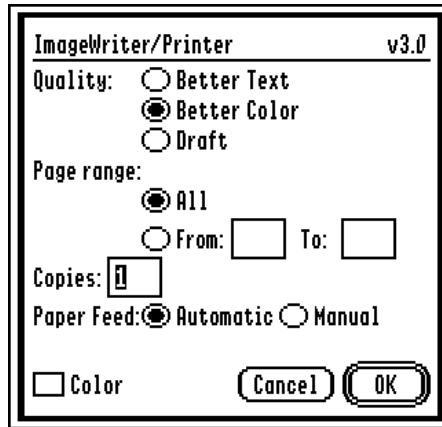
var
    prPort:      grafPortPtr;            {Print Manager's grafPort }
    currLine:    longint;                 {current line # to print }
    lastLine:    longint;                 {last line # to print }
    firstPage:   longint;                 {first page to begin printing }
    finalPage:   longint;                 {final page to print }
    copies:      integer;                 {# copies of document to print}
    spool:        boolean;                {false = draft mode; }
                                         {true = spooled printing }
    anError:     boolean;                 {true if error detected }
    printRecPtr: prRecPtr;                {pointer to print record }
    tmp:         longint;
    answer:      integer;

begin
    answer := PrJobDialog (printHandle);    {bring up Print Job dialog }
    errNum := ToolError;                    {error returned by PrJobDialog?}
    if errNum <> 0 then begin
        HandleError (errNum, memoryErr, cautionAlertTyp);
        goto 99;
    end;
    if answer = 0 then                        {want to print document?}
        goto 99;
```

We introduced the print record, our means of communicating with the Print Manager, in the last section, DoPSetUp. You may want to refer to the record while we discuss the fields set in it as a result of the choices entered in the Print Job dialog.

The Print Job dialog for the ImageWriter® printer is shown below:

Design Master



For the ImageWriter®, selecting one of the Quality buttons sets a bit in the wDev field of the style subrecord, and also fills in the bJDocLoop field in the job subrecord. If Draft mode is selected, bJDocLoop will be set to 0. If one of the Better mode buttons is selected, bJDocLoop will be set to 128.

The Page range items in the Print Job dialog allow the user to select which pages to print. The iFstPage and iLstPage of the job subrecord will be filled in with appropriate values, based on whether the user has requested all pages to be printed, or else has given a range of pages.

The edit-line box labeled Copies is used to specify the number of copies to print. This value is found in the print record in the iCopies field of the job subrecord.

The Paper Feed buttons allow the user to describe how paper is fed to the printer. The selection is reflected in the print record in the feed field of the style subrecord, either 0 for manual or 1 for automatic.

For the ImageWriter®, the user can select the Color check box. An appropriate bit will be set in the wDev field of the style subrecord.

The OK button is used to signal that printing is to begin, while the Cancel button is used to cancel the request for printing. The PrJobDialog routine returns true if OK was selected and false if Cancel was selected.

If the document is to be printed, we dereference the print handle to prepare for accessing five components of the print record.

The first field we obtain is the size of the rectangle in which to draw each page. We use the rPage field of the print record's style subrecord to set our prRect record. The style subrecord is set up by the Print Manager during the Page Setup dialog, when the user selects the Page setup command from the File menu. We looked at this dialog when we discussed the Page setup command, above.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds1.pas, line 200 }
printRecPtr := printHandle^;           {dereference print record handle}

{ Set up page rectangle based on printed page size calculated by Print Manager }
{ as derived from Job and Page setup dialogs.                                }

with printRecPtr^ do begin
  with prInfo.rPage do begin
    prRect.v1 := v1;
    prRect.h1 := h1;
    prRect.v2 := v2;
    prRect.h2 := h2;
  end;
end;
```

Next we get the `iFstPage` value from the print record, recording its value in our `firstPage` local variable, since we'll need this value later. We now need to translate the page number into a line count to pass to the Text Edit tool set. We first subtract 1 from the page number, since line numbers are counted by Text Edit starting at 0. We next multiply by 60, since we want 60 lines per page. You can change this constant to whatever value you prefer. The final beginning line number is stored into our local `currLine` variable. We change the `iFstPage` value to 1 in the print record, since the Print Manager counts the pages as it prints each one, beginning its count at 1.

```
{ Cmds1.pas, line 213 }
firstPage := prJob.iFstPage;           {get first page to print      }
currLine  := (firstPage - 1) * 60;     {calculate 1st line to print,}
                                           { counting lines from 0, and}
                                           { 60 lines per page        }
prJob.iFstPage := 1;                  {set page # to 1 for Print Manager, since it }
                                           { counts ea. page it prints, starting at 1 }
```

The next value we read from the print record is `iLstPage`, storing it into our local `finalPage` variable, since we'll need this value shortly. We need to calculate the ending line number to print, so we multiply `iLstPage` by 60, storing the result in our `lastLine` variable.

```
{ Cmds1.pas, line 220 }
finalPage := prJob.iLstPage;           {get last page to print      }
lastLine  := finalPage * 60;           {calculate last line to print}
```

Before we continue with `DoPrint`, we determine whether the user has given an ending page number that is less than the starting page number. We check the value by subtracting `firstPage` from `lastPage`, and exit `DoPrint` if the value is negative. If the value is positive or 0, we add 1 to the value and store the result into the `iLstPage` field of the print record, so that the last page to be printed corresponds with the first page to print, which we set to 1.

Design Master

```
{ Cmds1.pas, line 223 }
{ Ensure that starting page number not greater than ending page}

tmp := finalPage - firstPage;
if tmp < 0 then
    goto 99;

prJob.iLstPage := convert (tmp) .lsw + 1;      {reset last page to print for}
                                                { Print Mgr, relative to 1  }
```

Our next task is to obtain the iCopies value from the print record's job subrecord, storing it into our local copies variable. We'll need to control the printing of each copy requested if we're printing in draft mode. If we're not printing in draft mode, the Print Manager will handle producing multiple copies for us. At any rate, it takes as much time to figure out if we need the iCopies value as it does to simply obtain it, so we go ahead and get the value.

Another variable that we need to set, in case we must handle multiple copies, is firstPage. We give it currLine's value, which will change in the print loop, so that we can restore currLine to its original value when we begin printing the next copy.

```
{ Cmds1.pas, line 232 }
copies      := prJob.iCopies;      {get # copies to print      }
firstPage   := currLine;          {remember starting line # in case multiple}
                                      { copies wanted and printing in draft mode}
```

The last value that we obtain from the print record is bJDocLoop, contained in the job subrecord. As you may recall from our discussion of the Print Job dialog at the beginning of this section, bJDocLoop tells us whether we're printing in draft mode or spooling the print job. If the value is zero, we set our local boolean variable spool to false; if the value is not zero, we set spool to true.

```
{ Cmds1.pas, line 236 }
{ Determine whether printing in draft or spooled mode.}

if prJob.bJDocLoop = 0 then
    spool := false
else begin
    spool := true;
    copies := 1;          {PrPicFile handles mult. copies}
end;
```

The next check we need to make before continuing with our print routine is to determine whether the starting line number lies within the document. We can find the number of lines in the document by calling the Text Edit tool set's TEGetTextInfo routine, passing it a pointer to a textInfo record, an integer specifying the number of parameters in the textInfo record in which we're interested, and the handle of the applicable text edit control. The text info record contains several fields; the one we need is the second one, lineCount. For completeness, the text info record (which can be found in the tool interface files for your compiler) is shown below:

Chapter 3: Writing A Text Editor In A Few Hours

```
teInfoRec = record
    charCount:      longint;
    lineCount:      longint;
    formatMemory:   longint;
    totalMemory:    longint;
    styleCount:     longint;
    rulerCount:     longint;
end;
```

We subtract currLine from lineCount, and then check if the result is greater than or equal to 0. If it is, the starting line number lies within the document, and we proceed with DoPrint. If it is not, we exit the procedure.

```
{ Cmds1.pas, line 245 }
{ Ensure starting line # is in document by calling Text Edit's }
{ TEGetTextInfo to get # lines in document.                  }

TEGetTextInfo (textInfo, 2, textEdHandle [index]);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, getTextInfoErr, cautionAlertTyp);
    goto 99;
end;

if currLine > textInfo.lineCount then
    goto 99;

end;    {with printRecPtr}
```

Now that we've gotten everything set up, we're ready to enter our outer printing loop, which prints one copy of the document. The first call we make is to the Print Manager's PrOpenDoc routine, which opens the document for printing. We pass it the print record's handle and a value of nil; it returns a pointer to a grafPort in which to draw the text we want printed. The nil value indicates that we want the routine to allocate a new grafPort for us. If an error is detected by PrOpenDoc, we report the error, close the document for printing, and restore the document's grafPort. If there is no error, we continue on to the else clause, which prints the document.

Design Master

```
{ Cmds1.pas, line 261 }
{ Call Print Manager to open the document for printing; get Print Manager's }
{ printing grafPort. }

anError := false;

{ Outer print loop, to print multiple copies in draft mode. }

repeat

    prPort := PrOpenDoc (printHandle, nil);
    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, printErr, cautionAlertTyp);
        anError := true;
    end

    else begin
```

Next we enter our inner print loop, which is executed once for each page that we print. The first call to be made is to the Print Manager's PrOpenPage routine, which initializes the Print Manager's grafPort. We pass the routine the grafPort returned by PrOpenDoc and a value of nil that indicates we're not using a scaling rectangle for the page. If PrOpenPage returns an error, we report the error, close the printer's grafPort, restore the document's grafPort, and exit DoPrint.

```
{ Cmds1.pas, line 280 }
{ Inner print loop, to print each page in the document. }

repeat
    PrOpenPage (prPort, nil);      {init. grafPort, no scaling rect. passed}
    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, printErr, cautionAlertTyp);
        anError := true;
    end
```

The next two calls that we make are to Quick Draw II to initialize the drawing of the text into the printer's grafPort. PenNormal sets up a standard pen: its size is 1 pixel wide by 1 pixel high, the drawing mode is COPY, the pen color is black, and the pen mask is all ones. (Notice that the original pen state will be restored at the end of DoPrint, when we restore the document's grafPort.) The other Quick Draw II call we make is to MoveTo, which moves the pen to the pixel location given in the parameters to the call. We move to the upper left corner of the page by passing values of (0, 0).

```
{ Cmds1.pas, line 290 }
    else begin
        PenNormal;                {set pen to standard state }
        MoveTo (0, 0);            {move to top left corner of drawing rectangle}
                                   { drawing rectangle }
    end
```

Chapter 3: Writing A Text Editor In A Few Hours

In order to draw the text into the printer's grafPort, we call Text Edit's TEPaintText function, passing it a pointer to the printer's grafPort, the starting line number in the document to print, a pointer to the rectangle in which the text will be drawn, a flag of zero to indicate no special handling of the text, and the handle of the text edit control to which the call applies. TEPaintText returns the line number of the next line to print, or -1 if the end of the document was reached. We then check for errors returned by the call. If an error code of \$2209 is returned, we know that the starting line number exceeds the last line in the document, so that we are finished printing this copy of the document. If the error code is anything other than \$2209, we report the error and set the anError flag to drop out of the page-printing loop.

```
{ Cmds1.pas, line 294 }
    { Call TEPaintText to draw text into Print Manager's grafPort. }

    currLine := TEPaintText (prPort, currLine, prRect, 0,
                             textEdHandle [index]);
    errNum    := ToolError;
    if (errNum <> 0) and (errNum <> thruPrinting) then begin
        HandleError (errNum, printErr, cautionAlertTyp);
        anError := true;
    end;

    end; {no error from PrOpenPage}
```

The last tool call we need to make in our page-printing loop is to the Print Manager's PrClosePage routine. The routine causes internal clean-up tasks to be performed by the printer's driver and the Print Manager. We pass the call the a pointer to the printer's grafPort. If an error is returned by PrClosePage, we report the error and then close the document for printing.

```
{ Cmds1.pas, line 318 }
    PrClosePage (prPort); {close this printed page}
    errNum := ToolError;
    if (errNum <> 0) and (not anError) then begin
        HandleError (errNum, printErr, cautionAlertTyp);
        anError := true;
    end;
```

We've now reached the bottom of our page-printing loop. We check the value of our currLine variable, which received the line number to print next from the TEPaintText call. If the line number equals -1, we're finished printing this copy of the document. If the line number exceeds the value of the last line to print, contained in our lastLine variable, then we're also finished printing. If an error was detected, we stop printing. Otherwise, we go back to the top of this loop to print the next page of the document.

```
                                {end page-printing loop}
until (currLine = -1) or (currLine > lastLine) or (anError);
```

Design Master

After the document has been printed, we call the Print Manager's `PrCloseDoc` routine to deallocate the `grafPort` it created, passing it a pointer to the `grafPort`. If the call returns an error, we report the error, restore the document's `grafPort`, and exit `DoPrint`.

```
{ Cmds1.pas, line 318 }
PrCloseDoc (prPort);           {close document for printing}
errNum := ToolError;
if (errNum <> 0) and (not anError) then begin
  HandleError (errNum, printErr, cautionAlertTyp);
  anError := true;
end;
```

We've now reached the bottom of our copies-printing loop. We check to see if an error has occurred, or if all of the copies requested have been printed. If not, we start printing another copy.

```
{ Cmds1.pas, line 325 }
copies := copies - 1;           {one less copy to print}
currLine := firstPage;         {reset for next copy }

until (copies = 0) or (anError); {end print copies loop }
```

If we've spooled the print job, we begin the actual printing by calling the Print Manager's `PrPicFile` routine. (Printing takes place immediately if we're using draft mode.) We pass the routine the print record's handle, nil to indicate that we want it to allocate a new printer `grafPort` for us (recall that `PrCloseDoc` disposed of the other printer `grafPort`!), and a pointer to a printer status record. If the call returns an error, we report it. The printer status record is used to report on the status of the print job. If our program were more sophisticated, for example, we could use the record to tell the user which page we're currently printing.

```
{ Cmds1.pas, line 331 }
{ Handle spooled printing. }

if (spool) and (not anError) then begin
  PrPicFile (printHandle, nil, @prStatus); {let Print Mgr allocate new}
                                           { grafPort for printing }

  errNum := ToolError;
  if errNum <> 0 then
    HandleError (errNum, printErr, cautionAlertTyp);
end;
```

The last task that `DoPrint` performs is to restore the document's `grafPort`. We do this by calling the Quick Draw II routine `SetPort`, passing it a pointer to the `grafPort` contained in `currWindow`. Recall that the passed variable `currWindow` contains the active window's `grafPort` pointer.

```
{ Cmds1.pas, line 341 }
SetPort (currWindow);           {restore window's grafPort}
99:
end;
```

The Cmds2 Unit

The Cmds2 unit handles the menu commands New, Open, Save, and Save as. These are commands which interface with the file system.

The InitCmds2 Function

The Cmds2 unit has an initialization function named InitCmds2. It initializes the unit's global data structures, and incorporates the initialization code generated by Design Master for the document window and its text edit control.

InitCmds2 first initializes its untitledNum variable, an integer that holds the number to assign to the next untitled window opened on the desktop.

Its next action is to allocate some memory for a buffer for the Text Edit tool set. When we call Text Edit to obtain a window's text to save to disk, we pass it the handle of a buffer in which to store the text. (A handle is a pointer to a pointer. The value of the handle never changes, but the pointer can change if we set the memory attributes flag to allow the block of memory to be moved. Whenever possible, you should allow your data blocks to be moved. This lets the Memory Manager maintain the machine's memory more efficiently. Memory management on the Apple IIGS is discussed in detail at the beginning of chapter 12 of volume 1 of the *Apple IIGS Toolbox Reference* manual.)

```
{ Globals.pas, line 53 }
  buffer:      handle;           {buffer to hold text returned by TEGetText}
  bufferSize: longint;          {size of this buffer }

{ Cmds2.pas, line 71 }
  untitledNum: integer;          {# to assign to next untitled window}

{ Cmds2.pas, line 678 }
( *****
*
* InitCmds2 - Initialize the Cmds2 unit's data
*             structures.
*
***** )

function InitCmds2 (* : boolean *);

label 99;

begin
  InitCmds2 := true;
  untitledNum := 1;              {# to assign to next untitled window }
```

Design Master

```
{ Allocate memory block for TEGetText call. }

buffer := NewHandle (1024, myID, $0000, nil);           {don't purge, can move}
errNum := ToolError;
if errNum <> 0 then begin
  HandleError (errNum, memoryErr, stopAlertTyp);
  InitCmds2 := false;
  goto 99;
end;
bufferSize := 1024;
```

InitCmds2 needs to initialize all of the data structures used to communicate with the Standard File Operations tool set. These data structures include a reply record, whose fields are filled in by SFO when we make SFPutFile2 and SFGetFile2 calls; a type list record that SFO uses to filter the files that can be opened; and prompts that we want SFO to use in its standard open-file and save-file dialogs.

The reply record has been revised with GS/OS 5.0. You can find the definition for the reply record in the tool interface files for your compiler. Its current form is:

```
replyRecord5_0 = record
  good:           integer;
  fileType:       integer;
  auxFileType:    longint;
  nameVerb:       integer;
  nameRef:        longint;
  pathVerb:       integer;
  pathRef:        longint;
end;
```

The only fields we need to set in this record are the verbs. They are set to the newHandleVerb constant, since this is the recommended verb, as described in the *Toolbox Update Notes* on page 24-3. The newHandleVerb tells SFO to take care of allocating the handles for the file name and path name of the file we wish to open or save. The handles it returns are for GS/OS class 1 output strings. (The newHandleVerb has a value of 3; it is defined in the Common interface file for ORCA/Pascal.)

For those of you not familiar with GS/OS strings, an explanation is in order. GS/OS uses two types of strings: input and output. An input string starts with an integer that gives the length of the string; the integer is followed by the ASCII characters that comprise the string. Input strings are used when you supply a string to GS/OS. An output string begins with an integer that specifies the total length of the buffer, followed by an integer that gives the length of the string, and ends with the ASCII characters that make up the string. For example, an output string of 'AB' might have a total buffer length of 6, two bytes for the buffer, two bytes for the length of the string, and two bytes for the two characters. Output strings are used by GS/OS to return strings to the programmer. These types of strings are termed "class 1" to distinguish them from the Pascal-style strings used by ProDOS 16, called "class 0" strings.

The type list record has also been updated for GS/OS 5.0. It is documented on page 24-7 of the *Toolbox Update Notes*. Its form is:

Chapter 3: Writing A Text Editor In A Few Hours

```
typeRec = record
    flags:      integer;
    fileType:   integer;
    auxType:    longint;
end;

typeList5_0 = record
    numEntries:      integer;
    fileAndAuxTypes: array [1..10] of typeRec;      (* change array size *)
end;                                           (* as needed *)
```

We set the flags field of the typeRec record to indicate that we're specifying the types of files that the user can open; the two file types we'll allow include ASCII files and APW/ORCA source files, which also contain only ASCII data.

```
{ Cmds2.pas, line 77 }
                                {Standard File Operations data structures}

theReply: replyRecord5_0;
openMsg:  packed array [0..20] of char;    {msg to display in open file dlg}
openTypes: typeList5_0;
saveMsg:  packed array [0..27] of char;    {msg to display in save file dlg}
saveName: smGSOSinString;                 {default filename to appear in }
                                           { save file dialog          }

{ Cmds2.pas, line 704 }
{ Initialize SFO variables. }

with theReply do begin
    nameVerb := newHandleVerb;             {reference is undefined: SFO will}
    pathVerb := newHandleVerb;             { allocate and return a handle }
end;

openMsg := 'Select file to open: ';
with openTypes do begin
    numEntries := 2;                       {number of entries in filetype list}
    fileAndAuxTypes [1].flags := $8000;    {match filetype, any auxtype, allow}
                                           { user to select these files      }

    fileAndAuxTypes [1].fileType := $04;   {ASCII text file                }
    fileAndAuxTypes [1].auxType := 0;      {don't care about auxtype       }
    fileAndAuxTypes [2].flags := $8000;    {match filetype, any auxtype, allow}
                                           { user to select these files      }

    fileAndAuxTypes [2].fileType := $B0;   {APW source file                }
    fileAndAuxTypes [2].auxType := 0;      {don't care about auxtype       }
end;

saveMsg := 'Enter name of file to save: ';
with saveName do begin
    currSize := 7;
    theText := 'newFile';                  {default name to appear SFO save dlg}
end;
```

Design Master

The next set of variables that must be initialized are our GS/OS parameter blocks. These are records that we pass to GS/OS when we make file-access calls, such as to open, read, and write disk files. The records are described in volume 1 of the *GS/OS Reference* manual. Notice that each parameter block begins with an integer, the pCount field, that states the number of parameters that will be defined in the record. See the GS/OS manual for further information about each parameter block.

```
{ Cmds2.pas, line 46 }
type
  gsosMinOutputBuffer = record           {minimally sized GS/OS output buffer}
    totalSize: integer;
    currSize: integer;
    theText:   packed array [1..2] of char;
  end;

  smGSOSinString = record                {small GS/OS input string}
    currSize: integer;
    theText:   packed array [1..10] of char;
  end;

{ Cmds2.pas, line 86 }
                                {Data structures for GS/OS file handling calls}

  openRec:      openOSDCB;                {open file parameter block  }
  readRec:      readWriteOSDCB;           {read file parameter block  }
  writeRec:     readWriteOSDCB;           {write file parameter block }
  closeRec:     closeOSDCB;               {close file parameter block }
  destroyRec:   destroyOSDCB;             {delete file parameter block}
  getFileInfoRec: getFileInfoOSDCB;       {getFileInfo parameter block}
  createRec:    createOSDCB;              {create file parameter block}
  options:      gsosMinOutputBuffer;     {buffer for FST info, returned by GS/OS}

{ Cmds2.pas, line 730 }
{ Initialize GS/OS variables. }

with options do begin
  totalSize := 6;
  currSize  := 2;
end;

with openRec do begin
  pCount      := 15;                {parameter count          }
  access      := $0003;             {request read/write access }
  resourceNumber := 0;              {resource number: open data fork}
  optionList   := @options;         {pointer to GS/OS result buffer }
end;

with readRec do begin
  pCount      := 5;                {parameter count          }
  cachePriority := 0;              {don't cache file         }
end;
```


Chapter 3: Writing A Text Editor In A Few Hours

```

closeRec.pCount      := 1;                                {parameter count}
destroyRec.pCount    := 1;

with writeRec do begin
  pCount              := 5;                                {parameter count }
  cachePriority       := 0;                                {don't cache file}
end;

with getFileInfoRec do begin
  pCount              := 12;                                {parameter count      }
  optionList          := @options;                          {pointer to GS/OS output buffer}
end;

with createRec do begin
  pCount              := 7;                                {parameter count      }
  access              := $00C3;                            {destroy, rename, write, read access}
  fileType            := $0004;                            {ASCII file           }
  auxType             := 0;
  storageType         := $0001;                            {standard file        }
  dataEOF             := 0;                                {initial size of data fork is 0    }
  resourceEOF         := 0;                                {initial size of resource fork is 0}
end;

```

The final set of variables that need to be initialized have been created by Design Master. If you are working through the examples, you can paste over the variables at the top of the `cmds2.pas` file with your definitions, and paste over the following initialization code with the code Design Master generated in Chapter 2.

```

{ Cmds2.pas, line 58 }
var
  { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }

  windColorTable: wColorTbl;                                {document window definitions}
  window:        paramList;
  control001111: editTextControl;

{ Cmds2.pas, line 773 }
{ *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
{ Initialize window parameter list and editText control record. }

with window do begin
  paramLength := 78;                                {Window parameter list      }
  wFrameBits  := $C1E7;                              {parm list length          }
  wTitle      := nil;                                {frame bits                 }
  wRefCon     := 0;                                  {pointer to window's title  }
  with wZoom do begin
    v1 := $001E;                                       {we'll use to store index into }
    h1 := $0014;                                       { window-tracking arrays      }
    v2 := $003C;                                       { zoomed rectangle           }
    h2 := $0064;
  end;
end;

```

Design Master

```

wColor      := @windColorTable;    {color table pointer      }
wYOrigin    := $0000;              {vert offset of content   }
wXOrigin    := $0000;              {horiz offset of content  }
wDataH      := $0000;              {data area height        }
wDataW      := $0000;              {data area width         }
wMaxH       := $00C8;              {max grow height         }
wMaxW       := $0280;              {max grow width          }
wScrollVer  := $0000;              {vert. arrow scroll amount}
wScrollHor  := $0000;              {horiz arrow scroll amount}
wPageVer    := $0000;              {vert. page amount       }
wPageHor    := $0000;              {horiz page amount       }
wInfoRefCon := 0;                  {info bar ref con        }
wInfoHeight := 0;                  {info bar height         }
wFrameDefProc := nil;              {window definition procedure}
wInfoDefProc := nil;              {info bar draw routine   }
wContDefProc := nil;              {window content draw routine}
with wPosition do begin           {window position rectangle}
    v1 := $001E;
    h1 := $0014;
    v2 := $00C2;
    h2 := $025F;
end;

wPlane      := grafPortPtr (-1);   {window plane, -1 for front}
wStorage     := nil;                {address of memory for window record}
end;

with windColorTable do begin
    frameColor := $0010;             {window frame color}
    titleColor := $0D81;             {title color}
    tBarColor  := $021D;             {title bar color}
    growColor  := $0074;             {grow box color}
    infoColor  := $0000;             {info bar color}
end;

with control001111 do begin        {editText control template}
    pCount     := 23;                {parameter count}
    controlID  := $000087DA;         {ID number TaskMaster will use}
    with boundsRect do begin        {encloses entire control, including}
        v1 := $0002;                { scroll bars and grow box}
        h1 := $0002;
        v2 := $0000;
        h2 := $0000;
    end;
    procRef    := $85000000;         {Code for editText control}
    flags      := $0000;             {control is visible, not dirty yet}
    moreFlags  := $7C00;             {add a grow box to the control}
    refCon     := 0;
    textFlags  := $62200000;         {smart cut & paste; allow editing}
    with indentRect do begin        {use standard indentation}
        v1 := $FFFF;
        h1 := $FFFF;
        v2 := $FFFF;
        h2 := $FFFF;
    end;
end;

```

Chapter 3: Writing A Text Editor In A Few Hours

```
vertBar      := $FFFFFFF;      {create vert. scroll bar just inside}
                                { right edge of bounds rect      }
vertAmount   := 0;             {scroll 9 pixels, up- & down-arrows }
horzBar      := 0;             {MUST BE NULL version 1.0      }
horzAmount   := 0;             {MUST BE NULL version 1.0      }
styleRef     := 0;             {use default style and ruler    }
textDescriptor := 0;           {no initial text for control    }
textRef      := 0;
textLength   := 0;
maxChars     := 0;
maxLines     := 0;
maxCharsPerLine := 0;          {MUST BE NULL version 1.0      }
maxHeight    := 0;            {MUST BE NULL version 1.0      }
colorRef     := 0;            {no color table                  }
drawMode     := 0;
filterProcPtr := nil;         {use built-in filter procedures}
end;

99:
end; {InitCmds2}
```

The DoNew Procedure

The DoNew procedure is called when the user selects the New command from the File menu to create a new window on the desktop. DoNew calls four other subroutines to initialize the window's data structures and create the window.

```
{ Cmds2.pas, line 162 }
(*****
*
* DoNew - Handle New command.
*
*****)

procedure DoNew (* var index: integer; var numWindows: integer *);

begin
  InitWindow (window, index);          {init. window data structures }
  GetUntitledName (index, untitledNum); {create untitled window's name}
  if CreateWindow (index, numWindows) then begin {create new window on desktop }
    windowOpen [index] := fromNew;      {set flag that window from the New cmd}
    FiniWindow (numWindows);            {disable Open, New cmds, if necessary }
  end;
end; {DoNew}
```

The InitWindow Procedure

The first subroutine called when the program needs to create a document window is InitWindow, which initializes the data structures for the new window. Its first task is to find an empty slot in the window-tracking arrays. This is accomplished by searching through the windowOpen array until a noWindow enumerated constant is found. The windowOpen array

Design Master

contains a flag for each of our windows. If an array element contains a noWindow constant, then no window is open. If it contains a fromNew constant, then the window is a new window and not associated with a disk file. If an element contains a fromFile constant, then the window is tied to a disk file. Once InitWindow finds an open slot, it sets the passed variable index to the index of the open slot it found. We'll use this index to record other information about the window in the other window-tracking arrays.

InitWindow's last task is to set the window title's pointer to the correct wName element, using the index calculated.

```
{ Globals.pas, line 42 }
pString17 = packed array [0..17] of char;

{ Cmds2.pas, line 72 }
wName:      array [0..3] of pString17;          {array of window names      }

{ Cmds2.pas, line 863 }
(*****
*
* InitWindow - Perform window initialization,
*              common to both DoOpen and DoNew.
*
*****)

procedure InitWindow (* var window: paramList; var index: integer *);
begin
  { Find empty slot in windowOpen array for this window. }

  index := 0;
  while windowOpen [index] <> noWindow do
    index := index + 1;

  { Use window's RefCon to track position in window arrays. }

  window.wRefCon := index;

  { Set window's title pointer. }
  window.wTitle := @wName [index];
end;
```

The GetUntitledName Procedure

DoNew next calls the procedure GetUntitledName to create the next "Untitled" name for the new window. The untitledNum variable is used to assign the next number to the new untitled window. The integer needs to be converted to an ASCII string before we can use it in the window's title, so this is GetUntitledName's first job. We call the built-in ORCA/Pascal function cnvis (convert integer to string) to perform the conversion. We pass cnvis the integer to be converted to a Pascal-style string, and it returns a string containing the character representation of the number. The final "Untitled" name is built by concatenating the string constant 'Untitled ' with the number, using the built-in ORCA/Pascal concat function.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds2.pas, line 653 }
(*****
*
* GetUntitledName - Create next untitled window's
*                  name.
*
*****)

procedure GetUntitledName (* index: integer; var untitledNum: integer *);

var
    num : packed array [0..5] of char;

begin
    { Convert untitled window number to a Pascal-style string, then increment }
    { untitled number. }

    num      := cnvis (untitledNum);
    untitledNum := untitledNum + 1;

    { Move character representation of untitled number into the window's title. }

    window.wTitle^ := concat (' Untitled ', num, ' ');

end; {GetUntitledName}
```

The CreateWindow Function

The heart of the window creation routines is CreateWindow, which calls the Window Manager and Control Manager to create a new window containing a text edit control on the desktop.

The window is created by calling the Window Manager's NewWindow routine. We pass it a pointer to the window parameter record, originally created by Design Master, and altered by us for a new window title. If NewWindow is unable to create the window, probably because of lack of memory, we report the error and return false to the calling procedure to indicate that a new window could not be created. If the window was created, we record the pointer to the window's grafPort in the windowPtr array.

If NewWindow was successful in creating the window, we next call the Control Manager's NewControl2 routine to add a text edit control to the window. NewControl2 is a new call added to GS/OS version 5.0 to allow controls to be defined using resource forks. Since we're using source code to define the control, we pass NewControl2 a verb of pointer, and then the address of the control record created by Design Master. If NewControl2 is unable to create the control for the window, we report the error, close the window, and return false to the calling procedure. If NewControl2 was successful, we record the handle of the text edit control returned by the routine in the textEdHandle array, and then return true to the calling procedure to indicate that the new window was created on the desktop.

Design Master

```
{ Globals.pas, line 65 }
windowPtr:   array [0..3] of grafPortPtr;   {array of pointers to grafPorts}
textEdHandle: array [0..3] of ctlRecHndl;   {array of editText ctl handles }

{ Cmds2.pas, line 120 }
(*****
*
* CreateWindow - Create new window on the desktop.
*
* Output:
*     true if window was created; false otherwise
*
*****)

function CreateWindow (var index: integer; var numWindows: integer): boolean;

var
    tmp: longint;

begin
    CreateWindow := true;                {assume we'll be successful}

    { Call Window Manager's NewWindow function to create new window on desktop. }

    windowPtr [index] := NewWindow (window);
    errNum           := ToolError;
    if (errNum <> 0) then begin
        HandleError (errNum, memoryErr, cautionAlertTyp);
        CreateWindow := false;
    end

    { Add text edit control to window. }

else begin
    tmp := ord4 (@control001111);
    textEdHandle [index] := NewControl2 (windowPtr [index], pointerVerb, tmp);
    errNum := ToolError;
    if (errNum <> 0) then begin
        HandleError (errNum, memoryErr, cautionAlertTyp);
        DoClose (index, windowPtr [index], numWindows);
        CreateWindow := false;
    end;
end;

end; {CreateWindow}
```

The FiniWindow Procedure

The final routine called by DoNew is FiniWindow, which first increments the number of windows now open on the desktop, and then disables the New and Open commands in the File menu if this brings the number of windows open to four. Menu items are disabled by calling the Menu Manager's DisableMItem routine, passing it the item number of the item to be disabled.

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Cmds2.pas, line 518 }
(*****
*
* FiniWindow - Disable New and Open commands if
*             this is the 4th window opened.
*
*****)

procedure FiniWindow (* var numWindows: integer *);

begin
numWindows := numWindows + 1;           {one more open window    }
if numWindows = maxWindows then begin   {max # open windows reached?}
    DisableMItem (openID);              {yes - don't let 'em open or}
    DisableMItem (newID);               { create another window    }
end;

end; {FiniWindow}
```

The DoOpen Procedure

Our other window creation routine is DoOpen. It is called when the user selects Open from the File menu. It opens a disk file, creates a new window, and then paints the file's contents into the new window. It calls the InitWindow, CreateWindow, and FiniWindow routines, also used by the DoNew routine, but it must do quite a bit of additional work to access and display the file's contents.

As with the DoNew procedure, DoOpen first calls InitWindow to initialize the new window's data structures.

It then calls the Standard File Operations tool set's SFGetFile2 routine to obtain the pathname of the disk file to open. The SFGetFile2 call is new with GS/OS 5.0. It works like the old SFGetFile call, but it returns GS/OS class 1 strings, rather than Pascal-style strings, for easy interface with GS/OS calls. The parameters that we pass to the call are very important to understanding how the DoOpen subroutine functions, so we'll spend some time discussing each.

```
{ Cmds2.pas, line 179 }
(*****
*
* DoOpen - Handle Open command.
*
*****)

procedure DoOpen (* var index: integer; var numWindows: integer *);

label 99;

var
    aHandle:   handle;
    tmp: handle;
```

Design Master

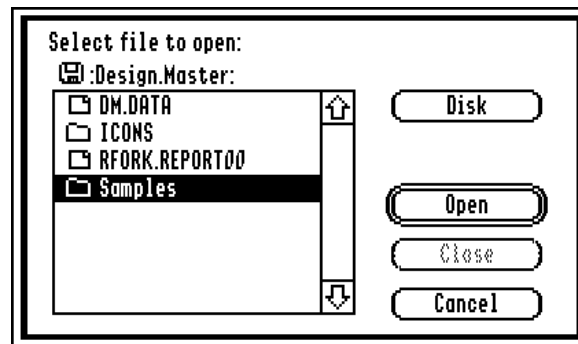
```
begin
InitWindow (window, index); {initialize window data structures}

{ Make SFGetFile2 call to bring up SFO Open dialog and get filename and }
{ pathname of file to open. }

SFGetFile2 (20, {upper left corner X-coord of SFGetFile2's dialog}
            20, {upper left corner Y-coord of SFGetFile2's dialog}
            pointerVerb, {prompt is pointer to P-string }
            @openMsg, {pointer to prompt }
            nil, {no filter procedure }
            openTypes, {fileTypes, auxTypes of files to open}
            theReply); {GS/OS 5.0 reply record }

errNum := ToolError;
if errNum <> 0 then
    HandleError (errNum, getFileErr, cautionAlertTyp)
```

The SFGetFile2 routine brings up a dialog box that is used to get the pathname of the file to open. The dialog is shown below:



The first two parameters that we pass to the call give the coordinates of the upper left corner of the dialog. The parameters are pixel locations, using global coordinates. The first parameter is the x-coordinate; that is, the number of pixels from the left side of the desktop. The second parameter is the y-coordinate – the number of pixels from the top of the screen. We use values of 20 and 20, which places the dialog directly below the menu bar.

The next parameter we pass SFGetFile2 is a verb describing the type of prompt we'll send to the routine. We're going to pass a pointer to the prompt string, so we use a verb of pointer. As with all of the new calls that accept verbs, the parameter following the verb is of the type the verb describes. For our program, this is a pointer to the Pascal-style string "Select file to open:"

SFGetFile2's fifth parameter is a pointer to a filter routine that will determine what type of files can be opened. We're going to specify the type of files that can be accessed in the next parameter, so we'll just use the standard filter procedure provided by SFGetFile2. We communicate this to SFGetFile2 by passing a nil pointer to the routine.

Chapter 3: Writing A Text Editor In A Few Hours

The next parameter that we pass is a pointer to a typeList record, which tells SFGetFile2 the kind of files to display and which ones can be opened. The typeList record was discussed in detail in the InitCmds2 function, above.

The last parameter that we pass to SFGetFile2 is a pointer to a reply record. The reply record was also covered in the InitCmds2 function, above.

After making the SFGetFile2 call, and checking for errors returned by the tool, we check the good field of the reply record to see if the user has chosen to open a file. If good is true, then the pathname returned by the call must be put into a form that we can pass to GS/OS to open the file. First, the handle is dereferenced. Next, we add 2 to the pointer to skip the buffer length integer at the beginning of the reply record's pathname string, since the GS/OS open call uses an input string.

DoOpen's next task is to open the file. We call GS/OS to open the file, passing it a pointer to an open file parameter block. The fields that we need to fill in are described in the InitCmds2 function, above.

If the file is successfully opened, we store the file's pathname in the pathName array, since we'll need this pathname later if the user wants to save the file to disk after making changes to it. If the file cannot be opened, we report the error returned by GS/OS and exit the DoOpen routine.

```
{ Cmds2.pas, line 211 }
else if theReply.good <> 0 then begin           {Does user want to open a file?}
                                                {Yes - get pointer to pathname }
    tmp                := handle (theReply.pathRef);
    openRec.pathname   := pointer (ord4 (tmp^) + 2);
    GSOSOpen (openRec);                          {open the file}
    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, openErr, cautionAlertTyp);
        goto 99;
    end;
```

DoOpen next calls the GetOpenName procedure to set the window's title to the filename of the file we just opened. We'll look at GetOpenName after we finish examining the DoOpen procedure.

```
{ Cmds2.pas, line 222 }
{ Record file's pathname - we'll need it when saving file to disk.  }
{ Pass file's reference # to GS/OS read and close parameter blocks. }

thePath [index] := gsosPathNamePtr (openRec.pathname);
readRec.refNum  := openRec.refNum;
closeRec.refNum := openRec.refNum;

GetOpenName (theReply, window); {get filename to display in window's title}
```

DoOpen next calls the CreateWindow routine to create a new window on the desktop. We looked at CreateWindow when we discussed the DoNew routine, above. If CreateWindow detects an error, we close the file and exit the DoOpen routine.

Design Master

```
{ Cmds2.pas, line 231 }
{ Create new window - if unable to create, close the file & exit. }

if not (CreateWindow (index, numWindows)) then begin
    GSOSClose (closeRec);
    goto 99;
end;
```

If the window was successfully created, we allocate a block of memory into which the disk file is read. We allocate memory from the Memory Manager with the NewHandle call, using a memory attributes flag of locked, don't purge, and don't cross a bank boundary. The amount of memory that we need was returned by the open call, in the dataEOF field of the open-file parameter block. If the memory cannot be allocated, we report the error, close the file, close the window, and exit DoOpen.

```
{ Cmds2.pas, line 238 }
{ Read the file into memory, pass its text on to TextEdit to paint into }
{ window. }
{ Allocate memory block to read file into. Attributes flag is locked, }
{ can't move, don't purge, don't cross bank boundary. }

aHandle := NewHandle (openRec.dataEOF, myID, $C010, nil);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, memoryErr, cautionAlertTyp);
    GSOSClose (closeRec);
    DoClose (index, windowPtr [index], numWindows);
    goto 99;
end;
```

If we were able to obtain the required memory, we dereference the handle returned by the Memory Manager, placing the pointer in the GS/OS read parameter block's dataBuffer field. We then issue a GS/OS read call, checking for errors. If we unable to read the file into memory, we report the error, close the file, close the window, and exit DoOpen.

```
{ Cmds2.pas, line 252 }
readRec.dataBuffer := aHandle^;           {dereference memory handle}
readRec.requestCount := openRec.dataEOF;   {get # bytes to read }
GSOSRead (readRec);
errNum := ToolError;

{ If unable to read the file, report the error, close the window, close the }
{ file. }

if errNum <> 0 then begin
    HandleError (errNum, readErr, cautionAlertTyp);
    DoClose (index, windowPtr [index], numWindows);
    GSOSClose (closeRec);
    goto 99;
end;
```

Chapter 3: Writing A Text Editor In A Few Hours

In order to get the file's contents into the window, we call the Text Edit tool set's `TESetText` routine, passing it a text descriptor field, a pointer to the read buffer, the amount of text contained in the buffer, a verb describing the style information we'll pass, and nil to indicate that no style information will be passed. The text descriptor is a bit flag that describes the form of the text sent. We use a text descriptor of \$0005 to indicate that the text is "raw data." If `TESetText` returns an error, we report the error, close the window, close the file, and exit `DoOpen`.

```
{ Cmds2.pas, line 267 }
{ Call Text Edit to transfer the text from the read buffer to the control. }

TESetText ($0005,                                {text is raw data      }
           ord4 (readRec.dataBuffer),              {pointer to data      }
           readRec.transferCount,                  {size of data         }
           pointerVerb,                            {style ref is pointer }
           0,                                       {not passing style info}
           nil);                                   {use active TE control }

errNum := ToolError;
if errNum <> 0 then begin
  { Only possible errs are messing up parms or propagation of memory errs }

  HandleError (errNum, fatalErr, stopAlertTyp);
  GSOSClose (closeRec);
  DoClose (index, windowPtr [index], numWindows);
  goto 99;
end;
```

At this point, all we need to do is some clean up, and we're finished opening the new window. We first record in the `openWindow` array that the window comes from a disk file. We then call `FiniWindow` to disable the Open and New commands in the File menu if this is the fourth window opened. Next we call `GS/OS` to close the file. Finally, we dispose of the read buffer, since Text Edit uses its own buffers to hold the file's contents.

```
{ Cmds2.pas, line 285 }
{ Perform clean-up: record that window is from a disk file, check if need }
{ to disable Open & New commands, close the file, release read buffer's   }
{ memory.                                                                    }

windowOpen [index] := fromFile;
FiniWindow (numWindows);
GSOSClose (closeRec);
DisposeHandle (aHandle);

end; {if user wants to open file}

99:
end; {DoOpen}
```

The GetOpenName Procedure

DoOpen calls the procedure GetOpenName to set the new window's title to the filename of the file opened. The SFGetFile2 call returns a handle to the filename in the nameRef field of the reply record.

GetOpenName first dereferences the filename handle, adding two to the resulting pointer to move beyond the buffer's total length field. The pointer now points to the integer preceding the filename string, which gives the length of the string. Using the pointer, GetOpenName obtains the length of the string, and then ANDs this value with \$000F in order to restrict the filename to 15 characters, the maximum number that can be stored into the wName array. After accessing the filename's length, the pointer is incremented by one to move beyond the first byte of the string's length field. The pointer now points to the byte before the string containing the filename. The altered size byte is stuffed into the byte we're pointing to, transforming the GS/OS input string into a Pascal-style string. The final window title is formed by concatenating the filename with surrounding blanks.

OpenFileName's last task is to dispose of the filename handle returned by SFGetFile2, since we no longer need it.

```
{ Cmds2.pas, line 536 }
(*****
*
* GetOpenName - Build filename when open a file.
*
*****)

procedure GetOpenName (* var theReply: replyRecord5_0; window: paramList *);
{ Will dispose of filename's handle; will change window's title. }

var
  namePtr: ptr;
  size:   byte;
  tmp:    handle;
  tmpPtr: pStringPtr;

begin
  { Use filename returned by SFO to place into window's title string. }
  { First need to dereference filename handle, then move beyond GS/OS }
  { total buffer length. }

  tmp      := handle (theReply.nameRef);
  namePtr  := pointer (ord4 (tmp^) + 2);
```

Chapter 3: Writing A Text Editor In A Few Hours

```
{ Get length of filename; AND with $000F to restrict length to 15 characters. }
{ Set filename pointer to point to 2nd byte of length word, to prepare for    }
{ converting a GS/OS input string into a Pascal-style string. Stuff the      }
{ length byte into the byte pointed to by namePtr to complete the conversion. }

size := namePtr^ & $0F;
namePtr := pointer (ord4 (namePtr) + 1);
namePtr^ := size;

{ Move filename into area pointed to by wTitle, surrounding it with blanks. }

tmpPtr      := pStringPtr (namePtr);
window.wTitle^ := concat (' ', tmpPtr^, ' ');

{ Dispose of filename handle since we won't need it after this. }

DisposeHandle (tmp);

end; {GetOpenName}
```

The DoSave Procedure

The next major subroutine we'll look at is DoSave, called when the user selects the Save command from the File menu.

DoSave first checks whether the window is associated with a disk file. If the window was created by the New command, and has not yet been saved to disk, the subroutine calls the DoSaveAs procedure, and exits.

The next action to be taken by DoSave is to determine if the file has changed since it was last saved to disk. If the file has not been changed, there's no reason to write it to disk, and the procedure exits. In order to determine whether the file has changed, the text edit control's dirty bit needs to be checked. If the bit is set, then the file has changed. The bit is bit 6 of the ctrlFlag field of the text edit record; the DoClose procedure, described earlier in this chapter, provides a more thorough discussion of the dirty bit.

```
{ Cmds2.pas, line 299 }
(*****
*
* DoSave - Handle Save command.
*
*****)

procedure DoSave (* index: integer, currWindow: grafPortPtr *);

label 99;

var
    tePtr: teRecPtr;           {ptr to active window's textEdit control record}
```

Design Master

```
begin
{ Check if window from the New command; if so, execute DoSaveAs. }

if windowOpen [index] = fromNew then begin
  DoSaveAs (index, currWindow);
  goto 99;
end;

{ Check if file has changed since it was last saved to disk. Check if the }
{ dirty bit in the textEdit record has been set by Text Edit.           }

tePtr := teRecPtr (textEdHandle [index]^);           {dereference teHandle}
if (tePtr^.ctrlFlag & isDirty) = 0 then
  goto 99;
```

We now need to obtain the text from the Text Edit tool set to write to disk. We do this in the `GetText` function, which we call next. We'll look at `GetText` in detail after we finish discussing `DoSave`.

If `GetText` was able to access the text, it returns the value `true`. In this case, it will also have placed the text in a memory buffer, whose handle is given in the variable named `buffer`. If `GetText` was unable to obtain the text, it returns `false`. In this case, we exit the `DoSave` procedure.

```
{ Cmds2.pas, line 327 }
{ Get the window's text to write to disk. }

if not (GetText (index, textInfo, writeRec.requestCount)) then
  goto 99;
```

At this point, we're now ready to write the file to disk. We get the file's path name from the `thePath` array, storing the path name to the `destroy`, `create`, and `open` parameter blocks to prepare for calling `GS/OS`. We delete the old file by making a `GS/OS` `destroy` call. If an error occurs, we report the error and exit `DoSave`. After deleting the old file, we create a new file with the same name by making a `GS/OS` `create` call. Notice the values we use in the `create` parameter block: the file's access flags allow the file to be destroyed, renamed, written, and read; its file type is an ASCII text file; its `auxType` is zero; its storage type is 1, for a standard file; its initial size is zero; and its initial resource fork size is zero. If we cannot create the new file, we report the error and exit the `DoSave` procedure. Once we've created the new file, we call `GS/OS` to open the file for writing. If we're unable to open the file, we report the error and exit `DoSave`. If we successfully opened the file, we store the file reference number assigned by `GS/OS` to the `write` and `close` parameter blocks.

```
{ Cmds2.pas, line 332 }
{ Get file's pathname for delete, create, and open operations. }

destroyRec.pathname := thePath [index];
createRec.pathname  := thePath [index];
openRec.pathname    := thePath [index];
```

Chapter 3: Writing A Text Editor In A Few Hours

```
GSOSDestroy (destroyRec);                                {delete the old disk file}
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, deleteErr, cautionAlertTyp);
    goto 99;
end;

GSOSCreate (createRec);                                  {create new file}
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, createErr, cautionAlertTyp);
    goto 99;
end;

GSOSOpen (openRec);                                     {open the new file}
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, openErr, cautionAlertTyp);
    goto 99;
end;
```

We now need to dereference the buffer handle of the text, since the GS/OS write routine requires a pointer to the data. We also need to store the size of the text to write into the write parameter block. After all of the appropriate fields have been initialized, we make the GS/OS write call to write the text to disk. If the write call fails, we report the error, close the file, and exit DoSave. We finally call GS/OS to close the disk file.

```
{ Cmds2.pas, line 359 }
{ Dereference buffer's handle to get pointer to data to write. }

writeRec.dataBuffer := buffer^;

{ Get file's reference # for write and close operations. }

writeRec.refNum := openRec.refNum;
closeRec.refNum := openRec.refNum;

GSOSWrite (writeRec);                                    {write window to disk}
errNum := ToolError;
if errNum <> 0 then
    HandleError (errNum, writeErr, cautionAlertTyp)
```

The final detail we need to handle in DoSave is to clear the dirty bit. We AND the ctrlFlag field with \$FFBF, which preserves all bits but the dirty bit. We then store the ctrlFlag back to the text edit record.

```
{ Cmds2.pas, line 373 }
{ Clear the dirty bit after saving file. }
else
    tePtr^.ctrlFlag := tePtr^.ctrlFlag & notDirty;
```

Design Master

```
GSOSClose (closeRec);                                {close the disk file}

99:
end; {DoSave}
```

The GetText Function

We'll now look at one of the subroutines called by DoSave, GetText.

The GetText subroutine is used to obtain a window's text in order to write the text to disk. The function returns true if it was able to get the text, and false otherwise.

GetText's first action is to call the Text Edit tool set's TEGetTextInfo procedure to find out the size of the text. The text info record contains several different fields; the one we need is the first one, charCount, which gives the number of characters in the specified text edit control. (The text info record was presented in the section DoPrint, earlier in this chapter.)

The next thing we need to do is to look to see if the current text buffer is large enough to hold the text. If it is not, then we attempt to allocate a larger buffer. If the buffer cannot be allocated, we exit GetText, returning false.

```
{ Cmds2.pas, line 582 }
( *****
*
* GetText - Call Text Edit's TEGetText routine
*           to get text to write to disk.
*
* Output:
*           true if text can be returned, false otherwise
*
***** )

function GetText (* index: integer; textInfo: teInfoRec;
                  var size: longint): boolean *);

label 99;

const
    NotEnoughMemory = $0201;                                {not enough memory error}

begin
    { First call Text Edit's TEGetTextInfo to determine if the current TEGetText  }
    { buffer is large enough to hold the window's text.                          }

    TEGetTextInfo (textInfo, 2, textEdHandle [index]);
    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, getTextInfoErr, cautionAlertTyp);
        GetText := false;
        goto 99;
    end;
```


Chapter 3: Writing A Text Editor In A Few Hours

```
{ Check if current buffer is too small; if so, attempt to allocate new buffer. }

if textInfo.charCount > bufferSize then
  if MaxBlock >= textInfo.charCount then begin
    DisposeHandle (buffer);
    buffer := NewHandle (textInfo.charCount, myID, $0000, nil);
    errNum := ToolError;
    if errNum <> 0 then begin
      HandleError (errNum, memoryErr, cautionAlertTyp);
      GetText := false;
      goto 99;
    end
  else
    bufferSize := textInfo.charCount;
  end
else begin
  HandleError (NotEnoughMemory, memoryErr, cautionAlertTyp);
  GetText := false;
  goto 99;
end;
```

Once we obtain a buffer that is large enough to accommodate the window's text, we can call Text Edit's `TEGetText` function to fill the buffer with the text. We pass the routine a text descriptor flag of `$000D`, which indicates that the text is to be returned as "raw data" and that we're passing a handle to the memory area in which to store the text. The second parameter we pass is the buffer's handle. The third parameter passed is the global variable `bufferSize`, which contains the length of the block referenced by `buffer`. The fourth parameter is a verb describing the way in which the style information about the text is to be returned. The fifth parameter is a reference to the style information. We really don't want any style information in our simple editor, so this parameter is `nil`. The final parameter is the text edit control handle for this window. `TEGetText` returns the size of the text, in number of bytes.

```
{ Cmds2.pas, line 636 }
{ Call TEGetText to get window's text. We'll pass a handle to the buffer in }
{ which to store the text, and not ask for any style information.           }

size := TEGetText ($000D, buffer, bufferSize, pointerVerb,
                  nil, textEdHandle [index]);
errNum := ToolError;
if errNum <> 0 then begin
  HandleError (errNum, getTextErr, cautionAlertTyp);
  GetText := false;
  goto 99;
end;

GetText := true;

99:
end; {GetText}
```

Design Master

The DoSaveAs Procedure

The other procedure used in MiniWord to store files to disk is DoSaveAs. The procedure is called when the user selects Save as from the File menu in order to save a window to a file other than that from which the window originally came, and is also called by MiniWord when the user selects the Save command for an "Untitled" window.

DoSaveAs first issues the Standard File Operations tool set's SFPutFile2 call to get the pathname of the file to receive the window's contents.

```
{ Cmds2.pas, line 382 }
(*****
*
* DoSaveAs - Handle Save as command.
*
*****)

procedure DoSaveAs (* index: integer; currWindow: grafPortPtr *);

label 99;

const
    fileNotFound = $0046;           {File not found error, reported by GS/OS}

var
    tePtr: teRecPtr;                {pointer to textEdit control's record}
    tmp: handle;

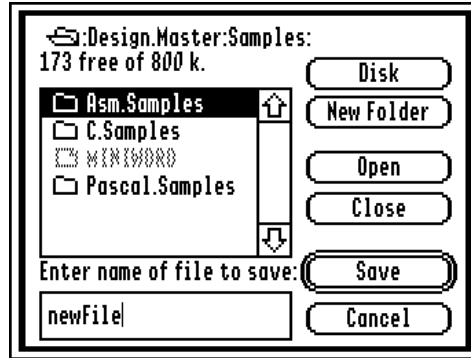
begin
    { Make SFPutFile2 call to bring up SFO Save dialog and get filename and }
    { pathname of file to create, open, and then write.                      }

    SFPutFile2 (20,                  {upper left corner X-coord of SFPutFile2's dlg}
                20,                  {upper left corner Y-coord of SFPutFile2's dlg}
                pointerVerb,          {prompt is pointer to P-string      }
                ord4 (@saveMsg),      {pointer to prompt              }
                pointerVerb,          {default name is pointer        }
                ord4 (@saveName),     {pointer to GS/OS input string  }
                theReply);            {pointer to GS/OS 5.0 reply record}

    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, fileErr, cautionAlertTyp);
        goto 99;
    end;

    if theReply.good = 0 then          {Does user want to open a file?}
        goto 99;
```

SFPutFile2 is similar to the SFGetFile2 call, which we looked at in connection with the DoOpen procedure. It brings up the standard Save File dialog box, shown below:



The first two parameters that we pass to `SFPutFile2` are the x- and y-coordinates of the upper left corner of the dialog. The values are pixel locations, in global coordinates. We use values of 20 and 20, which places the dialog in the upper left corner of the desktop, just under the menu bar.

The next parameter that we pass is a verb of pointer, that indicates that the parameter that follows will be a pointer. The next parameter is a pointer to the Pascal-style string "Enter name of file to save:"

The next pair of parameters is another pointer verb, followed by a pointer to a GS/OS class 1 input string containing the default name for the file. We use a default name of "newFile."

The final parameter is the address of the reply record, as described with the `SFGetFile2` call in the `DoOpen` procedure, above.

If the user selected the Cancel button in the Save File dialog, then the `SFPutFile2` call returns false in the good field of the reply record. In this case, we exit the `DoSaveAs` procedure. If the user selected the Save button, then good will contain true, and we continue with the `DoSaveAs` procedure. If `SFPutFile2` returns an error, we report the error and exit `DoSaveAs`.

`DoSaveAs` next calls the `GetText` function to get the window's text to write to disk. If `GetText` returns false, indicating that it couldn't obtain the text, we exit `DoSaveAs`.

```
{ Cmds2.pas, line 419 }
{ Fill in GS/OS write parameter block fields. Call TEGetText to get }
{ the text to write to disk.                                         }
```

```
if not (GetText (index, textInfo, writeRec.requestCount)) then
    goto 99;
```

Now we need to prepare to write the disk file. The first step we take is to dereference the path name's handle, returned by `SFPutFile2` in the `rrPathRef` field of the reply record. We need to skip the buffer length field, since the GS/OS calls we'll make require a pointer to an input string, rather than a pointer to an output string. After we fix the pathname pointer, we store it to the path name fields of our `getFileInfo`, `destroy`, `create`, and `open` parameter blocks.

We next need to determine whether the file already exists; if it does, we'll delete the old file. In order to find out if the file exists, we issue a GS/OS `GetFileInfo` call. If the call returns no

Design Master

errors, then we can assume that the file exists. If it does return an error, we check whether the error was a "File not found" error. If so, then we can skip deleting the old file. If any other type of error is returned, we report the error and exit DoSaveAs.

```
{ Cmds2.pas, line 426 }
{ Get pathname selected by user with SFPutFile2 call, in reply record. }
{ Then make GS/OS GetFileInfo call to see if file already exists.      }

tmp                := handle (theReply.pathRef);
getFileInfoRec.pathname := pointer (ord4 (tmp^) + 2);
GSOSGetFileInfo (getFileInfoRec);
errNum := ToolError;
if (errNum <> 0) and (errNum <> fileNotFound) then begin
    HandleError (errNum, fileErr, cautionAlertTyp);
    goto 99;
end;
```

If the file exists, we make a GS/OS destroy call to delete the file, checking for errors. If destroy returns an error, we report the error and exit DoSaveAs.

```
{ Cmds2.pas, line 438 }
{ If no error reported by GetFileInfo call, then need to delete old }
{ file before creating new file with the same pathname.            }

if errNum = 0 then begin
    destroyRec.pathname := getFileInfoRec.pathname;
    GSOSDestroy (destroyRec);
    errNum := ToolError;
    if errNum <> 0 then begin
        HandleError (errNum, deleteErr, cautionAlertTyp);
        goto 99;
    end;
end;
```

We now create a new file, using the pathname returned by SFPutFile2. We issue the GS/OS create call, checking for errors. If an error is returned by create, we report the error and exit DoSaveAs.

```
{ Cmds2.pas, line 451 }
{ Now create new file with pathname selected by user. }

createRec.pathname := getFileInfoRec.pathname;
GSOSCreate (createRec);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, createErr, cautionAlertTyp);
    goto 99;
end;
```

Once the file has been successfully created, we open the file for writing with the GS/OS open call. If we were unable to open the file, we report the error and exit DoSaveAs. If the file

Chapter 3: Writing A Text Editor In A Few Hours

was opened, we record the file reference number returned by GS/OS into the refNum fields of the write and close parameter blocks.

```
{ Cmds2.pas, line 461 }
{ Open the file to prepare for writing window's contents to disk. }

openRec.pathname := getFileInfoRec.pathname;
GSOSOpen (openRec);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, openErr, cautionAlertTyp);
    goto 99;
end;
```

DoSaveAs next checks to see if the window is an "Untitled" one, or if it is associated with a disk file. If it is tied to a disk file, we dispose of the window's previous path name handle, and then record our new path name handle. We check its association with a disk file by looking at the windowOpen array for this window; if a value of fromDisk is stored into this window's element, then it is associated with a disk file.

```
{ Cmds2.pas, line 471 }
{ Record file's reference # returned by GS/OS in write and close }
{ parameter blocks. }

writeRec.refNum := openRec.refNum;
closeRec.refNum := openRec.refNum;

{ Check if window previously belonged to another file, and if so, }
{ dispose of its old pathname handle. Record its new path handle. }

if windowOpen [index] = fromFile then
    DisposeHandle (pathHandle [index]);
pathHandle [index] := tmp;
```

In order to write the file to disk, we need to get a pointer to the text to write. We do this by dereferencing the buffer handle, which was passed to TEGetText in the GetText routine. GetText has already filled in the writeRequest field of the GS/OS write parameter block.

We now write the file to disk by issuing a GS/OS write call. If the call returns an error, we report the error, close the file, and exit DoSaveAs.

Design Master

```
{ Cmds2.pas, line 484 }
{ Dereference buffer's handle to get pointer to data to write to disk. }
{ Write the window's contents to disk. }

writeRec.dataBuffer := buffer^;
GSOSWrite (writeRec);
errNum := ToolError;
if errNum <> 0 then begin
    HandleError (errNum, writeErr, cautionAlertTyp);
    GSOSClose (closeRec);
    goto 99;
end;
```

DoSaveAs' remaining tasks are clean-up. We clear the dirty bit and write it back to the text edit record. We change the window's title to the new file name by first calling `GetOpenName` to set up the name, and then calling the Window Manager's `SetWTitle` routine, passing it a pointer to the new title (contained in the `wTitle` field of the window record, as set up by `GetOpenName`) and the window's `grafPort` pointer (contained in `currWindow` which was passed to us). We record that the window is connected to a disk file by placing the value from `disk` in the `windowOpen` array for this window. Finally, we close the disk file that contains our saved text.

```
{ Cmds2.pas, line 496 }
{ Clear the dirty bit after saving the file. }

tePtr := teRecPtr (textEdHandle [index]^);
tePtr^.ctrlFlag := tePtr^.ctrlFlag & notDirty;

{ Get opened file's name to put in window' title. }

GetOpenName (theReply, window) ; {get filename to display in window's title}
SetWTitle (window.wTitle, currWindow);

{ Record that window comes from a file, close the disk file, then record the }
{ new pathname. }

windowOpen [index] := fromFile;
thePath [index] := getFileInfoRec.pathname;
GSOSClose (closeRec);

99:
end; {DoSaveAs}
```

Creating The Final MiniWord Program

In order to create the final program, you will need to make any necessary changes to the source code, and then compile it. If you are not using ORCA/Pascal, you should consult your compiler's reference manual for instructions on how to create a stand-alone (S16) program on the Apple IIGS.

If you are using ORCA/Pascal, you can create the final program from the text environment by using the EXEC file named Build. Build's contents are:

```
unset exit

echo "Compiling globals.pas"
compile +t +e globals.pas
set  errNo {status}
if {errNo} != 0
    echo "Compilation of globals.pas failed:  status = {errNo}"
    exit {errNo}
end

echo "Compiling error.pas"
compile +t +e error.pas
set  errNo {status}
if {errNo} != 0
    echo "Compilation of error.pas failed:  status = {errNo}"
    exit {errNo}
end

echo "Compiling cmds1.pas"
compile +t +e cmds1.pas
set  errNo {status}
if {errNo} != 0
    echo "Compilation of cmds1.pas failed:  status = {errNo}"
    exit {errNo}
end

echo "Compiling cmds2.pas"
compile +t +e cmds2.pas
set  errNo {status}
if {errNo} != 0
    echo "Compilation of cmds2.pas failed:  status = {errNo}"
    exit {errNo}
end

echo "Compiling main.pas"
compile +t +e main.pas
set  errNo {status}
if {errNo} != 0
    echo "Compilation of main.pas failed:  status = {errNo}"
    exit {errNo}
end
```

Design Master

```
echo "Linking main, globals, error, cmds1, and cmds2"
link main globals error cmds1 cmds2 keep=miniword
set  errNo  {status}
if {errNo} != 0
    echo "Linking failed:  status = {errNo}"
else
    echo "Successfully built MiniWord"
    filetype miniword s16
end
```

If you are using ORCA/Pascal's desktop environment, you can create the final program with the following steps:

1. Select the `Compile...` command from the `Run` menu. In the `Compile...` dialog, deselect the `Generate debug code` and `Link after compiling` options, then click the `Set Options` button.
2. Open the source file `globals.pas`, then issue the `Compile To Disk` command from the `Run` menu. You can close the `globals.pas` window after compiling the source, if you wish.
3. Open the source file `error.pas`, then issue the `Compile To Disk` command from the `Run` menu. You can close the `error.pas` window after compiling the source, if you wish.
4. Open the source file `cmds1.pas`, then issue the `Compile To Disk` command from the `Run` menu. You can close the `cmds1.pas` window after compiling the source, if you wish.
5. Open the source file `cmds2.pas`, then issue the `Compile To Disk` command from the `Run` menu. You can close the `cmds2.pas` window after compiling the source, if you wish.
6. Open the source file `main.pas`, then issue the `Compile To Disk` command from the `Run` menu. You can close the `main.pas` window after compiling the source, if you wish.
7. In the `Shell` window, enter the commands:

```
link main global error cmds1 cmds2 keep=miniword
filetype miniword s16
```

Note: You may not have enough memory in the desktop environment to build the complete program. If you experience any out-of-memory errors, you can use the `Build` file from the text environment to create the final program.

Embellishing The MiniWord Program

Now that we've created a simple text editor, the question becomes, "What next?" One of the first things you can do to the program is to improve it. For example, you could raise the limit on the number of windows open at any one time. Another improvement that can be made is better error handling. If you've played with the program much, you may have noticed that the error reporting tends to be diagnostic in nature, and geared toward helping the programmer debug her program rather than helping the user of the program. Another (much needed) refinement would be to add more sophisticated memory management, simultaneously implementing better file handling. Other features you can add to the program are Search, Search with Replacement, and Goto commands.

Once you've improved the basic program, there are a number of items that could be added to the editor to turn it into a bona fide (albeit primitive) word processor.

For starters, you could use the Scrap Manager, Text Edit, and Design Master to create a clipboard, similar to the one used in Design Master. The clipboard could be a modeless dialog box, and would display the last block of cut or copied text.

Another feature you could add would be a ruler, since Text Edit supports one ruler for each text edit control. The ruler would allow setting the left and right margins, as well as insertion and removal of tab stops.

Of course, no word processor is complete without some means of changing the type face of blocks of the text. The styles supported on the Apple IIGS include bold, underline, outline, italic, and shadow. You could also allow the user to use different fonts in the text. You could add font and style menus to our menu bar, using Design Master, and then use Font Manager and Text Edit calls to implement the use of fonts and styles.

Chapter 4

Using Design Master to Create Resources

In this chapter, we'll learn how to use Design Master to create resources. If you are not familiar with resources, you may ask, "What is a resource and why would I want to use one?" Basically speaking, a resource is a data definition. It is a contiguous sequence of bytes that describe something, just like a record in Pascal or a structure in C. Every resource type has its own unique format; for example, menus are described in templates, while windows are initially defined in a parameter list record. The templates and records are encoded in a special way when they're turned into resources. That's why we need to use a program like Design Master to create resources: it handles the conversion for us, freeing us from needing to know the myriad of details required to perform the translation.

Files on the Apple IIGS and Macintosh computers contain two pieces, called forks. There's a data fork and a resource fork; either fork can be empty. In general, any information that the system needs to load into memory and maintain for the application is stored in the resource fork, while data that the program itself creates and manages is placed into the data fork.

When we use resources, the tools that support resources work with the Resource Manager to load and unload the resources from disk to memory. If all of the resources that we need have been stored in the resource fork of our file, then we can simply call the tools that use the resources, passing the resource IDs of the resources in our resource fork, and not have to worry about dealing with the Resource Manager directly. This is the approach we'll use in our next version of MiniWord, described in this chapter. We'll use Design Master to create a resource fork for our program, and let it add to the fork all of the resources that we need.

Before we dive into a detailed discussion of our next version of MiniWord, let's take a moment to examine the advantages and disadvantages of using resources.

One of the advantages of using resources is that they are created and changed separately from a program, so that you don't necessarily have to recompile the program every time a resource changes. For example, if you change the name of a menu item, and the menu is stored in a resource, you wouldn't have to recompile your program because this type of change doesn't impact the source code. Another advantage to using resources is that you can provide the operating system greater flexibility in managing the computer's memory. It takes care of loading resources as they are needed, and unloading them when they're not currently needed, yet the memory they occupy is. One last advantage of resources, and, in fact, the reason they were originally created, is that the end user can change information in the resource fork without having access to your source code. This makes it easy for a German user, for example, to substitute German menu names for the English names you put in the original program.

One of the disadvantages of using resources is that their use requires at least two steps: we need to create the resources, and also create the program that will use the resources. Another disadvantage is that, depending upon the program that we use to create the resources, we can't

Design Master

"tweak" individual fields in the records used in the creation of the resource. We are typically allowed to create generalized resource types; once the resource is created, we don't have direct access to its parts. It is possible to modify individual pieces of the definitions, but that requires other tools, such as Apple's Derez® utility, and an intimate knowledge of the format of Rez® source code. (Rez® is Apple's resource compiler. Derez® is Apple's resource decompiler.) If we can't directly reach individual fields, we must place great reliance on our resource editor. Blind faith can make some of us rather uncomfortable. One solution is to first build the program using source code, then later remove pieces of the code as resources are generated in their place. (This was actually the method employed in developing this chapter.)

Using Resource Forks With MiniWord

The changes we need to make to our MiniWord program to implement resource forks are amazingly few. Even for those of us who have never used resource forks, we probably already have a pretty clear picture of the tool calls we'll need to change, because of the "clues" given in the verbs we pass. The important question is, "Where do we start?"

The first thing we need to do is to return to our program, removing the source code we don't need, and changing a few tools calls. We'll then create an executable file from the source code. Finally, we'll use Design Master to save our menu bar and window definitions as resource forks, "attaching" them to our MiniWord program. The Resource Manager does not support dialogs, so we won't be changing any of our alerts or dialogs.

To understand the differences between the two versions of our MiniWord program, it is recommended that you start with the source code created in the last chapter, and change it as you read this chapter. To prepare for the tutorial presented here, you should create a folder on your MiniWord disk named Resources, and then copy the source code files from your Source folder into your Resources folder.

If you have not been working through the examples in chapters 2 and 3, then you can copy the source code files from the Samples folder on the Design.Master disk into a work folder. The source code we'll look at in this chapter can be found in the Resources folder of the Pascal.Samples folder.

Changing The Globals Unit

The documentation for the Resource Manager outlines the steps you need to follow to use resources. The first of these steps is to "log in" with the Resource Manager. You can then open resource files, and load and unload resources as you need them. The StartUpTools call handles the task of logging in with the Resource Manager. It also opens MiniWord's resource fork, so we don't have to explicitly open any resource files. The tool calls we'll make that use resources will load the resources as they're needed, so we are freed from this task, too!

Let's look at the changes we need to make to our global data structures first. When we use resources, we pass the tool a long integer containing the resource ID of the resource we want to use. This implies that we need to define resource IDs for our menus and the window. Add these constants to your globals.pas file:

Chapter 4: Using Design Master to Create Resources

```
appleMenuID = $00000000;           {Resource IDs}
fileMenuID   = $00000001;
editMenuID   = $00000002;
searchMenuID = $00000003;
windowID     = $00000004;

textEditID   = $00000005;           {Control IDs}

type
  windowType = (noWindow, fromFile, fromNew);           {Window types}
```

Notice that the resource ID is stored with the resource definition in the resource fork. We'll use Design Master to assign the IDs when we save the definitions as resource forks.

The other change we need to make to our Globals unit is to define a new type, which we'll use for our window titles. Since we won't have direct access to the `wTitle` field of the window parameter list record, we'll need to maintain a window title pointer that is separate from the window parameter list resource. We're going to define that variable in our `Cmds2` unit, where we actually deal with window titles. Add this line to `globals.pas`:

```
pString50 = packed array [0..50] of char;
pString17 = packed array [0..17] of char;
pString17Ptr = ^pString17;
```

Changing The Main Module

Next let's look at the changes we need to make to our Main programming module. The first thing we'll want to do is to remove all of the menu variables. Delete these lines from your `main.pas` file:

```
var
{ *** DATA STRUCTURES GENERATED BY DESIGN MASTER, and altered by B.A. *** }
{
{ Note: Design Master will generate the type "pString" instead of
{       "packed array of char." These have been changed to use
{       less space.
{
  dropmenutitle01: packed array [0..20] of char;           {menu titles}
  dropmenutitle02: packed array [0..20] of char;
  dropmenutitle03: packed array [0..20] of char;
  dropmenutitle04: packed array [0..20] of char;
```

Design Master

```
menu01itemtitle00: packed array [0..20] of char;           {menu item titles}

menu02itemtitle00: packed array [0..20] of char;
menu02itemtitle01: packed array [0..20] of char;
menu02itemtitle02: packed array [0..20] of char;
menu02itemtitle03: packed array [0..20] of char;
menu02itemtitle04: packed array [0..20] of char;
menu02itemtitle05: packed array [0..20] of char;
menu02itemtitle06: packed array [0..20] of char;
menu02itemtitle07: packed array [0..20] of char;

menu03itemtitle00: packed array [0..20] of char;
menu03itemtitle01: packed array [0..20] of char;
menu03itemtitle02: packed array [0..20] of char;

menu04itemtitle00: packed array [0..20] of char;

menu01: menuTemplate;
menu02: menuTemplate;
menu03: menuTemplate;
menu04: menuTemplate;

menu01item00: menuItemTemplate;
menu02item00: menuItemTemplate;
menu02item01: menuItemTemplate;
menu02item02: menuItemTemplate;
menu02item03: menuItemTemplate;
menu02item04: menuItemTemplate;
menu02item05: menuItemTemplate;
menu02item06: menuItemTemplate;
menu02item07: menuItemTemplate;
menu03item00: menuItemTemplate;
menu03item01: menuItemTemplate;
menu03item02: menuItemTemplate;
menu04item00: menuItemTemplate;
```

Since we no longer have these variables in our program, we no longer need to initialize them. Remove these lines from the InitMenus procedure in your main.pas file:

```
procedure InitMenus;

var
  tmp: longint;
  i: integer;
```

Chapter 4: Using Design Master to Create Resources

```

begin
{ *** GENERATED BY DESIGN MASTER, with comments provided by B.A. *** }

dropmenutitle01 := '@';                                {Apple menu}
with menu01 do begin
    version      := 0;
    menuID       := $0001;
    menuFlag     := $0008;                                {cache menu; will pass ptr to title}
    menuTitleRef := ord4 (@dropMenuTitle01);              {pointer to menu's title}
    itemRefs [1] := ord4 (@menu01item00);                  {About item reference }
    itemRefs [2] := 0;                                     {null terminator      }
end;

menu01itemtitle00 := 'About';
with menu01item00 do begin
    version      := 0;
    itemID       := 256;                                {About ID              }
    itemChar     := $00;                                {shortcut characters}
    itemAltChar  := $00;
    itemCheck    := $0000;
    itemFlag     := $0041;                                {bold, divider beneath, will}
                                                    { pass pointer to title }
    itemTitleRef := ord4 (@menu01itemtitle00);             {pointer to item's name}
end;

dropmenutitle02 := ' File ';
with menu02 do begin
    version      := 0;
    menuID       := $0002;
    menuFlag     := $0008;                                {cache menu; will pass ptr to title}
    menuTitleRef := ord4 (@dropMenuTitle02);              {pointer to menu's title }
    itemRefs [1] := ord4 (@menu02item00);                 {item reference: New    }
    itemRefs [2] := ord4 (@menu02item01);                 {item reference: Open   }
    itemRefs [3] := ord4 (@menu02item02);                 {item reference: Close  }
    itemRefs [4] := ord4 (@menu02item03);                 {item reference: Save   }
    itemRefs [5] := ord4 (@menu02item04);                 {item reference: Save as }
    itemRefs [6] := ord4 (@menu02item05);                 {item reference: Page setup}
    itemRefs [7] := ord4 (@menu02item06);                 {item reference: Print  }
    itemRefs [8] := ord4 (@menu02item07);                 {item reference: Quit   }
    itemRefs [9] := 0;                                     {null terminator        }
end;

menu02itemtitle00 := 'New';
with menu02item00 do begin
    version      := 0;
    itemID       := 257;                                {new ID                }
    itemChar     := $4E;                                {shortcut characters}
    itemAltChar  := $6E;
    itemCheck    := $0000;
    itemFlag     := $0000;                                {will pass pointer to title}
    itemTitleRef := ord4 (@menu02itemtitle00);             {pointer to item's name }
end;

```

. . .

Design Master

```
menu04itemtitle00 := 'Find...';
with menu04item00 do begin
    version      := 0;
    itemID       := 264;                {find ID      }
    itemChar     := $46;                {shortcut characters}
    itemAltChar  := $66;
    itemCheck    := $0000;
    itemFlag     := $0000;              {will pass pointer to title}
    itemTitleRef := ord4 (@menu04itemtitle00); {pointer to item's name }
end;
```

Now we need to change the Menu Manager calls to create our menu bar using resources rather than templates. Change these lines in InitMenus:

```
{ Create the menu bar. Start at the last menu, since we're inserting each new }
{ menu at the front of the current menu list.                                }

tmp := ord4 (@menu04); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu03); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu02); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
tmp := ord4 (@menu01); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
```

to these:

```
begin
{ Create the menu bar. Start at the last menu, since we're inserting each new }
{ menu at the front of the current menu list. Note: We're using menu          }
{ resources, whose IDs are given in the Globals unit.                          }
InsertMenu (NewMenu2 (resourceVerb, searchMenuID), 0);
InsertMenu (NewMenu2 (resourceVerb, editMenuID), 0);
InsertMenu (NewMenu2 (resourceVerb, fileMenuID), 0);
InsertMenu (NewMenu2 (resourceVerb, appleMenuID), 0);
```

Changing The Error Unit

Before we discuss the major changes we need to make to MiniWord in our Cmds2 unit, let's take care of one last set of minor changes. We'll be making a NewWindow2 call, rather than a NewWindow call, to create our document windows, so let's add an error message for the new call. We'll also be replacing our NewControl2 call with a GetCtlHandleFromID call, so we should define a new error message for this call, too. In your error.pas file, add these new error types:

```
type
{Error numbers}

errType = (OOM, fatalPrintErr, memoryErr, openErr, readErr, printErr,
           writeErr, fileErr, createErr, deleteErr, getTextErr, getFileErr,
           getTextInfoErr, fatalErr, window2Err, getCtlErr);
```

In the InitError procedure in your error.pas file, add these new error messages:

Chapter 4: Using Design Master to Create Resources

```
errMsg [fatalErr]      := 'Fatal error:  Cannot recover.';
errMsg [window2Err]    := 'Error returned by NewWindow2 call.';
errMsg [getCtlErr]     := 'Error returned by GetCtlHandleFromID call';
```

Changing The Cmds2 Unit

As we stated in the previous section, the change to resources in our MiniWord program will have its greatest impact on the Cmds2 unit.

The first change we need to make to our cmds2.pas file is to remove the window parameter list and text edit control variables. Delete these lines from the top of the cmds2.pas file:

```
var
{ *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }

windColorTable: wColorTbl;           {document window definitions}
window:        paramList;
control001111: editTextControl;
```

We want to have direct access to our window's title pointer, so we need to define a new variable to hold the pointer. Add this line to the global data area at the top of the cmds2.pas file:

```
thePath:      array [0..3] of gsosPathNamePtr;   {array of pathname pointers}
                                                    { for opening files      }
wTitle:       pString17Ptr;                      {pointer to window's title }
```

Since we no longer have a window parameter list or text edit control record in our program, we can remove the code to initialize these structures. Delete these lines from the InitCmds2 function:

```
{ *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
{ Initialize window parameter list and editText control record. }

with window do begin
    paramLength := 78;
    wFrameBits  := $C1E7;
    wTitle      := nil;
    wRefCon     := 0;
    {Window parameter list
    {parm list length
    {frame bits
    {pointer to window's title
    {we'll use to store index into
    { window-tracking arrays
    {zoomed rectangle
    }

with wZoom do begin
    v1 := $001E;
    h1 := $0014;
    v2 := $003C;
    h2 := $0064;
end;
```

Design Master

```

wColor      := @windColorTable;    {color table pointer      }
wYOrigin    := $0000;              {vert offset of content  }
wXOrigin    := $0000;              {horiz offset of content }
wDataH      := $0000;              {data area height        }
wDataW      := $0000;              {data area width         }
wMaxH       := $00C8;              {max grow height         }
wMaxW       := $0280;              {max grow width          }
. . .
wPlane      := grafPortPtr (-1);   {window plane, -1 for front }
wStorage    := nil;                {address of memory for window record}
end;

with windColorTable do begin
  frameColor := $0010;              {window frame color}
  titleColor := $0D81;              {title color        }
  tBarColor  := $021D;              {title bar color     }
  growColor  := $0074;              {grow box color      }
  infoColor  := $0000;              {info bar color      }
end;

with control001111 do begin        {editText control template }
  pCount      := 23;                {parameter count          }
  controlID    := $000087DA;         {ID number TaskMaster will use }
  with boundsRect do begin         {encloses entire control, including}
    v1 := $0002;                    { scroll bars and grow box }
    h1 := $0002;
    v2 := $0000;
    h2 := $0000;
  end;
  . . .
  drawMode     := 0;
  filterProcPtr := nil;              {use built-in filter procedures}
end;

```

We now need to remove all references to the window parameter list. One of the parameters passed to the `GetOpenName` procedure is the window parameter list; this needs to be erased from the procedure's forward declaration at the top of the file, from the procedure's definition, and from all calls to `GetOpenName`. Change the procedure's headings:

```

procedure GetOpenName (var theReply: replyRecord5_0; window: paramList); forward;
{ Gives new window a title based on file just opened. }

(*****
*
* GetOpenName - Build filename when open a file.
*
*****)

procedure GetOpenName (* var theReply: replyRecord5_0; window: paramList *);
{ Will dispose of filename's handle; will change window's title. }

```

to:

Chapter 4: Using Design Master to Create Resources

```
procedure GetOpenName (var theReply: replyRecord5_0); forward;
{ Gives new window a title based on file just opened. }

(*****
*
* GetOpenName - Build filename when open a file.
*
*****)

procedure GetOpenName (* var theReply: replyRecord5_0 *);
{ Will dispose of filename's handle; will change window's title. }
```

GetOpenName is called in the DoOpen and DoSaveAs procedures. Change the calls from:

```
GetOpenName (theReply, window); {get filename to display in window's title}
```

to:

```
GetOpenName (theReply); {get filename to display in window's title}
```

Within the GetOpenName procedure, we need to change the reference to the wTitle field of the window parameter list to simply wTitle. Change this line:

```
window.wTitle^ := concat (' ', tmpPtr^, '');
```

to:

```
wTitle^ := concat (' ', tmpPtr^, '');
```

One other subroutine in our Cmds2 unit, InitWindow, expects to be passed a window parameter list. As with GetOpenName, we need to remove the parameter from the procedure's declaration, definition, and in calls to it. Change its forward declaration at the top of the file from:

```
procedure InitWindow (var window: paramList; var index: integer); forward;
{ Initializes window creation. }
```

to:

```
procedure InitWindow (var index: integer); forward;
{ Initializes window creation. }
```

Change its procedure heading from:

Design Master

```
(*****
*
* InitWindow - Perform window initialization,
*             common to both DoOpen and DoNew.
*
*****)

procedure InitWindow (* var window: paramList; var index: integer *);
```

to:

```
(*****
*
* InitWindow - Perform window initialization,
*             common to both DoOpen and DoNew.
*
*****)

procedure InitWindow (var index: integer *);
```

InitWindow is called by the DoNew and DoOpen procedures. Change the calls to it from:

```
InitWindow (window, index);           {init. window data structures }
```

to:

```
InitWindow (index);                   {init. window data structures }
```

Within InitWindow, we need to remove the references to the window parameter list. We can delete the line that refers to the window's refCon, since we'll be using another method to set this value when we create the window. Therefore, remove this line from the InitWindow procedure:

```
{ Use window's RefCon to track position in window arrays. }

window.wRefCon := index;
```

InitWindow still must initialize the window's title pointer, so change this line:

```
{ Set window's title pointer. }

window.wTitle := @wName [index];
```

to:

```
{ Set window's title pointer. }

wTitle := @wName [index];
```

Chapter 4: Using Design Master to Create Resources

There are two other subroutines in `cmds2.pas` that refer to `window.wTitle`: `GetUntitledName` and `DoSaveAs`. In `GetUntitledName`, change the line:

```
{ Move character representation of untitled number into the window's title. }  
  
window.wTitle^ := concat (' Untitled ', num, ' ');
```

to:

```
{ Move character representation of untitled number into the window's title. }  
  
wTitle^ := concat (' Untitled ', num, ' ');
```

In `DoSaveAs`, change the line:

```
{ Get opened file's name to put in window' title. }  
  
GetOpenName (theReply, window) ; {get filename to display in window's title}  
SetWTitle (window.wTitle, currWindow);
```

to:

```
{ Get opened file's name to put in window' title. }  
  
GetOpenName (theReply, window) ; {get filename to display in window's title}  
SetWTitle (wTitle, currWindow);
```

We're now ready to discuss the biggest set of changes we need to make to our `Cmds2` unit, in our `CreateWindow` routine. To begin, delete the local `tmp` variable. Now change the `NewWindow` call to a `NewWindow2` call, by changing these lines:

```
{ Call Window Manager's NewWindow function to create new window on desktop. }  
  
windowPtr [index] := NewWindow (window);  
errNum           := ToolError;  
if (errNum <> 0) then begin  
    HandleError (errNum, memoryErr, cautionAlertTyp);  
    CreateWindow := false;  
end
```

Design Master

to:

```
{ Call Window Manager's NewWindow2 function to create new window on desktop. }
{ Pass pointer to window that can be maintained separately from the window }
{ record, pass refCon field for new window. }

windowPtr [index] := NewWindow2 (pStringPtr (wTitle),    {pointer to title }
                                index,                  {refCon          }
                                nil,                    {no ContentDraw proc}
                                nil,                    {standard defProc }
                                resourceVerb,           {parm list = resrc }
                                windowID,               {window's resrc ID }
                                $800E);                 {resource type    }

errNum := ToolError;
if (errNum <> 0) then begin
    HandleError (errNum, window2Err, cautionAlertTyp);
    CreateWindow := false;
end
```

Notice the parameters that we pass NewWindow2. As we mentioned earlier, we need access to the window's title, since we change it with every new window that is opened, so we keep the wTitle variable separate from the resource. Our new document window will be created with the correct window title by passing NewWindow2 the pointer to the desired title. The same method is used to assign a refCon to the window, by keeping a variable separate from the resource, and then passing the variable (index) to NewWindow2. The windowID parameter is a global constant. When we save the window as a resource fork with Design Master, we'll assign the same ID that we defined in our program. The last parameter of interest is the resource type, the constant \$800E. You can find a complete list of these constants in the *Apple IIGS Toolbox Reference Manual, Volume 3*.

NewWindow2 calls NewControl2 to create the window's text edit control. We made this call ourselves in the last version of MiniWord. From it, we obtained the control's handle. In this version, we call the Control Manager's GetCtlHandleFromID function to obtain the control's handle. Change the lines:

```
tmp := ord4 (@control001111);
textEdHandle [index] := NewControl2 (windowPtr [index], pointerVerb, tmp);
errNum := ToolError;
if (errNum <> 0) then begin
    HandleError (errNum, memoryErr, cautionAlertTyp);

    to:

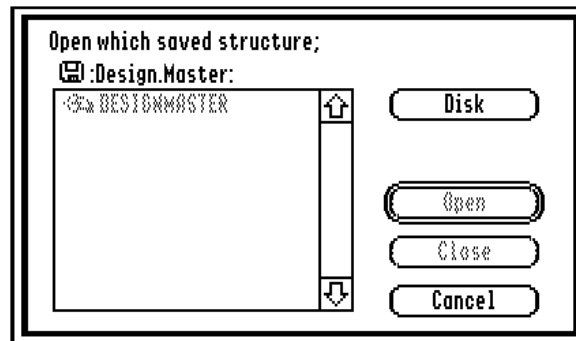
textEdHandle [index] := GetCtlHandleFromID (windowPtr [index], textEditID);
errNum := ToolError;
if (errNum <> 0) then begin
    HandleError (errNum, getCtlErr, cautionAlertTyp);
    DoClose (index, windowPtr [index], numWindows);
    CreateWindow := false;
end;
```

Notice that the control ID we pass was generated by Design Master, when we specified the ID in the text edit dialog.

Creating The MiniWord Program

The next step we need to take is to compile the source code into an executable file. The instructions for creating a stand-alone (file type S16) version of MiniWord are identical to those given at the end of the last chapter.

We're now ready to save our menu bar and window definitions as resource forks, and write the resource forks to our executable program. To do this, launch Design Master. Now pull down the `File` menu and select the `Open DM file` command. It presents us with a dialog similar to this:



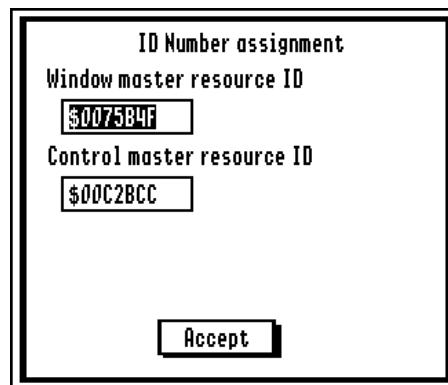
Use the `Disk` and `Folder` buttons to move around on your system until you reach the `Binary` folder on your data disk. Select the `menu.bin` file from the file list, and then click the `Open` button to open it. You should see the menu bar that we created in Chapter 2 on the desktop. (If you didn't save the menu bar definition, you can use the `New MenuBar` command in the `File` menu to create one now. You can refer to Chapter 2 to find the menus and menu items that we defined for our MiniWord program.)

To save the menu bar as a resource fork, select the `Save` command from the `File` menu. Click the radio button next to the `Resource fork` option and deselect the `Save binary image` box (if it's selected). Now move to the disk and folder containing the S16 MiniWord program we just created. In the `Save structure as` edit-line box, type **miniword**, then click the `Save` button. Design Master should respond with this alert box:

Design Master



Do not be alarmed; you will not wipe out your program by selecting the `Replace` button in this dialog. Instead, Design Master will add a resource fork to the program. Click the `Replace` button, and Design Master will bring up this dialog:



Now close the menu bar with the `Close` command from the `File` menu, and again select the `Open DM` file command from the `File` menu. This time open the `window.bin` file that we saved when we created our document window in Chapter 2.

After the window appears, make any changes you think it might need (such as different colors), and then select the `Save` command from the `File` menu, selecting the `Resource fork` option and deselecting the `Save binary image` box. Now move to the disk and folder containing the `miniword` program to which we just added a resource fork, enter the name **miniword** in the `Save structure as` box, and click the `Save` button. Again you'll be presented with the `Replace file?` alert. As with saving the resource fork for the menu bar, the `miniword` program will not be overwritten with the window resource fork. Instead, the window's resources will be added to the end of the current resource fork.

Once you've clicked the `Replace` button, you will see an `ID Number assignment` dialog box. Type **4** in the `Window master resource ID` box, since this is the resource ID we're using in our `MiniWord` program when we make the `NewWindow2` call. Type **5** in the `Control master resource ID` box, since this is the resource ID we're using for our text edit control. Click the `Accept` button when you've made the correct entries.

At this point, you should have a version of the `MiniWord` program that contains a resource fork!

Chapter 4: Using Design Master to Create Resources

Whenever you use Design Master to create a resource fork, it will issue a text file that reports the resources created. The file will be named REPORT.RFORK**xx**, where the **xx** is a number that starts at 00 and increments for each report file written to the prefix. A typical report file, generated for our menu bar, looks like this:

```
Report on resources added to resource fork.
The following resources were added to the indicated resource
fork, with the indicated ID numbers. Please keep track of
this information so you can use the resource you have created.
Fork created 3/28/90 2:07:52 PM
Resource Fork name - miniword

MenuBar resource, ID number; $00000000
Menu resource saved, ID number; $00000000
Menu resource saved, ID number; $00000001
Menu resource saved, ID number; $00000002
Menu resource saved, ID number; $00000003
```

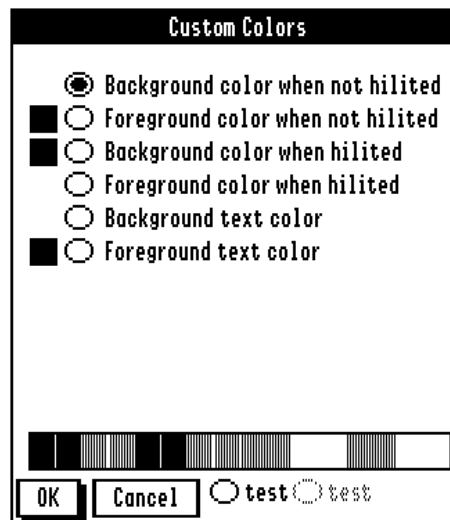

Chapter 5

Command Reference

This chapter provides detailed descriptions of every command available in Design Master. The entries are arranged by menu, beginning with the `Apple` menu and proceeding through the `Controls` menu. Items within each menu are discussed from top to bottom, as they appear in their menu.

"Coloring" With Design Master

Many of the commands provide a `Custom Colors` option, which allows you to "color" the item you are creating. The `Check box` command's `Colors` dialog is typical:



To color a field, click on the radio button next to the name of the field, then click on a color in the palette at the bottom of the dialog. The box next to the field's radio button will be filled with the selected color. The two `test` buttons below the palette show how the colored control will look normally and dimmed, respectively. Select the `OK` button to color the item, or `Cancel` to exit the `Custom Colors` dialog without coloring the item.

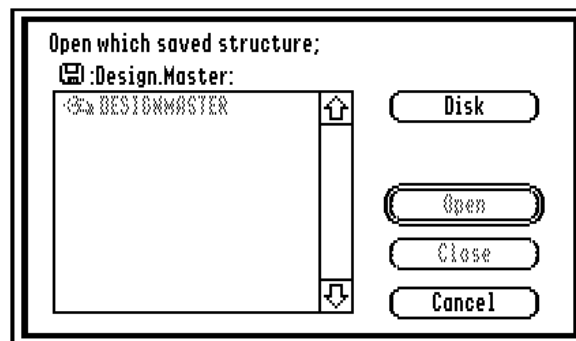
Design Master

You will want to use the same mode (320 pixels wide or 640 pixels wide) when using Design Master as you will be using in your program. This is because the appearance and color of a structure or control changes dramatically when you change screen modes.

The colors you choose can also dramatically affect the appearance of a control. By default, Design Master uses the same colors Quick Draw II chooses for your program. If you elect to change the master color palette, be sure to change it as you design the structures for your program, as well as in your program. Naturally, you should use the same master palettes in both places.

Accessing Disk Files

Most desktop programs use the Standard File Operations tool set to open and save files. SFO provides several standard dialog boxes. Several Design Master commands use these dialog boxes; the `Open DM file` command is typical. This section describes the general features of an SFO dialog.



At the top of the dialog is the message `Open which saved structure:`. Beneath the message is an icon depicting the current prefix, followed by the name of the prefix. For example, in the dialog above, the current prefix is the `Design.Master` disk. Below the current prefix display is a box containing the files and directories (folders) within the current prefix. Directories are always selectable; files that are not in the proper file format are dimmed and unselectable.

The `Disk` button is used to move through the disks currently on-line. (You can also use the `TAB` key to move from disk to disk.) The `Open` button is used to open directories. (The `Open` button is boldly outlined to show that it is the default button for the dialog; pressing the `RETURN` key will also select the default button.) The `Close` button is used to close folders. The `Cancel` button exits the dialog without opening a file. Note: the current prefix will be set to the disk/folder selected. The `Cancel` button will not cause the current prefix to be reset to its original setting.

If you have opened a folder, you can click on the icon or name in the current prefix display above the file list to move up a directory level; this short-cut has the same effect as clicking on the `Close` button.

The disk operation is performed by clicking on a file name in the file list and then selecting the button corresponding to the desired operation, or by double-clicking the file name.

The Apple Menu



The Apple menu is a standard menu in most desktop programs. The menu typically features an About item, and provides access to any new desk accessories that are installed in your system. Design Master's Apple menu follows this standard. The menu is divided into two parts. The top half features an About item and a Help command, and the bottom half provides access to the NDAs installed in the System/Desk.Accs subdirectory of your boot disk.

About Design Master

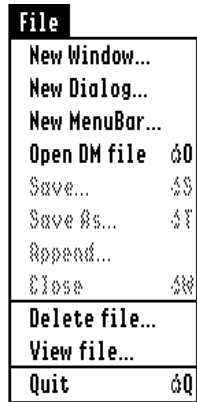
This item provides the name of the product, Design Master; the author, C. K. Haun; the publisher, Byte Works, Inc.; the version number; and a copyright message.

Help! (⌘H)

The Help command provides brief descriptions of Design Master's most important commands. It brings up a window that can be scrolled vertically. The topics covered are arranged by menu, and by items within menus, in the same way that this chapter is organized.

The File Menu

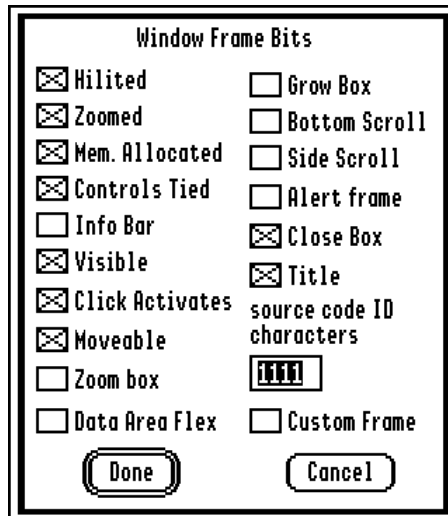
Design Master



The `File` menu is a standard menu in most desktop applications. It typically contains commands to open and save files, and to quit the program. You will use this menu to create new windows, menu bars, and dialog boxes; to open previously created structures; to create source files and resource forks; to remove the current structure from the desktop; to delete disk files; and to exit Design Master.

New Window...

`New Window` is used to create a new document window. It brings up a dialog box like the one pictured below:



Chapter 5: Command Reference

Each check box corresponds to a bit in the `wFrameBits` flag of the window parameter list record passed to the Window Manager's `NewWindow` call. For more in-depth information about each bit, refer to the *Apple IIGS Toolbox Reference* manual, volume 2, pages 25-84 through 25-86. The table below summarizes the use of the check boxes:

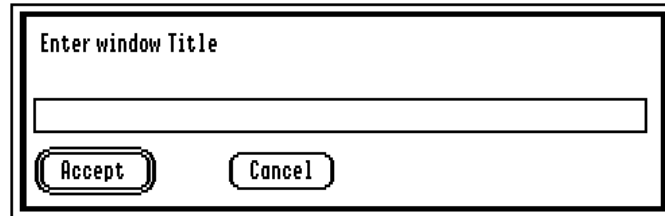
<u>wFrameBits name</u>	<u>New Window dialog name</u>	<u>Effect of selecting parameter</u>
<code>fTitle</code>	Title	Window will have a title bar in the frame.
<code>fClose</code>	Close box	Window will have a close box in the title bar.
<code>fAlert</code>	Alert frame	Window is to be an alert window. Note: Selecting this parameter causes the Info bar, Zoom box, Data area flex, Grow box, Bottom scroll, Side scroll, Close box, and Title items to become unselectable.
<code>fRScroll</code>	Side scroll	Window frame will contain a vertical scroll bar.
<code>fBScroll</code>	Bottom scroll	Window frame will contain a horizontal scroll bar.
<code>fGrow</code>	Grow box	Window frame will contain a grow box.
<code>fFlex</code>	Data area flex	The height and width of the window's content region are flexible.
<code>fZoom</code>	Zoom box	Window frame will contain a zoom box.
<code>fMove</code>	Movable	Window can be dragged by its title bar.
<code>fQContent</code>	Activate on click	Clicking in the content region of the window, when it is inactive, selects the window.
<code>fVis</code>	Visible	Window is visible.
<code>fInfo</code>	Info bar	Window will have an info bar.
<code>fCtrlTie</code>	Controls tied	The state of the controls will be tied to state of the window (either active or inactive).
<code>fAllocated</code>	Mem. allocated	The window's window record will be freed when the window is closed.
<code>fZoomed</code>	Zoomed	Window's initial size and position is the zoomed size and position. Note: This parameter will be unselectable if Zoom box has not been selected.
<code>fHilited</code>	Hilited	This parameter is set by the <code>NewWindow</code> call; its value is ignored by the Window Mgr.

The source code ID characters edit-line box allows you to enter a four-character code. This code is placed in the source code and resource forks generated by Design Master.

Once all options have been selected, click the Done button to create a new window, or select Cancel to exit the New Window dialog box without creating one.

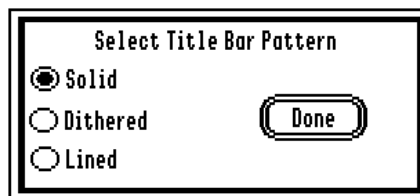
If you selected the Title check box, pressing the Done button brings up a dialog box that allows you to enter the window's title:

Design Master



The name is typed into the edit-line box. If you typed nothing in the edit-line box, the window will have no title. Clicking the `Cancel` button causes you to be returned to the `Window Frame Bits` dialog box, without setting the title. The `Accept` button sets the window's title to whatever characters you typed.

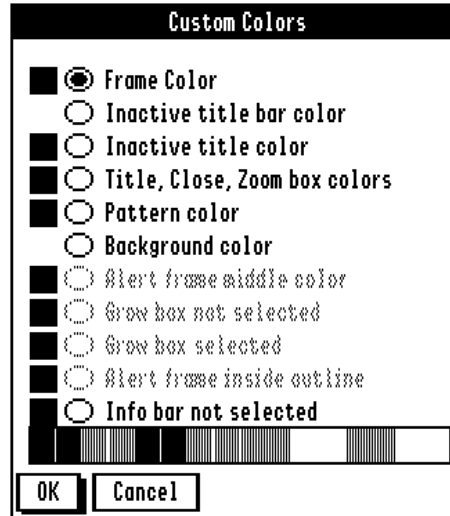
If you selected the `Custom Frame` check box (and selected the `Done` button) in the `Window Frame Bits` dialog box, you are presented with two additional dialog boxes. The first allows you to select the title bar's pattern:



The `Solid` radio button is used to color the title bar with a solid pattern, while the `Lined` button produces horizontal lines in the title bar. The `Dithered` button produces a "cross-hatch" pattern, similar to that depicted below:



The next dialog that appears allows you to "color" your new window:



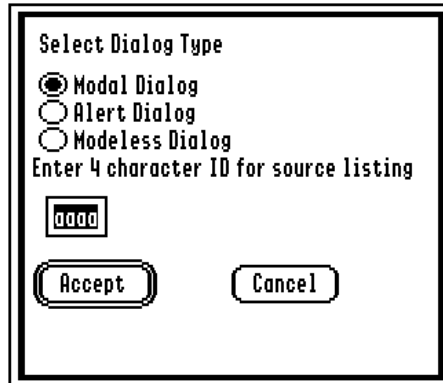
Frame color refers to the outline of the window. Inactive title bar color is the color that the title bar will have, not including the title, when the window is inactive. Inactive title color is the color of the window's title when the window is inactive. Title, close, zoom box colors is the color of these items in an active window. Pattern color refers to either a lined or dithered title bar pattern color of an active window, or the title bar color of an active window when the pattern is solid. Background color is the color beneath a lined or dithered title bar in an active window. Alert frame middle color is the color of that region between the outline of the window and the outline of the content region. Grow box not selected refers to the outline of the grow box, while Grow box selected refers to the inner rectangles of the control when the grow box is being dragged. Alert frame inside outline is the outline of the content region. Info bar not selected is the color of the info bar when it is not being selected.

After all selections have been entered, a new window is created on the desktop. In the lower right corner of the window is a solid box. You can use the mouse to click on the box, and then holding the mouse button down, size the window. The active commands in the menus can then be used to add controls to the window, test it, change its frame and title bar, and save the window description as source code or a resource fork.

New Dialog...

This command allows you to create a dialog box. You are presented with an initial dialog that allows you to select the type of dialog to be created:

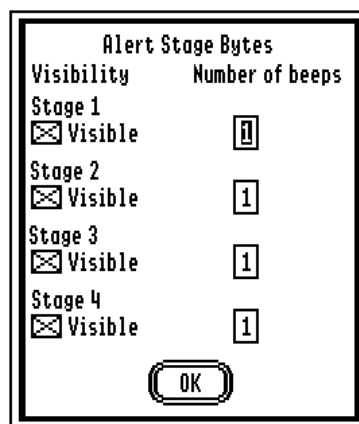
Design Master



Click on the radio button next to the type of dialog you wish to create, either Modal, Modeless, or Alert. As with the New Window command, the four-character ID is the name Design Master will use for the data structure in your source code. Click the Accept button to continue creation of the new dialog, or click Cancel to exit the New Dialog dialog box, without creating one.

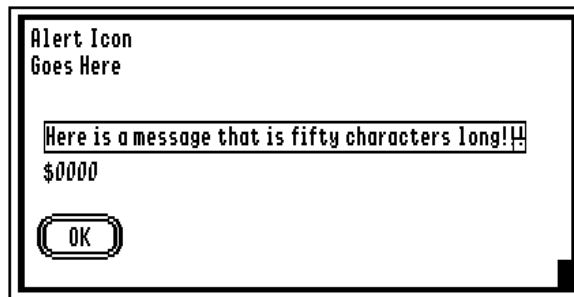
If you select the Modal Dialog radio button, clicking Accept will create a new modal dialog box on the desktop. In the lower right corner of the dialog is a solid box. You can use the mouse to click on the box, and then holding the mouse button down, size the dialog. The active commands in the menus can then be used to add controls to the dialog, test it, and save its parameters as source code or a resource fork.

Selecting the Alert Dialog button brings up a dialog that lets you select properties for an alert box:



Alerts have four stages, corresponding to four repetitions of the same action which cause the alert to appear in a program. You use the Alert dialog, as depicted above, to select the

sounds to be emitted at each stage, and whether or not the alert box should be drawn at each stage. After selecting the characteristics for each stage, click the `Done` button to complete the creation of a new alert box. A sample alert box is shown below:



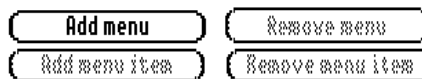
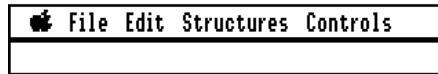
There are four different kinds of alerts: note alert, caution alert, stop alert, and standard alert. In a program, all of these except the standard alert have an icon in the upper left corner of the alert box. Since the type of alert call used determines the icon, and the icon itself is not a parameter in the alert template, Design Master has no way of knowing what the icon will be. The red-outlined box labeled `Alert Icon Goes Here` shows the proper size and position of the icon for layout purposes.

In the lower right corner of the alert is a solid box. You can use the mouse to click on the box, and then holding the mouse button down, size the alert. The active commands in the menus can then be used to add controls to it, test it, and save its parameters as source code or a resource fork.

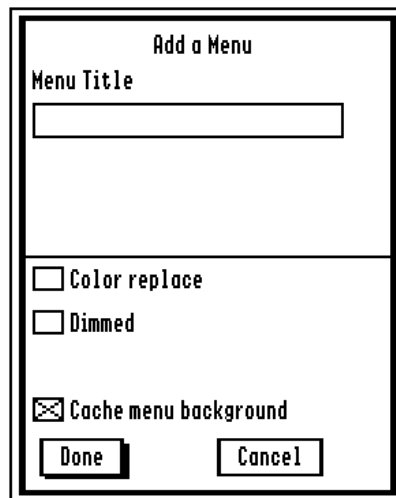
New MenuBar...

This command is used to create a menu bar. It creates a menu bar work area on the desktop, like that below:

Design Master



If a menu bar has not yet been created, the only selectable button will be `Add menu`. Selecting `Add menu` causes this dialog to appear:



In the edit-line box beneath `Menu Title`, you can enter the name of the new menu. The `Color replace` check box is used primarily with the Apple menu; it tells the menu manager to exchange black pixels for white pixels when the menu is inverted, instead of exchanging the bits of all colors, as it normally does. The `Dimmed` check box indicates that the menu's default appearance is dimmed and unselectable. The `Cache menu background` check box tells the menu manager to remember the appearance of a menu after it has been pulled down. This memory copy can be used the next time the menu is pulled down to draw the menu faster. These

Chapter 5: Command Reference

menu images are stored in purgeable memory blocks, and the menu manager is smart enough to redraw the menu if it changes, so there is no penalty for using this feature on standard menus. If you are writing custom menu procedures, as described in the *Apple IIGS Toolbox Reference* manual, you may need to disable this feature.

After making the menu selections, click on the **Done** button to create a new menu, or click **Cancel** to exit the **Add menu** dialog.

When the Menu Manager draws a menu bar, it packs the menu names into the menu bar without intervening spaces. You can separate the menu names by placing spaces in the menu titles. Because the spaces are highlighted along with the rest of the menu title when the menu is selected, we recommend putting the same number of spaces before and after every menu on the menu bar.

Most menu titles are simple text strings, but there is one common exception. In almost all desktop applications, the first menu is the Apple menu. To create this menu using Design Master, enter the @ character in the **Menu Title** box. No spaces may precede or follow the @ character.

After creating a new menu, the menu's name will appear in the menu bar area of the **New MenuBar** dialog, and the **Remove menu** and **Add menu item** buttons will become selectable.

The **Add menu item** button brings up a dialog similar to this:

The image shows a dialog box titled "Add menu item". It contains the following elements:

- Item title:** A text input field.
- Options:** A dropdown menu currently showing "Bold".
- Item ID number:** A text input field containing the number "301".
- Key:** A dropdown menu currently showing "None".
- Add to which menu?:** A section with a text input field containing "@" and a scrollable list box to its right.
- Buttons:** "Cancel" and "Done" buttons at the bottom left.

In the edit-line box beneath **Item title**, you can enter the name of the menu item.

Next to **Options** is a pop-up menu that allows you to select a number of attributes that will affect the appearance of the item. Selecting **divider** causes the item to have a dividing line beneath it in the menu. **Underline**, on the other hand, merely underlines the item. **Bold**, **italic**, **outline**, and **shadow** specify the type face used in drawing the item's name. Selecting **disabled** causes the item's initial appearance to be dimmed and unselectable. **Marked** places a check mark to the left of the item's name. **XOR** specifies XOR highlighting to be used.

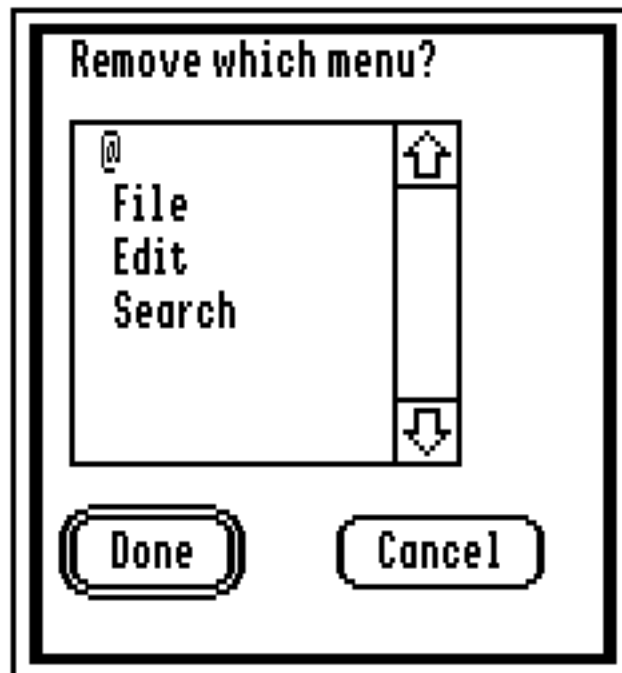
Design Master

Below the `Options` pop-up is an edit-line box labeled `Item ID number`, used to identify the menu item that has been selected in a desktop program. You can enter a decimal value in the range 1 to 65,534.

Beneath the `Item ID` box is a pop-up menu labeled `Key`. This menu is used to assign a key equivalent for the menu item, so that the item can be selected either from its menu or by holding down the `⌘` key and then pressing the keyboard equivalent. From the `Key` pop-up, you can select the `Character` item to enter a character for the keyboard equivalent in the edit-line box next to the pop-up, or you can select the `ASCII code` item to enter a decimal value corresponding to the character's ASCII code. Valid ASCII values range from 0 to 255. The `None` item in the pop-up is the default, and is used to specify that no keyboard equivalent is to be used.

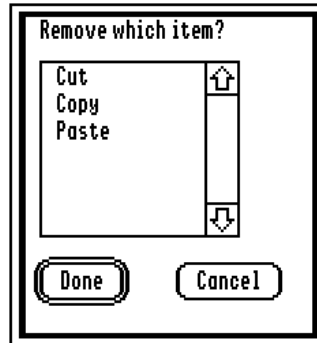
With the exception of `divider`, all other options can be used in any combination.

The `Remove menu` button cause a dialog similar to that below to appear:



Use the mouse to select a menu to delete from the current menu bar. Click the `Done` button to remove the menu, or `Cancel` to exit the `Remove menu` dialog.

The `Remove menu item` button brings up this dialog:



To remove a menu item, select it from the list of menu items and click on the `Cancel` button.

Open DM file... (⇧O)

This command is used to load a previously created description file for a menu bar, dialog box, or window. (See the `Save as` command, described later in this section, for information on creating a description file in Design Master format.) The command is available when no structures are open on the desktop. It brings up a Standard File open dialog, described at the beginning of this chapter in the section "Accessing Disk Files." After opening a Design Master binary file, the structure that the file describes will appear on the desktop. You can then change the structure, test it, and save it to the same file or to a different file, in the same or in a different format.

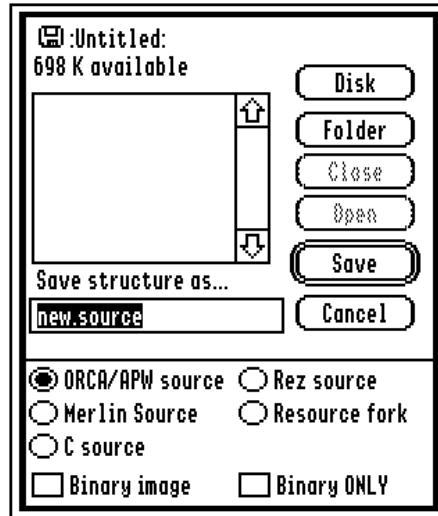
Save... (⇧S)

The `Save` command saves the current window, dialog, or menu bar to disk. If you have already used the `Save` command for the current structure, subsequent `Saves` will write the current information to the file specified for the first `Save` operation. The first `Save` command issued for a structure is identical to the `Save as` command, described next.

Save As...

Once a window, menu bar, or dialog has been created, the definitions can be saved to disk in a variety of formats. This is the purpose of the `Save as` command. It causes a dialog box like the one below to appear:

Design Master



The current prefix display, file list, and Disk, Open, Close, and Cancel buttons all function as described in the section "Accessing Disk Files" at the beginning of this chapter.

Below the file list is an edit-line box labeled with `Save structure as...`. The default name, filled in by Design Master, is `new.source`. You can type in the name of the new file in this box. The maximum number of characters that can be typed into the edit-line box is 15, since this is the maximum length for file names allowed by the ProDOS FST used by GS/OS.

The `Folder` button is used to create a new folder. Be sure to enter the name of the new folder in the `Save structure as...` box before clicking on `Folder`; otherwise, the name appearing in the box will be given to the new folder. If the current disk is write-protected, the `Folder` button will be dimmed and unselectable.

The `Save` button causes the current definition to be saved to disk. If you enter a name that is the same as another file in the current prefix, an alert will appear that gives you a second chance to consider over-writing the file. Simply click on `Replace` to go ahead with the over-write operation, or click `Cancel` to exit the operation.

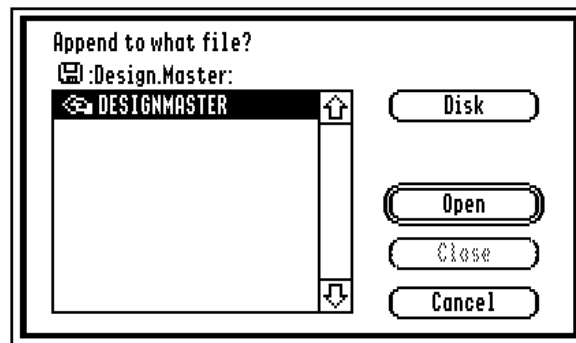
At the bottom of the `Save as` dialog is a list of available file formats. Click in the circle (radio button) to the left of the desired format to select it. The `ORCA/APW source` button indicates that the file will be saved as an ASM65816 assembly-language file. The `Merlin source` button is used to save the file as Merlin 816 assembly-language source code. The `C source` and `Pascal source` buttons will save the definitions as C and Pascal source code, respectively. The `Rez source` button is used to save the file as source code for Apple's Rez® compiler. The `Resource fork` button creates a resource fork from the definitions. If you are creating a dialog box, these last two options will be replaced with the message "Dialogs do not support resources."

The final two selections, a check-box labeled `Binary image` and another labeled `Binary ONLY`, save the file in Design Master's internal format. Such a file can be loaded into Design

Master with the `Open DM file` command. You can elect to save the file as both source code or a resource fork, as well as a binary image, if you select the `Binary image` box. Selecting both types of formats causes a second `Save as` dialog to appear, which allows you to specify the name of the binary file. The file name edit-line box will contain the name you entered in the first `Save as` dialog, with a suffix of `.BIN` appended to it. You can use this second dialog box in the same way as the first one to save the binary file. You can also choose to save the file as a binary image only, by selecting the `Binary ONLY` option. This will also bring up the second save dialog, with `.BIN` appended to the name in the filename box. When you select this option, all of the radio buttons and the other check box become dim and unselectable.

Append...

This command is used to append the current window, menu bar, or dialog definition to the end of an existing file. The file selected determines the format in which the definition will be saved. `Append` brings up a dialog similar to this:



The current prefix display, file list, and `Disk`, `Open`, `Close`, and `Cancel` buttons all function as described in the section "Accessing Disk Files" at the beginning of this chapter.

To perform the append operation, select a file from the file list and then click the `Open` button.

Close (⌘W)

The `Close` command removes the current window, dialog, or menu bar from the desktop. If the structure has not been saved to disk, or if changes have been made since the last save, you are presented with an alert box that gives you an opportunity to save the definition before closing it:

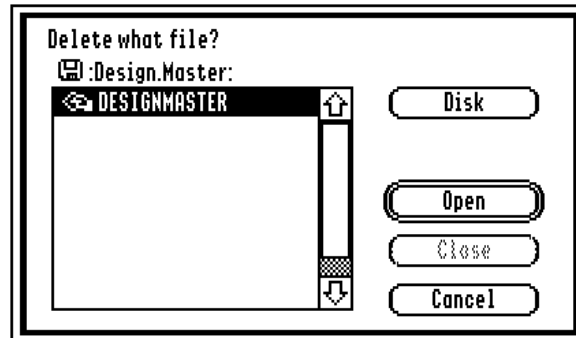
Design Master



Click the **No** button to remove the structure without saving it. Select **Cancel** to abort the **Close** command. Selecting the **Save** button causes the **Save** command to be issued, if you've previously saved the definition, or else invokes the **Save as** command if the definition has never been written to disk.

Delete file...

This command is used to delete a disk file. It brings up a dialog like this one:

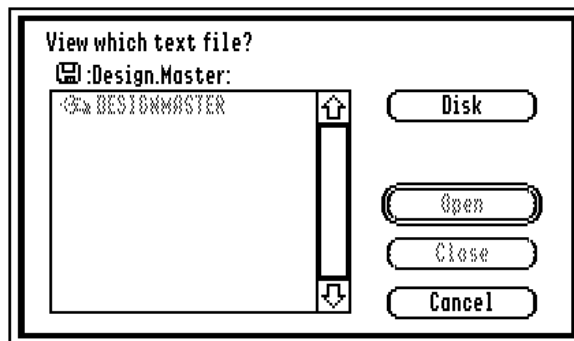


The current prefix display, file list, and **Disk**, **Open**, **Close**, and **Cancel** buttons all function as described in the section "Accessing Disk Files" at the beginning of this chapter.

To delete a disk file, select the file from the file list and then click the **Open** button. Note that files residing on write-protected disks and locked files (files having delete access disabled) will appear dimmed and unselectable in the file list.

View file...

You can use the View file command to look at the source code files created by Design Master. The command can also be used to examine the contents of any SRC or TXT file currently on-line. It brings up a Standard File open dialog similar to this:

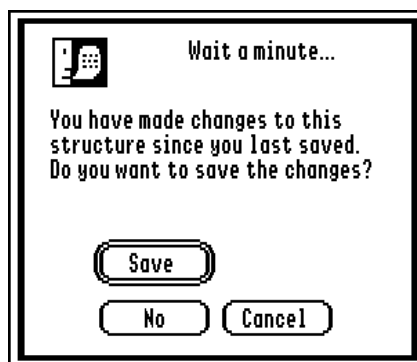


The current prefix display, file list, and Disk, Open, Close, and Cancel buttons all function as described in the section "Accessing Disk Files" at the beginning of this chapter.

To open a file for viewing, select the file from the file list and then click the Open button. Note that files that are not ASCII text files are dimmed and unselectable. Once a file has been opened, it is displayed in a text window. You can scroll through the file, and move and size its window, but you cannot alter the window's contents. Text can be selected in the window and copied to the Clipboard.

Quit (⌘Q)

This command is used to exit Design Master. If you have created a structure, but have not saved it to disk, you are asked if you wish to save your work before exiting:



Design Master

If you select **Save**, and have not yet saved the definition to disk, the **Save as** dialog appears. If you select the **Save** button, and have saved the structure at least once to disk, but have made changes since the last save operation, the **Save** command is invoked. If you select **No**, the program is exited immediately. If you select **Cancel**, the **Quit** operation is aborted.

The Edit Menu

Edit	
Undo Cut	⌘Z
Cut	⌘X
Copy	⌘C
Paste	⌘V
Clear	
Test run	⌘R
Turn Grid On	⌘G
Set Grid...	
Show Coordinates	
Source ID chars...	
Switch to 320 mode	
Preferences...	
Show Clipboard	

The **Edit** menu is a standard menu in most desktop applications. It typically contains commands to cut, copy, and paste items created in the application. Design Master's **Edit** menu allows you to undo operations; cut, copy, paste, and clear controls; test the current structure; set up a grid system on the desktop; toggle the screen between 640 and 320 modes; and customize the program.

Undo (⌘Z)

After you move, cut, delete, or paste a control, you can undo the operation. To undo the most recent operation, pull down the **Edit** menu and select the **Undo** command.

After you undo an operation, or before you have done anything that can be undone, the menu item is called **Can't Undo**, and is unselected. When you can undo an operation, the **Undo** menu item lists the operation. For example, right after using the **Cut** command to remove a control, the menu item will be called **Undo Cut**.

Cut (⌘X)

`Cut` is used to remove the selected control from a structure. (A control is selected by clicking on it.) In addition to deleting the control, `Cut` places it into the Clipboard. The new control replaces the last control placed into the Clipboard with a `Cut` or `Copy` command. You can then use the `Paste` command, described later in this section, to copy the control into a structure.

See the `Show Clipboard` command, described at the end of this section, for information about viewing the current Clipboard contents.

Copy (⌘C)

`Copy` is used to create a copy of the selected control in a structure. (A control is selected by clicking on it.) The control is copied into the Clipboard. The new control replaces the last control placed into the Clipboard with a `Cut` or `Copy` command. From there, you can use the `Paste` command to copy the control into a structure.

See the `Show Clipboard` command, described at the end of this section, for information about viewing the current Clipboard contents.

Paste (⌘V)

`Paste` is used to move a copy of a control from the Clipboard to a structure. A typical sequence of commands to make a copy of a control would be to `Copy` (or `Cut`) the control, `Paste` the control, and move it to its new location. This method can be used, for example, to quickly create a list of controls, like a list of radio buttons.

Note that dialog items can only be pasted into other dialogs, and that window controls can only be pasted into other windows. The pasted control is not removed from the Clipboard; therefore, multiple `Paste` commands can be executed.

Clear

`Clear` is used to remove a control from a structure. Select the control by clicking on it, and then remove it by issuing the `Clear` command.

Test run (⌘R)

This command is used to test the current structure. It gives you the opportunity to try out the structure as you create it, without having to write, compile, and run a program.

All of the controls, including those attached to a window frame (e.g. zoom boxes and grow boxes), will work exactly as they should in the finished program.

To end the test run, select the `End test run` command from the `Edit` menu. (This menu item replaces `Test Run` once testing begins.)

Design Master

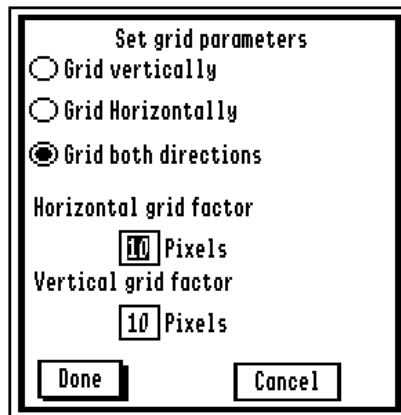
You can make any number of changes to a structure, and then test it, as many times as you want.

Turn Grid On/Off (⌘G)

Design Master provides a grid to aid in aligning controls, windows, and dialogs. The `Turn grid on/off` command toggles between bringing up and turning off the grid, which appears along the top and left side of the desktop. The default grid marks are set at 10 pixels. See the next command, `Set Grid...`, for a way to customize the grid.

Set Grid...

The command brings up a dialog box like the one below:



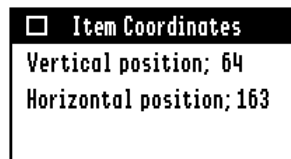
The radio buttons at the top of the dialog are to create a grid only along the right side of the desktop, only along the top of the desktop, or in both directions. The edit-line boxes in the center of the dialog let you specify the alignment factors, in pixels. Clicking on the `Done` button sets the grid parameters, while clicking on `Cancel` exits the `Set grid` dialog without affecting the original settings.

The table below gives pixel sizes (in pixels per inch) of most standard color monitors. Note that these values are approximate! They can vary from one monitor to the next.

<u>Screen Mode</u>	<u>Vertical size</u>	<u>Horizontal size</u>
640	33	86
320	33	43

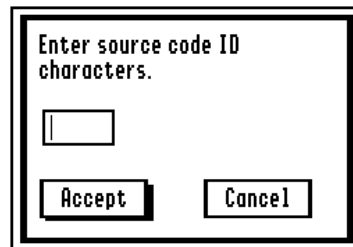
Show/Hide Coordinates

This command toggles between bringing up and closing a window showing the pixel coordinates of the selected control. (A control is selected by clicking on it.) The position displayed is relative to the structure in which the control resides (local coordinates), rather than to the desktop (global coordinates). An example of such a dialog is displayed below:



Source ID chars...

This command allows you to change the 4-character identification code attached to the source code definition of the current structure. It brings up the following dialog box:



Enter the new ID in the edit-line box, and then click on the `Accept` button to make the change, or click on `Cancel` to exit the dialog without changing the current source code ID. (See the "New" commands in the previous section of this chapter for more information about the source code ID.)

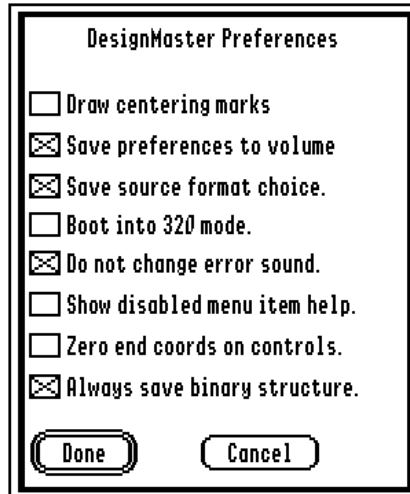
Switch to 320/640 mode

Screen format toggles the display between 640 by 200 pixels (the default) and 320 by 200 pixels.

Preferences...

This command allows you to customize a number of Design Master's features. It brings up this dialog box:

Design Master



The `Draw centering marks` option causes two lines to be painted onto the desktop, dividing it into four equal quadrants. This aids in aligning the windows and dialogs created with Design Master.

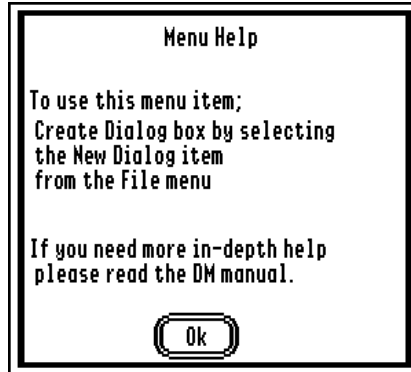
Selecting `Save preferences to volume` will create a disk file named `dm.prefs`, in the `SYSTEM` folder of the boot volume. The file will contain the preferences you select from this dialog. The next time that you run Design Master, the file will be read and your choices will be reflected.

`Save source format choice` sets the default source code format in the `Save as` dialog box. Design Master's default source code is APW/ORCA assembly language. If you select this option in the Preferences dialog, Design Master sets the new default when you save a structure to disk, using a source code format. The last source code saved before exiting the program will determine the default the next time you run Design Master.

Selecting `Boot into 320 mode` sets Design Master's default mode to the 320 by 200 pixel display. You must save the preferences to disk in order for Design Master to enter 320 mode. The mode will take effect the next time Design Master is launched.

The `error sound` option turns on an alternate error sound.

If you select `Show disabled menu item help`, selecting a disabled menu item will bring up a help screen that tells you how to access the item. For example, when creating a window, the `Static Text` command is not available. With this option selected, choosing `Static Text` would bring up this alert:



`Zero end coords on controls` sets the right and bottom control dimensions to zero. The toolkit will use the smallest value possible for these dimensions that still displays the entire control.

The `Always save binary structure` option provides an automatic file save feature. Upon closing a structure or quitting Design Master, the current structure is saved in Design Master's internal format.

You can select as many of the options as you wish.

Show/Hide Clipboard

`Show Clipboard` displays the current contents of the clipboard. Every item that is cut or copied is automatically placed into the Clipboard, replacing the last item that was stored there.

While the Clipboard is being displayed, the `Show Clipboard` command is changed to `Hide Clipboard`. This command is used to close the Clipboard window. You can also close the window by clicking in its close box, or selecting `Close` from the `File` menu.

The Structures Menu



This menu is available only when a window is being created; its commands are used to change the current window definition.

Design Master

Frame...

This command brings up the `New Window` dialog box, showing the selections that correspond to the current window. You can change any of the selections. Clicking the `Done` button will cause the current window to be updated to reflect the new parameters. Clicking `Cancel` will exit the `New Window` dialog, leaving the original window definition undisturbed. The functioning of the dialog differs slightly from that of the `New Window` command; selecting `Title` will not bring up the `Title` dialog, and selecting `Custom frame` will not bring up the `Frame` dialogs. The next two commands in this menu, `Title...` and `Colors...`, are used to change the title and window colors, respectively.

See the `New Window` command description, earlier in this chapter, for more information about its dialog box.

Title...

This command brings up the `New Window` command's `Title` dialog. To change the title, type the new name in the edit-line box and then click the `Accept` button. To abort the operation, click `Cancel`. You can remove the original title by simply clicking `Accept`, since the dialog's default title is the null string.

Colors...

`Colors` brings up the `New Window` command's `Custom frame colors` dialog box. You can change any of the colors in the original window definition by making the desired selections and then clicking the `OK` button. Selecting `Cancel` aborts the operation, leaving the original color choices intact.

See the `New Window` command, earlier in this chapter, for more information about coloring windows.

The Controls Menu

Controls	
Radio button...	⌘1
Push Button...	⌘2
Check Box...	⌘3
Edit Line...	⌘4
Static text...	⌘5
Long text...	⌘6
StatText...	⌘7
PopUp...	⌘8
List...	⌘9
Picture...	⌘0
Icon...	⌘-
TextEdit...	⌘=
Scroll...	⌘[
Grow box	⌘]

This menu is used to add controls to the current window or dialog. None of the controls can be added to menus.

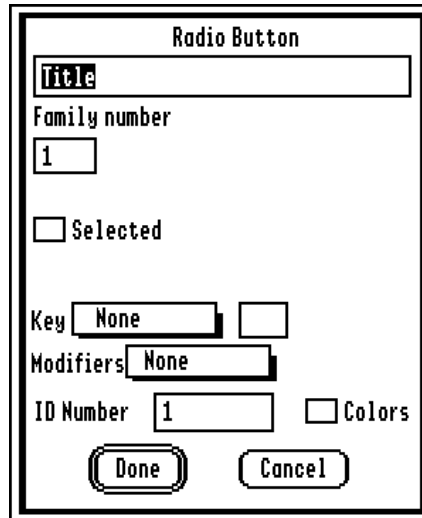
Radio button... (⌘1)

 Don't press this button!

This command is used to create a radio button control. Radio buttons are typically used to present the user with a series of simple choices, and all of the choices are mutually exclusive. An example of the use of radio buttons is in the `Save as` command's dialog box: the user can choose only one source code format, so that selecting one format excludes the possibility of selecting any of the other formats.

Radio buttons in windows are a little different from radio buttons in dialogs, so the options available also differ. If the radio button is being defined for a new window, it brings up the dialog:

Design Master



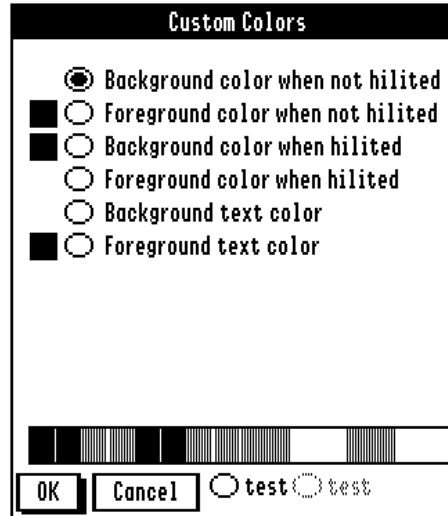
At the top of the dialog is an edit-line box where you can enter the text that is to appear to the right of the radio button.

Beneath the button's text is an edit-line box labeled `Family number`. Radio buttons are grouped by family number; only one member of the family may be "on" at any given time. The family number is limited to three digits; its value can range from 0 to 127.

Below the family number is a check box labeled `Selected`. Click in the box if the radio button's default setting is "on."

Pressing the mouse button on the `Key` option causes a pop-up menu to be displayed. This menu is used to assign a key equivalent for the radio button, so that the item can be selected either by clicking on it or by holding down the modifier key and then pressing the keyboard equivalent. From the `Key` pop-up, you can select the `Character` item to enter a character for the keyboard equivalent in the edit-line box next to the pop-up, or you can select the `ASCII code` item to enter a decimal value corresponding to the character's ASCII code. Valid ASCII values range from 0 to 255. The `None` item in the pop-up is the default, and is used to specify that no keyboard equivalent is to be used. After you've selected the keyboard equivalent, you can then select the key that is to be pressed simultaneously with the key. You select this key from the `Modifiers` pop-up below the `Key` pop-up.

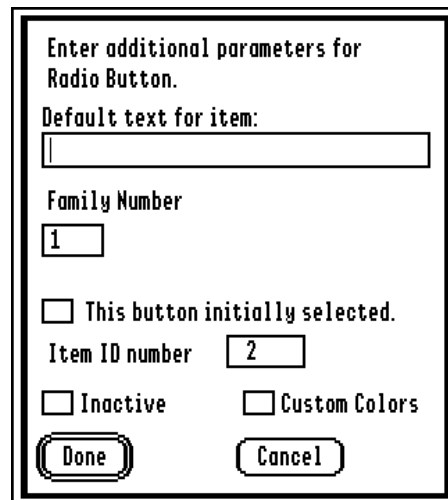
The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the radio button:



The two `Background color` fields correspond to the rectangle that encloses the radio button, while the `Foreground color` fields correspond to the circle in the center of the button. The `Background text` field indicates the rectangle that encloses the button's text, and the `Foreground text` field refers to the text itself.

Clicking the `OK` button will paint the radio button control. Selecting `Cancel` will exit the `Custom Colors` dialog without painting the control.

If the radio button is being defined for a new dialog, it brings up the dialog box:



Design Master

At the top of the dialog is an edit-line box where you can enter the text that is to appear to the right of the radio button.

Beneath the button's text is an edit-line box labeled `Family Number`. Radio buttons are grouped by family number; only one member of the family may be "on" at any given time. The family number is limited to three digits; its value can range from 0 to 127.

Below the family number is a check box labeled `This button initially selected`. Click in the box if the radio button's default setting is "on."

The next item in the dialog box is an edit-line box labeled `Item ID number`. Each item within a dialog box must have a unique item ID. Valid IDs range from 1 to 32767.

Selecting the `Inactive` check box will create an inactive radio button; mouse clicks in it will be completely ignored. Such a button will be painted as dimmed and unselectable.

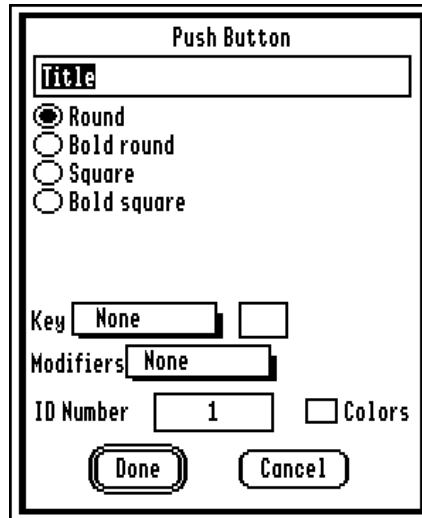
The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the radio button. This dialog, and its functioning, is identical to that described above for radio buttons within windows.

Push Button... (⌘2)

This command is used to create a push button control. Push buttons are also called simple buttons, because they consist of a rectangular box with a label on it. The action described by the label takes place when the button is "pushed," by clicking on it with the mouse or pressing the `RETURN` key if the button is the default button. Simple buttons come in two different shapes, round-cornered or square-cornered, and can be either singly or boldly outlined. According to Apple's *Human Interface Guidelines*, the default button is boldly outlined. Furthermore, the default button should never cause something to be destroyed (e.g. deleting a disk file). Two push buttons are shown below:



Push buttons in windows differ slightly from those used in dialogs, so the parameters required for the two structures are also different. If the push button is being defined for a new window, it brings up the dialog:



At the top of the dialog is an edit-line box where you can enter the text that is to appear within the push button.

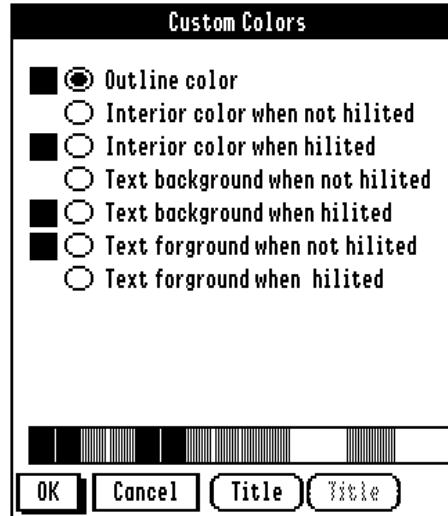
Beneath the button's text is a series of radio buttons that define the shape of the push button. A standard shape is selected by clicking on the desired button.

Pressing the mouse button on the `Key` option causes a pop-up menu to be displayed. This menu is used to assign a key equivalent for the button, so that the item can be selected either by clicking on it or by holding down the modifier key and then pressing the keyboard equivalent. From the `Key` pop-up, you can select the `Character` item to enter a character for the keyboard equivalent in the edit-line box next to the pop-up, or you can select the `ASCII code` item to enter a decimal value corresponding to the character's ASCII code. Valid ASCII values range from 0 to 255. The `None` item in the pop-up is the default, and is used to specify that no keyboard equivalent is to be used.

The next option in the dialog, the `Modifiers` pop-up, is used to select the key that must be pressed simultaneously with the keyboard equivalent in order to select the push button.

Below the `Modifiers` pop-up is an edit-line box labeled `ID Number`. You can use the box to assign a unique ID to the button.

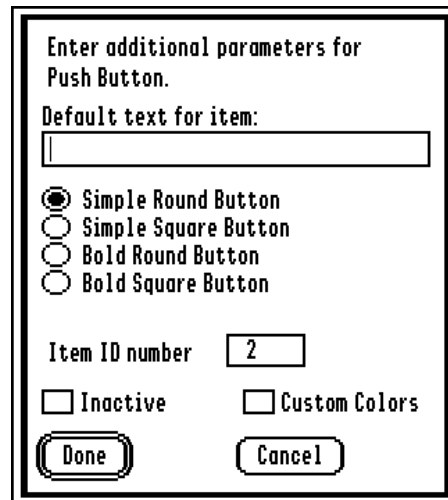
The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the push button:



The `Outline color` field corresponds to the button's outline. The two `Interior color` fields correspond to the interior of the button. The `Text background` fields indicate the rectangle that encloses the button's text, and the `Foreground text` fields refer to the text itself.

Clicking the `OK` button will paint the radio button control. Selecting `Cancel` will exit the `Custom Colors` dialog without painting the control.

If the push button is being defined for a new dialog, it brings up the dialog box:



At the top of the dialog is an edit-line box where you can enter the text that is to appear within the push button.

Beneath the button's text is a series of radio buttons that define the shape of the push button. A standard shape is selected by clicking on the desired button.

The next item in the dialog box is an edit-line box labeled `Item ID number`. Each item within a dialog box must have a unique item ID. Valid IDs range from 1 to 32767.

Selecting the `Inactive` check box will create an inactive push button; mouse clicks in it will be completely ignored. Such a button will be painted as dimmed and unselectable.

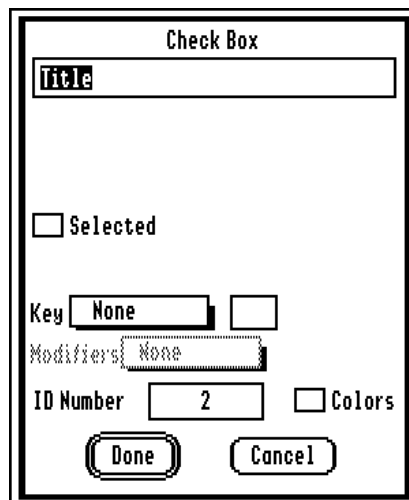
The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the push button. This dialog, and its functioning, is identical to that described above for push buttons within windows.

Check Box... (C3)

This command is used to create a check box control. Check boxes are used to select some option, generally when there are several options available, and none of which are mutually exclusive. Design Master's Preferences dialog box is a good example of the use of check boxes. Two all-too-typical check boxes are shown below:

☒ Pay most of your money in taxes
☐ Pay all of your money in taxes

Check boxes used in windows are a little different from those defined in dialogs, so that the parameters required by the two structures differ as well. If the check box is being defined for a new window, it brings up the dialog:



Design Master

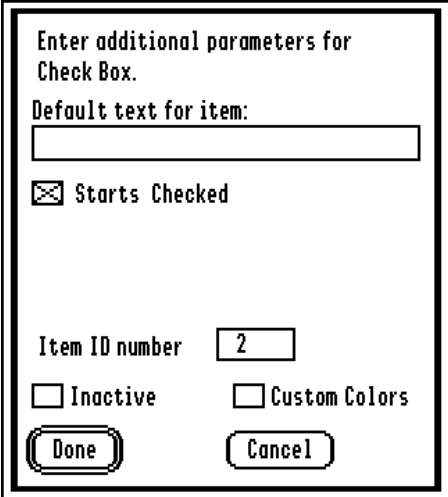
At the top of the dialog is an edit-line box where you can enter the text that is to appear to the right of the check box.

Beneath the edit-line box is a check box labeled `Selected`. Click on this box to make the check box' default setting be "selected."

Pressing the mouse button on the `Key` option causes a pop-up menu to be displayed. The menu is used to select a key that will have the same effect as clicking on the check box. Select its `Character` item to enter a character keyboard equivalent in the edit box next to the pop-up; select `ASCII code` to enter the ASCII code for the equivalent. If a key equivalent is selected for the check box, then you can use the next option in the dialog, `Modifiers`, to select the key that must be pressed simultaneously with the keyboard equivalent in order to select the check box.

The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the new control. This dialog box, as well as its functioning, is identical to that described for the `Radio button` command, presented earlier in this section.

If the check box is being defined for a new dialog, it brings up the dialog box:



Enter additional parameters for
Check Box.

Default text for item:

☒ Starts Checked

Item ID number

☐ Inactive ☐ Custom Colors

At the top of the dialog is an edit-line box where you can enter the text that is to appear to the right of the box.

Beneath the box' text is a check box labeled `Starts Checked`. Selecting this item causes the box' default value to be "selected."

The next item in the dialog box is an edit-line box labeled `Item ID number`. Each item within a dialog box must have a unique item ID. Valid IDs range from 1 to 32767.

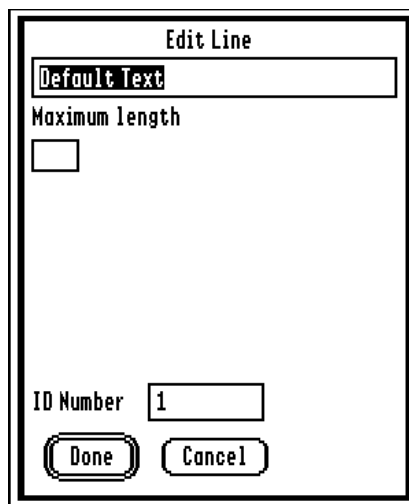
Selecting the `Inactive` check box will create an inactive check box; mouse clicks in it will be completely ignored. Such a box will be painted as dimmed and unselectable.

The final option in the dialog is a check box labeled `Custom Colors`. Selecting it, after clicking on the `Done` button, will bring up a dialog that lets you color the check box. This dialog, and its functioning, is identical to that described above for radio buttons within windows.

Edit Line... (C4)

This command is used to create an edit line control. Edit lines are used to receive input from the keyboard, such as the name of a file to save. The control is supported by the `LineEdit` tool set, which provides powerful editing capabilities, including cut, copy, and paste, freeing the programmer from having to implement these editing chores. Design Master's `Save as` command demonstrates the use of a typical edit-line control, in the box provided for the file name.

Edit lines used in windows differ slightly from those used in dialogs; therefore the options available in the two structures differ, too. If the edit line is being defined for a new window, it brings up the dialog:

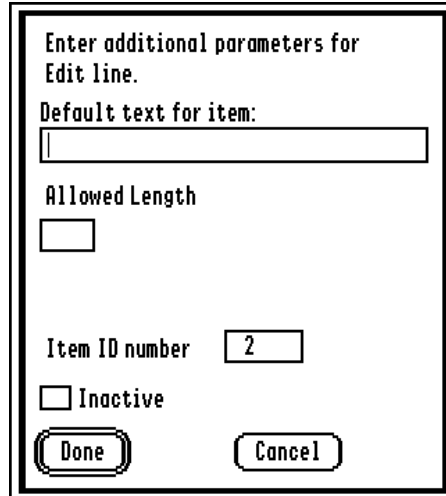


At the top of the dialog is an edit-line box where you can enter the default text that will appear within the edit-line box.

Beneath the box is an edit-line box labeled `Maximum length`. Enter the maximum number of characters that the edit line item can accept. Valid lengths range from 0 to 255.

If the edit line is being defined for a new dialog box, it brings up the dialog:

Design Master



At the top of the dialog is an edit-line box where you can enter the text that is to appear within the edit-line box.

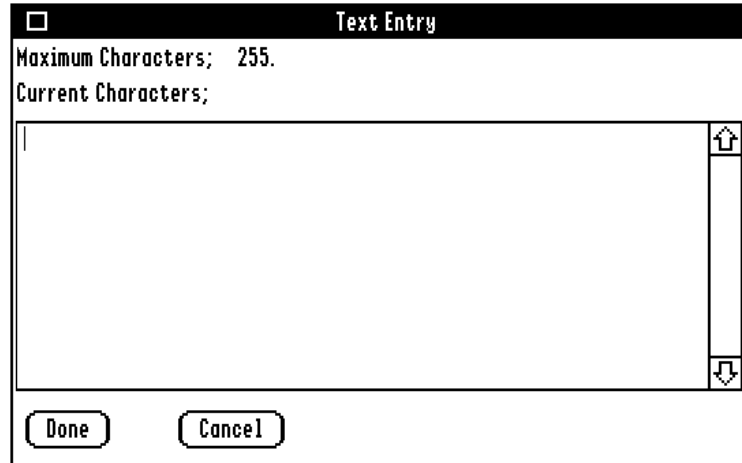
Beneath the box is an edit-line box labeled `Allowed Length`. Enter the maximum number of characters that the edit line item can accept. Valid lengths range from 0 to 255.

The next item in the dialog box is an edit-line box labeled `Item ID number`. Each item within a dialog box must have a unique item ID. Valid IDs range from 1 to 32767.

Selecting the `Inactive` check box will create an inactive edit line control; mouse clicks in it will be completely ignored. Such a control will be painted as dimmed and unselectable.

Static text... (⌘5)

This command is used to create a static text item in a dialog box. A static text item is typically a message that cannot be altered by the user; clicks on the item are ignored. An example of a static text item is the short message appearing at the top of all of Design Master's open file dialogs. The command brings up the dialog:



The dialog mainly consists of a large box where you can enter the static text. The text is limited to 255 characters. After entering the text, click on the `Done` button to create the item, or select `Cancel` to abort the command without creating the static text item.

Long text... (C6)

This command is used to create a long static text item in a dialog box. It brings up a dialog similar to the dialog for static text items. The only difference is that long text items can be up to 32767 characters long; the maximum characters field reflects this difference.

The dialog mainly consists of a large box where you can enter the static text. The text is limited to 32,767 characters. Beneath the text box is an edit-line box labeled `Maximum length`. Enter the maximum length of the item, as a decimal value in the range 0 to 32767. After filling in the boxes, click on the `Done` button to create the item, or select `Cancel` to abort the command without creating the long static text item.

StatText... (C7)

This command is used to create a static text control in a window. It brings up the same dialog used to enter text in a dialog.

The dialog mainly consists of a large edit-line box where you can enter the static text. The text is limited to 255 characters. After entering the text, click on the `Done` button to create the control, or select `Cancel` to abort the command without creating the control.

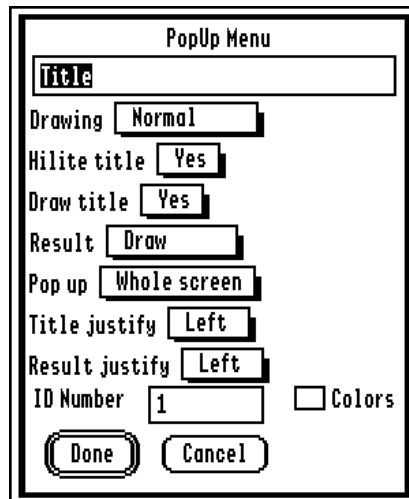
Design Master

PopUp... (⌘8)

This command is used to create a pop-up control in a window. Pop-up menus, like radio buttons, are used to present a series of choices. They are typically used in a crowded window, since the only space they occupy is the menu title and a display box. When the user selects the pop-up, its menu springs out of the display box. The `PopUp` command's dialog features several pop-up menus. The `Drawing` option of the dialog, when "popped," is depicted below:



`PopUp` brings up the dialog:



At the top of the dialog box is an edit-line box where you can enter the name of the pop-up menu.

The `Drawing` option is a pop-up menu containing the two items `Normal` and `White Space`. The option is used when the menu is confined to the window in which it appears, and therefore might need to be scrollable. Selecting `Normal` will cause the pop-up to display all of the items in its menu that it can, with a scrolling arrow shown at the top or bottom of the menu, as appropriate. The size of the menu may grow or shrink as needed, depending upon the current location in the menu, and the menu's position within the window. Selecting `White Space` causes the menu to always appear as the same size. Menu items that are currently not visible, because of scrolling, are filled in with white space.

Note that the `Drawing` option should be used in conjunction with the `Pop up` option, described below.

Chapter 5: Command Reference

The `Hilite title` option is a pop-up menu containing the two items `Yes` and `No`. Select `Yes` to cause the menu title to be highlighted when the pop-up control is selected; otherwise, select `No`.

Select `Yes` or `No` in the `Draw title` option's menu to have the pop-up control's title drawn or not drawn, respectively.

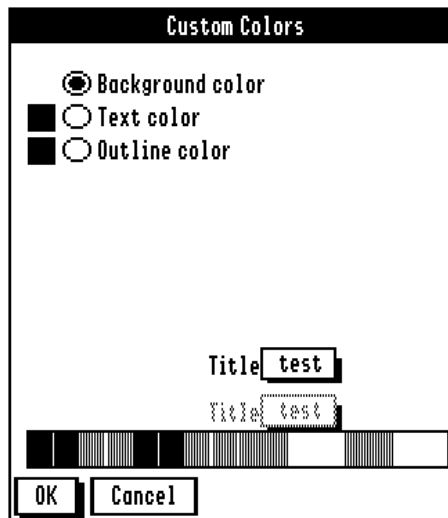
The `Result` option allows you to specify whether an item selected in the pop-up control's menu will be drawn in the pop-up's display rectangle. Select `Draw` in this option if the selected item is to be drawn in the rectangle, and `Don't draw` otherwise.

The `Pop up` option is used to allow the pop-up control's menu to overflow its window, or to confine it to the window, with a scrollable menu. Select `Whole screen` to allow overflow, and `Pin in window` otherwise.

`Title justify` is used to right- or left-justify the menu's title within its bounding rectangle.

`Result justify` is used to right- or left-justify a menu item within the pop-up's display rectangle.

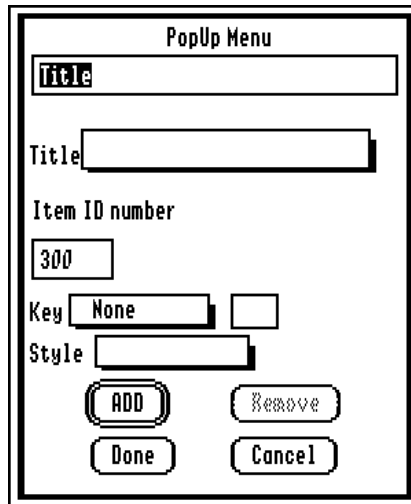
The final option in this dialog box, `Custom colors`, allows you to paint the pop-up control. If this check box is selected, it brings up this dialog, after clicking the `Done` button and adding all items to the pop-up menu:



The `Background color` option defines color of the menu title's bounding rectangle, as well as its display rectangle. The `Text color` option defines the color of all text in the menu, including the title and all items. The `Outline color` option defines the color used to paint the outline of the pop-up's display rectangle. The two pop-up controls beneath the radio buttons show how sample pop-ups looks, based on your current color selections. The top pop-up is for a selectable menu, while the bottom shows a dimmed pop-up.

Design Master

If you selected the `Done` button from the main `Pop up` dialog box, you are presented with a second dialog box that allows you to add items to the pop-up's menu:



At the top of the dialog is an edit-line box where you enter the name of the next item to be added to the pop-up menu.

Below the item name is the currently defined pop-up control. If you specified that the title is to be drawn, it will appear next to the pop-up's display rectangle. As items are added to the pop-up menu, they appear within this menu.

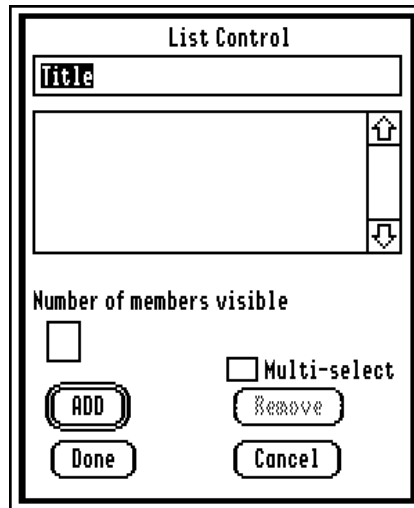
Below the pop-up control is another pop-up menu, labeled `Key`. Pop-up menu items can contain key equivalents; the items are limited to the keystroke combination of the open-Apple (⌘) key and a capital letter. To select a key equivalent for the current item, pop up the `Key` menu. Select its `Character` item to enter a character keyboard equivalent in the edit box next to the pop-up; select `ASCII code` to enter the ASCII code for the equivalent.

The next option in the items' dialog box is a pop-up menu labeled `Style`. This lets you select a type face for the item, and also provides the option of making the item a divider in the menu.

After you have made all the appropriate selections for the current item, click the `Add` button to add the item to the menu. An item can be deleted from the menu by selecting it from the pop-up control beneath the item name box, and then clicking the `Remove` button. After all items have been added to the pop-up control, click the `Done` button to exit the items' dialog, or select the `Cancel` button to exit the `Pop up` dialog without creating a pop-up control.

List... (⌘9)

This command is used to create a list control in a window. It brings up the dialog:



At the top of the dialog is an edit-line box where you can enter the next item in the list.

Beneath the item box is a window with a vertical scroll bar control. The items that are added to the list will be displayed in this window.

Below the current list display is an edit-line box labeled `Number of members visible`. Enter the number of list members that can be seen at any given time.

To the right of the `members visible` box is a check box labeled `Multi-select`. Select this box if the list should permit multiple selections. Leave the box unselected if the list should support single selections only. Enabling `Multi-select` allows the user to select more than one member at a time. Consecutive list members are selected by holding down the `SHIFT` key while selecting with the mouse. Random members can be selected by holding down the Apple (`⌘`) key while selecting with the mouse.

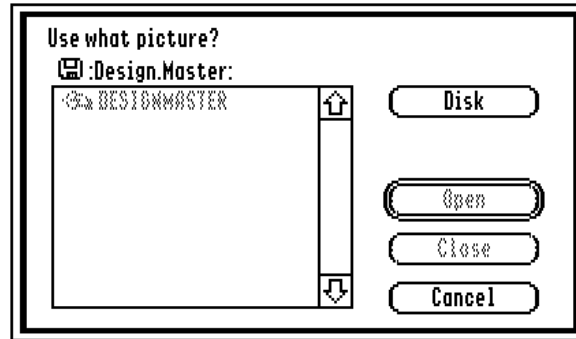
The `Add` button is used to add the next member to the list. A member can be deleted from the list by clicking on it in the current list display, and then clicking the `Remove` button.

Use the `Done` button to create the list control in the current window. Click the `Cancel` button to exit the `List` dialog without creating a list control.

Picture... (`⌘0`)

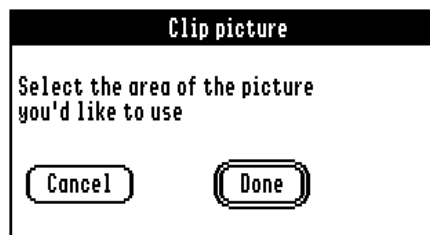
This command is used to create a picture control in a window. It brings up a Standard File open dialog similar to that depicted below:

Design Master



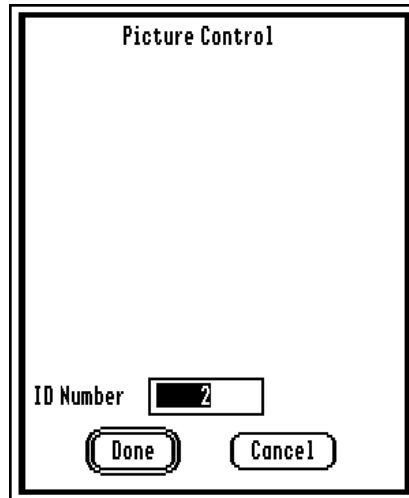
The current prefix display, file list, and Disk, Open, Close, and Cancel buttons all function as described in the section "Accessing Disk Files" at the beginning of this chapter. A file is selected for opening by clicking on its name and then clicking on the Open button, or by double-clicking its name.

After a picture file is opened, it appears in a window entitled `picture editor`, the mouse pointer turns into a selection tool, and this window appears:



You can move the dialog away from the picture's window, and use the selection tool to clip a portion of the picture. To use the new tool, click on the picture and drag the mouse. You will see a section of the picture selected. Click the Done button in the Clip picture dialog to create the picture control, or click Cancel to exit the Picture dialogs without creating a picture control.

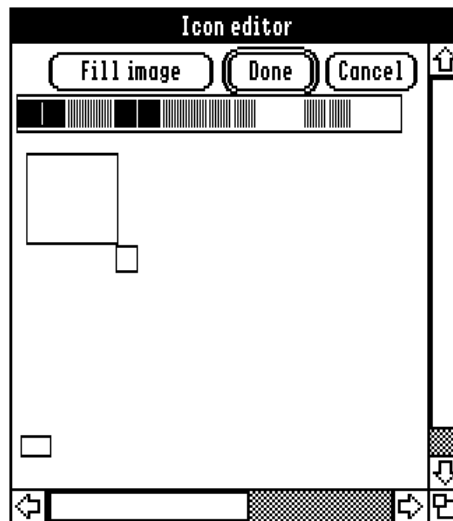
After clicking Done, a new dialog appears that asks for an ID number to assign to the picture control:



Enter the number you want to assign to the picture control, then click **Done** to complete the control creation. Click **Cancel** to exit the **Picture** dialogs without creating a picture control.

Icon... (🍏-)

This command is used to create a new icon. When you select this command, Design Master invokes its built-in icon editor:



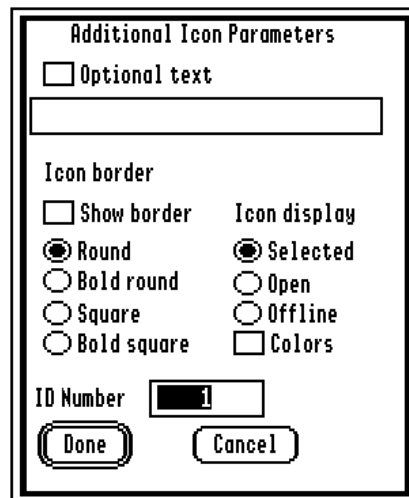
Design Master

The dialog contains three buttons, a color palette, a large box with a small square appended onto the lower right corner, and a small box. The large box is a pixel map that you paint to create the icon image, while the small box displays the actual image as it will appear in the window. The square appended to the large box is a sizing control. Drag it to grow or shrink the pixel map. The initial size of the map is eight by eight pixels.

The `Fill image` button is used to fill the entire map with a color. Select a color by clicking on it in the palette, and then click the `Fill image` button. Individual pixels can be painted by selecting a color, and then clicking on the pixel with the mouse. While the mouse is over the image, the pointer will change to a cross, which you use to center on a pixel.

Click the `Done` button to exit the icon editor after creating a new icon image. Click `Cancel` to exit the editor without creating a new icon.

After creating an icon, you are presented with another dialog that requests further definition of the icon:



The dialog box is titled "Additional Icon Parameters". It contains the following elements:

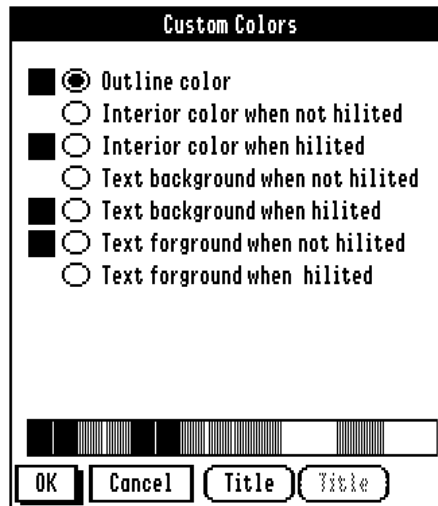
- A checkbox labeled "Optional text" with an empty text input field below it.
- A section titled "Icon border" containing four radio buttons: "Show border", "Round", "Bold round", "Square", and "Bold square".
- A section titled "Icon display" containing three radio buttons: "Selected", "Open", and "Offline", and a checkbox labeled "Colors".
- An "ID Number" field with a text input box containing the number "1".
- "Done" and "Cancel" buttons at the bottom.

If you wish text to appear below the icon, select the `Optional text` check box, and then enter the text in the edit-line box beneath the check box.

If the icon should be bordered, select the `Show border` check box and then select the type of border by clicking on the appropriate radio button below this check box.

The three radio buttons on the right side of the dialog are used to set the corresponding bits in the `displayMode` word when the icon is drawn. This word is one of the parameters passed to `QuickDraw` when the icon is actually drawn. `Selected` sets the `selectedIconBit` flag of the `displayMode` word; this causes the entire icon to be inverted. `Open` sets `openIconBit`, causing a light-grey pattern to be copied instead of the icon. `Offline` sets `offLineBit`, which adds a light-grey pattern with the icon as it is drawn.

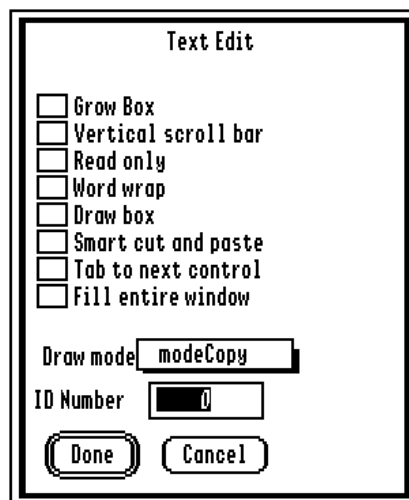
The final check box, `Custom Colors`, brings up a dialog box that allows you to color the icon's enclosing button:



Outline color refers to the button's outline. Interior color is the color of the interior of the button, not including the icon. Text background is the text's bounding rectangle. Text foreground is the text itself.

Text Edit... (⌘=)

This command is used to create a text edit control in a window. It brings up the dialog:



Design Master

You can select any combination of the check box items for the control. Selecting `Vertical scroll bar` will create a vertical scroll bar along the right edge of the edit text control's bounding rectangle. Selecting `Read only` prohibits editing of the text within the control. `Word wrap` is used to allow wrapping of the text (soft carriage returns) at the end of lines during text input into the control. `Draw box` causes the text of the control to appear with a border marking the edges of its bounding rectangle. Selecting `Smart cut and paste` causes support of "intelligent" cut and paste operations that take into account spacing around words. (Intelligent cut/paste is described in Chapter 25 of the *Toolbox Reference Manuals Update* document.) `Tab to next control` allows the user to press the tab key to move to the next control in the edit text window. `Fill entire window` sets the size of the box to the size of the window.

The `Draw mode` option provides a pop-up menu that allows you to specify the mode in which Quick Draw II will paint the text into the control's window. `modeCopy` copies the pixels directly into the window, ignoring the window's background pixels. `modeNotCopy` also performs a direct copy, but the text is inverted. The copy modes are the typical drawing modes. `modeOR` performs a nondestructive overlay of pixels onto the window. (The text pixels are ORed with the window's pixels.) `modeNotOR` also performs a nondestructive overlay, but the text pixel values are inverted before ORing with the background. `modeXOR` performs an exclusive OR operation of the text with the background. `modeNotXOR` also performs an XOR operation, but first inverts the text pixels before XORing them with the background. The XOR modes are typically used for cursor drawing and rubber banding. `modeBIC` (bit clear) ANDs the text pixels with the background. `modeNotBIC` also performs the AND operation, but inverts the text pixels before performing the AND. The BIC modes are generally used to clear an area before overlaying with another image.

If you did not select the `Fill entire window` option, you will need to place the control in the window. In this case, the following dialog appears:

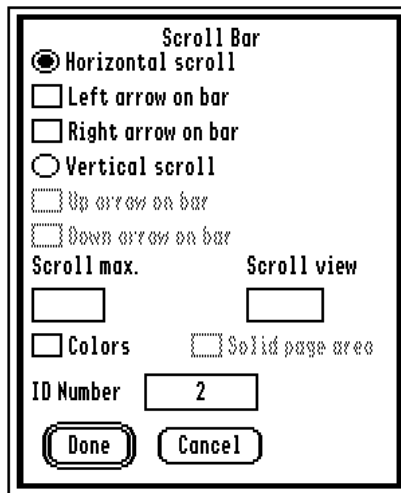


You can use the mouse to move this dialog out of the way of the window you are creating, if necessary, and then draw the edit text control into the new window. Do this by pressing the mouse button over the window, and then dragging the mouse while still keeping the button depressed. Release the mouse button when the text window is the size you want. If you make a mistake, simply redraw the text window; the old edit text window will vanish. Once the control is drawn, click the `Done` button to finish creation of the control, or click `Cancel` to abort the `TextEdit` command without creating the new control.

Scroll... (⌘L)

This command is used to create scroll bar controls in a window. Note that scroll bars are normally created as part of the window frame, as described in the `New window` command earlier in this chapter. The `Scroll` command allows you to create additional scroll bars in a window.

It brings up the dialog:



The two radio buttons at the top of the dialog box let you specify whether you're creating a horizontal scroll bar or a vertical scroll bar. The top two check boxes are used with the vertical scroll bar, while the two bottom check boxes are used with the horizontal scroll bar.

The `Scroll max.` edit-line box is used to specify the size of the area the scroll bar refers to. In a text screen, for example, this might be the number of lines. You can enter a decimal value in the range 0 to 65535. The `Scroll view` edit-line box is used to specify the size of the area that is visible in the window. You can enter a decimal value in the range 0 to 65535.

The last two check boxes allow you to color the scroll bar control. If you select `Custom Colors`, then the `Solid page area` box becomes selectable. Checking it will create a scroll control with a solid-colored page area, rather than the traditional dotted page.

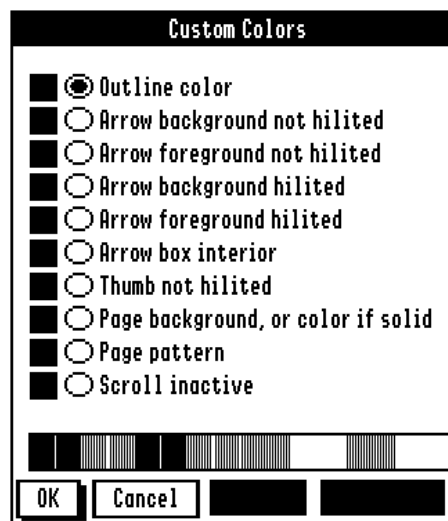
Click the `Done` button to continue creation of the scroll control, or click `Cancel` to exit the `Scroll` dialog without creating the control. Selecting `Done` changes the mouse pointer into a new tool and then brings up a dialog that informs you to position the scroll bar in the window:

Design Master



You can use the mouse to move this dialog out of the way of the window you are creating, if necessary, and then draw the scroll control into the new window. Do this by pressing the mouse button over the window, and then dragging the mouse while still keeping the button depressed. Release the mouse button when the scroll bar is the desired size and in the correct location. If you make a mistake, simply redraw the scroll bar; the old scroll will vanish when you begin dragging the mouse. Once the control is drawn, click the **Done** button to signal completion of this step, or click **Cancel** to abort the **Scroll** command without creating the new control.

If you selected **Custom Colors**, this dialog appears:



Outline color is the color of the outline of the rectangle that encloses the entire scroll bar, as well as the color of the outline of the arrow boxes and thumb. **Arrow background not hilited** is the rectangle enclosing the scrolling arrow, not including the arrow's outline. **Arrow background hilited** is the rectangle enclosing the arrow, not including the arrow's outline or interior. **Arrow foreground not hilited** is the outline of the arrow. **Arrow foreground hilited** is the color of the interior of the arrow when highlighted. **Arrow box interior** is the arrow's interior, not including the outline of the arrow. **Thumb not hilited** is the color of the

`thumb`. `Page background` refers to the color of the rectangle between the arrows. `Page pattern` is the color of the dots in the paging area, used when `Solid page` was not specified. `Scroll inactive` is the color of the scroll control when it has been made inactive.

Grow box... (⌘)

This command creates a grow box in the current window. Note that the grow box is normally created as part of the window frame, as described in the `New window` command earlier in this chapter. The `Grow box` command allows you to create additional size boxes in a window. After creating a new grow box, you can use the mouse to move it to any position in the window.

Appendix A

Complete MiniWord Program

This appendix contains the complete source code for the MiniWord text editor described in Chapter 3. Line numbers have been added to help you locate the part of the program discussed in a particular section of Chapter 3.

The complete source code for MiniWord can also be found on the Design Master disk. Please keep in mind that it is easier for us to change the program on disk than it is for us to change the manual. As the tools change and Design Master matures, there will eventually be minor differences between the source code you see here and the source code you find on the disk.

MAIN.PAS

```
1 {$keep 'main'}
2 program MiniWord;
3 {-----}
4 {
5 { MiniWord - A simple text editor
6 {
7 { Written by Barbara Allred and Design Master
8 {
9 { Copyright 1990
10 { Byte Works, Inc.
11 {
12 {-----}
13
14 uses
15   Common, MemoryMgr, ToolLocator, QuickDrawII, EventMgr, WindowMgr,
16   ControlMgr, MenuMgr, DialogMgr, DeskMgr, PrintMgr, IntegerMath,
17   TextEdit, GSOS;
18
19 {$LibPrefix '0/'}
20 uses
21   Globals, Error, Cmds1, Cmds2;
22
23 var
24   { *** DATA STRUCTURES GENERATED BY DESIGN MASTER, and altered by B.A. *** }
25   {
26   { Note: Design Master will generate the type "pString" instead of
27   {       "packed array of char." These have been changed to use
28   {       less space.
29
30   dropmenutitle01: packed array [0..20] of char;           {menu titles}
31   dropmenutitle02: packed array [0..20] of char;
32   dropmenutitle03: packed array [0..20] of char;
```

Design Master

```
33 dropmenutitle04: packed array [0..20] of char;
34
35 menu01itemtitle00: packed array [0..20] of char;           {menu item titles}
36
37 menu02itemtitle00: packed array [0..20] of char;
38 menu02itemtitle01: packed array [0..20] of char;
39 menu02itemtitle02: packed array [0..20] of char;
40 menu02itemtitle03: packed array [0..20] of char;
41 menu02itemtitle04: packed array [0..20] of char;
42 menu02itemtitle05: packed array [0..20] of char;
43 menu02itemtitle06: packed array [0..20] of char;
44 menu02itemtitle07: packed array [0..20] of char;
45
46 menu03itemtitle00: packed array [0..20] of char;
47 menu03itemtitle01: packed array [0..20] of char;
48 menu03itemtitle02: packed array [0..20] of char;
49
50 menu04itemtitle00: packed array [0..20] of char;
51
52 menu01: menuTemplate;
53 menu02: menuTemplate;
54 menu03: menuTemplate;
55 menu04: menuTemplate;
56
57 menu01item00: menuItemTemplate;
58 menu02item00: menuItemTemplate;
59 menu02item01: menuItemTemplate;
60 menu02item02: menuItemTemplate;
61 menu02item03: menuItemTemplate;
62 menu02item04: menuItemTemplate;
63 menu02item05: menuItemTemplate;
64 menu02item06: menuItemTemplate;
65 menu02item07: menuItemTemplate;
66 menu03item00: menuItemTemplate;
67 menu03item01: menuItemTemplate;
68 menu03item02: menuItemTemplate;
69 menu04item00: menuItemTemplate;
70
71
72 { Our data structures, global to main. }
73
74 masterID:      integer;                                {user ID passed by loader}
75 startStopAddr: longint;
76 startStopRec:  startStopRecord;                        {GS/OS 5.0 1-stop load tools call}
77
78
79 function MyWindow (var index: integer; var currWindow: grafPortPtr;
80                   numWindows: integer): boolean; forward;
81
82 (* *****
83 *
84 * DoQuit - Handle Quit command.
85 *
86 * ***** *)
87
```

Appendix A: Complete MiniWord Program

```

88 procedure DoQuit (var done: boolean; var index, numWindows: integer;
89                   var currWindow: grafPortPtr);
90 begin
91   while (MyWindow (index, currWindow, numWindows)) do {close all open windows}
92     DoClose (index, currWindow, numWindows);
93
94   done := true; {set the done flag}
95 end;
96
97
98 { Forward declarations of local subroutines that EventLoop calls. }
99
100 procedure HandleMenu (menuData: longint; var index, numWindows: integer;
101                      var currWindow: grafPortPtr; var done: boolean); forward;
102
103 procedure HandleSpecial (menuData: longint; var index, numWindows: integer;
104                        var currWindow: grafPortPtr); forward;
105
106 procedure HandleUpdate (theWindow: grafPortPtr); forward;
107
108 (*****
109 *
110 * EventLoop - Get next event, then dispatch the
111 *               appropriate routine to handle it.
112 *
113 *****)
114
115 procedure EventLoop;
116
117 {const} {Event codes returned by TaskMaster}
118
119 { inButtDwn      = $0001; } {button down event}
120 { mouseUpEvt     = $0002; } {button up event}
121 { inKey          = $0003; } {keystroke event}
122 { autoKeyEvt     = $0005; } {auto key event: key held down by user}
123 { inUpdate       = $0006; } {update event}
124 { activateEvt    = $0008; } {activate event}
125 { switchEvt      = $0009; } {switch event}
126 { deskAccEvt     = $000A; } {desk accessory event}
127 { driverEvt      = $000B; } {driver event}
128 { applEvt        = $000C; } {application 1 event}
129 { app2Evt        = $000D; } {application 2 event}
130 { app3Evt        = $000E; } {application 3 event}
131 { app4Evt        = $000F; } {application 4 event}
132 { wInDesk        = $0010; } {On Desktop}
133 { wInMenuBar     = $0011; } {On system menu bar}
134 { wClickCalled   = $0012; } {system click called}
135 { wInContent     = $0013; } {In content region}
136 { wInDrag        = $0014; } {In drag region}
137 { wInGrow        = $0015; } {In grow region, active window only}
138 { wInGoAway      = $0016; } {In go-away region, active window only}
139 { wInZoom        = $0017; } {In zoom region, active window only}
140 { wInInfo        = $0018; } {In information bar}
141 { wInSpecial     = $0019; } {Item ID selected was 250 - 255}
142 { wInDeskItem    = $001A; } {Item ID selected was 1 - 249}

```

Design Master

```

143 { wInFrame      = $001B; }      {in Frame, but not on anything else }
144 { wInactMenu    = $001C; }      {"selection" of inactive menu item }
145 { wClosedNDA    = $001D; }      {desk accessory closed }
146 { wCalledSysEdit = $001E; }      {inactive menu item selected }
147 { wTrackZoom    = $001F; }      {zoom box clicked, but not selected }
148 { wHitFrame     = $0020; }      {button down on frame, made active }
149 { wInControl    = $0021; }      {button or keystroke in control }
150 { wInSysWindow  = $8000; }      {hi bit set for system windows }
151
152 var
153   numWindows: integer;           {# of currently open windows }
154   index:      integer;           {window arrays index, active window }
155   currWindow: grafPortPtr;       {current active window }
156
157   taskRecord: wmTaskRec;         {used to communicate with TaskMaster}
158   eventCode:  integer;           {returned by TaskMaster }
159   i:          integer;
160
161 begin
162   done := false;                {we ain't done yet }
163   for i := 0 to 3 do             {no window open yet}
164     windowOpen[i] := noWindow;
165   numWindows := 0;
166   index := 0;
167   taskRecord.taskMask := $001FBFFF; {let TaskMaster do almost everything }
168
169   while not (done) do begin      {execute event loop 'til we're done}
170
171     { Call TaskMaster to get next event we need to handle. }
172
173     eventCode := TaskMaster ($076E, {event mask = just about everything}
174                               taskRecord); {pointer to extended task record }
175     errNum := ToolError;
176     if errNum <> 0 then            {only error possible}
177       HandleError (errNum, fatalErr, stopAlertTyp) { is messing up }
178                                                       { wmTaskMask field }
179
180     { Window update event? }
181
182     else if eventCode = inUpdate then
183       HandleUpdate (grafPortPtr (taskRecord.taskData))
184
185
186     { Cut, copy, paste, or close command? }
187
188     else if eventCode = wInSpecial then
189       HandleSpecial (taskRecord.taskData, index, numWindows, currWindow)
190
191
192     { Did user click in close box? }
193
194     else if eventCode = wInGoAway then begin
195       if MyWindow (index, currWindow, numWindows) then
196         DoClose (index, currWindow, numWindows)
197     end

```

Appendix A: Complete MiniWord Program

```
198
199
200   { Non-special command? }
201
202   else if eventCode = wInMenuBar then
203       HandleMenu (taskRecord.taskData, index, numWindows, currWindow, done)
204
205   end {while not done}
206
207 end; {EventLoop}
208
209 (*****
210 *
211 * HandleMenu - Handle menu selections, menu IDs
212 *           256 ->.
213 *
214 *****)
215
216 procedure HandleMenu (* menuData: longint; var index, numWindows: integer;
217                      var currWindow: grafPortPtr; var done: boolean *);
218
219 type
220   menuIDs = 256..264;
221
222 var
223   theItem: menuIDs;
224   menuNum: integer;
225
226 begin
227   { Menu item ID is in low-order word of the taskData field of our task record. }
228   { Menu ID is in the high-order word of the field. }
229
230   theItem := convert (menuData) .lsw;
231   menuNum := convert (menuData) .msw;
232
233   { Dispatch the appropriate routine, based on menu item selected. }
234
235   case theItem of
236     aboutID:    DoAbout;
237
238     newID:      DoNew (index, numWindows);
239
240     openID:     DoOpen (index, numWindows);
241
242     saveID:     if MyWindow (index, currWindow, numWindows) then
243                   DoSave (index, currWindow);
244
245     saveAsID:   if MyWindow (index, currWindow, numWindows) then
246                   DoSaveAs (index, currWindow);
247
248     pSetUpID:   DoPSetUp;
249
250     printID:    if MyWindow (index, currWindow, numWindows) then
251                   DoPrint (index, currWindow);
252
```

Design Master

```
253   quitID:      DoQuit (done, index, numWindows, currWindow);
254
255   findID:      DoFind;
256   end;
257
258
259 { Unhighlight the menu they just pulled down. }
260
261 HiliteMenu (false, menuNum);
262 end;
263
264 (*****
265 *
266 * HandleSpecial - Handle special menu commands,
267 *                 menu IDs 250 - 255.
268 *
269 *****)
270
271 procedure HandleSpecial (* menuData: longint; var index, numWindows: integer;
272                        var currWindow: grafPortPtr *);
273
274 var
275     theItem: integer;
276     menuNum: integer;
277
278 begin
279 { Menu item ID is in low-order word of the taskData field of our task record. }
280 { Menu ID is in the high-order word of the field. }
281
282 theItem := convert (menuData) .lsw;
283 menuNum := convert (menuData) .msw;
284
285 { Close command selected? All other special items handled by TaskMaster. }
286 if theItem = closeID then
287     if MyWindow (index, currWindow, numWindows) then
288         DoClose (index, currWindow, numWindows);
289
290
291 { Unhighlight the menu they just pulled down. }
292
293 HiliteMenu (false, menuNum);
294 end; {HandleSpecial}
295
296 (*****
297 *
298 * HandleUpdate - Handle update event for active
299 *                 window.
300 *
301 *****)
302
303 procedure HandleUpdate (* theWindow: grafPortPtr *);
304
305 begin
306 BeginUpdate (theWindow);
307 DrawControls (theWindow);
```


Appendix A: Complete MiniWord Program

```

308 EndUpdate      (theWindow);
309 end;
310
311 procedure InitMenus; forward;
312 (*****
313 *
314 * Init - Initialize all global variables, start
315 *       the tools we need, create our menu bar.
316 *
317 * Output:
318 *       Returns true if everything started OK,
319 *       and false otherwise.
320 *
321 *****)
322
323 function Init: boolean;
324
325 label 99;
326
327 var
328     ok: boolean;
329
330 begin
331 { Initialize all global variables. }
332
333 Init      := true;                {assume all is well to start with }
334 masterID := userID;              {get user ID passed by loader }
335 myID     := masterID | $0200;    {alter for our purposes, so we can do}
336                                     { a simple DisposeAll at the end }
337
338 { Start tools we need: Use GS/OS 5.0 one-call startup mechanism. }
339 with startStopRec do begin
340     flags      := 0;                {flags must be zero }
341     videoMode  := $80;              {640 mode }
342     numTools   := 20;              {we'll start 20 tools}
343     toolArray [1].toolNumber := $03;
344     toolArray [1].minVersion := $0300;    {Miscellaneous Toolset}
345     toolArray [2].toolNumber := $04;
346     toolArray [2].minVersion := $0300;    {Quick Draw II}
347     toolArray [3].toolNumber := $06;
348     toolArray [3].minVersion := $0300;    {Event Manager}
349     toolArray [4].toolNumber := $0E;
350     toolArray [4].minVersion := $0300;    {Window Manager}
351     toolArray [5].toolNumber := $10;
352     toolArray [5].minVersion := $0300;    {Control Manager}
353     toolArray [6].toolNumber := $0F;
354     toolArray [6].minVersion := $0300;    {Menu Manager}
355     toolArray [7].toolNumber := $14;
356     toolArray [7].minVersion := $0100;    {LineEdit Toolset}
357     toolArray [8].toolNumber := $15;
358     toolArray [8].minVersion := $0100;    {Dialog Manager}
359     toolArray [9].toolNumber := $08;
360     toolArray [9].minVersion := $0100;    {Sound Manager}
361     toolArray [10].toolNumber := $17;
362     toolArray [10].minVersion := $0100;    {Standard File Operations Toolset}

```

Design Master

```
363  toolArray [11].toolNumber := $16;
364  toolArray [11].minVersion := $0104;      {Scrap Manager}
365  toolArray [12].toolNumber := $09;
366  toolArray [12].minVersion := $0100;      {Apple Desktop Bus Toolset}
367  toolArray [13].toolNumber := $05;
368  toolArray [13].minVersion := $0100;      {Desk Manager}
369  toolArray [14].toolNumber := $1C;
370  toolArray [14].minVersion := $0100;      {List Manager}
371  toolArray [15].toolNumber := $1B;
372  toolArray [15].minVersion := $0204;      {Font Manager}
373  toolArray [16].toolNumber := $13;
374  toolArray [16].minVersion := $0100;      {Print Manager}
375  toolArray [17].toolNumber := $12;
376  toolArray [17].minVersion := $0206;      {Quick Draw II Auxiliary}
377  toolArray [18].toolNumber := $0A;
378  toolArray [18].minVersion := $0100;      {SANE Toolset}
379  toolArray [19].toolNumber := $0B;
380  toolArray [19].minVersion := $0100;      {Integer Math Toolset}
381  toolArray [20].toolNumber := $22;
382  toolArray [20].minVersion := $0100;      {Text Edit Toolset}
383  end;
384  startStopAddr := StartUpTools (masterID, pointerVerb, ord4 (@startStopRec));
385  errNum := ToolError;
386  if errNum <> 0 then begin
387    Init := false;
388    goto 99;
389  end;
390
391
392  { Initialize our units. }
393
394  okTitle := 'OK';
395  cancelTitle := 'Cancel';
396  InitError;
397  if not (InitCmds1) then begin
398    Init := false;
399    goto 99;
400  end;
401  if not (InitCmds2) then begin
402    Init := false;
403    goto 99;
404  end;
405  InitMenus;
406
407  99:
408  InitCursor;      {StartUpTools brings up the watch cursor}
409                  { so change it to the arrow }
410  end;
411
```

Appendix A: Complete MiniWord Program

```

412 (*****
413 *
414 * InitMenus - Initialize the menu bar.
415 *
416 *****)
417
418 procedure InitMenus;
419
420 var
421   tmp: longint;
422   i: integer;
423
424 begin
425   { *** GENERATED BY DESIGN MASTER, with comments provided by B.A. *** }
426
427   dropmenutitle01 := '@';                                     {Apple menu}
428   with menu01 do begin
429     version      := 0;
430     menuID       := $0001;
431     menuFlag     := $0008;                                     {cache menu; will pass ptr to title}
432     menuTitleRef := ord4 (@dropMenuTitle01);                 {pointer to menu's title}
433     itemRefs [1] := ord4 (@menu01item00);                     {About item reference }
434     itemRefs [2] := 0;                                         {null terminator      }
435   end;
436
437   menu01itemtitle00 := 'About';
438   with menu01item00 do begin
439     version      := 0;
440     itemID       := 256;                                       {About ID              }
441     itemChar     := $00;                                       {shortcut characters   }
442     itemAltChar  := $00;
443     itemCheck    := $0000;
444     itemFlag     := $0041;                                     {bold, divider beneath, will}
445                                     { pass pointer to title  }
446     itemTitleRef := ord4 (@menu01itemtitle00);               {pointer to item's name}
447   end;
448
449   dropmenutitle02 := ' File ';
450   with menu02 do begin
451     version      := 0;
452     menuID       := $0002;
453     menuFlag     := $0008;                                     {cache menu; will pass ptr to title}
454     menuTitleRef := ord4 (@dropMenuTitle02);                 {pointer to menu's title }
455     itemRefs [1] := ord4 (@menu02item00);                     {item reference: New     }
456     itemRefs [2] := ord4 (@menu02item01);                     {item reference: Open    }
457     itemRefs [3] := ord4 (@menu02item02);                     {item reference: Close   }
458     itemRefs [4] := ord4 (@menu02item03);                     {item reference: Save    }
459     itemRefs [5] := ord4 (@menu02item04);                     {item reference: Save as }
460     itemRefs [6] := ord4 (@menu02item05);                     {item reference: Page setup}
461     itemRefs [7] := ord4 (@menu02item06);                     {item reference: Print   }
462     itemRefs [8] := ord4 (@menu02item07);                     {item reference: Quit    }
463     itemRefs [9] := 0;                                         {null terminator        }
464   end;
465

```

Design Master

```
466 menu02itemtitle00 := 'New';
467 with menu02item00 do begin
468   version      := 0;
469   itemID        := 257;           {new ID          }
470   itemChar      := $4E;           {shortcut characters}
471   itemAltChar   := $6E;
472   itemCheck     := $0000;
473   itemFlag      := $0000;         {will pass pointer to title}
474   itemTitleRef  := ord4 (@menu02itemtitle00); {pointer to item's name  }
475 end;
476
477 menu02itemtitle01 := 'Open...';
478 with menu02item01 do begin
479   version      := 0;
480   itemID        := 258;           {open ID          }
481   itemChar      := $4F;           {shortcut characters}
482   itemAltChar   := $6F;
483   itemCheck     := $0000;
484   itemFlag      := $0000;         {will pass pointer to title}
485   itemTitleRef  := ord4 (@menu02itemtitle01); {pointer to item's name  }
486 end;
487
488 menu02itemtitle02 := 'Close';
489 with menu02item02 do begin
490   version      := 0;
491   itemID        := 255;           {close ID         }
492   itemChar      := $57;           {shortcut characters}
493   itemAltChar   := $77;
494   itemCheck     := $0000;
495   itemFlag      := $0000;         {will pass pointer to title}
496   itemTitleRef  := ord4 (@menu02itemtitle02); {pointer to item's name  }
497 end;
498
499 menu02itemtitle03 := 'Save';
500 with menu02item03 do begin
501   version      := 0;
502   itemID        := 259;           {save ID          }
503   itemChar      := $53;           {shortcut characters}
504   itemAltChar   := $73;
505   itemCheck     := $0000;
506   itemFlag      := $0000;         {will pass pointer to title}
507   itemTitleRef  := ord4 (@menu02itemtitle03); {pointer to item's name  }
508 end;
509
510 menu02itemtitle04 := 'Save as...';
511 with menu02item04 do begin
512   version      := 0;
513   itemID        := 260;           {save as ID       }
514   itemChar      := $00;           {shortcut characters}
515   itemAltChar   := $00;
516   itemCheck     := $0000;
517   itemFlag      := $0040;         {will pass pointer to title}
518                                     { draw divider beneath }
519   itemTitleRef  := ord4 (@menu02itemtitle04); {pointer to item's name  }
520 end;
```

Appendix A: Complete MiniWord Program

```

521
522 menu02itemtitle05 := 'Page setup...';
523 with menu02item05 do begin
524   version      := 0;
525   itemID       := 261;                      {page setup ID      }
526   itemChar     := $00;                      {shortcut characters}
527   itemAltChar  := $00;
528   itemCheck    := $0000;
529   itemFlag     := $0000;                    {will pass pointer to title}
530   itemTitleRef := ord4 (@menu02itemtitle05); {pointer to item's name  }
531 end;
532
533 menu02itemtitle06 := 'Print...';
534 with menu02item06 do begin
535   version      := 0;
536   itemID       := 262;                      {print ID          }
537   itemChar     := $50;                      {shortcut characters}
538   itemAltChar  := $70;
539   itemCheck    := $0000;
540   itemFlag     := $0040;                    {will pass pointer to title}
541   { divider beneath }
542   itemTitleRef := ord4 (@menu02itemtitle06); {pointer to item's name  }
543 end;
544
545 menu02itemtitle07 := 'Quit';
546 with menu02item07 do begin
547   version      := 0;
548   itemID       := 263;                      {quit ID           }
549   itemChar     := $51;                      {shortcut characters}
550   itemAltChar  := $71;
551   itemCheck    := $0000;
552   itemFlag     := $0000;                    {will pass pointer to title}
553   itemTitleRef := ord4 (@menu02itemtitle07); {pointer to item's name  }
554 end;
555
556 dropmenutitle03 := 'Edit';
557 with menu03 do begin
558   version      := 0;
559   menuID       := $0003;
560   menuFlag     := $0008;                    {cache menu; will pass ptr to title}
561   menuTitleRef := ord4 (@dropmenutitle03);  {pointer to menu's title  }
562   itemRefs [1] := ord4 (@menu03item00);     {item reference: Cut      }
563   itemRefs [2] := ord4 (@menu03item01);     {item reference: Copy     }
564   itemRefs [3] := ord4 (@menu03item02);     {item reference: Paste    }
565   itemRefs [4] := 0;                        {null terminator         }
566 end;
567
568 menu03itemtitle00 := 'Cut';
569 with menu03item00 do begin
570   version      := 0;
571   itemID       := 251;                      {cut ID            }
572   itemChar     := $58;                      {shortcut characters}
573   itemAltChar  := $78;
574   itemCheck    := $0000;
575   itemFlag     := $0000;                    {will pass pointer to title}

```

Design Master

```

576 itemTitleRef := ord4 (@menu03itemtitle00);      {pointer to item's name  }
577 end;
578
579 menu03itemtitle01 := 'Copy';
580 with menu03item01 do begin
581   version      := 0;
582   itemID        := 252;                          {copy ID          }
583   itemChar      := $43;                          {shortcut characters}
584   itemAltChar   := $63;
585   itemCheck     := $0000;
586   itemFlag      := $0000;                        {will pass pointer to title}
587   itemTitleRef  := ord4 (@menu03itemtitle01);    {pointer to item's name  }
588 end;
589
590 menu03itemtitle02 := 'Paste';
591 with menu03item02 do begin
592   version      := 0;
593   itemID        := 253;                          {paste ID         }
594   itemChar      := $56;                          {shortcut characters}
595   itemAltChar   := $76;
596   itemCheck     := $0000;
597   itemFlag      := $0000;                        {will pass pointer to title}
598   itemTitleRef  := ord4 (@menu03itemtitle02);    {pointer to item's name  }
599 end;
600
601 dropmenutitle04 := ' Search ';
602 with menu04 do begin
603   version      := 0;
604   menuID        := $0004;
605   menuFlag      := $0008;                        {cache menu; will pass ptr to title}
606   menuTitleRef  := ord4 (@dropmenutitle04);      {pointer to menu's title  }
607   itemRefs [1] := ord4 (@menu04item00);          {Find item reference      }
608   itemRefs [2] := 0;                             {null terminator         }
609 end;
610
611 menu04itemtitle00 := 'Find...';
612 with menu04item00 do begin
613   version      := 0;
614   itemID        := 264;                          {find ID          }
615   itemChar      := $46;                          {shortcut characters}
616   itemAltChar   := $66;
617   itemCheck     := $0000;
618   itemFlag      := $0000;                        {will pass pointer to title}
619   itemTitleRef  := ord4 (@menu04itemtitle00);    {pointer to item's name  }
620 end;
621
622 { The rest of the code is this procedure is ours. }
623 { Create the menu bar. Start at the last menu, since we're inserting each new }
624 { menu at the front of the current menu list. }
625
626 tmp := ord4 (@menu04); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
627 tmp := ord4 (@menu03); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
628 tmp := ord4 (@menu02); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
629 tmp := ord4 (@menu01); InsertMenu (NewMenu2 (pointerVerb, tmp), 0);
630

```

Appendix A: Complete MiniWord Program

```

631 FixAppleMenu (1);           {add desk accessories to Apple menu      }
632 i := FixMenuBar;           {compute standard sizes for menu bar and menus}
633                               { throw away returned height              }
634 DrawMenuBar;
635
636 end; {InitMenus}
637
638 (*****
639 *
640 * MyWindow - Checks if front window is one of ours.
641 *
642 * Output:
643 *       true if it's one of ours; false otherwise
644 *
645 *****)
646
647 function MyWindow (* var index: integer; var currWindow: grafPortPtr;
648                   numWindows: integer): boolean *);
649
650 label 99;
651
652 var
653   systemWind: boolean;           {true if the front window is a system window}
654   tmp:       grafPortPtr;
655   tmp2:      longint;
656
657 begin
658   MyWindow := true;               {assume the front window is ours}
659
660 { First check if any windows are open. }
661
662 if numWindows = 0 then begin
663   MyWindow := false;
664   goto 99;
665 end;
666
667 { Now check if the window is one of ours. }
668
669 tmp := FrontWindow;           {get grafPort pointer of active window}
670 systemWind := GetSysWFlag (tmp);           {active window belong to desk }
671                               { accessory? }
672
673 if systemWind then begin           {Yes - MyWindow is false, exit}
674   MyWindow := false;
675   goto 99;
676 end;
677
678 { It's one of ours, so get index into window arrays from wRefCon field. }
679
680 tmp2 := GetWRefCon (tmp);
681 index := convert (tmp2) .lsw;
682 currWindow := tmp;
683
684 99:
685 end;

```

Design Master

```
686
687 (*****
688 *
689 * ShutDown - Unload the tools we started.
690 *
691 *****)
692
693 procedure ShutDown;
694
695 begin
696   DisposeAll (myID);                                {dispose of all memory we allocated}
697   ShutDownTools (pointerVerb, startStopAddr);        {shut down tools we started}
698   errNum := ToolError;
699   if errNum <> 0 then
700     HandleError (errNum, fatalErr, stopAlertTyp);
701 end;
702
703
704 (*****
705 *
706 * Main program
707 *
708 *****)
709
710 begin
711
712   if Init then                                       {start tools, bring up menu bar}
713     EventLoop;                                     {execute main event loop      }
714   ShutDown;                                         {unload tools                  }
715
716 end.
```

CMDS1.PAS

```
1 {$keep 'Cmds1'}
2 unit Cmds1;
3 {-----}
4 {
5 { Cmds1 - Handles the menus commands About, Close,
6 {           Find, Page setup, and Print.
7 {
8 { Written by Barbara Allred and Design Master
9 {
10 { Copyright 1990
11 { Byte Works, Inc.
12 {
13 {-----}
14 interface
15
16 uses
17   Common, MemoryMgr, QuickDrawII, WindowMgr, ControlMgr, MenuMgr, DialogMgr,
18   PrintMgr, TextEdit;
19
```


Appendix A: Complete MiniWord Program

```
20 {$LibPrefix '0/'}
21 uses
22   Globals, Error;
23
24
25 { Subroutines that can be called from outside of the Cmdsl unit. }
26
27 procedure DoAbout;
28 { Handle the About menu item. }
29
30 procedure DoClose (index: integer; currWindow: grafPortPtr;
31                   var numWindows: integer);
32 { Close the front window, checked if it's ours before passing currWindow to us }
33
34 procedure DoFind;
35 { Handle the Find menu item. }
36
37 procedure DoPrint (index: integer; currWindow: grafPortPtr);
38 { Print a document. }
39
40 procedure DoPSetUp;
41 { Allow user to set some printer options. }
42
43 function InitCmdsl: boolean;
44 { Initialize the Cmdsl unit. }
45
46
47 { The rest of the file is private to the Cmdsl unit. }
48 implementation
49
50 var
51   { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
52
53   aboutDlg:          dialogTemplate;           {About dialog box}
54   ITEM00Abt1:        itemTemplate;
55   ITEM01Abt1:        itemTemplate;
56   item00pointerAbt1: packed array [0..100] of char;
57   item01colorsAbt1:  colorTable;
58
59   saveAlert:         alertTemplate;           {WantToSave alert box}
60   ITEM00Save1:       itemTemplate;
61   ITEM01Save1:       itemTemplate;
62   ITEM02Save1:       itemTemplate;
63   item00pointerSave1: packed array [0..30] of char;
64
65
66   { Our data structures, global to the Cmdsl unit. }
67
68   printHandle: prHandle;          {print record handle          }
69   prRect:      Rect;              {TEPaintText rect to draw into      }
70   prStatus:    prStatusRec;       {reports info to user during spooled printing}
71
```

Design Master

```
72 procedure DoSave (index: integer; currWindow: grafPortPtr); extern;
73 { Save a document to disk, where document is associated with a disk file. }
74
75
76 (*****
77 *
78 * DoAbout - Handle About command.
79 *
80 *****)
81
82 procedure DoAbout;
83
84 var
85   theDialog: grafPortPtr;           {pointer to About dialog's grafPort}
86   junk:      integer;               {item hit returned by ModalDialog }
87
88 begin
89   theDialog := GetNewModalDialog (aboutDlg);           {create modal dialog}
90   errNum    := ToolError;
91   if errNum <> 0 then
92     HandleError (errNum, memoryErr, cautionAlertTyp)
93   else begin
94     junk := ModalDialog (nil);           {call Dialog Mgr to detect}
95     { user clicking OK button}
96     { use default filter proc}
97     errNum := ToolError;
98     if errNum <> 0 then           {only error is front window}
99       HandleError (errNum, fatalErr, stopAlertTyp) { not modal dialog! }
100   else
101     CloseDialog (theDialog);
102   end;
103
104 end; {DoAbout}
105
106
107 procedure WantToSave (index: integer; currWindow: grafPortPtr); forward;
108
109 (*****
110 *
111 * DoClose - Handle Close command.
112 *
113 *****)
114
115 procedure DoClose (* index: integer; currWindow: grafPortPtr;
116                   var numWindows: integer *);
117
118 var
119   tmp:   ctlRecHndl;
120   tePtr: teRecPtr;           {pointer to textEdit control's}
121                               { record for front window }
122   flag: integer;             {flag containing dirty bit in }
123                               { teRecord for front window }
124
125 begin
126   EnableMItem (newID);           {can now create new window}
127   EnableMItem (openID);          {can now open a file }
```

Appendix A: Complete MiniWord Program

```

127
128 { Check if they want to save data before closing window. Dereference      }
129 { textEdit control's handle in order to check if the dirty bit has been set }
130 { by Text Edit. If it has, give user chance to save window before closing }
131 { it.                                                                    }
132
133 tmp      := textEdHandle [index];
134 tePtr    := teRecPtr (tmp^);
135
136 if (tePtr^.ctrlFlag & isDirty) <> 0 then
137   WantToSave (index, currWindow);
138 CloseWindow (currWindow);
139
140 numWindows := numWindows - 1;           {one less window open on desktop}
141
142 { If window allocated by Open command, free memory used by it. }
143
144 if windowOpen [index] = fromFile then
145   DisposeHandle (pathHandle [index]);
146
147 windowOpen [index] := noWindow;         {free up slot in window tracking array}
148
149 end; {DoClose}
150
151 (*****
152 *
153 * DoFind - Handle Find command.
154 *
155 *****)
156
157 procedure DoFind;
158
159 begin
160 end;
161
162 (*****
163 *
164 * DoPrint - Handle Print command.
165 *
166 *****)
167
168 procedure DoPrint (* index: integer; currWindow: grafPortPtr *);
169
170 label 99;                               {error label}
171
172 const
173   thruPrinting = $2209;                  {err code returned by TEPaintText when}
174   { starting line # exceeds last line #}
175

```

Design Master

```

176 var
177   prPort:      grafPortPtr;           {Print Manager's grafPort      }
178   currLine:    longint;               {current line # to print      }
179   lastLine:    longint;               {last line # to print         }
180   firstPage:   longint;               {first page to begin printing }
181   finalPage:   longint;               {final page to print          }
182   copies:      integer;               {# copies of document to print}
183   spool:       boolean;               {false = draft mode;         }
184                                   {true = spooled printing      }
185   anError:     boolean;               {true if error detected       }
186   printRecPtr: prRecPtr;               {pointer to print record      }
187   tmp:         longint;
188   answer:      integer;
189
190 begin
191   answer := PrJobDialog (printHandle); {bring up Print Job dialog     }
192   errNum := ToolError;                 {error returned by PrJobDialog?}
193   if errNum <> 0 then begin
194     HandleError (errNum, memoryErr, cautionAlertTyp);
195     goto 99;
196   end;
197   if answer = 0 then                    {want to print document?}
198     goto 99;
199
200   printRecPtr := printHandle^;          {dereference print record handle}
201
202   { Set up page rectangle based on printed page size calculated by Print Manager }
203   { as derived from Job and Page setup dialogs.                                }
204
205   with printRecPtr^ do begin
206     with prInfo.rPage do begin
207       prRect.v1 := v1;
208       prRect.h1 := h1;
209       prRect.v2 := v2;
210       prRect.h2 := h2;
211     end;
212
213     firstPage := prJob.iFstPage;         {get first page to print      }
214     currLine  := (firstPage - 1) * 60;   {calculate 1st line to print, }
215                                   { counting lines from 0, and }
216                                   { 60 lines per page           }
217     prJob.iFstPage := 1;                  {set page # to 1 for Print Manager, since it }
218                                   { counts ea. page it prints, starting at 1 }
219
220     finalPage := prJob.iLstPage;          {get last page to print       }
221     lastLine  := finalPage * 60;          {calculate last line to print }
222
223     { Ensure that starting page number not greater than ending page}
224
225     tmp := finalPage - firstPage;
226     if tmp < 0 then
227       goto 99;
228
229     prJob.iLstPage := convert (tmp) .lsw + 1; {reset last page to print for }
230                                   { Print Mgr, relative to 1      }

```

Appendix A: Complete MiniWord Program

```
231
232 copies      := prJob.iCopies;      {get # copies to print      }
233 firstPage    := currLine;          {remember starting line # in case multiple}
234                                     { copies wanted and printing in draft mode}
235
236 { Determine whether printing in draft or spooled mode.}
237
238 if prJob.bJDocLoop = 0 then
239     spool := false
240 else begin
241     spool := true;
242     copies := 1;                      {PrPicFile handles mult. copies}
243 end;
244
245 { Ensure starting line # is in document by calling Text Edit's }
246 { TEGetTextInfo to get # lines in document.                    }
247
248 TEGetTextInfo (textInfo, 2, textEdHandle [index]);
249 errNum := ToolError;
250 if errNum <> 0 then begin
251     HandleError (errNum, getTextInfoErr, cautionAlertTyp);
252     goto 99;
253 end;
254
255 if currLine > textInfo.lineCount then
256     goto 99;
257
258 end; {with printRecPtr}
259
260
261 { Call Print Manager to open the document for printing; get Print Manager's }
262 { printing grafPort.                                                         }
263
264 anError := false;
265
266 { Outer print loop, to print multiple copies in draft mode. }
267
268 repeat
269
270     prPort := PrOpenDoc (printHandle, nil);
271     errNum := ToolError;
272     if errNum <> 0 then begin
273         HandleError (errNum, printErr, cautionAlertTyp);
274         anError := true;
275     end
276
277 else begin
278
279     { Inner print loop, to print each page in the document. }
280
281     repeat
282
283         PrOpenPage (prPort, nil);          {init. grafPort, no scaling rect. passed}
284         errNum := ToolError;
285         if errNum <> 0 then begin
```

Design Master

```

286         HandleError (errNum, printErr, cautionAlertTyp);
287         anError := true;
288         end
289
290     else begin
291         PenNormal;           {set pen to standard state }
292         MoveTo (0, 0);       {move to top left corner of drawing rectangle}
293
294         { Call TEPaintText to draw text into Print Manager's grafPort. }
295
296         currLine := TEPaintText (prPort, currLine, prRect, 0,
297                                 textEdHandle [index]);
298         errNum := ToolError;
299         if (errNum <> 0) and (errNum <> thruPrinting) then begin
300             HandleError (errNum, printErr, cautionAlertTyp);
301             anError := true;
302             end;
303
304         end; {no error from PrOpenPage}
305
306         PrClosePage (prPort);           {close this printed page}
307         errNum := ToolError;
308         if (errNum <> 0) and (not anError) then begin
309             HandleError (errNum, printErr, cautionAlertTyp);
310             anError := true;
311             end;
312
313                                     {end page-printing loop}
314     until (currLine = -1) or (currLine > lastLine) or (anError);
315
316     end; {no error from PrOpenDoc}
317
318     PrCloseDoc (prPort);           {close document for printing}
319     errNum := ToolError;
320     if (errNum <> 0) and (not anError) then begin
321         HandleError (errNum, printErr, cautionAlertTyp);
322         anError := true;
323         end;
324
325     copies := copies - 1;           {one less copy to print}
326     currLine := firstPage;         {reset for next copy }
327
328 until (copies = 0) or (anError);   {end print copies loop }
329
330
331 { Handle spooled printing. }
332
333 if (spool) and (not anError) then begin
334     PrPicFile (printHandle, nil, @prStatus);           {let Print Mgr allocate new}
335                                                         { grafPort for printing }
336     errNum := ToolError;
337     if errNum <> 0 then
338         HandleError (errNum, printErr, cautionAlertTyp);
339     end;
340

```

Appendix A: Complete MiniWord Program

```

341 SetPort (currWindow);                                {restore window's grafPort}
342 99:
343 end;
344
345 (*****
346 *
347 * DoPSetUp - Handle Page setUp command.
348 *
349 *****)
350
351 procedure DoPSetUp;
352
353 var
354     junk: boolean;
355
356 begin
357 { Bring up Page Setup dialog; throw away result since Print Manager handles }
358 { everything for us. }
359
360 junk := PrStlDialog (printHandle);
361 errNum := ToolError;
362 if errNum <> 0 then
363     HandleError (errNum, memoryErr, cautionAlertTyp);
364
365 end; {DoPSetUp}
366
367 (*****
368 *
369 * InitCmdsl - Initialize the Cmdsl unit's data
370 *             structures.
371 *
372 *****)
373
374 function InitCmdsl (* : boolean *);
375
376 label 99;
377
378 begin
379 InitCmdsl := true;
380
381 { Create print record. First allocate memory to obtain handle to record, then }
382 { call the Print Manager to initialize it. Attributes are locked, don't purge, }
383 { don't move. }
384
385 printHandle := prHandle (NewHandle (140, myID, $C010, nil));
386 errNum := ToolError;
387 if errNum <> 0 then begin
388     HandleError (errNum, memoryErr, stopAlertTyp);
389     InitCmdsl := false;
390     goto 99;
391 end;
392
393 PrDefault (printHandle);
394 errNum := ToolError;
395 if errNum <> 0 then begin

```

Design Master

```

396  HandleError (errNum, fatalPrintErr, stopAlertTyp);
397  InitCmdsl := false;
398  goto 99;
399  end;
400
401
402 { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
403
404 with aboutDlg do begin
405   with dtBoundsRect do begin                                {Enclosing rectangle}
406     v1 := $002B;
407     h1 := $00C4;
408     v2 := $009C;
409     h2 := $01B9;
410   end;
411   dtVisible      := true;                                {Visiblilty flag      }
412   dtRefCon       := 0;                                    {RefCon, for application use}
413   dtItemList [1] := @item00Abt1;                          {item pointer: message  }
414   dtItemList [2] := @item01Abt1;                          {item pointer: OK button }
415   dtItemList [3] := nil;                                   {null terminator        }
416 end;
417
418 with ITEM00Abt1 do begin                                    {About dialog's static text item}
419   itemID := $0064;                                         {Item ID number        }
420   with itemRect do begin                                   {bounding rectangle     }
421     v1 := 0004;
422     h1 := 0008;
423     v2 := 0072;
424     h2 := 0241;
425   end;
426   itemType := $800F;                                       {static text + disable }
427   itemDescr := @item00pointerAbt1;                         {pointer to static text }
428   itemValue := 0096;                                       {length of text        }
429   itemFlag := 0;                                           {default flag          }
430   itemColor := nil;                                       {pointer to color table }
431 end;
432
433 item00pointerAbt1 := 'MiniWord';
434 item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), chr ($0D));
435 item00pointerAbt1 := concat (item00pointerAbt1, 'A simple word processor written');
436 item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), 'by Barbara Allred and ');
437 item00pointerAbt1 := concat (item00pointerAbt1, chr ($0D), 'Design Master', chr
($0D));
438
439 with ITEM01Abt1 do begin                                    {About dialog's OK button}
440   itemID := $0001;
441   with itemRect do begin                                   {bounding rectangle     }
442     v1 := 0089;
443     h1 := 0094;
444     v2 := 0102;
445     h2 := 0149;
446   end;
447   itemType := $000A;                                       {simple button          }
448   itemDescr := @okTitle;                                   {pointer to button's title}
449   itemValue := 0;

```


Appendix A: Complete MiniWord Program

```

450 itemFlag := $0001;                                {bold, round-cornered }
451 itemColor := @item01colorsAbt1;                    {ptr to button's color tbl}
452 end;
453
454 item01colorsAbt1 [0] := $0010;                      {button outline color }
455 item01colorsAbt1 [1] := $00D0;                      {interior color when not highlighted}
456 item01colorsAbt1 [2] := $0070;                      {interior color when highlighted }
457 item01colorsAbt1 [3] := $00E8;                      {text color when not highlighted }
458 item01colorsAbt1 [4] := $00B9;                      {text color when highlighted }
459
460 with saveAlert do begin                             {WantToSave alert template}
461   with atBoundsRect do begin                         {bounding rectangle }
462     v1 := $0028;
463     h1 := $009C;
464     v2 := $007A;
465     h2 := $017B;
466   end;
467   atAlertID := 2;                                     {Alert ID number }
468   atStage1 := $81;                                    {stage 1: draw alert, 1 beep }
469   atStage2 := $81;                                    {stage 2: draw alert, 1 beep }
470   atStage3 := $81;                                    {stage 3: draw alert, 1 beep }
471   atStage4 := $81;                                    {stage 4: draw alert, 1 beep }
472   atItemList [1] := @item00Savel;                    {item pointer: Save message }
473   atItemList [2] := @item01Savel;                    {item pointer: OK button }
474   atItemList [3] := @item02Savel;                    {item pointer: Cancel button}
475   atItemList [4] := nil;                             {null terminator }
476 end;
477
478 with ITEM00Savel do begin                             {save alert's message template}
479   itemID := $0064;
480   with itemRect do begin                             {bounding rectangle }
481     v1 := 37;
482     h1 := 8;
483     v2 := 47;
484     h2 := 217;
485   end;
486   itemType := $800F;                                  {static text + item disable}
487   itemDescr := @item00pointerSavel;                  {pointer to text }
488   itemValue := 28;                                    {length of text }
489   itemFlag := 0;                                       {default flag }
490   itemColor := nil;                                   {no color table }
491 end;
492 item00pointerSavel := 'Save changes before closing?';
493
494 with ITEM01Savel do begin                             {save alert's OK button template}
495   itemID := 1;
496   with itemRect do begin                             {bounding rectangle }
497     v1 := 62;
498     h1 := 12;
499     v2 := 75;
500     h2 := 67;
501   end;
502   itemType := $000A;                                  {simple button }
503   itemDescr := @okTitle;                              {pointer to button's title }
504   itemValue := 0;

```

Design Master

```
505 itemFlag := $0003;           {bold, square-cornered button}
506 itemColor := nil;           {no color table      }
507 end;
508
509 with ITEM02Save1 do begin      {save alert's Cancel button template}
510   itemID := 2;
511   with itemRect do begin      {bounding rectangle      }
512     v1 := 62;
513     h1 := 115;
514     v2 := 75;
515     h2 := 201;
516   end;
517   itemType := $000A;          {simple button      }
518   itemDescr := @cancelTitle;  {pointer to button's title }
519   itemValue := 0;
520   itemFlag := $0002;          {plain, square-cornered button}
521   itemColor := nil;           {no color table      }
522 end;
523
524 99:
525 end; {InitCmds1}
526
527 (*****
528 *
529 * WantToSave - Ask user if they'd like to save
530 *           a file before closing its window.
531 *
532 *****)
533
534 procedure WantToSave (* index: integer, currWindow: grafPortPtr *);
535
536 var
537   result: integer;
538
539 begin
540 { Bring up want-to-save alert.  If user selects OK button, call DoSave to }
541 { save the window to disk.      }
542
543 if (NoteAlert (saveAlert, nil)) = 1 then
544   DoSave (index, currWindow);
545
546 end; {WantToSave}
547
548
549 end. {Cmds1 unit}
```

CMDS2.PAS

Appendix A: Complete MiniWord Program

```
1 {$keep 'Cmds2'}
2 unit Cmds2;
3 {-----}
4 {
5 { Cmds2 - Handles the menus commands New, Open,
6 {      Save, and Save as.
7 {
8 { Written by Barbara Allred and Design Master
9 {
10 { Copyright 1990
11 { Byte Works, Inc.
12 {
13 {-----}
14 interface
15
16 uses
17   Common, MemoryMgr, WindowMgr, ControlMgr, MenuMgr, DialogMgr, SFToolset,
18   IntegerMath, TextEdit, GSOS;
19
20 {$LibPrefix '0/'}
21 uses
22   Globals, Error;
23
24
25 { Subroutines that can be called from outside of the Cmds2 unit. }
26
27 procedure DoNew (var index: integer; var numWindows: integer);
28 { Create an untitled window on the desktop. }
29
30 procedure DoOpen (var index: integer; var numWindows: integer);
31 { Open a disk file, displaying its contents in a new window. }
32
33 procedure DoSave (index: integer; currWindow: grafPortPtr);
34 { Save a document to disk, where document is associated with a disk file. }
35
36 procedure DoSaveAs (index: integer; currWindow: grafPortPtr);
37 { Save a document to disk; document may/may not be associated with a disk file }
38
39 function InitCmds2: boolean;
40 { Initialize the Cmds2 unit. }
41
42
43 { The rest of the file is private to the Cmds2 unit. }
44 implementation
45
46 type
47   gsosMinOutputBuffer = record           {minimally sized GS/OS output buffer}
48     totalSize: integer;
49     currSize: integer;
50     theText:   packed array [1..2] of char;
51   end;
52
53   smGSOSinString = record                {small GS/OS input string}
54     currSize: integer;
55     theText:   packed array [1..10] of char;
```

Design Master

```

56     end;
57
58 var
59 { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
60
61     windColorTable: wColorTbl;                {document window definitions}
62     window:        paramList;
63     control001111: editTextControl;
64
65
66 { Our data structures, global to the Cmdsl unit. }
67
68 { Window tracking information -- We're allowing only 4 windows to be opened }
69 { on the desktop. }
70
71 untitledNum: integer;                {# to assign to next untitled window}
72 wName:      array [0..3] of pString17;    {array of window names }
73 thePath:    array [0..3] of gsosPathNamePtr; {array of pathname pointers}
74                                     { for opening files }
75
76                                     {Standard File Operations data structures}
77 theReply:   replyRecord5_0;
78 openMsg:    packed array [0..20] of char;  {msg to display in open file dlg}
79 openTypes:  typeList5_0;
80 saveMsg:    packed array [0..27] of char;  {msg to display in save file dlg}
81 saveName:   smGSOSinString;                {default filename to appear in }
82                                     { save file dialog }
83
84                                     {Data structures for GS/OS file handling calls}
85
86 openRec:    openOSDCB;                    {open file parameter block }
87 readRec:    readWriteOSDCB;                {read file parameter block }
88 writeRec:   readWriteOSDCB;                {write file parameter block }
89 closeRec:   closeOSDCB;                    {close file parameter block }
90 destroyRec: destroyOSDCB;                  {delete file parameter block}
91 getFileInfoRec: getFileInfoOSDCB;          {getFileInfo parameter block}
92 createRec:  createOSDCB;                    {create file parameter block}
93 options:    gsosMinOutputBuffer; {buffer for FST info, returned by GS/OS}
94
95
96 { These are some of the private procedures and functions, forward declared }
97 { so that we can alphabetize the subroutines in this unit. }
98
99 procedure DoClose (index: integer; currWindow: grafPortPtr;
100                  var numWindows: integer); extern;
101 { Close the front window, checked if it's ours before passing currWindow to us }
102
103 procedure FiniWindow (var numWindows: integer); forward;
104 { Wraps up window creation. }
105
106 procedure GetOpenName (var theReply: replyRecord5_0; window: paramList); forward;
107 { Gives new window a title based on file just opened. }
108
109 function GetText (index: integer; textInfo: teInfoRec;
110                  var size: longint): boolean; forward;

```

Appendix A: Complete MiniWord Program

```
111 { Retrieves text from Text Edit for the currently active window. }
112
113 procedure GetUntitledName (index: integer; var untitledNum: integer); forward;
114 { Gives new window an "Untitled X" title. }
115
116 procedure InitWindow (var window: paramList; var index: integer); forward;
117 { Initializes window creation. }
118
119
120 (*****
121 *
122 * CreateWindow - Create new window on the desktop.
123 *
124 * Output:
125 *     true if window was created; false otherwise
126 *
127 *****)
128
129 function CreateWindow (var index: integer; var numWindows: integer): boolean;
130
131 var
132     tmp: longint;
133
134 begin
135     CreateWindow := true;                {assume we'll be successful}
136
137 { Call Window Manager's NewWindow function to create new window on desktop. }
138
139 windowPtr [index] := NewWindow (window);
140 errNum           := ToolError;
141 if (errNum <> 0) then begin
142     HandleError (errNum, memoryErr, cautionAlertTyp);
143     CreateWindow := false;
144 end
145
146
147 { Add text edit control to window. }
148
149 else begin
150     tmp := ord4 (@control001111);
151     textEdHandle [index] := NewControl2 (windowPtr [index], pointerVerb, tmp);
152     errNum := ToolError;
153     if (errNum <> 0) then begin
154         HandleError (errNum, memoryErr, cautionAlertTyp);
155         DoClose (index, windowPtr [index], numWindows);
156         CreateWindow := false;
157     end;
158 end;
159
160 end; {CreateWindow}
161
162 (*****
163 *
164 * DoNew - Handle New command.
165 *
```

Design Master

```

166 *****)
167
168 procedure DoNew (* var index: integer; var numWindows: integer *);
169
170 begin
171   InitWindow (window, index);           {init. window data structures }
172   GetUntitledName (index, untitledNum); {create untitled window's name}
173   if CreateWindow (index, numWindows) then begin {create new window on desktop }
174     windowOpen [index] := fromNew;      {set flag that window from the New cmd}
175     FiniWindow (numWindows);            {disable Open, New cmds, if necessary }
176   end;
177 end; {DoNew}
178
179 (*****
180 *
181 * DoOpen - Handle Open command.
182 *
183 *****)
184
185 procedure DoOpen (* var index: integer; var numWindows: integer *);
186
187 label 99;
188
189 var
190   aHandle: handle;
191   tmp:     handle;
192
193 begin
194   InitWindow (window, index);           {initialize window data structures}
195
196   { Make SFGGetFile2 call to bring up SFO Open dialog and get filename and }
197   { pathname of file to open. }
198
199   SFGGetFile2 (20,                {upper left corner X-coord of SFGGetFile2's dialog}
200               20,                {upper left corner Y-coord of SFGGetFile2's dialog}
201               pointerVerb,        {prompt is pointer to P-string }
202               @openMsg,           {pointer to prompt }
203               nil,                {no filter procedure }
204               openTypes,          {fileTypes, auxTypes of files to open }
205               theReply);          {GS/OS 5.0 reply record }
206
207   errNum := ToolError;
208   if errNum <> 0 then
209     HandleError (errNum, getFileErr, cautionAlertTyp)
210   else if theReply.good <> 0 then begin {Does user want to open a file?}
211     tmp := handle (theReply.pathRef); {Yes - get pointer to pathname }
212     openRec.pathname := pointer (ord4 (tmp^) + 2);
213     GSOSOpen (openRec);               {open the file}
214     errNum := ToolError;
215     if errNum <> 0 then begin
216       HandleError (errNum, openErr, cautionAlertTyp);
217       goto 99;
218     end;
219   end;
220

```

Appendix A: Complete MiniWord Program

```

221
222 { Record file's pathname - we'll need it when saving file to disk. }
223 { Pass file's reference # to GS/OS read and close parameter blocks. }
224
225 thePath [index] := gosPathNamePtr (openRec.pathname);
226 readRec.refNum := openRec.refNum;
227 closeRec.refNum := openRec.refNum;
228
229 GetOpenName (theReply, window); {get filename to display in window's title}
230
231 { Create new window - if unable to create, close the file & exit. }
232
233 if not (CreateWindow (index, numWindows)) then begin
234     GSOSClose (closeRec);
235     goto 99;
236 end;
237
238 { Read the file into memory, pass its text on to TextEdit to paint into }
239 { window. }
240 { Allocate memory block to read file into. Attributes flag is locked, }
241 { can't move, don't purge, don't cross bank boundary. }
242
243 aHandle := NewHandle (openRec.dataEOF, myID, $C010, nil);
244 errNum := ToolError;
245 if errNum <> 0 then begin
246     HandleError (errNum, memoryErr, cautionAlertTyp);
247     GSOSClose (closeRec);
248     DoClose (index, windowPtr [index], numWindows);
249     goto 99;
250 end;
251
252 readRec.dataBuffer := aHandle^; {dereference memory handle}
253 readRec.requestCount := openRec.dataEOF; {get # bytes to read }
254 GSOSRead (readRec);
255 errNum := ToolError;
256
257 { If unable to read the file, report the error, close the window, close the }
258 { file. }
259
260 if errNum <> 0 then begin
261     HandleError (errNum, readErr, cautionAlertTyp);
262     DoClose (index, windowPtr [index], numWindows);
263     GSOSClose (closeRec);
264     goto 99;
265 end;
266
267 { Call Text Edit to transfer the text from the read buffer to the control. }
268
269 TESSetText ($0005, {text is raw data }
270     ord4 (readRec.dataBuffer), {pointer to data }
271     readRec.transferCount, {size of data }
272     pointerVerb, {style ref is pointer }
273     0, {not passing style info}
274     nil); {use active TE control }
275 errNum := ToolError;

```

Design Master

```
276   if errNum <> 0 then begin
277       { Only possible errs are messing up parms or propagation of memory errs }
278
279       HandleError (errNum, fatalErr, stopAlertTyp);
280       GSOSClose (closeRec);
281       DoClose (index, windowPtr [index], numWindows);
282       goto 99;
283   end;
284
285   { Perform clean-up: record that window is from a disk file, check if need }
286   { to disable Open & New commands, close the file, release read buffer's }
287   { memory. }
288
289   windowOpen [index] := fromFile;
290   FiniWindow (numWindows);
291   GSOSClose (closeRec);
292   DisposeHandle (aHandle);
293
294   end; {if user wants to open file}
295
296 99:
297 end; {DoOpen}
298
299 (*****
300 *
301 * DoSave - Handle Save command.
302 *
303 *****)
304
305 procedure DoSave (* index: integer, currWindow: grafPortPtr *);
306
307 label 99;
308
309 var
310   tePtr: teRecPtr;           {ptr to active window's textEdit control record}
311
312 begin
313   { Check if window from the New command; if so, execute DoSaveAs. }
314
315   if windowOpen [index] = fromNew then begin
316       DoSaveAs (index, currWindow);
317       goto 99;
318   end;
319
320   { Check if file has changed since it was last saved to disk. Check if the }
321   { dirty bit in the textEdit record has been set by Text Edit. }
322
323   tePtr := teRecPtr (textEdHandle [index]^);           {dereference teHandle}
324   if (tePtr^.ctrlFlag & isDirty) = 0 then
325       goto 99;
326
327   { Get the window's text to write to disk. }
328
329   if not (GetText (index, textInfo, writeRec.requestCount)) then
330       goto 99;
```


Appendix A: Complete MiniWord Program

```
331
332 { Get file's pathname for delete, create, and open operations. }
333
334 destroyRec.pathname := thePath [index];
335 createRec.pathname  := thePath [index];
336 openRec.pathname    := thePath [index];
337
338 GSOSDestroy (destroyRec);                {delete the old disk file}
339 errNum := ToolError;
340 if errNum <> 0 then begin
341   HandleError (errNum, deleteErr, cautionAlertTyp);
342   goto 99;
343 end;
344
345 GSOSCreate (createRec);                  {create new file}
346 errNum := ToolError;
347 if errNum <> 0 then begin
348   HandleError (errNum, createErr, cautionAlertTyp);
349   goto 99;
350 end;
351
352 GSOSOpen (openRec);                      {open the new file}
353 errNum := ToolError;
354 if errNum <> 0 then begin
355   HandleError (errNum, openErr, cautionAlertTyp);
356   goto 99;
357 end;
358
359 { Dereference buffer's handle to get pointer to data to write. }
360
361 writeRec.dataBuffer := buffer^;
362
363 { Get file's reference # for write and close operations. }
364
365 writeRec.refNum := openRec.refNum;
366 closeRec.refNum := openRec.refNum;
367
368 GSOSWrite (writeRec);                    {write window to disk}
369 errNum := ToolError;
370 if errNum <> 0 then
371   HandleError (errNum, writeErr, cautionAlertTyp)
372
373 { Clear the dirty bit after saving file. }
374 else
375   tePtr^.ctrlFlag := tePtr^.ctrlFlag & notDirty;
376
377 GSOSClose (closeRec);                    {close the disk file}
378
379 99:
380 end; {DoSave}
381
382 (*****
383 *
384 * DoSaveAs - Handle Save as command.
385 *
```

Design Master

```
386 *****)
387
388 procedure DoSaveAs (* index: integer; currWindow: grafPortPtr *);
389
390 label 99;
391
392 const
393     fileNotFound = $0046;                {File not found error, reported by GS/OS}
394
395 var
396     tePtr: teRecPtr;                    {pointer to textEdit control's record}
397     tmp: handle;
398
399 begin
400 { Make SFPutFile2 call to bring up SFO Save dialog and get filename and }
401 { pathname of file to create, open, and then write. }
402
403 SFPutFile2 (20,                        {upper left corner X-coord of SFPutFile2's dlg}
404             20,                        {upper left corner Y-coord of SFPutFile2's dlg}
405             pointerVerb,                {prompt is pointer to P-string }
406             ord4 (@saveMsg),            {pointer to prompt }
407             pointerVerb,                {default name is pointer }
408             ord4 (@saveName),           {pointer to GS/OS input string }
409             theReply);                  {pointer to GS/OS 5.0 reply record}
410 errNum := ToolError;
411 if errNum <> 0 then begin
412     HandleError (errNum, fileErr, cautionAlertTyp);
413     goto 99;
414 end;
415
416 if theReply.good = 0 then                {Does user want to open a file?}
417     goto 99;
418
419 { Fill in GS/OS write parameter block fields. Call TEGetText to get }
420 { the text to write to disk. }
421
422 if not (GetText (index, textInfo, writeRec.requestCount)) then
423     goto 99;
424
425
426 { Get pathname selected by user with SFPutFile2 call, in reply record. }
427 { Then make GS/OS GetFileInfo call to see if file already exists. }
428
429 tmp := handle (theReply.pathRef);
430 getFileInfoRec.pathname := pointer (ord4 (tmp^) + 2);
431 GSOSGetFileInfo (getFileInfoRec);
432 errNum := ToolError;
433 if (errNum <> 0) and (errNum <> fileNotFound) then begin
434     HandleError (errNum, fileErr, cautionAlertTyp);
435     goto 99;
436 end;
437
438 { If no error reported by GetFileInfo call, then need to delete old }
439 { file before creating new file with the same pathname. }
440
```

Appendix A: Complete MiniWord Program

```
441 if errNum = 0 then begin
442   destroyRec.pathname := getFileInfoRec.pathname;
443   GSOSDestroy (destroyRec);
444   errNum := ToolError;
445   if errNum <> 0 then begin
446     HandleError (errNum, deleteErr, cautionAlertTyp);
447     goto 99;
448   end;
449 end;
450
451 { Now create new file with pathname selected by user. }
452
453 createRec.pathname := getFileInfoRec.pathname;
454 GSOSCreate (createRec);
455 errNum := ToolError;
456 if errNum <> 0 then begin
457   HandleError (errNum, createErr, cautionAlertTyp);
458   goto 99;
459 end;
460
461 { Open the file to prepare for writing window's contents to disk. }
462
463 openRec.pathname := getFileInfoRec.pathname;
464 GSOSOpen (openRec);
465 errNum := ToolError;
466 if errNum <> 0 then begin
467   HandleError (errNum, openErr, cautionAlertTyp);
468   goto 99;
469 end;
470
471 { Record file's reference # returned by GS/OS in write and close }
472 { parameter blocks. }
473
474 writeRec.refNum := openRec.refNum;
475 closeRec.refNum := openRec.refNum;
476
477 { Check if window previously belonged to another file, and if so, }
478 { dispose of its old pathname handle. Record its new path handle. }
479
480 if windowOpen [index] = fromFile then
481   DisposeHandle (pathHandle [index]);
482 pathHandle [index] := tmp;
483
484 { Dereference buffer's handle to get pointer to data to write to disk. }
485 { Write the window's contents to disk. }
486
487 writeRec.dataBuffer := buffer^;
488 GSOSWrite (writeRec);
489 errNum := ToolError;
490 if errNum <> 0 then begin
491   HandleError (errNum, writeErr, cautionAlertTyp);
492   GSOSClose (closeRec);
493   goto 99;
494 end;
495
```

Design Master

```
496 { Clear the dirty bit after saving the file. }
497
498 tePtr          := teRecPtr (textEdHandle [index]^);
499 tePtr^.ctrlFlag := tePtr^.ctrlFlag & notDirty;
500
501
502 { Get opened file's name to put in window' title. }
503
504 GetOpenName (theReply, window) ; {get filename to display in window's title}
505 SetWTitle (window.wTitle, currWindow);
506
507
508 { Record that window comes from a file, close the disk file, then record the }
509 { new pathname. }
510
511 windowOpen [index] := fromFile;
512 thePath [index]    := getFileInfoRec.pathname;
513 GSOSClose (closeRec);
514
515 99:
516 end; {DoSaveAs}
517
518 (*****
519 *
520 * FiniWindow - Disable New and Open commands if
521 *           this is the 4th window opened.
522 *
523 *****)
524
525 procedure FiniWindow (* var numWindows: integer *);
526
527 begin
528   numWindows := numWindows + 1;           {one more open window }
529   if numWindows = maxWindows then begin   {max # open windows reached?}
530     DisableMItem (openID);                {yes - don't let 'em open or}
531     DisableMItem (newID);                 { create another window }
532   end;
533
534 end; {FiniWindow}
535
536 (*****
537 *
538 * GetOpenName - Build filename when open a file.
539 *
540 *****)
541
542 procedure GetOpenName (* var theReply: replyRecord5_0; window: paramList *);
543 { Will dispose of filename's handle; will change window's title. }
544
545 var
546   namePtr: ptr;
547   size:   byte;
548   tmp:    handle;
549   tmpPtr: pStringPtr;
550
```

Appendix A: Complete MiniWord Program

```

551 begin
552 { Use filename returned by SFO to place into window's title string. }
553 { First need to dereference filename handle, then move beyond GS/OS }
554 { total buffer length. }
555
556 tmp      := handle (theReply.nameRef);
557 namePtr   := pointer (ord4 (tmp^) + 2);
558
559
560 { Get length of filename; AND with $000F to restrict length to 15 characters. }
561 { Set filename pointer to point to 2nd byte of length word, to prepare for }
562 { converting a GS/OS input string into a Pascal-style string.  Stuff the }
563 { length byte into the byte pointed to by namePtr to complete the conversion. }
564
565 size := namePtr^ & $0F;
566 namePtr := pointer (ord4 (namePtr) + 1);
567 namePtr^ := size;
568
569
570 { Move filename into area pointed to by wTitle, surrounding it with blanks. }
571
572 tmpPtr      := pStringPtr (namePtr);
573 window.wTitle^ := concat (' ', tmpPtr^, ' ');
574
575
576 { Dispose of filename handle since we won't need it after this. }
577
578 DisposeHandle (tmp);
579
580 end; {GetOpenName}
581
582 (*****
583 *
584 * GetText - Call Text Edit's TEGetText routine
585 *           to get text to write to disk.
586 *
587 * Output:
588 *         true if text can be returned, false otherwise
589 *
590 *****)
591
592 function GetText (* index: integer; textInfo: teInfoRec;
593                  var size: longint): boolean *);
594
595 label 99;
596
597 const
598     NotEnoughMemory = $0201; {not enough memory error}
599
600 begin
601 { First call Text Edit's TEGetTextInfo to determine if the current TEGetText }
602 { buffer is large enough to hold the window's text. }
603
604 TEGetTextInfo (textInfo, 2, textEdHandle [index]);
605 errNum := ToolError;

```

Design Master

```
606 if errNum <> 0 then begin
607   HandleError (errNum, getTextInfoErr, cautionAlertTyp);
608   GetText := false;
609   goto 99;
610 end;
611
612
613 { Check if current buffer is too small; if so, attempt to allocate new buffer. }
614
615 if textInfo.charCount > bufferSize then
616   if MaxBlock >= textInfo.charCount then begin
617     DisposeHandle (buffer);
618     buffer := NewHandle (textInfo.charCount, myID, $0000, nil);
619     errNum := ToolError;
620     if errNum <> 0 then begin
621       HandleError (errNum, memoryErr, cautionAlertTyp);
622       GetText := false;
623       goto 99;
624     end
625   else
626     bufferSize := textInfo.charCount;
627   end
628 else begin
629   {can't allocate new buffer}
630   HandleError (NotEnoughMemory, memoryErr, cautionAlertTyp);
631   GetText := false;
632   goto 99;
633 end;
634
635
636 { Call TEGetText to get window's text. We'll pass a handle to the buffer in }
637 { which to store the text, and not ask for any style information.           }
638
639 size := TEGetText ($000D, buffer, bufferSize, pointerVerb,
640   nil, textEdHandle [index]);
641 errNum := ToolError;
642 if errNum <> 0 then begin
643   HandleError (errNum, getTextErr, cautionAlertTyp);
644   GetText := false;
645   goto 99;
646 end;
647
648 GetText := true;
649
650 99:
651 end; {GetText}
652
```

Appendix A: Complete MiniWord Program

```

653 (*****
654 *
655 * GetUntitledName - Create next untitled window's
656 *           name.
657 *
658 *****)
659
660 procedure GetUntitledName (* index: integer; var untitledNum: integer *);
661
662 var
663   num : packed array [0..5] of char;
664
665 begin
666   { Convert untitled window number to a Pascal-style string, then increment }
667   { untitled number. }
668
669   num      := cnvis (untitledNum);
670   untitledNum := untitledNum + 1;
671
672   { Move character representation of untitled number into the window's title. }
673
674   window.wTitle^ := concat (' Untitled ', num, ' ');
675
676 end; {GetUntitledName}
677
678 (*****
679 *
680 * InitCmds2 - Initialize the Cmds2 unit's data
681 *           structures.
682 *
683 *****)
684
685 function InitCmds2 (* : boolean *);
686
687 label 99;
688
689 begin
690   InitCmds2 := true;
691   untitledNum := 1; {# to assign to next untitled window }
692
693   { Allocate memory block for TEGetText call. }
694
695   buffer := NewHandle (1024, myID, $0000, nil); {don't purge, can move}
696   errNum := ToolError;
697   if errNum <> 0 then begin
698     HandleError (errNum, memoryErr, stopAlertTyp);
699     InitCmds2 := false;
700     goto 99;
701   end;
702   bufferSize := 1024;
703
704   { Initialize SFO variables. }
705
706   with theReply do begin
707     nameVerb := newHandleVerb; {reference is undefined: SFO will}

```

Design Master

```

708 pathVerb := newHandleVerb;           { allocate and return a handle }
709 end;
710
711 openMsg := 'Select file to open: ';
712 with openTypes do begin
713   numEntries := 2;                    {number of entries in filetype list}
714   fileAndAuxTypes [1].flags := $8000; {match filetype, any auxtype, allow}
715                                     { user to select these files }
716   fileAndAuxTypes [1].fileType := $04; {ASCII text file }
717   fileAndAuxTypes [1].auxType := 0;    {don't care about auxtype }
718   fileAndAuxTypes [2].flags := $8000; {match filetype, any auxtype, allow}
719                                     { user to select these files }
720   fileAndAuxTypes [2].fileType := $B0; {APW source file }
721   fileAndAuxTypes [2].auxType := 0;    {don't care about auxtype }
722 end;
723
724 saveMsg := 'Enter name of file to save: ';
725 with saveName do begin
726   currSize := 7;
727   theText := 'newFile';              {default name to appear SFO save dlg}
728 end;
729
730 { Initialize GS/OS variables. }
731
732 with options do begin
733   totalSize := 6;
734   currSize := 2;
735 end;
736
737 with openRec do begin
738   pCount := 15;                      {parameter count }
739   access := $0003;                   {request read/write access }
740   resourceNumber := 0;               {resource number: open data fork}
741   optionList := @options;            {pointer to GS/OS result buffer }
742 end;
743
744 with readRec do begin
745   pCount := 5;                       {parameter count }
746   cachePriority := 0;                {don't cache file}
747 end;
748
749 closeRec.pCount := 1;                {parameter count}
750 destroyRec.pCount := 1;
751
752 with writeRec do begin
753   pCount := 5;                       {parameter count }
754   cachePriority := 0;                {don't cache file}
755 end;
756
757 with getFileInfoRec do begin
758   pCount := 12;                      {parameter count }
759   optionList := @options;            {pointer to GS/OS output buffer}
760 end;
761
762 with createRec do begin

```


Appendix A: Complete MiniWord Program

```

763 pCount      := 7;           {parameter count          }
764 access      := $00C3;       {destroy, rename, write, read access}
765 fileType    := $0004;       {ASCII file            }
766 auxType     := 0;
767 storageType  := $0001;       {standard file         }
768 dataEOF      := 0;           {initial size of data fork is 0     }
769 resourceEOF  := 0;           {initial size of resource fork is 0  }
770 end;
771
772
773 { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
774 { Initialize window parameter list and editText control record. }
775
776 with window do begin          {Window parameter list          }
777     paramLength := 78;        {parm list length              }
778     wFrameBits  := $C1E7;     {frame bits                    }
779     wTitle      := nil;       {pointer to window's title     }
780     wRefCon     := 0;         {we'll use to store index into }
781
782     with wZoom do begin       { window-tracking arrays       }
783         v1 := $001E;          {zoomed rectangle             }
784         h1 := $0014;
785         v2 := $003C;
786         h2 := $0064;
787     end;
788     wColor      := @windColorTable; {color table pointer          }
789     wYOrigin     := $0000;        {vert offset of content       }
790     wXOrigin     := $0000;        {horiz offset of content      }
791     wDataH       := $0000;        {data area height             }
792     wDataW       := $0000;        {data area width              }
793     wMaxH        := $00C8;        {max grow height              }
794     wMaxW        := $0280;        {max grow width               }
795     wScrollVer   := $0000;        {vert. arrow scroll amount     }
796     wScrollHor   := $0000;        {horiz arrow scroll amount     }
797     wPageVer     := $0000;        {vert. page amount            }
798     wPageHor     := $0000;        {horiz page amount            }
799     wInfoRefCon  := 0;            {info bar ref con             }
800     wInfoHeight  := 0;            {info bar height              }
801     wFrameDefProc := nil;         {window definition procedure   }
802     wInfoDefProc := nil;         {info bar draw routine        }
803     wContDefProc := nil;         {window content draw routine   }
804     with wPosition do begin      {window position rectangle     }
805         v1 := $001E;
806         h1 := $0014;
807         v2 := $00C2;
808         h2 := $025F;
809     end;
810     wPlane      := grafPortPtr (-1); {window plane, -1 for front    }
811     wStorage     := nil;            {address of memory for window record}
812 end;
813
814 with windColorTable do begin
815     frameColor := $0010;          {window frame color}
816     titleColor := $0D81;          {title color       }
817     tBarColor  := $021D;          {title bar color   }

```

Design Master

```

818   growColor   := $0074;           {grow box color   }
819   infoColor   := $0000;           {info bar color  }
820 end;
821
822 with control001111 do begin        {editText control template }
823   pCount       := 23;              {parameter count          }
824   controlID    := $000087DA;       {ID number TaskMaster will use }
825   with boundsRect do begin        {encloses entire control, including}
826     v1 := $0002;                  { scroll bars and grow box }
827     h1 := $0002;
828     v2 := $0000;
829     h2 := $0000;
830   end;
831   procRef      := $85000000;       {Code for editText control  }
832   flags        := $0000;           {control is visible, not dirty yet }
833   moreFlags    := $7C00;           {add a grow box to the control }
834   refCon       := 0;
835   textFlags    := $62200000;       {smart cut & paste; allow editing }
836   with indentRect do begin        {use standard indentation   }
837     v1 := $FFFF;
838     h1 := $FFFF;
839     v2 := $FFFF;
840     h2 := $FFFF;
841   end;
842   vertBar      := $FFFFFFFF;        {create vert. scroll bar just inside}
843                                     { right edge of bounds rect }
844   vertAmount   := 0;               {scroll 9 pixels, up- & down-arrows }
845   horzBar      := 0;               {MUST BE NULL version 1.0 }
846   horzAmount   := 0;               {MUST BE NULL version 1.0 }
847   styleRef     := 0;               {use default style and ruler }
848   textDescriptor := 0;             {no initial text for control }
849   textRef      := 0;
850   textLength   := 0;
851   maxChars     := 0;
852   maxLines     := 0;
853   maxCharsPerLine := 0;            {MUST BE NULL version 1.0 }
854   maxHeight    := 0;              {MUST BE NULL version 1.0 }
855   colorRef     := 0;              {no color table }
856   drawMode     := 0;
857   filterProcPtr := nil;            {use built-in filter procedures}
858 end;
859
860 99:
861 end; {InitCmds2}
862
863 (*****
864 *
865 * InitWindow - Perform window initialization,
866 *               common to both DoOpen and DoNew.
867 *
868 *****)
869
870 procedure InitWindow (* var window: paramList; var index: integer *);
871
872 begin

```

Appendix A: Complete MiniWord Program

```
873 { Find empty slot in windowOpen array for this window. }
874
875 index := 0;
876 while windowOpen [index] <> noWindow do
877   index := index + 1;
878
879 { Use window's RefCon to track position in window arrays. }
880
881 window.wRefCon := index;
882
883 { Set window's title pointer. }
884
885 window.wTitle := @wName [index];
886 end;
887
888
889 end. {Cmds2 unit}
```

ERROR.PAS

```
1 {$keep 'Error'}
2 unit Error;
3 {-----}
4 { }
5 { Error - Text editor's error handling routines. }
6 { }
7 { Written by Barbara Allred and Design Master }
8 { }
9 { Copyright 1990 }
10 { Byte Works, Inc. }
11 { }
12 {-----}
13 interface
14
15 uses
16   Common, DialogMgr, IntegerMath, ControlMgr, TextEdit;
17
18 {$LibPrefix '0/'}
19 uses
20   Globals;
21
22 type
23                                     {Error numbers}
24
25   errType = (OOM, fatalPrintErr, memoryErr, openErr, readErr, printErr,
26             writeErr, fileErr, createErr, deleteErr, getTextErr, getFileErr,
27             getTextInfoErr, fatalErr);
28
29 var
30   { Variables accessible from outside the Error unit. }
31
32   errNum: integer;                  {error # returned by tools}
33   errMsg: array [errType] of pString50; {error messages}
34
```

Design Master

```
35
36 { Subroutines that can be called from outside of the Error unit. }
37
38 procedure InitError;
39 { Initializes the Error unit. }
40
41 procedure HandleError (error: integer; whichErr: errType; whichAlert: alertType);
42 { Reports error detected by the MiniWord program. }
43
44
45 implementation
46
47 var
48   { *** GENERATED BY DESIGN MASTER *** }
49
50   errAlert:      alertTemplate;
51   ITEM00Err1:    itemTemplate;
52   ITEM01Err1:    itemTemplate;
53   ITEM02Err1:    itemTemplate;
54   item00pointerErr1: packed array [0..60] of char;
55   item01pointerErr1: packed array [0..5]  of char;
56
57 (*****
58 *
59 * HandleError - Report errors detected in
60 *               MiniWord.
61 *
62 * Input:
63 *       error      - error number returned by tool
64 *       whichErr   - error message #
65 *       whichAlert - alert type
66 *
67 *****)
68
69 procedure HandleError (* error: integer; whichErr: errType;
70                       whichAlert: alertType *) ;
71
72 var
73   tmp: longint;
74   junk: integer;
75
76 begin
77 { Get error message to display. }
78
79 item00err1.itemDescr := @errMsg [whichErr];
80
81
82 { Convert integer error number to hex string in order to display error number }
83 { in the same format as used by GS/OS and the tools. }
84
85 tmp := ord4 (item01err1.itemDescr) + 2;      {adjust address obtained from }
86                                           { alert item template to point }
87                                           { beyond length byte and '$' }
88 Int2Hex (error, tmp, 4);      {call Integer Math toolset to perform conversion}
89
```

Appendix A: Complete MiniWord Program

```

90 { Bring up alert. }
91
92 case whichAlert of
93   stdAlertTyp:      junk := Alert (errAlert, nil);
94
95   noteAlertTyp:     junk := NoteAlert (errAlert, nil);
96
97   cautionAlertTyp:  junk := CautionAlert (errAlert, nil);
98
99   stopAlertTyp:     begin
100     junk := StopAlert (errAlert, nil);
101     done := true;
102   end;
103 end;
104 end;
105
106 (*****
107 *
108 * InitError - Initializes the Error unit.
109 *
110 *****)
111
112 procedure InitError;
113
114 begin
115   errMsg [OOM] := 'Out of memory. Aborting MiniWord.';
116   errMsg [fatalPrintErr] := 'Fatal error reported by Print Manager.';
117   errMsg [memoryErr] := 'Memory error: Unable to perform operation.';
118   errMsg [openErr] := 'Error returned by GS/OS when opening file.';
119   errMsg [readErr] := 'Error returned by GS/OS when reading file.';
120   errMsg [printErr] := 'Error: Aborting printing.';
121   errMsg [writeErr] := 'Error returned by GS/OS when writing file.';
122   errMsg [fileErr] := 'Error when accessing file.';
123   errMsg [createErr] := 'Error returned by GS/OS when creating file.';
124   errMsg [deleteErr] := 'Error returned by GS/OS when deleting file.';
125   errMsg [getTextErr] := 'Error returned by TextEdit when reading file.';
126   errMsg [getFileErr] := 'Error returned by SFO when accessing file.';
127   errMsg [getTextInfoErr] := 'Error returned by Text Edit when getting info.';
128   errMsg [fatalErr] := 'Fatal error: Cannot recover.';
129
130 { *** GENERATED BY DESIGN MASTER, with comments by B.A. *** }
131 { Initialize the error alert. }
132
133 with errAlert do begin
134   with atBoundsRect do begin
135     v1 := $002A;
136     h1 := $004C;
137     v2 := $0082;
138     h2 := $01F9;
139   end;
140   atAlertID := 1;
141   atStage1 := $81;
142   atStage2 := $81;
143   atStage3 := $81;
144   atStage4 := $81;

```

Design Master

```

145  atItemList [1] := @item00Err1;      {item pointer: Error message  }
146  atItemList [2] := @item01Err1;      {item pointer: Error number  }
147  atItemList [3] := @item02Err1;      {item pointer: OK button    }
148  atItemList [4] := nil;              {null terminator            }
149 end;
150
151 with ITEM00Err1 do begin              {Error message item template }
152   itemID := $0064;                   {item ID number             }
153   with itemRect do begin              {bounding rectangle         }
154     v1 := 30;
155     h1 := 10;
156     v2 := 45;
157     h2 := 409;
158   end;
159   itemType := $800F;                  {static text + item disable }
160   itemDescr := @item00pointerErr1;    {pointer to error message   }
161   itemValue := 50;                    {length of static text to display }
162   itemFlag := 0;                      {default flag               }
163   itemColor := nil;                   {no color table             }
164 end;
165 item00pointerErr1 := 'Here is a message that is fifty characters long!!!';
166
167 with ITEM01Err1 do begin              {Error alert's error number item}
168   itemID := $0065;                   {Item ID number             }
169   with itemRect do begin              {bounding rectangle         }
170     v1 := 50;
171     h1 := 150;
172     v2 := 65;
173     h2 := 200;
174   end;
175   itemType := $800F;                  {static text + item disable }
176   itemDescr := @item01pointerErr1;    {pointer to Pascal-style string}
177   itemValue := 5;                     {length of the text         }
178   itemFlag := 0;                      {default flag               }
179   itemColor := nil;                   {no color table             }
180 end;
181 item01pointerErr1 := '$0000';
182
183 with ITEM02Err1 do begin              {Error alert's OK button template}
184   itemID := 1;
185   with itemRect do begin              {bounding rectangle         }
186     v1 := 70;
187     h1 := 200;
188     v2 := 85;
189     h2 := 230;
190   end;
191   itemType := $000A;                  {simple button               }
192   itemDescr := @okTitle;              {pointer to button's title  }
193   itemValue := 0;
194   itemFlag := $0001;                  {bold, round-cornered button}
195   itemColor := nil;                   {no color table             }
196 end;
197
198 end; {InitError}
199

```

```
200 end. {Error unit}
```

GLOBALS.PAS

```
1 {$keep 'globals'}
2 unit Globals;
3 {-----}
4 {                                     }
5 {   Global data for MiniWord       }
6 {                                     }
7 {-----}
8 interface
9 uses
10  Common, ControlMgr, TextEdit;
11
12 const
13  isDirty    = $0040;           {mask to check dirty bit      }
14  notDirty   = $FFBF;           {mask to clear dirty bit after saving }
15                                     { file to disk                }
16
17  maxWindows = 4;               {maximum # windows allowed on desktop }
18
19                                     {Menu item IDs}
20  cutID      = 251;
21  copyID     = 252;
22  pasteID    = 253;
23  closeID    = 255;
24  aboutID    = 256;
25  newID      = 257;
26  openID     = 258;
27  saveID     = 259;
28  saveAsID   = 260;
29  pSetUpID   = 261;
30  printID    = 262;
31  quitID     = 263;
32  findID     = 264;
33
34
35 type
36  windowType = (noWindow, fromFile, fromNew);           {Window types}
37
38                                     {Alert types}
39  alertType = (stdAlertTyp, stopAlertTyp, noteAlertTyp, cautionAlertTyp);
40
41  pString50 = packed array [0..50] of char;
42  pString17 = packed array [0..17] of char;
43
44  convert = record                                     {for choosing whether to handle 4 bytes}
45    case boolean of                                   { as 1 longint or 2 integers          }
46      true: (long:      longint);
47      false: (lsw, msw: integer);
48    end;
49
50 var
```

Design Master

```
51  done:          boolean;          {true if user is ready to exit MiniWord  }
52  myID:          integer;          {MiniWord's user ID                }
53  buffer:        handle;           {buffer to hold text returned by TEGetText}
54  bufferSize:    longint;          {size of this buffer                }
55  textInfo:      teInfoRec;        {gives # lines, # chars in a document  }
56
57  okTitle:        packed array [0..2] of char;          {common button titles}
58  cancelTitle:    packed array [0..6] of char;
59
60
61  { Window tracking information -- We're allowing only 4 windows to be opened }
62  { on the desktop.                                                         }
63
64  windowOpen:     array [0..3] of windowType;          {array of open window flags    }
65  windowPtr:      array [0..3] of grafPortPtr;         {array of pointers to grafPorts}
66  textEdHandle:   array [0..3] of ctlRecHndl;          {array of editText ctl handles }
67  pathHandle:     array [0..3] of handle;              {array of pathname handles,    }
68                                                         { returned by SFO              }
69
70 implementation
71 end. {Globals unit}
```


Appendix B

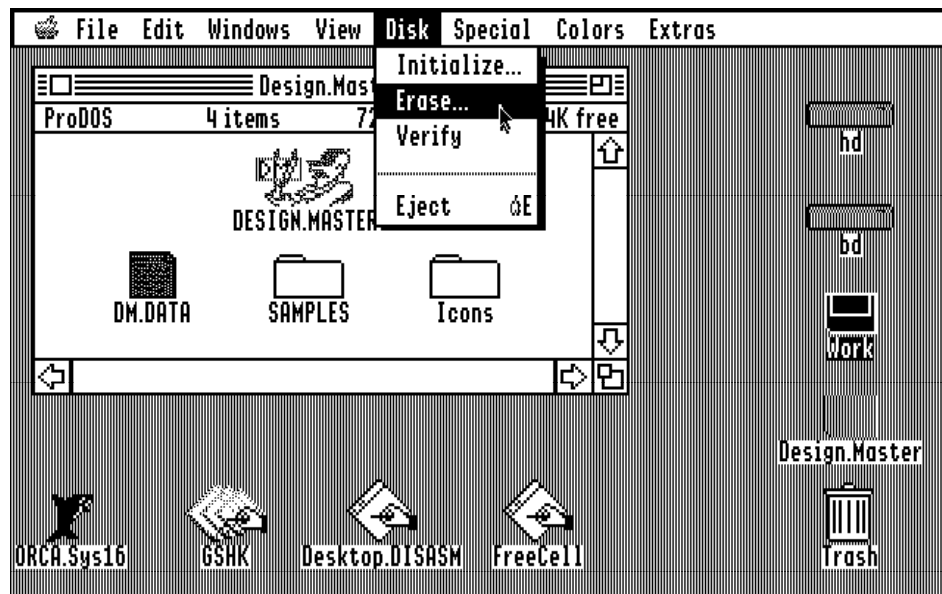
The Boot Disk and How To Use It

The Boot Disk that came in your Design Master package can be used to boot your Apple IIGS computer. The Boot Disk contains version 5.0.2 (or later) of the GS/OS operating system. Note that Design Master requires version 5.0 or later of GS/OS.

When you boot your computer with the Boot Disk, it boots into the Finder, Apple's program launcher. This appendix provides a brief overview of the Finder, showing you how to copy disks and files, initialize disks, and rename disks and files. The Finder can do much more than what is covered here. If you'd like to learn more about the Finder, you should contact A.P.D.A. for complete documentation.

Introduction To The Finder

To begin our tutorial of the Finder, insert the Boot Disk into your boot drive, then boot your machine. After the machine has booted, the screen will look like a desktop, and all of the disks that you have on-line will be depicted:



Design Master

Apple's Finder is an icon based file manager and program launcher. Most of the small pictures on the right-hand side of the screen (these pictures are called icons) represent disk drives. Your desktop probably does not have this many disk icons. Each kind of disk drive has a different icon. Two of the disk drives, named HD and BD, are hard disks. Two more, ICON.EDITOR and Design.Master, are 3.5" floppy disks. ARCHIVES and LIBRARY are disks on an AppleTalk network; these disks are available to any computer on the network. The last disk icon is called RAM5. RAM5 is treated as a disk, but is physically made up of some of the memory in the computer.

The Trash icon at the lower right-hand side of the screen is used to delete files. We'll look at how that is done later in this appendix.

The window you see at the lower-right of the desktop represents the contents of the Design.Master disk. Each of the files or folders on the disk is again represented by an icon.

Across the top of the screen is the menu bar. The menu bar is used to execute commands in the Finder.

Opening Disks, Folders, And Files

The Finder can launch stand-alone (S16) applications. To launch a program, you simply use the mouse to double-click on the icon of the program. For example, to start Design Master, you would double-click on the picture of the man cutting out a window.

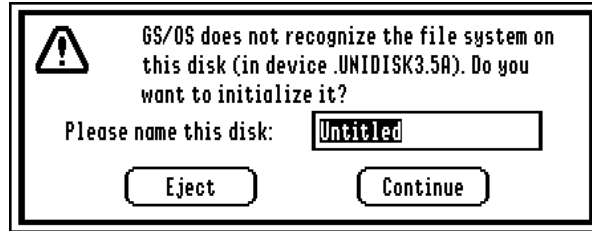
You can use the Finder to open disks and folders. To open an object, use the mouse to double-click on the object's icon. When you open a disk or folder, it will appear in a new window on the desktop. Within the window will be icons of all of the files and folders within the disk or folder that you opened.

To close the window, click in the small box at the left side of the title bar.

Initializing Disks

The Finder can be used to initialize disks. We will walk through the instructions for making a back-up copy of your Boot Disk to show you how to format and copy disks.

Place a 3.5-inch disk on-line that is to contain your back-up copy of the Boot Disk. (If you have only one 3.5-inch drive, simply eject the Boot Disk and replace it with your back-up disk.) If the disk is not formatted for the Apple IIGS, you will be presented with this alert:



If you want to format the disk, type **My.Boot.Disk** in the box labeled *Please name this disk*, and then click the *Continue* button. Click the *Eject* button if you don't want to initialize the disk.

Copying Disks, Folders, and Files

Once you've created a clean, formatted disk, you can make a copy of the disk by using the mouse to drag the image of Design Master's Boot Disk onto *My.Boot.Disk*. "Dragging" is accomplished by clicking the mouse on an object, and, keeping the mouse button depressed, moving the mouse. The destination is reached when its icon changes to inverse as the object being dragged touches it.

Your decision to replace the contents of *My.Boot.Disk* with *Boot.Disk* will be confirmed with this alert:



Design Master

Creating Folders

You can create folders with the Finder by first opening the disk or folder that is to contain the new folder, and then selecting the `New Folder` command from the `File` menu. The new folder will appear in inverse in the window of the disk or folder that you opened, and will be named `Untitled`.

Renaming Disks, Folders, and Files

To rename an object on the desktop, use the mouse to move the arrow cursor onto the name of the object. When the arrow cursor turns into a bar with inverted arrows on either end, you can edit the name. Use the mouse to drag with the bar over the name, selecting it. Once you've selected it, you can type in the name you prefer. Be sure to press the `RETURN` key after entering the new name! If you do not hit `RETURN`, the name will not be changed.

Deleting Folders and Files

You can delete folders and files by dragging their icons onto the trash can icon. You will see the trash can in inverse, as well as see it "bulge" with the objects you've thrown away, when you've successfully moved them into it. The objects you've thrown away are not actually deleted until you select the `Empty Trash` item from the `Special` menu. This feature gives you the chance to "remove" items from the trash, by simply not selecting `Empty Trash`. If you drag a disk into the trash can, the disk will be ejected from its drive. The files on the disk are not erased.

Index

Numbers

320 mode display 149, 150
640 mode display 149

A

A.P.D.A. 3
About Design Master command 131
About item 131
Add menu dialog 10
Add menu item dialog 12
alert boxes, creating 22, 24
alert windows, creating 135
Append command 17, 22, 30
Apple Numerics Manual, Second Edititon manual 4
Apple Computer, Inc..i.Apple Developers and Programmers Association 3
Apple IIGS computer 4
Apple IIGS Firmware Reference manual 4
Apple IIGS Hardware Reference manual 4
Apple IIGS Owners Guide manual; 1
Apple IIGS Technical Manual suite 3
Apple IIGS Toolbox Reference manuals10, 26, 63
Apple IIGS Toolbox Reference Volume I manual 4
Apple IIGS Toolbox Reference Volume II manual 4
Apple IIGS Toolbox Reference Volume III 4
Apple menu 8
Apple menu, creating 10, 139
Apple menu, MiniWord program 46
APW assembly language 1
auxID field of userID 36

B

backing up Design Master 7
BeginUpdate call, in the MiniWord program 55
binary format file, Design Master 1
binary image file 142
Boot Disk 7
Boot Disk, Design Master 2
buttons, simple, creating 21

C

C language source code 142
C programming language 1
caution alert 23
Check box command 159
Clear command 20, 22, 147
clipboard 147, 151
Clipboard file 19, 22
close box 8
Close command 8, 9, 18
CloseDialog call, in the MiniWord program 67
CloseWindow call, in the MiniWord program 70
Cmds1 unit, in the MiniWord program 63
Cmds2 unit, in the MiniWord program 119
Cmds2 unit, MiniWord program 83
cnvis function call, in the MiniWord program 90
Colors command 152
colors, check boxes 161
colors, for windows 27
colors, general 129
colors, icon controls 171
colors, pop-up menus 165
colors, push buttons 22, 158
colors, radio buttons 155

Design Master

colors, scroll controls	174
colors, title bar patterns	134
colors, windows	135, 152
concat function call, in the MiniWord program	90
Control Manager	34, 55, 91
Controls menu	9, 153
controls, removing	147
Copy command	19, 147
Create call, GS/OS, in the MiniWord program	106
CreateWindow function, in the MiniWord program	123
CreateWindow function, MiniWord program	91
custom window frames	134
customizing Design Master	149
Cut command	20, 22, 147

D

data structures, MiniWord program	34
default button, dialogs	21
delete key	20
Design Master disk	7
designing a program	31
desk accessories	8
Desk Manager	46
Destroy call, GS/OS, in the MiniWord program	106
dialog boxes, creating	135
dialog items	9
dialog items, editing	22
Dialog Manager	22, 24, 67
dialog templates, in the MiniWord program	67
dialogs, creating	18
dirty bit, textEdit controls, in the MiniWord program;	99
DisableMenuItem call, in the MiniWord program	92
disk drives, required	2
disks, copying	7

disks, initializing	7
DisposeAll call	36
DisposeAll call, MiniWord program	54
dividing lines, in menus	14
DoAbout procedure, in the MiniWord program	67
DoClose procedure, in the MiniWord program	68
DoNew procedure, in the MiniWord program	122
DoNew procedure, MiniWord program	89
DoOpen procedure, in the MiniWord program	121
DoOpen procedure, MiniWord program	93
DoPrint procedure, in the MiniWord program	74
DoPSetUp procedure, in the MiniWord program	71
DoQuit procedure, in the MiniWord program	58
DoSave procedure, in the MiniWord program	99
DoSaveAs procedure, in the MiniWord program	104, 121
DrawControls call, in the MiniWord program	55
DrawMenuBar call, MiniWord program	46

E

Edit line command	161
Edit menu	9, 19, 146
edit text controls, creating	171
edit-line items	10
End test run command	22, 30
EndUpdate call, in the MiniWord program	55
error detection, in the MiniWord program	118
error handling, in the MiniWord program	32
error handling, MiniWord program	62
error reporting	150
Error unit, in the MiniWord program	58, 118

		Index	
event mask, in the MiniWord program	49	GetWRefCon call, in the MiniWord program	53
event record, extended, in the MiniWord program	48	Globals unit, in the MiniWord program	114
EventLoop procedure, in the MiniWord program	47	grafPort pointer, MiniWord program	91
expertise, expected of user of Design Master	1	grid	148
		Grow box command	175
		GS/OS	2, 4, 7, 34
		GS/OS operating system	4
		GS/OS Reference manuals	95
		GS/OS Reference, Volume 1, Beta Draft manual	4
		GS/OS Reference, Volume 2, Beta Draft manual	4
		GS/OS strings	84
		GS/OS version 5.0	37
F		H	
family numbers, radio buttons	154, 156	HandleError Procedure, in the MiniWord program	61
File menu	9, 125	HandleMenu procedure, in the MiniWord program	56
Finder	7	HandleSpecial procedure, in the MiniWord program	55
FiniWindow procedure, MiniWord program	92	help command	131
FiniWindow routine, in the MiniWord program	92	Help! command	8
FixAppleMenu call, MiniWord program	46	Hide Clipboard command	151
FixMenuBar call, MiniWord program	46	HiliteMenu call, in the MiniWord program	55
folders, creating	7	hot keys	8, 13
formatting disks	7	Human Interface Guidelines	156
Frame command	152	The Apple Desktop Interface	4
FrontWindow call, in the MiniWord program	53		
G		I	
GetCtlHandleFromID call, in the MiniWord program	124	Icon command	169
GetFileInfo call, GS/OS, in the MiniWord program	105	icon controls, creating	169
GetNewModalDialog call, in the MiniWord program	67	icon editor, in Design Master	169
GetOpenName procedure, in the MiniWord program	98, 120	icons, in alert boxes	23
GetSysWFlag call, in the MiniWord program	53	inactive controls	156, 159, 160, 162
GetText function, in the MiniWord program	102	Init function, MiniWord program	35
GetUntitledName procedure, in the MiniWord program	123	InitCmds1 function, in the MiniWord program	40
GetUntitledName procedure, MiniWord program	90		

Design Master

InitCmds2 function, in the MiniWord program	40, 83, 119
InitCursor call, in the MiniWord program	40
InitError procedure, in the MiniWord program	40, 58, 118
initializing disks	7
InitMenus procedure, in the MiniWord program	118
InitWindow procedure, in the MiniWord program	121
InitWindow procedure, MiniWord program	89
insertion point	10
InsertMenu call, MiniWord program	46
Integer Math tool set	62
items, removing from dialogs	20

K

key equivalents, check boxes	160
key equivalents, pop-up menus	166
key equivalents, push buttons	157
key equivalents, radio buttons	154
keyboard equivalents	8, 13

L

List command	166
list controls	12
list controls, creating	166
long static text item	163
Long text command	163

M

main program, MiniWord program	35
Main programming module, in the MiniWord program	115
manual, type face conventions	3
manuals, recommended for programming	3
Memory Manager	36, 63
memory, required to run Design Master	2
menu bars, creating	9, 137

menu item templates, MiniWord program	40
menu items	139
menu items, creating	12
menu items, deleting	14
menu items, removing	140
Menu Manager	40, 55, 92
menu resources, in the MiniWord program	115
menu templates, MiniWord program	40
menus, creating	10
menus, removing	140
Merlin 816 source code	142
MiniWord disk	8
MiniWord folder	31
MMStartUp call	36
ModalDialog call, in the MiniWord program	67
MoveTo call, in the MiniWord program	80
MyWindow function, in the MiniWord program	52

N

NDA's	8
New command, MiniWord program	89
new desk accessories	8, 46
New Dialog command	18, 22, 24, 135
New MenuBar command	9, 137
New Window command	25, 132
NewControl2 call, in the MiniWord program	91
NewHandle call, in the MiniWord program	63
NewMenu2 call, in the MiniWord program	118
NewMenu2 call, MiniWord program	45
NewWindow call	26, 133
NewWindow routine, in the MiniWord program	91
NewWindow2 call, in the MiniWord program	123
note alert	23

		Index
O		
open call, GS/OS, in the MiniWord program	95, 106	
Open DM file command	125	
ORCA assembly language	1	
ORCA/APW source code	142	
ORCA/Pascal	31	
ORCA/Pascal compiler	36	
P		
Pascal language source code	142	
Pascal programming language	1, 31	
Paste command	19, 22, 147	
PenNormal call, in the MiniWord program	80	
Picture command	167	
picture controls, creating	167	
PopUp command	164	
pop-up controls, creating	164	
pop-up menus	12	
PrClosePage call, in the MiniWord program	81	
PrDefault call, in the MiniWord program	63	
Preferences command	149	
Print Manager	34, 63, 71, 74	
print record, in the MiniWord program	71	
printing documents, in the MiniWord program	74	
PrJobDialog call, in the MiniWord program	74	
ProDOS 16	4	
ProDOS 8	4	
Programmers Introduction to the Apple IIGS manual;	4	
PrOpenDoc call, in the MiniWord program	79	
PrOpenPage call, in the MiniWord program	80	
PrPicFile call, in the MiniWord program	82	
PrStdDialog call, in the MiniWord program	71	
Push button command	21, 156	
push buttons, removing	22	
Q		
Quick Draw II	80	
Quick Draw II tool set	40	
Quit command	9	
R		
Radio button command	153	
RAM	2	
read call, GS/OS, in the MiniWord program	96	
reading list	3	
reference list	3	
registration card	2	
Release.Notes file	2	
Remove menu dialog	11	
Remove menu item dialog	14	
removing menus	11	
requirements, disk drives		
see disk drives, required	2	
requirements, memory		
see memory, required	2	
requirements, software		
see software, required	2	
resource editor	113	
resource fork, creating	142	
resource forks	113	
resource IDs, in the MiniWord program	114	
Resource Manager	22, 37, 113, 114	
Rez compiler	1	
Rez source code	142	
S		
samples, for Design Master	7	
SANE toolset	4	
Save as command	30	
Save command	16, 125	
Screen format command	149	

Design Master

scroll bar controls, creating	173
Scroll command	173
selecting items, in dialog boxes	19
SetPort call, in the MiniWord program	82
SetWTitle call, in the MiniWord program	108
SFGetFile2 call, in the MiniWord program	93
SFPutFile2 call, in the MiniWord program	104
Show Clipboard command	19, 151
ShutDown procedure, MiniWord program	54
ShutDownTools call, MiniWord program	54
simple buttons, creating	156
simple buttons, removing	22
size box controls, creating	175
software, required to run Design Master	2
source code files, creating	16
source code ID	133
source code, generated by Design Master	31
Source ID chars command	149
standard alert	23
Standard File open dialog	130
Standard File Operations tool set	34, 93, 104
StartUpTools call, in the MiniWord program	114
StartUpTools call, MiniWord program	37
Static text command	162
StatText command	163
stop alert	23
Structures menu	9, 151
system requirements	2

T

task record, extended, in the MiniWord	
program	48
TaskMaster	32
TaskMaster call, in the MiniWord program	48
Technical Introduction to the Apple IIGS	
manual	4

TEGetText call, in the MiniWord program	102
TEPaintText call, in the MiniWord program	81
TESetText call, in the MiniWord program	97
Test run command	22, 30, 147
Text edit command	171
text edit control, in the MiniWord program	91
Text Edit tool set	34, 81, 83, 102
TextEdit command	28
title bar patterns, defining	27
Title command	152
Toolbox Update Notes	37
Toolbox Update Notes manual	99
Toolbox, Apple IIGS	4
tools, starting	37
top down design	31
Turn grid on/off command	148

U

user ID, MiniWord program	36
---------------------------	----

V

verbs	37
version number	131
version numbers	2
View file command	18, 22, 24
visual cues	3

W

WantToSave procedure, in the MiniWord	
program	70
warranty registration card	2
window controls	9
Window Manager	26, 32, 48, 53, 55, 70, 91, 108, 133
window parameter list	133
window titles	134, 152
windows, changing	152

windows, creating	25, 133	Write call, GS/OS, in the MiniWord program	Index 107
-------------------	---------	--	--------------