

Clayton Walnum's

# C-manship



**COMPLETE**

**Learn to program your ST in C!**

```
#include <stdio.h>
#include <osbind.h>
#include <gendefs.h>
#include <obdefs.h>

main()
{
    appl_init();
    open_vwork();
    if (Getrez() != LOW)
        v_clsvwk();
    else
        main_loop();
    appl_exit();
}
```

**A Taylor Ridge Book**

# **C-MANSHIP COMPLETE**

**CLAYTON WALNUM**

**TAYLOR RIDGE BOOKS  
MANCHESTER, CONNECTICUT**

Copyright © 1990 by Clayton Walnum. All rights reserved.

Any reproduction of this work, in part or whole, mechanical or electronic, is strictly forbidden without the written consent of the author, with the exception of brief passages to be used in a review.

While every effort has been made to ensure the accuracy of the contents of this book, the author and the publisher accept no liability for losses that may arise from the use of the information contained herein. This book is sold without warranty, either express or implied. Published by Taylor Ridge Books, P.O. Box 78, Manchester, CT 06045.

This book was produced on an Atari Mega 4 workstation, using Calamus desktop publishing software.

Designed by  
Bryan Schappel

Cover designed by  
Maurice Molyneaux

This material was originally published in both ANALOG Computing and ST-Log.

Printed in the United States of America.

***To Lee Pappas and Michael DesChenes, for letting me through the door.***

## ACKNOWLEDGMENTS

Many people over the years have contributed, either knowingly or unknowingly, to this book. At the top of the list are Lee Pappas and Michael DesChenes, who, even though I wore a suit to my interview, hired me as a full-time staff member of A.N.A.L.O.G. Computing. I'd also like to acknowledge the Massachusetts staff of A.N.A.L.O.G. Computing and ST-Log, particularly the following: Diane Gaw, who told me my writing was coherent even when I swore it wasn't; Douglas Weir, who bailed me out whenever the source code stopped making sense; Patrick Kelly, who supplied the dirty jokes; and Charles Bachand, who was, and still is, a buddy. In addition, gratitude is due to LFP, Inc., for rescuing a sinking ship, and to Bryan Schappel, for motivating me to publish this book, and for doing so much of the work. Scott Whittlesey gets a thank you for blowing my mind with a Timex/Sinclair, all those many years ago. And, as always, I'm indebted to my wife, Lynn, who played the computer widow so well. Finally, to all the Atari computer owners who read my articles, used my programs, and still said such nice things...hey, you're the best.

## DOCUMENTATION PORTING CREDIT

The documentation has been ported to HYPertext by [Lonny Pursell](#).

This HYPertext was created from the original Calamus \*.CDK files. Unfortunately the fonts were missing. The images are not exact since other fonts were used. He has tried to match them as close as possible. The image of Clayton Walnum on the inside of the back cover is missing. It was not contained in the CDK file. He suspects it was put in at the time of printing since it's a rather high resolution black & white photo. Tools used:

- GFABASIC      CDK ascii extraction
- QED            text editor
- STeno          reformat paragraphs
- ST-Guide      testing
- HCP            HYPertext compiler
- Calamus       export ASCII text, print to IMG
- GEM-View     crop IMG files
- Interface      recreated some of the images, touched up others
- Book           the actual book for comparison

Again many thanks to Lonny for porting the original document as this project have taken a lot of his time. He had to write a program to extract the text from something like 30 or so individual files, formatted them all, fixed all the fonts that were characters (wingdats), some of the images were made by hand because he could not get Calamus to export them correctly.

The HYPertext V0.70 (8/6/2008) has been ported to PDF by DrCoolZic (jlg).

## TABLE OF CONTENT

ACKNOWLEDGMENTS .....	3
DOCUMENTATION PORTING CREDIT .....	3
TABLE OF CONTENT .....	4
INTRODUCTION .....	10
<i>Some History</i> .....	10
<i>C-Manship, the Book</i> .....	11
<i>Some Important Details</i> .....	11
<i>What about the Disks?</i> .....	12
<i>Let's Boogie</i> .....	12
CHAPTER 1 - SOME BASICS .....	13
<i>Why C?</i> .....	13
<i>C, Wherefore Art Thou?</i> .....	13
<i>Underway At Last</i> .....	14
<i>A Simple Program</i> .....	14
<i>Where's the Beef?</i> .....	17
CHAPTER 2 - A LOOK AT STRINGS .....	20
<i>A Look at the Program</i> .....	20
<i>Some Fancy Stuff</i> .....	23
<i>Type Conversions</i> .....	24
<i>Odds and Ends</i> .....	25
CHAPTER 3 - LOOPING AND IF STATEMENTS .....	26
<i>Onward</i> .....	26
<i>The Golden Moment</i> .....	27
<i>Back to the Program</i> .....	28
<i>Another Break in the Proceedings</i> .....	31
<i>Back To It</i> .....	31
<i>Take a Breath</i> .....	32
Program Listing #1 .....	32
CHAPTER 4- FLOW OF CONTROL AND FUNCTIONS .....	35
<i>The Game's Afoot (Without Toes)</i> .....	35
<i>Digging Deeper</i> .....	37
<i>Breathing Time</i> .....	40
Program Listing #1 .....	41
CHAPTER 5 - STORAGE CLASSES AND ARRAYS .....	44
<i>Game Time Again</i> .....	44
<i>Some Classy Information</i> .....	45
<i>Hip, Hip Array!</i> .....	46
<i>Another Dimension</i> .....	48
<i>Whambles For Sale</i> .....	49
Program Listing #1 .....	50
Program Listing #2 .....	52
Program Listing #3 .....	53
CHAPTER 6 - FILE HANDLING AND CUSTOM INPUT ROUTINES .....	54

# C-MANSHIP COMPLETE – by CLAYTON WALNUT

<i>The Innards</i> .....	54
<i>Doing it Our Way</i> .....	54
<i>A Bit of Construction</i> .....	56
<i>Disk Files</i> .....	56
<i>Starting Our File</i> .....	57
<i>Writing Our File</i> .....	57
<i>Simple, but Cute</i> .....	57
Program Listing #1.....	58
Program Listing #2.....	60
Program Listing #3.....	60
<b>CHAPTER 7 - POINTERS AND MACROS</b> .....	63
<i>A Point of Declaration</i> .....	63
<i>Putting Them to Work</i> .....	64
<i>Incrementing and Decrementing</i> .....	66
<i>The Proof</i> .....	66
<i>A Glimpse of Macros</i> .....	67
Program Listing #1.....	67
Program Listing #2.....	68
<b>CHAPTER 8 - STRUCTURES AND MORE ON POINTERS</b> .....	69
<i>Filling It In</i> .....	69
<i>Getting It Out</i> .....	70
<i>Layers Upon Layers</i> .....	70
<i>More Layers!</i> .....	71
<i>An Important Point</i> .....	71
<i>Pointing to a Member</i> .....	72
<i>Functions and Structures</i> .....	72
<i>The Listing</i> .....	73
Program Listing #1.....	73
<b>CHAPTER 9 - MORE LOOPING STRUCTURES AND FILE I/O</b> .....	81
<i>Unfinished Business</i> .....	81
<i>A Quick Look at GEM</i> .....	81
<i>And a Peek at VDI</i> .....	82
<i>Moving Along</i> .....	83
<i>The VDI Cursor Stuff</i> .....	84
<i>Printer Output</i> .....	84
<i>Odds and Ends</i> .....	85
<i>A New Loop</i> .....	86
<i>Break, Continue, and Goto</i> .....	86
<b>CHAPTER 10 - THE FIRST LOOK AT GEM AND THE VDI</b> .....	88
<i>A Review of GEM</i> .....	88
<i>Presenting the VDI</i> .....	88
<i>The VDI functions</i> .....	88
<i>The Sample Program</i> .....	89
<i>Let's Get Virtual</i> .....	89
<i>Polylines</i> .....	90

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

<i>Rounded Rectangles</i> .....	91
<i>Filled Rounded Rectangles</i> .....	91
<i>Circles</i> .....	91
<i>Polymarkers</i> .....	92
<i>Filled Rectangles</i> .....	92
<i>Ellipses</i> .....	93
<i>Arcs</i> .....	93
<i>Pie Slices</i> .....	93
<i>Fill Patterns</i> .....	94
<i>Use Those Tools!</i> .....	94
Program Listing #1 .....	95
<b>CHAPTER 11 - VDI TEXT FUNCTIONS</b> .....	99
<i>Who's a Dummy?</i> .....	99
<i>Converting Between Resolutions</i> .....	100
<i>Of Mice and C</i> .....	101
<i>Menus and Varmints with Buttons</i> .....	101
<i>Text Effects</i> .....	103
<i>Text Height</i> .....	103
<i>Text Rotation</i> .....	104
<i>Mouse Prestidigitation</i> .....	104
<i>Break Time</i> .....	104
Program Listing #1 .....	105
<b>CHAPTER 12 - ALERT BOXES AND CUSTOM MOUSE FORMS</b> .....	109
<i>Getting to Work</i> .....	109
<i>A Small Matter of Incompatibility</i> .....	109
<i>Alert Boxes</i> .....	110
<i>They Don't Fit!</i> .....	111
<i>Custom Mice</i> .....	112
<i>Coding It</i> .....	112
<i>Mission Accomplished</i> .....	113
Program Listing #1 .....	114
<b>CHAPTER 13 - THE FILE SELECTOR AND RASTER OPERATIONS</b> .....	116
<i>Picking a File</i> .....	116
<i>Calling Up a File Selector</i> .....	116
<i>File Selector Housekeeping</i> .....	117
<i>Raster Operations</i> .....	118
<i>Filling in the Blanks</i> .....	119
<i>The Next Listing</i> .....	120
<i>The Raster Details</i> .....	121
<i>Off Again</i> .....	122
Program Listing #1 .....	122
Program Listing #2 .....	124
<b>CHAPTER 14 - OBJECT TREES AND DIALOG BOXES</b> .....	127
<i>The Definitions</i> .....	127
<i>RCP: A Mini Tutorial</i> .....	128

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

<i>Crankin' with the RCP .....</i>	<i>129</i>
<i>So How About Some Details?.....</i>	<i>132</i>
<i>Editable Text .....</i>	<i>132</i>
<i>Your First Dialog Box .....</i>	<i>133</i>
<i>Taking It Apart .....</i>	<i>134</i>
<i>The Mysterious TEDINFO .....</i>	<i>136</i>
<i>As the Fear Sets In .....</i>	<i>136</i>
<i>Breathing Time.....</i>	<i>136</i>
Program Listing #1.....	137
<b>CHAPTER 15 - MORE ON DIALOG BOXES.....</b>	<b>142</b>
<i>The Workings .....</i>	<i>142</i>
<i>And Speaking of the Program... ..</i>	<i>142</i>
<i>Finding the Data.....</i>	<i>145</i>
<i>Dealing with TEDINFO .....</i>	<i>146</i>
<i>Releasing Resource Memory.....</i>	<i>147</i>
<i>Knowing Who Your Friends Are .....</i>	<i>148</i>
<i>Closing Up Shop.....</i>	<i>148</i>
Program Listing #1.....	149
Program Listing #2.....	149
<b>CHAPTER 16 - MENU BARS .....</b>	<b>153</b>
<i>Another RCP Tutorial .....</i>	<i>153</i>
<i>Steppin' Through the Menu Bar .....</i>	<i>153</i>
<i>The Program.....</i>	<i>155</i>
<i>Menu Bars in Your Program.....</i>	<i>155</i>
<i>A Nifty Message System .....</i>	<i>156</i>
<i>Enough of this Event Junk .....</i>	<i>157</i>
<i>Another Lesson Learned .....</i>	<i>159</i>
Program Listing #1.....	160
<b>CHAPTER 17 - WINDOWS - PART 1 - DRAWING .....</b>	<b>163</b>
<i>What Are Windows Really? .....</i>	<i>163</i>
<i>The Window Demo .....</i>	<i>163</i>
<i>Drawing a Window.....</i>	<i>164</i>
<i>Handling a Window.....</i>	<i>167</i>
<i>Window Moved .....</i>	<i>167</i>
<i>Full Size or Previous Size? .....</i>	<i>167</i>
<i>Closed For Business.....</i>	<i>168</i>
<i>More to Come.....</i>	<i>168</i>
Program Listing #1.....	169
<b>CHAPTER 18 - WINDOWS - PART 2 - SIZING.....</b>	<b>172</b>
<i>The Demo Program.....</i>	<i>172</i>
<i>Any Size You Like .....</i>	<i>172</i>
<i>Redraw Messages .....</i>	<i>173</i>
<i>Lock the Window .....</i>	<i>174</i>
<i>The Rectangle List.....</i>	<i>174</i>
<i>The Clipping Rectangle .....</i>	<i>175</i>

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

<i>Emptying the Rectangle List</i> .....	175
<i>Something of Interest</i> .....	176
<i>The Agenda</i> .....	176
Program Listing #1.....	177
<b>CHAPTER 19 - WINDOWS - PART 3 - THE RECTANGLE LIST</b> .....	181
<i>Rectangles Revealed</i> .....	181
<i>Out of the Fog</i> .....	186
<i>Sidelines</i> .....	186
<i>Another Day, Another Dollar</i> .....	186
Program Listing #1.....	187
<b>CHAPTER 20 - WINDOWS - PART 4 - SLIDERS AND ARROWS</b> .....	192
<i>Getting a Directory</i> .....	192
<i>Slipping and Sliding</i> .....	193
<i>Me and My Arrow</i> .....	194
<i>Paging All Sliders</i> .....	196
<i>Anywhere You Like</i> .....	196
<i>An Important Note</i> .....	197
Program Listing #1.....	197
<b>CHAPTER 21 - D.E.G.A.S. PICTURE VIEWER</b> .....	204
<i>Hey! That Space is Reserved!</i> .....	204
<i>Putting It Back Where We Found It</i> .....	208
<i>Mission Complete</i> .....	209
Program Listing #1.....	209
<b>CHAPTER 22 - THE INTERNAL CLOCK/CALENDAR</b> .....	214
<i>Computer Dating</i> .....	215
<i>A Bit About Bits</i> .....	216
<i>But What About the Date?</i> .....	217
<i>Some Timely Information</i> .....	219
<i>Setting the Time and Date</i> .....	219
<i>All Ashore Who's Going Ashore</i> .....	220
Program Listing #1.....	221
Program Listing #2.....	227
Program Listing #3.....	228
Program Listing #4.....	228
<b>CHAPTER 23 - Desk Accessories with Built-In Resource Trees</b> .....	229
<i>Our Resource Tree</i> .....	229
<i>Writing a Desk Accessory</i> .....	231
<i>Waiting Forever</i> .....	233
<i>The Desk Accessory Link</i> .....	233
Program Listing #1.....	234
Program Listing #2.....	241
<b>CHAPTER 24 - THE GRAPHICS MANAGER LIBRARY</b> .....	242
<i>The Sample Program</i> .....	242
<i>Déjà Vu</i> .....	242
<i>Our Program</i> .....	243



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

<i>Some Leftovers</i> .....	244
<i>Put on the Coffee</i> .....	245
Program Listing #1 .....	246
<b>CHAPTER 25 - THE MYSTERY OF COMPILE AND LINK</b> .....	<b>252</b>
<i>Stating the Obvious</i> .....	<b>252</b>
<i>Compilation</i> .....	<b>253</b>
<i>Linking</i> .....	<b>254</b>
<i>The File Types</i> .....	<b>255</b>
<i>Moving Along</i> .....	<b>256</b>
<b>CHAPTER 26 - SIMPLE ANIMATION TECHNIQUES</b> .....	<b>257</b>
<i>The Program</i> .....	<b>257</b>
<i>The First Step</i> .....	<b>257</b>
<i>Programming the Animation</i> .....	<b>258</b>
<i>The Photon</i> .....	<b>259</b>
<i>Kaboom!</i> .....	<b>260</b>
Program Listing #1 .....	261

## INTRODUCTION

This book was four years in the making.

Whew! When I read that sentence, I feel like a refugee from the Twilight Zone. Four years! If someone back in 1986 had whispered in my ear that I would write a C programming tutorial totaling over 80,000 words and including hundreds of K of source code, I would have asked how he had escaped from his rubber room.

But here it is, in black and white: C-manship Complete. Where did all that time go?

### Some History

I had been working as a full-time employee of A.N.A.L.O.G. 400/800 Corp. (publishers of A.N.A.L.O.G. Computing and ST-Log magazines) for less than a year when the editors asked me to write a C programming tutorial for the ST. Being a rookie on the staff, and anxious to write as much as possible, I swore that they could count on me. Yes, indeed, I'd teach all those new ST owners to make their computers perform the most amazing tricks.

But, I mumbled to myself as I slinked back to my desk, who was going to teach me?

At that time, GEM and I were not on good terms. GEM was an intimidating beast that leered from the pages of poorly written documentation, page after page of obscure text through which I would have to muddle if I was to fulfill the challenge that had been laid before me.

Was I nervous? You bet! Back in those early days, only high-tech wizards knew anything about windows and dialog boxes. They locked themselves in dusty little rooms, and, shrouded in the glow from their monitors, tapped endlessly at their keyboards, while gulping gallons of Coke and munching bushels of Twinkies. They conversed in a secret language. Words like "workstation," "tedinfo," "raster," and "touchexit" fell from their lips in a stream of jargon that could bring other professional programmers to their knees.

I was terrified.

I realized, though, that I had something to offer that none of the high-tech wizards had: a novice's viewpoint. As I learned to tame the beast called GEM, I would stumble into all the traps, then learn to avoid them, immediately passing on what I had learned to my readers. We would learn together, the readers and me.

So, I set to work.

Some month's, particularly in the beginning, the job was easy: Take some notes. Write a sample program. Compose a tutorial. Hey, this wasn't so bad, after all! But there were months when producing a column was tougher than slogging through a room full of week-old jello. The research crawled. The programs bombed. I'd stare, perplexed and panicked, at a line of error messages as long as the source code for TOS, as my deadline, the demon of the magazine biz, slipped ever closer.

Nevertheless, I made every deadline. Yes, I know; there were months when "C-manship" was missing from ST-Log, but that was always due to situations beyond my control -- having to attend a trade show, for example. As my responsibilities grew, as I advanced from programmer to technical editor and, eventually, to executive editor, more and more installments of "C-manship" were missed. But never once because GEM pinned me to the mat.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

What's the point? If you're willing to apply yourself, you too can learn C and GEM programming. That's a promise. If an idiot like me can write this book, then a smart cookie like you can understand and apply it. Really.

### C-Manship, the Book

By the last issue of ST-Log (December, 1989), 31 installments of "C-manship" had been published. Each of those installments is included in this book.

Some chapters have been edited in order to bring the material up to date. For example, all the programs will now compile with either the original Megamax C or with Laser C. In addition, errors (at least those of which I was aware) have been corrected. Finally, in the course of editing this book, I deleted the reader's letters that started off some of the earlier columns.

Except for the above changes, each of the "C-manship" columns is presented in this book exactly as it was originally published in A.N.A.L.O.G. Computing and ST-Log, and in the same order. Even the illustrations have been reproduced (and some extra ones added).

Because "C-manship" was written as a series of monthly tutorials rather than as a book, you might find that the chapters jump erratically from one topic to another, rather than progressing smoothly forward in text-book fashion. That's okay. This book is a series of C programming experiments, not a C programming reference.

That's not to say that C-manship Complete can't be used as a reference. It can. By taking advantage of the index, you should have little difficulty finding information you need. C-manship Complete, however, is not organized as a reference book. (For a C programming reference, I recommend The C Programming Language by Kernighan and Ritchie, published by Prentice-Hall. The manual that came with your Laser C or Megamax C compiler makes a good GEM and TOS reference.) Even so, each chapter builds upon the information covered in previous ones. If you read the chapters in order, you'll always be prepared for the current topic. I promise you that, if you start with [Chapter 1](#) and read through to Chapter 31, studying the sample programs and doing the experiments, you will get a good grasp of both C and GEM programming.

### Some Important Details

Most of the sample programs in C-manship Complete are compatible with both Megamax C and Laser C. If you have a different compiler, you may have to make some changes to the source code to get it to run. However, because there are major differences between Megamax C and Laser C, most notably the 32K segment restriction with the former, a few of the sample programs will compile only with Laser C.

Further, although most of the sample programs will run fine on either color or monochrome systems, a few can be run only on one or the other.

In summary, all the programs in C-manship Complete are compatible with both Megamax C and Laser C, and with color or monochrome systems, with the following exceptions:

<a href="#">Chapter 10:</a>	Color suggested.
<a href="#">Chapter 13:</a>	Program 2, color only.
<a href="#">Chapter 20:</a>	Laser compatible only with header files changes.
<a href="#">Chapter 26:</a>	Color only.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Chapters 27-31: Laser C only.

If your system won't run all the sample programs, don't panic. Even though it's helpful to see examples in action, each chapter covers the material completely enough (sometimes even providing sample output) that you'll have no difficulty understanding the topic. Whether or not a program will run on your system, however, do study the source code carefully. Moreover, if a sample program is incompatible with your system, find the problem and correct it. Program debugging is an art that can be learned only through application.

### What about the Disks?

As you know, C-manship Complete is also available in a disk version. The disks are incredibly reasonably priced (only \$10 for two single-sided disks), so I would urge anyone buying this book to also get the disks.

But do you really need the disk version? The answer depends on your definition of "need." If you like to type -- and then debug -- a lot of program listings, everything required for every sample program is included in these pages. However, some of the program listings, particularly in the chapters on GEM, are extremely long. You would save a great deal of time by having the source code on disk.

Also, in addition to all the sample programs in the book (in source and compiled form), the disk version contains the complete MicroCheck ST home checkbook program, portions of which make up the final five chapters of this book. The complete documentation and source code for MicroCheck ST are also on the disk. In my humble (yeah, right) opinion, MicroCheck ST, which is a commercial-quality application, is itself worth much more than a measly ten bucks.

Bottom line: It's up to you.

### Let's Boogie

Writing C-manship has been one of the greatest challenges and pleasures of my life. I'm delighted that I have had this opportunity to share that challenge with you and thank you for the trust you've shown by buying this book. I hope you'll be pleased.

All set?

Then hoist the anchor, and let's set sail.

Clayton Walnum

August, 1990

## CHAPTER 1 - SOME BASICS

When it comes to programming languages for the ST, we have a lot of options, including BASIC (in many different "dialects"), LOGO, Pascal, Modula-2, assembly language and C. (There are a few others, but they haven't received widespread use.) Each of these languages has its advantages and disadvantages.

LOGO -- one of the languages that are packed with the ST -- is a good beginner's language, but has many limitations and is very slow. On top of that, few people are familiar with it (which makes its inclusion with the ST seem strange).

Another possibility is assembly language. If you happen to be familiar with the Motorola 68000's instruction set and have the time, patience and necessary documentation to make your ST perform its tricks, go to it! As for the rest of us? Next.

How about BASIC? This is a popular language for the ST, if for no other reason than it's an old friend. But in considering BASIC as a programming environment, one has to ask an important question: why are programmers of the 8-bit Atari machines, slowly but surely, abandoning BASIC and moving to Action!? Answer: because Action! provides the convenience of a high-level language with the speed of assembly language.

And guess what? There is just such a language available for use on the ST.

For those who haven't guessed the obvious, this language is C.

### Why C?

C is a high-level language that's compiled into machine language form. This means programs can be developed quickly and easily, but still retain the speed of machine language.

Also, C encourages the use of structured programming techniques. If that buzzword "structured" doesn't mean anything to you now, it will when you've finished learning about C. I promise you that, once you get accustomed to structured programming, you won't want to go back to the old "spaghetti code" of BASIC.

Another important characteristic of C is its compactness. There are only about thirty reserved words. This yields a language that is easy to learn, yet extremely powerful.

One of the qualities of C that has made it popular with professional software developers is its portability. Programs can be transferred from one machine to another with a minimum of effort. This means that, once a software package has been developed, it can be marketed for many machines with very little extra expense.

If none of the above makes an impression, consider that a large quantity of the software that is available for the ST was written in C. Does that tell you something?

### C, Wherefore Art Thou?

I'll assume at this point that you're all hopping up and down, anxious to start your first programming experiments with C. Unfortunately, if you go through all that packaging your computer was packed in, you'll quickly discover that there's nothing with "C" written on the label. That's right, folks. You're going to have to track down a copy on your own.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

There are many C compilers available for your machine, but the most popular ones seem to be Megamax C, Mark Williams C and the DRI compiler that comes with the Atari Developer's Kit. For the purposes of this book, I've chosen the Megamax compiler (and the new version of the Megamax compiler, Laser C). It's fast, fairly easy to use and is an excellent compiler to use for your first forays into C programming, due to its easy to use GEM interface. If you choose to use a compiler other than Megamax or Laser C, you may have to make some slight changes to the programs presented in this book in order to get the programs to run properly.

### Underway At Last

Now that we've gotten all the preliminaries out of the way, let's take a look at the way C programs are created.

C programs are written using a text editor. How sophisticated the editor is will depend upon the C package you're using. Since C source code is really nothing more than a text file, you can use many different word processing programs. The only restriction is that the text must be saved to disk without the extra codes that some word processors automatically add to your files.

Once the source code has been written, it must be "compiled." How complicated this process is depends, once again, on what software you're using. But essentially, during the compilation, the source file is read in from the disk and translated into an "object" file. This object file is stored on your disk in a form that your computer can understand (machine language). The object file is then "linked" with the other object files that may be needed, and the executable file (the runnable program) is written to your disk. Sounds easy, right? Good! Let's get on with it.

### A Simple Program

Get your text editor loaded up and type in the following code exactly as it appears here (Don't type the line numbers.):

```
1      #include <stdio.h>
2      main()
3      {
4          char ch;
5
6          printf("Press return\n");
7          ch = getchar();
8      }
```

Now compile and link the program (refer to your compiler's manual for instructions on how to do this) and run it. What happened? Bet you made some typing errors! You'll find that the compiler is very fussy. If you're used to programming in BASIC, you've been spoiled by getting syntax error messages immediately upon entering a new line of code. Life isn't so simple when you're dealing with a text editor. It will let you enter any kind of mumbo-jumbo. Your compiler, however, will do a lot of whining if it doesn't see exactly what it expects.

So go back to your text editor (unless you managed to get the program typed right the first time), and correct your source code; then try to compile it again. Got it?

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

When you run the program, the words "Press return" should be printed on your screen. Pressing Return will bring you back to the ST's desktop. Let's take a look at the code and figure out what's going on. I'll refer to the program lines by line numbers, even though C does not use line numbers. (That's why there are line numbers to the left of the source code lines.)

Line 1 tells the compiler to look on your disk for a file called "STDIO.H" and insert whatever code it contains into your program. This file is supplied with your compiler and contains input/output information for your ST. The filename stands for STandard Input/Output Header, and though not every program you write will need it, it's a good idea, until you really know what you're doing, to include it -- just in case.

Line 2 is a function name. C programs are made up of one or more functions, executed in a sequence specified by the programmer. Large programs will contain many functions, each doing a small part of the total job. Breaking a program up into small portions makes the programmer's job easier and results in source code that's more readable.

A function can be quickly identified by the parentheses which follow its name. In our example, the parentheses are empty, but this won't always be the case. Sometimes we may wish to send values to a function when we call it. These values are called "arguments," and the variables which will receive these values are placed within the parentheses. In our case, there are no arguments being passed, so the parentheses remain empty. If you're a bit confused, don't worry about it. We'll get into functions in greater detail later on.

The function `main()` is not your everyday, garden-variety function. Every C program must contain `main()`, for it's here that program execution begins.

Line 3 contains nothing but a left brace. It marks the beginning of our function. The body of every function must begin with a left brace and end with a right brace. All the program statements that make up the function fall in between.

Line 4 is a declaration statement. It declares a variable of the type `char` and gives it the name `ch`. Every variable in your C program must be declared before it's used. This allows the compiler to allocate the proper type of storage and supplies your computer with the information it needs to interpret the data properly.

The word *char* is a C keyword, a word that's been set aside for specific use within the language. Keywords may never be used for any other purpose, such as function or variable names. Here is a list of C keywords:

C Keywords		
<code>auto</code>	<code>Extern</code>	<code>short</code>
<code>break</code>	<code>float</code>	<code>sizeof</code>
<code>Case</code>	<code>for</code>	<code>static</code>
<code>Char</code>	<code>goto</code>	<code>struct</code>
<code>continue</code>	<code>if</code>	<code>switch</code>
<code>Default</code>	<code>int</code>	<code>typedef</code>
<code>do</code>	<code>long</code>	<code>union</code>
<code>Double</code>	<code>register</code>	<code>unsigned</code>
<code>else</code>	<code>return</code>	<code>while</code>

Notice, in our declaration of `ch`, the semicolon at the end. All program statements in C must be followed by a semicolon. But, wait a minute! What about Lines 1 and 2? They're missing their

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

semicolons! Not really. The former is a compiler directive, not a program statement, so it doesn't require a semicolon.

Function names are also excluded from the semicolon rule.

Line 6 is the equivalent of a print statement in BASIC. See the parentheses following the word `printf()`? What does this tell you? If you said that it has something to do with a function, you're absolutely correct. This line is a function call. The text inside the parentheses is the argument we wish sent to the function.

The astute among you may now be checking the program listing for a function called `printf()`. Don't bother. It's an additional C function that's added to your program when your program is linked. Your C compiler provides many "extra" functions like `printf()`. Though they are not really part of the C language, they are functions that are used a lot by programmers and so are supplied for your convenience. Also note that `printf()`, being a function name, isn't a keyword.

The argument for `printf()` is the text you want printed, plus any format control characters you wish to include. See the `n` with the backslash in front of it? This is the escape sequence that moves the cursor to the next line. If we hadn't inserted it in our text, the next line we printed would be on the same line as the first. There are five control characters that may be used with `printf()`. They are:

```
\n -- new line
\r -- carriage return
\t -- tab
\b -- backspace
\f -- form feed
```

When the compiler sees the backslash, it knows that it should interpret the next character as a control code. (Note that, even though each control appears as two characters on your screen, they're stored as a single character in the computer's memory.) What if you want a backslash in your text? No problem. There are three escape sequences to let you print characters which may confuse the compiler. They are:

```
\\ -- backslash
\" -- quote
\' -- single quote
```

See those parentheses in Line 7? That's right. We're calling another function. The function `getchar()` accepts a single character from the keyboard and is one of the functions defined in the `STDIO.H` file. Aren't you glad we included it? Here, we're taking the character returned by `getchar()` and placing it in the variable `ch`.

There is, however, one complication with `getchar()` that I should mention here. Because the ST's keyboard is buffered, `getchar()` won't return anything from the console until the return key is pressed, and then, of course, the character it gives back to us will be Return. To get a single character from the ST's keyboard, we have to have to resort to a call to the ST's operating system. But we'll save that discussion for later.

Something worth noting at this point is the way the expression `ch = getchar()` is evaluated. The equal sign is an assignment operator and doesn't mean "equal to." C has a separate operator, `==`, for the equal condition. The expression `a = b` should be read as "a gets the value of b." Contrast that with the statement `if a == b`, which is read "if a equals b."



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 8 is the right brace to mark the end of our function (and our program as well, in this case). The program has ended, and control is returned to your ST's operating system.

### Where's the Beef?

Okay, now let's look at something with a little more meat to it. Type the following code into your text editor and compile it.

```
#include <stdio.h>
main()
{
    char ch;
    int num1,num2,ans;
    printf("Enter two numbers: ");
    scanf("%d %d",&num1,&num2);
    ans = num1 + num2;
    printf("\n\n");
    printf("The sum of %d & %d is %d.",num1,num2,ans);
    printf("\n\nPress return\n");
    ch = getchar();
    ch = getchar();
}
```

When you run this one, you'll be prompted to enter two numbers. Enter them one after the other, separated by one space (5 10), and then press Return. Presto! Now your computer's doing first grade math. Let's see what's going on.

Lines 1 through 4 are now old hat. You should be able to figure them out with little difficulty. Line 5, though, gives us something new to chew on. Here, we're declaring three variables of the type integer, another of C's basic data types. C interprets an integer as any whole number from -32,768 to 32,767.

Notice I said whole number. No decimal portions are allowed.

Why are we restricted to this range? On the ST computer, an integer is stored in two bytes. That gives us 16 bits, or a maximum value of 65,535. Unfortunately, all 16 bits aren't used to store the value, since the most significant bit (bit 15 counting from left to right and starting with 0) holds the number's sign. It's set when the number is negative and cleared when the number is positive. With 15 bits, the maximum value that can be represented is 32,767.

If you need to use a larger integer value, you may declare the variable as unsigned int, or the abbreviated version, unsigned. This will free up the sign bit and allow any whole positive number up to 65,535. If this range is still not satisfactory, use the long int (abbreviated long) data type. This increases the length of the variable's storage to four bytes. Now you can work with numbers from about negative two billion to positive two billion. That big enough for you? Here are some declaration examples:

```
unsigned num1, num2;
long ans;
float num3;
```

You may declare as many variables on one line as you wish, as long as they're all the same type and are separated by commas.

Line 6 is our old friend printf(), only this time we've left off the newline character, so that whatever text is printed next will appear on the same line.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 7 introduces you to a new function, `scanf()`. This is an input function, and, in our case, it is looking for two numbers separated by spaces to be input from the keyboard.

The arguments for `scanf()` consist of a control string and a list of pointers. The control string may consist of white space characters (blanks, tabs, etc.) and conversion specifications. The pointers are the addresses where you wish the data to be stored.

Okay, okay, I'll slow down. First, let's look more closely at the control string.

The control string is the portion of the function call that appears between the quotes. There are two conversion specifications in our example, both of which tell the computer to expect the input of an integer. Take note of the syntax.

The control string is within quotes, just like any other string, and the conversion character `d` is preceded by the percent sign. Each control specification is matched with its corresponding argument. In other words, in our example, the first `%d` is paired with `&num1` and the second with `&num2`. The ampersand (`&`) tells the compiler we want the address of `num1` and `num2`, not their value. This is important. If you leave off the ampersands (and believe me, sooner or later you will), you're guaranteed to see those famous ST bombs appear on your screen. For instance, look at this program fragment:

```
num1 = 100;
num2 = 150;
scanf ( "%d %d", num1, num2 );
```

When we enter the data into `scanf()` in this example, the first value will be stored in memory at address 100 decimal and the second at address 150 decimal. Ouch!

A word of warning about `scanf()`: Many C programmers will not use this function because it assumes too much on the part of the person typing in the data and the person who wrote the `scanf()` routine. In other words, if the data that's input to `scanf()` is not exactly what the function expects to see, you may get unpredictable results. In following chapters, we'll see how to get around `scanf()` by writing our own input routines.

Getting back to the program, Line 8 is where we do the calculation, which brings us to a short discussion of data types. In C there are many different data types, including not only integer and floating point, but also long, short, unsigned and character. Some danger lies in the fact that C allows you to mix these data types with impunity. When you do, C will do type conversions on the numbers, and you may not end up with what you expect.

In this example, we're adding two integers, `num1` and `num2`, and assigning their value to a third integer, `ans`. Although we're not mixing data types, we're still not out of danger. We must be sure that the result of our calculation falls within the -32,768 to +32,767 range that we discussed earlier.

Hmmm. What happens if we try to add 1 to an integer that's already at its maximum value of 32,767? Can you guess? You'll end up with a result of -32,768. This is called an "overflow." You get a result of -32,768 because the integer wraps around from its highest value back to its lowest.

What will you end up with if you add 2 to an integer value of 32,767? If you guessed -32,767, you're right!

Be forewarned: C doesn't care about overflows and will not give you an error message.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Now that we've done our calculation, we have to get the answer out to the user. We do this with our old pal `printf()`.

Line 9 does nothing more than leave a blank line between the earlier text and the text we'll be printing next.

Line 10 actually prints out the final data. Take a good look at the control string. There are those conversion specifications again, only this time we're not accepting values from the keyboard; we're printing them to the screen. We put the conversion specification `%d` wherever we wish to have an integer printed in the text.

Following the control string are the matching arguments for the conversion specifications. Each `%d` pairs with an argument in the same manner as `scanf()`. You must be sure that the arguments match the control string properly, or you'll get unpredictable results.

There are nine basic conversion specifications you may use. They are:

```
%d -- decimal integer
%f -- floating point
%e -- floating point, scientific notation
%c -- single character
%s -- string
%g -- use the shorter of %f or %e
%u -- unsigned decimal integer
%o -- unsigned octal integer
%x -- unsigned hex integer
```

If some of these confuse you, don't worry about it. We'll get to them in due time.

Getting back to Line 10, when the text is printed, whatever you entered as the first number (now stored in `num1`) will be substituted for the first `%d`, the second number (`num2`) will replace the second `%d`, and the sum (`ans`) will be printed in place of the last `%d`.

Study this use of the `printf()` function closely until you understand it. You'll see it a lot, and many times it'll be much more complicated. Just remember that none of the conversion specifications are printed literally.

Lines 11, 12, and 13 bring us back to familiar territory. But why are we using the `getchar()` function twice? Remember when you entered those two numbers, then pressed Return? Well, the first number was stored in the variable `num1` and the second in `num2`, but the Return didn't have any place to go, so it stayed in the keyboard buffer. When we call `getchar()` the first time, the Return gladly jumps into our character variable `ch`, and the program goes on its merry way. If we didn't have the second `getchar()`, the program wouldn't pause for our next input.

Now spend some time writing a few simple programs using what you've learned. When you feel comfortable with the material presented here, move on to the next chapter.

## CHAPTER 2 - A LOOK AT STRINGS

In [chapter 1](#), we looked at how a simple C program is constructed. Along the way, we also learned a bit about the basic data types and got an introduction to the C functions `printf()`, `scanf()` and `getchar()`. Now that we've got all that mastered, it's time to learn how to handle strings. We'll also take a closer look at the `printf()` and `scanf()` functions.

Of course, you have a job to do first ... namely, typing and compiling the following program:

```
#include <stdio.h>
#define TEXT "Your full name is"

main()
{
    char ch;
    char fname [20], lname [20];

    printf( "Enter your first name: " );
    scanf( "%s", fname );
    printf( "\n\n" );
    printf("Hi, %s! Enter your last name: ", fname);
    scanf( "%s", lname );
    printf( "\n\n" );
    printf("%s %s %s.\n\n", TEXT, fname, lname);
    ch = getchar();
    ch = getchar();
}
```

Got it? When you run the program, you will be prompted to enter your first name. Type in your name, terminating the input with Return. You'll get a personal hello and be asked for your last name. When you enter it, the program will print some important information (your name), after which it'll wait for you to press Return to end the program. A program run will look something like this:

```
Enter your first name: Clay
Hi, Clay! Enter your last name: Walnum
Your full name is Clay Walnum.
```

### A Look at the Program

Line 1 instructs the compiler to add the contents of the `STDIO.H` file to our program.

Line 2 introduces us to the `#define` statement. The format of this statement is the word `#define`, followed by a symbolic name and the value we wish placed in the name. In our example, the symbolic name `TEXT` will contain the string constant `Your full name is`.

Since C doesn't provide the programmer with a special data type for strings, they are stored as an array of characters. The last character in this array will be the null character (zero). We don't have to worry about supplying the null character, though. It's added automatically. Here's a graphic representation of how the string constant `TEXT` looks in memory:

Y	O	U	R		N	A	M	E		I	S	\0
---	---	---	---	--	---	---	---	---	--	---	---	----

Your C compiler contains a program known as the "preprocessor." When you compile a program, the preprocessor searches for any occurrence of items that were defined by the `#define` statement.

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

Wherever it finds a match, it replaces the symbolic name with the value the name contains. In other words, in our program, every place the word TEXT appears, the string Your full name is will be substituted. Notice that there's no semi-colon at the end of a #define statement. It's a compiler directive and not subject to the semi-colon rule.

Here are a few other examples of the #define:

```
#define ZERO 0
#define PI 3.14159
#define PLUS +
```

Why do we bother with the #define statement? Why not just use a regular variable? After all, aren't the statements pi = 3.14 and #define PI 3.14 really equivalent?

No! The difference has to do with the preprocessor we discussed earlier. By using #define, your program will actually run a bit faster than if you used a variable name. The reason is that each time a variable is encountered in the program; its storage location must be "peeked" to get its value. This is called "run-time substitution." With #define, the substitution is accomplished during compilation (compile-time substitution), so that when the program is run, the values are already in place.

You're probably beginning to realize just how powerful the #define statement is. The following program listing shows an extreme use of #define. The program hardly looks like C anymore.

```
#include <stdio.h>
#define START {
#define STOP }
#define INPUT scanf
#define OUTPUT printf
#define TIMES *
#define EQUALS =
#define SQUARES main()
#define WAIT ch=getchar()

int _isconio;

SQUARES
START
    int num,ans;
    char ch;

    _isconio = 1;
    OUTPUT ( "Enter a number: " );
    INPUT ( "%d", &num );
    OUTPUT ( "\n\n" );
    ans EQUALS num TIMES num;
    OUTPUT("The square of %d is %d.", num, ans);
    WAIT;
    WAIT;

STOP
```

Notice that the constants defined in the #define statement are written in upper case. This is standard practice in C and makes it easy to distinguish our variables from our constants.

We can clean up the above program by putting all those #define statements in a separate file called NEWC.H -- then we delete them from the main program and substitute the statement #include <newc.h> in their place. When we compile the program, the contents of the file NEWC.H will be added to the main program.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 4 is our function name. Remember, all C programs must contain the function `main()`.

Line 5 marks the beginning of the function.

Line 6 declares a variable of type character.

Line 7 should look a bit strange to you. Here we're declaring two arrays of type character.

Remember, C doesn't have a data type for strings. We declare character arrays whenever we need a string.

In this example, we're declaring two character arrays, one to hold a first name and another to hold a last name. We've set aside 20 bytes for each, which means the longest name each array can hold is 19 letters (one byte is used for the null char).

The syntax for declaring an array is the name of the array followed by the number of bytes (within brackets) we wish reserved for it. As with any declaration, we may declare as many arrays as we like on one line, as long as they are separated by commas. Remember, the line must end with a semi-colon.

Line 9 brings us back to familiar territory. Here we are printing our first prompt.

Line 10 shows a new use for the `scanf()` function. The conversion specification `%s` tells `scanf()` to expect the input of a string. The corresponding argument is the address where we wish the string stored.

Notice something a bit different about this pointer? In the last chapter, when we were using this function to get integer values, each variable name was preceded by an ampersand, telling the compiler that we wanted the address of the variable not its value. There's no ampersand here, though. That's because array names are pointers. The value of `fname` is the address of the first byte of the array.

Line 11 prints a blank line after the first prompt.

Line 12 prints our second prompt, but with an extra something special. If you take a close look at this line, you'll see that `%s` again. In the argument list, you'll also note the array `fname`. The `%s` works just like `%d` except it tells `printf()` to substitute a string instead of an integer.

Line 13 calls `scanf()` again to allow input of the last name.

Line 14 prints another blank line.

Line 15 prints out our program's final message. Notice that there's no text in the control string -- only conversion specifications and newline characters. So where's all the text coming from?

Look at the control string. The function is being instructed to print three strings, followed by two newlines. The strings that'll be substituted for the conversion specifications are in the argument list. The first string will be the string constant `TEXT`, which we defined at the beginning of the program. The second and third strings are the first and second names we previously input with `scanf()`.

Line 16 and 17 wait for a keypress.

Line 18 marks the end of the program.

# C-MANSHIP COMPLETE – by CLAYTON WALNUT

## Some Fancy Stuff

Now that we know how to use `printf()` and `scanf()` in their most basic form, it's time to take a look at some of the tricks we can do with them.

All along, you've probably been wondering what the "f" in each of these function names stood for. Well, your wondering is over. It stands for "formatted." Both `printf()` and `scanf()` allow us to format input and output in various ways, as well as to do "type conversions." First, let's take a closer look at `printf()`.

In the last chapter I gave you a list of conversion specifications that could be used with `printf()` and `scanf()`. The output of `printf()` can be edited by adding conversion specification modifiers. Here are some examples:

```
%3d, %03d, %-5d, %ld, %5.3f.
```

The first of the specification modifiers above sets the minimum field width to 3. If the number or string is smaller than the minimum length, the field will be padded with spaces. If the data to be printed is larger than or equal to the minimum length, it'll be printed normally.

In the second modifier the leading 0 in the conversion specification causes the field to be padded with 0's rather than spaces. The -5 in the third causes the data to be left justified (-) in a minimum field length of 5. The fourth tells `printf()` that the matching data should be interpreted as long rather than int. The final example shows how to edit floating point numbers. Here, the data will be printed with a minimum field length of five and with three decimal places following the whole number portion.

The following is a program example that utilizes the above editing techniques.

```
#include <stdio.h>
main()
{
    int num=5555;
    char ch;

    printf ( ">%d<\n", num );
    printf ( ">%10d<\n", num );
    printf ( ">%010d<\n", num );
    printf ( ">%3d<\n", num );
    printf ( ">%-10d<\n", num );
    printf ( ">%f<\n", 3.14159 );
    printf ( ">%2.3f<\n", 3.14159 );
    printf ( ">%10.4f<\n", 3.14159 );
    printf ( ">%-10.4f<\n", 3.14159 );
    ch = getchar();
}
```

When the above program is run, the output looks like this:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
>5555<
> 5555<
>0000005555<
>5555<
>5555    <
>3.141590<
>3.142<
> 3.1416<
>3.1416 <
```

The arrows in the output mark the beginning and ending of each field. Notice that floating point numbers are automatically rounded when we limit the size of the decimal portion. Also, take a look at the way the variable `num` is defined in this listing. This form of the declaration allows you to assign a value to the variable immediately.

The following program uses a similar technique to format strings.

```
#include <stdio.h>
#define TEXT "strings"
main()
{
    char ch;

    printf ( ">%s<\n", "strings" );
    printf ( ">%10s<\n", TEXT );
    printf ( ">%-10s<\n", "strings" );
    printf ( ">%10.5s<\n", TEXT );
    printf ( ">%-10.5s<\n", "strings" );
    ch = getchar();
}
```

The output of the above program looks like this:

```
>strings<
>  strings<
>strings  <
> strin<
>strin  <
```

The formatting features work much the same way with strings as with numerical data. As a matter of fact, the only real difference between the two is that, when we use the precision modifier (the number after the decimal point), it refers to the number of characters we wish printed, rather than the number of decimal places.

### Type Conversions

One of C's handy -- and dangerous -- features is the ability to convert from one data type to another. I say "dangerous" because C doesn't check for type mismatch and will allow us to do all sorts of strange things without complaining in the least. If we're not careful, this can lead to some hard-to-find problems.

The `printf()` function won't complain either, as long as we have the right number of arguments. We can print our data out in just about any form we want. The trick is the proper use of the conversion specifications. If we have a decimal number we'd like printed in an octal form, we just use the `%o` conversion specification. We can even convert between decimal and character.

The following is an example of using `printf()` for type conversions.



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
#include <stdio.h>
main()
{
    char ch;

    printf ( "Decimal: %d\n", 100 );
    printf ( "Hexadecimal: %x\n", 100 );
    printf ( "Octal: %o\n", 100 );
    printf ( "Character: %c\n", 100 );
    ch = getchar();
}
```

The output of the program looks like this:

```
Decimal: 100
Hexadecimal: 64
Octal: 144
Character: d
```

### Odds and Ends

In these beginning chapters, we've taken advantage of many I/O functions, such as `printf()` and `scanf()`. These are handy for general use, but as we learn more about C -- especially about using C with GEM -- we'll outgrow them.

You should note that the C language really doesn't support I/O routines at all. The functions we've been using have been added for the programmer's convenience and shouldn't be considered an integral part of the language.

## CHAPTER 3 - LOOPING AND IF STATEMENTS

I HOPE YOU'VE KEPT up with your studying, because in this chapter we're going to get down to serious business. Looping structures are on our agenda, as well as more about functions. And, just so we end up with something useful, the sample program I've chosen incorporates a function that should prove useful in the future -- a sort routine.

### Onward

It's typing time again. Type in the program shown at the end of this chapter (page 30) and compile it.

When you run the program, you'll be asked how many numbers you wish to sort. Enter a number between 1 and 10, and then press Return. You'll be asked to enter each of the numbers. When you're done, the numbers will be sorted in ascending order and printed to the screen. A program run looks something like this:

```
How many numbers? 5
Enter number 1: 56
Enter number 2: 25
Enter number 3: 12
Enter number 4: 99
Enter number 5: 12
Sort complete!
12 12 25 56 99
```

### Digging In

Now let's take a good look at this program's innards. Since this one's much longer than any of the others we've done, you might want to number each line in your listing so you can follow the explanation more easily. I include blank lines when numbering.

Lines 1 through 6 are comments. A comment starts with `/*` and ends with `*/`. Everything the compiler finds between the two is as good as invisible. Comments allow us to document our programs within the source code itself.

Line 7 instructs the compiler to add the contents of the `STDIO.H` file to our program.

Line 8 instructs the compiler to add the contents of the `OSBIND.H` file to our program.

Line 9 defines the symbolic name `MAX` as 10. This is the maximum number of values to sort. Take a quick look at the listing. `MAX` is referenced in three places. If we didn't use the `define` statement, we'd have to substitute the number 10 for each occurrence of `MAX`. When we wanted a different maximum, we'd have a lot of changes to make. The `#define` allows a modification by simply changing the value assigned to `MAX` at the start of the program. See how handy this is? Imagine how much time it would save you if you were working on a thousand-line program.

Line 16 is a function name.

Line 17 marks the beginning of the function.

Line 18 declares the variable `num` as type integer.

Line 19 declares `val` as an array of type integer. Because we used the symbolic name `MAX` to dimension its size, this array will contain 10 elements, 0 through 9.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 20 declares the variable `ch` as type `character`.

Line 22 gives us something new to discuss. Here we're calling the function `how_many()`, Line 35, and assigning the value it returns to the variable `num`. This will be the number of items we want to sort (not to be confused with `MAX`, which is the maximum items). Notice that this function call has the same format as another we've used quite frequently, `ch=getchar()`. All function calls work the same, whether we're calling a library routine like `getchar()` or our own function.

Line 23 calls another of our functions, `get_nums()`. Since this function doesn't return a value, we aren't assigning its return to a variable. We simply call it by name, just like `printf()`. We do, however, have to pass arguments to the function: `num` (the number of values we wish to sort) and `val` (the beginning address of the array where we'll store the values).

Line 24 calls `sort()`, the function that does the sorting. This function doesn't return a value either, but it must be passed the same arguments as `get_nums()`.

Line 25 calls `output()`, the function that prints the sorted numbers to the screen. It requires the same arguments as the two previous functions.

Line 26 waits for us to press a key. This statement probably looks pretty alien to you. I'm going to ask you to take it on faith for now.

Line 27 marks the end of the function.

### The Golden Moment

We've now stumbled upon the perfect time to discuss structured programming techniques. Our function `main()` is constructed so that anyone can easily see what's going on. Each function call performs a logical step in the sequence of actions that must be completed in order to sort our numbers.

This type of construction matches the way people think. When you're going to make a lunch of beans and hot dogs, you don't consciously dwell over all the details in each step. Your thoughts would run like this: First heat the beans, and then boil the hot dogs and put them in the buns.

But there are many details you take for granted: what about taking the pans out of the drawer and placing them on the stove? Don't forget, you've got to open the can before you can get to the beans. And where did the hot dogs come from? Did you open the refrigerator? Who turned on the stove?

We don't worry about these minor details, because, if we did, we'd get so confused we'd starve. A programmer should think in this same structured way. Projects that seem impossible when we're mired in details become a snap when viewed from a more general viewpoint.

It's this form of thinking that's the essence of structured programming. To get our sort routine working, all we have to do is find out how many items there will be, get the items, sort them, and then print them out. At this point, we're not concerned with how we're going to do each of these steps. One thing at a time, slow and easy.

When we have the general logic worked out, then we can get into the details, taking each step and writing a function to accomplish it. In large programs, this process becomes even more important. Using structured techniques will make your job much easier and will result in very readable code.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

### Back to the Program

Line 35 is a function name. This is the function called from Line 22.

Line 36 marks the start of the function.

Line 37 declares the variable `n` as type integer.

Line 39 sets `n` equal to the value of `MAX+1`, or, in this case, 11.

Line 40 is the start of a while loop. This type of loop will repeatedly perform a statement or series of statements as long as the expression within the parentheses remains true. Here's another example:

```
while ( z > 2 && ch != 'e' )
```

This line is read: while `z` is greater than 2 and `ch` doesn't equal the letter "e." C uses some unusual character combinations for operators. The double ampersand (`&&`) is the equivalent to BASIC's AND. The `!=` is the symbol for "not equal to." It's the opposite of another operator we learned a while back, `==`. Remember the difference between `==` and `=`?

We're using a while loop here to insure the input of a value no larger than MAX. Looking back, Line 35 initializes `n`, the variable we're using in the conditional expression, to a value greater than MAX. If we didn't do this, we might not get a chance to enter our number. Whatever garbage happened to be in the memory location we labeled `n` would be used to evaluate the conditional expression. If it was less than MAX, the loop would be skipped and whatever value `n` happened to contain would be passed to the program.

If you don't initialize your variables, they'll contain whatever value happened to be in the address they were assigned.

The brace following the while statement marks the beginning of the statements within the loop. Whenever a loop contains more than one statement, the start and end of the loop are marked with left and right braces, just like a function. The braces are not necessary if a loop contains only one statement. Here's an example of a single statement while loop:

```
while ( x < 5 ) x = x + 1;
```

Line 41 prints a prompt.

Line 42 accepts a number from the keyboard and assigns it to the integer `n`.

Line 43 prints a blank line.

Line 44 marks the end of the loop. At this point, the value of `n` is checked, and if it's greater than MAX, the loop repeats. This will continue until the user enters a number less than MAX.

Notice the indenting of the statements that make up the loop. This isn't required, but makes our programs much more readable, by clearly delineating the body of the loop.

Line 41 introduces us to the return statement. Whenever a return is encountered, control is passed back to the calling function, along with the value in parentheses. The return may be anywhere within

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

the function. If you don't want to pass a value, delete the parentheses. In this case, we're sending the value `n` back to `main()`, where it will be stored in the variable `num`.

The variable `n` in `how_many()` is a local variable. It's created when the function is called and destroyed when control is passed back to the calling function. It has no relationship with other variables in the program (except maybe `num`, which will get only its value). You could even have another `n` elsewhere in your program without conflict.

Arguments in C are passed "by value" rather than "by reference." This means that only the values of the arguments are passed, not their addresses. The original values are safe from change. If we want to access a variable by reference, we must pass the address using a "pointer." We'll discuss pointers a little later on.

Line 46 marks the end of the function.

Line 58 is a function name. This function is called by Line 21. Notice something a little different here? There are two variables enclosed in the parentheses, which means two arguments are being passed from the calling function. The argument's values will be stored in `n` and `v` and are passed between the functions in the same order in which they appear in the function call; that is, `n` receives the value of `num`, and `v` receives the value of `val`.

Line 59 tells `get_nums()` that it should interpret the data being passed into `n` as an integer. All arguments within the function name's parentheses must be defined, and you must do so before the beginning brace.

Line 60 tells the function that `v` is an integer array. We're not dimensioning the size of `v`, since it's really the same array we dimensioned in Line 18 (`val[]`). How can that be? Aren't arguments in C passed by value, not address? So how can `v[]` be the same array as `val[]`? Why am I asking all these silly questions?

I'll tell you why. Because I'll bet you forgot that an array name without an index is an address. `val` is being passed as I described previously, but its value is the address of the array's first byte. What does this mean to us? It means that we're very definitely going to be monkeying with the contents of the original array. It's not safely protected from our clumsy fingers like `num` is.

Line 61 marks the start of the function.

Line 62 declares some local variables. These variables exist only in the function. They're forgotten the second we exit.

Line 64 gives you a look at a new looping technique. The `for` loop in C is very similar to the `FOR...NEXT` loop in BASIC. Its syntax is the keyword `for` followed by three expressions within parentheses which define the limits of the loop. The three expressions are separated by semicolons.

The first expression initializes the loop variable. In Line 56, we're setting `x` to 0. The second expression is the condition that controls the loop. As long as this condition yields a true result, the loop will continue executing. The third expression is the loop's step value or reinitialization. Line 56 in BASIC would look like this:

```
FOR X=0 TO N-1 STEP 1
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Of course, in BASIC we don't need the "STEP 1," since it's assumed. I just included it for purposes of clarity.

What do you think of that `++x` in Line 56? Got any ideas? This expression performs the same calculation as `x=x+1`. As a matter of fact, we can use either form of the expression in C, although the former is preferred because it's shorter. The two plus signs together form the C increment operator. There is also a decrement operator which is, of course, made up of two minus signs. These operators may be placed before or after the variable; however, there's a subtle difference. The expression `++x` increments `x` before the value is used. The expression `x++` increments `x` after the value is used. For example, let's say that `x` starts with a value of 1. Then `z = ++x` will set `z` equal to 2, whereas `z = x++` will set `z` equal to 1.

The brace following the `for` statement marks the start of the loop.

Line 65 prompts the user for a number. The prompt uses the value of `x` to tell the user which value he's entering.

Line 66 gets the number the user types from the keyboard and stores it in the variable `num`. Note that this variable has nothing to do with the variable `num` declared in `main()`.

Line 67 stores the number into the storage array's next element. In C, arrays are indexed just as in BASIC. In our first pass through the loop, `x` has a value of 0. Therefore, the first element of the array (in the context of our function, the first element is `v[0]`, but this is really the element of our original array, `val[0]`) gets the first number input. As `x` gets incremented, each consecutive element of the array is filled with its appropriate value.

Line 68 moves the cursor to the next line.

Line 69 marks the end of the loop.

At this point, `x` is incremented, and the control statement is evaluated. If the result is true, then another iteration of the loop is performed. This continues until the loop's condition evaluates to false.

Line 70 passes control back to the calling function, `main()`. There are no parentheses in the return statement because we aren't sending a value back.

Line 71 marks the end of the function.

Line 81 is a function name. This function is called from Line 22. The same arguments are being passed as in our call to `get_nums()`.

Line 82 defines the first argument as integer.

Line 83 defines the second argument as an integer array.

Line 84 marks the beginning of the function.

Line 85 defines some variables of type integer.

Line 87 initializes the variable used to evaluate the conditional expression in the following while loop. This ensures that we enter the loop properly.

Line 88 is the beginning of our while loop.

### Another Break in the Proceedings

Before we get too far into this function, I should give you a little background on the sort. We're going to use a "bubble" sort, one of the simplest (and slowest). It works like this: We compare the first two values in the array and switch them if they're in the wrong order. We then compare the second and third values and switch them if necessary. We continue this process until the last value in the array has been compared, at which point, the highest value will have been "bubbled" up to the top. We then start all over, repeating the loop until we make it through the array without a switch.

### Back To It

Line 89 marks the beginning of the body of the while loop.

Line 90 turns off the "switch" flag. If this variable retains the value of 0 through the next loop, then the sort is complete.

Line 91 sets up a for loop; we will use the loop variable for an array index. This will move us through the array, element by element.

Line 92 should be strangely familiar. This is C's version of the IF...THEN statement. Its construction is very similar to its BASIC counterpart, but there are two differences. First, the expression that follows the if is always within parentheses. Second, don't include the word "then." The body of the if statement is constructed following the same rules that apply to loops. If you have more than one statement, the entire block must be enclosed in braces. A single statement may be placed after the if statement with no braces.

Our if statement compares two consecutive elements of the array that contains the values to be sorted. If the first is larger than the second, the statements contained in the braces are executed, switching the two array elements. If they're already in the proper order, the switching is skipped. The next iteration of the for loop is then initiated.

Line 93 marks the beginning of the body of the if statement.

Line 94 is the first step of the switch. The value in v[x] is placed in temp.

Line 95 stores array element v[x+1] into v[x].

Line 96 places temp (originally v[x] ) into v[x+1], and the switch is complete.

Line 97 sets the "switch" flag to its true condition, so the loop will be performed again.

Line 98 marks the end of the if statement.

Line 99 marks the end of the while loop.

Line 100 returns control to the calling function -- in this case, main().

Line 101 marks the end of the function.

Line 111 is a function name.

Line 112 declares the first argument passed to the function.

Line 113 declares the second argument passed to the function.

Line 114 marks the beginning of the function.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 115 declares the integer variable x.

Line 117 prints a message.

Line 118 initiates a loop to print the sorted array values.

Line 119 prints the array values using the loop variable as an index.

Line 120 prints a blank line.

Line 121 returns control to main(), the calling function.

Line 122 marks the end of the function.

### Take a Breath

We covered a lot of material in this chapter. If you're still with me, pat yourself on the back; you've learned most of what you need to know to write usable C programs. In the next chapter, we'll continue studying, but we'll also have some fun.

### Program Listing #1

```
/* **** */
/*          C-MANSHIP          */
/*          Chapter 3          */
/*          Listing 1          */
/*          Developed with Megamax C          */
/* **** */
#include <stdio.h>
#include <osbind.h>
#define MAX 10

/* **** */
* main ()
*
* Main program
**** */
main ()
{
    int num;
    int val[MAX];
    char ch;

    num = how_many ();
    get_nums ( num, val );
    sort ( num, val );
    output ( num, val );
    Cconin ();
}
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* how_many ()
*
* Retrieves from user the # of values to be
* sorted and returns that value to main ().
*****/
how_many ()
{
    int n;

    n = MAX +1;
    while ( n > MAX ) {
        printf ( "How many numbers? " );
        scanf ( "%d", &n );
        printf( "\n\n" );
    }
    return ( n );
}

/*****
* get_nums ()
*
* Retrieves from user the values to be sorted
* and stores those values in the array v[].
* Input to the function is the number of values
* to be sorted and the address of the array in
* which to store the values.
*****/
get_nums ( n, v )
int n;
int v[];
{
    int x, num;

    for ( x=0; x<n; ++x ) {
        printf ( "Enter number %d: ", x+1 );
        scanf ( "%d", &num );
        v[x] = num;
        printf ( "\n" );
    }
    return;
}

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* sort ()
*
* Uses a bubble sort to sort the #'s stored in
* the input array. Input to the function is the
* number of values to be sorted and the address
* of the array in which the values are stored.
*****/
sort ( n, v )
int n;
int v[];
{
    int swtch, x, temp;

    swtch = 1;
    while ( swtch == 1 )
    {
        swtch = 0;
        for ( x=0; x<n-1; ++x )
            if ( v[x] > v[x+1] )
            {
                temp = v[x];
                v[x] = v[x+1];
                v[x+1] = temp;
                swtch = 1;
            }
    }
    return;
}

/*****
* output ()
*
* Prints the sorted values to the screen. The
* input to the function is the # of values to
* print and the address of the array where the
* values are stored.
*****/
output ( n, v )
int n;
int v[];
{
    int x;

    printf ( "Sort complete!\n\n" );
    for ( x=0; x<=n-1; ++x )
        printf( "%d ", v[x] );
    printf( "\n\n" );
    return;
}

```

### CHAPTER 4- FLOW OF CONTROL AND FUNCTIONS

Feeling lucky? Good. Get out all that green stuff that's been cluttering up your wallet and give Lady Luck a wink. This chapter we're all going to learn how to play craps. (I know that was top priority on your things-I've-got-to-do-today list.) If you haven't already done so, type in the program found at the end of this chapter and compile it.

Now I admit that our program isn't the most stunning version of computer craps that'll cross your eyeballs, but it's a good programming exercise and demonstrates a lot of new techniques. If you already know the rules of craps (that's where you've been all those late nights, huh?), skip ahead to the next section. For those who've led sheltered lives, craps is a dice game which has the dubious reputation for making and breaking many a fortune. In our case, we'll try to leave your savings intact -- only the rules remain the same.

Step one is to roll the dice. If on your first roll you get a seven or an eleven, you win. A two, three or twelve, on the other hand, leaves you the loser. If you manage to avoid all lucky and unlucky combinations, you must roll again...and continue to do so...until you either reroll your original number (in which case you win) or you roll a seven or an eleven (in which case, you lose).

#### The Game's Afoot (Without Toes)

Now that you know how to play, take a moment to try the program out. Have a little fun and get a general idea of how the program works.

Now let's take a look at the listing. You might want to number each line, so you can refer to them more easily as we go through the program. Count blank lines too.

I don't think it's necessary to go through every line as we have in the past. You've had most of the basics pounded into your head, right?

Let's skip up to Line 26. You've probably noticed that we usually use `ch` as a character variable. This time, however, we have it declared as an `int`. Does that mean that we've abandoned our poor friend `ch` to a new and unknown fate? No, we're still going to use it to hold character information, because it just so happens that the only difference between a character variable and an integer is the number of bytes they take up in memory.

You may remember that a character is stored in one byte and an integer is stored in two. For our purposes, the two are really interchangeable. What you should be aware of is that, in C, character variables are converted to integers for processing, and then truncated back to a single byte.

By declaring them as integers in the first place, you'll always be reminded of what's going on in your machine's innards. And you may come across a time in your illustrious programming career where the difference will be critical.

Now skip ahead to Line 30. This is the beginning of the main game loop. You remember the while loop, right? The variable we're testing, `play`, was initialized to 1 (or true) in Line 28. As long as it retains this value, the game loop will repeat.

Notice that we aren't using the statement `while (play == 1)`. Any non-zero value is evaluated to true, therefore `play==1` and `play` are really the same expression from a Boolean point of view (when `play`

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

does indeed equal 1); they are both true. The way to test for a false condition (0) is with the not operator: while (!play).

The game loop is another example of structured programming. Each major task of the program has been allotted to a function. First we roll the dice; then we check to see if the player won, lost or has to roll again. If the call to check\_roll() leaves the variable win in its zero state, then the second while loop is executed. The dice are rolled until win changes to 1 (win) or -1 (lose).

The variable win is then tested in an if statement, and the appropriate message is relayed to the player. The percentage of games won is calculated, and the player is asked if he wishes to play again. If he answers with a Y, then play remains true and the game loop repeats. Otherwise, play becomes false, and the program terminates, returning you to the desktop.

Now the details, starting with Line 30. Here we initiate the main loop. As long as the expression in parentheses is true, the loop will repeat. Since we initialized play to 1, we enter the loop. The first thing we have to do in the loop is initialize a couple more variables. This is important, since the values of first and roll are passed to the function that "rolls" our dice.

The variable first is used as a flag to indicate whether it's the player's first roll. What roll we're on is important. For example, a seven on the first roll is a winner, but a seven on the second roll is a loser. The variable roll will hold the value of the current roll (except the first one). Line 33 calls the function roll\_dice(), and the value returned is placed in first\_roll.

Line 34 calls check\_roll() and stores its return value in win. To evaluate the player's roll, check\_roll() needs some information; so we're passing the values of first, first\_roll and roll to the function.

Line 35 sets the flag first to its false condition. If the player neither won nor lost with his first roll, the value of win will still be 0, and the second while loop, which begins on Line 39, will be performed.

See the win==0? Why didn't we use the while (!win) construction mentioned previously? There's really no reason, as far as the program goes. I used the former construction to make the program more readable. Using !win might make someone looking at the source code think that if win was 0 the player lost. This isn't true. A value of 0 means that the player hasn't won -- and he hasn't lost either. It's a neutral state. If you want to use !win, go right ahead. It'll work just fine.

Line 37 calls roll\_dice() a second time. This time the variable roll is where its return value is stored. We need this second variable, since we need to compare the first roll with all subsequent rolls.

Line 38 calls check\_roll() again. If the value of win remains 0, meaning the player still hasn't either won or lost -- the loop repeats. Once the player has managed to make his roll -- or has blown it, with a seven or eleven -- we exit the loop.

Line 40 increments the game counter, num\_games. We'll use this value to calculate the percentage of games won.

Lines 41 through 45 make up an if statement. It uses the value contained in win to determine the appropriate message to give to the player, as well as keep track of the number of wins. If win is -1, the player has lost, and the program prints "You lose" -- deep, huh? If win equals 1, the player has won the game, and a statement of equal profundity is printed. Also, the counter num\_win is incremented, keeping track of the number of games our lucky player has managed to win.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

We're also calling a new library function here, `puts()`. This function prints the string argument contained in the parentheses. The main difference between `puts()` and `printf()` is that the former has no formatting options.

In the previous chapter we just touched on the format of the `if` statement. Now we're going to look at some more complex examples. The `if` statement starting on Line 41 is a slight variation of the one we saw before. The difference is the addition of the `else` portion.

Thinking back, you'll remember that the body of an `if` statement is performed only when the expression in the parentheses is true. When we add the `else`, the rules change just a bit. We now have a kind of "either/or" condition. If the expression being tested is true, the statements following the `if` and preceding the `else` will be performed. If the expression tested is false, the statements associated with the `else` are performed.

The syntax rules for the `else` portion of the statement are the same as for the `if`. If the body of the `else` portion consists of more than one statement, we must enclose them in brackets, and -- remember -- each statement must end with a semicolon.

Line 46 calls the function `percent()`, which prints out the percentage of games won.

Line 47 calls the function `play_again()` to find out whether the player wishes to continue or quit.

### Digging Deeper

Now that we've taken a look at the general scheme of things, we can get into the details of each function.

The function `roll_dice()` does exactly as its name implies. The first thing you should take note of is the way this function is declared. There's something extra here. See what it is? Up till now, our functions have been declared simply by the function name. Now the key word `int` has been added in front of the name. This specifies that the value to be returned by the function will be an integer. We could have left the `int` off, since the default is always integer.

But if we want to return some other data type from a function, we must declare the function at the top of our program, as well as add the data type specification to the function name itself. For instance, if we wanted to return a character from a function named `ret_char()`, we would first declare the function name, with its data type, at the top of our program like this:

```
char ret_char();
```

Then the beginning of the function itself might look like this:

```
char ret_char(l, b)
int l, b;
```

The variables `l` and `b` are the values being passed to the function and are included here only to differentiate between the two examples.

Lines 60 through 66 declare some local variables, print a prompt and wait for Return to be pressed.

Line 67 gets a random number and places it in `d1`. `Random()` is a function specific to the ST and is an extension of the BIOS (Basic Input/Output System). It returns a 24-bit random number. In our case, we need an integer. Take a good look at Line 74. See the `int` in parentheses? This is a "cast" operator. What we're doing is forcing the return of `Random()` into a 16-bit integer, rather than doing it implicitly through automatic conversion (just leaving the cast operator out). In this particular case,

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

the statement would have worked either way, but sometimes the difference can be critical. Look at these two code segments:

```
int i;  
i = 3.4 + 7.8;
```

```
int i;  
i = (int) 3.4 + (int) 7.8;
```

In the first example, the addition is performed, yielding a result of 11.2. Then, since the variable `i` is defined as an integer, the conversion from float to int is done automatically by truncation, making `i` equal to 11. In the second example, 3.4 and 7.8 are converted to integers before the addition is performed. This yields a result of 10. Not quite the same answer.

Line 68 takes the value in `d1` and converts it to a positive number between 1 and 6, using modulo arithmetic and the absolute value function. The `abs()` function is defined in the `STDIO.H` file. It looks and works exactly as in BASIC, returning the absolute value of a single argument.

The percent sign is the modulus operator. It is used only in integer arithmetic and yields the remainder when the number on the left is divided by the number on the right. For example, the expression `6 % 4` gives a result of 2.

So, in Line 68, we're taking the absolute value of `d1` (in case we got a negative number from `Random()`), dividing it by 6, then adding 1 to the remainder. Using 6 in the modulo math assures us we'll always get a remainder less than six (zero through five, to be exact). Adding one gives us our roll of the die (one through six).

Lines 69 and 70 get a value for the second die in the same manner. The function then prints out the value of each die, as well as the total. The total, `t`, is then passed back to `main()`.

Line 89 declares the function `check_roll()` as returning an integer. Three values are being passed to the function. Notice that the variables being passed (Line 34) and the variables accepting the values have the same names. This is purely for reasons of clarity. They're still completely separate identities.

Now look at the body of the function. This is surely the most complex piece of code we've tackled yet. Basically, the whole thing is an if statement, but with layer upon layer. This function will give you great insight into the problems inherent in nested if statements.

Before we get too far into this function, I should introduce you to the else if construction. I mentioned previously that, with the if...else statement, we have an either/or situation. The else if takes this one step further, and allows us to add a test to the else portion of the statement. Look at this example:

```
if (exp1) statement1;  
else if (expr2) statement2;  
else statement3;
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

If exp1 is true, statement1 will be executed and the elses ignored. If exp1 is false, exp2 is tested. If we get a true result, statement2 is executed and the final else is ignored. Finally, if both exp1 and exp2 are false, statement3 is executed.

In check\_roll(), we're using the flag first to decide which set of "rules" apply to the player's roll. If it's his first roll, first will be equal to 1, and we'll evaluate the second if statement, which checks to see whether the roll was a seven or an eleven. If it was, the player wins. The flag wn is set to 1, and the program continues at Line 103.

See those vertical bars in the middle of Line 96? That's the logical OR operator. Line 96 reads: if first\_roll equals seven or first\_roll equals eleven. The logical OR operator yields a true result if one or more of the expressions are true. Here are a couple of examples: If we assume that a equals 1, b equals 2, and c equals 3, the following expressions evaluate as shown:

a==1		b==6	TRUE		
a==4		b==2	TRUE		
a==1		b==2	TRUE		
a==2		b==5	FALSE		
a==3		b==3		c==3	TRUE
a==1		b==5		c==3	TRUE
a==2		b==3		c==4	FALSE

Continuing with check\_roll(), if the roll wasn't a seven or eleven, we evaluate the else if portion of the statement. Here we check for a two, three or twelve. If we find one of these values, the player loses. The flag wn is set to -1, and, as in the first case, program execution continues at Line 103. If neither of the previous conditions are true, wn retains its initialized value of 0 (Line 94), and once again, the program continues at Line 103, which returns the value of the flag to main().

Whew! All that's only if the player's on his first roll. If first is 0, program execution jumps to the else if statement on Line 101.

Before we continue, I'd like to see if I can help you avoid a good deal of teeth-gnashing and hair-pulling in your future C programming. Look at those brackets on Lines 95 and 100. They're absolutely essential with nested if statements containing else constructions. Without those brackets the compiler

has no way of knowing that the last two else if statements go with the outer if and not the inner. Keep in mind that the indenting is only cosmetic; it means absolutely nothing to the compiler. This is an easy trap to fall into, since the indenting makes everything so clear to the programmer. Just remember -- use brackets.

Now let's take the second possible path in this function. If first is 0, all the stuff between the brackets is skipped, and we continue at Line 101. This line checks to see whether the player's roll was equal to his first. If it was, wn is set to 1 (win), and its value is returned to main() at Line 103.

If the first condition isn't true, we drop down to test the second. Line 102 checks for a roll of seven or eleven. If it evaluates to true, wn is set to -1 (lose), and its value is returned at Line 103.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

If none of the above conditions are met, the only thing that happens in this function is that `wn` is set to 0 (Line 94) and its value is returned to `main()` (Line 103). The player has neither won nor lost and must roll again. This process repeats until `wn` -- and, subsequently, `win` -- gets a non-zero value.

Moving on, Line 112 begins the function `percent()`. The word `VOID` in front of the function name indicates to the programmer that the function doesn't return a value. Like the `int` in some of the previous functions, it could've been left off. `VOID` is really just an empty comment. In other words, even though we've labeled `percent()` as `VOID`, it's still capable of returning an integer value. We're writing it this way for the sake of clarity only.

This function does nothing more than calculate the percentage of games won and print the result out to the player. A few things should be said about Line 117, though.

First of all, in case it isn't obvious, the `/` (not to be confused with the backslash) is the division operator. The value on the left of the operator is divided by the value on the right. You'll notice that the integer variables `num_win` and `num_games` are being cast to floating point. This is critical in this calculation.

When we divide integers in C, we get an integer result; the decimal portion is truncated. If we allow this to happen with our percent calculation, we'll get two possible results, only one of which will be accurate. If we've won every game, `num_win/num_games` will give us 1, which multiplied times 100 equals 100%. Fine and dandy. But what happens if we've only won one game out of two? In integer division, `num_win/num_games` will give a result less than 1. When the decimal portion is truncated, we'll end up with 0. And what's 0 times 100? It's certainly not 50%, the result we want.

Okay, we're almost done. Just one more function to look at. The function `play_again()` is responsible for finding out if the player wants to play another game. There's really nothing very new here. Something that we had a brief encounter with was the way we're using `getchar()` in Line 133. We could rewrite this line as follows:

```
ch = getchar();  
if (ch == 'Y' || ch == 'y');
```

One of the neat things about C is the way we can cram a lot of stuff on one line. Here, `getchar()` is called, and its returned value is stored in `ch` and compared to the character "Y." The variable `ch` is also compared to the character "y." If either of these compares finds a match, the flag `p` is set to true and returned to `main()`, to be evaluated at Line 47. This way, the game repeats until the call to `play_again()` results in a 0.

### Breathing Time

That's it -- class dismissed. If any of the program is still fuzzy to you, study up on it, especially the function `check_roll()`. When you feel you've got it all down pat, try your hand at writing a simple game. How about that classic guess the number game? It should be fairly easy to write. Have the computer pick a random number between 1 and 100. As the player tries to guess the number, have the computer tell him whether he's too high or too low. When you've got the program working, follow me over to the next chapter.



# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```

/*****
*                               C-MANSHIP                               *
*                               Chapter 4                               *
*                               Listing 1                               *
*                               Developed with Megamax C               *
*****/

#include <stdio.h>
#include <osbind.h>
#define VOID /**/

/*****
* main ()
*
* Main Program
*****/
main()
{
    int first_roll,          /* Value of player's first roll. */
        win,                /* Win, loss or no change flag. */
        roll,               /* Value of player's rolls. */
        play,               /* Game play continue flag. */
        first;              /* First roll flag. */
    int num_win = 0;         /* Number of games won. */
    int num_games = 0;      /* Number of games played. */
    int ch;                 /* Single character storage. */

    play = 1;
    win = 0;
    while (play) {
        first = 1;
        roll = 0;
        first_roll = roll_dice();
        win = check_roll (first, first_roll, roll);
        first = 0;
        while (win == 0) {
            roll = roll_dice();
            win = check_roll (first, first_roll, roll);
        }
        ++num_games;
        if (win == -1) puts("You lose. ");
        else {
            ++num_win;
            puts("You win! ");
        }
        percent(num_games, num_win);
        play = play_again();
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* roll_dice ()
*
* Retrieves a random number from 1 to 6 for each die
* and calculates the total, reporting the sum to
* the player.
*****/
int roll_dice()
{
    int    d1,                /* Value of die 1. */
           d2,                /* Value of die 2. */
           t;                 /* Total of dice. */
    int    ch;                /* Character storage. */

    puts ("Press RETURN to roll:\n");
    ch = getchar();
    d1 = (int) Random();
    d1 = abs(d1) % 6 + 1;
    d2 = (int) Random();
    d2 = abs(d2) % 6 + 1;
    printf ("Die #1: %d ", d1);
    printf ("Die #2: %d\n\n", d2);
    t = d1 + d2;
    printf ("Your roll: %d\n\n", t);
    return (t);
}

/*****
* check_roll ()
*
* Checks to see whether the player has won, lost or
* must roll again. The input is a flag indicating
* whether this is the first roll of the game, the
* value of the first roll, and the value of the
* current roll if it applies. The output is a -1
* for a lose condition, 1 for a win condition, and
* a zero if the player must roll again.
*****/
int check_roll(first, first_roll, roll)
int first, first_roll, roll;
{
    int wn;

    wn = 0;
    if (first == 1) {
        if (first_roll == 7 || first_roll == 11) wn = 1;
        else if (first_roll == 2 ||
                 first_roll == 3 ||
                 first_roll == 12) wn = -1;
    }
    else if (first_roll == roll) wn = 1;
    else if (roll == 7 || roll == 11) wn = -1;
    return (wn);
}

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* percent ()
*
* Calculates & reports the percentage of games won. The
* input is the # of games played & the # of games won.
*****/
VOID percent (num_games, num_win)
int num_games, num_win;
{
    float pc;

    pc = ((float) num_win / (float) num_games) * 100.0;
    printf ("You've won %d %% of the games\n", (int) pc);
}

/*****
* play_again ()
*
* Asks the player if he wants to play again, & returns
* a boolean value based on his answer: yes=1 and no=0.
*****/
int play_again ()
{
    int p;
    int ch;

    puts ("Play again? ");
    if ((ch = getchar()) == 'Y' || ch == 'y') p = 1;
    else p = 0;
    puts ("\n\n");
    return(p);
}

```

### CHAPTER 5 - STORAGE CLASSES AND ARRAYS

Okay, Pass Your Homework to the front of the class. What was that? Did I hear someone in the back say, "What homework?"

For those who need their memories refreshed, in the previous chapter I suggested that you try writing a C version of a simple number-guessing game. You were to have the computer pick a number from 1 to 100; then allow a player to enter guesses. With each guess, the player was to receive a clue as to whether he was too high or too low.

My solution for this project is found in Listing 1. Does your program look something like this? Maybe, maybe not. At this early point in your C career, I think the following qualities are most important.

First of all, does your program work? If you can give me an affirmative, you've earned 70 points. At this stage of the game, getting programs up and running is a very large part of the battle.

Now, did you use a structured approach? Does the function `main()` concern itself with the major steps of the game, allotting details to other functions? If so, give yourself 20 more points. When you become more familiar with C, this area will be more pointworthy. In fact, eventually, an unstructured program will be an automatic zero. Strict, huh?

Finally, how readable is your code? Have you used indentation? Are there blank lines between each function? Did you use meaningful and descriptive names for your functions and variables? Are there enough comments? Do the comments adequately describe the purpose of the function? Another 10 points to those who've added these touches of elegance.

#### Game Time Again

Now that you've tallied up your homework score, type in Listing 1 and compile it. To play the game, run the program and follow the prompts. Everything work okay? Let's examine this program in detail.

`main()` is written in a manner that makes the program's general operation quite apparent. The details are taken care of in other functions. Put simply, the program is structured.

We start off by initializing the flag `play` to `TRUE`. This will get us into the while loop at Line 23. As long as `play` is true, this loop will repeat, allowing the user to play as many games as he wants without rerunning the program each time.

Once in the loop, we must initialize some variables. The counter turns tallies the player's guesses. The flag `win` tells `main()` when the player has made a correct guess.

After initializing the variables, we call the function `getnum()`, which returns a random number between 1 and 100. Next, since we had the forethought to initialize `win` to `FALSE`, we enter the while loop at Line 27. This loop will repeat until `win` becomes `TRUE`, keeping the player guessing until he comes up with the right number.

In the body of the loop, we increment the turn counter, get the player's guess and check if he's right. If he's not, `win` remains `FALSE` and the loop repeats. If the number has been guessed correctly, program execution drops through to Line 32, where the player is told how many guesses were made.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Line 33 calls `play_again()` to see whether the player wants to continue. If so, the flag `play` remains `TRUE`, and the outer while loop repeats. When `play` becomes `FALSE`, the program ends, and the user is returned to the Desktop.

Easy, right? You should've followed all of the above explanation with little difficulty.

The other functions are just as simple. The function `get_num()` uses the same method we incorporated last chapter in our dice game to get a random number. The only difference is that now we're getting a number between 1 and 100 rather than one between 1 and 6.

The function `get_guess()` incorporates a while loop, forcing the player to enter a number within the proper range. The loop will repeat until the gamester bends to our will.

The function `check_guess()` checks whether the player's guess was too high, too low or right on the money, and then prints the appropriate message. If the player has guessed right, then `wn` is set to `TRUE` (and thus `win` becomes `TRUE`, too), and the game is over.

Finally, the function `play_again()` asks whether the player wants another whack at it. Once again, we use a while loop to guarantee a proper response.

### Some Classy Information

Before we take a look at the next two listings, we need to discuss a fun topic called "Storage Classes." All the variables you define in your C programs have a storage class, whether you're aware of it or not. In our previous program examples, the storage classes were set automatically. We didn't have to concern ourselves with the details. That's all fine and dandy for a beginner, but sooner or later we're going to have to know how our variables are treated by the system.

There are four C keywords that refer to the storage classes. They are: `extern`, `auto`, `static` and `register`.

The keyword `extern` stands for external. Any variable that's not defined within a function, one that is external to the function, falls into this class. Both Listing 1 and Listing 2 contain examples. Notice the arrays `week[]` and `weeks[]`.

Unlike local variables that disappear once we're through with them, external variables may be accessed anywhere within your program. The only rule to remember is that, if their declaration appears in another file or after a function that refers to them, they must be declared as external in the function where they're used. Here's a declaration example:

```
extern int numbers;
```

Automatic (or `auto`) variables are those declared within a function. They remain healthy and happy as long as we stay within the function where they were declared. The moment we exit, they vanish into that great CPU in the sky. It's not necessary to declare these variables by their storage class (we never have, right?)--but, if you wanted to, this is what the declaration would look like:

```
auto int number;
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Variables of the class static are similar to automatic variables, except their values aren't forgotten when the function is exited. Don't try to access them in other parts of your program, though; they're still strangers there. Look at this code fragment:

```
main()
{
    for (x = 0; x < 5; ++x) counter();
}

counter()
{
    static int count = 1;

    printf("%d", ++count);
}
```

The output from this example would be:

```
2 3 4 5 6
```

Each time we call `counter()`, the variable `count` is incremented and printed out. If we hadn't declared `count` as a static variable, the output would have been a string of twos. Do you see why? When a static variable is initialized as we did in `counter()`, it receives its initial value the first time we call the function. Thereafter, the declaration and initialization is ignored. This is only logical, since what good would a static variable be if it was reinitialized each time we called the function?

By not declaring `count` as static, by default, it becomes automatic. Each time we call the function, it gets set to 1, and then it's incremented and printed. This gives us that string of twos.

One last note on static variables. An interesting variation of this class can be created by defining it outside any function. This type of variable is called external static. This class varies from regular external variables, in that it can be accessed only within the file where it appears and only in functions following its declaration.

The last class we need to discuss are register variables. They're defined like this:

```
register int number;
```

When we declare a register variable, we're requesting that the value be stored in one of the the computers registers where processing is much quicker. Notice I used the word requesting. If there's no register free in which to store our variable, it becomes an automatic variable.

### Hip, Hip Array!

We took a brief look at arrays when we wrote our sort program a couple of chapters ago. Now we're going to dig a little deeper.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

First, let's tackle Listing 2. Suppose you're selling a peculiar product called a whamble (a what?) in your small business. At the end of the week, you want to write a quick and dirty program that'll print the number of units sold that week. Listing 2 is just such a program. When you run it, your output should look like this:

```
Sales for day 1: 5
Sales for day 2: 7
Sales for day 3: 2
Sales for day 4: 10
Sales for day 5: 7
Sales for day 6: 1
Sales for day 7: 6
Total sales: 38
```

The first thing we must do in this program is initialize an array. In our sorting program, we didn't worry about that. All we did was declare the array, and then fill it, later in the program, with the numbers the user input. Sometimes, though, you'll need to have the array data stored and ready to process at run time. Line 7 of Listing 2 shows you how to do this.

To initialize an array as part of its declaration, the array name is followed by an equal sign, which, in turn, is followed by the elements of the array, separated by commas and placed between brackets. Here are some more examples:

```
int numbers[] = { 1, 2, 4 };
int numbers[3] = { 1, 2, 4 };
float numbers[] = { 1.1, 2.2, 4.4 };
```

The first is just like the declaration on Line 7, and the second example is, in this case, functionally the same as the first. However, it does present potential difficulties and can create some hard to locate errors.

For instance, in the first example the compiler automatically makes the array size the same as the number of values that follow. In the second example we're telling the compiler that, no matter what, we want a three-element array.

Here's an odd one:

```
int numbers[4] = { 1, 2 };
```

What do you suppose happens here? Well, the compiler sets aside an array containing four elements, then looks to see what we've got between the brackets. The first value goes into the first element, the second into the second. After that, if it's an external or static array, the remaining elements are set to 0. Otherwise, whatever garbage happens to be in those locations stays there. Trouble, for sure.

Here's another problem maker:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
int numbers[2] = { 1, 2, 4 };
```

There's no way you're going to get away with this. Your compiler is sure to present you with some snide comments on your programming skills -- and they'll be well deserved. You can't get three data items into a two-element array.

Continuing with Listing 2, after we've initialized our array, the program uses a for loop to access each element, add it to the total and print it out. Except for a little nuance with the way we've initialized the for loop, you've seen all this before. Just remember that an array starts at element 0.

Now, how about that nuance I mentioned? Look at Line 14. I hope you remember about for loops. The first expression in the parentheses is the initialization, the second is the loop control, and the third is the loop's step value.

In this example, we've taken the opportunity to initialize not only the loop variable, but the accumulator total as well. This is a handy way to set variables used within a loop to their starting values.

Line 15 offers a new assignment operator for your inspection, one that's quite similar to the increment and decrement operators. Line 15 does the same work as this line of code:

```
total = total + week[i];
```

The right side of the expression is added to the left.

### Another Dimension

C is also capable of handling multi-dimensional arrays. You can think of these as arrays of arrays. Listing 3 illustrates how to handle them.

The declaration is similar to that of a one-dimensional array, except we've added another set of brackets to tell the compiler how we would like the array set up. Look at Line 8. Here we're declaring an array with two sets of seven elements. You can think of this as a matrix with two rows and seven columns.

When we initialize the array, each row of data is placed within its own set of braces. The rows, just like the data within, are separated by a comma. Finally, the entire array is enclosed with another set of braces. This tells the compiler how we want each element placed. Take a look at this:

```
int a[2][3] = { { 1, 2 }, { 3, 4, 5 } } ;
```

Here, we've declared an array which contains two arrays of three elements each. But wait a minute! In our initialization, we're missing a data element for the first subarray. How's this going to work out? Is the first element of the second row going to end up as the third element in the first?



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Nope. The 1 will be placed in the first element of the first row. The 2 will go in the second. The third element of the first row will be initialized to 0. (Remember that rule about external data?) The second row will be initialized just the way we want it. No mix-ups.

To tell you the truth, you don't need all those extra braces. We could've initialized `weeks[][]` by placing all the data between one set of brackets, like this:

```
{ 3, 6, 7, 4, 3, 8, 9, 5, 3, 7, 9, 3, 2, 6 }
```

The array will still function properly, but it's much harder to see how the data's divided up -- and we've left ourselves open for possible errors. If we should accidentally (or deliberately, if you happen to enjoy that sort of thing) leave out one of the data elements, the compiler will no longer sort it out for us, making sure everything gets into its proper location. It'll assign each element consecutively until it runs out of data, and then initialize the rest to 0. Our program is then sure to act peculiarly. This type of error can be extremely difficult to locate.

### Whambles For Sale

Okay, enough talk. Compile Listing 3. A program run looks like this:

```
Sales for day 1: 3
Sales for day 2: 6
Sales for day 3: 7
Sales for day 4: 4
Sales for day 5: 3
Sales for day 6: 8
Sales for day 7: 9
Total sales: 40
Sales for day 1: 5
Sales for day 2: 3
Sales for day 3: 7
Sales for day 4: 9
Sales for day 5: 3
Sales for day 6: 2
Sales for day 7: 6
Total sales: 35
Total sales for month: 75
```

Two weeks; what a short month. Yes, I know there are usually four weeks in a month. The output was limited, to fit the screen.

This program is an example of indexing a two-dimensional array. Lines 17 and 18 set up nested for loops. The outer loop handles the indexing of the weeks; the inner loop indexes days.

The day loop is performed seven times for each iteration of the week loop. Line 19 shows how all this relates to our array. The first subscript in `weeks[w][d]` refers to each row of data (weeks). The second is the columns, or days. The first time we get to Line 19, `w` and `d` both equal 0, so we're looking at

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

weeks[0][0] -- that is, the data in row 0 and column 0. If we look at the array initialization, we see that this is the value 3.

The day's total sales are printed, and then the inner loop is repeated, incrementing d and advancing us to row 0's next element. Looking at the data, we see that weeks[0][1] equals 6. This loop repeats until d is no longer less than 7. At that point we drop through to Line 22 and print the total for the week, as well as add to our monthly total (in the next line).

When the program returns to the outer loop, w is incremented, and we re-enter the inner loop, resetting d to 0. Now we're referencing weeks[1][0], row 1 and column 0, or the value 5. The inner loop continues through row 1 just as it did with row 0.

When we return to the outer loop, the value of w is incremented again, and thus is no longer less than 2. The looping is completed, and program execution continues at Line 32 where the monthly total is printed.

That's it for this chapter. Sit back and relax. Put your feet up, massage your temples to get rid of that thundering headache. (Arrays are like that; yeah, they are.) Next chapter, we'll start developing our own input routines, so we won't be at the mercy of such functions as scanf(). In the meantime, fool around a bit more with arrays. They're neat little critters.

### Program Listing #1

```

/*****
*
*           C-MANSHIP
*           Chapter 5
*           Listing 1 - Developed with Megamax C
*
*****/
#include <stdio.h>
#include <osbind.h>
#define TRUE 1
#define FALSE 0

/*****
*   MAIN PROGRAM
*****/
main () {
    int num,                /* Number to guess. */
        guess,             /* Player's guess. */
        win,               /* Game end flag. */
        turns,             /* Number of guesses. */
        play;              /* Repeat game flag. */

    play = TRUE;
    while ( play ) {
        turns = 0;
        win = FALSE;
        num = get_num ();
        while ( !win ) {
            ++turns;
            guess = get_guess ();
            win = check_guess ( num, guess );
        }
        printf ( "It took you %d turns.\n\n", turns );
        play = play_again ();
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* get_num()
*
* Returns a random number from 1 to 100.
*****/
int get_num ()
{
    int n;

    n = (int) Random ();
    n = abs(n) % 99 + 1;
    return ( n );
}

/*****
* get_guess ()
*
* Retrieve a number from 1 to 100 from the
* keyboard.
*****/
int get_guess ()
{
    int g;
    g = 0;
    while ( g<1 || g>100 ) {
        printf( "Enter a number from 1 to 100: " );
        scanf ( "%d", &g );
        printf ( "\n\n" );
    }
    return ( g );
}

/*****
* check_guess()
*
* Compare the player's guess with the random
* number, and print the appropriate message. The
* input to the function is the original number
* and the player's guess. This function returns
* a value of TRUE if the number has been guessed,
* and FALSE otherwise.
*****/
int check_guess ( num, guess )
int num, guess;
{
    int wn = FALSE;

    if ( guess < num )
        printf ( "Too low\n\n" );
    else if ( guess > num )
        printf ( "Too high\n\n" );
    else {
        printf ( "You guessed it!\n" );
        wn = TRUE;
    }
    return ( wn );
}

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* play_again()
*
* Asks the player if he wishes to play again and
* returns a value of TRUE if he does or FALSE if
* he doesn't.
*****/
int play_again ()
{
    int ch, p;

    p = -1;
    ch = getchar ();
    while ( ( p!=TRUE ) && ( p!=FALSE ) ) {
        printf( "Play again? " );
        if ( ( ch=getchar () ) == 'y' || ch == 'Y')
            p = TRUE;
        else if ( ch == 'n' || ch == 'N' )
            p = FALSE;
    }
    printf ( "\n\n" );
    return ( p );
}

```

### Program Listing #2

```

/*****
*                               C-MANSHIP                               *
*                               Chapter 5                               *
*       Listing 2 - Developed with Megamax C                           *
*****/
#include <stdio.h>
int week[] = { 5, 7, 2, 10, 7, 1, 6 };
main ()
{
    int i,                      /* Loop variable. */
        total,                  /* Sum of weekly sales. */
        ch;                     /* Character storage. */

    for ( i=0, total=0; i<7; i++ ) {
        total += week[i];
        printf ( "Sales for day %d: %d\n", i+1, week[i] );
    }
    printf ( "\n" );
    printf ( "Total sales: %d", total );
    ch = getchar ();
}

```

## Program Listing #3

```

/*****
*
*           C-MANSHIP
*           Chapter 5
*           Listing 3 - Developed with Megamax C
* *****/
#include <stdio.h>
int weeks[2][7] = { { 3, 6, 7, 4, 3, 8, 9 },
{ 5, 3, 7, 9, 3, 2, 6 } };
main()
{
    int    w,           /* Loop variable--weeks. */
          d,           /* Loop variable--days. */
          mtot,        /* Weekly total. */
          wtot,        /* Monthly total. */
          ch;          /* Character storage. */

    for ( w=0, mtot=0; w<2; w++ ) {
        for ( d=0, wtot=0; d<7; d++ ) {
            wtot += weeks[w][d];
            printf ( "Sales for day %d: %d\n", d+1, weeks[w][d]);
        }
        printf ( "\n" );
        printf( "Sales for week %d: %d\n\n", w+1, wtot );
        mtot += wtot;
    }
    printf ( "\n\n" );
    printf ( "Total sales for month: %d\n", mtot );
    ch = getchar ();
}

```

## CHAPTER 6 - FILE HANDLING AND CUSTOM INPUT ROUTINES

This chapter, as I promised, we'll get busy designing our own input routines. We're no longer going to suffer with the limitations of such library routines as `scanf()`. And, to add a little spice to the proceedings, how about learning a little about disk file handling?

Listing 1 is this chapter's sample program. Type it in and compile it. The program is an embarrassingly simple text editor. When you run it, you'll be asked for a filename. If the filename you enter already exists on the disk, you'll be asked if you wish to delete the file. If you answer Y, the file will be deleted and a new one created. Any other response will let you enter a different filename.

The text is entered one line at a time. When you reach the right margin (medium resolution), press Return for the next line. If you try to type beyond the right margin, the program will automatically terminate the line. You'd be wise to avoid this, since the last character you typed will be lost. You should also check each line for typos before pressing Return. There are no editing features (except backspace) in this program.

Press CTRL-Z (that's the Control key and the Z, simultaneously) to close the file. You may then print or view the text from the GEM desktop, by double-clicking the file you created.

### The Innards

There's nothing fancy going on in this program -- just a couple of new functions to learn and, most importantly, a new method for accepting input. No more `scanf()`. From now on, every key will be under our control. First take a look at the `#define` statements at the top of the program. `MAX` is the length limit for each line. `RETURN`, `BACKSP` and `CTRL_Z` are the ASCII values for some of the keys we'll be checking for in our input routine. Don't pay any attention to `NOFILE` right now; we'll get to that later. Notice, also, that here we're declaring an integer variable, `code`. Since it's defined outside of any function, it's a global variable -- one we can access from anywhere in the program.

If you look at the function `main()`, you'll see that we've declared two character arrays, `filename[]` and `text[]`. The first will hold the name of the disk file we'll be working with; the second will store each line of text as it's typed.

The body of `main()` consists of only three statements. These represent the activities we must complete to create our text file. The function call at Line 27 will open our file; Line 28 will allow us to enter our text; and Line 29 will close the file. And you thought programming in C would be tough. Only three function calls!

Well, if you've been following the lessons carefully, you're aware that `main()` is only the general outline of the program; the trickier stuff is still to follow. But don't get panicky. Handling files in C is a snap, not much tougher than in BASIC.

### Doing it Our Way

In the past, we've been at the mercy of C's built-in I/O functions. Actually, these functions are not part of C at all. They're small routines other programmers have put together, then gathered into a library for our convenience. It's nice to have these functions lying around in case we need them, but there's always a price to pay when we take a shortcut. The price is a loss of flexibility.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

If we use library functions like `scanf()`, we have to follow the rules somebody else made up. Frequently, these rules will be at odds with what we wish to accomplish. The solution? Write our own input routines, using our own set of rules.

This might sound a bit scary, but, depending on how fancy we want to get, there's really nothing to it. For our simple text editor, we don't need to convert strings to decimal values or perform any of the other tricks a complete input routine must be capable of. All we have to do is let the user type in one character after another, terminating his line with a Return.

In Listing 1, down near the bottom, you'll see a function called `get_str()`. This is our input routine. The body of the function is only 12 lines long (not counting comments) -- a veritable piece of cake. As you can see by the function declaration, `get_str()` receives one argument from the calling function: the address of the character array where we wish the string stored.

We start off at Line 124 by initializing `n`, our array index, to 0. Then, in order to slip neatly into the while loop at Line 130, we get our first character from the console (the ASCII value), utilizing one of the GEMDOS functions, `conin`. Note that this function is not a part of C; rather it is a call to the ST's operating system. If we were to try to port this program to another system, we would have to replace the call to `conin` with the new machine's equivalent function.

What's all this GEMDOS stuff? The ST's operating system (OS) is called TOS, right? It even says so right there on my old boot disk. T-O-S. Well, TOS is an incredibly complex animal, made up of two main parts: the BIOS (Basic Input/Output System) and GEMDOS (actually, there's also the XBIOS, but that's just an extension of the BIOS). The BIOS is the lowest level of the OS, and handles all the ST's primary input/output functions.

You can think of the BIOS as the software that runs the hardware, the meat in the sandwich between GEMDOS and all those data buses and microchips. GEMDOS provides the programmer with convenient access to the BIOS.

GEMDOS supplies over fifty functions, of which `conin` is function number one. In upcoming chapters, we'll be exploring GEMDOS in more depth.

Notice that in Lines 130 and 154 we're calling a function named `Cconin()`. This is the function that will get us those keystrokes. What happened to `conin`? One of the files we included at the beginning of our program was `OSBIND.H`. If you get a print out of this file, you'll see that it's nothing more than a long list of `#define` statements. About half-way down, you'll see this statement:

```
#define Cconin() gemdos(0x1)
```

You should be familiar with how the `#define` statement works. Wherever the compiler sees the word `Cconin()` in our source code, it'll substitute `gemdos(0x1)`. The word `conin` is just a name someone came up with for GEMDOS function 1.

To access this function we must use the call `gemdos(0x1)`. (Don't let the "0x" in front of the function number throw you off. It just means the number should be interpreted as hexadecimal, rather than decimal.) Using names like `Cconin()` for GEMDOS functions reminds us of what the function does. We could have put the call `gemdos(0x1)` directly into our source code and not bothered with including `OSBIND.H`.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

### A Bit of Construction

The function `get_str()` begins on Line 119. All this function does is get characters one by one and place them in successive bytes of the character array. There's a small complication, however. Several keys have special functions. For instance, Return ends a line, CTRL-Z closes the file, and the backspace key allows the user to correct mistakes. We'll have to check for these keys as the user types.

At Line 124, we initialize the array index `n`. We then get our first character and slip into the while loop that follows. The loop checks for a Return or a CTRL-Z and makes sure we haven't gone past the end of our array.

Line 133 checks for a Backspace. If we didn't get one, the character that was typed is added to our array, `text[]`. Line 134 accomplishes this, as well as incrementing the index `n` (notice that `n` is being post-incremented; that is, the array is first indexed by `n`, then `n` is incremented). Program execution then drops down to Line 151, where we get our next keystroke.

If a Backspace is entered, and we have at least one character in the array, we replace the last character typed with a null (Line 141). We also have to adjust the screen display. This is done in Lines 144 and 147. Since the cursor was moved on top of the last character in the line when the Backspace was typed, all we have to do is print a space (Line 143), then place the cursor back in its proper position by printing a Backspace to the screen (Line 147). To print these characters to the screen, we're using `Cconin()`'s counterpart, `Cconout()`, which writes a single character to the screen.

Sooner or later, the user will type a Return to end a line or a CTRL-Z to close the file, at which point we exit `get_str()`.

### Disk Files

Fortunately for us struggling programmers, there are many functions for handling disk files. Four of these functions concern us at the moment. They are: `open()`, `creat()`, `write()` and `close()`.

The function `open()` opens a file already in existence. It requires two arguments: the address of the filename and the type of access required. The latter may be one of three values: 0 (read only), 1 (write only), or 2 (read and write). We can add 8192 to any of these three values in order to open the file in "untranslated" binary mode. Untranslated means that the data is interpreted as a continual stream of bytes, rather than lines ending with carriage returns and line feeds. The difference between the two modes can be critical, depending on our usage.

The function `open()` also returns a value. If it encounters an error and fails to open the file (the file didn't exist), it'll return a -1. Now you know why I defined `NOFILE` at the top of the program equal to this value. If the file is opened successfully, the function will return a file descriptor. We'll use this number whenever we wish to access the file.

The function `creat()` starts a new file and also requires two arguments: the address of the filename and a flag value. The flag must either be 0 or 8192, the latter meaning we want the file created for use in the untranslated mode. If, when we call this function, the file we wish to start already exists on the disk, the file's pointer will be moved to the beginning of the file, effectively deleting it. Just like `open()`, a -1 is returned in the case of an error, or a file descriptor if successful.



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The function `write()` saves data to a file. It requires three arguments: the file descriptor, the buffer starting address (where the data is stored) and the number of bytes to write. A successful write will return a value equal to the number of bytes actually written. Otherwise, a `-1`, indicating an error, will be returned.

The function `close()` closes a file and requires the file descriptor as its argument. If the file is closed successfully, a `0` will be returned. An unsuccessful close, meaning we used an unknown file descriptor, will yield a `-1`.

### Starting Our File

Look at the function `start_file()` in Listing 1. It receives one argument from `main()`, the address of the character array, `filename[]`. This will be the first argument for `open()` and `creat()`. The variable `file` will hold our file descriptor and is initialized to `-1` (Line 47), so we can get into the while loop that follows. As long as `file` is equal to `-1`, this loop will repeat, prompting the user for a filename until a file is successfully created.

Within the loop, we print a prompt, then call `get_str()` to allow the user to input the filename. At Line 57, if the file descriptor we receive from `open()` equals `-1`, we know the file doesn't already exist, so we go ahead and create it (Line 60).

If we get a value other than `-1`, it means there's already a file by that name on the disk, and the program continues at Line 67. Here we reinitialize `file` to `-1`, then ask the user if he wants to delete the file. If he answers yes, the old file becomes the new file (Line 73), otherwise the loop repeats, asking for another filename.

### Writing Our File

Now let's study the function `get_text()` in Listing 1. You should have little difficulty figuring it out.

First we prompt the user to input his text; then we initialize code (the global variable that'll contain the ASCII value of each keystroke) to `0`. We then call `get_str()` to get the first line of text. This function will return the number of characters typed.

In Lines 103 and 104, a line feed and a null are added to the string (otherwise, when we try to print the file, the lines will be concatenated). Finally, in Line 107, we write the text to disk. We repeat the while loop until code equals `26` (a CTRL-Z), at which point the function terminates, and the file is closed at Line 29.

### Simple, but Cute

So there you have it. There's not much to this program, but it can be useful for creating small README.DOC files for your disks. It's certainly easier than loading up a full-fledged word processor when all you want to do is type in a couple of lines. Most importantly, we now know how to save data to disk and how to get input from the user without relying on such undependable functions as `scanf()`.

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```

/*****
*
*           C-MANSHIP
*           Chapter 5
*           Listing 1 - Developed with Megamax C
* *****/
#include <stdio.h>
#include <osbind.h>
#define TRUE 1
#define FALSE 0

/*****
*   MAIN PROGRAM
*****/
main ()
{
    int    num,                /* Number to guess. */
           guess,             /* Player's guess. */
           win,               /* Game end flag. */
           turns,             /* Number of guesses. */
           play;              /* Repeat game flag. */

    play = TRUE;
    while ( play ) {
        turns = 0;
        win = FALSE;
        num = get_num ();
        while ( !win ) {
            ++turns;
            guess = get_guess ();
            win = check_guess ( num, guess );
        }
        printf ( "It took you %d turns.\n\n", turns );
        play = play_again ();
    }
}

/*****
*   get_num()
*
*   Returns a random number from 1 to 100.
*****/
int get_num ()
{
    int n;

    n = (int) Random ();
    n = abs(n) % 99 + 1;
    return ( n );
}

/*****
*   get_guess ()
*
*   Retrieve a number from 1 to 100 from the
*   keyboard.
*****/
int get_guess ()
{
    int g;
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
g = 0;
while ( g<1 || g>100 ) {
    printf( "Enter a number from 1 to 100: " );
    scanf ( "%d", &g );
    printf ( "\n\n" );
}
return ( g );
}

/*****
* check_guess()
*
* Compare the player's guess with the random
* number, and print the appropriate message. The
* input to the function is the original number
* and the player's guess. This function returns
* a value of TRUE if the number has been guessed,
* and FALSE otherwise.
*****/
int check_guess ( num, guess )
int num, guess;
{
    int wn = FALSE;

    if ( guess < num )
        printf ( "Too low\n\n" );
    else if ( guess > num )
        printf ( "Too high\n\n" );
    else {
        printf ( "You guessed it!\n" );
        wn = TRUE;
    }
    return ( wn );
}

/*****
* play_again()
*
* Asks the player if he wishes to play again and
* returns a value of TRUE if he does or FALSE if
* he doesn't.
*****/
int play_again ()
{
    int ch, p;

    p = -1;
    ch = getchar ();
    while ( ( p!=TRUE ) && ( p!=FALSE ) ) {
        printf( "Play again? " );
        if ( ( ch=getchar () ) == 'y' || ch == 'Y' )
            p = TRUE;
        else if ( ch == 'n' || ch == 'N' )
            p = FALSE;
    }
    printf ( "\n\n" );
    return ( p );
}
```

## Program Listing #2

```

/*****
*                               C-MANSHIP                               *
*                               Chapter 5                               *
*       Listing 2 - Developed with Megamax C                           *
*****/
#include <stdio.h>
int week[] = { 5, 7, 2, 10, 7, 1, 6 };
main ()
{
    int    i,                /* Loop variable. */
    total,    /* Sum of weekly sales. */
    ch;      /* Character storage. */

    for ( i=0, total=0; i<7; i++ ) {
        total += week[i];
        printf ( "Sales for day %d: %d\n", i+1, week[i] );
    }
    printf ( "\n" );
    printf ( "Total sales: %d", total );
    ch = getchar ();
}

```

## Program Listing #3

```

/*****
*                               C-MANSHIP                               *
*                               Chapter 6                               *
*       Listing 1 - Developed with Megamax C                           *
*****/

#include <stdio.h>
#include <osbind.h>

#define RETURN 13
#define BACKSP 8
#define MAX 78
#define NOFILE -1
#define CTRL_Z 26

int code;

/*****
* MAIN PROGRAM
*****/
main()
{
    char filename[15], /* Filename for text file. */
    text[MAX]; /* Text entered by user. */
    int file; /* File ID. */

    file = start_file ( filename );
    get_text ( file, text );
    close ( file );
}

/*****
* start_file ()
*
* Gets the filename from the user, and then opens the

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
* file. The input to the function is the address of
* storage for the filename. The function's output is
* the open file's file ID.
*****/
start_file ( filename )
char filename[];
{
    int file,          /* File ID. */
        ch;           /* Character storage. */

    /* Initialize file ID to error condition. */
    file = NOFILE;

    /* Continue trying to open a file until successful. */
    while (file == NOFILE) {
        /* Get filename from user. */
        printf ( "Filename: \n" );
        get_str ( filename );
        /* Check if file already exists. */
        if ( ( file = open(filename,2) ) == NOFILE )

            /* If it doesn't exist, create it. */
            file = creat ( filename, 0 );
        /* The file the user wants to open already exists. */
        else {

            /* Reset file ID to error condition in case user */
            /* doesn't want to delete already existing file. */
            file = NOFILE;

            /* Find out if user wants to delete the existing */
            /* file and create a new one. */
            printf ( "File already exists! Delete it? " );
            if ( (ch = getchar() ) == 'Y' || ch == 'y' )
                file = creat ( filename, 0 );
        }
    }
    printf ( "\n" );
    return ( file );
}

*****/
* get_text ()
*
* Get the text to be stored in the file from the user.
* The inputs to the function are the file's ID and
* the address of string storage.
*****/
get_text ( file, text )
int file;
char text[];
{
    int num_char; /* Number of characters in string. */

    printf( "Type your message:\n\n" );
    code = 0;

    /* Get text from keyboard until user wants to exit. */
    while ( code != CTRL_Z ) {
        /* Get a string of text. */
        num_char = get_str ( text );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        /* Add a LF and a null to the string. */
        text[num_char++] = '\n';
        text[num_char] = '\0';
        /* Save the text string to the file. */
        write ( file, text, num_char );
    }
}

/*****
* get_str ()
*
* Gets each of the strings that make up the text file.
* The input is the address of the string storage. The
* output is the number of characters in the string.
*****/
get_str ( text )
char text[];
{
    int n; /* Character count. */
    n = 0;

    /* Get character code from keyboard. */
    code = Cconin();

    /* Check if end of string or end of text. */
    while ( code != RETURN && code != CTRL_Z && n <= MAX ) {
        /* Add character to string if not a backspace. */
        if ( code != BACKSP ) {
            text[n++] = code;
        }
        /* Handle backspace character. */
        else if ( n > 0 ) {

            /* Shorten string by one character. */
            text[--n] = '\0';

            /* Erase character on screen. */
            Cconout ( ' ' );

            /* Move cursor back one space. */
            Cconout ( BACKSP );
        }
        /* Get next character code. */
        code = Cconin ();
    }
    printf ( "\n" );
    return ( n );
}
```

## CHAPTER 7 - POINTERS AND MACROS

Handling pointers can be confusing at times, though the basic concept is quite simple. Believe it or not, we've been using pointers for several chapters now, whenever we referred to an array name.

What exactly is a pointer? Simply put, it's a variable containing the address of a data item we wish to access. For example, look at this line of code:

```
pointer = &var;
```

After this line has been executed, pointer will contain the address of var, or to say it another way, pointer will point to that section of memory where the value of var is stored.

So, what's all the hoo-ha? Why not use the & operator and be done with all this nonsense? Because there's a subtle difference between pointer and &var. The first is a variable; the second is a constant. Still not impressed? Okay, let me ask you a question: what makes variables so handy? Give up? We can perform mathematical procedures on variables; not so with constants.

Another advantage to pointers is that, when declared properly, they're much "smarter" than constants or run-of-the-mill variables. We'll see why in a minute.

### A Point of Declaration

In order for us to use a pointer, the compiler needs some information, namely the type of data the pointer points to. We supply this information in the pointer's declaration:

```
int *p1;  
char *p2;  
float *p3;
```

The first example above tells the compiler that we want a pointer to an integer value. The second sets up a pointer to character data. The third points our way to floating point information.

Each of these data types (as well as others) is stored in a special way in memory. A pointer to integer won't function as we expect if we try to use it on character data. The "\*" before the name identifies the variable as a pointer and requests "special handling" from the compiler. Don't confuse this symbol with the multiplication operator.

Once we've declared our pointer, we have to assign it a value. We want it to point to something, don't we? We assign an address to a pointer in exactly as we would to any other variable. Take a look at this segment of code:

```
int var, array[10];  
int *p1, *p2, *p3;  
  
p1 = &var;  
p2 = array;  
p3 = &array[5];
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

First, we've declared an integer variable and an integer array. Following that are the declarations for three pointers to integer. After declaration, these pointers are still useless to us. We have to assign them values -- addresses to point to.

In the first case, `p1` is assigned the address of `var` (or `&var`). Don't forget the ampersand; without it, we'd be assigning the value of `var`, not its address. In the second assignment, `p2` gets the address of the first element of the array `array[]`. What? No ampersand? Don't tell me you've forgotten already! An array name is an address.

Ah, but what about the third assignment in our example? There's that address operator. No mistake here. Once we add the brackets to the array name, we're referring to the contents of an element of the array, not its address. Just remember: the only time we don't need the address operator is when we're doing our assignment with an array name. The following two lines do exactly the same thing:

```
p1 = array;  
p1 = &array[0];
```

### Putting Them to Work

Okay, now we've got our pointers declared and assigned addresses. Now what? There are several operations we can perform with pointers, including: assignment, getting the address, getting a value, and incrementing or decrementing.

The first, assignment, we've already learned about. The second, getting the address, is nothing new, either. To get the address of a pointer -- the place in memory where the pointer itself is stored -- place the address operator in front of the pointer name:

```
adrp1 = &p1;
```

A more useful operation is getting the value the pointer is pointing to:

```
var = 5;  
p1 = &var;  
z = *p1;
```

In the above example, `z` becomes equal to `var`. How? Our pointer, `p1`, is assigned the address of the variable `var`. The third line is read "z gets the contents of the address pointed to by `p1`." The asterisk is referred to as an "indirection operator," since it allows us to access data indirectly.

Of course, this is a pretty silly example. It would have been more efficient to directly assign the value of `var` to `z` (`z = var`), but there are times when we can't get at variables in the conventional way. Remember, C passes arguments between functions by value, not address.

Take, for instance, our bubble sort program from a couple of chapters ago. What if, instead of using an array, we had three integer variables we wanted to sort, then pass back to the calling function? The following lines show a function call and the first two lines of the function. Will it work?



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
sort(x, y, z);

sort(a, b, c)
int a, b, c;
```

Think about it for a minute. The three arguments passed to the sort function are placed in the three automatic variables a, b and c. No problem there, so we go ahead and sort the three values (code not shown), putting y into x, and z into y, and x into z -- or whatever's necessary to complete the sort. Hurray! We did it.

Wrong.

We forgot one tiny detail. We now have to pass all three values back to the calling function. Any suggestions? The return() statement will allow only one argument. Looks like we're stuck.

What did we do wrong? Why is C being so obstinate? Shall we forget the whole thing and go back to BASIC? Nope.

The solution to our dilemma is (drumroll, please): pointers.

Let's change our function call to this:

```
sort(&x, &y, &z);
```

We're still passing our arguments to automatic variables, but now those variables will contain the addresses of the original three. And, to make things as efficient and elegant as possible, we're going to make those automatic variables pointers. The first two lines of our function will look like this:

```
sort(p1, p2, p3)
int *p1, *p2, *p3;
```

Now we have access to the variables from the calling function. We can switch them around any way we want, using code similar to this:

```
save = *p1;
*p1 = *p2;
*p2 = *p3;
*p3 = save;
```

In English, the above reads: "save gets the contents of the address pointed to by p1; the contents of the address pointed to by p2 gets stored in the location pointed to by p1"; and so on. What we're actually doing is this:

```
save = x;
x = y;
y = z;
z = save;
```

Once we've got the variables the way we want them, we exit the function. We don't have to return any values now; we've done all our work on the variables themselves.

### Incrementing and Decrementing

I stated earlier that pointers were much smarter than conventional variables. One reason is that they're mathematical whizzes. When we perform addition or subtraction on a pointer, the compiler does a lot of the work for us, taking into account the data type it's pointing to and the way that data is stored in memory.

For instance, if we add 1 to an integer pointer, we don't end up with an address one byte higher in memory; we actually move forward two bytes. The compiler knows that integers are two-byte animals, and if we're going to end up with a usable address, the pointer we're incrementing had better end up pointing to the beginning of the next integer.

Now let's have a short quiz. A character array has a beginning address of 73455. A pointer to character, `p1`, has been initialized to the starting address of the array. What address will we get if we increment the pointer? Answer: 73456. Character data requires one byte of storage for each element in the array. Adding 1 to the pointer yields the address of the next element in the array. In this case, the next element is one byte higher in memory.

### The Proof

Now's a good time to dig into Listing 1. Type it in, compile and run it. The output should look something like this:

```
+0 p1=71926 &p1=72910 *p1=65
+0 p1=71926 &p1=72910 *p1=65
+1 p1=71927 &p1=72910 *p1=66
+2 p1=71928 &p1=72910 *p1=67

+0 p2=71930 &p1=72914 *p1=10
+1 p2=71932 &p1=72914 *p1=11
+2 p2=71934 &p1=72914 *p1=12
```

```
Press any key
```

The table this program prints sums up everything we've discussed about pointers. Take a look at the first line of the table. Using what you've just learned, what's the address of the letter A (65) in the character array `array1[]`? If you answered 71926, then you probably have a good basic understanding of how pointers work.

For those of you who are still confused, don't fret. It'll sink in as you get accustomed to using pointers. Let's go through the program and see what's going on.

Lines 9 and 10 declare the arrays and pointers, as well as initializing the arrays. Line 16 puts the address of the first element of `array1[]` into the character pointer `p1`. Line 17 prints out the four values in our table: the amount added to the pointer, the contents of the pointer, the address of the pointer, and the contents of the address the pointer's pointing to. The first line of the table will be printed again when we get into the loop at Line 18. The reason for this is to show you that setting the pointer with the array name is equivalent to setting it with the address operator preceding an array element. In this case, we're comparing `array1` with `&array1[0]`.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Lines 18 through 21 move the pointer through array1[], using the address operator. Each pass through the loop prints one line of our table. Lines 23 through 26 accomplish the same thing, only now we're cycling through an array of integers and incrementing the pointer itself, rather than assigning a new address to it with the address operator.

### A Glimpse of Macros

Notice that, in Listing 1, we've used printf() three times, in almost exactly the same way. In fact, the only difference between them is the name of the pointer we're working with. If the programmer's voice within you is screaming that it's stupid to code the same thing three times, then listen to it. It's right. C provides us with a handy technique to avoid this type of redundant code. The technique involves the use of macros.

Just as was true with pointers, you've already been exposed to macros -- though you were probably unaware of it. Every time we use the #define statement, we're setting up a macro. We've done this dozens of times, but only in the simplest fashion. Macros can be quite complex and are powerful programming aids.

Listing 2 is a modification of Listing 1. Here, each occurrence of the printf() call has been replaced with a macro call. The macro itself is defined in Line 9. Any legal variable name can be used as a macro name.

See the parentheses? This macro contains an argument that will be passed when the macro is expanded (when the substitution string replaces the macro's name in the code). In our example, the argument will be the pointer name to be used in the table.

Of course, just placing the argument in the macro name isn't enough. We've got to tell the macro where we want the argument used in the expansion. In our example, every Z in the replacement string will be replaced by the argument supplied when the macro is called.

Lines 19, 23 and 27 show the macro calls. In Lines 19 and 23, p1 will be substituted into the replacement string. In Line 27, p2 will be substituted.

One interesting note: When I first wrote the program shown in Listing 2, I was using the C compiler that was supplied with the Atari Developer's Kit (Alcyon C). That compiler allows the programmer to place a macro argument within a string, so that the output of Listing 2 could be made identical to the output of Listing 1. Unfortunately, Megamax C doesn't allow macro arguments to be used within a string, so the outputs of Listing 1 and Listing 2 are slightly different.

### Program Listing #1

```
/******  
/*                                C-MANSHIP                                *  
/*                                Chapter 7                                *  
/*          Listing 1 - Developed with Megamax C                        *  
/******/  
#include <stdio.h>  
#include <osbind.h>  
  
char *p1, array1[] = "ABC";  
int *p2, array2[] = {10, 11, 12};
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
main ()
{
    int x, ch;

    p1 = array1;
    printf("+0 p1=%ld &p1=%ld *p1=%d\n\n", p1, &p1, *p1 );
    for ( x=0; x<3; ++x ) {
        p1 = &array1[x];
        printf("+%d p1=%ld &p1=%ld *p1=%d\n",x,p1,&p1,*p1);
    }
    printf( "\n" );
    for ( x=0, p2=array2; x<3; ++x ) {
        printf("+%d p2=%ld &p2=%ld *p2=%d\n",x,p2,&p2,*p2);
        ++p2;
    }
    printf ( "\nPress any key\n" );
    Cconin ();
}
```

### Program Listing #2

```
/*
*****
/*                               C-MANSHIP                               *
/*                               Chapter 7                               *
/*       Listing 2 - Developed with Megamax C                           *
/******
#include <stdio.h>
#include <osbind.h>

#define PRINT(Z) printf("+%d p=%ld &p=%ld *p=%d\n",x,Z,&Z,*Z)

char *p1, array1[] = "ABC";
int *p2, array2[] = {10, 11, 12};

main()
{
    int x = 0, ch;

    p1 = array1;
    PRINT ( p1 );
    printf ( "\n" );
    for ( x=0; x<3; ++x ) {
        p1 = &array1[x];
        PRINT ( p1 );
    }
    printf ( "\n" );
    for ( x=0, p2=array2; x<3; ++x ) {
        PRINT ( p2 );
        ++p2;
    }
    printf ( "\nPress any key\n" );
    Cconin ();
}
```

## CHAPTER 8 - STRUCTURES AND MORE ON POINTERS

Structures offer a way to keep related data items together, allowing easy access to each element. Database applications are a perfect example. Suppose you're the owner of a store and want to keep track of your receivables. You'll need to know, at a minimum, the customer's name, address and amount owed. It would be nice if there were an array type that could store both character strings and floating point numbers. Guess what? Structures to the rescue.

When we set up a structure, we're really defining a new data type, one that's custom designed for our own use. Each "member" of the structure can be any data type we want, even another structure. Let's set up a structure for our store's receivables:

```
struct account {  
    char name[20];  
    char address[36];  
    char city[30];  
    float balance;  
};
```

The keyword `struct`, followed by the name `account`, tells the compiler we're setting up our own data type, and that we're going to call this data type `account`. The structure's members are declared in the same way we'd declare conventional variables, though enclosed with C's ubiquitous braces. The structure declared above contains four members: a 20-element character array called `name`, a 36-element character array called `address`, a 30-element character array called `city`, and a floating-point variable called `balance`.

Now that we've declared our structure, we have a new data type at our disposal, but we still don't have a variable of that type we can use. Think about it for a minute. If we want an integer variable, we must declare it as type `int`. If we want a character variable, we must declare it as type `char`. So it follows that, if we want an account variable (the name we gave our new data type), we must declare it as type `account`:

```
struct account record;
```

We've just told the compiler we want a variable called `record` which is a structure of type `account`. That's all there is to it -- almost.

### Filling It In

We've got our variable `record` set up, but there's still one minor problem: it contains no data. As I'm sure you suspect, initializing a structure is going to be different from initializing the simpler data types. Well, yes...and no.

```
struct account record = {  
    "Clay Walnum",  
    "15 Notreallygonnagivemyaddress Ave.",  
    "Atariland, MA 06116",  
    155.97  
};
```

The main difference between this initialization and that of other data types is that we don't have to include the element's name along with the data. We have to fill in only the information. The compiler

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

knows the first element goes into the field called name, the second into the field called address, etc. We gave it that information when we defined the structure type account.

When initializing a structure, be sure to enclose the data in braces and separate each element with a comma.

### Getting It Out

We now have our structure declared and initialized with data. Just as we need access to each element of an array, we need access to each member of a structure. How can we get at the data? We simply refer to the name of the structure and the name of the element within the structure, separating each with a period:

```
record.name  
record.address  
record.city  
record.balance
```

The first example will give us the string "Clay Walnum." We can manipulate this data the same way we would any string of characters. For example:

```
s = record.name;
```

will point the character pointer s to the string stored in the first member of the structure record.

The second and third examples are similar to the first. The fourth example will give us the floating-point value of 155.97. We might want to use it in this way:

```
printf("Balance = %f\n", record.balance);
```

### Layers Upon Layers

I stated that the elements of a structure could be of any data type, including another structure. Let's take the structure we've created one step further. It might be nice to have the city, state, and zip code in their own elements. We could, of course, just add a couple of members to our original structure. But what if we wanted, for the sake of clarity, to keep all the information within the member city? We'd do it like this:

```
struct where {  
    char c[20];  
    char s[2];  
    char z[5];  
};  
  
struct account {  
    char name[20];  
    char address[36];  
    struct where city;  
    float balance;  
};  
  
struct account record;
```

Now take a deep breath, and we'll attempt to wade through the above example. Our structure account still contains the same information. The difference is that the member city is now a structure of type where, and where contains the members c, s, and z.

Got it? Imagine the structure account as a big box. Inside this box are three other boxes called name, address, city, and balance. Inside the city box are three even smaller boxes called c, s, and z.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Now when we refer to the city member, we need to access the nested members c, s, and z:

```
record.city.c
record.city.s
record.city.z
```

In the first case, we're accessing c, which is a member of city, which is a member of record. In the second, we end up with s, which is a member of city, which is a member of record. I bet you can figure the third one out for yourself.

### More Layers!

I'm not through confusing you, yet. Just as you can have arrays of integers or arrays of characters, you can have arrays of structures. In fact, in the case of the database for our imaginary store, arrays of structures are a necessity. What good is a database with only one entry? We could leave things the way they are and load the records from disk one at a time, but that would be inefficient. Imagine trying to sort a database that way. Not me, buddy. I want them all in memory where I can play with them fast.

Arrays of structures aren't as scary as they sound. One small change to our structure variable declaration, and we've got it:

```
struct account record[100];
```

We now have room for one hundred records of type account.

Accessing each element of our structure array is just as simple:

```
record[index].name
record[index].address
record[index].city.c
record[index].city.s
record[index].city.z
record[index].balance
```

As we vary index from zero to the maximum number of elements in our array, we can access each member as shown above. We also retain control over arrays that make up some of the members of our structure. For instance, if we wanted the third letter in the character array name:

```
record[index].name[2]
```

### An Important Point

In the last chapter, we talked about pointers. Can we use pointers with structures? Sure can. The first step is to declare our pointer, a simple process:

```
struct account *sptr;
```

Now that we have our pointer, we must initialize it:

```
sptr = &record[0];
```

or

```
sptr = record;
```

The above assigns the address of the first byte of our array of structures to the pointer sptr. Suppose this address turned out to be 72000. Using what you've learned about pointers and structures, see if you can calculate the address we'd be pointing to if we added 1 to sptr.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The answer is 72096. How did you do? Remember that a pointer is kept well informed about the data type it's associated with, even if that data type is one we made up, as is a structure. `sptr` knows that there are 96 bytes in each of our array elements. We get this figure by adding together the length of each structure member:

name	30
address	35
c	20
s	2
z	5
balance	4
-----	
	96 bytes

Let's say that `x` is the length, in bytes, of the data type to which we're pointing. Then, when we increment a pointer, we're asking it to point to a location in memory which is `x` bytes ahead of our current location. In the case of our array, we're pointing to the next element, `record[1]`, which begins at an address 96 bytes higher than our current address, or a final address of 72096.

### Pointing to a Member

A pointer to the first member of a structure is only slightly useful. We need to access all of the members. As always, C is there with the answer. Assuming `sptr` equals `&record[0]`, then:

```
(*sptr).name equals record[0].name
(*sptr).city.c equals record[0].city.c
```

A more popular (and less cryptic) way of writing the above would be:

```
sptr->name equals record[0].name
sptr->city.c equals record[0].city.c
```

Either method is fine and gives the same results.

### Functions and Structures

The last thing we need to know in order to take full advantage of structures is how to pass them to functions. As has been evident throughout this chapter, structures are handled the same, for the most part, as any other data type.

The most obvious method of passing information from a structure to a

function is by value:

```
total = add_em( record[index].balance, record[index+1].balance );
float add_em(x, y)
float x, y;
{
    return(x + y);
}
```

Here, two values from our array of structures are passed into the parameters `x` and `y`. The values are added and the result returned to the calling function.



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

But what if we want to modify the contents of the structure directly? As in the past, we resort to pointers:

```
change_em(&record[1]);

change_em(sptr)
struct account *sptr;
{
    sptr->name = "Felix";
}
```

In the above example, we've passed the address of the second member of our array of structures to the function `change_em()`. This address is stored in the pointer `sptr`, where it's used to access the member name.

### The Listing

This chapter's sample program is larger than anything we've done so far. I wanted to offer something moderately usable. There are many techniques in the program we haven't covered. In the next chapter, we'll clear up some of the leftover mysteries. At any rate, the program contains working examples of everything we've discussed here, as well as many other little tidbits you can sort through.

What does it do? I thought you'd never ask. The program is a simple address database. You can enter addresses from the keyboard or disk, then print them to the screen or to the printer in label format. As I said, it's simple. There's plenty of room for enhancements. A sorting feature could be added, or maybe a fancier input routine. To keep data from scrolling off the screen, labels are limited to a maximum of eight. You could add code that would wait for a keypress each time the screen fills, then increase the number of addresses in the database.

### Program Listing #1

```
/* **** */
/*          C-MANSHIP          */
/*          Chapter 8          */
/*          Listing 1          */
/*          Developed with Megamax C          */
/* **** */
#include <stdio.h>
#include <osbind.h>

#define RETURN 13
#define BACKSPACE 8
#define MAX 8
#define PRINTER_OFF 0
#define NOFILE ((FILE *)0)

FILE *fopen();

int work_in[11];
int work_out[57];
int handle;
int contrl[12];
int intin[128];
int ptsin[128];
int intout[128];
int ptsout[128];
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
struct name {
    char fname[30];
    char lname[30];
};
struct rec {
    struct name names;
    char street[30];
    char city[30];
};

/*****
* MAIN PROGRAM
*****/
main ()
{
    int    num_recs, /* Number of addresses in the file. */
           load;     /* File flag. */
    char ch;
    struct rec address[MAX];

    /* Open virtual workstation. */
    open_vwork ();

    /* Get the address data from disk or keyboard. */
    num_recs = get_data ( address, &load );

    /* Convert integer flag to character-type data. */
    ch=load;

    /* Output addresses to screen or printer. */
    output ( address, num_recs );

    /* Save address data to disk if it was entered */
    /* from the keyboard rather than from disk. */
    if ( ch=='N' || ch=='n')
        save_file( address, num_recs );

    /* Wait for a key press. */
    printf ( "Press key\n" );
    Cconin ();

    /* Close the virtual workstation. */
    v_clsvwk(handle);
}

/*****
* open_vwork ()
*
* Initializes a virtual workstation.
*****/
open_vwork ()
{
    int i;
    for (i=0; i<10; work_in[i++] = 1);
    work_in[10] = 2;
    v_opnvwk(work_in, &handle, work_out);
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* get_data ()
*
* Allows the user to choose to enter address records
* either from disk or from the keyboard. The inputs
* are a pointer to the array of structures that will
* hold the data and a pointer to the flag that will
* tell the program whether the data has been loaded
* from disk or typed from the keyboard. The output
* is the total number of records entered.
*****/
get_data ( recp, load )
struct rec *recp;
int *load;
{
    int num_recs;

    /* Print the prompt. */
    Cconws ( "Load file? " );
    /* Loop until we get a proper keystroke. */
    while ( (*load=Cconin())!='Y' && *load!='y'
            && *load!='N' && *load!='n' );

    printf("\n\n");

    /* If the user answered "N" to the prompt, get the addresses
    from the keyboard, or else get the addresses from the disk. */
    if ( *load == 'N' || *load == 'n' )
        num_recs = get_records ( recp );
    else
        num_recs = disk_file ( recp );

    /* Return the number of addresses that were entered. */
    return ( num_recs );
}

/*****
* get_records ()
*
* Retrieves address data from the keyboard. The
* input is a pointer to the structure that will hold
* the address data. The output is the total number
* of records entered.
*****/
get_records ( recp )
struct rec *recp;
{
    int ans, /* Character storage. */
        i; /* Record counter. */

    /* Initialize our variables. */
    ans = 'Y';
    i = -1;

    /* Keep getting addresses until the user indicates */
    /* that he is finished or until we run out of room. */
    while ( (ans=='Y' || ans=='y') && i+1<MAX ) {
        ++i;
        Cconws ( "FIRST NAME: " );
        get_str ( recp->names.fname, 29 );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
Cconws ( "\n LAST NAME: " );
get_str ( recp->names.lname, 29 );
Cconws ( "\n      STREET: " );
get_str ( recp->street, 29 );
Cconws ( "\n      CITY: " );
get_str ( recp->city, 29 );
Cconws ( "\n\nAnother (y/n)? " );
ans = Cconin ();
printf ( "\n\n" );
++recp;
}

/* Return the record count. */
return ( i+1 );
}

/*****
* disk_file ()
*
* Reads address records from a disk file.  The input
* is a pointer to the structure in which to store
* the records.  The output is the number of records
* read.
*****/
disk_file(recp)
struct rec *recp;
{
    FILE *p_file;
    char filename[15];
    int num_recs, x, l;

    p_file = NOFILE;
    /* Get valid filename. */
    while (p_file == NOFILE) {
        Cconws("Filename: ");
        get_str(filename,14);
        printf("\n\n");
        p_file = fopen(filename, "r");
        if (p_file == NOFILE)
            printf("No such file!\n\n");
    }

    /* Read in number of records in file. */
    num_recs = getw(p_file);

    /* Read in all address records. */
    for (x=0; x<num_recs; ++x) {
        fgets(recp->names.fname, 30, p_file);
        l = strlen ( recp->names.fname );
        recp->names.fname[l-1] = 0;
        fgets(recp->names.lname, 30, p_file);
        fgets(recp->street, 30, p_file);
        fgets(recp->city, 30, p_file);
        ++recp;
    }

    /* Return number of records read. */
    return(num_recs);
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* output ()
*
* Asks the user if the address records should be
* sent to the printer or to the screen. The inputs
* are pointer to the structure holding the records
* and the number of records in the structure.
*****/
output(recp, num_recs)
struct rec *recp;
int num_recs;
{
    int status, device;

    /* Initialize loop variable. */
    status = PRINTER_OFF;

    /* Loop until records have been output. */
    while (status==PRINTER_OFF) {

        /* Get device from user. */
        Cconws("Print to screen or printer (s/p)? ");
        device = Cconin();
        printf("\n\n");

        /* Send address records to requested device. */
        if (device == 'p' || device == 'P')
            status = printer(recp, num_recs);
        else {
            screen(recp, num_recs);
            status = -1;
        }
    }
}

/*****
* save_file ()
*
* Writes the address records out to a disk file. The
* inputs are a pointer to the structure holding the
* records and the number of records in the structure.
*****/
save_file(recp, num_recs)
struct rec *recp;
int num_recs;
{
    FILE *p_file;
    char r,x;
    char filename[15];

    /* Ask if user wants to save file. */
    Cconws("Save file? ");
    while ((r=Cconin())!='Y' && r!='y' && r!='N' && r!='n');
    printf("\n\n");

    if (r == 'Y' || r == 'y') {
        p_file = NOFILE;

        /* Loop until we get a valid filename. */
        while (p_file == NOFILE) {

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
Cconws("Filename: ");
get_str(filename, 14);
printf("\n\n");

/* If file doesn't exist. open it. */
if ((p_file=fopen(filename, "r")) == NOFILE)
    p_file = fopen(filename, "w");

/* If file does exist, check if okay to delete. */
else {
    p_file = NOFILE;
    Cconws("File already exists. Delete it? ");
    if ((r=Cconin()) == 'Y' || r == 'y')
        p_file = fopen(filename, "w");
    printf("\n\n");
}

/* Write out the number of address records. */
putw(num_recs, p_file);

/* Write out all the address records. */
for (x=0; x<num_recs; ++x) {
    fprintf(p_file, "%s\n", recp->names.fname);
    fprintf(p_file, "%s\n", recp->names.lname);
    fprintf(p_file, "%s\n", recp->street);
    fprintf(p_file, "%s\n", recp->city);
    ++recp;
}
fclose(p_file);
}

}

/*****
* screen ()
*
* Writes the address records out to the screen. The
* input is a pointer to the structure holding the
* records and the number of records in the structure.
*****/
screen(recp, num_recs)
struct rec *recp;
int num_recs;
{
    int x;

    /* Enter alphanumeric screen mode. */
    v_enter_cur(handle);
    /* Write out each line of each record. */
    for (x=0; x<=num_recs-1; ++x) {
        pos_cur(x,0);
        printf("Record #%d\n", x+1);
        pos_cur(x,1);
        printf("%s %s\n", recp->names.fname, recp->names.lname);
        pos_cur(x,2);
        printf("%s\n", recp->street);
        pos_cur(x,3);
        printf("%s\n\n", recp->city);
        ++recp;
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* printer ()
*
* Writes the address records out to a printer.  recp
* is a pointer to the structure holding the
* records and num_recs is the number of records stored
* in the structure.
*****/
printer(recp, num_recs)
struct rec *recp;
int num_recs;
{
    int x, status, i;
    FILE *p_file;

    /* Wait for printer to be turned on. */
    status = Cprnout(0);
    if (status == PRINTER_OFF) {
        printf("Turn on printer!\n");
        return(status);
    }

    /* Send each line of each record to the printer. */
    for ( x=0; x<num_recs; ++x ) {
        for ( i=0; i<strlen(recp->names.fname); ++i )
            Cprnout(recp->names.fname[i]);
        Cprnout ( ' ' );
        for ( i=0; i<strlen(recp->names.lname); ++i )
            Cprnout(recp->names.lname[i]);
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        for ( i=0; i<strlen(recp->street); ++i )
            Cprnout(recp->street[i]);
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        for ( i=0; i<strlen(recp->city); ++i )
            Cprnout(recp->city[i]);
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        Cprnout ( '\n' );
        Cprnout ( '\r' );
        ++recp;
    }
    return(status);
}

/*****
* pos_cur ()
*
* Positions the cursor on the screen.  i is
* the record number and l is the number of the line
* within the record being printed.
*****/
pos_cur(i,l)
int i,l;

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
{
    int x, y;

    /* If even-numbered record, position on */
    /* right side of screen.                */
    if ((i+1)%2 == 0)
        x = 50;

    /* If odd-numbered record, position on */
    /* left side of screen.                */
    else
        x = 10;
    /* Calculate vertical position of line. */
    y = ((i/2)*5)+4+1;

    /* Position cursor. */
    vs_curaddress(handle,y,x);
}

/*****
 * get_str ()
 *
 * Gets a string from the keyboard. s is a pointer
 * to a character array and mx is the maximum allowable
 * length of the string.
 *****/
int get_str(s, mx)
char s[];
int mx;
{
    int p, code;

    p = 0;

    /* Get character from console. */
    code = Cconin();

    /* Add character to string. */
    while (code != RETURN && p <= mx-1) {
        if (code != BACKSPACE) {
            s[p++] = code;
        }

        /* Handle backspace. */
        else if (p > 0) {
            s[--p] = '\0';
            putchar(BACKSPACE);
            putchar(' ');
            putchar(BACKSPACE);
        }

        /* Get next character. */
        code = Cconin();

        /* Add null to end of string. */
        s[p] = '\0';
    }
    if (p == mx)
        printf("\r\n");
}
```



## CHAPTER 9 - MORE LOOPING STRUCTURES AND FILE I/O

Everyone give a hearty cheer. This chapter we'll be finishing up the most grueling details of programming in C, so that next chapter we can start learning about GEM. It's been a long time coming, but you can't bake a cake until you've heated the oven, right?

Chapter 8's program listing overflowed with new material. Let's tackle that first.

### Unfinished Business

At the top of the listing, beneath the block of defines, you'll see a function, `fopen()`, being declared as returning a pointer to type `FILE`. If you think back, you'll remember that any time a function is going to return something other than an integer, it must be declared. But what the heck is `FILE`, anyway? We've never discussed this data type, have we?

Actually, in a way, we have. In Chapter 8, we talked about structures, data types that are specifically tailored by the programmer. `FILE` is a structure defined in the `stdio.h` file, containing the data elements required to handle file I/O.

Wait a minute. That `fopen()` isn't our function. Except for the function calls, this guy is nowhere to be found in our program listing.

True. `fopen()` is a library function. Now, one would think that, if whoever composed the `stdio.h` file went to all the trouble to set up the `FILE` structure, he would have at least gone to the extra effort to make `fopen()` "ready to go," by declaring it as returning a pointer to `FILE`.

For some strange reason, the version of `stdio.h` that comes with the Atari developer's kit doesn't include the declaration, so we must do it ourselves. If you have the Megamax compiler, however, you can delete this declaration from the program; they did award us the courtesy of finishing the job.

### A Quick Look at GEM

Just beyond the file declaration for `fopen()`, there are declarations for a number of global arrays: `work_in[]`, `work_out[]`, `contrl[]`, `intin[]`, `ptsin[]`, `intout[]`, and `ptsout[]`. If you've looked at some of the C source code for various GEM programs in the public domain, or those published in magazines, you've noticed that these arrays are almost always present. In fact, you've probably seen them used in ST BASIC programs, as well.

All the above arrays have one thing in common: they provide GEM a place to store or retrieve information about the program. This information can then be easily manipulated by the programmer.

I know, I know. I told you we weren't going to be getting into GEM until Chapter 10. But we are going to learn a little about initializing a GEM program, since the cursor control functions I used in Chapter 8's listing are found in the VDI portion of GEM.

What's VDI? GEM is made up of many libraries of functions, each of which is responsible for handling a certain portion of the system's activities. These libraries are grouped into two major units, called AES (Application Environment Services) and VDI (Virtual Device Interface). The libraries making up the AES handle such things as windows, dialog boxes, menu bars and event processing. (An event is

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

some action from the user, such as typing a letter or moving the mouse.) The VDI contains the subroutines to control the ST's graphics, as well as some mouse and cursor control functions.

Since GEM is capable of handling several programs at once (such as using a desk accessory with a word processor), there has to be a way of keeping one job separate from another. GEM tackles this by assigning each program and its associated device (in our case, the screen) a "workstation" which can then be referred to by an identifier known as a "handle." The first thing any GEM application must do is open a workstation.

Which brings us back to the arrays that started this discussion. When we open a workstation, we have to tell GEM how we want the system's attributes initialized. What color should the text be? And should it be shadowed? Or maybe bold? What style fill do we want? Solid? Checkered? All these attributes should be placed in the array `work_in[]` before we open the workstation, since that's where GEM is going to expect to find them.

We're not going to worry, at the moment, about which elements of the array hold information for which attribute. We're just going to take it on faith that `work_in[10]` should be initialized to 2, and the rest will be perfectly happy initialized to 1.

After we've set up the array, we tell GEM to open the workstation with the `v_opnvwk[]` call:

```
v_opnvwk(work_in, &handle, work_out);
```

The parameter `work_in` is the address of our array `work_in[]`, which contains the attribute information we wish to pass to GEM. And `&handle` is the address where GEM should store the handle, the integer value that will allow us to refer to this program's workstation. In our sample program, it's the address of the variable `handle`, which is defined after the `work_in[]` and `work_out[]` arrays at the top of the listing. The parameter `work_out` is, of course, the address of our array `work_out[]`.

When we open the workstation, GEM will load the `work_out[]` array with all the information a programmer needs about the workstation. For instance, `work_out[12]` will contain the number of hatch styles available, while `work_out[13]` will contain the number of colors that can be displayed at one time. We don't have to be concerned with this information now, but it is important that you understand why we need these two arrays.

You can see the mechanics of opening a workstation in the `open_vwork()` function of Chapter 8's program listing. Also, at the end of `main()`, notice the function call:

```
v_clsvwk(handle);
```

This closes the workstation to further output. The argument `handle` is the device handle passed to you by the `v_opnvwk()` call.

### And a Peek at VDI

The remaining five arrays -- `contrl[]`, `intin[]`, `ptsin[]`, `ptsout[]` and `intout[]` -- are directly associated with the VDI. The first three are used to pass information to the VDI routines, while the last two

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

provide a means for the VDI to return information to the program. These arrays are used by GEM for its own purposes; you need do nothing more than declare them at the beginning of your program.

### Moving Along

If you spent the time to examine Chapter 8's program listing, you probably wondered what was going on with this call:

```
Cconws("FIRST NAME: ");
```

This function does nothing more than write a string to the screen. Why didn't I just use `printf()` and avoid all this confusion? It has to do with another discovery I made concerning Megamax C. With Megamax, `printf()` will print only when it encounters a `\n`. This makes handling prompts tricky, if you want the user's input on the same line as the prompt. Resorting to `Cconws()` solved this problem.

Three other new function calls, `getw()`, `fgets()`, and `strlen()`, were used in the sample program in the `disk_file()` function.

```
value = getw(file);
fgets(string, n, file);
l = strlen(s);
```

The function `getw()` reads a word (integer) from a disk file and stores it in `value`. The argument `file` is a pointer to a `FILE` structure. In our program, this function call is retrieving the number of records in our address file.

The function `fgets()` reads a string from a disk file. In the above example, `string` is a pointer to a character array, `n` is the maximum number of characters to read from the file, and `file` is a pointer to a `FILE` structure. When `fgets()` is called, it will continue to read until it has read `n-1` characters, finds a newline character (which is added to the end of the string), or gets an EOF. A null character is tacked onto the string after the read is complete. In our program, we're using `fgets()` to read in the strings that make up each record in our address file.

Finally, the function `strlen()` simply returns the length of string `s`. We're using `strlen()` to find the location of the newline character that was read in by `fgets()`, which we have to replace with a null. Let's say we just read in the name FRED. In our character array `s`, we now have the letters F, R, E, D, followed by a `\n` and a null. The function call

```
l = strlen ( s );
```

will return the length of the string up to the null, which in this case is 5. But our `\n` isn't really in `s[5]`, is it? Remember: arrays start counting at 0. So to replace the `\n` with a null we do this:

```
s[l-1] = 0;
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Another new function, `fprintf()`, can be found in our program within the function `save_file()`:

```
fprintf(p_file, "%d\n", num_recs);  
fprintf(p_file, "%s\n", recp->city);
```

This function is almost identical to `printf()`, the only change being the extra argument, the pointer to the FILE structure.

In the first example, we're printing to the file the integer value stored in `num_recs`, followed by a newline. In the second, we're printing to the file the character string stored in the structure member `city`, also followed by a newline.

### The VDI Cursor Stuff

If you look at the functions `screen()` and `pos_cur()` in the program listing, you'll see the cursor control function calls I mentioned earlier. In order to take advantage of these functions, we must first make this call:

```
v_enter_cur(handle);
```

This function call gets us out of graphics mode and into text mode. In this function, as with all the following, `handle` is the workstation identifier that was returned to us by the `v_opnvwk()` call.

We can position the cursor anywhere on the screen by passing the X,Y-coordinates to the function `vs_curaddress()`:

```
vs_curaddress(handle, y, x);
```

Notice that the coordinates are passed in the opposite order of what you'd expect; that is, Y followed by X. Also, keep in mind that we're now in text mode. The cursor location is based on character positions, not raster coordinates. In medium-resolution text mode, the screen's size is interpreted as 80x24, whereas in graphics mode it's 640x200. Quite a difference!

### Printer Output

Take a look at the function `printer()` in the sample listing. The first thing we have to do is check to see if the printer is on.

```
status = Cprnout(0);
```

The line above accomplishes its task by sending a null character to the printer. If the printer times out, a 0 will be returned by the function. Another way to check the printer is with the function `Cprnos()` which returns a nonzero value if the printer is ready to receive:

```
status = Cprnos();
```

Once we know that the printer is ready to respond, we can start sending text. There are several ways of doing this. The method I chose uses a function called `Cprnout()`, which sends characters to the printer one at a time. The format for this function call is:

```
status = Cprnout(ch);
```

Here, the value returned into `status` will be -1 if the character was sent okay, or 0 if, for some reason, the printer didn't respond. The variable `ch` is the character we want printed.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

In our program, we've omitted status. Since we've already checked the printer status, it's probably not necessary to check it again. However, in a real application program, we must be sure to check the value of status. How would we know if the printer ran out of paper or went off-line unexpectedly?

Note also that we can send a character literal to the printer, as well as the character stored in a variable. In our program, for example, we're printing a space like this:

```
Cprnout ( ' ' );
```

Because we need to print full strings rather than only a single character, we've set up a for loop for each of the strings, using the loop variable as an index into the character array. In this way, we loop through the string, sending it to the printer one character at a time.

Finally, notice that we're ending each string by printing a `\n` and a `\r`. Without a line feed and carriage return, the strings will be printed side by side rather than one above the other.

### Odds and Ends

That covers all the material from Chapter 8's sample program. Now we have a final task to complete before we can move on to GEM: touching on a few details of the C language we haven't yet covered.

What do you make of the following line?

```
z = (x<4) ? x : y;
```

Believe it or not, this is nothing more than a shortcut version of:

```
if (x < 4)
    z = x;
else
    z = y;
```

The `?:` is a conditional operator that requires three operands. The first operand (within the parentheses) is the expression to be tested. If the expression is true, the statement yields the evaluation of the second operand (between the `?` and `:`). If the first expression is false, the statement yields the evaluation of the third operand (between the `:` and `;`). Here's another example that'll get the highest value of two variables:

```
highest = (x > y) ? x : y;
```

C also has a construction similar to BASIC's `ON...GOTO`:

```
switch (exp) {
case 1 :
    printf("exp = 1");
    break;
case 2 :
    printf("exp = 2");
    break;
case 3 :
    printf("exp = 3");
    break;
default :
    printf("exp < 1 or > 3");
}
```

The switch statement works by first evaluating the expression in the parentheses, then checking the following labels to see if there's one that matches the expression's value.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

If there is, program execution jumps to the matching line and continues until it encounters the statement break.

But what if there's no match? What if, in the above example, exp is not 1, 2, or 3? That's where the label default comes in. Program execution will jump to this line if none of the other labels match. Otherwise, if there's no default, program execution will jump to the next line following the end of the switch statement (after the closing brace).

What happens if we leave out the break statements? Remember I said that, once the expression following switch is evaluated, the program jumps to the matching label and continues until it encounters a break. The program doesn't care if there's no break before the next label. It'll go on, past the succeeding labels (ignoring them), and execute every statement it finds -- until it either finds a break or reaches the closing brace. In the example above, if we left out all the break statements and exp evaluated to 2, the output would look like this:

```
exp = 2exp = 3exp < 1 or > 3
```

Similarly, if exp evaluated to 3, we would see:

```
exp = 3exp < 1 or > 3
```

### A New Loop

We've become used to the while and for loop constructions. Both are entry condition loops; that is, the loop conditional is checked before each iteration of the loop. There's another loop construct we've ignored so far, the do while loop.

The do while construct is an exit condition loop. The loop conditional is evaluated after each iteration:

```
x = 0;
do {
    ++x;
    printf("x = %d\n", x);
} while (x < 4);
```

The above prints values of x from 1 to 4. Contrast that with:

```
x = 0;
while (++x < 4) {
    printf("x = %d\n", x);
}
```

which will print values of x from 1 to 3.

### Break, Continue, and Goto

We talked about the break statement earlier, in conjunction with switch, but it can also be used to get out of for, while, and do while loops. When used in a nested loop construction, it only terminates the loop in which it's used. The outermost loops will continue normally.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
while (x < 10) {  
    if (x == 5) break;  
    else printf("x is not 5\n");  
}
```

Another method of affecting loop execution is with `continue`. When `continue` is encountered within a loop, the loop doesn't terminate, but, instead, starts the next iteration.

```
x = 0;  
while ((ch = getchar()) != '*') {  
    if (ch == ' ') continue;  
    s[x++] = ch;  
}
```

Finally -- although I hate to mention it, due to its inevitable abuse -- C has a `goto` statement. The keyword `goto` is followed by the label identifying where program execution should continue:

```
goto print_name;  
  
print_name: printf("Name: %s", name);
```

Frankly, there's little or no use for the `goto` statement in a structured language like C. The same goes, though not as strongly, for `break` and `continue`, except when the former is used within a `switch` statement. There's almost always a more structured and elegant way to get around the use of these statements. If you're a BASIC programmer, it will take you a while to get accustomed to structuring your programs in such a way as to avoid the use of a `goto`. But, trust me, it can be done, and the results are much more readable than BASIC's typical tangle.

## CHAPTER 10 - THE FIRST LOOK AT GEM AND THE VDI

Hurray! The long wait is over. This chapter, as promised, we're going to start digging into GEM and learn how to get the most out of our STs. You've worked hard getting familiar with C, so give yourself a quick pat on the back for a job well done. Now put those thinking caps back in place. All set?

### A Review of GEM

In Chapter 9, we took a brief look at what GEM really is. We stated that GEM (Graphics Environment Manager) is made up of many libraries of functions, each of which handles certain portions of the system's activities. These libraries are grouped into two major units, called the AES (Applications Environment Services) and the VDI (Virtual Device Interface). The AES contains the functions we need to handle windows, dialog boxes, menu bars and event processing. The VDI controls most of the ST's graphic capabilities, as well as providing some mouse and cursor control functions.

What's so hot about GEM, anyway? Why all the hoo-hah? You've been using your computer for quite a while now, and you know one great advantage of GEM already: its ease of use. The system is designed in a logical, almost real-world sort of way, supplying icons that represent activities we're used to in everyday life, like file drawers and trash cans. That's why GEM's main screen is called a desktop. We can access calculators, documents, writing utensils, clocks, calendars, appointment books, and any of a hundred other items you might find on your desk.

But another advantage of programming in GEM is its portability. It's been said that GEM is the most portable operating system in existence. This means your programs can easily be ported to other machines using the GEM environment, so your programming efforts are even more valuable.

### Presenting the VDI

The VDI plays an important role in making your graphics programs operate on many different devices. Unfortunately, one of the crucial elements in the graphics interface, GDOS (Graphics Device Operating System) is not built into the current operating system. GDOS is the portion of the VDI which links the graphics functions to the drivers needed to assure that the graphics operate properly on all graphics devices. GDOS also makes it possible to load different fonts into your ST, using the standard VDI functions.

At this time, however, we're concerned only with one device: the screen.

### The VDI functions

The VDI provides the programmer with a series of functions that let him quickly draw many graphic shapes. This makes development of programs that rely heavily on graphics a breeze. If you programmed an 8-bit Atari (or still do), think of all the work involved in drawing a circle. The VDI provides a function that will draw any size circle we want -- with a single call. There are also functions for drawing ellipses, lines, rectangles, rounded rectangles, arcs, pie slices and a number of other useful graphics.

And it doesn't stop there. Each graphic function has a group of related attributes that may be set before the graphic is drawn, allowing various types of lines, fill patterns and colors.



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

This chapter's sample program shows how to call most of the VDI's graphics functions. It was developed using the Megamax C compiler, but is also compatible with Alcyon C, the compiler that comes with the Atari Developers Kit.

When the program is run, the first screen will show the different types of line styles available to you through the VDI. Each time you press a key, the program will display another set of graphics generated by a VDI function.

### The Sample Program

Let's take a look at the listing and see what's going on. The first thing we must do when writing a GEM program is initialize the application. We do this with the call:

```
appl_int();
```

This tells the AES about our application and sets aside the resources the AES needs to function. Next, we must make the call:

```
handle=graf_handle(&gr_hwchar, &gr_hhchar, &gr_hwbox, &gr_hhbox);
```

This returns the handle for the currently open device or workstation, as well as the size of the system font. Because GEM is capable of having many programs in memory at once, each requires some identification, to keep commands for one program from messing up another. This is accomplished by assigning each program a handle. The variable handle in the above call is an integer value that identifies the current workstation.

The graf\_handle() call also returns some information about the system font. We must declare four variables of type integer to hold this information, then pass their addresses to the function. In the above call, gr\_hwchar will get the width of a character cell in pixels; gr\_hhchar will get the height of a character cell in pixels; gr\_hwbox will get the width, in pixels, of a box large enough to hold a single character; and gr\_hhbox will get the height, in pixels, of a box large enough to hold a single character. We won't be using any of this information now, but you should be aware of why we supply these variables.

### Let's Get Virtual

The graf\_handle() call returns the handle to the physical workstation. What we really need for our program is a handle to a virtual workstation. It's kind of tough to explain the difference, but I'll give it a shot.

A particular device may have many virtual workstations, but only one physical workstation. The physical workstation is directly associated with the device itself, usually the screen. You can think of a virtual workstation as a "pretend" device. It has its own section of memory, and keeps its data and status completely separate from all other virtual workstations. When you activate an application (such as clicking on a desk accessory), it is bound to the physical workstation. In a sense, it becomes the physical workstation.

We get the handle for our virtual workstation with the call:

```
v_opnvwk(work_in, &handle, work_out);
```

This function expects the system attributes to be in the work\_in[] array. If you're not sure why we need the arrays work\_in[] and work\_out[], review Chapter 9.

## Polylines

Now that we've got our workstation set up, we can get down to business. The first graphic we'll experiment with is called polylines. Those of you who are up on your linguistics know that the prefix poly means "many." Polylines are one or more lines connected from point to point, which allow the programmer to draw complex shapes with a single function call. The function call looks like this:

```
v_pline(handle,num_pairs,pxy);
```

The variable handle is, of course, the handle returned from the v\_opnvwk() call. Every function we use requires this handle. That way, we're sure we won't mess with another application which may be in memory at the same time. If we're writing a desk accessory to be used with a word processor, for example, we want to be positive we don't change anything in the word processor application. Otherwise, we're liable to have an irritated user, to say the least.

The parameter pxy is a pointer to an array of integers which holds each of our polyline's end points in X,Y pairs. For instance, if we wanted to draw a box, pxy[] might look like this:

```
int pxy[]={24,18,176,18,176,118,24,118,24,18}
```

The integer parameter num\_pairs is the number of coordinate pairs in the pxy array. By the way, the pxy values are pixel values; in other words, in a low resolution screen we'd have possible values of 0-319 for the X-coordinates and 0-199 for the Y-coordinates.

As I mentioned previously, there are a number of attributes we can set for each of the VDI graphics functions. For polylines, we can set the color, type and width, and the end style. We set the color with:

```
vsl_color(handle,color);
```

Here, color is an integer from 0 to the device maximum (low resolution=15, medium resolution=3, and high resolution=1). If we use a number higher than the maximum, the function will default to color 1. On the ST, the default color palette, starting with 0 and ending with 15, is white, black, red, green, blue, cyan, yellow, magenta, white, black, light red, light green, light blue, light cyan, light yellow and light magenta. The function will return the color value chosen.

If we're drawing a line at the smallest width, we can choose between six system line types with:

```
vsl_type(handle,type);
```

Here, type is an integer value from 1 to 7 as follows:

1	solid	_____
2	long dash	_____
3	dots	_____
4	dash dot	_____
5	dash	_____
6	dash dot dot	_____
7	user defined	_____

Type 7 lets you set up your own line types, but we're not going to get into that now.

When you're drawing lines, you can also choose an end style with the call:

```
vsl_ends(handle,end1,end2);
```

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

In this case, end1 and end2 are integer values from 0 to 2. A value of 0 will yield a square end, 1 will get you an arrow, and 2 will result in a rounded end. The variable end1 is the beginning style, and end2 is the ending style.

Finally, we can set the thickness of our lines with the call:

```
vsl_width(handle,width);
```

The variable width must be an odd positive integer. The line will be set to the closest width less than or equal to the value of width. The value chosen is returned from the function.

### Rounded Rectangles

We can employ `v_pline()` to draw a standard square-cornered box, but the VDI also supplies a function which will let us draw rectangles with rounded corners. The function is called in this manner:

```
v_rbox(handle,pxy);
```

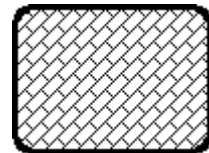


Once again, we tell GEM where to draw our rectangle with the pxy array, except this time we must supply only the pixel coordinates of the lower-left and upper-right corners. The line attributes -- color, style and width -- are used with `v_rbox()`, allowing a wide variety of rectangles.

### Filled Rounded Rectangles

If you want a solid, rounded rectangle, you can make this function call:

```
v_rfbox(handle,pxy);
```



The pxy array is used the same way as in `v_rbox()`, supplying the function with the lower-left and upper-right corners. The body of the rectangle is filled with the active fill pattern, which we'll see how to set later on. The default is a solid fill.

### Circles

Want to draw a circle? No sweat! Just use this function call:

```
v_circle(handle,x,y,radius);
```



The integer parameters x and y are the pixel coordinates of the circle's center, and radius is, obviously, the circle's radius (also an integer). The `v_circle()` function, like `v_rfbox()`, uses the current fill attributes.

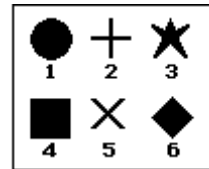
## Polymarkers

Polymarkers are a number of predefined shapes you can use in your graphics. You call the function this way:

```
v_pmarker(handle,number,pxy);
```

The integer parameter number is the number of markers you wish to draw. Coordinates for each marker are stored in the pxy array, one X,Y pair for each marker.

But what do these markers look like? You have a choice of six predefined shapes which (from 1 to 6, respectively) are dot, plus sign, asterisk, square, diagonal cross and diamond.



To set the polymarker type, call:

```
vsm_type(handle,type);
```

Here, type is an integer from 1 to 6. If you should choose a value out of this range, the function will select the asterisk as a default. The value chosen will be returned from the function.

There are two other attributes which affect polymarkers: color and height. Color is set with the call:

```
vsm_color(handle,color);
```

Here, color is an integer from 0 to the device maximum. All the rules of the vsl\_color() call apply in this case.

You can change the size of all polymarkers, except the dot (which always appears in the smallest size), with the call:

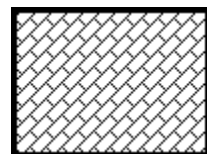
```
vsm_height(handle,height);
```

Here, the integer parameter height is the polymarker's size on the Y-axis. Actual height will be the greatest height available on the device, less than or equal to the height parameter.

## Filled Rectangles

Solid rectangles can be drawn with the call:

```
v_bar(handle,pxy);
```

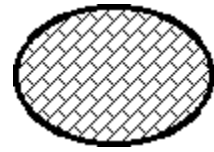


As usual, the lower-left and upper-right corners are stored in the pxy array. The active fill attributes are used to color the body of the rectangle.

### Ellipses

An ellipse looks something like a squashed circle or a solid oval. You can draw it with the call:

```
v_ellipse(handle,x,y,xrad,yrad);
```



Here, the integers x and y denote the ellipse's center point, and the integers xrad and yrad are the X- and Y-radii in pixels. Once again, the active fill attributes are used.

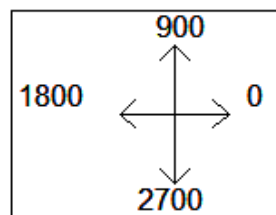
### Arcs

Arcs are simple to draw, with this call:

```
v_arc(handle,x,y,radius,bang,eang);
```



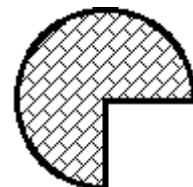
The integers x, y, and radius are the X,Y-coordinates of the center and the radius, respectively. The integers bang and eang are the beginning and ending angles of the arc, in tenths of a degree. The following diagram illustrates the possible angle values:



### Pie Slices

Here's a handy function that'll help you draw those fancy pie charts. To draw a pie slice, use the call:

```
v_pieslice(handle,x,y,radius,bang,eang);
```



The parameters are the same as those for the arc function. The body of the pie slice will be colored by whatever fill pattern is active.

## Fill Patterns

GEM supplies us with many patterns we can use to fill our figures. There's a series of functions to let us set these patterns up the way we want them. The first step is the function call:

```
vsf_interior(handle, style);
```

Here, style is an integer 0 to 4. The values are interpreted as follows:

0	Hollow (background color)
1	Solid
2	Pattern
3	Hatch
4	User-defined

If you choose style 0 or 1, you need go no further, but style 2 allows you to choose between 24 different patterns, and style 3 provides 12 hatch styles. You choose the pattern you wish to use, with the call:

```
vsf_style(handle, style);
```

Here, style is an integer value from 0 to 23. Consult your reference manual to see what these styles look like (or run the sample program).

The color of your fill is selected with the call:

```
vsf_color(handle, color);
```

All the rules for the `vsl_color()` function apply here, also.

Finally, you can choose between a visible or invisible border for your fill, with the call:

```
vsf_perimeter(handle, vis);
```

Here, vis is any integer. A value of 0 will give you an invisible border; any other value will cause the border to be drawn in the current fill color.

## Use Those Tools!

Now that you've been introduced to many of the graphics functions available to you through the VDI, study the sample program to see them in action, then take some time and experiment with the VDI on your own. See if you can write a program to draw a simple picture, maybe a graph or two.

## Program Listing #1

```

/*****
/*      C-MANSHIP, LISTING 1      */
/*      CHAPTER 10                */
/*      DEVELOPED WITH MEGAMAX-C  */
*****/
#include <osbind.h>

int work_in[11], work_out[57];
int handle;
int contrl[12], intin[128];
int ptsin[128], intout[128], ptsout[128];
int gr_hwchar, gr_hhchar, gr_hwbox, gr_hhbox;

main()
{
    appl_init();
    open_vwork();
    do_pline();
    do_roundrec();
    do_froundrec();
    do_circle();
    do_pmarker();
    do_bar();
    do_ellipse();
    do_arc();
    do_pieslice();
    do_fills();
    v_clsvwk(handle);
    appl_exit();
}

open_vwork()
{
    int i;

    for (i=0; i<10; work_in[i++] = 1);
    work_in[2] = 2;
    handle = graf_handle(&gr_hwchar,&gr_hhchar,&gr_hwbox,&gr_hhbox);
    v_opnvwk(work_in, &handle, work_out);
}

do_pline()
{
    int pxy[4];
    int color, end, type, width;

    pxy[0] = 30;   pxy[1] = 20;
    pxy[2] = 280;  pxy[3] = 20;
    end = 0; width = 1;
    v_clrwk(handle);
    for (color=1; color<5; ++color) {
        vsl_color(handle,color);
        vsl_ends(handle,end,end);
        vsl_width(handle,width);
        v_pline(handle,2,pxy);
        pxy[1] += 10;  pxy[3] += 10;
        end += 1; width += 2;
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
    }
    vsl_width(handle,1);
    vsl_ends(handle,0,0);
    vsl_color(handle,1);
    for (type=1; type<7; ++type) {
        vsl_type(handle,type);
        pxy[1] += 10; pxy[3] += 10;
        v_pline(handle,2,pxy);
    }
    Cconin();
}

do_roundrec()
{
    int pxy[4];
    int color, width;

    pxy[0] = 10;    pxy[1] = 10;
    pxy[2] = 300;   pxy[3] = 190;
    width = 1;
    v_clrwk(handle);
    vsl_type(handle,1);
    for (color=1; color<7; ++color) {
        vsl_width(handle,width);
        vsl_color(handle,color);
        v_rbox(handle,pxy);
        width += 2;
        pxy[0] += 20;  pxy[1] += 20;
        pxy[2] -= 10;  pxy[3] -= 10;
    }
    Cconin();
}

do_froundrec()
{
    int pxy[4];
    int color;

    pxy[0] = 10;    pxy[1] = 10;
    pxy[2] = 300;   pxy[3] = 190;
    v_clrwk(handle);
    for (color=1; color<7; ++color) {
        vsf_color(handle,color);
        v_rfbox(handle,pxy);
        pxy[0] += 20;  pxy[1] += 20;
        pxy[2] -= 10;  pxy[3] -= 10;
    }
    Cconin();
}

do_circle()
{
    int color, radius;

    v_clrwk(handle);
    radius = 100;
    for (color=1; color<8; ++color) {
        vsf_color(handle,color);
        v_circle(handle,150,100,radius);
        radius -= 15;
    }
}
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
Cconin();
}

do_pmarker()
{
    int color, height, type;
    int pxy[2];

    v_clrwk(handle);
    pxy[1] = 10;
    for (type=1; type<7; ++type) {
        vsm_type(handle,type);
        height = 2; pxy[0] = 10;
        for (color=1; color<6; ++color) {
            vsm_color(handle,color);
            vsm_height(handle,height);
            v_pmarker(handle,1,pxy);
            height += 16; pxy[0] += 60;
        }
        pxy[1] += 35;
    }
    Cconin();
}

do_bar()
{
    int pxy[4], color;

    pxy[0] = 10; pxy[1] = 190;
    pxy[2] = 300; pxy[3] = 10;
    v_clrwk(handle);
    for (color=1; color<6; ++color) {
        vsf_color(handle,color);
        v_bar(handle,pxy);
        pxy[0] += 25; pxy[1] -= 20;
        pxy[2] -= 20; pxy[3] += 10;
    }
    Cconin();
}

do_ellipse()
{
    int color, xradius, yradius;

    v_clrwk(handle);
    xradius = 150; yradius = 100;
    for (color=1; color<11; ++color) {
        vsf_color(handle,color);
        v_ellipse(handle,150,100,xradius,yradius);
        xradius -= 15;
    }
    Cconin();
}

do_arc()
{
    int color, radius, bang, eang;

    v_clrwk(handle);
    vsl_width(handle,3);
    bang = 900; eang = 0; radius = 10;
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
for (color=1; color<6; ++color) {
    vsl_color(handle,color);
    v_arc(handle,150,100,radius,bang,eang);
    bang += 60; eang -= 60; radius += 20;
}
Cconin();
}

do_pieslice()
{
    int color, radius, bang, eang;

    v_clrwk(handle);
    radius = 100; bang = 3200; eang = 600;
    for (color=1; color<6; ++color) {
        vsf_color(handle,color);
        v_pieslice(handle,150,100,radius,bang,eang);
        radius -= 15; bang -=200; eang -= 100;
    }
    Cconin();
}

do_fills()
{
    int pxy[4], style, i, num, x;

    pxy[0] = 50; pxy[1] = 30;
    pxy[2] = 250; pxy[3] = 170; num = 25;
    for (i=2; i<4; ++i) {
        vsf_color(handle,i);
        vsf_interior(handle,i);
        for (style=1; style<num; ++style) {
            vsf_style(handle,style);
            v_clrwk(handle);
            v_bar(handle,pxy);
            for (x=0; x<32000; ++x);
        }
        num = 13;
    }
    Cconin();
}
```

## CHAPTER 11 - VDI TEXT FUNCTIONS

Those of you who programmed the 8-bit Ataris were limited in your text displays. Sure, you had graphics 1 and 2, which endowed your computer with oversized text in four colors, and you could, when in graphics 0, inject life with some inverse video.

If those alternatives did nothing to satisfy your critical eye, you could always take refuge in a redesigned character set. And, if you were into self-brutalization -- or were desperate to the point where opened wrists seemed preferable to another moment of programming -- you could draw your characters pixel by pixel, line by line, until your masterwork emerged amidst the ruins of your mental health.

But those are bygone times. Now you own an ST. Because the ST's screen is bit-mapped rather than character-mapped, you may fire your shrink and discard all schemes of self-destruction. Text, like any other graphic, is drawn on the screen.

Stop right there! Wasn't it the drawing of text on the 8-bits -- that ghastly alternative to the normal displays -- that forced many talented bit-and-byte managers to take up residence in the local Institute for the Incredibly Nervous? Yes, indeed. But, on the ST, GEM's VDI takes on the task, supplying the programmer with simple functions to graphically manipulate text. There are about two dozen text sizes available, as well as numerous special effects, which can be combined in any way the programmer sees fit.

To get a quick introduction to the VDI text functions, type in this chapter's program listing, compile and run it. Use the mouse to click on the menu options. Clicking the left button when viewing a demo screen returns you to the menu; clicking the right button when at the menu returns you to the GEM desktop.

### Who's a Dummy?

Now that you've seen some of the things you can do with text on an ST (I suspect you've seen this stuff before), let's dig into the listing. The program first calls `appl_init()`, after which it opens a virtual workstation. We discussed these procedures in Chapter 10, but take a look at the parameters for the `graf_handle()` call. See something a little strange? Four of the parameters are the address of the variable `dummy`.

In Chapter 10, I told you that `graf_handle()` returns information about the system font. This information is stored in four variables whose addresses you pass with the call. In this chapter's demo program, we've no need for this information, so why clutter up the program with extra variables? The `graf_handle()` call doesn't care where it stores the information, as long as you give it an address. In fact, it doesn't even care if you give it the same address for all four values. It'll happily store one value on top of the previous one (wiping the older value out, of course; you'll have no way to retrieve any but the last).

The integer variable `dummy` is used throughout the program in just this way. Anytime we must supply storage for a dispensable value, we'll use the `dummy` variable.

## Converting Between Resolutions

After we've got our workstation opened, function `init()` sets up the program for our current resolution, then changes the mouse pointer to the hand icon. In order to do this, we first need to get the resolution. We do this with the call:

```
res = Getrez();
```

This returns an integer from 0 to 2. A value of 0 means the screen is currently in low resolution; a value of 1 indicates medium resolution; and a value of 2 tells you you're in high resolution. This function is defined in the `osbind.h` file, and is a part of the XBIOS.

In low resolution, the screen dimensions are 320x200. In medium, the horizontal resolution is doubled, giving us a screen 640x200. Finally, in high resolution, both the horizontal and vertical resolutions are doubled (as compared to low resolution), yielding a screen 640x400. These relationships are important if we're going to write software compatible with all three resolutions.

Let's say we're in low resolution. We draw a rectangle with the coordinates 20 20, 60 20, 60 40, 20 40 and 20 20 (these are the coordinate pairs you would load into the `pxy` array before calling `v_pline()`). Now we switch to medium resolution and draw the same rectangle.

What happened? The rectangle is only half as long, right? This is because the horizontal resolution has been increased by a factor of 2; the screen pixels are half as wide, so they produce a rectangle half as long. If we want the rectangle the same size in medium resolution as in low (and in the same place on the screen), we have to double the value of the horizontal coordinates. A rectangle drawn in medium resolution between the coordinates 40 20, 120 20, 120 40, 40 40 and 40 20 will look like one drawn with the previous coordinates in low resolution.

Now let's use the medium resolution coordinates to draw the same rectangle in high resolution. Whoops! The figure is the same length, but now it's only half as high. No surprise, right?

The vertical dimension of a high resolution screen is twice that of low or medium resolution screens. If we want to draw that same rectangle yet again, but in high resolution, we must multiply the vertical coordinates by a factor of 2, giving us 40 40, 120 40, 120 80, 40 80 and 40 40.

Text output is affected by changes in resolution, too. In medium resolution, text is half as wide as in low. High resolution, which uses a different font, yields text the same width as that in medium resolution, but half as high.

How's all this handled in `init()`? Well, let's see. Once we get the resolution with a call to `Getrez()`, we use the returned value in a switch statement to set `h_factor` (horizontal factor), `v_factor` (vertical factor) and `t_factor` (text factor) to their appropriate values. We'll use these values in calculating screen coordinates for the resolution we're in.

Some of the shapes to be drawn by our program have coordinates hard coded into arrays. This saves us from setting up a `pxy` array each time we draw one of these shapes; we can, instead, pass the address of the array that contains the coordinates.

To avoid calculations later on in the program, we immediately modify these arrays for our current resolution. The for loop near the bottom of `init()` accomplishes this, by multiplying each element of

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

the array by one of the factors initialized by the switch statement. The figures whose coordinates are stored in these arrays will then be displayed properly in any resolution.









### Of Mice and C

The function `init()`'s last task is to change the mouse form from the arrow to the hand. The call that accomplishes this is:

```
graf_mouse (form,mouse_form);
```

Here, `form` is an integer value from the table below and `mouse_form` is the address of a 35-element array containing the data for the mouse form. At this point, we're not going to discuss this array, since it pertains to user-defined mouse forms rather than those supplied by the system. We'll discuss custom mouse forms in an upcoming chapter.

The acceptable values for `form` are as follows:

0		Arrow
1		Line cursor
2		Bee
3		Pointing hand
4		Flat hand
5		Thin crosshair
6		Thick crosshair
7		Outlined crosshair
255		User-defined mouse form
256		Hide mouse form
257		Show mouse form

Any value from 0 to 7 will yield the mouse form shown. A value of 255 directs the function toward a user-defined mouse form stored in the `mouse_form` array. A value of 256 removes the mouse form from the screen, and a value of 257 restores it. As we'll see later, the ability to hide the mouse form is critical when drawing on the screen.

The `graf_mouse()` function is a part of GEM's AES libraries.

### Menus and Varmints with Buttons

The main program loop, found in `do_menu()`, utilizes the mouse for menu selection. The outer while loop repeats the menu process until the user wishes to exit the program, while the inner while loop samples the mouse until one of the buttons is pressed.

Also within the inner loop is a call to `mouse_print()`. This function (found at the end of the listing) prints the coordinates of the mouse in the upper-left corner of the screen (actually, it'll print any two integers). I use this function to help me find the mouse X,Y-positions I need for my test statements. For instance, when writing this chapter's sample program, I used `mouse_print()` to determine what

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

coordinates fell within each of the menu selections. Once the program was completed, I thought that, rather than delete `mouse_print()` from the listing, I'd leave it for you to fool with. What a guy, huh?

Also, there are a couple of interesting function calls in `mouse_print()`. One of them, `v_gtext()`, we'll be using extensively, since it's the VDI function that displays text. The syntax for this call is:

```
v_gtext (handle,x,y,string);
```

The integers `x` and `y` are the location the text is to be printed, and `string` is a pointer to the text. (You may use a string literal within the call by enclosing it in quotes.) Remember that an array name (a string is an array of character) is a pointer. Since `v_gtext()` will handle only strings, how do we output other forms of data to the screen? What if we're writing a game and need to display a score? No problem. All we have to do is convert the data we want to print into a string. The following example will prepare an integer for printing with `v_gtext()`:

```
sprintf (s,"%d",i);
```

The parameter `s` is the address of the string where the function is to store the converted data. (Don't forget to leave space for the null!) The rest of the parameters are the same as for `printf()`. If you're a little fuzzy on `printf()`, review Chapter 1.

Getting back to `do_menu()`, once a button press is detected, a series of if/else statements check which button was pushed and the location of the mouse at the time. The VDI function that returns the mouse status is:

```
vq_mouse (handle,&button,&mx,&my);
```

The parameter `handle` is, of course, the handle that was returned by the `v_opnvwk()` call. The parameters `&button`, `&mx`, and `&my` are the addresses of integer variables that will hold the button pushed, the mouse's X-position, and the mouse's Y-position, respectively. The value returned in `button` will be 0 if no button is pressed, 1 if the left button is pressed, 2 if the right button is pressed, and 3 if both buttons are pressed.

After we exit the inner while loop, we check for a button value of 1 (left button pressed). If the left button was pressed, we then check the mouse coordinates at the time the button was pressed, to see if the pointer was within one of our menu selections. If it wasn't, `repeat` retains its true condition, and the outer while loop is repeated.

If the mouse pointer was within the menu, we perform the appropriate function, redraw the menu, then return to the main while loop (`repeat` is still true). If `button` equals 2 (right button pressed), we set `repeat` to 0, which breaks us out of the main loop and returns us to `main()`, where we close the virtual workstation and return to the desktop.

Notice that, when checking for mouse coordinates, we're utilizing `h_factor` and `v_factor`. Just as when drawing a shape, the horizontal and vertical mouse coordinates are dependent on the current resolution. We must multiply each coordinate in the if statements by the appropriate factor.

# C-MANSHIP COMPLETE – by CLAYTON WALNUT

## Text Effects

The ST has several built-in text effects you can use to enhance your programs. Text can be printed bold, light intensity, skewed, underlined, outlined, or any combination of the above. The function `do_effects()` in the sample program demonstrates these effects.

First, a call to `v_hide_c()` hides the mouse form, then `v_clrwk()` clears the screen. The text color is set with the call:

```
vst_color (handle,color);
```

Here, `color` is an integer from 0 up to the maximum colors available for the current resolution. (You know what `handle` is, right?) Next, we set the text height (we'll cover this function a little later) and enter the loop that prints the text. The different effects are set with the call:

```
vst_effects (handle,effect);
```

Here, the bits of the integer effect are set as below:

<u>Bit</u>	<u>Value</u>	<u>Effect</u>
0	1	<b>Bold</b>
1	2	Light
2	4	<i>Skewed</i>
3	8	<u>Underlined</u>
4	16	<b>Outlined</b>

Note that the value in the bit column is the number of the bit to set, not the value to send to the function. You need to do some binary arithmetic to arrive at the decimal values shown in the second column. Any combination of effects can be used by adding the values together. For instance, if you want just bold text, the parameter `effect` in the above call should be set to 1; if you want underlined and bold text, `effect` should be set to 9 (1+8); for skewed, outlined, bold text, `effect` needs the value 21, and so on.

## Text Height

As I mentioned earlier, the ST is capable of displaying text in many different heights. Best of all, you may mix these heights on the screen in any way you wish. To set the height of text to be printed, use the call:

```
vst_height (handle,height,&char_w,&char_h,&cell_w,&cell_h);
```

The integer `height` is the requested height, and the parameters `&char_w`, `&char_h`, `&cell_w`, and `&cell_h` are pointers to integer. Respectively, the values returned in these addresses are the character width, the character height (from the base line to the top of the cell), the cell width, and the cell height. In the sample listing, since we don't need this information, we just return all these values to our old standby, `dummy`.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Another function we can use to set text height is:

```
vst_point(handle,point,&char_w,&char_h,&cell_w,&cell_h);
```

Here, point is the height of text in points (a point equals 1/72 inch). The other parameters are the same as for vst\_height().

### Text Rotation

The GEM operating system allows text to be printed at any angle. Unfortunately, the ST implementation of GEM allows rotation in 90-degree increments only. To set the base line rotation of the text, use the call:

```
vst_rotation (handle,angle);
```

The integer angle is the angle of rotation in tenths of degrees. Because of the limitation placed on this function for the ST, this value must be 0, 900, 1800 or 2700.

In the sample listing, the function do\_rotate() demonstrates the use of text rotation. Handy for graphs!

### Mouse Prestidigitation

In all cases, before we draw something on the screen, we must hide the mouse form. If we don't, we may find a block of the old screen pasted in over the new one as soon as the mouse is moved. This may seem peculiar at first, but the logic behind it is simple. In order to allow mouse movement, the operating system must save for later redraw the section of the screen covered by the mouse cursor. When the mouse is again moved, the screen is restored by "pasting" back the saved block. The saved screen block remains unchanged if we draw to the screen, so when the mouse is moved, and GEM pastes in the old block, we may find a portion of the old screen coming back to haunt us.

The VDI provides the following functions for turning the mouse form on and off:

```
v_hide_c (handle);  
v_show_c (handle);
```

There's something to keep in mind when using these functions. Every call to v\_hide\_c() must have a corresponding call to v\_show\_c() -- unless, of course, you don't plan to see your mouse again. This doesn't mean you can't call v\_hide\_c() twice in a row; it just means that if you do call it twice in a row, you must also call v\_show\_c() twice to get your mouse back.

### Break Time

Now that you've learned a good deal about the VDI and how to use a mouse, you have the tools to begin some serious GEM programming. The best way to become confident with these tools is to use them. So, practice what you've learned.



# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```

/*****
/*      C-MANSHIP, LISTING 1      */
/*      CHAPTER 11                */
/*      DEVELOPED WITH MEGAMAX-C  */
*****/
#include <osbind.h>

#define BLACK 1
#define RED 2
#define GREEN 3
#define HOLLOW 0
#define SOLID 1
#define HAND 3
#define NORMAL 0

int work_in[11], work_out[57];
int contrl[12], intin[128];
int ptsin[128], intout[128], ptsout[128];
int mouse_form[35];
int rec1[] = {106,150,206,50};
int rec2[] = {108,148,204,52};
int line1[] = {108,84,204,84};
int line2[] = {108,116,204,116};

int res, h_factor, v_factor, t_factor;
int handle, dummy;

main()
{
    appl_init();
    open_vwork();
    init();
    do_menu();
    v_clswnk(handle);
    appl_exit();
}

do_menu()
{
    int repeat, button, mx, my;

    repeat = 1;
    draw_menu();
    while (repeat) {
        button = 0;
        while (button == 0) {
            vq_mouse(handle, &button, &mx, &my);
            mouse_print (mx, my);
        }
        if (button == 1) {
            if (mx>112*h_factor && mx<199*h_factor) {
                if (my>54*v_factor && my<81*v_factor) {
                    do_effects();
                    draw_menu();
                }
                else if (my>86*v_factor && my<113*v_factor) {
                    do_height();
                    draw_menu();
                }
            }
        }
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        else if (my>118*v_factor && my<145*v_factor) {
            do_rotate();
            draw_menu();
        }
    }
    else if (button == 2)
        repeat = 0;
}
}
```

```
do_effects()
{
    int x, y, effect, b_effect, n_effect, height;

    v_hide_c (handle);
    v_clrwk (handle);
    vst_color (handle,BLACK);
    if (res == 0)
        height = 4;
    else
        height = 8;
    vst_height (handle,height,&dummy,&dummy,&dummy,&dummy);
    b_effect = 1;
    for (x=5*h_factor; x<260*h_factor; x+=62*h_factor) {
        n_effect = 1;
        for (y=25*v_factor; y<126*v_factor; y+=25*v_factor) {
            effect = b_effect | n_effect;
            vst_effects (handle,effect);
            v_gtext (handle,x,y,"EFFECTS");
            n_effect <= 1;
        }
        b_effect <= 1;
    }
    v_show_c (handle);
    button_wait();
}
```

```
do_height()
{
    int height, x, y;

    v_hide_c (handle);
    v_clrwk (handle);
    vst_effects (handle,0);
    for (height=1; height<27; ++height) {
        x += 8; y += 7;
        vst_height (handle,height,&dummy,&dummy,&dummy,&dummy);
        v_gtext (handle,x*h_factor,y*v_factor,"Height");
    }
    v_show_c (handle);
    button_wait();
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_rotate()
{
    int angle;

    v_hide_c (handle);
    v_clrwk (handle);
    vst_height (handle,8,&dummy,&dummy,&dummy,&dummy);
    for (angle=0; angle<2701; angle+=900) {
        vst_rotation (handle,angle);
        v_gtext (handle,160*h_factor,96*v_factor,"ROTATION");
    }
    vst_rotation (handle,0);
    v_show_c (handle);
    button_wait();
}
```

```
draw_menu()
{
    int height;

    v_hide_c (handle);
    v_clrwk(handle);
    draw_rec (rec1,GREEN,SOLID,0);
    draw_rec (rec2,BLACK,HOLLOW,0);
    v_pline (handle,2,line1);
    v_pline (handle,2,line2);
    vst_height (handle,10,&dummy,&dummy,&dummy,&dummy);
    vst_color (handle,RED);
    vst_effects (handle,NORMAL);
    v_gtext (handle,110+152*t_factor,72*v_factor,"EFFECTS");
    v_gtext (handle,116+152*t_factor,104*v_factor,"HEIGHT");
    v_gtext (handle,116+152*t_factor,136*v_factor,"ROTATE");
    v_show_c (handle);
}
```

```
draw_rec(rec,fcolr,inter,style)
int rec[];
int fcolr,inter,style;
{
    int rxc[4];
    int x;

    for (x=0; x<4; ++x)
        rxc[x] = rec[x];
    vsf_color(handle,fcolr);
    vsf_interior(handle,inter);
    vsf_style(handle,style);
    v_bar(handle,rx);
}
```

```
open_vwork()
{
    int i;

    handle = graf_handle(&dummy,&dummy,&dummy,&dummy);
    for (i=0; i<10; work_in[i++] = 1);
    work_in[10] = 2;
    v_opnvwk (work_in, &handle, work_out);
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
init()
{
    int x;

    res = Getrez();
    switch (res) {
case 0 :
    h_factor = 1;
    v_factor = 1;
    t_factor = 0;
    break;
case 1 :
    h_factor = 2;
    v_factor = 1;
    t_factor = 1;
    break;
case 2 :
    h_factor = 2;
    v_factor = 2;
    t_factor = 1;
    }
    for (x=0; x<4; ++x)
        if (x == 0 || x == 2) {
            rec1[x] = rec1[x] * h_factor;
            rec2[x] = rec2[x] * h_factor;
            line1[x] = line1[x] * h_factor;
            line2[x] = line2[x] * h_factor;
        }
        else {
            rec1[x] = rec1[x] * v_factor;
            rec2[x] = rec2[x] * v_factor;
            line1[x] = line1[x] * v_factor;
            line2[x] = line2[x] * v_factor;
        }
    graf_mouse (HAND,mouse_form);
}

button_wait()
{
    int button, mx, my;

    button = 0;
    while (button == 0)
        vq_mouse(handle, &button, &mx, &my);
    while (button > 0)
        vq_mouse(handle, &button, &mx, &my);
}

mouse_print(mx,my)
int mx, my;
{
    char tx[5], ty[5];

    vst_height (handle, 6, &dummy, &dummy, &dummy, &dummy);
    sprintf(tx, "%d ", mx);
    sprintf(ty, "%d ", my);
    v_gtext(handle, 20, 30, tx);
    v_gtext(handle, 52, 30, ty);
}
```

## CHAPTER 12 - ALERT BOXES AND CUSTOM MOUSE FORMS

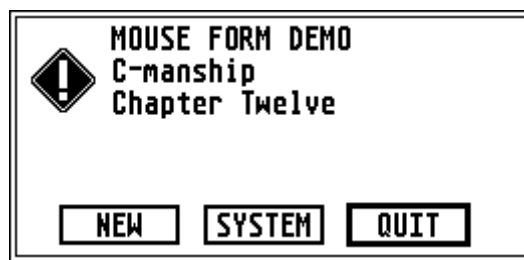
We've spent the last couple of chapters examining GEM's VDI. We didn't cover everything, but we managed to touch upon most of the major functions. Some of the ones we glossed over are easy enough to figure out from the documentation supplied with your compiler; others, we'll get to as we need them, particularly the raster functions.

In this chapter, we'll get started with GEM's AES (Application Environment Services). We'll learn how to create GEM alert boxes, a little about the interaction of the AES with the VDI, and how to define our own mouse forms.

### Getting to Work

When you run this chapter's program, you'll be presented with an alert box like the one shown in Figure 1. Use the mouse to click on the first button (the one labeled "New"). The screen will clear, and the mouse pointer will change to a custom form. (You may have seen this cursor before. I used it in Moonlord ST, a game that was published some time ago in ST-Log.) Clicking the left button will return you to the alert box.

Now, click the button labeled "System." The mouse form will change to one of the system cursors, the pointing finger. The last button is self-explanatory (I hope).



### A Small Matter of Incompatibility

Before we get started with the nitty-gritty material, there's something you should be aware of whenever you're going to use AES or VDI mouse routines. The AES has close ties with the VDI; in fact, it relies on the VDI to do much of the dirty work. For instance, when you call the AES window-drawing functions, the window is created, in part, using VDI graphics. That's why some of the VDI routines are referred to as "graphics primitives." They're the foundation upon which all the sophisticated ST graphics are built. The VDI is, in a way, a subordinate of the AES.

In most cases, when dealing with graphics, there's no problem with this hierarchy, but when you start handling mouse events (a fancy name for mouse input), it's easy to confuse the AES. Basically, you can use the mouse-handling routines in the AES or in the VDI, but not both at the same time. If you want to be on the safe side, use only the AES mouse functions.

That's why `button_wait()` in Listing 1, a function that appeared in a different form in Chapter 11, had to be modified, replacing the VDI calls in the original with the AES calls found in this chapter's version. The AES alert box routines must, obviously, also handle the mouse. If we tried to use the VDI mouse routines, we'd have trouble. (Try it if you like; replace the new `button_wait()` with the old one. Then, if you leave the mouse in one place when clicking on a button, you'll find that the mouse will

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

reclick the alert box with no help from you). If you look at `button_wait()`, you'll see that we've replaced the VDI call `vq_mouse()` with an AES call, `evnt_button()`.

The function `evnt_button()` is a higher-level call and, as a result, is more complicated and flexible. When called, the function waits for a mouse button to be pressed. The call looks like this:

```
n_times = evnt_button(n_clicks, btn, state, &mx, &my, &ex_state, &k_state);
```

All the parameters, including the return parameter, are integers or pointers to integers, and are described here:

n_times	The number of times the button attained the desired state.
n_clicks	The number of times the button must be clicked.
button	The button (left or right) which must be clicked. A value of 1 indicates the left button, a value of 2, the right.
state	The state the mouse button must attain. A value of 0 is up, and a value of 1 is down.
mx	The X-coordinate of the mouse when the button event occurred.
my	The Y-coordinate of the mouse when the button event occurred.
ex_state	The state of the mouse buttons after exiting the function.
k_state	The keyboard's state after exiting the function. The values 1, 2, 4 or 8 indicate that the right shift, left shift, control key or alternate key were pressed, respectively.

As you can see, this call is more complicated than our old friend `vq_mouse()`, but allows us more options.

### Alert Boxes

The alert box is the simplest of GEM's form library to use, since the system handles virtually everything for you. All you need to do is provide the proper information for the function call. To draw an alert box:

```
choice = form_alert(deflt, string);
```

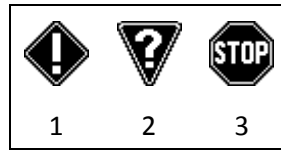
Here, `choice` is the button number pressed (returned from the function), `deflt` is the number of the default button (the button, if any, that will respond to the Return key), and `string` is a pointer to a string containing the alert box description. You may also use a string literal for the second parameter, by enclosing it in quotes. In fact, it's done that way in the sample program.

The alert box description contains all the information GEM needs to draw your box: the icon that will be displayed, and the text for both the box and the buttons. The string actually has three segments separated by square brackets:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
[icon #][box text][button text]
```

The icon # is the number of the icon you wish displayed, defined as follows (a value of 0 will not display an icon):



The text for the box may be up to thirty-two characters per line, with a maximum of five lines. How does GEM know where to divide the text? We tell it, by placing an OR symbol between each line:

```
[line1|line2|line3|line4|line5]
```

Each alert box you design must have at least one button, but you may have up to three. The information for the buttons comes after the box text and consists of the text to be printed within each button. The buttons' texts are placed between square brackets, with each buttons' text separated by an OR symbol:

```
[button1|button2|button3]
```

The text for each exit button must be less than twenty characters. You can see the completed string in this chapter's sample program in the function `do_alert()`, found about halfway down the listing. Notice that, due to the length of the string, it had to be wrapped around to the next line. Normally, you can't divide a string, but by using a backslash (at the end of the first portion of the string), we can get around that limitation. When the C preprocessor sees the backslash, it knows that the rest of the string will begin at the left margin of the next line. If you want to use a backslash within a string, you must type two.

### They Don't Fit!

When the `form_alert()` function is called, it uses the information you've supplied to figure out the number of buttons and the size of the box. Almost everything is taken care of for you, but you may find it necessary to "clean up" the box description a bit, in order to force GEM to do exactly what you want.

For instance, the number of buttons that will fit in the box is largely dependent on the length of the text and the size of the icon (if any) printed in the box. If the resultant alert box is only slightly smaller than the space needed for the buttons, GEM will place the buttons so that they overlap the box's borders. This type of box is not particularly attractive but will be fully functional. If the box is significantly undersized, GEM will start leaving buttons out, and you can't live with that.

These problems don't usually crop up with single-button alert boxes (unless the button text is unusually long), but when you start dealing with three exit buttons, the glitches will likely introduce themselves.

How can we force GEM to do what we want? Remember that the size of the box is dependent on the length of the text lines and the size of the icon, while the size of the buttons is dependent on the text printed within them. The icon size is unchangeable; it's set by the system, and the only way we can

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

manipulate the icon is either to print it or not. But the box text and button text is fully under our control.

One way, then, to help fit the required buttons into the box is to shorten the text within them. If the button text is just the way we want it, and we still can't fit all the buttons, we have to resort to the second method: padding the beginning or end of the box text with spaces. This will force GEM to draw the box larger. You can see an example of this in the `form_alert()` call in Listing 1. Try removing the additional spaces and recompiling the program. You'll find that, in medium resolution, the left-most button will overlap the box's border; in low resolution, the button is missing.

### Custom Mice

GEM provides us with a number of built-in mouse forms, but we may sometimes find a need for something better suited to our application. When this occurs, `graf_mouse()` comes to the rescue. We discussed this function in Chapter 11, but didn't cover the method for designing custom mouse forms. Now, you'll be pleased to know, we're going to make up for that lack.

A mouse form is actually two graphics, 16x16 pixels in size, placed one on top of the other. The first graphic is the shape of the pointer itself. The second is the pointer's mask, which enhances its visibility. If you examine a mouse form, you'll notice that there's a one-pixel wide border around it. The border is a different color so that the mouse pointer won't "vanish" if it should be moved over something of the same hue. This border is the mask.

The first step in designing a mouse form is to draw the form and its associated mask on a sheet of graph paper, each within a 16x16 grid. We must then translate the graphics to something the computer can understand.

Our C program is going to need the data for the new form in some sort of numerical notation. Hexadecimal notation is best for our purposes, if for no other reason than the ease with which it's calculated from the binary representation of our graphics. If you don't know how to make these conversions, I suggest you go to your local library or bookstore for something which explains binary to hexadecimal conversions.

The binary version of our graphic is simple enough to explain, though. Each grid location not filled in is an "off" bit; the others are "on." Each of these bits is represented in binary, by a 0 for off or a 1 for on. Figures 2 and 3 illustrate the conversion of our custom pointer from its graphic state to hexadecimal.

### Coding It

Once we've done the conversion, we must incorporate the result into our program. The easiest way to do this is to store the data for the form and its mask in two integer arrays. If you look at the sample listing, you'll see our custom mouse pointer in the arrays `mouse_data[]` and `mouse_mask[]`. The "0x" preceding each value tells the compiler that the number should be interpreted as hexadecimal.



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

Just above the mouse form data in the sample listing is the declaration for a structure named `mfrmstr` (mouse form structure). To be more precise, it's not a structure declaration, but the declaration of a new data type consisting of a structure. We've defined this new data type by prefacing the structure definition with the C keyword `typedef`. Right below the declaration is where the actual structure variable, `mouse`, is declared.

This structure will hold all the information GEM needs to enable our new mouse form. As you can see, the block contains 37 words of information. The first two words will hold the X- and Y-coordinates of the form's "hot spot." (Sounds pretty sleazy, doesn't it?) This is the location within the form which determines

the X- and Y-coordinates for the entire mouse cursor. The third word will contain the number of bit planes. For high, medium, and low resolutions, this value will be 1, 2, and 4, respectively. The fourth word will indicate what color the mouse form should be, and the fifth word will hold the color for the mask. The next thirty-two words are storage for the actual mouse form data. We'll move the values found in the arrays `mouse_data[]` and `mouse_mask[]` into these locations.

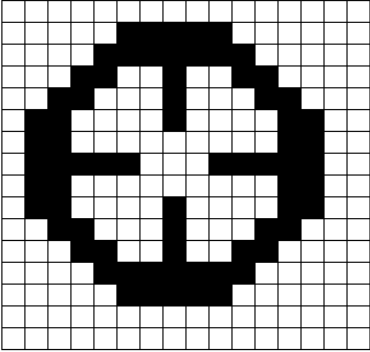
To change the mouse form now, all we need do is fill in each member of the structure with the appropriate information and perform the following call:

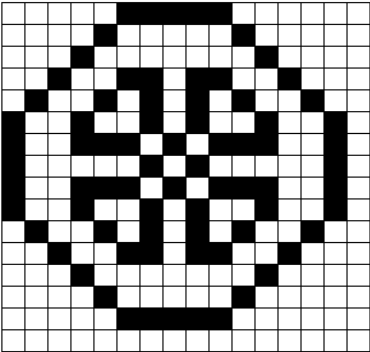
```
graf_mouse(255, &mouse);
```

We went over this function in Chapter 11. What's important here, is that the parameter 255 tells GEM that we want to change to a user-defined mouse form. The parameter `&mouse` is the address of the block of data containing the form's description.

### Mission Accomplished

There you have it: everything you need to know, to get the most out of alert boxes and to design your own mouse forms. As soon as you tear your eyes from this page, yank out your C compiler and fool around with the `form_alert()` function. Try different combinations of text and buttons, until you feel comfortable with the function. Then, design some alternate mouse pointers and modify Listing 1 (or write your own code from scratch; that's really the best way to learn) to install your new forms. How about changing Listing 1, so the alert box buttons allow you to alternate between two custom mouse forms? That will mean having two sets of data, one for each form, and changing the `new_mouse()` function so you can pass it the address of the form description for the cursor you want to implement.

GRAPHIC	BINARY	HEX
	0000000000000000	\$0000
	0000011111000000	\$07C0
	0000111111100000	\$0FE0
	0001100100110000	\$1930
	0011000100011000	\$3118
	0110000100001100	\$610C
	0110000000001100	\$600C
	0111110001111100	\$7C7C
	0110000000001100	\$600C
	0110000100001100	\$610C
	0011000100011000	\$3118
	0001100100110000	\$1930
	0000111111100000	\$0FE0
	0000011111000000	\$07C0
	0000000000000000	\$0000
	0000000000000000	\$0000
The Shape		

GRAPHIC	BINARY	HEX
	0000011111000000	\$07C0
	0000100000100000	\$0820
	0001000000010000	\$1010
	0010011011001000	\$26C8
	0100101010100100	\$4AA4
	1001001010010010	\$9292
	1001110101110010	\$9D72
	1000001010000010	\$8282
	1001110101110010	\$9D72
	1001001010010010	\$9292
	0100101010100100	\$4AA4
	0010011011001000	\$26C8
	0001000000010000	\$1010
	0000100000100000	\$0820
	0000011111000000	\$07C0
	0000000000000000	\$0000
The Mask		

## Program Listing #1

```

/*****
/*          C-MANSHIP, Listing 1          */
/*          CHAPTER 12                    */
/*          Developed with Megamax C      */
*****/

#define TRUE 1
#define FALSE 0
#define FINGER 3
/* Required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* A couple of global int variables */
int handle, dum;

/* Mouse form definition block */
typedef struct mfrmstr
{
    intx_hot;    /* x-coordinate of pointer hot spot. */
    inty_hot;    /* y-coordinate of pointer hot spot. */
    intplanes;   /* number of bit planes. */
    intfg_color; /* mouse form color. */
    intbg_color; /* mouse mask color. */
    intmask[16]; /* Data for mouse mask. */
    intdata[16]; /* Data for mouse form. */
} MOUSEFORM;
MOUSEFORM mouse;

/* Data for the new mouse form */
int mouse_data[] = {0x0000,0x07C0,0x0FE0,0x1930,
0x3118,0x610C,0x600C,0x7C7C,
0x600C,0x610C,0x3118,0x1930,
0x0FE0,0x07C0,0x0000,0x0000};

int mouse_mask[] = {0x07C0,0x0820,0x1010,0x26C8,
0x4AA4,0x9292,0x9D72,0x8282,
0x9D72,0x9292,0x4AA4,0x26C8,
0x1010,0x0820,0x07C0,0x0000};

main() /* Main program */
{
    appl_init(); /* Initialize our application. */
    open_vwork(); /* Go set up our workstation. */
    do_alert(); /* Go to the main loop. */
    appl_exit(); /* Back to the desktop. */
}

open_vwork() /* Initialize a virtual workstation */
{
    int i;
    handle = graf_handle(&dum,&dum,&dum,&dum);
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
for (i=0; i<10; work_in[i++] = 1);
work_in[10] = 2;
v_opnvwk(work_in, &handle, work_out);
}

do_alert() /* This is the main loop. It calls the alert box */
{
    /* function and the functions to change the mouse */
    /* form. The loop repeats until REPEAT == false. */

    int    choice, /* Will hold button choice.      */
           repeat, /* Loop control variable.        */
           deflt;  /* Holds default button choice. */

    repeat = TRUE; /* We set this so the loop will repeat. */
    deflt = 3;     /* Set default button to Quit (see below). */
    while (repeat) {
        choice = form_alert (deflt,
            "[1][    MOUSE FORM DEMO      |    C-manship|\n"
            "Chapter Twelve][New|System|Quit]"); /* Draw alert box. */
        if (choice == 1) { /* CHOICE contains the button pressed. */
            new_mouse();   /* If button was NEW, show new form. */
            button_wait();
            deflt = 2;     /* Change default button to 2. */
        }
        if (choice == 2) { /* If the second button was pressed, */
            graf_mouse (FINGER,&dum); /* then change to Hand icon. */
            button_wait();
            deflt = 1;     /* Change default button to 1. */
        }
        if (choice == 3)
            repeat = FALSE; /* When REPEAT == false (0), we get out
*/
    }
    /* of the while loop and go to main(). */
}

new_mouse() /* Changes mouse form to the user-defined form */
{
    /* found in the global arrays at top of listing */
    int x;

    mouse.x_hot = 8; /* These two assignments set "hot spot" */
    mouse.y_hot = 8; /* to the center of the mouse form. */
    mouse.planes = 4; /* Set to 1 for high res and 2 for med. */
    mouse.fg_color = 0; /* Mouse form drawn with color 0. */
    mouse.bg_color = 2; /* Mouse mask drawn in color 2. */
    for (x=0; x<16; ++x) { /* This loop moves the data from */
        mouse.mask[x] = mouse_mask[x]; /* global arrays into the */
        mouse.data[x] = mouse_data[x]; /* mouse form def'n block */
    }
    graf_mouse(255,&mouse); /* Presto! Our new mouse lives. */
}

button_wait() /* Waits for left button to be pressed and released. */
{
    int dum;

    evnt_button(1,1,1,&dum,&dum,&dum,&dum);
    evnt_button(1,1,0,&dum,&dum,&dum,&dum);
}
```

## CHAPTER 13 - THE FILE SELECTOR AND RASTER OPERATIONS

As I Mentioned Before, GEM's AES contains a number of libraries, one of which is the form library. In Chapter 12, we were briefly exposed to the form library when we learned how to handle alert boxes, the simplest of the ready-to-use forms. However, most of the forms you'll employ once you get used to programming with GEM will be dialog boxes.

Dialog boxes are complex and can be put together in almost any form imaginable. In an upcoming chapter, we'll sit down and have a long talk about these puzzling creatures, but for now, there's still one other ready-to-use form that we haven't explored yet: the file selector.

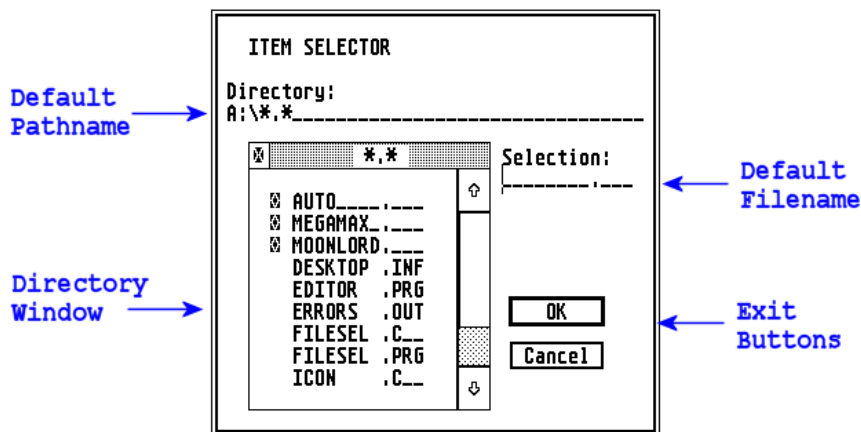


FIGURE 1 - The File Selector Box

At first glance, one might think the file selector is difficult to handle (from a programmer's point of view), what with those slider bars and the editable text fields, and the exit buttons. The truth is that file selectors aren't much more difficult to program than alert boxes, because, just as with alert boxes, GEM handles much of the busy work for us.

### Picking a File

Listing 1 shows how to use a file selector within a C program. When you run the program, you'll be presented with a file selector that looks something like Figure 1 (unless, of course, you've got one of those fancy file selector replacements in your AUTO folder). Choose one of the files. Then press an exit button. The file selector will be replaced with two lines of text showing the chosen file and the exit button you clicked.

At this point, you're probably a little fuzzy on exactly what a file selector does for you. What kind of information does it return? Even though the file selector looks complicated, and allows the user to fiddle with scroll bars and buttons and text fields, all it really does is return a filename and the number of the exit button pressed. It's up to the programmer to decide what to do with the information. In most cases, the user will be selecting a data file -- such as a document for a word processor -- and we'll use the filename returned to open that file and read the data into the program.

### Calling Up a File Selector

One simple call will get the file selector box up on your screen:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
fsel_input (path,file,&button);
```

But there's a bit of preparation that must be done first. In the above function call, path is a pointer to a string in which the default pathname is stored, and file is a pointer to a string containing the default filename (the text field to the center right of the box). The integer button will contain the value of the exit button chosen, where 0 equals the cancel button and 1 equals the OK button.

The pointers path and file actually serve a dual purpose. Both of the text fields they represent are editable. Upon exit from the selector, they'll contain the strings typed by the user (if nothing was typed, they'll still contain whatever you put there). This is how we can find the file or path the user selected.

But there are still a couple of things you need to know before you can start using file selectors. For instance, how are the strings pointed to by path and file formatted? The answer can be found in the function sel\_file() in Listing 1.

### File Selector Housekeeping

The first thing we must do in sel\_file() is declare the variables we need and set aside some space for filenames. It's important that you reserve enough memory. Otherwise, strings typed by the user may overrun their allotment and tromp over other data. The storage area for the default pathname, path[50], is probably larger than we'll need, but it's better to be safe.

Let's see what might happen if the array were smaller, say only 20 bytes (path[20]). Now, what if the file the user wants to select is found buried within two folders? We could end up with a pathname like:

```
A:\FOLDER.ONE\FOLDER.TWO\FILENAME.EXT
```

That gives us a pathname that's 37 bytes long. Our storage area will hold only twenty characters. Watch out for that.

The storage for the default filename, file[13], isn't as tricky, since no filename will ever exceed thirteen characters (including the \0 terminator).

After we've set up our variables and storage space, we must do some initialization. First, we fill the default path and filename areas with nulls, getting rid of all the junk. We then ask the system for the default drive (the one the program was loaded from; any filename that doesn't specify a drive will use the default), convert it to ASCII, and store it in the first element of path[] with the line:

```
path[0] = Dgetdrv() + 65;
```

Dgetdrv() is a GEM DOS call (gemdos(0x19) for those who are interested) and returns the number of the default drive as an integer where 0 means drive A, 1 means drive B, and so on. Since our pathname must be a string, we need to convert the drive number to the ASCII equivalent. And what's ASCII for A? Sixty-five, right? So all we have to do is add 65 to the drive number, then place this value in the first element of our string, and we're on our way to creating the default pathname.

We finish our pathname with the statement:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
strcpy (&path[1], ":\*.");
```

The function `strcpy()` copies the string (including the null) pointed to by the second argument to the string pointed to by the first argument. In the example above, the colon will be copied into the second element of our pathname, with the rest of the characters in the string literal following. This is just one of many handy string-handling functions available with Megamax C.

Other string-handling functions include `strcat()` and `strncat()` which concatenate strings; `strcmp()` and `strncmp()` which compare strings; `strlen()` which returns the number of characters in a string; and `index()` and `rindex()` which return a pointer to the first or last occurrence of a character in a string, respectively. The details of these functions can be found in your compiler manual.

Finally, in `sel_file()` we open the file selector, then print the results of the user's selections.

Now that we've got the file selector mastered, let's move on to something really challenging.

### Raster Operations

Many of you may have heard the term Bit Block Transfer, or BITBLT as it's more commonly known. This is the name sometimes given to the VDI's raster operations. What's a raster operation? Simply, it's the movement of blocks of memory, usually from screen memory to someplace else in RAM or vice versa. Many of the programming techniques you'll be learning will require a good knowledge of raster operations. Rastering is used to draw icons and sprites, and also to update windows.

GEM's VDI contains a number of functions that help the programmer perform this memory juggling, but in order to take advantage of these functions, we must first have a way to describe the blocks of memory we want to move. We supply this information with a Memory Form Definition Block (MFDB).

The MFDB consists of ten words of information: the address of the memory we want to move; its height and width; the coordinate system we're using (raster or normalized); and the number of bit planes that make up our screen. (There are also several words that, although ignored, must be present.)

C provides a handy way to group this information into a single unit: the structure. Our MFDB, then, looks something like this:

```
typedef struct mfrmbblk {
    long f_addr;
    int f_w;
    int f_h;
    int f_wdwidth;
    int f_stand;
    int f_nplanes;
    int f_r1, f_r2, f_r3;
} MFDB;
```

Here, `f_addr` is the address of the memory block, `f_w` and `f_h` are the width and height of the block in pixels, `f_wdwidth` is the width of the block in words, `f_stand` is the coordinate system (0 for raster, 1 for normalized), and `f_nplanes` is the number of bit planes. The integers `f_r1`, `f_r2` and `f_r3` are reserved for future use and may be ignored (I usually set them to 0 just to be safe).

## Filling in the Blanks

Confused yet? I thought you might be. All this talk of coordinate systems and bit planes can be -- if you've never been exposed to it before -- daunting. Bit planes were mentioned in Chapter 12, when we designed our own mouse forms, but now it's time to learn a little more about the way your ST's screen memory works.

The ST reserves 32K of memory for the screen, no matter what resolution you're in. In high resolution, the organization of this memory is simple: each bit in memory represents one pixel on the screen.

The first 640 bits (80 bytes) represent the first row of the screen; the second 640 bits represent the second row of the screen; and so on, for 400 rows. If we multiply 80 bytes per row times 400 rows, we get the magical number 32,000, the size of screen memory. If a bit is on, the corresponding pixel will be a black dot; if a bit is off, the pixel will be white.

When we talk about low or medium resolution, however, we throw in an extra complication: color. Now, we have to know more than just whether a pixel is on or off; we need to know its color. And we still need to get all this information into 32K.

In medium resolution, we're allowed four colors. It takes two bits to store this information (four possible combinations: 00, 01, 10, 11) versus the one bit needed to represent black and white, which means 32K of screen memory can hold only enough information for 128,000 pixels instead of the 256,000 pixels we had in monochrome. (Wow! A quarter of a million!) In order to compensate for this, the designers of the ST decided to halve the vertical resolution, giving us a screen 640x200. Eighty bytes (640 bits) times 200 lines times two bit planes per pixel gives us a total screen memory of 32K.

In low resolution, we have 16 colors to work with. Since it takes four bits to represent 16 combinations, we find that we must again cut the number of pixels in half, to 64,000. This time, the ST's designers made up the difference by halving the horizontal resolution and the vertical resolution (as compared to high resolution), to give us a screen 320x200.

That's not the end of the story. In the color modes, the screen memory is divided into bit planes (see Figure 2). You can think of the bit planes as transparencies laid one on top of the other. In order to get the color value for the first pixel on the screen, you must combine the first bit of each plane. The second bit of each plane forms the color value for the second pixel; the third for the third; etc. In medium resolution, there are two bit planes. In low, there are four.

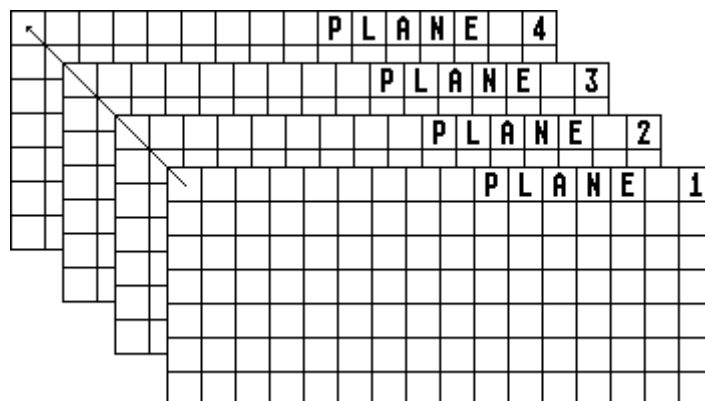


FIGURE 2 - Bit planes for low resolution

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Now that we understand (yeah, right) all this nonsense about bit planes, what's with these coordinate systems? When programming in GEM, there are two coordinate systems you may choose between when you open a new workstation: Normalized Device Coordinates (NDC) or Raster Coordinates (RC).

The NDC system divides the screen into a grid that's 32,768x32,768 with the origin (point 0,0) in the lower-left corner. Moving to the right from the origin increases the value of the X-coordinate, while moving upward from the origin increases the value of the Y-coordinate.

The RC system is the one we usually use on the ST, where the origin is in the upper-left corner of the screen, and the width and height of the screen depend on the current resolution. The sample program (Listing 2) uses the RC system.

### The Next Listing

When you compile and run Listing 2, the infamous ANALOG "A" (in multicolors) will appear. Use the mouse to point and click anywhere on the screen. The "A" will move to that location. When you're through, press the right mouse button to return to the desktop.

The "A" that you've been moving around the screen is an example of an icon, and is an image we've stored in memory. An icon is usually designed by drawing it with a graphics program, such as NeoChrome or DEGAS, then converting the image to its hexadecimal equivalent. This can be a complicated procedure, especially if you're dealing with a color mode and all those bit planes. Your best bet is to get hold of one of the public domain icon editors floating around. At any rate, the icon editor will take the image you've created and convert it to data which can be merged with your source code.

Take a look at Listing 2. About a third of the way down, you'll find the array, `icon[]`. See all that data? That's the hexadecimal representation of our ANALOG icon as it appears in low resolution (remember -- four bit planes). The data would look different in medium or high resolution, because we wouldn't be dealing with as many bit planes. In fact, in high resolution, there would be only one-fourth as much data, since there would be no colors to keep track of.

Now, take a look at the function `do_icon()`, where the main logic for the demo program is found. First, we change the mouse form to the pointing finger and initialize some variables. Then, after setting our control variable, `repeat`, to true, we enter the main while loop.

Once in the loop, we adjust the mouse coordinates (X and Y) so the icon will be drawn in the right place. We have to do this, because the coordinates we get from the mouse are the location of the mouse's hot spot (oooh, I love it when we talk dirty); the coordinates we need for the raster functions are the upper-left and lower-right corners (actually, any diagonally opposed corners) of the block of screen memory. If we didn't do this extra calculation, the icon would always be drawn below and to the right of the mouse pointer.

We then save the coordinates so that when we get a new mouse X and Y, we will be able to erase the first drawing. Finally, we turn off the mouse and call the function `draw_icon()` to actually draw the icon on the screen.



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

### The Raster Details

Which brings us to the point of this lengthy discussion (you knew there had to be a point, right?): the VDI call `vro_cpyfm()`. This function actually performs the rastering (there's a second VDI raster function, `vrt_cpyfm()`, which is very similar except that it's used to copy forms designed for monochrome onto a color screen). The function is called like this:

```
vro_cpyfm(handle, mode, pxy, &mfdb1, &mfdb2);
```

Here, the integer `handle` is the handle returned when we opened the virtual workstation; the integer `mode` is the raster writing mode; the pointer `pxy` is the address of an array of integers describing coordinates of the two rectangles; and `&mfdb1` and `&mfdb2` are pointers to the two MFDBs that describe the areas to be rastered. Gasp!

The parameter `mode` can be any number from 0 to 15. The writing mode is the logical operation that's used to combine the source and destination values. There are 16 logical operations available to us. In the sample program, we're exclusive ORing the source and destination. This way, an image can be easily erased by redrawing it in the same (exclusive OR) mode. The disadvantage to this mode is that, if we're not working with a blank screen, our image will be transparent, allowing the background to bleed through.

I'm not going to spend a lot of time describing the different writing modes. You should look them up in your manual or experiment with them. You'll probably find that there are only a couple you'll use; the rest are there should you need them.

The `pxy` array holds our rectangles' coordinates: the upper-left and lower-right corners of both the source and destination rectangles. They should be stored in this order:

```
sx1, sy1, sx2, sy2, dx1, dy1, dx2, dy2
```

Now that we understand the `vro_cpyfm()` call, let's take a look at `draw_icon()` for the details. First, we must initialize the two MFDBs. We don't have to talk much about this, since the MFDBs are fairly well described above. However, there are a couple of things that should be clarified.

For one thing, if you look at the data for the icon in the source code, it would appear to be 6 long words wide or 192 bits -- a big icon! Now, anyone who can tell me why our data is 192 bits wide, raise your hand. Those of you who are slinking down under your desks in embarrassment can relax; I'm not going to call on you. But do the words "bit planes" jar any memories?

"Yes!" you say. "Yes! That's why we've got 192 bits. We're dealing with 4 bit planes, and 192 divided by four is...is...is..."

Forty-eight.

"Yeah...thanks."

You're welcome. Now you know why we've made `icn_w` equal to 48 instead of 192. But here's another question for you: How come the icon, when it's on the screen, appears only 23 pixels wide?

"I don't know," you mumble, climbing back under your desk.

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

Well, climb back into the light, my friend; it's not your fault that you can't answer the last question. There's something I haven't told you. Because of the way the raster functions move data, the width of a data block must be a multiple of 16 (this allows more efficient movement of data). In the case of our "A" icon, we've had to pad the left and right of the image with "off bits" in order to bring the width up to the next highest multiple of 16 which is, of course, 48.

Now, we get to the screen MFDB. If you look at `draw_icon()`, you'll see that each member of the icon's MFDB, `s_m`, had to be initialized properly. Yet, for the screen MFDB, `scr_m`, we've only one line of code:

```
scr_m.F_addr = 0L;
```

Why? Whenever the form address element (`f_addr`) of the MFDB is set to 0, the system knows it will be dealing with the screen and will automatically handle everything for you. You don't have to fill in the rest of the MFDB. You can if you want, but, unless you're the type of person who enjoys painting houses with a hair, why bother?

### Off Again

I know I say this at the end of just about every chapter, but I'm going to say it again: practice! Everything you're learning builds upon what has gone before. Just like a course in mathematics, if you miss a lesson or don't understand it completely, you'll get more and more confused as you try to go on.

Try designing your own icons and raster them to the screen, experimenting with the different writing modes to see the results (there's an excellent illustration of the 16 modes on page 228 of the Programmer's Guide to GEM, published by Sybex, for those of you who have that book). Of course, in order to experiment with the writing modes, you're going to need some sort of graphic in the background. Looks like you'll be getting some more practice with the VDI, eh?

### Program Listing #1

```
/* **** */
/*          C-MANSHIP, Listing 1          */
/*          CHAPTER 13                     */
/*          Developed with Megamax C      */
/* **** */

#include <osbind.h>

/* The usual required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum;
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Main program */
main ()
{
    appl_init ();          /* Initialize application.          */
    open_vwork ();         /* Set up workstation.          */
    sel_file ();           /* Go select file.              */
    button_wait ();        /* Wait for a mouse button press. */
    v_clsvwk (handle);     /* Close virtual workstation.    */
    appl_exit ();          /* Back to the desktop.          */
}

/* Initialize a virtual worksation. */
open_vwork ()
{
    int i;

    handle = graf_handle (&dum,&dum,&dum,&dum); /* Get handle. */
    for (i=0; i<10; work_in[i++] = 1); /* Init GEM arrays. */
    work_in[10] = 2;
    v_opnvwk (work_in, &handle, work_out); /* Open virtual
                                           workstation.
*/
}

/* Do file selector box. */
sel_file ()
{
    int button,          /* File selector button value. */
        i;              /* Loop variable.              */
    char path[50], /* Storage for filenames.      */
        file[13];

    for (i=0; i<20; path[i++]='\0'); /* Fill filename w/ nulls. */
    for (i=0; i<13; file[i++]='\0');
    path[0] = Dgetdrv() + 65; /* Make drive # a char. */
    strcpy (&path[1], ":\*."); /* Complete the pathname. */
    fsel_input (path,file,&button); /* Open file selector box. */
    prnt_info (file,button); /* Go print results. */
}

/* Print out the user's choices. */
prnt_info (file,button)
char *file; /* Pointer to the chosen filename. */
int button; /* Value of the button pressed. */
{
    v_gtext (handle,28,50,"The file you chose was: ");
    v_gtext (handle,220,50,file);
    v_gtext (handle,28,66,"And you pressed the ");
    if (button == 0)
        v_gtext (handle,188,66,"CANCEL button.");
    else
        v_gtext (handle,188,66,"OK button.");
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Waits for left button to be pressed and released. */
button_wait()
{
    evnt_button (1,1,1,&dum,&dum,&dum,&dum);
    evnt_button (1,1,0,&dum,&dum,&dum,&dum);
}
```

### Program Listing #2

```
/******
/*          C-MANSHIP, Listing 2          */
/*          CHAPTER 13                    */
/*          Developed with Megamax C      */
/******

#include <osbind.h>

#define S_XOR_D 6
#define TRUE 1
#define FALSE 0
#define LEFT 1
#define RIGHT 2
#define HAND 3
#define OFF 256
#define ON 257

/* The required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum;

/* Memory Form Definition Block */
typedef struct mfrmblk {
    long f_addr;      /* Addr of form data.          */
    int f_w;          /* Width of the form in pixels. */
    int f_h;          /* Height of the form in pixels. */
    int f_wdwidth;    /* Width of the form divided by 16. */
    int f_stand;      /* 0 = raster (RC); 1 = normalized (NDC). */
    int f_nplanes;    /* Number of bit planes (1, 2 or 4). */
    int f_r1;         /* The last three words are reserved. */
    int f_r2;
    int f_r3;
} MFDB;

/* Data for the ANALOG "A" icon. */
long icon[] = {
    0x00000000,0x00000000,0x1FFF1FFF,0x1FFF1FFF,0xF000F000,0xF000F000,
    0x00000000,0x00000000,0x35552CCB,0x3C3823F8,0x58003800,0xF8000800,
    0x00000000,0x00000000,0x55554CCB,0x7C3843F8,0x58003800,0xF8000800,
    0x00000000,0x00000000,0xD555CCCB,0xBC3883F8,0x58003800,0xF8000800,
    0x00010001,0x00010001,0x57FDCFFF,0x3FFC07FC,0x58003800,0xF8000800,
    0x00030002,0x00020002,0x5803C803,0x38020802,0x58003800,0xF8000800,
    0x00030002,0x00020002,0x5803C803,0x38020802,0x58003800,0xF8000800,
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
0x00030002,0x00020002,0x5403CC03,0x3C020402,0x58003800,0xF8000800,
0x00030002,0x00020002,0x57F3CFF3,0x3FF207F2,0x58003800,0xF8000800,
0x00030002,0x00020002,0x555BCCCB,0x3C3A03FA,0x58003800,0xF8000800,
0x00030002,0x00020002,0x555BCCCB,0x3C3A03FA,0x58003800,0xF8000800,
0x00030002,0x00020002,0x55F3CDF3,0x3DF203F2,0x58003800,0xF8000800,
0x00030002,0x00020002,0x5503CD03,0x3D020302,0x58003800,0xF8000800,
0x00030002,0x00020002,0x5583CC83,0x3C820382,0x58003800,0xF8000800,
0x00030002,0x00020002,0x5583CC83,0x3C820382,0xF800F800,0xF800E800,
0x00030002,0x00020002,0x5543CCC3,0x3C4203C2,0xF800F800,0xF800E800,
0x00030002,0x00020002,0x5543CCC3,0x3C4203C2,0xF800F800,0xF800E800,
0x00030002,0x00020002,0x5563CCE3,0x3C2203E2,0x58003800,0xF8000800,
0x00010001,0x00010001,0xFFC1FFC1,0xFFC1FFC1,0xF000F000,0xF000F000
};

int icn_w = 48, /* Width of icon. */
icn_h = 18; /* Height-1 of icon. */

/* Main program. */
main ()
{
    appl_init (); /* Initialize application. */
    open_vwork (); /* Set up workstation. */
    graf_mouse (OFF,&dum); /* Shut off mouse. */
    v_clrwk (handle); /* Clear the screen. */
    graf_mouse (ON,&dum); /* Bring the critter back. */
    do_icon (); /* Go draw icon. */
    v_clsvwk (handle); /* Close virtual workstation. */
    appl_exit (); /* Back to desktop. */
}

/* Initialize a virtual workstation. */
open_vwork ()
{
    int i;

    handle = graf_handle (&dum,&dum,&dum,&dum); /* Get handle. */
    for (i=0; i<10; work_in[i++] = 1); /* Init GEM arrays. */
    work_in[10] = 2;
    v_opnvwk (work_in, &handle, work_out); /* Open v. workstation. */
}

/* Main program loop */
do_icon ()
{
    int button, /* Mouse button pressed. */
        x, /* Mouse X coordinate. */
        y, /* Mouse Y coordinate. */
        ox, /* Old Mouse X coordinate. */
        oy, /* Old Mouse Y coordinate. */
        repeat; /* Loop flag. */

    graf_mouse (HAND,&dum); /* Switch mouse forms. */
    x = 50 ; y = 50; /* Init loc. of icon */
    repeat = TRUE; /* Get into WHILE loop. */
    while (repeat) { /* Begin WHILE loop. */
        x -= 30; y -= 20; /* Adjust mouse coords */
        ox = x; oy = y; /* Save old coords. */
        graf_mouse (OFF,&dum); /* Turn off mouse. */
        draw_icon (icon,S_XOR_D,icn_w, /* Go draw icon. */
            icn_h,x,y,x+icn_w,y+icn_h);
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
graf_mouse (ON,&dum);          /* Turn on mouse. */
button = 0;                    /* Get into WHILE loop. */
while (button == 0)           /* Begin WHILE loop. */
    vq_mouse (handle,&button,&x,&y); /* Get mouse status. */
if (button == LEFT) {        /* If left button pushed... */
    graf_mouse (OFF,&dum);     /* Turn off mouse. */
    draw_icon (icon,S_XOR_D,icn_w, /* Erase old icon. */
               icn_h,ox,oy,ox+icn_w,oy+icn_h);
    graf_mouse (ON,&dum);      /* Turn mouse back on. */
}
if (button == RIGHT)         /* If right button pushed...*/
    repeat = FALSE;          /* get out of loop. */
}

/* Perform raster operation. */
draw_icon (data,mode,width,height,dx1,dy1,dx2,dy2)
long data[];
int mode, /* Raster writing mode. */
width, /* Icon width. */
height, /* Icon height. */
dx1, /* Upper left X coordinate of dest'n rectangle. */
dy1, /* Upper left Y coordinate of dest'n rectangle. */
dx2, /* Lower right X coord of dest'n rectangle. */
dy2; /* Lower right Y coord of dest'n rectangle. */
{
    MFDB s_m, /* Form definition block for source. */
          scr_m; /* Form definition block for screen. */
    int pxy[8]; /* Coords for source and dest'n rectangles. */

    s_m.f_addr = (long) data; /* Put addr of icon data in MFDB */
    s_m.f_w = width; /* Store width of icon in MFDB */
    s_m.f_h = height; /* Store height of icon in MFDB */
    s_m.f_wdwidth = width/16; /* Store icon width/16 in MFDB */
    s_m.f_stand = 0; /* Raster coordinates. */
    s_m.f_nplanes = 4; /* Low res (4 bit planes). */
    s_m.f_r1 = s_m.f_r2 = s_m.f_r3 = 0; /* Zero reserved words. */
    scr_m.f_addr = 0; /* Set up screen MFDB. */
    pxy[0] = 0; /* Upper left X coord of source block. */
    pxy[1] = 0; /* Upper left Y coord of source block. */
    pxy[2] = width; /* Lower right X coord of source block. */
    pxy[3] = height; /* Lower right Y coord of source block. */
    pxy[4] = dx1; /* Upper left X coord of dest'n block. */
    pxy[5] = dy1; /* Upper left Y coord of dest'n block. */
    pxy[6] = dx2; /* Lower right X coord of dest'n block. */
    pxy[7] = dy2; /* Lower right Y coord of dest'n block. */
    vro_cpyfm(handle,mode,pxy,&s_m,&scr_m); /* Do the raster op. */
}
```

### CHAPTER 14 - OBJECT TREES AND DIALOG BOXES

Now that we know how to handle the two simplest of GEM's forms, the alert box and the file selector, it's time to move on to the granddaddy of them all: the dialog box. Because the dialog box is so versatile, we could discuss its uses endlessly and still not exhaust its possibilities. For that reason, this chapter's discussion should not be considered as a complete guide to dialog boxes, but only as an introduction. Once you understand the way dialogs work, the only limit will be your imagination.

#### The Definitions

Before we get into a detailed discussion of dialog boxes, we must first define a couple of terms: objects and trees. Objects are used to visually represent each item that makes up a dialog box. You've seen them hundreds of times: boxes and buttons and text strings. Each object has its own set of attributes that tailor it to the programmer's (and eventually, the user's) needs.

From a programming point of view, an object is a data structure, the members of which describe the object, storing all the necessary information to bring that object up on the screen.

The objects of a dialog box are connected in an object tree. A tree is a way to link items in a hierarchical manner. That is, there's one main item (the tree's root), which has connected to it other items (which, relative to the tree's root are called children, and relative to each other are called siblings). The children may also have children of their own (and thus become parents), and so on down the line, each new group of siblings subordinate to the ones that have gone before.

An object tree is an array of objects, the attributes of which are stored in the previously mentioned data structure. Three elements of an object's data structure determine the way the object fits in with the rest of the tree. Specifically, each object contains, among other things, a pointer to the next sibling, a pointer to the first child (the head) and a pointer to the last child (the tail).

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

Figure 1 illustrates the principles of this type of tree structure, using a simple dialog box as an example. As you can see, even a simple dialog box has quite a maze of connections. Because of this complexity, few programmers bother to try and design dialog boxes from scratch. They instead use the resource construction program that came with their compiler.

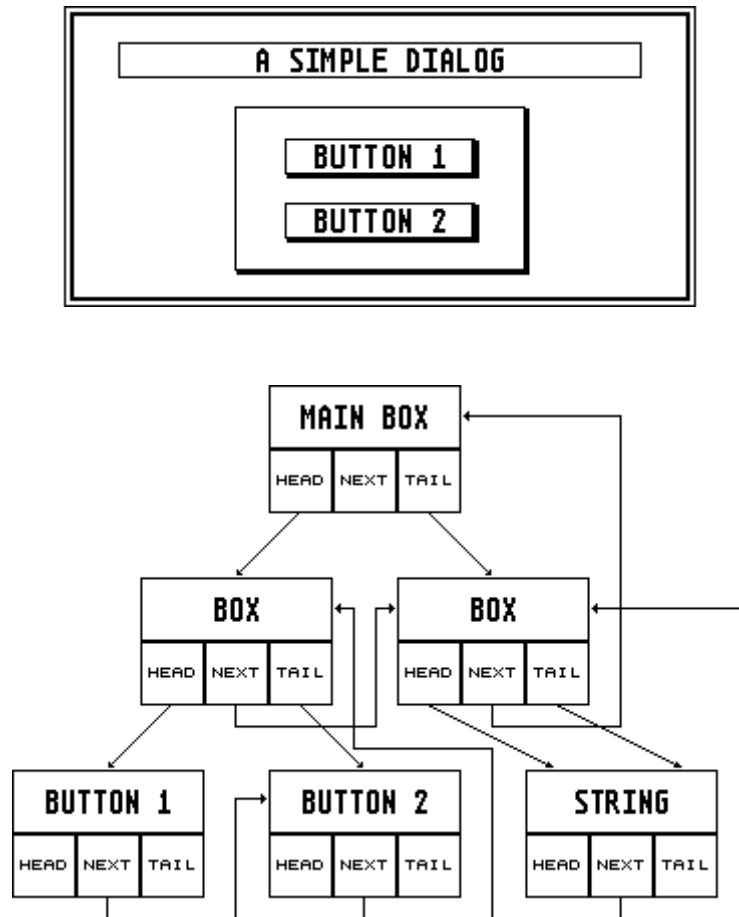


FIGURE 1 - Tree Structure of a Dialog Box

### RCP: A Mini Tutorial

Currently, the two most popular resource construction programs are the ones included with the Atari Developer's Kit and the Megamax C compiler. Since the programs in this column are developed with Megamax, we'll use the Megamax Resource Construction Program (RCP) to build our dialog box. If you're using the Atari Developer's Kit, don't fret; the Resource Construction Set (RCS) that came with your kit will work equally well for our purposes. The only difference is the operation of the programs.

So, everybody load up their resource construction programs, and let's get busy. Figure 2 is the dialog box we'll be building. You should refer to this illustration as you construct your version.



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

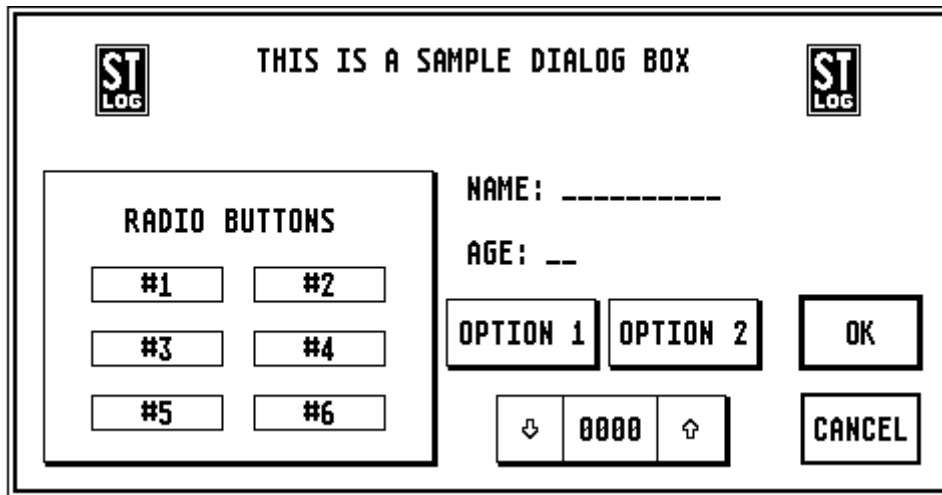


FIGURE 2

Once you have your resource construction program loaded, go to the File option of the menu bar and select New. A window titled NONAME will appear. To the left of this window are the types of resources we can build, represented in icon form. Place the mouse pointer over the dialog icon, press and hold down the left button, and drag the icon into the window. Release the button, and a dialog box will appear, asking you for the name of the new tree. Clear the NAME field by pressing Escape. Then type SAMPLE and press Return.

Now double click the new dialog icon (the one you dragged to the window). This will open the dialog, presenting you with a "blank slate." The icons to the left will change to a dialog box "parts kit." The parts shown are icon representations of the types of objects you can use to build your dialog box. The types are as follows:

Button	A box containing centered text
String	A line of text
FText	Formatted text
FBoxText	A box containing formatted text
IBox	An invisible graphic box
Box	A graphic box
Text	Graphic text
BoxChar	A graphic box enclosing a single character
BoxText	A graphic box enclosing text
Icon	Description of an icon

Now let's start filling out our dialog box with objects.

### Crankin' with the RCP

Step 1: Drag the STRING object onto your dialog box, and then double-click it. A dialog box containing a number of attributes will appear. At the bottom will be a line labeled TEXT. Clear the line by pressing the escape key. Then enter (without the quotes) "THIS IS A

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

SAMPLE DIALOG BOX," and press Return. Drag your new string to the top of the dialog box and center it, as shown in Figure 2.

- Step 2: Drag an ICON object onto your dialog box, and double-click it. Click the EDIT ICON button from the dialog box that appears, and draw the ST-Log icon (or any icon you like) with your mouse. When complete, click the OK button. Drag the icon to the left of the text created in Step 1 and position it as shown in Figure 2.
- Step 3: If the icon is not shown in inverse (selected), give it a single mouse click. When the icon is selected, type Control-C (copy), point the mouse to the right-hand side of your dialog box, and type Control-V (paste) twice (the first keystroke deselected the original icon). You should now have a duplicate of the first icon. Drag it into position, to the right of the string, as shown in Figure 2.
- Step 4: Drag the BOX object (the empty rectangle) onto your dialog box. Place the mouse cursor on the lower-right corner of the box and, holding down the left button, stretch the box until it's about the same size as the box labeled RADIO BUTTONS in Figure 2. Position the box, and double-click it. An attribute dialog box will appear. Use the mouse to select the SHADOWED attribute. Then click the OK button.
- Step 5: Using the same method as in Step 1, create a string that reads "RADIO BUTTONS," and position it at the top of the box created in Step 4.
- Step 6: Drag a BUTTON object into the box created in Step 4, and double-click it. When the attribute dialog appears, select the following options: SELECTABLE, RADIO BUTN and TOUCHEXIT. (Note that sometimes an attribute -- in this case, SELECTABLE -- has already been activated for you.) Modify the text field to read "#1." Then click the OK button. While the radio button is still highlighted (shown in inverse), type Control-N, and name the object "RADIO1." Position this button in the upper left of the "Radio Button" box, beneath the string, as shown in Figure 2.
- Step 7: Use the copy and paste functions (as in Step 2) to place another radio button to the right of the one created in Step 6. Change the button's text to read "#2" and change the object's name to "RADIO2."
- Step 8: Using the method in Step 7, create four buttons labeled "#3," "#4," "#5," and "#6," and name them "RADIO3," "RADIO4," "RADIO5," and "RADIO6," respectively. See Figure 2 for placement.
- Step 9: Drag the EDIT:\_\_\_\_\_ (not the one surrounded by a box) object into your dialog box, and double click it. If it isn't already selected, turn on the EDITABLE option. Clear the PTMPLT field with the escape key, and then type "NAME:" followed by one space and 10 underline characters. Use the down arrow on your keyboard to move the text cursor to the PVALID field. Then press Escape to clear the field. Type "aaaaaaaaa." Use the down arrow key to move the text cursor to the PTEXT field. Then backspace till you reach a tilde (~) character. Now type "@" followed by nine spaces. Click the OK button. Then name the object "NAME." Position the object as shown in Figure 2.
- Step 10: Drag a second EDIT:\_\_\_\_\_ object into your dialog box, and double-click it. Make sure the EDITABLE option is set. Change the PTMPLT field to "AGE:" followed by one space and two

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

underlines. Change the PVALID field to "99." Move to the PTEXT field and backspace until you reach a tilde character. Then type "@" followed by one space. Click the OK button, name the object "AGE," and then position it as shown in Figure 2.

Step 11: Drag another BUTTON object into your dialog box, and double-click it. Select the attributes SELECTABLE, SHADOWED and TOUCHEXIT. Change the button's text to "OPTION 1." Click the OK button, name the object "OPTION1," and position it as shown in Figure 2.

Step 12: Use the copy and paste functions to create a duplicate of the button created in Step 11. Change the button's text to "OPTION 2," name the object "OPTION2," and position it as shown in Figure 2.

Step 13: Drag the BOXTEXT object into your dialog box, and double-click it. Select the SHADOWED attribute. Clear the PTMPLT and PVALID fields (if necessary). Then change the PTEXT field to four spaces followed by four 0s and four more spaces. Using the method shown in Step 4, stretch the box one segment higher (as you pull down on the mouse, the box will automatically "snap" to the next size). Name the object "NUMBERS," and position it as shown in Figure 2.

Step 14: Drag another BOXTEXT icon onto your dialog box, and double-click it. Set the TOUCHEXIT option. Make sure the PTMPLT, PVALID and PTEXT fields are clear. Then, when positioned on the PTEXT field, hit space, Control-A, space (the keys, not the words). Resize the object as in Step 13, and name it "UPARROW." Position it on top of the box created in Step 13 as shown in Figure 2.

Step 15: Use the copy and paste functions to create a duplicate of the UPARROW object. Then clear the PTEXT field and press space, Control-B, space. Name the object "DWNARROW." Then position it on top of the NUMBERS object as shown in Figure 2.

Step 16: Drag another BUTTON object into your dialog box, and double-click it. Set the SELECTABLE, DEFAULT and TOUCHEXIT options. Change the text field to "OK." Resize and position the object as shown in Figure 2 and name it "OK."

Step 17: Drag yet another button into your dialog box, and double-click it. Set the SELECTABLE and TOUCHEXIT options. Then change the TEXT field to "CANCEL." Resize and position the object as shown in Figure 2. Then name it "CANCEL."

And that's it. You've just created your first dialog box. Now, to save all your hard work, close the dialog box by clicking on the upper-left corner of the window. Then select the SAVE AS option from the FILE menu. Name the file "SAMPLE," and you're on your way. To leave the RCP, select the Quit option from the File menu.

You should now have three files on your disk: SAMPLE.H, SAMPLE.DEF and SAMPLE.RSC. These are the files that the RCP created. SAMPLE.H contains all your object and tree names as a series of #defines. If you want to refer to the objects by name in your program, you must #include this file in your source code.

The SAMPLE.DEF file contains information the RCP uses for its own purposes, and the SAMPLE.RSC file is the tree data for our dialog box. We'll load this data into memory when we run our program.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

### So How About Some Details?

That was a fast course in the use of a resource construction program. You probably have many unanswered questions. For example, what do all those attributes do?

SELECTABLE simply means that the user can select the object. When the object is selected, it will be displayed in inverse video. If you set the DEFAULT option when editing an object, the object will be selectable with the Return key, as well as with a mouse click. Obviously, only one object at a time can be set as a default.

The EXIT and TOUCHEXIT attributes are similar: they both cause the dialog box to be exited when selected. The difference is that, with TOUCHEXIT, the mouse button need not be released to exit the dialog box.

What did you think about the RADIO BUTN option? Radio buttons are handy devices, allowing the programmer to set up a series of related buttons, only one of which may be selected at a time. As soon as a button is selected, the previously selected button is turned off. They get their name from those old car radio tuners with the push buttons to select the channel. An important note: In order for radio buttons to operate properly, they must have the same parent object; that is, they all must be "on top of" the same object.

The CHECKED, SHADOWED, OUTLINED, CROSSED and DISABLED options affect the way the objects will be graphically represented on the screen. You can easily see their effect by using your RCP to set them for various objects. The options' names describe their effect fairly accurately.

An EDITABLE object may be modified in some manner by the user.

### Editable Text

Now, what's the story behind those strange text fields PTMPLT, PVALID, and PTEXT? These three strings combine in such a way as to tell GEM which part of the text is editable and what characters the user is allowed to input.

PTMPLT is used as an input mask. Any text entered here will be displayed on the screen and will be unchangeable (except underline characters) by the user. PTMPLT also tells GEM where the user can edit the text. We indicate this with underline characters. In Step 9 above, the unalterable text is "NAME:" and the editable area, where the user will enter his or her name, is represented by the 10 underlines.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The PVALID field tells GEM what type of characters we want the input restricted to. Each underline character in the PTMPLT field must have an entry in the PVALID field as follows:

<u>Code</u>	<u>Characters Allowed</u>
9	0 to 9
A	A to Z, space
a	A to Z, a to z, space
N	0 to 9, A to Z, space
n	0 to 9, A to Z, a to z, space
F	DOS filename characters, plus ? * :
P	DOS filename characters, plus \ ? * :
p	DOS filename characters, plus \ :
X	Any character

In Step 9 we entered 10 lowercase A's in the PVALID field, limiting the user's input to upper- and lowercase letters. A logical choice for a person's name.

Finally, the PTEXT string will be combined with PTMPLT when the latter is printed. Unlike the text in PTMPLT, the string stored in PTEXT is editable. This is handy when you want an editable text field displayed with a default setting. For example, in Step 9, if we had made the PTEXT string "FRED," when the dialog box appeared on the screen, the text cursor would appear to the right of the string "FRED." We could then just leave the string as it is, and thus select FRED as our name, or we could backspace over it (or use the escape key to clear it) and type in something new. When the user exits the dialog box, the new information will be found in PTEXT, replacing what we had stored there previously.

When we set up our editable text objects in Steps 9 and 10, however, we wanted to end up with the text cursor to the left of a blank field, ready for the user's input. To do this, we must either enter an "@" or a null as the first character of the PTEXT string. To reserve space for any text the user may enter, we must fill the rest of the PTEXT string with blanks (actually, any character will work; once GEM sees the "@" or null, it'll ignore the rest of the string and go on its merry way).

### Your First Dialog Box

When you run this chapter's program (make sure the SAMPLE.RSC file is on the disk!), you'll be presented with the dialog box you created with the RCP. Clicking on the OK or CANCEL buttons will exit you from the dialog. Clicking on the up or down arrows will cause the value displayed in the NUMBERS object to change. Clicking any of the other buttons will cause the name of the object selected to be printed at the top of the screen. Notice that, with the radio buttons, only one may be selected at a time, while the OPTION 1 and OPTION 2 buttons can be on or off in any combination.

You may enter your name and age (lie if you want to) in the text fields. Use the arrow keys on your keyboard to move between the two fields (or click on them with the mouse), since, due to the OK button being set up as a default, pressing Return will exit the dialog box. Try to enter something

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

other than upper- or lowercase letters in the name field, or something other than a number in the age field. No dice, right?

When you exit the dialog box, the name and age fields -- as well as the final value of the NUMBERS object -- will be printed to the screen. Notice that whatever is in the PTEXT field is what gets printed. If you left the name and age fields blank, you'll see exactly what we put there to start off with, a line of spaces preceded by the "@" character.

After exiting the dialog box, click the left mouse button to return to the desktop.

### Taking It Apart

Now that we've created our dialog box and played with it a little, it's time to dig into the program a bit.

The first things we should look at are the two structures, object and text\_edinfo found near the top of the listing. I said earlier that an object, from the program's point of view, was a data structure containing the object's description. The data is organized within a C structure as follows:

```
typedef struct object {
    int ob_next;
    int ob_head;
    int ob_tail;
    unsigned int ob_type;
    unsigned int ob_flags;
    unsigned int ob_state;
    char *ob_spec;
    int ob_x;
    int ob_y;
    int ob_w;
    int ob_h;
} OBJECT;
```

Here, ob\_next is the index of the object's next sibling, ob\_head is the index of the object's first child, and ob\_tail is the index of the object's last child. (Remember that the objects are stored in an array of structures. The indices mentioned above are the location of the object within the array.)

The member ob\_type is the object type and will contain one of the following values:

<u>Object Type</u>	<u>Value</u>	<u>Object Type</u>	<u>Value</u>
Box	20	BoxChar	27
Text	21	String	28
BoxText	22	FText	29
Image	23	FBoxText	30
ProgDef	24	Icon	31
IBox	25	Title	32
Button	26		

The member ob\_flags contains the object flags and will be one of the values shown below:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

NONE	0x0000
SELECTABLE	0x0001
DEFAULT	0x0002
EXIT	0x0004
EDITABLE	0x0008
RBUTTON	0x0010
LASTOB	0x0020
TOUCHEXIT	0x0040
HIDETREE	0x0080
INDIRECT	0x0100

You should recognize most of these from your work with the RCP.

The member `ob_state` holds the current state of the object as follows:

NORMAL	0x0000
SELECTED	0x0001
CROSSED	0x0002
CHECKED	0x0004
DISABLED	0x0008
OUTLINED	0x0010
SHADOWED	0x0020

You've seen most of these before, right?

The member `ob_spec` contains object specific information and changes depending on the type of object that's being described. The possible values of this field are as below:

Object Type	Contents of <code>ob_spec</code>
Box	Object's color and thickness
Text	Pointer to TEDINFO structure
BoxText	Pointer to TEDINFO structure
Image	Pointer to BITBLK structure
IBox	Border's color and thickness
Button	Pointer to text string
BoxChar	Object's color and thickness, and the character to display
String	Pointer to text string
FText	Pointer to TEDINFO structure
FBoxText	Pointer to TEDINFO structure
Icon	Pointer to ICONBLK structure
Title	Pointer to text string

Notice that the value stored in `ob_spec` can be a pointer to another structure containing additional information on the object. We'll take a look at one of the structures, TEDINFO, shortly.

Finally, `ob_x`, `ob_y`, `ob_w`, and `ob_h` contain the object's coordinates, width and height.

## The Mysterious TEDINFO

The second structure type in the sample program, `text_edinfo`, is the declaration for the previously mentioned TEDINFO. Whenever our object has an editable text field, we need to store information about it in a TEDINFO structure (actually, when using an RCP, we don't have to worry about storing information in the structure; it's done for us) as follows:

```
typedef struct text_edinfo {
    char *te_ptext;
    char *te_ptmplt;
    char *te_pvalid;
    int te_font;
    int te_junk1;
    int te_just;
    int te_color;
    int te_junk2;
    int te_thickness;
    int te_txtlen;
    int te_tmplen;
} TEDINFO;
```

Here, `te_ptext` is a pointer to the PTEXT string, `te_ptmplt` is a pointer to the PTMPLT string, `te_pvalid` is a pointer to the PVALID string, `te_font` is the text font (3 = system font; 5 = small font), `te_just` is the justification (0 = left; 1 = right; 2 = centered), `te_color` is the color and pattern type, `te_thickness` is the thickness in pixels of the border (0 = no border; 1 to 128 = thickness inward from the edge; -1 to -127 = thickness outward from the edge), `te_txtlen` contains the length of the string pointed to by `te_ptext`, and `te_tmplen` contains the length of the string pointed to by `te_ptmplt`.

## As the Fear Sets In

Relax. In most cases, when using the RCP to put together your dialog box, you won't have to worry about the contents of the above structures. But, in case you want to do something more sophisticated, you do need to understand where to find information about your dialog box. An example of this is the NUMBERS object in the sample dialog. In order to get the up and down arrows to change the value shown, we have to be able to get at the displayed strings. This is just one example of the creative ways you can use a dialog box.

## Breathing Time

Your head is probably spinning from all this technical talk. We'll take a break here and give you a chance to ponder what you've learned. Spend some time with your resource construction program, experimenting with different attribute settings on different objects. As you continue with your career as a GEM programmer, the RCP is going to become one of your most valued tools.



## Program Listing #1

```

/*****
/*          C-manship, Listing 1          */
/*          CHAPTER 14                    */
/*          Developed with Megamax C      */
*****/

#include "SAMPLE.H"
#include <OSBIND.H>

#define FMD_START 0
#define FMD_GROW 1
#define FMD_SHRINK 2
#define FMD_FINISH 3
#define R_TREE 0
#define FINGER 3

/* The usual required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum;

int dial_x, /* Dialog x coordinate. */
dial_y, /* Dialog y coordinate. */
dial_w, /* Dialog width. */
dial_h, /* Dialog height. */
num, /* Value of number option. */
n_x, /* NUMBERS object x coord. */
n_y; /* NUMBERS object y coord. */

char number_str[13] = " 0000 "; /* NUMBERS string. */

char *find_str(); /* Function declaration. */

/* Structure to hold an object's description. */
typedef struct object
{
    int ob_next; /* Next sibling of object. */
    int ob_head; /* Head of object's children. */
    int ob_tail; /* Tail of object's children. */
    unsigned int ob_type; /* Type of object. */
    unsigned int ob_flags; /* Flags. */
    unsigned int ob_state; /* State of object. */
    char *ob_spec; /* Miscellaneous information. */
    int ob_x; /* x pos of object upper left */
    int ob_y; /* y pos of object upper left */
    int ob_w; /* Width of object. */
    int ob_h; /* Height of object. */
} OBJECT;
OBJECT *tree_addr; /* Pointer to our object structure. */

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Structure to hold object text information. */
typedef struct text_edinfo
{
    char *te_ptext;      /* Pointer to text.          */
    char *te_ptmplt;     /* Pointer to template.     */
    char *te_pvalid;     /* Pointer to validation chars. */
    int te_font;         /* Font.                    */
    int te_junk1;        /* Unused.                  */
    int te_just;         /* Justification.           */
    int te_color;        /* Color information.       */
    int te_junk2;        /* Unused.                  */
    int te_thickness;    /* Border thickness.        */
    int te_txtlen;       /* length of text string.   */
    int te_tmplen;       /* length of template string. */
} TEDINFO;

main ()
{
    appl_init ();        /* Initialize application.   */
    open_vwork ();       /* Set up workstation.      */
    do_dialog();         /* Go do the dialog box.    */
    button_wait();       /* Wait for mouse button.   */
    rsrc_free ();        /* Release resource memory. */
    v_clsvwk (handle);   /* Close virtual workstation. */
    appl_exit ();        /* Back to the desktop.     */
}

open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open */
    /* a virtual workstation.                                   */

    handle = graf_handle (&dum,&dum,&dum,&dum);
    for (i=0; i<10; work_in[i++] = 1);
    work_in[10] = 2;
    v_opnvwk (work_in, &handle, work_out);
}

do_dialog ()
{
    int choice; /* Button choice from dialog. */

    /* Here we load the resource file. If the file is missing, */
    /* we warn the user with an alert box then terminate the */
    /* program by skipping the code following the else.       */

    if (! rsrc_load ("\SAMPLE.RSC"))
        form_alert (1, "[1][SAMPLE.RSC missing!][I'll do better!]");

    /* If the resource file loads OK, we get the address of the */
    /* tree, get the coords for centering the dialog, save the */
    /* portion of the screen that'll be covered by the dialog, */
    /* and draw the dialog. The mouse pointer is changed to */
    /* pointing finger.                                       */

    else {
        rsrc_gaddr (R_TREE, SAMPLE, &tree_addr);
        form_center (tree_addr, &dial_x, &dial_y, &dial_w, &dial_h);
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
objc_offset (tree_addr, NUMBERS, &n_x, &n_y);
form_dial(FMD_START, 0, 0, 10, 10,
          dial_x, dial_y, dial_w, dial_h);
form_dial(FMD_GROW, 0, 0, 10, 10,
          dial_x, dial_y, dial_w, dial_h);
objc_draw(tree_addr, 0, 2, dial_x, dial_y, dial_w, dial_h);
graf_mouse (FINGER,&dum);

/* Here we allow the user to interact with the dialog then,*/
/* based on the chosen button, perform the necessary action.*/
/* The form_do function is repeated until the user chooses */
/* either the OK button or the CANCEL button.                */
num = 0;
do {
    choice = form_do (tree_addr, NAME);
    if (choice == RADIO1) v_gtext(handle,160,20,"Radio 1 ");
    if (choice == RADIO2) v_gtext(handle,160,20,"Radio 2 ");
    if (choice == RADIO3) v_gtext(handle,160,20,"Radio 3 ");
    if (choice == RADIO4) v_gtext(handle,160,20,"Radio 4 ");
    if (choice == RADIO5) v_gtext(handle,160,20,"Radio 5 ");
    if (choice == RADIO6) v_gtext(handle,160,20,"Radio 6 ");
    if (choice == OPTION1) v_gtext(handle,160,20,"Option 1");
    if (choice == OPTION2) v_gtext(handle,160,20,"Option 2");
    if (choice == UPARROW) do_up();
    if (choice == DWNARROW) do_down();
}
while (choice != CANCEL && choice != OK);

/* Once the CANCEL or OK buttons have been pressed, we clean */
/* up after ourselves by performing the "shrinking box" and */
/* then redrawing the screen.                                   */

form_dial(FMD_SHRINK, 0, 0, 10, 10,
          dial_x, dial_y, dial_w, dial_h);
form_dial(FMD_FINISH, 0, 0, 10, 10,
          dial_x, dial_y, dial_w, dial_h);
print_results(tree_addr); /* Print user's choices.          */
}

do_up ()
{
    /* First we increment our value and make sure it stays in */
    /* range. If the value has become larger than 9999, we must */
    /* also reinitialize display string for the object NUMBERS. */
    /* We then call our function to update the NUMBERS object. */

    num += 1;
    if (num > 9999) {
        num = 0;
        strcpy (number_str,"    0000    ");
    }
    edit_object ();
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_down ()
{
    /* Here we decrement the value and check for its range, */
    /* after which we update the NUMBERS object. */

    num -= 1;
    if (num < 0) num = 9999;
    edit_object ();
}

edit_object ()
{
    TEDINFO *ob_tedinfo;
    char temp_str[10];

    /* Here we edit the string we're using for the text display */
    /* in the object NUMBERS so that it reflects the new value. */

    sprintf (temp_str, "%d", num);
    strcpy (&number_str[8 - strlen (temp_str)], temp_str);
    strcpy (&number_str[8], "    ");

    /* Then we find the object NUMBERS' TEDINFO and point the */
    /* te_ptext member to our updated string, after which we */
    /* redraw the object NUMBERS. */

    ob_tedinfo = (TEDINFO *) tree_addr[NUMBERS].ob_spec;
    ob_tedinfo -> te_ptext = number_str;

    /* For high resolution, change the 16 below to 32. */
    objc_draw (tree_addr, NUMBERS, 1, n_x, n_y, 96, 16);
}

print_results (tree_addr)
OBJECT tree_addr[];
{
    char *string;

    /* Here we call the function that locates the string, then */
    /* print the user's input to the screen. */

    string = find_str (NAME, string);
    v_gtext (handle, 160, 20, "Your name is ");
    v_gtext (handle, 264, 20, string);
    string = find_str (AGE, string);
    v_gtext (handle, 160, 36, "Your age is ");
    v_gtext (handle, 264, 36, string);
    string = find_str (NUMBERS, string);
    v_gtext (handle, 160, 52, "Final number value: ");
    v_gtext (handle, 320, 52, &string[4]);
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
char *find_str (object, string)
int object;
char *string;
{
    TEDINFO *ob_tedinfo;

    /* In this function, we locate the object's TEDINFO */
    /* structure then set our string pointer to the pointer */
    /* found in the te_ptext member. */

    ob_tedinfo = (TEDINFO *) tree_addr[object].ob_spec;
    string = ob_tedinfo -> te_ptext;
    return (string);
}

/* Waits for left button to be pressed and released. */
button_wait()
{
    evnt_button (1,1,1,&dum,&dum,&dum,&dum);
    evnt_button (1,1,0,&dum,&dum,&dum,&dum);
}
```

### CHAPTER 15 - MORE ON DIALOG BOXES

In chapter 14 we looked at the process of creating a dialog box. In this chapter, we'll tie up some loose ends by taking a close look at Chapter 14's sample program. You'll want to refer to that listing during the following discussion.

#### The Workings

At the top of the listing, we include the file `SAMPLE.H`. This file contains the name of the object tree that represents our dialog box, as well as the names of all the objects within the tree. Each object is given a number. This number is the index used to find the object within the array. (A tree is an array of objects, remember?) For those of you who don't have a resource construction program (RCP), Listing 1 at the end of this chapter shows what the `SAMPLE.H` file contains.

If you do have an RCP, and you put together Chapter 14's dialog box, you may find that your objects are numbered differently than those in the example. Don't worry about it. That just means we constructed our dialog boxes a little differently. For instance, the example's object numbers don't run in perfect order. Some numbers are missing because, in the course of constructing the dialog, I removed a couple of objects from the screen without actually deleting them from the tree. Those objects were assigned numbers, but since they remain unnamed (and unused), they don't appear in the `.H` file.

The `#defines` in Chapter 14's listing assign to logical names the values of various parameters we'll be using when handling the dialog. Following that, we have the required GEM array declarations and some global variables.

The character array `number_str[]` is the string we'll use to change the displayed value of the `NUMBERS` object, in response to one of the arrows being clicked.

Finally, we get to the two structure types we discussed in Chapter 14, `OBJECT` and `TEDINFO`. Of course, these are not standard C data types, right? These are our own data types, and we told the compiler this by using the `typedef` keyword in front of the declaration.

We also declare a pointer to our object structure. This pointer does not, at this time, contain an address. We've told the compiler only that when there is an address in `*tree_addr`, it'll be the address of a block of data of the type `OBJECT`. Also, we don't have an `OBJECT` or `TEDINFO` structure in memory yet. In our source code, we've only described what they'll look like. If you're confused, take a little time to review structures and their declarations.

#### And Speaking of the Program...

Finally, we get to `main()`. There's not much to discuss here. In keeping with structured style, there's only a general "outline" of the program contained in this function. As you can see, our program must execute seven main steps. We've left the details of those steps to other functions. The function `do_dialog()` is where the action really begins.

The first thing we must do to get the dialog box on the screen is load into memory the resource file containing all the dialog's information. We do this with the call:

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
rsrc_load("filename.rsc");
```

Here, filename.rsc is the name of the resource file that was created by the RCP (note that the extension .RSC is a convention you should stick to when naming resource files). This call returns a 0 if an error was encountered and a nonzero number if the file loaded okay.

Simple, no? In our case, we've used an if statement to prevent the program from continuing if the .RSC file is missing. If a 0 is returned, the conditional becomes true, in which case we call up an alert box, informing the user that the SAMPLE.RSC file was missing, then skip over the rest of the program and exit to the desktop. Once we've got our resource file loaded, we need to find its address. We do this with the call:

```
rsrc_gaddr(type, index, tree_addr);
```

Here, type is the type of data structure loaded, index is the index of the object within the tree (in our case, the "object" is a tree), and tree\_addr is the address of the information (the tree) loaded from the .RSC file. The parameter type should be a value from the following table:

0	object tree
1	OBJECT
2	TEDINFO
3	ICONBLK
4	BITBLK
5	string
6	image data
7	obspec (object specification)
8	te_ptext (pointer to text)
9	te_ptmplt (pointer text template)
10	te_pvalid (pointer to text validation string)
11	ib_pmask (pointer to icon image mask)
12	ib_pdata (pointer to icon image data)
13	ib_ptext (pointer to icon text)
14	ib_pdata (pointer to bit image)
15	address of a pointer to a free string
16	address of a pointer to a free image

The next thing we must do is modify the coordinates of our dialog box so that it'll appear in the center of the screen. This is done with the call:

```
form_center(tree_addr, &x, &y, &w, &h);
```

Here, tree\_addr is the address of the object tree (returned from rsrc\_gaddr()), and &x, &y, &w, and &h are the addresses of the integer variables that will contain the dialog's centered X,Y-coordinates, width, and height, respectively.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Since we'll be doing some work by hand, as it were, on the NUMBERS object in order to update the number it displays, we must find its eventual position on the screen. We do this with the call:

```
objc_offset (tree_addr, index, &x, &y);
```

Here, `tree_addr` is the address of the tree that contains the object, `index` is the object's index within the tree, and `&x` and `&y` are the addresses of the integer variables that will contain the object's coordinates returned from the function.

Now we must reserve space on screen for the dialog. We have to do this so that, when we remove the dialog from the screen, GEM will be able to restore the display. The call

```
form_dial(flag,s_x,s_y,s_w,s_h,l_x,l_y,l_w,l_h);
```

takes care of this, where `flag` is the operation you wish the function to perform (in this case, it should be 0); `s_x`, `s_y`, `s_w`, and `s_h` are the X,Y coordinates, width, and height of the smallest rectangle (we'll talk about this in a moment); and `l_x`, `l_y`, `l_w`, and `l_h` are the X,Y coordinates, width, and height of the largest rectangle (the actual size of the dialog). The acceptable values for `flag` are:

0	FMD_START	reserves screen space
1	FMD_GROW	draws expanding box
2	FMD_SHRINK	draws shrinking box
3	FMD_FINISH	releases screen space and does a redraw

Next, we call `form_dial()` to perform operation 1, drawing the expanding box. The call looks exactly the same as above, except the value of `flag` is 1. The expanding box is drawn starting with the coordinates and size of the smallest rectangle, and ending with the coordinates and size of the largest rectangle. Note that the drawing of both the expanding and shrinking boxes is optional. If you wish, you can skip over this step and go directly to the call below, which actually draws the dialog. One reason you might want to do this is to bring the dialog up faster.

Finally, we're ready to draw our dialog, with the call:

```
objc_draw(tree_addr, object, depth, x, y, w, h);
```

Here, `tree_addr` is the address of the object tree; `object` is the number of the object to draw; `depth` is how many levels deep the object should be drawn; and `x`, `y`, `w`, and `h` are the X,Y coordinates, width, and height, respectively, of the area of the screen in which the object will actually be drawn, also called the "clipping rectangle."

The clipping rectangle is the portion of the display to which all our screen output is limited. For instance, if we print text that'll extend beyond the rectangle's border, the text will be "clipped" to fit; anything that would be drawn outside the clipping rectangle will be ignored. Thus we can protect the integrity of the rest of the display.



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

When we set object to 0 in the above call, we're asking for the first object in the tree to be drawn. The first object in a tree is the root -- in our case, the box containing the rest of the objects that make up our dialog. To be sure all the objects contained within the dialog are drawn, we must set depth to the proper value. If we had set it to 0, only the main box would have been drawn. If we had set it to 1, only the main box and its children would have been drawn, meaning that our radio button box would be missing its buttons and our number box would be missing its arrows. By setting depth to 2 in the sample program, the main box plus its children and grandchildren (the children of the children) are drawn, thus completing our dialog. If you want to be sure you get everything, just set depth to its maximum value of 8.

Now that our dialog is on the screen, how do we find out what the user is doing with it? Simple!

```
form_do(tree_addr, object);
```

Here, tree\_addr is the address of the object tree, and object is the index of an editable text field (0 if there are no editable text fields). The value in object tells form\_do() the number of the text editable field we want to be active at the time of the call.

Now that we've made our call to form\_do(), GEM will handle the dialog for us, highlighting any selectable fields clicked on and letting us enter text into any editable text field. When an exit button is clicked, the dialog will be terminated and form\_do() will return the number of the button clicked. The button number is the only piece of information it returns directly. If we want to see what was entered in the strings, we have to hunt.

Obviously, only EXIT buttons will ever have their values returned from the form\_do() call, so if you want to know when a button is clicked, make sure, when you design your dialog, that one of the button's attributes is EXIT or TOUCHEXIT.

Actually, that's not entirely true. There is another way to get this information. Each time we click a button or fiddle with the dialog in some other way, the object's status is changed and recorded in the ob\_state member of the OBJECT structure. We'll see how to access the OBJECT structure in a while.

Because the only time we want to close our sample dialog box is when the OK or CANCEL button is clicked, we place the form\_do() call within a do/while loop. In the body of the loop, we check to see which button was clicked, and perform the necessary action. The loop repeats, continually activating and deactivating the dialog, until OK or CANCEL is clicked. Note that the call to form\_do() doesn't redraw the dialog; it only notifies GEM to accept more input from the form.

When the user has finished with the dialog, we must remove it from the screen. We do this by performing two more form\_dial() calls: one to display the shrinking box (flag=2), and one to restore the screen (flag=3).

### Finding the Data

In our sample dialog, when a button is clicked, all the program does is print the object's name. But when the user clicks on one of the arrows, we have to find a way to change the value shown in the NUMBERS object. This is where things get a little sticky. Your knowledge of pointers and structures is about to be pushed to the limit.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Think back to when you created the dialog. The object NUMBERS is a BOXTEXT object -- a graphic box containing a string. It's this string that displays the value of num (the variable that holds the most current value selected by the arrows). When the user clicks on one of the arrows, we must increment or decrement num, then modify the string displayed in the object NUMBERS. In our program, the string we'll be modifying is number\_str. But, until we change the pointer contained in te\_ptext, the dialog doesn't know anything about our string. It's perfectly happy with the string we gave it at the beginning.

So...

In Chapter 14 we discussed the OBJECT and TEDINFO structures. I told you that the OBJECT structure contained a field called ob\_spec that holds object-specific information. When the object being described is of type BOXTEXT, ob\_spec holds a pointer to a TEDINFO structure. (See the ob\_spec chart in Chapter 14.) And guess where we'll find that pointer we want to fiddle with?

Yep.

So, let's say the user clicks on the up arrow. At that point, program execution jumps to the function do\_up(), where num is incremented. The if statement makes sure the displayed value doesn't exceed 9999. If num gets too big, we reset it to 0 and copy a new string reflecting this change into number\_str. We then call edit\_object(), the function that'll force NUMBERS to display the string we want it to, instead of its own.

### Dealing with TEDINFO

This is where things get tricky, so make sure your thinking caps are in working order.

The first thing we do in edit\_object() is to declare a pointer, \*ob\_tedinfo, to a TEDINFO structure. Also, at the beginning of this function, we declare a 10-character string to temporarily hold the text we'll be setting up.

The first three lines of actual code in edit\_object() convert the value in num to string form, then place this new string into number\_str. We have to do all that fancy string handling to make sure the numbers are placed in the proper position, retaining any leading zeros, as well as the four spaces before and after the number.

That was the easy part. Now we have to find the pointer that points to the string contained in the NUMBERS object, so that we can change it to point to our own string. Remember that our tree, which is now pointed to by tree\_addr, is an array of structures, each structure describing one of the objects within the tree. Just like any other array, it lets us get to a particular element by using an index. The object whose structure we wish to locate is NUMBERS, and, thanks to our handy RCP, NUMBERS has been #defined in the SAMPLE.H header file to the value of the index we need.

The ob\_spec member of the structure that describes NUMBERS contains a pointer to the TEDINFO structure that holds the pointer to our string. Yikes! I feel an illustration coming on. Figure 1 ought to help you sort this tangle out.

Got it?

So the address of NUMBERS's TEDINFO structure is:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
tree_addr[NUMBERS].ob_spec
```

This is the address we store in `ob_tedinfo` (after casting it into a pointer to `TEDINFO`).

Now that `ob_tedinfo` points to `NUMBERS`'s `TEDINFO` structure, it's a simple matter to get at the address of the string.

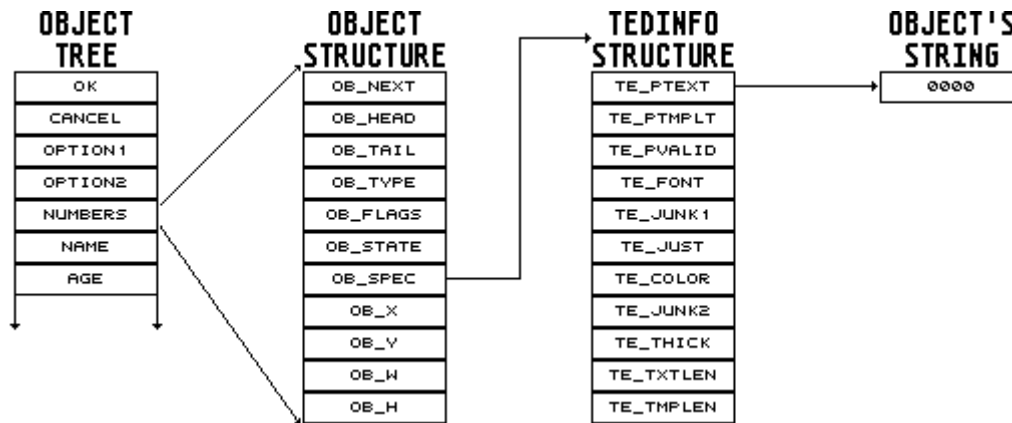


FIGURE 1 - Locating an object's text

The address is contained in the element `te_ptext`, so the statement below changes this pointer to point to our own string.

```
ob_tedinfo->te_ptext = number_str;
```

Once we have the string we created included as part of `NUMBERS`, all we have to do is redraw the object with the call:

```
objc_draw(tree_addr, NUMBERS, depth, x, y, w, h);
```

Here, the parameters are the same as those described above. The height value of 16 used in the program is for medium resolution. To get the dialog to work properly in high resolution, since the monochrome screen has twice the resolution, you must change this value to 32. To make your dialog work automatically in any resolution, you can get the values for `w` and `h` from the `ob_w` and `ob_h` members of the object's `OBJECT` structure.

### Releasing Resource Memory

When we load a resource file, we are obviously using up some of our system's memory. The memory that has been reserved for our resource file should be returned to the system before we exit the program or before we try to load another resource file. We release the memory occupied by a resource file with this function call:

```
rsrc_free();
```

If you look at our sample program from Chapter 14, in `main()`, you'll see that we are releasing the resource before we close the virtual workstation and exit the program.

### Knowing Who Your Friends Are

In Chapter 14 I said that the resource construction program was going to be one of your most valued tools. It saves the programmer an immense amount of time, by generating most of the data needed to bring a dialog box (or any other object tree) up on the screen.

But you don't have to use an RCP to program a dialog. You can do it directly from C -- if you're the type of person who likes outrageously meticulous tasks.

Listing 2 at the end of this chapter will give you some idea of what I'm talking about. This listing was created by the Resource Construction Set that comes with the Atari Developer's Kit. Compare what you see with all you've learned about dialog boxes, especially with respect to OBJECT and TEDINFO structures. You should be able to identify many of the components of our dialog.

For instance, right at the top of the listing, immediately after the #define statements, are all the strings we need, including templates and string validation fields. Below that is the data for the ANALOG "A" icons. Each icon consists of the actual data plus a mask, so we end up with four arrays, two for each icon. A little farther down, you can see the arrays of TEDINFO and OBJECT structures. Each line of these arrays makes up the data for one of the structures. Compare what you see with the structures illustrated in Figure 1.

Listing 2 appears here in exactly the same form as it was output from the RCS. There's still some information that needs to be filled in by the programmer. For instance, look at the first line of the TEDINFO array. The "11L" is the first TEDINFO's string pointer (te\_ptext). Right now, this value is just an offset from the beginning of the strings defined at the top of the listing. If you start at zero and count down to eleven, you'll find that this te\_ptext is associated with the string "@\_\_\_\_\_" (each "\_" represents a space character), which is the blank field for our NAME object.

The array of objects that make up our dialog is found right below the TEDINFOs. Just like the TEDINFO array, each line here consists of the data for each object's structure. Isn't this fun? How'd you like to code all this stuff by hand?

### Closing Up Shop

This finishes up our introduction to dialog boxes. By now, you should have a good idea of how versatile they can be. With a little creativity, you could write an entire program that used nothing but dialog boxes for all its input and output. Don't be afraid to experiment. Get as outrageous as you like!

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```
#define SAMPLE 0
#define OK 4
#define CANCEL 3
#define RADIO2 6
#define RADIO1 8
#define RADIO4 7
#define RADIO3 9
#define RADIO5 10
#define RADIO6 11
#define NAME 17
#define AGE 18
#define OPTION2 14
#define OPTION1 13
#define NUMBERS 19
#define DWNARROW 16
#define UPARROW 15
```

## Program Listing #2

```
#define T0OBJ 0
#define FREEBB 0
#define FREEIMG 4
#define FREESTR 29
```

```
BYTE *rs_strings[] = {
    "",
    "",
    "CANCEL",
    "OK",
    "#2",
    "#4",
    "#1",
    "#3",
    "#5",
    "#6",
    "RADIO BUTTONS",
    "@ ",
    "NAME: _____",
    "aaaaaaaaaa",
    "@ ",
    "AGE: ____",
    "99",
    "OPTION 1",
    "OPTION 2",
    "    0000    ",
    "",
    "",
    "| ",          /* this contains 'SPACE CONTROL-B SPACE' */
    "",
    "",
    "+ ",          /* this contains 'SPACE CONTROL-A SPACE' */
    "",
    "",
    "THIS IS A SAMPLE DIALOG BOX"
};
```

[illegible]

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
WORD IMAG3[] = {
    0x7, 0xFFFF, 0xFFFC, 0xF,
    0xFFFF, 0xFFFE, 0x1F, 0xFFFF,
    0xFFFE, 0x3F, 0xFFFF, 0xFFFE,
    0x7F, 0xFFFF, 0xFFFE, 0xFF,
    0xFFFF, 0xFFFE, 0xFF, 0xF000,
    0xFFE, 0xFF, 0xF000, 0xFFE,
    0xFF, 0xF000, 0xFFE, 0xFF,
    0xFFFF, 0x8FFE, 0xFF, 0xFFFF,
    0xCFFE, 0xFF, 0xFFFF, 0xCFFE,
    0xFF, 0xFFFF, 0x8FFE, 0xFF,
    0xF000, 0xF1E, 0xFF, 0xF800,
    0xE0E, 0xFF, 0xFC00, 0xE0E,
    0xFF, 0xFE00, 0xF1E, 0xFF,
    0xFF00, 0xFFE, 0xFF, 0xFF00,
    0xFFE, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0,
    0x0, 0x0, 0x0, 0x0
};

LONG rs_frstr[] = {0};

BITBLK rs_bitblk[] = {0};

LONG rs_frimg[] = {0};

ICONBLK rs_iconblk[] = {
    0L, 1L, 0L, 4096,0,0, 0,0,48,24, 4,21,40,2,
    2L, 3L, 1L, 4096,0,0, 0,0,48,24, 4,21,40,2
};

TEDINFO rs_tedinfo[] = {
    11L, 12L, 13L, 3, 6, 0, 0x1180, 0x0, 255, 11,17,
    14L, 15L, 16L, 3, 6, 0, 0x1180, 0x0, 255, 3,8,
    19L, 20L, 21L, 3, 6, 2, 0x1180, 0x0, 255, 13,1,
    22L, 23L, 24L, 3, 6, 0, 0x1180, 0x0, 255, 4,1,
    25L, 26L, 27L, 3, 6, 2, 0x1180, 0x0, 255, 4,1
};
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
OBJECT rs_object[] = {
    -1, 1, 20, G_BOX, NONE, OUTLINED, 0x21140L, 0,0, 58,14,
    2, -1, -1, G_ICON, NONE, NORMAL, 0x0L, 4,1, 6,3,
    3, -1, -1, G_ICON, NONE, NORMAL, 0x1L, 47,1, 6,3,
    4, -1, -1, G_BUTTON, 0x41, NORMAL, 0x2L, 48,11, 8,2,
    5, -1, -1, G_BUTTON, 0x43, NORMAL, 0x3L, 48,8, 8,2,
    13, 6, 12, G_BOX, NONE, SHADOWED, 0xFF1100L, 2,4, 24,9,
    7, -1, -1, G_BUTTON, 0x51, NORMAL, 0x4L, 13,3, 8,1,
    8, -1, -1, G_BUTTON, 0x51, NORMAL, 0x5L, 13,5, 8,1,
    9, -1, -1, G_BUTTON, 0x51, NORMAL, 0x6L, 3,3, 8,1,
    10, -1, -1, G_BUTTON, 0x51, NORMAL, 0x7L, 3,5, 8,1,
    11, -1, -1, G_BUTTON, 0x51, NORMAL, 0x8L, 3,7, 8,1,
    12, -1, -1, G_BUTTON, 0x51, NORMAL, 0x9L, 13,7, 8,1,
    5, -1, -1, G_STRING, NONE, NORMAL, 0xA, 5,1, 13,1,
    14, -1, -1, G_FTEXT, EDITABLE, NORMAL, 0x0L, 28,4, 16,1,
    15, -1, -1, G_FTEXT, EDITABLE, NORMAL, 0x1L, 28,6, 7,1,
    16, -1, -1, G_BUTTON, 0x41, SHADOWED, 0x11L, 27,8, 9,2,
    17, -1, -1, G_BUTTON, 0x41, SHADOWED, 0x12L, 37,8, 9,2,
    20, 18, 19, G_BOXTEXT, NONE, SHADOWED, 0x2L, 31,11, 12,2,
    19, -1, -1, G_BOXTEXT, TOUCHEXIT, NORMAL, 0x3L, 0,0, 3,2,
    17, -1, -1, G_BOXTEXT, TOUCHEXIT, NORMAL, 0x4L, 9,0, 3,2,
    0, -1, -1, G_STRING, LASTOB, NORMAL, 0x1CL, 15,1, 29,1
};

LONG rs_trindex[] = {0L};

struct foobar {
    WORDdummy;
    WORD*image;
} rs_imdope[] = {
    0, &IMAG0[0],
    0, &IMAG1[0],
    0, &IMAG2[0],
    0, &IMAG3[0]
};

#define NUM_STRINGS 29
#define NUM_FRSTR 0
#define NUM_IMAGES 4
#define NUM_BB 0
#define NUM_FRIMG 0
#define NUM_IB 2
#define NUM_TI 5
#define NUM_OBS 21
#define NUM_TREE 1

BYTE pname[] = "SAMPLE.RSC";
```



## CHAPTER 16 - MENU BARS

Now that we know everything there is to know about dialog boxes (well, maybe not everything), it's time to move on to menu bars. Second only to windows, menu bars are one of the most characteristic features of GEM. Because they're an excellent way to organize the large number of options complex programs offer the user, virtually every GEM program uses them.

You'll be surprised to hear that menu bars are actually much easier to program than dialog boxes. In fact, they're so easy that we can cover them in a single chapter, rather than the two it took for dialog boxes.

### Another RCP Tutorial

Before we can go any further, you're going to have to load up your Resource Construction Program and create the object tree for the sample menu bar. The following steps will guide you through the entire construction process. It's not as detailed as the instructions I gave in the last RCP tutorial; you should be familiar with using the RCP by now. I can't hold your hands forever -- they get too sweaty.

So get to work, and I'll meet you after Step 24.

### Steppin' Through the Menu Bar

- Step 1: Click on the "New" selection from the File menu. A window titled NONAME will be opened. Just as when we constructed our dialog box a couple of chapters ago, this window is where we'll work on our menu bar.
- Step 2: Drag the menu icon from the left of the screen into the newly created window. A dialog box will appear, prompting you for the name of the menu tree. Press Return to select the default name of TREE00. The menu tree icon will appear in the work window.
- Step 3: Double-click the menu tree icon. The beginnings of your menu bar will appear in the work window.
- Step 4: Give the Desk menu selection (on your menu bar, not the RCP's) a single click, and then press Control-N. The dialog box for naming objects will appear. Name this object "DESK."
- Step 5: Repeat Step 4 for the File menu selection, naming this object "FILE."
- Step 6: Drag the word TITLE from the parts list and place it to the right of the File title. Double-click this new menu bar title. A dialog box will appear. Change the text to two spaces followed by the word "Options," followed by another two spaces.
- Step 7: Place the mouse pointer on the lower right-hand corner of the title's shaded area and, holding down the left button, expand the shading to the right, centering the title within it. Click once on the options title to select it. Then press Control-N and name the object "OPTIONS."
- Step 8: Set up another menu title in the same way, entering the text as two spaces followed by the word "Selections," followed by two more spaces. Name the object "SELECTS."

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

- Step 9: Give the Desk menu selection a single click. Then double-click the "Your message here" entry. Change the text to two spaces followed by "C-manship info..." and press Return. Press Control-N, and name the entry "INFO."
- Step 10: Give the File menu selection a single click. Then place the mouse pointer on the lower-right corner of the QUIT object. Holding down the left button, reduce the length of the object by dragging the corner to the left. You have to do this in order to uncover the menu box beneath.
- Step 11: Place the mouse pointer on the lower-left corner of the menu box and, holding down the left mouse button, drag the box downward, enlarging it so that it can accommodate three more entries.
- Step 12: Place the mouse pointer on the QUIT object and, holding the left button down, move the object to the bottom-most position of the menu box.
- Step 13: Drag the word "ENTRY" from the parts list and place it in the top position of the File menu box, making sure you place it as far to the left as it'll go. Double-click it, and change the text to two spaces followed by "Load...", followed by two more spaces. Name the object "LOAD."
- Step 14: Create another menu entry below LOAD. The text should be two spaces followed by "Save...", followed by two more spaces. Name this object "SAVE."
- Step 15: Drag the ----- icon from the parts list and place it below the SAVE entry, all the way to the left. Then move the QUIT object just below it and name it "QUIT."
- Step 16: Reduce the menu box to its smallest size using the same method as when you enlarged it (Step 11). Add enough dashes to the ----- object (by double-clicking on it and changing the text) to extend it to the right-hand margin of the menu box.
- Step 17: Single-click the Options menu title, and enlarge the menu box to accommodate three entries.
- Step 18: Drag an ENTRY icon to the top of the Options menu box. Set the text to two spaces followed by "Option 1," followed by two more spaces. Before closing the dialog, set the CHECKED option in the attributes list. Name the object "OPTION1."
- Step 19: Create two more entries in the options menu box, named "OPTION2" and "OPTION3," and place them in order below OPTION1. The spacing of the text will be the same as in Step 18. Do not set the CHECKED attribute for these two objects. Reduce the Options menu box to its smallest possible size.
- Step 20: Single-click the Selections title. Then stretch the menu box to accommodate five entries.
- Step 21: Create an entry in the Selections menu named "ONOFF," and enter into the text field five spaces followed by "On" followed by five more spaces.
- Step 22: Drag the ----- icon to a position below the ONOFF entry. Add two dashes to the already existing ten in the text field.
- Step 23: Create three entries below the dashed line. The entries should be named "SELECT1," "SELECT2," and "SELECT3." Their text fields should contain two spaces followed by "Select

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

n" (where n is the entry's number as indicated by the names above), followed by two additional spaces.

Step 24: Reduce the Selections menu box to its smallest possible size.

### The Program

Now that you've got your version of the menu bar saved in a resource file, compile Listing 1, found at the end of this chapter. When you run the program, the menu bar you created will come up on the screen (make sure you have the MENU.RSC file on the same disk as your .PRG file). First pull down the Desk menu and click on "C-manship info..." An alert box should appear, giving you a little information about C-manship (very little, actually).

There are two GEM menu conventions used here. First, you should always place the "Info" selection of your menu bar as the first choice of the Desk menu. Second, any menu entry that will lead to a dialog box of some sort should be followed by ellipsis dots (three periods).

If you had any accessories on your program disk when you booted it, they should also be available on the desk menu. Go ahead; check them out.

Now pull down the File menu. There are a few more conventions to take note of here. If you plan to allow the loading and saving of files from your application, this is where the appropriate menu entries should go. Ditto for Quit commands. When you follow these conventions, users will always know where in a menu bar to find these basic functions. Any other disk handling activities, such as Delete, should also be located here.

In our sample menu, clicking on Load or Save just prints a message to the screen. Quit, of course, returns you to the desktop. Note that the Load and Save entries are followed by ellipsis dots, even though, in this case, they don't lead to a dialog box. Why? Well, when you use them in a real program, they'll almost certainly lead to a file selector box, right?

Now we get to the Options menu. The three entries found here may be turned on and off by clicking them with the mouse. Any options that are active will have a checkmark next to them. You can have as many of them active as you wish (especially considering, due to the stripped-down nature of the demo program, they don't really do anything, anyway!)

Moving right along, we get to the Selections menu. The top entry will toggle between the words ON and OFF each time it's clicked. When the entry is "on," the selections below will be selectable, and will print a message to the screen when they're clicked.

When the entry is "off," the other selections will be "grayed out." This means they have been disabled. No amount of clicking on a disabled menu entry will give you the slightest response, except removing the drop-down menu from the screen. This is why we can get away with those dashes separating different sections of the menus. Since they're disabled, the user can't do anything with them.

### Menu Bars in Your Program

Now that you've had a chance to fiddle with your creation, let's see how the program works. All the action is in the function `do_menu()`, so let's skip over the other stuff. You should be familiar enough by now with how to initialize GEM.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

First, we declare `menu_adr`, a longword that'll contain the address of our menu tree. The actual code begins with the initialization of some flags we'll use to keep track of the status of the various menu options. Then we set the mouse pointer to the arrow form.

Next, we load the resource file from disk, using `rsrc_load()`, which I described in Chapter 15. If the `MENU.RSC` file is missing, we warn the user with an alert box, then return to the desktop.

If the resource file loads okay, we find the address of the menu tree, using the `rsrc_gaddr()` call -- which we also discussed in Chapter 15 -- storing the returned address in `menu_adr`.

To display our menu bar, we call:

```
menu_bar(menu_adr, flag);
```

Here, `menu_adr` is the address of the menu's object tree and `flag` is a Boolean value indicating whether the menu should be displayed (a nonzero value) or removed (a zero value).

### A Nifty Message System

Up until now we've been able to get our user's input with either specific calls to the mouse or keyboard, or by using dialog and alert boxes. We've now gotten to the point in our GEM programming careers where the simple calls just won't do the job. What if we want to be alerted to more than one form of input? What if we want to know about the mouse and the keyboard at the same time? More to the point for this chapter, how do we know what the user is doing with the menu bar?

GEM supplies us with a single function call that will monitor the entire system for us and tell us everything we need to know about the user's actions. Are you ready? Take a deep breath, because you're going to need it. The call (as shown in the Megamax manual) is:

```
evnt_multi(mmflags, mbclicks, mbmask, mbstate, mm1flags, mm1x, mm1y,
            mm1width, mm1height, mm2flags, mm2x, mm2y, mm2width,
            mm2height, mmgpbuff, mtlocount, mthicount, mmox, mmoy,
            mmobutton, mmokstate, mkreturn, mbreturn);
```

Sheesh! I warned you! But before you burn your compilers and open your wrists, you should know that, at this point, there are only a few of the above parameters we're interested in. We'll be discussing this function a lot in upcoming chapters. We'll cover the parts we need bit by bit. For now, let's just say that you don't need to use all the parameters. We can send 0s for any outgoing parameters we're not interested in, and supply dummy locations for any unneeded information being sent back. In our menu bar program, the `evnt_multi()` call looks like this:

```
evnt_multi(MU_MESAG, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
            msg_buf, 0, 0, &dum, &dum, &dum, &dum, &dum, &dum);
```

`MU_MESAG` is the event we want to watch for (in this case, a message event), and `msg_buf` is the address of a 16-byte message buffer where our messages will be stored.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

There are six event types we can wait for. To wait for more than one type of event at a time, we just OR the appropriate flags together. For example, to wait for a message event, a keyboard event, and a mouse button event, our first parameter in the `evnt_multi()` call would be:

```
MU_MESAG | MU_KEYBD | MU_BUTTON
```

Here, `MU_MESAG` has been defined as `0x0010`, `MU_KEYBD` has been defined as `0x0001`, and `MU_BUTTON` has been defined as `0x0002`. This sets the proper bits, as shown in the following table.

<u>Set bit #</u>	<u>Event</u>
0	Keyboard
1	Mouse button
2	Mouse event 1
3	Mouse event 2
4	Message event
5	Timer event

Note that, even though we're not using the call this way in the sample program, since we're only waiting for one type of message, `evnt_multi()` returns a word value that will have bits set based on the events that occurred. These bits follow the same format as the table above. It's possible to have more than one event detected by a single call to `evnt_multi()`, so it's up to the programmer to check each bit in the returned integer, in order not to miss events. If you're checking for more than one event, your `evnt_multi()` call should look something like this:

```
event = evnt_multi(...);
```

Here, `event` is the integer that will contain bits set based on the events detected by the call. Of course, the `"..."` in the parentheses will be replaced by that horrendous parameter list. When we make the `evnt_multi()` call, everything comes to a stop until the event we're waiting for occurs. If the event is a message event, information about the message is stored in the message buffer. In our program, we've set up the array `msg_buf[]` to handle this duty. The type of message will be returned into `msg_buf[0]`. There are thirteen possible messages, but the only one we're interested in right now is the menu selected message, which has a value of 10. When the user makes a selection from the menu, therefore, `msg_buf[0]` will contain 10.

Since this is the only type of message we're waiting for, we don't have to check `msg_buf[0]`. We can just assume that we've received the message we expected.

If this discussion is getting confusing, you might not be sure of the differences between an event and a message. They are not the same thing. A message is received only when a message event -- one of six different events -- is detected.

### Enough of this Event Junk

Now that we have an idea of how `evnt_multi()` works, the program's workings are fairly simple. The object number of the chosen menu title is stored in `msg_buf[3]`, and the object number of the

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

chosen menu entry is stored in `msg_buf[4]`. Once we have that information, we set up nested switch statements to perform the appropriate actions.

The outer switch tests `msg_buf[3]` to see which menu title was selected. In the sample listing, you'll see that there's a case for each menu title. Within each menu title's case is a switch to test `msg_buf[4]`, the object number of the selected menu entry. This is the best way to handle menu messages -- the code looks very much like the menu bar it represents, and so is clear and easy to follow.

Now, let's look at this code in a little more detail, starting with DESK. Since we have only one action we have to take care of, we really didn't need the inner switch statement; we could have used an if, but I wanted to keep that "menu-looking" structure I mentioned before. Here, if the user clicks on the "C-MANSHIP INFO..." entry, we just call up an alert box. Simple. We don't have to handle any desk accessories the user might run; GEM does that for us -- almost. I say almost, because, if a desk accessory is called up, it's up to your program to redraw the screen after the accessory has done its thing. Since we don't have anything special happening in the sample program, we can ignore this bothersome detail.

The File menu selections do nothing fancy, just print a message to the screen. But look at the case statements. Where's Quit? Wasn't that part of the File menu? Yes, indeed. And you'll find it at the very bottom of the sample listing. We're using it to break out of the do/while loop.

The Options selections must handle the check marks that indicate active options. We use the call:

```
menu_icheck(menu_adr, OBJECT, flag);
```

Here, `menu_adr` is the address of the menu's object tree, `OBJECT` is the number of the entry you wish to check or uncheck, and `flag` is a Boolean value that indicates whether a check mark should be drawn (a nonzero value) or removed (a zero value).

In our `menu_icheck()` calls, we're taking care of not only the check marks, but also the flags -- reversing them within the function call with the statement below:

```
op1=!op1
```

How does this work? First, the flag is changed to its opposite state with the NOT operation. Then this new value is passed as the flag to the function. This is one of the advantages of C, being able to nest assignments and expressions.

In the Selects menu section, if the ONOFF entry is selected, we must first change the text in ONOFF, then either enable or disable the other selections. To change the text of the entry, we use the call:

```
menu_text(menu_adr, OBJECT, s);
```

Here, `menu_adr` is the address of the menu bar's object tree, `OBJECT` is the entry's object number, and `s` is the address of the string you wish placed in the object. You should make sure that you've left room in the menu for the largest string you'll be using. Otherwise, you'll mess up the desktop. Also, you must make sure that the string is statically allocated; that is, it's global, not declared within a function. You never know when the user is going to activate that particular menu selection, and if

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

your string isn't available globally, it may not be there when the menu is displayed. What a surprise that'll be.

Finally, we use the flag on to determine if we should enable or disable the rest of the menu entries. To perform this function, we use the call:

```
menu_ienable(menu_adr, OBJECT, flag);
```

Here, menu\_adr is (you guessed it) the address of the menu tree, OBJECT is the object of the entry you wish to modify, and flag is a Boolean value that indicates whether the entry is to be enabled (a nonzero value) or disabled (a zero value). In the sample program, we're using the same trick we used with menu\_ichk() to handle the flag, but since we have three calls to menu\_ienable(), we reverse the flag only in the first one.

When the user moves the mouse pointer over a menu selection, the title of that selection is highlighted, and the associated menu drops down. Once the user has clicked on an entry, the drop-down menu is removed, but the title remains highlighted. The highlighting reminds the user which menu selection is currently being processed. By leaving the title highlighted even after the user has made his selection, we can perform the actions required by the user's choice, then turn the highlighting off when we're ready. We turn off the highlighting with the call:

```
menu_tnormal(menu_adr, title, flag);
```

Here, menu\_adr is the address of the menu tree, title is the object number of the menu title we wish to unhighlight, and flag is a Boolean value that indicates if the title is to be highlighted (a nonzero value) or unhighlighted (a zero value). The value for title will be the one found in msg\_buf[3].

And, last but not least, once we're through with the menu bar, we must remove it from memory. We just use the menu\_bar() call discussed previously to do this. Just change the flag to false.

### Another Lesson Learned

Your repertoire of GEM programming tricks is building fast. It won't be long before you'll be ready to put together some professional looking software. Learning to handle menu bars is an important step in that direction. Without them, you can't really consider yourself a GEM programmer.

Of course, there's still a lot more we have to cover before we can consider GEM a challenge met. But we're getting there.

## Program Listing #1

```

/*****
/*          C-manship, Listing 1          */
/*          CHAPTER 16                    */
/*          Developed with Megamax C      */
*****/

#include "MENU.H"

#define MU_MESAG 0x0010
#define ARROW    0
#define R_TREE   0
#define TRUE     1
#define FALSE    0

/* The usual required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum;
int msg_buf[8], op1, op2, op3, on;

char *alrt = "[1][C-manship, Chapter 16|by Clayton \
            Walnum][Okay]";
char *on_str = "    On    ";
char *off_str = "    Off   ";

main ()
{
    appl_init ();          /* Initialize application.          */
    open_vwork ();         /* Set up workstation.          */
    do_menu();             /* Go do the MENU.              */
    v_clsvwk (handle);     /* Close virtual workstation.    */
    appl_exit ();          /* Back to the desktop.          */
}

open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open */
    /* a virtual workstation.                                     */

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_menu ()
{
    long menu_adr; /* Address of the tree containing our menu. */

    /* First, we initialize our option flags, so we can keep track of which ones are active. Also, we change the mouse pointer to an arrow. */
    op1 = TRUE;
    op2 = FALSE;
    op3 = FALSE;
    on  = TRUE;

    graf_mouse ( ARROW, &dum );

    /* Here we load the resource file. If the file is missing, we warn the user with an alert box then terminate the program by skipping the code following the ELSE. */

    if ( ! rsrc_load ("MENU.RSC") )
        form_alert ( 1, "[1][MENU.RSC missing!][Okay]" );

    /* If the resource file loads OK, we get the address of the tree, then handle menu messages from evnt_multi(). */

    else {
        rsrc_gaddr ( R_TREE, TREE00, &menu_adr );
        menu_bar ( menu_adr, TRUE );
        do {
            evnt_multi ( MU_MESAG, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, msg_buf,
                        0, 0, &dum, &dum, &dum, &dum, &dum, &dum );

            switch ( msg_buf[3] ) {

                case DESK:
                    switch ( msg_buf[4] ) {
                        case INFO:
                            form_alert ( 1, alrt );
                            break;
                    }

                case FILE:
                    switch ( msg_buf[4] ) {
                        case LOAD:
                            v_gtext ( handle, 20, 120, "Load file " );
                            break;
                        case SAVE:
                            v_gtext ( handle, 20, 120, "Save file " );
                            break;
                    }

                case OPTIONS:
                    switch ( msg_buf[4] ) {
                        case OPTION1:
                            menu_ichk ( menu_adr, OPTION1, op1!=op1 );
                            break;
                        case OPTION2:
                            menu_ichk ( menu_adr, OPTION2, op2!=op2 );
                            break;
                        case OPTION3:
                            menu_ichk ( menu_adr, OPTION3, op3!=op3 );
                            break;
                    }
            }
        }
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
    }

    case SELECTS:
        switch ( msg_buf[4] ) {
            case ONOFF:
                if ( on )
                    menu_text ( menu_adr, ONOFF, off_str );
                else
                    menu_text ( menu_adr, ONOFF, on_str );
                menu_ienable ( menu_adr, SELECT1, on!=on );
                menu_ienable ( menu_adr, SELECT2, on );
                menu_ienable ( menu_adr, SELECT3, on );
                break;
            case SELECT1:
                v_gtext ( handle, 20, 120, "Select 1 " );
                break;
            case SELECT2:
                v_gtext ( handle, 20, 120, "Select 2 " );
                break;
            case SELECT3:
                v_gtext ( handle, 20, 120, "Select 3 " );
                break;
        }
        menu_tnormal ( menu_adr, msg_buf[3], TRUE );
    }
}
while ( msg_buf[4] != QUIT );
menu_bar ( menu_adr, FALSE );
}
```

### CHAPTER 17 - WINDOWS - PART 1 - DRAWING

So far, we've talked about dialog boxes (including alert boxes and file selector boxes) and menu bars. That leaves us with one important area of GEM we've yet to touch upon: windows. This is a complex subject, one that we'll need several chapters to cover. The subject of windows can get as complex as you'd like. There's almost no end to the ways we can use them.

#### What Are Windows Really?

We've all used them, but how many of us have sat down and thought about what a window really is? From the user's point of view, a window truly lives up to its name, allowing him to move a transparent opening over information that may be too lengthy to fit on his screen, giving him a glimpse at data stored somewhere beyond the borders of our desktop.

But this "windowness" is just an illusion, the result of some programmer's tedious and careful work. A window is not a magical creation; it's just a box.

Imagine, if you will, a child's slate on which you've written as many film titles as will fit. Now some guy comes up to you and says, "The movie I'm looking for isn't on that list. Let me see some more." So you take out your eraser, erase the slate, and chalk some more titles onto its surface. The man shakes his head, mumbles something like "Maybe it wasn't a movie after all," and tells you to put the slate down on a table with several others. He then points to a different slate and asks you to pick it up. On this one are written book titles. The man smiles (Gee, look! There's a piece of spinach in his teeth!) and points to the title *Foundation and Earth* by Isaac Asimov. You erase the slate, set it back on the table, then go to the library and retrieve the book. The end.

Who's the bossy guy in the story? The user, of course. And "you" are the programmer, manipulating the "windows" in the manner the user requests.

Okay, maybe windows are a little fancier than a chalk slate. They do have some extra parts (if we want to use them), such as sliders, movers, fullers, closers, etc., and GEM does provide a small amount of help with handling windows. But, for the most part, a window is just what I said before: a box -- a box that you, the programmer, have to maintain in accordance with messages received from your program's user.

Figure 1 shows all the components contained in a complete window. You can use any or all of these parts, depending on your application's needs.

#### The Window Demo

When you run this chapter's program, a simple window will come up on the screen. This window won't contain all of the parts shown in Figure 1; it will have only a title bar, an information bar, a mover bar (actually the same as the title bar), a fuller button and a closer button -- the parts we're going to cover in this chapter.

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

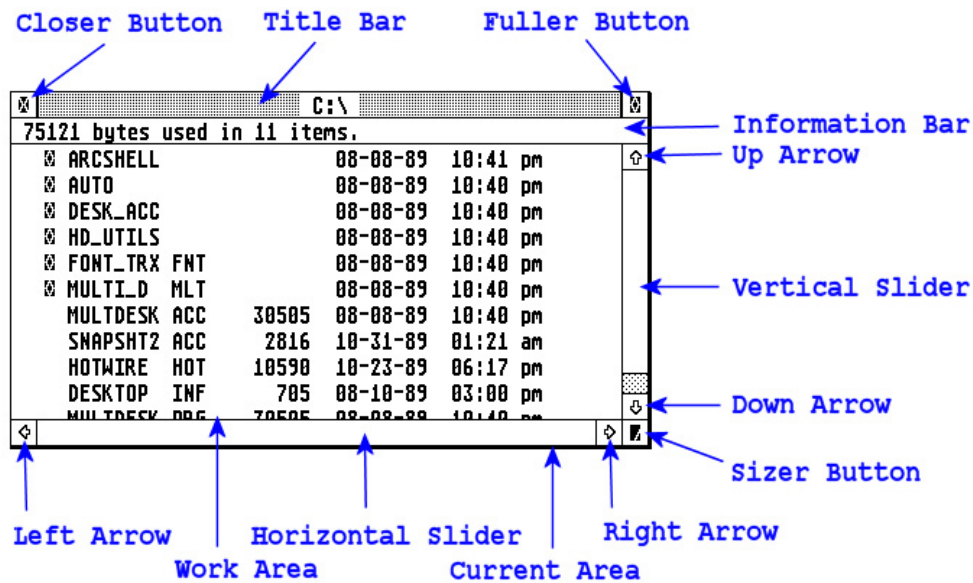


FIGURE 1 - Window Components

Clicking on the fuller button causes the window to fill the entire desktop workspace. Clicking it a second time will return the window to its previous size. If you place the mouse pointer on the mover bar, then press and hold the left button, you can drag the window to any location you like on screen - even off the screen, if you want. However, if you do move part of the window off the screen, when you move it back again, you'll notice that the window's workspace isn't redrawn. We'll see why this happens in an upcoming chapter.

When you're through experimenting, click on the closer button. The window closes, and you're returned to the desktop.

### Drawing a Window

Now let's take a look at this chapter's program listing in detail and see how all this window stuff works. The function `do_wndw()` is where most of the fun takes place, so we'll start there.

The first thing we must do is decide what our window's maximum allowable size will be. We can limit the size to anything we want, but, in most cases, a window's maximum size is equal to the desktop workspace. The desktop is actually a window itself, the workspace of which is all the area of the screen, excluding the menu bar. The size of this workspace, measured in pixels, varies with the resolution, so we need a way to find out what the actual coordinates are. Luckily, GEM provides us with a function that'll supply the information we need. The call below will return requested information about a window:

```
wind_get(w_h, flag, &x, &y, &w, &h);
```

Here, `w_h` is the window's handle (in the case of the desktop, the handle is always 0), the integer `flag` is a flag telling the function what information we want, and `&x`, `&y`, `&w`, and `&h` are the addresses where the returned information will be stored. What information is actually placed in these locations depends on the value of `flag`. To get the work area's rectangle, we need to make `flag` equal to `WF_WORKXYWH`, which is defined in the Megamax header file, `GEMDEFS.H`.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The function `wind_get()` can provide us with much information about our window, including the size of the work area, the size of the entire window, the window's maximum allowable size, the previous window's size, the position or size of either the vertical or horizontal sliders, and the coordinates of the first or next rectangle in the rectangle list (something we'll cover in another chapter). All the flags you need to request any of this information are already set up for you in the `GEMDEFS.H` header file that came with your compiler.

Now, where were we? Oh, yes! To get the size of the desktop's work area (which we'll use as our window's maximum allowable size), our call to `wind_get()` should look like this:

```
wind_get(0,WF_WORKXYWH,&fullx,&fully,&fullw,&fullh);
```

Remember: window handle 0 is always the desktop.

Now we know the maximum allowable size for our window, and we've stored that information in `fullx` (X-coordinate of the window's upper left corner), `fully` (Y-coordinate of the window's upper left corner), `fullw` (the window's width), and `fullh` (the window's height). Next, we need to generate the window, as well as get its handle. (A window's handle is its name; that way we can differentiate it from other windows that may also be in use.) We do this with the call:

```
w_handle=wind_create(PARTS,fullx,fully,fullw,fullh);
```

Here, the integer `w_handle` will receive the window's handle (a negative value indicates that the window couldn't be opened), `PARTS` is a flag representing the components we want included in the window, and `fullx`, `fully`, `fullw`, and `fullh` are the window's maximum allowable size. A call to `wind_create()` does not actually draw the window; it only sets up the window in memory.

In our sample listing, `PARTS` is defined as:

```
NAME | CLOSER | FULLER | MOVER | INFO
```

The definitions for these labels (and all the others needed for a complete window) are defined in the Megamax header file `GEMDEFS.H` as on the next page.

Label	Value
NAME	0x0001
CLOSER	0x0002
FULLER	0x0004
MOVER	0x0008
INFO	0x0010
SIZER	0x0020
UPARROW	0x0040
DNARROW	0x0080
VSLIDE	0x0100
LFARROW	0x0200
RTARROW	0x0400
HSLIDE	0x0800

As you can see, each of the above values sets a particular bit in the flag. To select the parts you wish included in your window, you OR the appropriate values together. Though you may include as many

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

or few of the parts as you need for your application, you should never include in a window parts you don't plan on handling in your code. It tempts the user to play around with things he shouldn't, even though, in most cases, it won't do any harm; only your program can actually change a window.

Since we've included the title and information bars in our window, we need to tell GEM where the associated strings can be found. If we neglect to do this, we'll get unpredictable results; we may even end up staring at a row of bombs across our screen (nasty old things). The call below fits the bill nicely:

```
wind_set(w_handle, WF_NAME, title, 0, 0);
```

Here, `w_handle` is the window's handle, `WF_NAME` (defined in the Megamax header file `GEMDEFS.H`) is a value indicating the field we wish to change, `title` is the address of the string we want displayed, and the two 0s are dummy arguments.

Just as with `wind_get()`, `wind_set()` has many possible values for its flag (represented by `WF_NAME` in the above call), each of which lets you change one of your window's attributes, including the title or information text, the window's position, the window's components, the sliders' size or position, and whether or not the window is the topmost (active) window. All the flags for this function are defined in your `GEMDEFS.H` file.

The above call takes care of the title bar. We must make another call to `wind_set()` for the information line. The call is exactly as above except you would replace `WF_NAME` with `WF_INFO` and `title` with a pointer to the window info string.

Now we're ready to actually bring the window up on the screen. First, we make a call to draw the animated, expanding box:

```
graf_growbox(startx, starty, startw, starth, endx, endy, endw, endh);
```

Here, the integers `startx`, `starty`, `startw`, and `starth` are the X- and Y-coordinates of the upper-left corner and the width and height, respectively, of the box's starting rectangle. The integers `endx`, `endy`, `endw`, and `endh` are the equivalent values for the box's ending rectangle.

The call below opens and draws a window:

```
wind_open(w_h, x, y, w, h);
```

Here, `w_h` is the window's handle, and the integers `x`, `y`, `w`, and `h` are the X- and Y-coordinates of the upper-left corner and the width and height of the window, respectively. You can open the window to any size less than or equal to the maximum you set with the `wind_create()` call.

Next, we call our own function, `draw_backgrd()`, to fill in the new window's work area. The call to `wind_open()` actually draws only the window's borders and whatever parts we requested when we created the window. The work area is the programmer's responsibility. It's there for us to do with it what we like.

Let's follow the flow of the program now by taking a look at the function `draw_backgrd()`. First, we turn off the mouse so the pointer doesn't interfere with our drawing. Then we call `wind_get()`, as we

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

did before, to get the coordinates and size of the work area of the window we just opened. Now that we know this information, we simply draw a filled rectangle at the coordinates returned. A piece of cake!

### Handling a Window

Okay, our window's on the screen. Now what do we do with it? We get information about what the user is doing with our window the same way we did with menu bars -- through messages. Since we're interested only in one type of message in the sample program, we're not going to bother with that bulky `evnt_multi()` call. There's an easier way:

```
evnt_mesag(msg_buf);
```

This call allows us to wait for messages without all of `evnt_multi()`'s extra and burdensome baggage. Here, `msg_buf` is the address of a 16-byte buffer where the message will be stored. Every time the user does something with our window, `evnt_mesag()` will notify us.

The messages we'll receive are limited to those generated by the parts we included in our window when we created it. More specifically, in Listing 1, the only actions we're looking for are: the window was moved, the fuller button was clicked or the close button was clicked.

When one of these actions occurs, a message is written to our message buffer. The first word, `msg_buf[0]`, will contain the message type received. We'll use this value in a switch statement to choose the appropriate action.

### Window Moved

If the window is moved, we'll receive a `WM_MOVED` (defined in the Megamax `GEMDEFS.H` file) message, telling us we have to reposition the window. The handle of the window moved will be found in `msg_buf[3]`. The coordinates and size will be found in `msg_buf[4]` through `msg_buf[7]` (X, Y, W and H, respectively). We move the window with the call:

```
wind_set(msg_buf[3], WF_CURRXYWH,  
         msg_buf[4],msg_buf[5],msg_buf[6], msg_buf[7]);
```

The label `WF_CURRXYWH` is defined in the Megamax `GEMDEFS.H` file and tells `wind_set()` that we want to change the current window's coordinates, automatically moving the window to the new position.

What if the user moved the window and we ignored the message by not calling `wind_set()`? The user would be able to move the window's outline around the screen all he wanted, but as soon as he released the button, the outline would vanish, leaving the window in its original location.

### Full Size or Previous Size?

Another message we might receive in our sample program is `WM_FULLED` message. We get this message when the user clicks on the fuller button, at which time we must either expand the window to its maximum size or, if it's already at its maximum, return it to its previous size. It's up to the programmer to figure out which size is the right one.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The first thing we do is call our own function, `full_wind()`, to set `full_flag` to its proper state. All `full_wind()` does is get the coordinates of the current and full-size windows and compare them. If any of the current coordinates don't match the full-size coordinates, we know that the window is not at its maximum, and we return a value of `TRUE`. If all the coordinates match, we're already at maximum and need to set the window back to its previous size. We signal this by setting our flag to `FALSE`.

If we need to reduce the window to its old size, we first need to know the original coordinates. A call to `wind_get()`, where the second argument is `WF_PREVXYWH` takes care of that. Once we have the old coordinates, we call:

```
graf_shrinkbox(startx, starty, startw, starth, endx, endy, endw, endh);
```

to animate the shrinking box (the parameters are the same as for `graf_growbox()`), then reposition the window with `wind_set()`.

The process of setting the window to its full size is similar, except we draw an expanding box instead of a shrinking one and use the full-size coordinates for the call to `wind_set()`. Also, when expanding the window to its maximum, we have to perform a window redraw (in this case, it's just a matter of drawing that rectangle in the work area).

### Closed For Business

Now, all we have to do is provide a way for the user to get out of our program. The window's close button is perfect for this. When the user clicks it, we'll receive a `WM_CLOSED` message, which will cause us to exit our `do/while` loop.

When we exit the loop, we find the coordinates of the current window with a call to `wind_get()`. Then we use those coordinates in a call to `graf_shrinkbox()`. To get rid of the window, we must first close it with the call:

```
wind_close(w_handle);
```

Here, the integer `w_handle` is the window's handle. Then we must remove the window from memory with the call:

```
wind_delete(w_handle);
```

### More to Come

In the following chapters, we'll learn what to do with redraw messages, how to handle sliders and arrows, and how to deal with multiple (gasp!) windows. Betcha can't wait, huh?



## Program Listing #1

```

/*****
/*          C-manship, Listing 1          */
/*          CHAPTER 17                    */
/*          Developed with Megamax C      */
*****/

#include <osbind.h>
#include <gemdefs.h>
#include <obdefs.h>

#define TRUE 1
#define FALSE 0
#define PARTS NAME | CLOSER | FULLER | MOVER | INFO

/* The usual required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum, fullx, fully, fullw, fullh,
curx, cury, curw, curh, oldx, oldy, oldw, oldh;

int msg_buf[8];

char *title = "C-manship - Chapter 17";
char *info = "Learning about windows";
main ()
{
    appl_init ();          /* Initialize application.          */
    open_vwork ();         /* Set up workstation.             */
    do_wndw ();            /* Go do the window stuff.         */
    v_clsvwk (handle);     /* Close virtual workstation.      */
    appl_exit ();          /* Back to the desktop.            */
}

open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open
    /* a virtual workstation.

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_wndw ()
{
    int w_handle, full_flag;

    /* Find the size of the desktop's (handle 0) work area. */
    wind_get ( 0, WF_WORKXYWH, &fullx, &fully, &fullw, &fullh );

    /* Create window in memory. */
    w_handle = wind_create ( PARTS, fullx, fully, fullw, fullh );

    /* Set the window's title and info text. */
    wind_set ( w_handle, WF_NAME, title, 0, 0 );
    wind_set ( w_handle, WF_INFO, info, 0, 0 );

    /* Draw the window on the screen. */
    graf_growbox ( 10, 10, 10, 10, 50, 50, 250, 200 );
    wind_open ( w_handle, 50, 50, 250, 150 );
    draw_backgrd ( w_handle );

    /* Change mouse to arrow. */
    graf_mouse ( ARROW, 0L );

    /* Receive event messages until the closer is clicked. */
    do {
        evnt_mesag ( msg_buf );
        switch ( msg_buf[0] ) { /* msg_buf[0] is message type.

                                /* If window is moved, set window at new
                                location found in msg_buf[4] through
                                msg_buf[7]. The handle of the
                                window moved is in msg_buf[3]. */

        case WM_MOVED:
            wind_set ( msg_buf[3], WF_CURRXYWH, msg_buf[4],
                msg_buf[5], msg_buf[6], msg_buf[7] );
            break;

            /* If the fuller button has been clicked, set window to
            appropriate size based on full_flag. */

        case WM_FULLED:
            full_flag = full_wind ( w_handle );
            if ( !full_flag ) {
                wind_get ( w_handle, WF_PREVXYWH,
                    &oldx, &oldy, &oldw, &oldh );
                graf_shrinkbox ( oldx, oldy, oldw, oldh,
                    fullx, fully, fullw, fullh );
                wind_set ( msg_buf[3], WF_CURRXYWH,
                    oldx, oldy, oldw, oldh );
            }
            else {
                wind_get ( w_handle, WF_CURRXYWH,
                    &curx, &cury, &curw, &curh );
                graf_growbox ( curx, cury, curw, curh,
                    fullx, fully, fullw, fullh );
                wind_set ( msg_buf[3], WF_CURRXYWH,
                    fullx, fully, fullw, fullh );
                draw_backgrd ( w_handle );
            }
            break;
        }
    }
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
}
while ( msg_buf[0] != WM_CLOSED );

/* Get current size of window for use in graf_shrinkbox, */
/* then close and delete the window. */

wind_get ( w_handle, WF_CURRXYWH, &curx, &cury, &curw, &curh );
graf_shrinkbox ( 10, 10, 10, 10, curx, cury, curw, curh );
wind_close ( w_handle );
wind_delete ( w_handle );
}

/* This function calculates if the window should be drawn to */
/* its maximum size or reset to its previous size. */

full_wind ( w_h )
int w_h;
{
    int    c_x, c_y, c_w, c_h,
           f_x, f_y, f_w, f_h;

    wind_get ( w_h, WF_CURRXYWH, &c_x, &c_y, &c_w, &c_h );
    wind_get ( w_h, WF_FULLXYWH, &f_x, &f_y, &f_w, &f_h );
    if ( c_x != f_x || c_y != f_y || c_w != f_w || c_h != f_h )
        return ( TRUE );
    else
        return ( FALSE );
}

/* This function draws a white background in a window's */
/* work area. w_h is the window's handle. */

draw_backgrd (w_h)
int w_h;
{
    int wrk_x, wrk_y, wrk_w, wrk_h;
    int pxy[4];

    /* Turn off mouse for all drawing operations. */
    graf_mouse ( M_OFF, 0L );

    /* Get the size of the window's work area. */
    wind_get ( w_h, WF_WORKXYWH, &wrk_x, &wrk_y, &wrk_w, &wrk_h );

    /* Set the color and fill style. */
    vsf_interior ( handle, 1 );
    vsf_color ( handle, WHITE );

    /* Draw the rectangle in the window work area. */
    pxy[0] = wrk_x;
    pxy[1] = wrk_y;
    pxy[2] = wrk_x + wrk_w - 1;
    pxy[3] = wrk_y + wrk_h - 1;
    vr_recfl ( handle, pxy );

    /* Drawing over, so turn mouse back on. */
    graf_mouse ( M_ON, 0L );
}
```

## CHAPTER 18 - WINDOWS - PART 2 - SIZING

In our last gem experiments, we learned some of the basic functions used to manipulate windows. We discovered how to initialize, open and close a window, as well as draw its interior (the work area), and handle the messages GEM sends to our application when the user wishes to move the window or make it full-size.

As I'm sure you've guessed, there's still more to learn before we can call ourselves window experts -- much more. We still need to know how to handle many other types of window messages, not the least of which is the redraw message. We also need to know how to use the window's sizer button, scroll bars and arrow buttons. And then there's the matter of multiple windows!

In the next couple of chapters, we'll tackle all of the above topics, giving you a fairly complete understanding of programming with windows.

### The Demo Program

When you run this chapter's program, a full-screen window containing some text will be opened. You can drag the window around the screen by placing the mouse pointer on the title bar and holding down the left button. You can change the size of the window, using the mouse to click and drag on the sizer button. When you're through experimenting with the window, exit the program by clicking the closer button.

It doesn't seem as if there's much more going on here than in Chapter 17's program, does it? But there is...there is...

### Any Size You Like

The GEM window form provides a button that is used to set a window to any size the user requests. When the user activates the sizer button (by click/dragging it), an outline of the window appears on the screen. The outline expands and contracts with the movement of the mouse pointer -- as long as the left button is held down. The moment the button is released, the window is redrawn at the size selected by the outline.

Handling the sizer button is simple (well, there are a couple of complications that we'll get to in a minute). First, of course, when initializing the window, you must tell GEM to include the sizer button in its parts list. Then it's just a matter of waiting for the WM\_SIZED message and using the call:

```
wind_set(msg_buf[3], WF_CURRXYWH,  
         msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7]);
```

This sets the window's current size to that returned in the message buffer. (If you've forgotten how this message stuff works, review the last couple of chapters.) Easy.

However, (have you noticed that there's always a "however"?) there are two things we have to watch out for. The first is a redraw message. We will receive one of these when the window is moved from a partially off-screen position back to the desktop, or when it's increased in size, either horizontally or vertically. Also, it's up to us, the programmers, to keep the size of a window within certain limitations.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

The maximum size of a window really isn't a problem. The window will never be larger than the desktop, anyway. It's just nice to know that, if we ever decide we want the window kept below a certain maximum size, we can.

The minimum size, however, is important. If we don't check the requested size of the window before setting it, we may find we've left the user with a tiny, unusable blob on the screen. It's the programmer's responsibility to make sure the window isn't set to a size too small to contain the parts it needs to function.

Take a look at the `do_wndw()` function in Listing 1. It's here that we handle the messages GEM sends us. You'll notice that both `WM_MOVED` and `WM_SIZED` messages are dealt with the same way, via a call to our function `do_move()`. The reason is that both these window events are handled with the `wind_set()` call.

In the function `do_move()`, the first thing we do is check the requested window size, returned to us in `msg_buf[4]` through `msg_buf[7]`. If the height or width of the requested window is smaller than the minimums we've set (the actual values can be found in the `#defines` at the top of the listing), we replace the unacceptable values with our minimum values. The user can try to set the window as small as he likes, but it will never be drawn at a size smaller than our minimum.

If we need to control the window's maximum size, we would do it the same way, adding another set of `if` statements to check for compliance with whatever maximum values we wished to force on the window.

### Redraw Messages

As we move our window around the screen, and as long as we stick to certain limitations, GEM is perfectly happy to redraw the window and its contents for us. But there are times when GEM stops right in its tracks, scratches its silicon head and says, "Here! You figure it out!" To understand why this happens, we need to know a little about how GEM does its window tricks.

Basically, GEM has no trouble redrawing a window's contents (its work area), as long as all the data needed is present on the screen. For instance, let's say we've drawn a small window in the center of the screen and filled its work area in with a white background. Now we move that window a little to the right. GEM can redraw the window because all the information it needs is still on the screen. When it does the redraw, all it does is "blit" the old screen information to its new location. The complication begins when we do one of two things with our window: move it on to the screen from a partially off-screen position, or enlarge it.

In both of the above cases, as I'm sure you can see, the information GEM needs to redraw the window isn't available on the screen. GEM will make the attempt and redraw as much information as it can, but when it gets to the missing data, it'll give up and tell us to handle it, by sending a redraw message.

This description is, of course, an over simplification. Things get much more complicated when you start dealing with multiple windows, since any movement of the active window (the topmost window) will surely result in redraw messages for some or all windows underneath.

Redrawing a window is not, unfortunately, a simple process. To do it properly, you must perform the following steps:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

- Step 1 Lock the window for an update.
- Step 2 Get the first rectangle to be redrawn from the rectangle list.
- Step 3 Check whether the rectangle returned from the redraw message (stored in msg\_buf[4] through msg\_buf[7]) intersects the rectangle obtained from the rectangle list.
- Step 4 If there's an intersection, set the clipping rectangle equal to the rectangle obtained from the rectangle list, and redraw the window.
- Step 5 Get the next rectangle from the rectangle list.
- Step 6 Perform steps 3 through 5 until the rectangle list is empty (width and height both equal to 0).
- Step 7 Unlock the window.

Sound like fun? Let's take an in-depth look at each step.

### Lock the Window

The first thing we must do before updating a window is lock it away from GEM. This prevents the user from doing anything to "pollute" the screen -- such as activating the menu bar or a desk accessory -- while we're trying to get the window redrawn. Essentially, it stops two applications from writing to the screen at the same time. To lock a window, we use the call:

```
wind_update ( BEG_UPDATE );
```

The integer BEG\_UPDATE (defined as 1) tells GEM we're going to start updating the window and that other writes to our window should be disallowed. The wind\_update() call also supports three other functions, depending on the value of the flag. The four flags and their values as defined in the GEMDEFS.H header file are as follows:

END_UPDATE	0	Unlock window
BEG_UPDATE	1	Lock window
END_MCTRL	2	Lock mouse control
BEG_MCTRL	3	Begin mouse control

### The Rectangle List

In order to facilitate redrawing, GEM divides each window into a series of complete rectangles, then stores the coordinates of these rectangles in a rectangle list. We'll be discussing the rectangle list in greater depth in Chapter 19, when we start working with multiple windows; for now, it's enough to say that each time we get a redraw message, we must read each rectangle in the list, compare it to the rectangle returned from the redraw message, and redraw those rectangles that need updating.

In Listing 1, the function do\_redraw() demonstrates how to handle the rectangle list. We get the first rectangle in the list with the call:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
wind_get(msg_buf[3],WF_FIRSTXYWH,&x,&y,&w,&h);
```

Here, the integer msg\_buf[3] is the window's handle (returned from the redraw message) and &x, &y, &w, and &h are the addresses of integers in which the coordinates of the rectangle will be stored. The flag WF\_FIRSTXYWH is defined as 11 in the GEMDEFS.H file.

We know we're at the end of the rectangle list when both w and h are 0. So, after getting the first rectangle, we enter a while loop that tests for this condition.

Once we have a rectangle, we must test to see if it lies within the "dirty" area of the screen (usually the full area of the window that generated the redraw message). We use the call:

```
rc_intersect( rec1, rec2 );
```

Here, rec1 and rec2 are pointers to data of type GRECT. This returns a 1 if the rectangles intersect and a 0 if they don't. GRECT is defined in the GEMDEFS.H file and is nothing more than a structure consisting of four integers: the X- and Y-coordinates of the rectangle, and its width and height.

### The Clipping Rectangle

If the two rectangles intersect, we've found an area of the screen that must be redrawn. In order to be sure the data we're going to write doesn't overflow its area, we set a clipping rectangle.

A clipping rectangle confines all writing to a specific portion of the screen. Anything that we try to draw outside of this area will be "clipped" off. Once we've set a clipping rectangle, we don't have to worry about figuring out exactly what to draw and where; we just redraw the entire window and let the clipping function do the hard part. To set a clipping rectangle, we use the call:

```
vs_clip( handle, flag, pxy );
```

Here, handle is our application's handle; flag is an integer that, when FALSE (0), turns clipping off, and, when TRUE (1), turns clipping on; and pxy is a pointer to an array of four integers where the coordinates of the upper-left and lower-right corners of the clipping rectangle have been stored.

See a small problem? The rectangle we want to redraw is given to us in the usual AES form of X, Y, width, and height; yet the clipping rectangle must be set using the VDI type of rectangle form. That's why, in our function set\_clip(), we first have to do some simple conversions.

Once the clipping rectangle is set, we just redraw the window's interior, letting the clipping function figure out where and where not to place data. When we're through updating the window, we call vs\_clip() a second time with flag set to FALSE to turn off clipping.

### Emptying the Rectangle List

In order to be sure we've updated the entire screen (wherever it needed it), we must "walk the rectangle list." That is, check every rectangle in the list against the rectangle returned from the redraw message. To get the remaining rectangles in the list, we use the call:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
wind_get(msg_buf[3], WF_FIRSTXYWH, &x, &y, &w, &h);
```

The parameters here are the same as for the call we used to get the first rectangle, except that the flag is now `WF_NEXTXYWH`, which is defined as 12 in the `GEMDEFS.H` file.

We continue pulling rectangles from the list and redrawing them, if necessary, until the width and height values we get are both 0. At that point, we know we've checked all the rectangles. Once we've completed the rectangle list, the only thing left to do is unlock the window with the call:

```
wind_update( END_UPDATE );
```

### Something of Interest

At the top of Listing 1, we've defined the text we need for our window work area like this:

```
char *text[] = {  
    "This is some sample text",  
    "for use in the C-manship",  
    "window demonstration found",  
    "in Chapter 18."  
};
```

This is a way to simulate string arrays in C. The array `text[]` is actually an array of pointers to character, each containing the address of one of the strings found within the quotes. The element `text[0]` contains the address of "This is some sample text"; element `text[1]` contains the address of "for use in the C-manship," and so on.

In Listing 1, the function `draw_interior()` shows how to access this array to print the text to the window. We simply use a for loop to advance through each element of the array, while in each iteration of the loop, we use the current array element as an argument to `v_gtext()`.

### The Agenda

That's enough book work for now. Next time around, we'll look at some sample code for handling more than one window at a time. We'll also dig deeper into this confusing rectangle business. And, who knows? Maybe we'll find some other trouble to get into, as well.



# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```
/* **** */
/*      C-manship, Listing 1      */
/*      CHAPTER 18                */
/*      Developed with Megamax C  */
/* **** */

#include <gemdefs.h>
#include <obdefs.h>

#define TRUE 1
#define FALSE 0
#define PARTS NAME|CLOSER|MOVER|SIZER
#define MIN_WIDTH 64
#define MIN_HEIGHT 64

/* GEM global arrays */
int  work_in[11],
     work_out[57],
     pxyarray[10],
     contrl[12],
     intin[128],
     ptsin[128],
     intout[128],
     ptsout[128];

/* Global variables */
int  handle, fullx, fully, fullw, fullh, wrkx, wrky,
     wrkw, wrkh, curx, cury, curw, curh, w_handle,
char_w, char_h, box_w, box_h;

int msg_buf[8];

char *title = "C-MANSHIP #17";

char *text[] = {
    "This is some sample text",
    "for use in the C-manship",
    "window demonstration found",
    "in Chapter 18."
};

int num_lines = 4;

main ()
{
    appl_init ();          /* Initialize application.      */
    open_vwork ();         /* Set up workstation.         */
    do_wndw();             /* Go do the window stuff.     */
    v_clsvwk (handle);     /* Close virtual workstation.   */
    appl_exit ();          /* Back to the desktop.        */
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open */
    /* a virtual workstation. */
    handle = graf_handle ( &char_w, &char_h, &box_w, &box_h);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

do_wndw ()
{
    /* Initialize and open our window. */
    open_window();

    /* Change mouse to arrow. */
    graf_mouse ( ARROW, 0L );

    /* Receive event messages until the closer is clicked. */
    do {
        evnt_mesag ( msg_buf );
        switch ( msg_buf[0] ) { /* msg_buf[0] is message type. */

case WM_MOVED:
case WM_SIZED:
            do_move();
            break;

case WM_REDRAW:
            do_redraw ( (GRECT *) &msg_buf[4] );
            break;
        }
        while ( msg_buf[0] != WM_CLOSED );

        /* Close and delete the window. */
        close_window();
    }

do_move()
{
    /* Set window at new location. Also disallow any */
    /* window sizes less than our minimum allowable size. */

    if ( msg_buf[6] < MIN_WIDTH )
        msg_buf[6] = MIN_WIDTH;
    if ( msg_buf[7] < MIN_HEIGHT )
        msg_buf[7] = MIN_HEIGHT;
    wind_set ( msg_buf[3], WF_CURRXYWH,
        msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7] );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
draw_interior ( clip )
GRECT clip;
{
    int pxy[4], y, x;

    /* Turn mouse off prior to drawing. */
    graf_mouse ( M_OFF, 0L );

    /* Calculate clip rectangle and turn clipping on. */
    set_clip ( TRUE, clip );

    /* Get coordinates of window's work rectangle. */
    wind_get ( w_handle, WF_WORKXYWH, &wrkx, &wrky, &wrkw, &wrkh);

    /* Set the color and fill style. */
    vsf_interior ( handle, 1 );
    vsf_color ( handle, WHITE );

    /* Draw the background in the window's work area. */
    pxy[0] = wrkx;
    pxy[1] = wrky;
    pxy[2] = wrkx + wrkw - 1;
    pxy[3] = wrky + wrkh - 1;
    vr_recfl ( handle, pxy );
    /* Write the text to the window. */
    y = wrky + box_h;
    for ( x=0; x<num_lines; ++x ) {
        v_gtext ( handle, wrkx+8, y, text[x] );
        y += box_h;
    }

    /* Drawing over, so turn the clipping */
    /* off and turn the mouse back on. */
    set_clip ( FALSE, clip );
    graf_mouse ( M_ON, 0L );
}

do_redraw ( rec1 )
GRECT *rec1;
{
    GRECT rec2;

    /* Lock window for update. */
    wind_update ( BEG_UPDATE );

    /* Get first rectangle from list. */
    wind_get ( msg_buf[3], WF_FIRSTXYWH,
               &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );

    /* Loop through entire rectangle list, */
    /* redrawing where necessary. */
    while ( rec2.g_w && rec2.g_h ) {
        if ( rc_intersect ( rec1, &rec2 ) )
            draw_interior ( rec2 );
        wind_get ( msg_buf[3], WF_NEXTXYWH,
                   &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );
    }
    /* Unlock window after update. */
    wind_update ( END_UPDATE );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
set_clip ( flag, rec )
int flag;
GRECT rec;
{
    int pxy[4];

    /* Convert rectangle to pxy coords. */
    pxy[0] = rec.g_x;
    pxy[1] = rec.g_y;
    pxy[2] = rec.g_x + rec.g_w - 1;
    pxy[3] = rec.g_y + rec.g_h - 1;

    /* Turn clipping on or off. */
    vs_clip ( handle, flag, pxy );
}

open_window()
{
    /* Find the size of the desktop's work area. */
    wind_get ( 0, WF_WORKXYWH, &fullx, &fully, &fullw, &fullh );

    /* Create window in memory. */
    w_handle = wind_create ( PARTS, fullx, fully, fullw, fullh );

    /* Set the window's title. */
    wind_set ( w_handle, WF_NAME, title, 0, 0 );

    /* Draw the window on the screen. */
    graf_growbox ( 10, 10, 10, 10, fullx, fully, fullw, fullh );
    wind_open ( w_handle, fullx, fully, fullw, fullh );
}

close_window()
{
    /* Get current size of window for use in graf_shrinkbox, */
    /* then close and delete the window. */

    wind_get ( w_handle, WF_CURRXYWH, &curx, &cury, &curw, &curh );
    graf_shrinkbox ( 10, 10, 10, 10, curx, cury, curw, curh );
    wind_close ( w_handle );
    wind_delete ( w_handle );
}
```

## CHAPTER 19 - WINDOWS - PART 3 - THE RECTANGLE LIST

I'd wager that you're a little perplexed about the rectangle handling that we started discussing in Chapter 18. Don't feel bad. Not only is it a complex topic, but it's also virtually impossible to find complete documentation on it anywhere. Most of the books I've seen merely gloss over the subject, as if the reader were born with an intimate knowledge of GEM's rectangle list.

Well, friends and neighbors, I, for one, was not born with that knowledge. I spent a month in research, trying to dig out all the facts I could about rectangle lists, not only because I wanted to clarify the issue for myself, but because I wanted to put together a decent tutorial to help you, the reader, understand this mysterious process. This chapter's program is based on what

I've discovered.

### Rectangles Revealed

In the course of my research, I spoke with Frank Cohen of Regent Software, and he told me that one method he used to sort out this rectangle nonsense was to update each rectangle returned from the redraw message with a different fill pattern. I told him I thought that idea was sheer genius (well, slightly clever, anyway), and as soon as I hung up the phone, I set about stealing his idea.

Steal it, I did (with his blessings, I hope). The demo program at the end of this chapter uses Frank's method to graphically illustrate the process of walking the rectangle list. Figures 1 through 12 take you step by step through the following tutorial. These screens were taken from a monochrome monitor, so if you have a color system, you may get slightly different results.

When you run the program, you'll first see the screen shown in **Figure 1**.

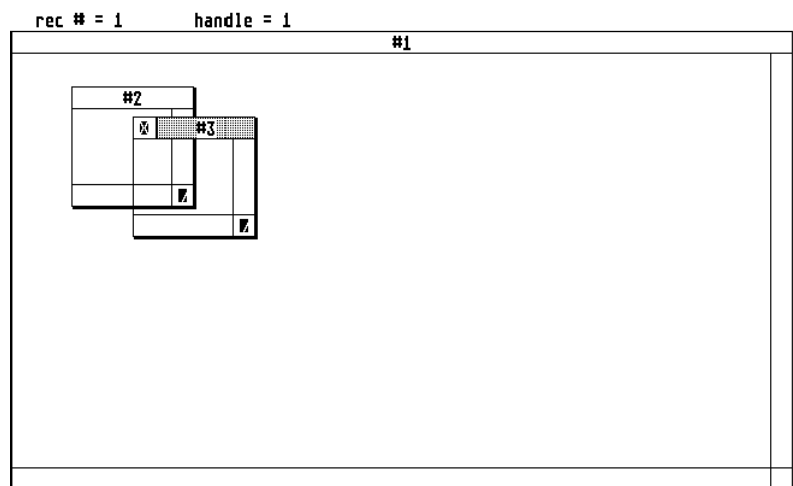
Three windows have been opened, and, since the windows need to have their work areas drawn, GEM has sent the program three redraw messages, one for each window. Because I inserted a call to `Cconin()` in the rectangle list processing loop, nothing further will happen until you press Return.

But before we let GEM do its thing,

take a look at the top line of the screen. Here you'll see the handle of

the window for which the redraw message was sent and the number of the rectangle from the rectangle list we're currently working on. In **Figure 1**, we see that window 1 is waiting for a redraw, and that we're about to process rectangle 1 from the list.

We can't go blindly ahead and fill in window 1's workspace because part of that space is covered by the other two windows; we don't want to erase them. For this reason, when GEM created the redraw message for window 1, it took that window's workspace (only those areas not covered by the other two windows) and divided it into a series of non-overlapping rectangles. It then loaded the



**Figure 1**

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

coordinates and sizes of those rectangles into the rectangle list where we can get at them for processing.

Now press Return once. The first rectangle from the list will be processed, the screen will be updated, and the program once again waits for a key press. But before we give it that key press, let's take a closer look at what happened with that first rectangle.

If you think back to Chapter 18, you'll remember that, when processing the rectangle list, we're always dealing with two rectangles: one returned in the redraw message and one retrieved from the list. In the case of the rectangle we just processed, the rectangle received from the redraw message was window 1's complete work area. The rectangle returned from the list was the one you just saw filled in (the light gray rectangle at the top of **Figure 2**). The first thing we had to do when we got these rectangles was check whether they overlapped. You'll remember that the call

```
rc_intersect( rec1, rec2 );
```

accomplishes this for us. What I didn't mention in Chapter 18 was that, after the call, the rectangle found in rec2 may not be the same one we started with; it'll actually be a rectangle representing the intersection of the two original rectangles.

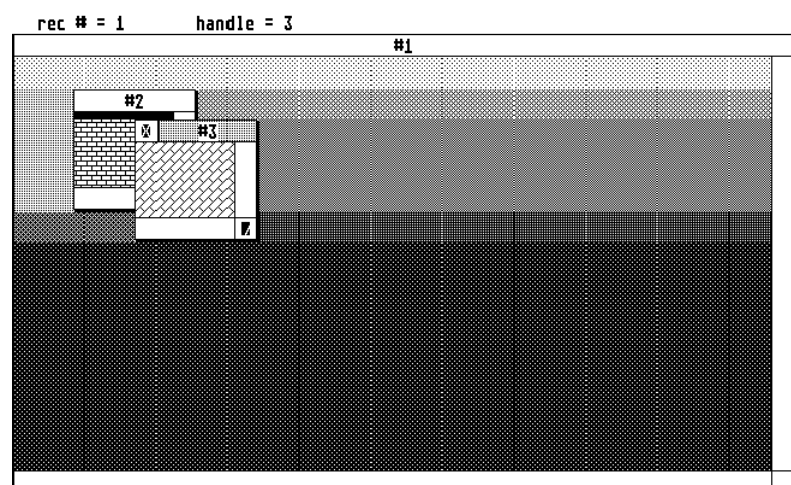
Now, do our first two rectangles intersect? Sure enough! We know we've found an area that must be updated, and we send the rectangle found in rec2 to draw\_interior(), our actual drawing routine. Note that, in this case, the rectangle returned in rec2 was the same rectangle we started out with, because its entire area is in intersection with the one returned in the redraw message.

Look at the information at the top of the screen. We're now ready to process rectangle 2 from window 1's list. Press Return, and this area will be updated. Keep pressing Return, filling in each of window 1's rectangles, until you hear the computer's bell ring, indicating you've reached the end of the rectangle list and completed the processing of the first redraw message.

Now the information at the top of the screen will show that we're about to process a redraw message for window 2. Before you press Return, see if you can figure out how many rectangles it'll take to do the job (hint: nothing should be drawn in window 3's workspace). Figured it out? Everyone who guessed that we need to update two rectangles may look into a mirror and tell yourself how clever you are. Press the return key twice to update window 2. The bell should ring after the second press, leaving us ready to process the redraw message for window 3.

How many rectangles for window 3? One. Window 3 is the topmost window, so has nothing covering it; we need only fill in the entire work area.

Now your screen should look similar to **Figure 2**. Grab window 3 by the mover bar, and drag it into the



**Figure 2**

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

upper-right corner of the screen, as shown in **Figure 3**. Notice that, even though the image of the old window 3 is still on the screen, we didn't have to redraw the window in its new position. Since its complete image was on the screen, GEM did the job for us, blitting the image to its new location.

After blitting the window image, GEM then redrew window 2, not including the workspace. Why not the workspace, too? Before we moved window 3, window 2 was partially hidden, so after the move, GEM didn't know what we wanted to put in the uncovered area. It happily dropped the whole mess -- including a rectangle list -- into our laps. Our status line at the top of the screen now tells us that we've received a redraw message for window 2, and we're waiting to process rectangle 1.

Now think for a minute. (Aw, come on, it won't hurt much.) What portion of window 2 is going to be redrawn when we press the return key? Any guesses? The entire work area, you say? Wrong! Why take the time to update the entire work area when only a small section -- that that was covered by window 3 -- needs redrawing? Because it's easy? Well, yes, but it's just as easy to do it right, because, after our call to `rc_intersect`, we have the area of intersection -- the exact rectangle we need -- in `rec2`. Press Return to see this rectangle get its due.

Since GEM wants to get rid of the rest of window 3's old image, we now have a redraw message for window 1.

Press Return. Hmmmmmm. Nothing happened. If you look at the rectangle number in our status display, though, you'll see that something did happen, because we're now on rectangle 2. So what

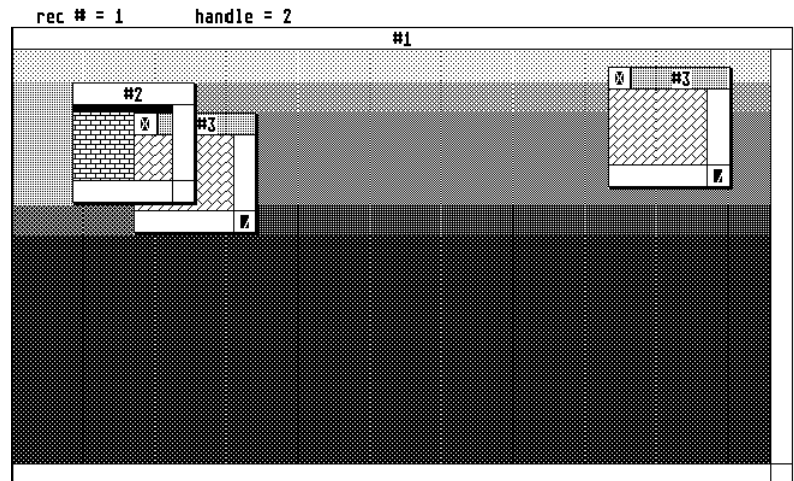


Figure 3

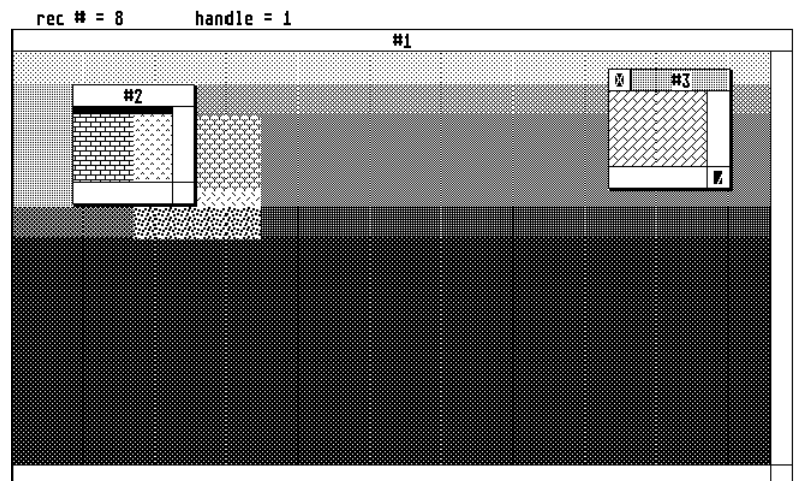


Figure 4

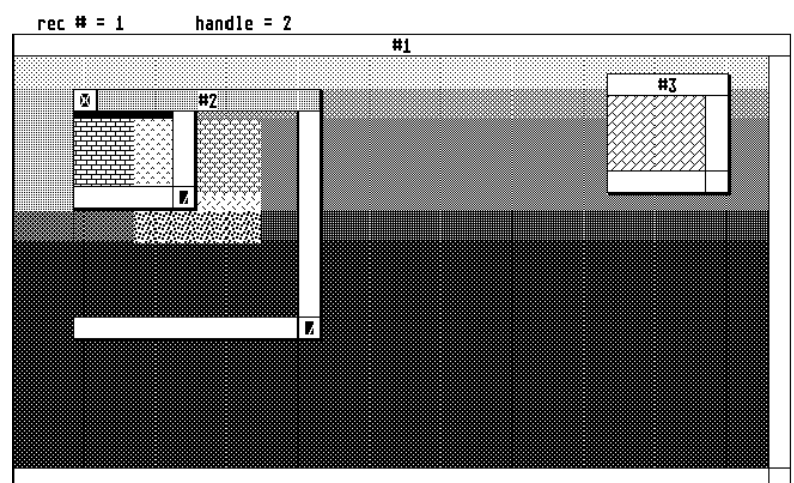


Figure 5



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

happened to rectangle 1? We processed it just as all the others, but because it didn't intersect the dirty area (the rectangle returned in the redraw message) we skipped over it. If you look at the screen, you can see that rectangle didn't need to be redrawn.

Continue pressing Return until the bell rings, watching to see which rectangles are redrawn and which are skipped. Your screen should end up resembling **Figure 4**.

As you may recall, another way to generate a redraw message is to increase the size of a window. Grab window 2's sizer button and enlarge the window as shown in **Figure 5**. Predictably, a redraw message for window 2 is sent. How many rectangles this time? Only one. Sorry, but I guess it was a trick question. Go ahead and press Return. The entire work area of window 2 is redrawn, leaving your screen looking like **Figure 6**. Strange, considering I just said it was a waste of time to redraw portions of a window that didn't need it. That top, left-hand corner didn't need to be redrawn, did it? The fact is, whenever you enlarge a window or make it uppermost, the rectangle returned in rec2 will be the entire work area. I never promised GEM was consistent.

Next step in our experimentation: move window 2 to the lower, right corner of the screen, below window 3 (not over-lapping). Then press Return until the bell rings, watching as window 1 is updated. **Figure 7** shows the results. Now click the mouse pointer on window 1's workspace,

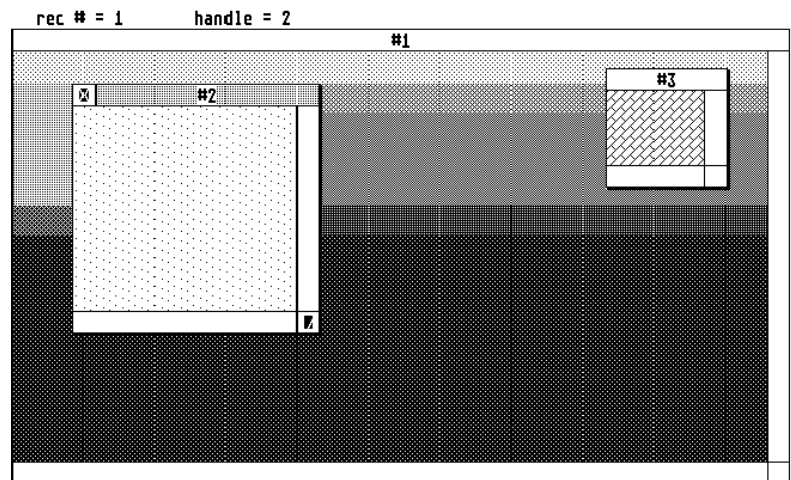


Figure 6

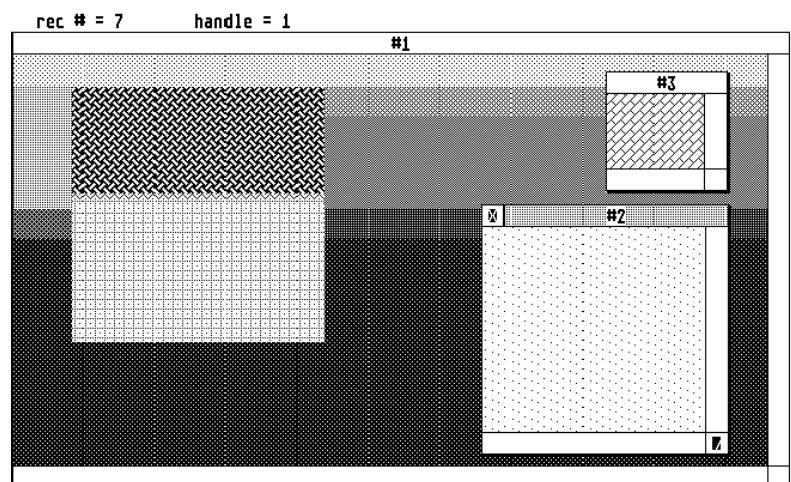


Figure 7

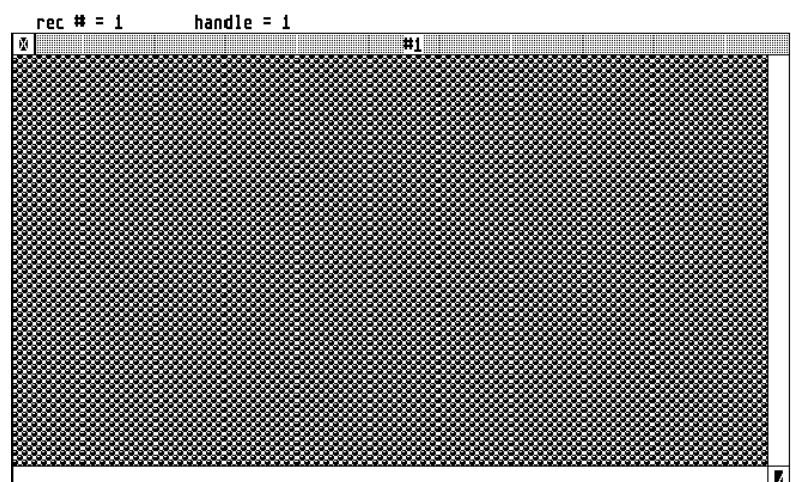


Figure 8



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

making it uppermost. Press Return, and your screen will look like **Figure 8**. Using window 1's sizer button, reduce the window to its smallest size, as shown in **Figure 9**.

Note that, when we reduced window 1, GEM cleaned up the desktop on its own, leaving only the work areas of windows 2 and 3 for us to worry about. As they stand now, both windows 2 and 3 contain information left behind by window 1. Press Return twice to update both windows.

Move window 1 around the desktop, without overlapping any of the other windows or going off screen. We get no redraw messages. GEM is delighted to blit window 1 from one place to another with no help from us.

Now place window 1 in the center of window 2, as shown in **Figure 10**. Still no redraw messages. GEM blitted the window to its new location then erased the old image, just like before. Windows 2 and 3 don't get redraw messages because none of their visible data has been corrupted. Sure, we covered some of it up, but that's not a problem until we move window 1 out of the way again. Do so now, as shown in **Figure 11**. Whoops! GEM did the blit all right, but it left a mess behind. Press Return to conclude this fascinating journey through GEM's rectangle lists, leaving the screen shown in **Figure 12**.

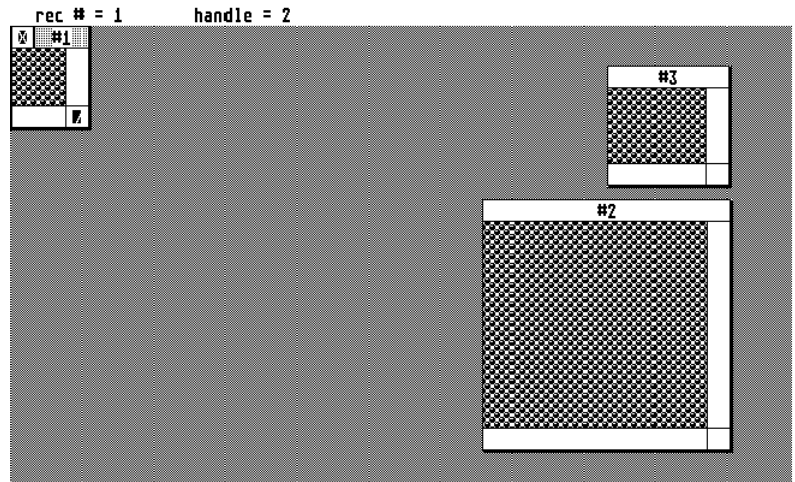


Figure 9

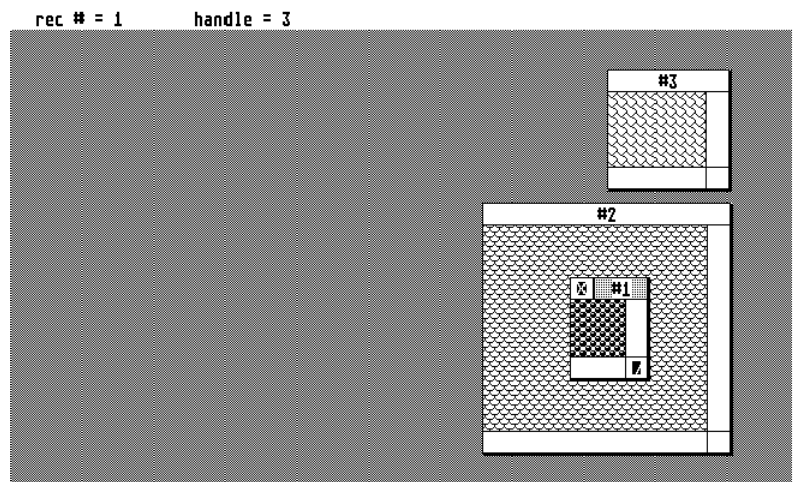


Figure 10

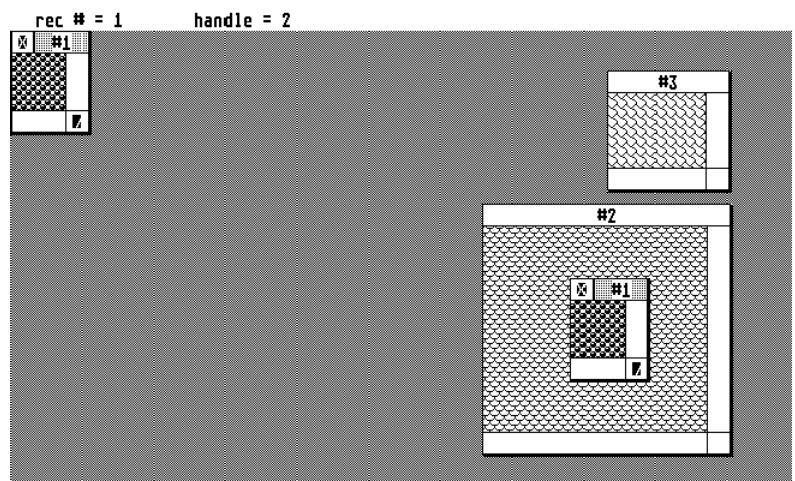


Figure 11

## Out of the Fog

Hopefully, our experiment has taken some of the mystery out of GEM's rectangles and how the system works. Play around with the program all you want, moving and changing windows, all the while watching to see how GEM sets up its rectangle list and how the program processes it. There are an infinite number of possibilities. It'll be a long time before you exhaust them.

If you want the program to run without waiting for a key press, remove the call to `Cconin()` found in the function `do_redraw()`. The information printed at the top of the screen won't do you much good if you do, though. You'll never be able to read it as fast as our program can update those rectangles.

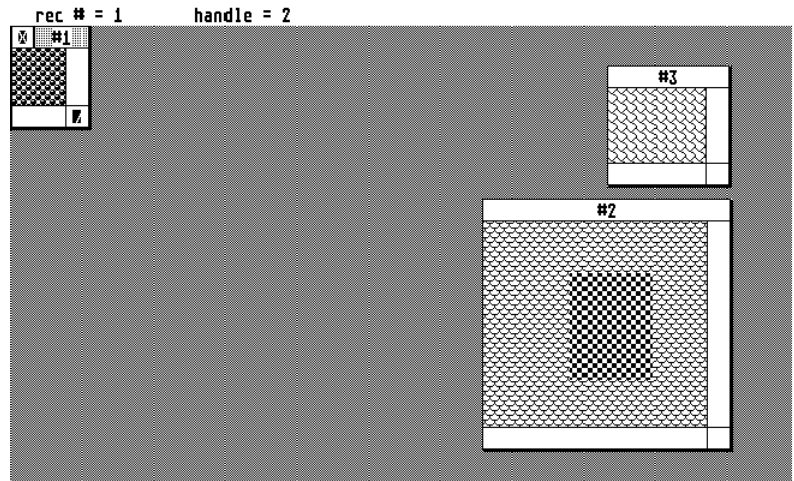


Figure 12

## Sidelines

Before we close up shop for this chapter, there are a couple of things in Listing 1 we ought to go over.

This is the first time we've handled the `WM_TOPPED` message in a program. We get this message from GEM whenever the user clicks the mouse over an inactive window. The call

```
wind_set(msg_buf[3], WF_TOP, 0, 0);
```

is all it takes to "top" the window, make it uppermost and active. Here, `msg_buf[3]` is, as usual, the window's handle; `WF_TOP` is the function we want `wind_set()` to perform, defined in `GEMDEFS.H` as 10; and the two zeros are dummy arguments.

Another nuance worthy of note is the way we're using arrays to cut down the window handling code in the program. Wherever we perform the same function concurrently on all windows, we can use a for loop, the control variable of which becomes an index into an array (one element for each window).

If you look at the function `open_window()` in Listing 1, you'll see the loop that opens the windows and places their handles into the array `w_h[]`. We need only one set of `wind_create()` and `wind_set()` calls. However, when we actually open the windows, we use a separate `wind_open()` call for each window. Why? Because each window is opened at its own set of coordinates, and for the sake of clarity, I decided to "hardcode" those coordinates into the function calls rather than use arrays.

## Another Day, Another Dollar

In Chapter 20 our subject will be...Yes! You guessed it. Windows, again. Maybe we'll figure out how to use those sliders and arrows to change the contents of a window's workspace. Sounds like a good idea to me. How about you?

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```

/*****
/*          C-manship, Listing 1          */
/*          CHAPTER 19                    */
/*          Developed with Megamax C      */
*****/

#include <gemdefs.h>
#include <obdefs.h>
#include <osbind.h>

#define TRUE 1
#define FALSE 0
#define PARTS NAME|CLOSER|MOVER|SIZER
#define MIN_WIDTH 64
#define MIN_HEIGHT 64
#define PATTERN 2
#define BELL 7
#define HIGH 2

/* GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, fullx, fully, fullw, fullh, wrkx, wrky,
wrkw, wrkh, res, char_w, char_h, box_w, box_h, fill;
int msg_buf[8];
int w_h[3];
char *titles[] = {"#1", "#2", "#3"};

main ()
{
    appl_init ();           /* Initialize application.          */
    open_vwork ();          /* Set up workstation.              */
    do_wndw();              /* Go do the window stuff.          */
    v_clsvwk (handle);       /* Close virtual workstation.        */
    appl_exit ();           /* Back to the desktop.              */
}

open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open
    /* a virtual workstation.

    handle = graf_handle ( &char_w, &char_h, &box_w, &box_h);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_wndw ()
{
    /* Clear screen. */
    graf_mouse ( M_OFF, 0L );
    v_clrwk ( handle );
    graf_mouse ( M_ON, 0L );

    /* Find screen resolution. */
    res = Getrez();

    /* Initialize and open our windows. */
    open_window();

    /* Change mouse to arrow, and initialize fill style. */
    graf_mouse ( ARROW, 0L );
    fill = 0;

    /* Receive event messages until the closer is clicked. */
    do {
        evnt_mesag ( msg_buf );
        switch ( msg_buf[0] ) { /* msg_buf[0] is message type. */

            case WM_MOVED:
            case WM_SIZED:
                do_move();
                break;

            case WM_TOPPED:
                wind_set ( msg_buf[3], WF_TOP, 0, 0 );
                break;

            case WM_REDRAW:
                do_redraw ( (GRECT *) &msg_buf[4] );
                break;

        }
    }
    while ( msg_buf[0] != WM_CLOSED );

    /* Close and delete the windows. */
    close_window();
}

do_move()
{
    /* Set window at new location. Also disallow any      */
    /* window sizes less than our minimum allowable size. */

    if ( msg_buf[6] < MIN_WIDTH )
        msg_buf[6] = MIN_WIDTH;
    if ( msg_buf[7] < MIN_HEIGHT )
        msg_buf[7] = MIN_HEIGHT;
    wind_set ( msg_buf[3], WF_CURRXYWH,
        msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7] );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
draw_interior ( clip )
GRECT clip;
{
    int pxy[4], y, x;

    /* Turn mouse off prior to drawing. */
    graf_mouse ( M_OFF, 0L );

    /* Calculate clip rectangle and turn clipping on. */
    set_clip ( TRUE, clip );

    /* Get coordinates of window's work rectangle. */
    wind_get(msg_buf[3],WF_WORKXYWH, &wrkx, &wrky, &wrkw, &wrkh);

    /* Set the color and fill style. */
    vsf_interior ( handle, PATTERN );
    fill += 1;
    if ( fill>24 )
        fill = 1;
    vsf_style ( handle, fill );
    vsf_color ( handle, BLACK );

    /* Draw the background in the window's work area. */
    pxy[0] = wrkx;
    pxy[1] = wrky;
    pxy[2] = wrkx + wrkw - 1;
    pxy[3] = wrky + wrkh - 1;
    vr_recfl ( handle, pxy );

    /* Drawing over, so turn the clipping */
    /* off and turn the mouse back on. */
    set_clip ( FALSE, clip );
    graf_mouse ( M_ON, 0L );
}

do_redraw ( rec1 )
GRECT *rec1;
{
    GRECT rec2;
    int rec_cnt, y;

    /* Init rectangle count, and set y coord for text. */
    rec_cnt = 0;
    if ( res == HIGH )
        y = 15;
    else
        y = 8;

    /* Lock screen for update. */
    wind_update ( BEG_UPDATE );

    /* Get first rectangle from list. */
    wind_get ( msg_buf[3], WF_FIRSTXYWH,
        &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );

    /* Loop through entire rectangle list, */
    /* redrawing where necessary. */
    while ( rec2.g_w && rec2.g_h ) {
        test_print ( "handle", msg_buf[3], 150, y );
        rec_cnt += 1;
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
test_print ( "rec #", rec_cnt, 20, y );
Cconin();
if ( rc_intersect ( rec1, &rec2 ) )
    draw_interior ( rec2 );
wind_get ( msg_buf[3], WF_NEXTXYWH,
           &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );
}
/* Unlock screen after update. */
wind_update ( END_UPDATE );
Cconout ( BELL );
}

set_clip ( flag, rec )
int flag;
GRECT rec;
{
    int pxy[4];

    /* Convert rectangle to pxy coords. */
    pxy[0] = rec.g_x;
    pxy[1] = rec.g_y;
    pxy[2] = rec.g_x + rec.g_w - 1;
    pxy[3] = rec.g_y + rec.g_h - 1;

    /* Turn clipping on or off. */
    vs_clip ( handle, flag, pxy );
}

open_window()
{
    int x;

    /* Find the size of the desktop's work area. */
    wind_get ( 0, WF_WORKXYWH, &fullx, &fully, &fullw, &fullh );

    /* Create the windows. */
    for ( x=0; x<3; ++x ) {
        w_h[x] = wind_create ( PARTS, fullx, fully, fullw, fullh );
        wind_set ( w_h[x], WF_NAME, titles[x], 0, 0 );
    }

    /* Draw the windows. */
    wind_open ( w_h[0], fullx, fully, fullw, fullh );
    wind_open ( w_h[1], 50, 65, 100, 100 );
    wind_open ( w_h[2], 100, 90, 100, 100 );
}

close_window ()
{
    int x;

    /* Close and delete the windows. */
    for ( x=0; x<3; ++x ) {
        wind_close ( w_h[x] );
        wind_delete ( w_h[x] );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
test_print ( label, number, x, y )
int number, x, y;
char *label;
{
    char t[100];

    sprintf ( t , "%s = %d%s", label, number, "    " );
    v_gtext ( handle, x, y, t );
}
```



## CHAPTER 20 - WINDOWS - PART 4 - SLIDERS AND ARROWS

We've stated previously that a window is just a box that allows us to display data in a convenient manner, and the programmer is completely responsible for what is done with the window's work area. One of the things that GEM's windows provide, to help the user manipulate the displayed data, is the slider/arrow system. By moving the sliders or clicking the arrows, the user can "move" the data within the window to any position he likes.

This convenience is paid for by the programmer, however, because, when a slider or arrow is used, GEM does nothing but send a message to the program. It's up to the programmer to decide what to do with the message and how to update the window's display.

When this chapter's program is run, a window will be opened, the directory of the default drive (the one you ran the program from) will be read, and the filenames found there displayed in the window's work area. You may then use the sliders and arrows in their conventional way to move the data within the window. You can also enlarge or shrink the window by dragging (with the mouse, of course) the lower-right corner of the window.

Note that the example program provides only the vertical arrows and sliders. I didn't include the horizontal ones because they're handled almost exactly the same way as their vertical counterparts.

### Getting a Directory

The first item of interest in the sample program is the method with which we can read a disk's directory. The code to accomplish this can be found in the `get_fnames()` function in Listing 1. Let's take a look at that now.

First, we must initialize a couple of variables. We'll be using the integer `p` as an array index, and the integer `files.count` (this is a member of the structure `files`, which is declared near the top of the listing) will contain the number of filenames read from the directory.

Next, we set an important address with the call:

```
Fsetdta( dta );
```

Here, `dta` is a pointer to character data (in our case, the address of the character array `dta[]`). The function `Fsetdta()` is GEMDOS function 0x1a and is declared in `OSBIND.H`. It sets the address of the DTA (Disk Transfer Address), a 44-byte buffer in which the directory data is stored. We supplied the buffer by defining the array `dta[]`.

When we get around to actually reading a filename, the DTA will contain all sorts of useful information, as shown here:

<u>Byte</u>	<u>Contains</u>
0 – 20	For OS use only
21	File attribute
22 – 23	Integer, file time stamp
24 – 25	Integer, file date stamp
26 – 29	Long integer, file size
30 – 43	Filename



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

So let's fill that DTA, shall we? We get the first filename with the call:

```
end = Fsfirst( p, a );
```

Here, p is the pathname you want to use for the file search and a is an integer whose bit settings determine the search's attributes. The function call returns a negative integer in the case of an error (for instance, when there are no more file-names to be read). In the sample listing, we placed the path-name string, " \*.\*," directly into the call. This pathname takes advantage of "wild cards," so the search will match any file.

The attributes set by a are determined as follows:

<u>Bit</u>	<u>Result</u>
0	Search limited to normal files
1	Read-only files included
2	Hidden files included
3	System files included
4	Folder names included
5	Subdirectory files included

By setting bits 0 and 4 in the search attributes argument, we'll read all the file and folder names from the root directory (and, because of the pathname, these will be read from the default drive), just as if you had just opened the drive from the desktop.

Now that we've got the first filename, we set up a while loop to get the rest, as well as to process the filenames into the form we need, later storing them into the two-dimensional array files.fnames[[]]. All we're doing in the processing is making sure each entry in the filename array is exactly 15 characters long: the filename padded with spaces and ending with a null.

The rest of the filenames are retrieved, one by one, with the call:

```
end = Fsnext ( );
```

This function (GEMDOS 0x4f), defined in OSBIND.H, also returns a negative value for an error condition. We continue executing the while loop until end becomes negative, or files.count becomes greater than MAX.

### Slipping and Sliding

Now that we've got all those filenames stored, we're ready to open our window. We've done all this stuff before; there's nothing new here, except the use of the integer top to keep track of where in the filename list the window's data display is to start (remember, the topmost filename in the display won't necessarily be the first one in our list), and setting up the sliders.

Once we open the window, we need to set the size and position of the slider. Our function calc\_slid() takes care of this, and requires three integers as arguments: the handle of the window; the total

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

number of text lines (in this case, the number of filenames in the list); and the total width of the text in columns.

The function first calls `wind_get()` to get the size of the window's work area, then calculates the number of lines and columns that'll fit that area. The size of the slider is then calculated like this:

```
size = 1000*lines_avail/line_count;
```

The size of the slider can range anywhere from 1 to 1000, and represents the relative portion of the document displayed. By dividing the number of lines available in the window by the number of lines in the "document," we end up with a value representing the portion of the document that'll fit the window. Multiplying this value by 1000 will give us the equivalent size of the slider.

For example, let's say we have 10 lines to display, but the window can hold only 6. Dividing 6 by 10 gives us .6 -- the portion of the document that the window can display. When we multiply this value by 1000, we get 600 -- the slider's relative size. Keep in mind that, for greater accuracy, you might want to use floating-point math, rather than integer math.

Once the size of the slider is calculated, it's set with the call:

```
wind_set(w_h,WF_VSLSIZE,size,0,0,0);
```

Here, `w_h` is the window's handle, `WF_VSLSIZE` is the function's operation flag (`WF_HSLSIZE` for horizontal sliders) as defined in `GEMDEFS.H`, `size` is an integer value between 1 and 1000, and the three zeroes are unused arguments.

Next, we need to calculate the slider's position within its track, which is also a value between 1 and 1000. The following calculation, done with floating-point math, the result of which is cast to an integer, does this:

```
pos=(int)(float)top/(float)(line_cnt-lines_avail)*1000;
```

The integer `top` is the number of the uppermost displayed line in the window, `line_cnt` is the total number of lines in the document, and `lines_avail` is the number of lines that'll fit the window.

The slider's position is then set with the call:

```
wind_set(w_h,WF_VSLIDE,pos,0,0,0);
```

Here, `w_h` is the window's handle, `WF_VSLIDE` is the function's operation flag (`WF_HSLIDE` for horizontal sliders) as defined in `GEMDEFS.H`, `pos` is an integer between 1 and 1000, and the three zeroes are unused arguments.

### Me and My Arrow

Whenever the user clicks on one of the arrows, or in the slider's tracks, the program will receive a `WM_ARROWED` message. This message is a little different from the others we've worked with. It actually contains a sub-message, which GEM stores in `msg_buf[4]`. This forces us to do a little more work before we can take care of the user's request. This extra work is tackled in the function

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

`do_arrow()`, where we use the value stored in `msg_buf[4]` to determine exactly what the user wants to do.

If the user has clicked on the down arrow, `msg_buf[4]` will contain a `WA_DNLINE` message, and program execution will continue with the function `do_dnline()`.

Before we look at that function, let's stop and think about what we're doing. What exactly is the user requesting when he clicks on the down arrow? Generally, it means that we must move the window's display one unit upward and the slider one unit downward. Exactly what that "unit" is depends on your application. If our window contained some sort of graphic information, such as a map, a unit could be anything from a single scan line to dozens of scan lines. Luckily, our decision is a little easier. We're working with text, so the obvious unit of measurement is a text line.

We know now what we want to do, but how are we going to go about it? The easy way of updating the display would be just to increment `top`, then call `draw_interior()` to redraw the window's work area. The problem with that is that it's too slow, too sloppy. We're going to need something much more elegant.

When you're working with the desktop's file windows, the text displays move neatly upward one line each time you click on the down arrow; you can't see the redrawing.

Almost all the information we need is on the screen, right? Only the bottom of the new list isn't shown. You know what that means? We can update most of the window using raster operations. All we have to do is "blit" the area of the window from the second filename down to a position one text line higher, then fill in the bottom with the new filename. And that's exactly what we do in `do_dnline`.

Let's run through that function now. First, we use `wind_get()` to get the size of the window's work area. Then we calculate the number of text lines that'll fit. Armed with that information, we use an if statement to make sure we don't bother updating the window if there are no filenames left to display. In other words, if the last file in our list is already shown in the window, we want to ignore the request to scroll downward.

If our calculations show that the arrow message is okay to process, we increment `top`, calculate where in our list of filenames we'll find the new data that needs to be displayed, set clipping to on, turn off the mouse and do our raster stuff.

Because -- depending on the size of the window -- we may have a partial filename displayed at the bottom, we actually have to print two filenames, and if we're at the end of the list, the second filename should be a line of spaces. Here's what happens in our function:

If the window is a size in which an even number of filenames will fit, our code will blit the display up one line, then print at the bottom the next two filenames in the list. The first will go at the very bottom line, and the second won't appear at all, because we're trying to print it outside of the clipping area.

Now, let's take a case where the size of the window allows a partial filename to show at the bottom. When we do our calculations for the number of lines that'll fit the window, the result is an integer, which means the decimal portion has been truncated (i.e., rounded down to the nearest integer). Because of this, only the number of complete lines that'll fit the window are counted. That's why we always want to print two filenames, since the second may represent one that is only partially visible.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

In the case of a partially displayed filename, we'll actually have to print one and a half filenames. Sound tricky? Naw. The clipping rectangle makes this little complication easy to handle. We just print two filenames, and anything that lies outside of the area gets clipped off, leaving us with a partial filename (the second one we printed) at the bottom.

Once we've gotten our display written, we have to recalculate the position of the slider. This is done the same way we did it when we first opened our window, with a call to our function `calc_slid()`.

That completes the processing of the down-arrow request. The function `do_upline()` does the same thing for the `WA_UPLINE` message as `do_dnline()` did for the `WA_DNLINE` message, except the process is reversed -- and a little simpler.

In `do_upline()` we're rastering the window's work area down one line, then printing the next filename (thinking backwards) at the top. Because we'll never have a partial line here, we don't have the extra complications we had with `do_dnline()`.

### Paging All Sliders

Two other messages we may receive from the original `WM_ARROWED` message (not including messages for horizontal sliders and arrows, which are `WA_LFPAGE`, `WA_RTPAGE`, `WA_LFLINE`, and `WA_RTLINE`) are `WA_DNPAGE` and `WA_UPPAGE`. These are sent to us by GEM whenever the user clicks in the slider's track, indicating that he wants the next "page" of information, the next windowful of lines following that already shown in the window.

Because the information we want displayed in the window is not available anywhere on the screen, we can't use raster operations. We have to do it the sloppy way: figure out the new line for the top of the window, then call our function `draw_interior()` to do the work. But there are a couple of things we have to watch out for when calculating the new top. We have to make sure that top doesn't end up less than zero and that it doesn't become a value that will cause our display to go beyond the bottom of our text.

Let's look at the function `do_dnpage()`. First, we use a `wind_get()` call to find the size of the window's work area. Then we calculate the number of lines that'll fit. Since we want to move down that number of lines in the document, all we have to do to calculate the new value for top is to add `lines_avail` to its existing value. The only thing to check for is the bottom of the document. If our new top, plus the number of lines in the display equal a value greater than the total number of lines in the document, we have to set back the value of top in such a way that the last line of our document will also be the bottom line of the window. Not too tricky, really.

The function `do_uppage()` (which executes when we receive a `WA_UPPAGE` message) works in the same manner, except we subtract `lines_avail` from top, then check to make sure top isn't less than zero.

### Anywhere You Like

Another way the user can change the window's display is to grab the slider with the mouse pointer and move it to a new position. When this occurs we get a `WM_VSLID` message from GEM (or `WM_HSLID`, if it's a horizontal slider).

This message is almost as easy to handle as the `WA_UPPAGE` and `WA_DNPAGE` messages. The key thing to know here is that the slider's new position is returned in `msg_buf[4]`. To find the

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

corresponding position in our document, all we have to do is get the size of the window's work area, calculate the number of lines that'll fit the window, then perform the following calculation:

```
top=msg_buf[4] * (line_cnt - lines_avail) / 1000;
```

We then set the slider to its new position with the call:

```
wind_set(w_h,WF_VSLIDE,msg_buf[4],0,0,0);
```

A call to our function `draw_interior()` (which will use the new value calculated for `top`) completes the task.

### An Important Note

You should be aware that, many times, the calculations for finding the size and position of the sliders may have to be done with 32-bit math (using long integers) to avoid overflow problems, or with floating-point math when you need greater accuracy. Ignoring these possibilities could give you some perplexing results. If ever you find your sliders behaving mysteriously -- and you're sure your logic is correct -- check your math.

### Program Listing #1

```
/******  
/*          C-manship, Listing 1          */  
/*          CHAPTER 20                    */  
/*          Developed with Megamax C      */  
/******  
  
#include <gemdefs.h>  
#include <obdefs.h>  
#include <gembind.h>  
#include <osbind.h>  
  
#define TRUE 1  
#define FALSE 0  
#define PARTS NAME | CLOSER | SIZER | UPARROW | DNARROW | VSLIDE  
#define MAX 50  
#define SOLID 1  
#define MIN_WIDTH 64  
#define MIN_HEIGHT 64  
  
/* GEM arrays. */  
int work_in[11],  
work_out[57],  
contrl[12],  
intin[128],  
ptsin[128],  
intout[128],  
ptsout[128];  
  
/* Global variables. */  
int handle, w_h, top,  
fullx, fully, fullw, fullh,  
char_w, char_h, box_w, box_h,  
wrkx, wrky, wrkw, wrkh;
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Message buffer. */
int msg_buf[8];

struct {
    char fnames[MAX][15]; /* Char array for filenames. */
    int count;             /* Number of filenames read. */
} files;

/* Window title. */
char *title = "C-manship";

main ()
{
    appl_init ();           /* Initialize application.          */
    open_vwork ();          /* Set up workstation.          */
    do_wndw();              /* Go do the window stuff.      */
    v_clsvwk (handle);      /* Close virtual workstation.    */
    appl_exit ();           /* Back to the desktop.          */
}

open_vwork ()
{
    int i;

    handle = graf_handle ( &char_w, &char_h, &box_w, &box_h);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

do_wndw ()
{
    top = 0;

    get_fnames ();
    wind_get ( 0, WF_WORKXYWH, &fullx, &fully, &fullw, &fullh );
    w_h = wind_create ( PARTS, fullx, fully, fullw, fullh );
    wind_set ( w_h, WF_NAME, title, 0, 0 );
    wind_open ( w_h, 100, 20, 150, 151 );
    calc_slid ( w_h, files.count, 14 );
    graf_mouse ( ARROW, 0L );

    do {
        evnt_mesag ( msg_buf );
        switch ( msg_buf[0] ) { /* msg_buf[0] is message type. */

            case WM_SIZED:
                do_move ();
                break;

            case WM_ARROWED:
                do_arrow ();
                break;

            case WM_VSLID:
                do_vslide ();
                break;
        }
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        case WM_REDRAW:
            do_redraw ( (GRECT *) &msg_buf[4] );
            break;
    }
}
while ( msg_buf[0] != WM_CLOSED );

wind_close ( w_h );
wind_delete ( w_h );
}

do_arrow ()
{
    switch ( msg_buf[4] ) {

        case WA_UPPAGE:
            do_uppage ();
            break;

        case WA_DNPAGE:
            do_dnpage ();
            break;

        case WA_UPLINE:
            do_upline ();
            break;

        case WA_DNLINE:
            do_dnline ();
            break;

    }
}

do_vslide ()
{
    GRECT r;
    int lines_avail;

    wind_get ( w_h, WF_WORKXYWH, &r.g_x, &r.g_y, &r.g_w, &r.g_h );
    lines_avail = r.g_h / char_h;
    top = msg_buf[4] * (files.count - lines_avail) / 1000;
    wind_set ( w_h, WF_VSLIDE, msg_buf[4], 0, 0, 0 );
    draw_interior ( r );
}

do_uppage ()
{
    GRECT r;
    int lines_avail;

    wind_get(w_h, WF_WORKXYWH, &r.g_x, &r.g_y, &r.g_w, &r.g_h);
    lines_avail = r.g_h / char_h;
    top -= lines_avail;
    if ( top < 0 )
        top = 0;
    draw_interior ( r );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_dnpage ()
{
    GRECT r;
    int lines_avail;

    wind_get ( w_h, WF_WORKXYWH, &r.g_x, &r.g_y, &r.g_w, &r.g_h
    );
    lines_avail = r.g_h / char_h;
    top += lines_avail;
    if ( top > files.count - lines_avail )
        top = files.count - lines_avail;
    draw_interior ( r );
}

do_upline ()
{
    FDB s, d;
    GRECT r;
    int pxy[8];

    if ( top != 0 ) {
        top -= 1;
        wind_get(w_h, WF_WORKXYWH, &r.g_x, &r.g_y, &r.g_w, &r.g_h);
        set_clip ( TRUE, r );
        graf_mouse ( M_OFF, 0L );
        s.fd_addr = 0L;
        d.fd_addr = 0L;
        pxy[0] = r.g_x;
        pxy[1] = r.g_y + 1;
        pxy[2] = r.g_x + r.g_w;
        pxy[3] = r.g_y + r.g_h - char_h - 1;
        pxy[4] = r.g_x;
        pxy[5] = r.g_y + char_h + 1;
        pxy[6] = r.g_x + r.g_w;
        pxy[7] = r.g_y + r.g_h - 1;
        vro_cpyfm ( handle, S_ONLY, pxy, &s, &d );
        v_gtext ( handle, r.g_x+char_w, r.g_y+char_h,
            &files.fnames[top][0] );
        set_clip ( FALSE, r );
        calc_slid ( w_h, files.count, 14 );
        graf_mouse ( M_ON, 0L );
    }
}

do_dnline ()
{
    FDB s, d;
    GRECT r;
    int pxy[8];
    int lines_avail, index;

    wind_get ( w_h, WF_WORKXYWH, &r.g_x, &r.g_y, &r.g_w, &r.g_h );
    lines_avail = r.g_h / char_h;
    if ( (top + lines_avail) < files.count ) {
        top += 1;
        index = top + lines_avail - 1;
        set_clip ( TRUE, r );
        graf_mouse ( M_OFF, 0L );
        s.fd_addr = 0L;
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
d.fd_addr = 0L;
pxy[0] = r.g_x;
pxy[1] = r.g_y + char_h + 1;
pxy[2] = r.g_x + r.g_w;
pxy[3] = r.g_y + r.g_h - 1;
pxy[4] = r.g_x;
pxy[5] = r.g_y + 1;
pxy[6] = r.g_x + r.g_w;
pxy[7] = r.g_y + r.g_h - char_h - 1;
vro_cpyfm ( handle, S_ONLY, pxy, &s, &d );
v_gtext ( handle, r.g_x+char_w, r.g_y+(lines_avail)*char_h,
          &files.fnames[index][0] );
if ( index != files.count-1 )
    v_gtext ( handle, r.g_x+char_w,
              r.g_y+(lines_avail)*char_h+char_h,
              &files.fnames[index+1][0] );
else
    v_gtext ( handle, r.g_x+char_w,
              r.g_y+(lines_avail)*char_h+char_h,
              " " );
set_clip ( FALSE, r );
calc_slid ( w_h, files.count, 14 );
graf_mouse ( M_ON, 0L );
}

}

get_fnames ()
{
    char dta[44];
    int end, p, x, null_found;

    p = 0;
    files.count = 0;
    Fsetdta ( dta );
    end = Fsfirst ( "*.*", 17 );

    while ( (end > -1) && (files.count <= MAX) ) {
        null_found = FALSE;
        files.count += 1;
        for ( x=0; x<14; ++x ) {
            if ( dta[30+x] == 0 )
                null_found = TRUE;
            if ( null_found )
                dta[30+x] = ' ';
            files.fnames[p][x] = dta[30+x];
        }
        files.fnames[p][14] = 0;
        p += 1;
        end = Fsnext ();
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
calc_slid ( w_h, line_cnt, col_cnt )
int w_h, line_cnt, col_cnt;
{
    int lines_avail, cols_avail, vslid_siz, pos;

    wind_get ( w_h, WF_WORKXYWH, &wrkx, &wrky, &wrkw, &wrkh );
    lines_avail = wrkh / char_h;
    cols_avail = wrkw / char_w;
    vslid_siz = 1000 * lines_avail / line_cnt;
    wind_set ( w_h, WF_VSLSIZE, vslid_siz, 0, 0, 0 );
    pos = (int) ( (float)(top) ) /
        ( (float)(files.count - lines_avail) ) * 1000;
    wind_set ( w_h, WF_VSLIDE, pos, 0, 0, 0 );
}

do_move()
{
    if ( msg_buf[6] < MIN_WIDTH )
        msg_buf[6] = MIN_WIDTH;
    if ( msg_buf[7] < MIN_HEIGHT )
        msg_buf[7] = MIN_HEIGHT;
    wind_set ( msg_buf[3], WF_CURRXYWH,
        msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7] );
    calc_slid ( w_h, files.count, 14 );
}

draw_interior ( clip )
GRECT clip;
{
    int pxy[4];
    int x, lines_avail, lines_shown;

    graf_mouse ( M_OFF, 0L );
    set_clip ( TRUE, clip );
    wind_get(msg_buf[3], WF_WORKXYWH, &wrkx, &wrky, &wrkw, &wrkh);

    vsf_interior ( handle, SOLID );
    vsf_color ( handle, WHITE );
    pxy[0] = wrkx;
    pxy[1] = wrky;
    pxy[2] = wrkx + wrkw - 1;
    pxy[3] = wrky + wrkh - 1;
    vr_recfl ( handle, pxy );

    lines_avail = wrkh / char_h;
    lines_shown = files.count - top;
    if ( lines_avail > lines_shown ) {
        top = files.count - lines_avail;
        if ( top < 0 )
            top = 0;
    }

    for ( x=top; x<files.count; ++x )
        v_gtext ( handle, wrkx+8, wrky+(x+1-top)*char_h,
            &files.fnames[x][0] );

    set_clip ( FALSE, clip );
    calc_slid ( w_h, files.count, 14 );
    graf_mouse ( M_ON, 0L );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
do_redraw ( rec1 )
GRECT *rec1;
{

    GRECT rec2;

    wind_update ( BEG_UPDATE );
    wind_get ( msg_buf[3], WF_FIRSTXYWH,
               &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );

    while ( rec2.g_w && rec2.g_h ) {
        if ( rc_intersect ( rec1, &rec2 ) )
            draw_interior ( rec2 );
        wind_get ( msg_buf[3], WF_NEXTXYWH,
                   &rec2.g_x, &rec2.g_y, &rec2.g_w, &rec2.g_h );
    }

    wind_update ( END_UPDATE );
}

set_clip ( flag, rec )
int flag;
GRECT rec;
{
    int pxy[4];

    pxy[0] = rec.g_x;
    pxy[1] = rec.g_y;
    pxy[2] = rec.g_x + rec.g_w - 1;
    pxy[3] = rec.g_y + rec.g_h - 1;
    vs_clip ( handle, flag, pxy );
}
```

## CHAPTER 21 - D.E.G.A.S. PICTURE VIEWER

Everyone who's getting tired of studying GEM's windows please raise your hand. Yeah, that's what I thought. Okay, it's time to take up a new subject, something that, though it'll give you a lot of information on how your computer works, it won't give you a headache trying to understand it.

One of the more useful things about the ST is the ability to have many screens of data in memory at once and flip between them as you like. I thought this would be a good subject to tackle since it enables us to see not only how we can accomplish "screen flipping" (which is really a simple process), but how to apply some of the other techniques we've learned, such as the programming of file selectors. We'll also take a look at some new information, such as the D.E.G.A.S. picture file format.

We're going to load two D.E.G.A.S. format pictures into memory, and then use an alert box to choose which picture to view. We'll have to tell the program which files to load, so the first thing the program will do is bring up a file selector. Use it in the normal way to select two D.E.G.A.S. pictures for loading.

While you're doing this, keep in mind that the program presented here is a stripped down model. In other words, it doesn't incorporate much in the way of error checking. In fact, it'll let you load just about any type of file into memory, whether it's D.E.G.A.S. or not. So do your own error checking, and make sure you're selecting the right type of file.

If you click on the file selector's Cancel button for either picture, or if the program gets a file error, you'll be returned to the desktop. Once you get two files loaded, an alert box with three buttons will appear. Clicking on the first button will cause the first loaded picture to be displayed. Clicking on the second button will show the second picture. The Quit button should be used to leave the program and return to the desktop. Once a picture is displayed on the screen, clicking the left button will bring the alert box back, allowing you to make another choice or quit the program.

### Hey! That Space is Reserved!

The first step in getting our picture files loaded into the computer is figuring out where we're going to store them. We need a lot of space -- 32K for each picture -- and we have to make sure that, wherever we store the picture information, it doesn't get in the way of our program or its data. Also, since we're going to be displaying a couple of different screens, we have to make sure we store the address of the original screen, as well as its color palette, so we can restore it when the program's finished.

Take a look at the function `init_screens()` in Listing 1. The first thing we do here is store the desktop's color palette with the line:

```
for (x=0; x<16; desk_palette[x++] = Setcolor(x,-1));
```

The function `Setcolor()` is an XBIOS function and is defined in the `OSBIND.H` file. This function requires two integers as arguments. The first is the color you want to change (from 0 to 15), and the second is the color to change it to.

Colors on the ST are formed by mixing the correct proportions of red, green, and blue, each of which can have a value from 0 (minimum) to 7 (maximum). The color value for blue is placed in the first

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

nibble (four bits) of the integer; the value for green is placed in the second nibble; and the value for red is placed in the third. This works out well in hexadecimal: 0x007 is the brightest blue, 0x070 is the brightest green, and 0x700 is the brightest red. White is all the values at their maximum (0x777), while black is formed by setting all colors to the minimum (0x000). By combining the three basic colors in varying intensities, we can conjure up any of the ST's 512 possible colors.

But all that is beside the point (go ahead and boo; I deserve it). We don't want to change the colors (at least, not yet); we want to know what value they're currently set at, so we can store them for later retrieval. One thing I didn't tell you about the Setcolor() function is that it always returns a color's previous setting (its color value before we changed it). If we make the second argument a negative number, it won't change the color register at all; it'll just return the color's setting.

Now you can see how the above code segment works. We use a for loop to step through all 16 possible elements of the color palette, calling Setcolor() in each iteration with a color value of -1, in order to have the current color returned to us. Each of these colors is stored in the array desk\_palette[], where they'll be when we're ready to restore the desktop's colors. Now that we've gotten that taken care of, we have to store the address of the desktop's screen (we do want to get back there eventually, you know). This line will do the job:

```
scrn = Physbase();
```

Here, the variable scrn is a long integer that'll hold the address returned from Physbase(). The function Physbase() returns the address of the physical screen, the area of memory currently displayed on your monitor. The function Logbase() returns the address of the logical screen, an area of memory where all output to the screen is to go.

In most cases, the physical and logical screens are in the same location. For example, as I'm writing this article, I can see the new text I'm typing appearing on the screen. That means that the displayed screen and the one the program is sending text to are at the same address. Sometimes, though, we may find it handy to be able to direct screen data to a different place in memory, so we can update the screen "behind the user's back." Once the logical screen has been set up the way we want it, we can simply flip to it, creating the illusion of the screen being instantly updated. We'll see how all this works a little later.

Now that we know where our physical screen is, we're ready to allocate some memory for a couple of logical screens. We can have only one physical screen, but we can have as many logical screens as you can store in memory. In the function init\_screens(), we set up a while loop that first allocates a block of screen memory, then calls a function to read the picture data into it. To allocate a block of memory, we use the call:

```
addr = Malloc(bytes);
```

Here, the pointer addr will hold the address of the block of memory, and the long integer bytes is the number of bytes we wish to reserve. This function returns a 0 if the amount of memory we've requested is unavailable. One variation on the Malloc() call, making bytes equal to -1L, will return the total amount of memory available.

You've probably noticed, though, that our call to Malloc() in Listing 1 looks quite a bit more complex:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
pic[x] = (Malloc(32768L) & 0xffffffff00 ) + 0x0100;
```

Am I just trying to show off? No; not at all.

First, even though `pic[x]` doesn't look like a pointer, it is. In fact, `pic[]` is an array of pointers (actually, long integers, but for our use that amounts to the same thing). For programming purposes, it's very convenient to store the addresses of our screens in an array, so that we can get at them easily with some sort of loop.

Next comes that strange looking `Malloc()` call. It looks strange to you because there's one little detail I've yet to mention, the fact that the ST's screen memory must always start on a 256-byte boundary. And, since `Malloc()` doesn't know or care about this little requirement, it's up to us to smooth things over.

The first step in getting to a safe 256-byte boundary is to use C's AND operator to mask off the eight right-most bits of the address, using the hex value `0xFFFFF00` as our mask. This value has every bit set except the right-most eight. The AND operator compares the bits of two values, returning a true (1) when both bits are on and a false (0) when either or both the bits are off. What that means for us is that every bit we have off in the mask will result in a 0 in the bit it's being ANDed with. Let's say the address returned from `Malloc()` was `0x034CC3E2`. After ANDing it with our mask, we'd have `0x034CC300`, which is an address on a 256-byte boundary.

But even though we're now on the boundary we wanted, it's not a safe boundary. Why? Because the address we have now is lower than the one returned from `Malloc()`. That means we're no longer in the area we just reserved; we're actually before it. If we try to load data there, we'll probably end up clomping all over our program -- and getting a delightful string of bombs up on the screen.

That's why, after completing the AND operation, we add `0x00000100` (256 decimal) to the resultant address. That pushes it back into our reserved area.

"Ah!" you cry, in that smug manner you use when you think you've caught the professor with his foot in it. "If we're pushing the address forward, doesn't that mean that, when we load our picture data, the last few bytes will be placed outside the reserved area, beyond the other end?"

Nope. You see, we've reserved 32,768 bytes (that's a full 32K), and we really need only 32,000 bytes for our picture data. When people tell you that screen memory on the ST is 32K, they're not telling you the whole truth. It's actually a bit short of a full 32K. We just like to round it off when we speak. (You ever hear people refer to the SF314 disk drive as a one meg drive, even though you can only store 720,000 bytes on the disk? Same idea.)

One thing we do have to watch out for, though, is how we handle any subsequent calls to `Malloc()`, because it doesn't know we've finagled the address it gave us the first time around. The next time we allocate some memory, we have to remember to add the same amount to the returned address, or we're sure to make digital footprints in the previous areas. And digital footprints often result in the Big Kablooey. (In our case, since we're using those areas only for a screen display, we'd simply end up with some funny looking pictures.)

Okay, we've got the memory we need to store our pictures. Now let's think about how we're going to load them. The first step is to get the picture's filename, and the obvious way to do that is with GEM's handy file selector box. Included in Listing 1 is a function called `select_file()`. This is a generic

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

file-selector routine that I came up with that you can use in your own programs. It handles some of the minor details for you, allowing you to just call a file selector and have the complete filename (including the path) returned to you. (You're welcome.)

If you look at the function `get_pic()`, you'll see how we get started. First, because it's required by `select_file()`, we have to come up with a default filename. This will appear tacked on to the end of the pathname field in the file selector and allows us to narrow the number of files shown when the box first comes up. In our example, we start with the string `"* .PI "` then finish the default name by adding the proper D.E.G.A.S. resolution indicator. Adding the ASCII value of `"1"` to the value returned from `Getrez()` performs that trick.

Our file selector function, `select_file()`, returns the complete chosen filename and the button that was clicked to exit the file selector. The call to the function looks like this:

```
select_file(path, file, default, flag);
```

Here, `path` is a pointer to a 64-byte character array and is where the function will store the completed filename. The pointer `file` is the address of a 13-byte character array that'll hold the selected filename after the call to `fsel_input()`. We may also, before the function call, store a filename here that we want to appear in the filename field of the file selector box. The pointer `default` contains the address of a string containing the text we want added to the selector's pathname field. And finally, `flag` is a Boolean value that tells the function whether we want the string pointed to by `file` to appear in the file selector's file field.

It sounds complicated at first, but I've found that using this function is a lot easier than trying to remember how to handle the file selector each time I need it.

As I mentioned before, `select_file()` returns the value of the file selector button that was clicked. Strangely enough (or perhaps it was done purposely), these values also correspond to obvious Boolean values: the Cancel button returns 0, and the OK button returns 1. In the function `get_pic()`, we use this returned value as a Boolean to evaluate an if statement. In other words, if the user clicks on the file selector's Cancel button for either of the two files we're going to be loading, we'll know not to read the file and, instead, exit the program.

If the user clicks the file selector's OK button, we call the function `read_degas()` to attempt to load the file chosen. If the file loads all right, this function will return a value of `TRUE`. If an error is encountered (maybe the file doesn't exist), it returns a value of `FALSE`. We use this returned value in another if statement to determine whether we should continue or return to the desktop. In a full-scale application program, you would want to give the user a message if you ran into an error, but for the sake of brevity, we've kept things to a minimum in the example program.

Let's turn our attention now to `read_degas()`. It's here that we actually read the selected picture file into memory. This function needs to know which picture we're loading and the complete filename. The first thing we must do is open the file, but we have to make sure we open it to "read binary." We covered the `open()` function previously, but we didn't talk about the `O_BINARY` flag. When we open the file with this flag (it's defined at the top of the listing as 8192), we're telling the system that we want the file read from the disk in an untranslated form, as a continuous block of data, rather than a series of lines ending with carriage returns and line feeds.

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Before we go any further, we need to discuss the format in which D.E.G.A.S. (the unsqueezed variety) pictures are saved to disk. If you've ever looked at a disk directory containing these picture files, you've undoubtedly noticed that they are 32,034 bytes. In order to get the picture up on the screen properly, we have to know what each of these bytes are.

The first two bytes of a D.E.G.A.S. file indicate the picture's resolution. It's interpreted as a word value: 0x0000, 0x0001, or 0x0002, for low, medium or high resolution, respectively. Normally, we'd want to check the resolution of the picture against the computer's current resolution, to make sure they match, and if they don't, give the user an error message. But, as I said before, for the sake of brevity, we're going to do things quick and sloppy and just throw away those two bytes after we've read them.

The next 32 bytes (16 words) are the picture's color palette. That we don't want to throw away; we want to read it into the array we've set up for storing this information.

Finally, the last 32,000 bytes are the actual picture data. We read that information into the area of memory starting at the address stored in the appropriate element of the `pic[]` array.

Now that we've got all the data read, we close the file and return a value of `TRUE` to the calling function. Notice that, in the function `read_degas()`, we're using the value returned from the `open()` function in an if statement. Doing this guarantees, in the case of a file error, that we skip over all the subsequent file handling code, and just return from the function a value of `FALSE`.

Once we get two picture files loaded okay, program execution is turned over to the function `flip_screens()`, where we get a chance to actually view the pictures. We begin by calling up an alert box with three buttons, one button for each picture plus a Quit button. We use the value returned from the alert box as an index into the `pic[]` array, where the pointers to the screens are stored. To flip between the different screens, we use the call:

```
Setscreen(log,phys,res);
```

Here, `log` is the address of the logical screen, `phys` is the address of the physical screen, and `res` is the screen resolution we want to switch to. If we don't want to switch screen resolutions, we just give `res` a negative value. In fact, all parameters with a negative value will be ignored.

In most cases, you would set both the logical and physical screen to the same address. As for the resolution, you'll almost always want to leave it unchanged (use a negative value) because GEM is never informed of resolution changes, and that can lead to nasty complications.

After flipping the screen, we wait for a mouse button click using a call to `evnt_button()`, after which we bring up the alert box to get another choice. We keep displaying the selected picture until the Quit button is clicked, after which we close things up and return to the desktop.

### Putting It Back Where We Found It

But we can't just go blithely on our way, returning to the desktop by just closing the virtual workstation and calling `apl_exit()`. Nosiree. We've got cleaning up to do. We've allocated a bunch of memory for our picture files, and before we leave, we have to give it back. Not a tough thing to do. The following call will return a block of memory (one that was allocated with `Malloc()`) to the system:



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
Mfree(adr);
```

The pointer `adr` is the address of the block we want to de-allocate. You need to make a separate call to `Mfree()` for each block allocated, and you must return the blocks in the reverse order you allocated them.

Once we've returned all the memory to the system, we can exit the program in the usual manner. You can see all this being done in Listing 1 in the function `clean_up()`.

### Mission Complete

Now that you know how all this screen flipping stuff works, why don't you modify the program so that you can load more than two D.E.G.A.S. pictures? Use the mouse to flip through them. When you get to the last of the pictures, use an alert box to ask the user whether he wants to see the pictures again or quit. Practice makes perfect!

### Program Listing #1

```
/* **** */
/*      C-manship, Listing 1      */
/*      CHAPTER 21                */
/*      Developed with Megamax C  */
/* **** */

#include <osbind.h>

#define TRUE      1
#define FALSE    0
#define O_BINARY  8192
#define QUIT      3
#define LEFT_BUTTON 1
#define DOWN      1

/* The usual required GEM global arrays */
int work_in[11],
work_out[57],
pxyarray[10],
contrl[12],
intin[128],
ptsin[128],
intout[128],
ptsout[128];

/* Global variables */
int handle, dum;

long pic[2], /* Pointers to logical screens. */
scrn; /* Pointer to physical screen. */

int desk_palette[16]; /* Desktop color palette. */
int pic_palette[2][16]; /* Picture color palettes. */

main ()
{
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
    appl_init ();          /* Initialize application.          */
    open_vwork ();         /* Set up workstation.          */
    do_pictures ();        /* Go do the picture stuff.     */
    clean_up ();           /* Get everything back to normal.*/
    appl_exit ();          /* Back to the desktop.         */
}

open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open
    /* a virtual workstation. */

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

do_pictures ()
{
    /* If the pictures are loaded okay, */
    /* then allow user to view them.    */

    if ( init_screens () )
        flip_screens ();
}

init_screens ()
{
    int x,          /* Index variable.          */
        okay;      /* File load flag.         */

    /* Store the desktop's color palette. */
    for ( x=0; x<16; desk_palette[x++]=Setcolor (x, -1) );

    /* Store the address of the desktop's screen. */
    scrn = Physbase ();

    /* Reserve memory for pictures and load them */
    /* into the allotted space, storing pointers */
    /* to them in the pic[] array.               */

    okay = TRUE;
    x = 0;
    while ( (okay == TRUE) && (x < 2) ) {
        pic[x] = ( Malloc (32768L) & 0xffffffff00 ) + 0x0100;
        okay = get_pic ( x++ );
    }
    return ( okay );
}

flip_screens ()
{
    int choice; /* Button number clicked in alert box. */
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
choice = 1;
/* View pictures until QUIT button is clicked. */
while ( choice != QUIT ) {

    /* Call up alert box to get user's picture choice. */
    choice = form_alert ( 0, "[2][Choose picture to \
                                view][One|Two|Quit]" );

    /* We only want to show a picture if the */
    /* QUIT button hasn't been clicked.      */

    if ( choice != QUIT ) {

        /* Set the screen to show the chosen picture. */
        Setscreen ( pic[choice-1], pic[choice-1], -1 );

        /* Set the palette to the picture's settings. */
        Setpalette ( &pic_palette[choice-1][0] );

        /* Wait for a button click. */
        evnt_button(1,LEFT_BUTTON,DOWN,&dum,&dum,&dum,&dum);

    }
}
}
```

```
get_pic ( num )
int num; /* Number of picture to load. */
{
    char path[64], /* Storage for picture's pathname. */
        file[13], /* Storage for picture's filename. */
        pictype[6]; /* Storage for default picture filename. */

    /* Build default picture filename. */
    strcpy ( pictype, "*.PI " );
    pictype[4] = Getrez () + '1';

    /* If file selector CANCEL button wasn't clicked, */
    /* read the chosen DEGAS file into memory. If an */
    /* error is returned, the program will abort.     */

    if ( select_file ( path, file, pictype, FALSE ) )
        if ( read_degas ( num, path ) )
            return ( TRUE );
        else
            return ( FALSE );
    else
        return ( FALSE );
}
```

```
read_degas ( num, pathname )
int num; /* Picture number to read. */
char *pathname; /* Picture's pathname. */
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
{
    int f_h,          /* File handle. */
        buf[10]; /* Temp buffer for unused bytes. */

    /* Process file only if no error is returned when opening. */
    if ( (f_h = open ( pathname, O_BINARY )) != -1 ) {

        /* First two bytes is resolution data. */
        read ( f_h, buf, 2 );

        /* Next 32 bytes (16 words) is the color palette. */
        read ( f_h, &pic_palette[num][0], 32 );

        /* Finally, we have 32K of picture data. */
        read ( f_h, pic[num], 32000 );

        /* Close file and tell calling function */
        /* that everything went all right. */

        close ( f_h );
        return ( TRUE );
    }
    /* In case of error opening the file. */
    else
        return ( FALSE );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
select_file ( path, fnme, deflt, display)
char *path, /* Address for path storage. */
*fnme, /* Address for filename storage. */
*deflt; /* Address of default filename. */
int display; /* Display default filename? */
{
    int x, /* Loop variable. */
        choice, /* Button clicked from file selector box. */
        len; /* String length. */
    char ch; /* Temp character storage. */

    /* Clear filename string if not to be displayed. */
    if ( display == FALSE )
        for ( x=0; x<13; fnme[x++] = '\0' );

    /* Build file selector box pathname. */
    Dgetpath ( path, 0 );
    len = strlen ( path );
    path[len] = '\\';
    strcpy ( &path[ len + 1 ], deflt );

    /* Call up file selector box to get user's choice. */
    fsel_input ( path, fnme, &choice );

    /* Find last significant character in pathname in
    /* order to delete the filename portion of the path. */

    len = strlen ( path );
    x = len-1;
    while ( path[x] != '\\ ' && path[x] != ':' && x > 0 )
        --x;
    strcpy ( &path[x+1], fnme );

    return ( choice );
}

clean_up ()
{
    /* Setscreen back to desktop. */
    Setscreen ( scrn, scrn, -1 );

    /* Restore original color palette. */
    Setpalette ( desk_palette );

    /* Return the reserved memory back to the system. */
    Mfree ( pic[1] );
    Mfree ( pic[0] );

    /* Close virtual workstation. */
    v_clsvwk ();
}
```

## CHAPTER 22 - THE INTERNAL CLOCK/CALENDAR

This time around we'll tackle a subject we've managed to avoid so far -- the Atari ST's real-time clock. Actually, "avoid" probably isn't a good word to use here, since reading and setting the ST's clock is really not very hard. You just need to become proficient with handling data in a bitwise fashion rather than as words or bytes. And as we'll soon see, attaining those skills will not require an inordinate amount of effort, and those same skills will be a valuable addition to your future C programming projects.

But first you should get this chapter's sample program up and running, and that involves a little more work than usual. You're going to need to create the dialog box shown in Figure 1. There's two ways you can do this. The first is to type in Listing 3 with ST BASIC (make sure you check your typing with ST Check, see Appendix A), and then run it. The program will create the necessary resource file for you.

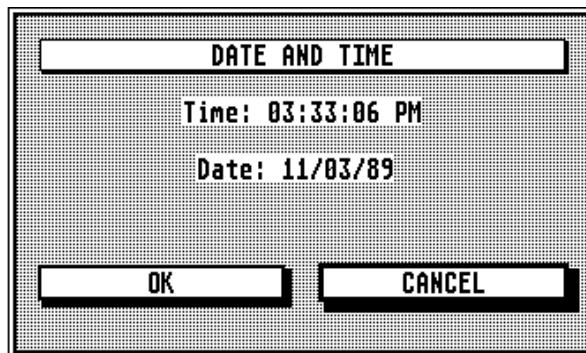


Figure 1

The other way to produce the dialog box is to use a resource construction program. The dialog contains only four objects, but they must be created and named carefully. The objects are the editable text fields that show the time and date and the two exit buttons. If you want to create your own dialog box, here is the information you need to know:

The dialog box itself is named DATEDIAL. The OK button is named OKBUTN and is simply a shadowed, exit button. The CANCEL button is named CANBUTN and is a shadowed, exit button, but it is also set as the default. The "Time" field is an unboxed, editable text string that is named TIMEFLD. Its ptmplt, pvalid, and ptext strings, shown in order, are:

```
Time: __:__:__ __
999999AA
000000AM
```

The "Date" field is also an unboxed, editable text field. It's named DATEFLD, and its ptmplt, pvalid, and ptext strings, also in their respective order, are:

# C-MANSHIP COMPLETE – by CLAYTON WALNUT

Date: \_\_/\_\_/\_\_  
999999  
000000

That's all you need to know to reproduce the dialog box shown in Figure 1 (except that you must name the RSC file DATE.RSC). If all of this sounds confusing, then either review Chapters 14 and 15, which cover dialog boxes, or use Listing 3 to create your resource file.

Now that you've created your resource file, you may type in Listing 1 and compile it. If you used the ST BASIC program to create your resource file, you must also type in Listing 2, before you try to compile the program, and save it to disk as DATE.H.

Now run the program. If you've got the resource file in the same directory as the program, you'll see the dialog box shown in Figure 1. (If you're missing your resource file, the program will warn you, and then return to the Desktop.) The time and date shown in the dialog box are the current settings of your system clock. If you'd like to reset the clock, just edit the time and date strings and click on the OK button. If the strings you've entered are valid, the program will reset your system's clock and return to the Desktop. Otherwise you'll receive an error alert box, and you'll have to reenter the information. If you're satisfied with the time as it is, click on the CANCEL button or simply press Return.

## Computer Dating

Let's take a look at Listing 2 and see what's going on here. Most of what's being done in the program you should already be familiar with. For instance, we long ago discussed how to load a resource file and get a dialog box up on the screen. In case you've gotten a little rusty, the program listing is commented enough so that you can easily see what's being done.

Take a look at the function `get_date()`. It's here that we retrieve the system date from the computer's clock and convert it into a form that we can use in our dialog box. First we get the date with the call

```
date = Tgetdate();
```

where `date` is an integer. The function `Tgetdate()` is defined in your `OSBIND.H` file as `gemdos(0x2a)` and returns all the information we need to figure out the current date. Piece of cake, right? Not quite. If your noodle is active today, you'll remember that our dialog box displays the date by month, day, and year. However, the `Tgetdate()` call returned only one value. See a problem here?

In order to simplify the process of storing and passing the system date, the people who designed your ST's OS decided to cram all the information we need to extract the current month, day, and year into a single integer; and if you're really on the ball today, you'll realize that that means we're going to have to finagle some bits in order to separate the information we want from the information we don't care about.

The system date returned from the `Tgetdate()` function is formatted in the following manner: Bits 0 to 4 (counting from right to left, remember) contain the day, bits 5 to 8 contain the month, and bits 9 to 15 contain the year since 1980, or, in other words, the current year minus 80. Figure 2 illustrates

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

this format. What we have to do is figure out a way to extract the day, month, and year from the entire integer. Thank heavens for bitwise operations!

Year							Month				Day				
0	0	0	1	0	0	0	1	0	1	1	1	0	1	0	1
Hour						Minutes						Seconds			
0	1	1	1	1	1	1	0	1	1	0	0	1	1	0	1

FIGURE 2

### A Bit About Bits

The C programming language supplies us with five operators that can manipulate the bits that make up a piece of data. Some of them you've seen before; a couple of them are new to you. Those operators are:

&	Bitwise AND
^	Bitwise exclusive OR
	Bitwise inclusive OR
<<	Left shift
>>	Right shift

We've already had experience with the bitwise AND and bitwise inclusive OR operators. The AND operator compares the bits of two values and places into the result a 1 in any position where both bits of the compared values are set and a 0 in every other case. This allows us to "mask" out the bits in a value that we're not interested in. We create a mask by setting the bits of the mask that correspond to the information we wish to extract from the value of interest. Every other bit is turned off.

Let's say we wanted to get the value of the low byte of a word. We would create a mask that looked like this: 0000000011111111. Suppose the value from which we want to extract information is called number and the binary representation of number is 0010110101100110. The calculation would look like this:

```
0010110101100110 <-- number
0000000011111111 <-- Mask
-----
000000001100110 <-- Result
```

As you can see, the result contains only the bit values we wanted to retain. In a C program the above calculation would be written as follows:



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
result=number&0x00ff
```

The inclusive OR operator is almost the opposite of the AND operator. Rather than extracting portions of a value, it lets us insert them. When you inclusive OR two values together, the result will have a bit set wherever there was a bit set in either one or both of the compared values. Let's say we wanted to merge the values contained in two variables called var1 and var2. The binary representation of var1 is 0000000010101011, and the binary representation of var2 is 1101101100000000. The inclusive OR operation looks like this:

```
0000000010101011 <-- num1
1101101100000000 <-- num2
-----
1101101110101011 <-- result
```

You can see from the result that we've combined the low byte of num1 with the high byte of num2. There's one important thing you must be aware of, though. This combining of values will work only when the positions that will hold the merged value all contain zeroes. In other words, we would not get the proper result in the above operation if the high byte of num1 was not cleared:

```
1111111110101011 <-- num1
1101101100000000 <-- num2
-----
1111111110101011 <-- result
```

The same problem would crop up if the low byte of num2 hadn't been cleared.

A bitwise exclusive OR is similar except that the result will contain a 1 only in those positions where either one or the other bit is set. If both bits are set or both bits are cleared, the result will be a 0.

The left shift operator causes the bits in the first operand to be shifted to the left the number of times found in the second operand. The right-hand, emptied bits will be filled with zeroes. For instance, let's take a variable named num that contains the binary value 1011010110101101. If we were to perform the operation num<<5, the result would be 1011010110100000.

The right shift operator works much the same way, except that the emptied left-hand bits may or may not be zero-filled, depending on the machine and data type you're using. The rule is if the data type is unsigned, you are guaranteed to get a zero fill; otherwise, the left-hand bits may be filled with the value of the sign bit (the most significant bit).

### But What About the Date?

So here we are, finally back to the original problem of extracting the day, month, and year from the single integer returned by the Tgetdate() call. Think about it for a minute. Have you got it figured out yet? No?

Let me explain, then. The information we need to get the day is contained in bits 0 through 4, right? So, what we need to do is mask out bits 5 through 15. Then our result will contain only the value stored in the lower five bits -- and that value is current day. (Of course, whether this value matches

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

your calendar or not depends on whether your system clock has been set properly.) Let's say the value returned from `Tgetdate()` is the one shown in Figure 2. Figure 3a then illustrates the operation involved in extracting the day.

```
0001000101110101
& 0x001f
0000000000010101  21
```

FIGURE 3a

First, we create a mask to AND with our integer, a mask that will clear bits 5 through 15, while at the same time maintain the values of the lower five bits. The proper mask is 0000000000011111 in binary or 0x001f in hexadecimal. (Note that it's much easier to create your mask in binary first then convert it to hexadecimal. That way you can easily see which bits you're setting.) Then all we have to do is AND the system date with our mask. In Listing 1, the line that does this is:

```
day = date & 0x001f;
```

Say! That was pretty easy, wasn't it? The next step is to get the month, but we run into a little complication with that one. If we were to just AND out the bits we weren't interested in, we'd end up with the value 0000000101100000 which translates to a decimal value of 352! Ouch! Aren't there only 12 months? To get the value we really want, we have to move the four bits we're interested in to the right five places. Sounds to me like a good job for the right shift operator.

But let's perform the shifting first and then mask out the unnecessary bits. That way we're sure we get no garbage in the upper bits as a result of the shift operation. Theoretically, it would work either way, since our sign bit will be a zero. But I learned a long time ago that, when it comes to computers, you can only trust what you know. And I know that if I do the AND operation last, I'll have the result I'm looking for. In Listing 1, the line that gets us our month looks like this:

```
mnth = (date >> 5) & 0x000f;
```

This operation is illustrated in Figure 3b.

```
0001000101110101
>>5
0000000010001011
& 0x000f
0000000000001011  11
```

FIGURE 3b

Finally, to get the year, we have to perform the same operation, only we'll be shifting the bits down nine places instead of five, and we'll be using a different mask because we're interested in a different number of bits. Figure 3c illustrates this operation, and the equivalent line in Listing 1 looks like this:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
year = ((date >> 9) & 0x007f) + 80;
```

```
0001000101110101
>>9
0000000000001000
& 0x007f
0000000000001000 8
```

FIGURE 3c

Although Figure 3c doesn't show it, we have to remember to add 80 to the result because, as I mentioned before, the year returned from the `Tgetdate()` call is the year since 1980.

### Some Timely Information

Now let's look at the function `get_time()` in Listing 1. We get the system time with the call

```
time = Tgettime();
```

where `time` is an integer. Bits 0 through 4 of this value will contain the seconds divided by two, bits 5 through 10 will contain the minutes, and bits 11 through 15 will contain the hour. We can extract this information in the same way we calculated the date -- by shifting the bits in which we're interested all the way to the right, and then using a mask and the AND operation to clear the bits in which we're not interested.

I don't think we need to go into a lot of detail here, but there is one thing I want to mention -- something that we didn't have to deal with when we calculated the date. The value for the hour portion of the system time is in 24-hour format; that is, it'll be a value from 0 to 23. Values from 0 to 11 represent the hours of midnight to 11 a.m., and the values from 12 to 23 represent the hours from noon to 11 p.m. In order to make the time more readable, our function `get_time()` does some converting so that the time will be displayed in the manner we're most used to seeing it. (Of course, if you're in the military, you may not approve of this conversion!)

Also, keep in mind that the value for the seconds is the number of seconds divided by two. This means that you must multiply times two the value for the seconds that was returned by the `Tgettime()` function. This also means that your ST's clock is only accurate to the nearest even second.

### Setting the Time and Date

Setting the system's time and date requires only that we reverse the process we used to get the time and date. Instead of using an AND operation, we'll be using the inclusive OR, and instead of shifting bits to the right, we'll be shifting them to the left.

In Listing 1, the function `set_date()` handles both the setting of the time and the setting of the date. To set the time, we use this call:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
Tsettime( time );
```

where the integer time uses the same bit format we studied when we discussed the Tgettime() call. To set the date, we use this call:

```
Tsetdate( date );
```

where the integer date used the same bit format we learned about when we discussed the Tgetdate() call. These functions are defined in your OSBIND.H file. Let's take just a quick look at how we prepare the integers for these calls. Let's use time as our example this time around. Suppose the time we wanted to set the system clock to was 14:36:34 (that's 2:36 p.m. for those of you who could never get the hang of a 24-hour clock). Setting the seconds is easy:

```
time = seconds;
```

Here, seconds is equal to 17. (Remember that the number of seconds must be divided by two; that's the only way the designers of the OS could get the time to fit into an integer.)

Now time contains the binary value 0000000000010001, which equals 17 in decimal. Our value for minutes is 36, which is 0000000000100100 in binary. We have to move this information up into bits 5 through 10. The operation minutes=minutes<<5 gives us a result of 0000010010000000 which is exactly what we want.

Now we have to combine the seconds (the value of which is already stored in time) with the minutes. The operation time=time|minutes does the trick handily. On a binary level that operation looks like this:

```
00000000000010001 <-- time (seconds)
0000010010000000 <-- minutes
-----
0000010010010001 <-- time (seconds and minutes)
```

To add the hours, we do the same sort of operation, only we'll be shifting the value for hours 11 places to the left. I might also add that it doesn't matter in what order we store the seconds, minutes and hours, as long as we follow the general procedure outlined above. If you look at Listing 1, you'll see that I started with the hours instead of the seconds.

### All Ashore Who's Going Ashore

That about covers it. As you peruse this chapter's program, you may come across a couple of functions that aren't familiar to you. If so, just look them up in your manual. There's nothing complicated with any of them, and you should be easily able to figure out how everything in the sample program works.

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```
/******  
/*C-manship, Listing 1*/  
/* CHAPTER 22 */  
/*Developed with Megamax C*/  
/******  
#include <osbind.h>  
#include <gemdefs.h>  
#include <obdefs.h>  
#include "date.h"  
  
#define TRUE 1  
#define FALSE 0  
#define MATCH 0  
  
/* GEM arrays */  
int work_in[11],  
work_out[57],  
contrl[12],  
intin[128],  
ptsin[128],  
intout[128],  
ptsout[128];  
  
int handle, /* Application handle. */  
dum; /* Dummy storage.*/  
  
char *get_tedinfo_str ();  
  
/******  
* Main program.  
*****/  
main ()  
{  
    appl_init (); /* Init application. */  
    open_vwork (); /* Open virtual workstation. */  
    do_date (); /* Go do our thing.*/  
    rsrc_free (); /* Release resource memory.*/  
    v_clsvwk ( handle ); /* Close virtual workstation.*/  
    appl_exit (); /* Back to the desktop.*/  
}  
  
/******  
* do_date ()  
* Loads the resource file and handles the dialog box.  
*****/  
do_date ()  
{  
    int dial_x, /* Dialog's X coord.*/  
        dial_y, /* Dialog's Y coord.*/  
        dial_w, /* Dialog's width.*/  
        dial_h, /* Dialog's height. */  
        choice, /* Exit button clicked from dialog. */  
        okay; /* Flag indicating if entered date valid. */  
  
    OBJECT *datedial_addr; /* Address of dialog box. */
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
char date_str[8],/* String to hold date. */
time_str[10]; /* String to hold time. */

char *string; /* Temp string pointer. */

graf_mouse ( ARROW, 0L );

/* Load resource file. */
if ( !rsrc_load ( "\date.rsc" ) )
    form_alert ( 1, "[1][date.rsc missing!][OK]" );

else {

    /* Get address of dialog and init time and date strings. */
    rsrc_gaddr ( R_TREE, DATEDIAL, &datedial_addr );
    get_time ( time_str );
    get_date ( date_str );

    /* Copy system time and date into dialog box, */
    string = get_tedinfo_str ( datedial_addr, TIMEFLD );
    strcpy ( string, time_str );
    string = get_tedinfo_str ( datedial_addr, DATEFLD );
    strcpy ( string, date_str );

    /* Prepare dialog box for drawing, and init flag. */
    form_center(datedial_addr,&dial_x,&dial_y,&dial_w,&dial_h);
    form_dial(FMD_START,0,0,10,10,dial_x,dial_y,dial_w,dial_h);
    okay = TRUE;

    /* This loop repeats until the user clicks CANCEL */
    /* or until the user enters a valid date and clicks OK. */
    do {
        /* Draw dialog and allow user to manipulate it. */
        objc_draw(datedial_addr,0,8,dial_x,dial_y,dial_w,dial_h);
        choice = form_do ( datedial_addr, TIMEFLD );

        /* Reset the state of the chosen button. */
        datedial_addr[choice].ob_state = SHADOWED;

        /* If OK clicked, check entered date and set system */
        /* date if date entered is valid, */
        if ( choice == OKBUTN ) {
            okay = chk_date ( datedial_addr );
            if ( okay )
                set_date ( datedial_addr );
        }
    }
    while ( okay == FALSE && choice == OKBUTN );

    /* Get rid of the dialog box. */
    form_dial(FMD_FINISH,0,0,10,10,dial_x,dial_y,dial_w,dial_h);
}

}

/*****
* chk_date ()
* Examines the strings in dialog for a valid date
* and valid time.
*****/
chk_date ( dial_addr )
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
OBJECT *dial_addr; /* Address of dialog box. */
{
    int mnth, day, year, /* Date and time broken into integers.*/
        hour, min, sec,
        space, /* Flag for bad chars in time string.*/
        okay, /* Flag indicating valid time & date. */
        x; /* Loop variable. */

    char m[3], d[3], y[3], /* Date & time as character arrays. */
        h[3], mn[3], s[3],
        ap[3]; /* "AM" or "PM" */

    char *date_str, /* Pointer to string containing date. */
        *time_str; /* Pointer to string containing time. */

    /* Init date and time integers to error condition. */
    mnth = day = year = hour = min = sec = -1;

    /* Get address of string containing date. */
    date_str = get_tedinfo_str ( dial_addr, DATEFLD );

    /* Convert date string to integer format. */
    if ( strlen ( date_str ) == 6 ) {
        strncpy ( m, date_str, 2 );
        m[2] = 0;
        strncpy ( d, &date_str[2], 2 );
        d[2] = 0;
        strncpy ( y, &date_str[4], 2 );
        y[2] = 0;
        mnth = atoi ( m );
        day = atoi ( d );
        year = atoi ( y );
    }

    /* Get address of string containing time. */
    time_str = get_tedinfo_str ( dial_addr, TIMEFLD );

    /* Check for spaces in time string. */
    space = FALSE;
    for ( x=0; x<6; ++x )
        if ( time_str[x] == ' ' )
            space = TRUE;

    /* Convert time string to integer format. */
    if ( (strlen ( time_str ) == 8) && !space ) {
        strncpy ( h, time_str, 2 );
        h[2] = 0;
        strncpy ( mn, &time_str[2], 2 );
        mn[2] = 0;
        strncpy ( s, &time_str[4], 2 );
        s[2] = 0;
        hour = atoi ( h );
        min = atoi ( mn );
        sec = atoi ( s );
        strcpy ( ap, &time_str[6] );
    }

    /* Examine time and date for validity. */
    if ( mnth < 1 | mnth >12 | day < 1 | day > 31
        | year < 0 | year > 99 | hour < 0 | hour > 23 | min < 0
        | min > 59 | sec < 0 | sec > 59 |
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
((strcmp(ap, "AM") != MATCH) && (strcmp(ap, "PM") != MATCH)) {
    okay = FALSE;
    form_alert ( 1, "[1][Date or time not valid!][CONTINUE]"
);
}
else
    okay = TRUE;

return ( okay );
}

/*****
* set_date ()
* Sets the system time and date to the values
* entered into the dialog box.
*****/
set_date ( dial_addr )
OBJECT *dial_addr; /* Address of dialog box. */
{
    char *string; /* Temporary string pointer. */
    char s[3]; /* Temporary string storage. */
    int h, /* Work variable. */
        time, /* Time in system format. */
        date; /* Date in system format. */

    /* Get address of string containing time. */
    string = get_tedinfo_str ( dial_addr, TIMEFLD );

    /* Extract "hours" portion and convert to integer. */
    strncpy ( s, string, 2 );
    h = atoi ( s );

    /* Adjust hour to the 24-hour clock format. */
    if ( (strcmp ( &string[6], "PM" ) == MATCH) && (h != 12) )
        h += 12;
    if ( (strcmp ( &string[6], "AM" ) == MATCH) && (h == 12) )
        h = 0;

    /* Shift bits into the proper position and place them */
    /* into the time integer. */
    h = h << 11;
    time = h;

    /* Get the "minutes" portion, convert to integer, */
    /* shift bits and place them into the time integer. */
    strncpy ( s, &string[2], 2 );
    h = atoi ( s );
    h = h << 5;
    time = time | h;

    /* Process the "seconds" portion of the time. */
    strncpy ( s, &string[4], 2 );
    h = atoi ( s ) / 2;
    time = time | h;

    /* Set the system clock to the new time. */
    Tsettime ( time );

    /* Get the address of the string containing the date. */
    string = get_tedinfo_str ( dial_addr, DATEFLD );
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Process the "month" portion. */
strncpy ( s, string, 2 );
h = atoi ( s );
h = h << 5;
date = h;
/* Process the "day" portion. */
strncpy ( s, &string[2], 2 );
h = atoi ( s );
date = date | h;

/* Process the "year" portion. */
strncpy ( s, &string[4] );
h = atoi ( s ) - 80;
h = h << 9;
date = date | h;

/* Set the system to clock to the new date. */
Tsetdate ( date );
}

/*****
* get_time ()
* Gets system time and converts it to string format.
*****/
get_time ( string )
char *string; /* Pointer to string in which to store time. */
{
    int time, /* Time in system format. */
        hour, min, sec; /* Time broken down into separate ints. */

    char s[3]; /* "AM" or "PM" */

    /* Get system time & break down into individual components. */
    time = Tgettime ();
    sec = ( time & 0x001f ) * 2;
    min = ( time >> 5 ) & 0x003f;
    hour = ( time >> 11 ) & 0x001f;

    /* Convert system 24-hour format to regular 12-hour format. */
    if ( hour > 11 ) {
        strcpy ( s, "PM" );
        if ( hour > 12 )
            hour -= 12;
    }
    else {
        strcpy ( s, "AM" );
        if ( hour == 0 )
            hour = 12;
    }

    /* Convert and add hours to time string. */
    if ( hour < 10 ) {
        string[0] = '0';
        sprintf ( &string[1], "%d", hour );
    }
    else
        sprintf ( string, "%d", hour );

    /* Convert and add minutes to time string. */
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
if ( min < 10 ) {
    string[2] = '0';
    sprintf ( &string[3], "%d", min );
}
else
    sprintf ( &string[2], "%d", min );

/* Convert and add seconds to time string. */
if ( sec < 10 ) {
    string[4] = '0';
    sprintf ( &string[5], "%d", sec );
}
else
    sprintf ( &string[4], "%d", sec );

/* Add "AM" or "PM" to time string. */
strcpy ( &string[6], s );
}

/*****
* get_date ()
* Gets system date and converts it to string format.
*****/
get_date ( string )
char *string; /* Pointer to string that will contain the date. */
{
    int date, /* Date in system format. */
        day, mnth, year; /* Date broken into components. */

    /* Get system date and convert to individual components. */
    date = Tgetdate ();
    day = date & 0x001f;
    mnth = (date >> 5) & 0x000f;
    year = ((date >> 9) & 0x007f) + 80;
    year = year % 100;

    /* Convert and add "months" portion to date string. */
    if ( mnth < 10 ) {
        string[0] = '0';
        sprintf ( &string[1], "%d", mnth );
    }
    else
        sprintf ( string, "%d", mnth );

    /* convert and add "days" portion to date string. */
    if ( day < 10 ) {
        string[2] = '0';
        sprintf ( &string[3], "%d", day );
    }
    else
        sprintf ( &string[2], "%d", day );

    /* Convert and add "year" portion to date string. */
    sprintf ( &string[4], "%d", year );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* get_tedinfo_str ()
* Returns a pointer to an editable string in a
* dialog box.
*****/
char *get_tedinfo_str ( tree, object )
OBJECT *tree; /* Address of dialog box. */
int object; /* Object that contains the string. */
{
    TEDINFO *ob_tedinfo; /* Pointer to a tedinfo structure. */

    ob_tedinfo = (TEDINFO *) tree[object].ob_spec;
    return ( ob_tedinfo->te_ptext );
}

/*****
* open_vwork ()
* Opens a virtual workstation.
*****/
open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open*/
    /* a virtual workstation. */

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

```

### Program Listing #2

```

#define DATEDIAL 0/* TREE */
#define TIMEFLD 2/* OBJECT in TREE #0 */
#define DATEFLD 3/* OBJECT in TREE #0 */
#define OKBUTN 4/* OBJECT in TREE #0 */
#define CANBUTN 5/* OBJECT in TREE #0 */

```

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #3

ST Basic

```
100 OPEN"R",#1,"A:DATE.RSC",16:FIELD#1,16 AS B$
110 A$="":FOR I=1 TO 16:READ V$:IF V$="*" THEN 140
120 A=VAL("&H"+V$):PRINT "*";A$=A$+CHR$(A):NEXT
130 LSET B$=A$:R=R+1:PUT 1,R:GOTO 110
140 CLOSE 1:PRINT:PRINT "ALL DONE!"
1000 data 00,00,00,D4,00,80,00,80,00,80,00,00,00,24,00,80
1010 data 00,00,01,64,00,06,00,01,00,03,00,00,00,00,00,00
1020 data 00,00,01,68,44,41,54,45,20,41,4E,44,20,54,49,4D
1030 data 45,00,00,00,30,30,30,30,30,30,41,4D,00,54,69,6D
1040 data 65,3A,20,5F,5F,3A,5F,5F,3A,5F,5F,20,5F,5F,00,39
1050 data 39,39,39,39,39,41,41,00,30,30,30,30,30,30,00,44
1060 data 61,74,65,3A,20,5F,5F,2F,5F,5F,2F,5F,5F,00,39,39
1070 data 39,39,39,39,00,4F,4B,00,43,41,4E,43,45,4C,00,00
1080 data 00,00,00,24,00,00,00,32,00,00,00,33,00,03,00,06
1090 data 00,02,11,80,00,00,FF,FF,00,0E,00,01,00,00,00,34
1100 data 00,00,00,3D,00,00,00,4F,00,03,00,06,00,00,11,80
1110 data 00,00,FF,FF,00,09,00,12,00,00,00,58,00,00,00,5F
1120 data 00,00,00,6E,00,03,00,06,00,00,11,80,00,00,FF,FF
1130 data 00,07,00,0F,FF,FF,00,01,00,05,00,14,00,00,00,30
1140 data 00,02,11,21,00,00,00,00,00,29,00,0C,00,02,FF,FF
1150 data FF,FF,00,16,00,00,00,20,00,00,00,80,00,02,00,01
1160 data 00,25,00,01,00,03,FF,FF,FF,FF,00,1D,00,08,00,00
1170 data 00,00,00,9C,00,0C,00,03,00,11,00,01,00,04,FF,FF
1180 data FF,FF,00,1D,00,08,00,00,00,00,00,B8,00,0D,00,05
1190 data 00,0E,00,01,00,05,FF,FF,FF,FF,00,1A,00,05,00,20
1200 data 00,00,00,75,00,02,00,09,00,11,00,01,00,00,FF,FF
1210 data FF,FF,00,1A,00,27,00,20,00,00,00,78,00,16,00,09
1220 data 00,11,00,01,00,00,00,D4,00,00,00,00,00,00,00,00
1230 data *
```

## Program Listing #4

ST Check BUG Data [for](#) Listing #3

```
100 data 469,544,391,421,536,623,487,693,656,884,5704
1050 data 644,872,720,503,703,556,707,710,686,685,6786
1150 data 679,860,716,758,875,677,722,530,196,6013
```

•

## CHAPTER 23 - Desk Accessories with Built-In Resource Trees

In chapter 22 we looked at a short utility that will set the ST's date and time. Although that program works fine when run from the Desktop, it would be more convenient to have it as a desk accessory, so that it would be available to us from within other GEM programs. Programming a desk accessory isn't much more complicated than programming any other GEM application, but, in the case of our Date/Time utility, there arises one complication.

How often have you seen a desk accessory that requires a .RSC file? Not too often. Oh, there's a couple of them floating around, but it is not a good programming practice to force a desk accessory to rely on a .RSC file -- and for a very good reason.

A desk accessory, once it has been loaded, is constantly active. Even if you haven't selected it from the Desk drop-down menu, it is still running, waiting for the cue that will set it in motion. In fact, the only way to terminate a desk accessory is to shut off your machine.

Now think about what we know about .RSC files. They have to be loaded into the ST's memory with a call to `rsrc_load()`, but more importantly, the memory the resource file takes up has to be returned to the system at the termination of a program by a call to `rsrc_free()`. Since a desk accessory is always "running," when do we return the memory used by our resource trees?

I know what you're thinking. "What difference does it make whether or not we ever call `rsrc_free()` when the only way to terminate a desk accessory is to turn the machine off? I mean, last time I heard, turning off the machine was a great way to release all the memory!" Right you are. But there is one situation where a desk accessory gets reinitialized: when you switch resolutions. If your desk accessory has to load a resource file, then each time you switch resolutions, more memory will be taken up, because the resource is being reloaded, even though the old one hasn't been released.

So before we can convert our Date/Time utility to a desk accessory, we have to build our resource tree right into the program. How complicated this process will be depends on what tools you have available. If you have a resource construction program that'll save your object trees out in source code form, then you're three-quarters of the way there. All you have to do is plug in a few addresses, and you're on your way. If you don't have access to an RCP that will do this for you, you'll have to write your resource by hand, a tedious project indeed.

In either case, though, we're going to have to have a clear understanding of resource trees in order to get our desk accessory's dialog box working. This means we'll be doing a review of some material and applying what we learn directly to our Date/Time utility.

### Our Resource Tree

Take a look at Listing 1. Near the top you'll see some data labeled "Resource tree." This is all the data for our dialog box as it was saved from Atari's RCS2 in source code form. (Actually, RCS2 isn't too bright and saves data for everything, even data structures not used in our tree. I've already deleted the unneeded data for the sake of clarity.)

At the top of the data, you'll see an array of pointers called `rs_strings[]`. (Even though the strings themselves are shown in this array, we still have an array of pointers here, each pointer holding the address of its associated string.) These pointers point to all the strings we need for our dialog box, with some of the objects being allotted three strings. (The first nine strings shown are actually three

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

groups of three.) Why three? Because editable text fields require not only a text string but a format template and validation string, as well. Remember?

The first string shown in `rs_strings[]` is our dialog box's title line. We don't want an editable text field here so the format and validation strings (the second and third strings shown in `rs_strings[]`) are empty. The next group of three strings is for the time field of our dialog box. First is the `te_ptext` string (the text that will be displayed when the dialog box is first drawn and the area where the text the user enters will be found after he exits the dialog box), followed by the `te_ptmplt` string (the uneditable text that is displayed in the text field), and the `te_pvalid` string (the string that determines what type of data is allowable in each position of the string). If all of this is confusing, please review Chapters 14 and 15 on dialog boxes.

The next three strings are the `te_ptext`, `te_ptmplt`, and `te_pvalid` strings for our dialog box's date field. And finally there is the text for our OK and CANCEL buttons.

Following `rs_strings[]` is another array called `rs_tedinfo[]`. As you can tell by the name of the array, the data here makes up the `tedinfo` structures for the editable text strings in our dialog box. If you think back, you'll remember that a `tedinfo` structure contains all the information GEM needs to draw and handle editable text fields. There are three `tedinfos` contained in `rs_tedinfo[]`, one each for our dialog's title, time, and date fields.

The first three long words of a `tedinfo` structure are pointers to the object's `te_ptext`, `te_ptmplt`, and `te_pvalid` strings, respectively. If you look at the first three long words in the first `tedinfo` structure shown in `rs_tedinfo[]`, you'll see that we've got the values 0L, 1L, and 2L. These don't look very much like pointers, do they? That's because they're not! They are actually offsets into the `rs_strings[]` array, telling us which pointers we need to place in the `tedinfo` structure.

This `tedinfo` is for our dialog's title field. We are told here that the `te_ptext` string for this field is pointed to by element 0 of the `rs_strings[]` array, and the `te_ptmplt` and `te_pvalid` strings for this field are pointed to by elements 1 and 2, respectively. When we initialize our desk accessory, it is up to us to see that the right pointers get placed into the `tedinfo`.

The next six members of the `tedinfo` structure contain (in order) the font size, a reserved word, the horizontal justification of the text, color information, another reserved word, and the you'll see the value 0x21121L. This value contains information on the box's color and border thickness. It doesn't hold a pointer to a `tedinfo` because a `G_BOX` contains no editable text.

But look at the second object in the tree. This is our dialog's title field, and if we look up the `G_BOXTEXT` object type in our reference materials, we'll find that the `ob_spec` field of this type of object does hold a pointer to a `tedinfo` structure. If you look at the `ob_spec` field for this object in our listing, you'll see the value 0x0L which is obviously not a pointer. Again, this is an offset, telling us that the `tedinfo` for our title field is element 0 of the `rs_tedinfo[]` array.

The next two objects in our tree, the time and date fields, also contain offsets in the `ob_spec` member, telling us that the `tedinfos` for these objects are element 2 and 3 of the `rs_tedinfo[]` array.

The last two objects in our tree are the OK and CANCEL buttons. They are objects of the type `G_BUTTON`, and if we look up these objects in our reference materials, we'll find that their `ob_spec` fields should contain not a pointer to a `tedinfo`, but instead a pointer to the string that will be displayed in the button. Looking at the `ob_spec` fields in the button objects, we see that we once

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

again have some offsets, 0x9L and 0xAL. These values tell us that the pointers to the button strings are found in element 9 and 10 of the `rs_strings[]` array.

The last four members of our object structures contain the X coordinate, Y coordinate, width, and height of the objects. It's important to note that these values are given as character coordinates rather than pixel coordinates. This is because the resource construction program has no idea what resolution we'll be running the program in. We must adjust these values so that the dialog box is drawn properly for whatever resolution in which we happen to be.

Our code-resident resource tree is concluded with the array `rs_trindex[]`. This array will contain the addresses of each of the separate trees that make up our resource tree. In our case, we have only one tree, a dialog box, so this array contains space for only one entry. Had we had several trees -- for instance, three dialog boxes -- this array would have had places for the addresses of each.

If you look at the value stored in the `rs_trindex` array, you will see that, once again, we are dealing with an offset, 0L. This tells us that the address of object 0 of `rs_object[]` should be placed in this element of the array.

### Writing a Desk Accessory

As I said before, writing a desk accessory is a fairly simple process. As we've done with our Time/Date utility, we can actually write the program as a normal GEM application, and then, once we've got it running, convert it to a desk accessory.

Listing 1 is just such a conversion. Most of the program is identical to Chapter 22's, and so, if you typed in Chapter 22's listing, you can use most of the code directly in this program. The following functions have not changed: `chk_date()`, `set_date()`, `get_time()`, `get_date()`, `get_tedinfo_string()`, and `open_vwork()`. When you compile the new version, remember to have the DATE.H file from Chapter 22 on the same disk with your source code.

Let's take a look at the function `do_acc()`, since it is here that we set up our desk accessory, as well as initialize our dialog box. The first thing we have to do to get our desk accessory up and running is to get its name placed on the menu bar so that the user can select it. This is done with the call

```
menu_id = menu_register(gl_apid, " Name ");
```

where the integer `menu_id` is the ID number of our desk accessory returned from the call to `menu_register()` and the integer `gl_apid` is the application ID assigned by `appl_init()`. We don't have to worry about retrieving the value of `gl_apid` ourselves; we just define it as an external variable, much like the GEM global arrays.

Once we've got our accessory registered on the menu bar, we must initialize the dialog box. This requires replacing the offsets in the various structures with the proper pointers and changing the object coordinates and sizes from character form to pixel form.

First, we store the address of our resource tree into the `rs_trindex[]` array, which is done like this:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
rs_trindex[0] = (long) rs_object;
```

Had we had two object trees in our resource, we would have had to fill in an address for `rs_trindex[1]`, too. Because we have six objects in our dialog box, the first object of a second tree (if it existed) would be the seventh object in the `rs_object[]` array, and we would have stored its address like this:

```
rs_trindex[1] = (long) &rs_object[6];
```

The addresses of additional trees are stored in the same way, each address being placed in the next element of `rs_trindex[]` and each address being derived from the element of `rs_objects[]` that contains the first object in the object tree.

Now we move down one step in the hierarchy from the tree to the objects in the tree. Five of our objects require that the `ob_spec` fields be filled in with pointers. The first three -- the title, time, and date fields -- require pointers to `tedinfos`. Using the offsets that the resource construction program left for us, we initialize these fields using this method:

```
rs_object[1].ob_spec = (char *) &rs_tedinfo[0];
```

In English the above code says, "The `ob_spec` field of the second object in the array `rs_object[]` gets the value of a pointer to character, and that pointer points to the first `tedinfo` structure in the array `rs_tedinfo[]`." Yikes! I think I liked the C version better! Anyway, we initialize the other two `tedinfo` pointers the same way, as you can see in Listing 1.

Next we have two objects -- the OK and CANCEL buttons -- that must have pointers to strings placed in their `ob_spec` fields. The code that accomplishes that job (for one of the buttons) looks like this:

```
rs_object[4].ob_spec = rs_strings[9];
```

Once again in English, the above says, "The `ob_spec` field of the fifth object in the array `rs_object[]` gets the string pointer stored in element 9 of the array `rs_strings[]`."

Now we have to take another step down in our hierarchy and initialize our `tedinfo` structures. We have three of them that must have pointers to their `te_ptext`, `te_ptmplt`, and `te_pvalid` strings filled in. We do that with code similar to what we used for the `ob_spec` pointers above, like this:

```
rs_tedinfo[0].te_ptext = rs_strings[0];  
rs_tedinfo[0].te_ptmplt = rs_strings[1];  
rs_tedinfo[0].te_pvalid = rs_strings[2];
```

The other `tedinfos` are handled the same way.

Now all we have left to do is convert the dialog's character coordinates to pixel coordinates. Luckily, there's a function that does that dirty work for us. The call



## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
rsrc_obfix( tree_addr, object );
```

where `tree_addr` is the address of the object tree and the integer `object` is the index of the object within the tree, does all the conversions for us. In Listing 1, we've used a for loop to adjust all six objects with a single statement.

### Waiting Forever

As I said before, a desk accessory, once installed on the menu bar, waits to be called by the user. When the user clicks on the desk accessory's entry on the menu bar, the desk accessory is notified by a message from GEM. What we need to do is construct a while loop that'll loop "forever." In that loop we'll check to see if we've received a GEM message.

Once we get a message (in this case, from a call to `evnt_mesag()`), we'll check to see if the message type (stored in `msg_buf[0]`) is an `AC_OPEN` message. An `AC_OPEN` message is sent by GEM whenever a desk accessory menu entry is clicked on by the user. When we receive this message, we then have to compare the menu ID sent to us in `msg_buf[4]` with the menu ID we obtained with our call to `menu_register()`. If they match, we go ahead and bring up our dialog box so that the user can edit the date and time.

And that's about all there is to it. I should mention that there is also a `AC_CLOSE` message that is sent to your desk accessory when the desk accessory is closed. This message is important if you're using windows in the desk accessory because GEM automatically closes these windows when it returns to the desktop. Without the `AC_CLOSE` message, you'd have no way of knowing when to reset any window flags or other related data items you may need to update. Both the `AC_OPEN` and `AC_CLOSE` messages are defined in the `GEMDEFS.H` file.

### The Desk Accessory Link

One last note: Whenever you link a program with Megamax C, some start-up code (contained in the `INIT.O` file) is automatically linked to your object file. Desk accessories, however, have to be initialized differently when they are run, so require different start-up code. The desk accessory start-up code is contained in the `ACC.L` file supplied with the Megamax compiler. (Other compilers will have their own versions of the start-up module.) The source code for your desk accessory is compiled in the same manner as any other program, but when you get to the link step, you must be sure to select `ACC.L` as the first file in your link list. If you don't, your desk accessory will not run.

NOTE FOR LASER C USERS: The new version of Megamax C, Laser C, no longer needs the separate `ACC.L` file. To produce an accessory with Laser C, you must compile and link your source code separately. After your source code has compiled, go to the LINK dialog box and change the target name, below the left file selector, to `progrname.ACC`. The new Laser linker will automatically add the special code necessary to produce a working desk accessory.

# C-MANSHIP COMPLETE - by CLAYTON WALNUT

## Program Listing #1

```
/******  
/*C-manship, Listing 1*/  
/* CHAPTER 23 */  
/*Developed with Megamax C*/  
/******  
#include <osbind.h>  
#include <gemdefs.h>  
#include <obdefs.h>  
#include "date.h"  
  
#define TRUE 1  
#define FALSE 0  
#define MATCH 0  
  
/* GEM arrays */  
int work_in[11],  
    work_out[57],  
    contrl[12],  
    intin[128],  
    ptsin[128],  
    intout[128],  
    ptsout[128];  
  
extern int gl_apid; /* Global application ID. */  
  
int handle, /* Application handle. */  
    dum, /* Dummy storage.*/  
    menu_id; /* Our accessory's ID. */  
char *get_tedinfo_str ();  
  
int msg_buf[8]; /* Message buffer. */  
  
OBJECT *datedial_addr; /* Pointer to dialog box. */  
  
/******  
* Resource tree  
*****/  
  
char *rs_strings[] = {  
    "DATE AND TIME",  
    "",  
    "",  
    "000000AM",  
    "Time: __:__:__ __",  
    "999999AA",  
    "000000",  
    "Date: __/__/__",  
    "999999",  
    "OK",  
    "CANCEL"};  
  
TEDINFO rs_tedinfo[] = {  
    0L, 1L, 2L, 3, 6, 2, 0x1180, 0x0, -1, 14,1,  
    3L, 4L, 5L, 3, 6, 0, 0x1180, 0x0, -1, 9,18,  
    6L, 7L, 8L, 3, 6, 0, 0x1180, 0x0, -1, 7,15};
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
OBJECT rs_object[] = {
    -1, 1, 5, G_BOX, NONE, 0x30, 0x21121L, 0,0, 41,12,
    2, -1, -1, G_BOXTEXT, NONE, SHADOWED, 0x0L, 2,1, 37,1,
    3, -1, -1, G_FTEXT, EDITABLE, NORMAL, 0x1L, 12,3, 17,1,
    4, -1, -1, G_FTEXT, EDITABLE, NORMAL, 0x2L, 13,5, 14,1,
    5, -1, -1, G_BUTTON, 0x5, SHADOWED, 0x9L, 2,9, 17,1,
    0, -1, -1, G_BUTTON, 0x27, SHADOWED, 0xAL, 22,9, 17,1};

long rs_trindex[] = {
    0L};

#define NUM_OBS 6

/*****
 * Main program.
 *****/
main ()
{
    appl_init (); /* Init application. */
    open_vwork (); /* Open virtual workstation. */
    do_acc (); /* Go do our thing.*/
}

/*****
 * do_acc ()
 * Initialize and handle desk accessory.
 *****/
do_acc ()
{
    int x; /* Loop variable. */

    /* Place our accessory on the menu bar. */
    menu_id = menu_register ( gl_apid, "Date/Time " );

    /* Initialize resource tree. */
    rs_trindex[0] = (long) rs_object;
    datedial_addr = (OBJECT *) rs_trindex[0];
    rs_object[1].ob_spec = (char *) &rs_tedinfo[0];
    rs_object[2].ob_spec = (char *) &rs_tedinfo[1];
    rs_object[3].ob_spec = (char *) &rs_tedinfo[2];
    rs_tedinfo[0].te_ptext = rs_strings[0];
    rs_tedinfo[0].te_ptmplt = rs_strings[1];
    rs_tedinfo[0].te_pvalid = rs_strings[2];
    rs_tedinfo[1].te_ptext = rs_strings[3];
    rs_tedinfo[1].te_ptmplt = rs_strings[4];
    rs_tedinfo[1].te_pvalid = rs_strings[5];
    rs_tedinfo[2].te_ptext = rs_strings[6];
    rs_tedinfo[2].te_ptmplt = rs_strings[7];
    rs_tedinfo[2].te_pvalid = rs_strings[8];
    rs_object[4].ob_spec = rs_strings[9];
    rs_object[5].ob_spec = rs_strings[10];

    /* Set all the objects' coordinates. */
    for ( x=0; x<NUM_OBS; ++x )
        rsrc_obfix ( datedial_addr, x );

    /* Wait forever for messages. */
    while ( 1 ) {
        evnt_mesag ( msg_buf );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
switch ( msg_buf[0] ) { /* msg_buf[0] is message type. */
    /* Open our accessory. */
case AC_OPEN:
    if ( msg_buf[4] == menu_id )
        do_date ();
}
}

/*****
* do_date ()
* Loads the resource file and handles the dialog box.
*****/
do_date ()
{
    int dial_x, /* Dialog's X coord.*/
        dial_y, /* Dialog's Y coord.*/
        dial_w, /* Dialog's width.*/
        dial_h, /* Dialog's height. */
        choice, /* Exit button clicked from dialog. */
        okay; /* Flag indicating if entered date valid. */

    char date_str[8], /* String to hold date. */
          time_str[10]; /* String to hold time. */

    char *string; /* Temp string pointer. */

    graf_mouse ( ARROW, 0L );

    get_time ( time_str );
    get_date ( date_str );

    /* Copy system time and date into dialog box, */
    string = get_tedinfo_str ( datedial_addr, TIMEFLD );
    strcpy ( string, time_str );
    string = get_tedinfo_str ( datedial_addr, DATEFLD );
    strcpy ( string, date_str );

    /* Prepare dialog box for drawing, and init flag. */
    form_center(datedial_addr,&dial_x,&dial_y,&dial_w,&dial_h);
    form_dial(FMD_START,0,0,10,10,dial_x,dial_y,dial_w,dial_h);
    okay = TRUE;

    /* This loop repeats until the user clicks the CANCEL */
    /* or until the user enters a valid date and clicks OK. */
    do {
        /* Draw dialog and allow user to manipulate it. */
        objc_draw(datedial_addr,0,8,dial_x,dial_y,dial_w,dial_h);
        choice = form_do ( datedial_addr, TIMEFLD );

        /* Reset the state of the chosen button. */
        datedial_addr[choice].ob_state = SHADOWED;

        /* If OK was clicked, check entered date and set system */
        /* date if date entered is valid, */
        if ( choice == OKBTN ) {
            okay = chk_date ( datedial_addr );
        }
    } while ( !okay );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        if ( okay )
            set_date ( datedial_addr );
    }
    while ( okay == FALSE && choice == OKBTN );

    /* Get rid of the dialog box. */
    form_dial(FMD_FINISH,0,0,10,10,dial_x,dial_y,dial_w,dial_h);
}

/*****
* chk_date ()
* Examines the strings in dialog for a valid date
* and valid time.
*****/
chk_date ( dial_addr )
OBJECT *dial_addr; /* Address of dialog box. */
{
    int mnth, day, year, /* Date and time broken into integers.*/
        hour, min, sec,
        space, /* Flag for bad chars in time string. */
        okay, /* Flag indicating valid time & date. */
        x; /* Loop variable. */

    char m[3], d[3], y[3], /* Date & time as character arrays. */
        h[3], mn[3], s[3],
        ap[3]; /* "AM" or "PM" */

    char *date_str, /* Pointer to string containing date. */
        *time_str; /* Pointer to string containing time. */

    /* Init date and time integers to error condition. */
    mnth = day = year = hour = min = sec = -1;

    /* Get address of string containing date. */
    date_str = get_tedinfo_str ( dial_addr, DATEFLD );

    /* Convert date string to integer format. */
    if ( strlen ( date_str ) == 6 ) {
        strncpy ( m, date_str, 2 );
        m[2] = 0;
        strncpy ( d, &date_str[2], 2 );
        d[2] = 0;
        strncpy ( y, &date_str[4], 2 );
        y[2] = 0;
        mnth = atoi ( m );
        day = atoi ( d );
        year = atoi ( y );
    }

    /* Get address of string containing time. */
    time_str = get_tedinfo_str ( dial_addr, TIMEFLD );

    /* Check for spaces in time string. */
    space = FALSE;
    for ( x=0; x<6; ++x )
        if ( time_str[x] == ' ' )
            space = TRUE;

    /* Convert time string to integer format. */
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
if ( (strlen ( time_str ) == 8) && !space ) {
    strncpy ( h, time_str, 2 );
    h[2] = 0;
    strncpy ( mn, &time_str[2], 2 );
    mn[2] = 0;
    strncpy ( s, &time_str[4], 2 );
    s[2] = 0;
    hour = atoi ( h );
    min = atoi ( mn );
    sec = atoi ( s );
    strcpy ( ap, &time_str[6] );
}

/* Examine time and date for validity. */
if ( mnth < 1 | mnth >12 | day < 1 | day > 31
    | year < 0 | year > 99 | hour < 0 | hour > 23 | min < 0
    | min > 59 | sec < 0 | sec > 59 |
    ((strcmp (ap,"AM")!=MATCH) && (strcmp (ap,"PM")!=MATCH))) {
    okay = FALSE;
    form_alert( 1, "[1][Date or time not valid!][CONTINUE]"
);
}
else
    okay = TRUE;

return ( okay );
}

/*****
* set_date ()
* Sets the system time and date to the values
* entered into the dialog box.
*****/
set_date ( dial_addr )
OBJECT *dial_addr; /* Address of dialog box. */
{
    char *string; /* Temporary string pointer. */
    char s[3]; /* Temporary string storage. */
    int h, /* Work variable.*/
        time, /* Time in system format.*/
        date; /* Date in system format.*/

    /* Get address of string containing time. */
    string = get_tedinfo_str ( dial_addr, TIMEFLD );

    /* Extract "hours" portion and convert to integer. */
    strncpy ( s, string, 2 );
    h = atoi ( s );

    /* Adjust hour to the 24-hour clock format. */
    if ( (strcmp ( &string[6], "PM" ) == MATCH) && (h != 12) )
        h += 12;
    if ( (strcmp ( &string[6], "AM" ) == MATCH) && (h == 12) )
        h = 0;

    /* Shift bits into the proper position and place them */
    /* into the time integer. */
    h = h << 11;
    time = h;
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Get the "minutes" portion, convert to integer, */
/* shift bits and place them into the time integer. */
strncpy ( s, &string[2], 2 );
h = atoi ( s );
h = h << 5;
time = time | h;

/* Process the "seconds" portion of the time. */
strncpy ( s, &string[4], 2 );
h = atoi ( s ) / 2;
time = time | h;

/* Set the system clock to the new time. */
Tsettime ( time );

/* Get the address of the string containing the date. */
string = get_tedinfo_str ( dial_addr, DATEFLD );

/* Process the "month" portion. */
strncpy ( s, string, 2 );
h = atoi ( s );
h = h << 5;
date = h;

/* Process the "day" portion. */
strncpy ( s, &string[2], 2 );
h = atoi ( s );
date = date | h;

/* Process the "year" portion. */
strncpy ( s, &string[4] );
h = atoi ( s ) - 80;
h = h << 9;
date = date | h;

/* Set the system to clock to the new date. */
Tsetdate ( date );
}

/*****
* get_time ()
* Gets system time and converts it to string format.
*****/
get_time ( string )
char *string; /* Pointer to string in which to store time. */
{
    int time, /* Time in system format. */
        hour, min, sec; /* Time broken down into separate ints. */

    char s[3]; /* "AM" or "PM" */

    /* Get system time & break down into individual components. */
    time = Tgettime ();
    sec = ( time & 0x001f ) * 2;
    min = ( time >> 5 ) & 0x003f;
    hour = ( time >> 11 ) & 0x001f;

    /* Convert system 24-hour format to regular 12-hour format. */
    if ( hour > 11 ) {
        strcpy ( s, "PM" );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        if ( hour > 12 )
            hour -= 12;
    }
    else {
        strcpy ( s, "AM" );
        if ( hour == 0 )
            hour = 12;
    }

    /* Convert and add hours to time string. */
    if ( hour < 10 ) {
        string[0] = '0';
        sprintf ( &string[1], "%d", hour );
    }
    else
        sprintf ( string, "%d", hour );

    /* Convert and add minutes to time string. */
    if ( min < 10 ) {
        string[2] = '0';
        sprintf ( &string[3], "%d", min );
    }
    else
        sprintf ( &string[2], "%d", min );

    /* Convert and add seconds to time string. */
    if ( sec < 10 ) {
        string[4] = '0';
        sprintf ( &string[5], "%d", sec );
    }
    else
        sprintf ( &string[4], "%d", sec );

    /* Add "AM" or "PM" to time string. */
    strcpy ( &string[6], s );
}

/*****
* get_date ()
* Gets system date and converts it to string format.
*****/
get_date ( string )
char *string; /* Pointer to string that will contain the date. */
{
    int date, /* Date in system format. */
        day, mnth, year; /* Date broken into components. */
    /* Get system date and convert to individual components. */
    date = Tgetdate ();
    day = date & 0x001f;
    mnth = (date >> 5) & 0x000f;
    year = ((date >> 9) & 0x007f) + 80;
    year = year % 100;

    /* Convert and add "months" portion to date string. */
    if ( mnth < 10 ) {
        string[0] = '0';
        sprintf ( &string[1], "%d", mnth );
    }
    else
        sprintf ( string, "%d", mnth );
}
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* convert and add "days" portion to date string. */
if ( day < 10 ) {
    string[2] = '0';
    sprintf ( &string[3], "%d", day );
}
else
    sprintf ( &string[2], "%d", day );

/* Convert and add "year" portion to date string. */
sprintf ( &string[4], "%d", year );
}

/*****
* get_tedinfo_str ()
* Returns a pointer to an editable string in a
* dialog box.
*****/
char *get_tedinfo_str ( tree, object )
OBJECT *tree; /* Address of dialog box. */
int object; /* Object that contains the string. */
{
    TEDINFO *ob_tedinfo; /* Pointer to a tedinfo structure. */

    ob_tedinfo = (TEDINFO *) tree[object].ob_spec;
    return ( ob_tedinfo->te_ptext );
}

/*****
* open_vwork ()
* Opens a virtual workstation.
*****/
open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open*/
    /* a virtual workstation. */

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}
```

### Program Listing #2

```
#define DATEDIAL 0/* TREE */
#define TIMEFLD 2/* OBJECT in TREE #0 */
#define DATEFLD 3/* OBJECT in TREE #0 */
#define OKBUTN 4/* OBJECT in TREE #0 */
#define CANBUTN 5/* OBJECT in TREE #0 */
.
```

## CHAPTER 24 - THE GRAPHICS MANAGER LIBRARY

Certainly all of us have used GEM's various constructions such as windows and dialog boxes. When we use these things we don't pay much attention to what is going on. We just take it for granted that when we drag the lower-right corner of a window with the mouse pointer the window will get larger, or that when we grab its title bar we'll be able to move it about the screen.

What you may not realize is that these routines that control windows and dialog boxes and the other GEM constructions are actually "high-level" routines that call various other "low-level" routines on the AES and VDI. We've touched on this subject before when we talked about the VDI and why those graphics routines were called "primitives" and when we learned that it's not a good idea to call VDI mouse functions if you're using AES functions that may also call the VDI mouse routines.

Specifically, what we'll be talking about are a few of the AES functions used by windows and dialog boxes. We're going to be dropping down a level, as it were, from the sophisticated window and dialog routines to some of the functions these routines depend upon in order to do their tricks.

### The Sample Program

When you run this chapter's program, a large rectangle will be drawn on the screen. The program will then wait for input via the mouse. The large rectangle is a border within which all the program's activities will be contained.

Hold down the left mouse button and drag the mouse pointer down to the right. A box that expands and contracts with the movement of the mouse pointer will be drawn on the screen. When you release the mouse button, the final box will be filled in. In the lower right corner of the box is a small button. By placing the mouse pointer over this button and holding down the left button, you'll be able to change the size of the box by dragging on its corner. Note that, if you try to resize the box beyond the boundary we've set up, the program will ignore your request.

If you place the mouse pointer inside the box and hold down the left button, the mouse pointer will change into a hand that you can use to reposition the box anywhere on the screen, as long as the box stays within the border we've drawn. (As a matter of fact, the program won't allow you to drag the box outside of the border.)

To get out of the sample program and back to the Desktop, hold down the right mouse button.

### Déjà Vu

I'm sure you recognize all of the effects we're using in our sample program. We've spent much time on our STs moving and sizing boxes. These routines are the foundations upon which the more advanced abilities of GEM are built. Fortunately, these routines are available to us as GEM programmers, so that we can construct our own specialized, GEM-like routines.

All of the functions we're using to simulate GEM in this program are part of the AES Graphics Manager library. We've used a couple of these functions, like `graf_handle()` and `graf_mouse()`, in the past. We've also used most of the Graphics Manager's functions indirectly. For instance, when we studied dialog boxes, we talked about a function called `form_dial()` that, among other things, allowed us to have an expanding or shrinking box displayed on the screen. The `form_dial()` function performed some of its tricks by calling the `graf_growbox()` and `graf_shrinkbox()` functions found in the AES

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

Graphics Manager. (Note that all the Graphics Manager functions start with the prefix graf.) If we wish to draw an animated expanding box of our own, we use the call

```
graf_growbox( x1, y1, w1, h1, x2, y2, w2, h2 );
```

where the first four arguments are the X coordinate, Y coordinate, width, and height of the starting rectangle; and the second four arguments are the X coordinate, Y coordinate, width, and height of the final rectangle. This function will return a zero upon failure or a positive integer if successful. Its complementary function, the one that draws an animated shrinking box, is called in the same way:

```
graf_shrinkbox( x1, y1, w1, h1, x2, y2, w2, h2 );
```

### Our Program

In our sample program, we haven't used `graf_growbox()` or `graf_shrinkbox()`, but we have used many of the other functions available in the Graphics Manager library.

Look at the function `do_box()`. Near the top you'll see a nested while loop. The first while is set up to repeat until we receive acceptable coordinates for the box the user wants to draw. The inner while loop forces the program to wait for a press of the left mouse button. We continually poll the mouse's state until we detect that the button has been pressed. The function call we use to check the mouse's state is:

```
graf_mkstate(&mouse_x, &mouse_y, &mouse_but, &key_state );
```

where `&mouse_x`, `&mouse_y`, `&mouse_but`, and `&key_state` are the addresses of the integers that will receive the mouse's current X coordinate, the mouse's current Y coordinate, the mouse's current button state, and the state of the keyboard's Shift, Control and Alternate keys. The integer `mouse_but` will contain a 1 if the left button was pressed, a 2 if the right button was pressed and a 3 if both were pressed. The integer `key_state` will contain a 1 if the right Shift was pressed, a 2 if the left Shift was pressed, a 4 if the Control key was pressed, and an 8 if the Alternate key was pressed. For multiple key presses, we would just add the appropriate values together. (You remember how to handle bit settings, right?)

When we detect that the left mouse button has been pressed, we get the mouse's coordinates and use them as the coordinates for the upper-left corner of the box. The next step, then, is to allow the user to drag the mouse to outline the box that he wants. We do this with the call

```
graf_rubberbox(box_x, box_y, min_w, min_h, &box_w, &box_h);
```

where `box_x` and `box_y` are the coordinates of the box's upper-left corner, `min_w` and `min_h` are the box's minimum allowable size in pixels, and `&box_w` and `&box_h` are the addresses of integers where the final selected width and height of the box will be stored.

After our call to `graf_rubberbox()`, we'll have all the information we need to draw the box the user selected. But before we actually draw the box, we must check to make sure that the box will fit inside our boundary. In the if statement that checks this condition, you'll notice that we're using the values

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

stored in `work_out[0]` and `work_out[1]`. The `work_out[]` array is one of our GEM global arrays, and its elements were filled in when we initialized the application program. Element 0 of this array contains the width of our device (in this case, the device is the screen, so it is measured in pixels), and element 1 contains the height of our device.

Once we've drawn the box, we enter another nested while loop. The outer loop checks for a program-exit condition, and the inner loop again polls the mouse. If the right mouse button is pressed, we exit the program. If the left mouse button is pressed, we have to check the location of the mouse, so that we know whether the user wants to move the box or size it.

If the user is holding down the mouse button while on the box's sizing button, we need to call `graf_rubberbox()` again to let the user choose the new size of the box. There are two complications. The first is something we've already handled before, and that is that we can't allow the size of the new box to exceed the boundary that we've set up.

The second problem arises when the user chooses to make the box smaller. In this case it isn't good enough to just redraw the box at its new size, because we'll be leaving on the screen parts of the old box. To erase the leftovers, we first have to calculate their size and then redraw them in the background color. Of course, we could simplify matters by erasing the entire old box before we draw the new one, but that's an inelegant solution and one that the user would catch us using, since he'd be able to see us erasing the old box. We want our programs to operate so smoothly that they seem magical. The entire resizing process is handled in our function `size_box()`.

If the user has pressed the left mouse button while within the box (but not on the box's sizing button), we have to allow the user to move the box to a new location. Our function `move_box()` illustrates how to do this.

We can get the new location of a box with the call

```
graf_dragbox(box_w, box_h, box_x, box_y, bound_x, bound_y, bound_w,
             bound_h, &end_x, &end_y ;
```

where the integers `box_w` and `box_h` are the box's width and height; `box_x` and `box_y` are the addresses of integers where the box's new X and Y coordinates will be stored; the integers `bound_x`, `bound_y`, `bound_w`, and `bound_h` are the coordinates, width, and height of the boundary within which the box must remain; and `&end_x` and `&end_y` are the addresses of integers where the box's new X and Y coordinates will be stored.

Once we get the coordinates of the new box, all we must do is erase the old box and draw the new one.

### Some Leftovers

There are a couple of functions in the Graphics Manager library that we haven't looked at yet. One of them is a function that allows you to draw an animated box moving from one location to another. The call looks like this:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
graf_movebox(box_w, box_h, old_x, old_y, new_x, new_y );
```

Here, the integers `box_w` and `box_h` are the width and height of the box, the integers `old_x` and `old_y` are the current X and Y coordinates of the box, and the integers `new_x` and `new_y` are the final coordinates of the box. Note the spelling of the function call; it's different than the spelling in the Megamax manual. If you try to call the function using the manual's spelling, your program will not link properly because the linker won't be able to find the label. Anyway, this function is really of limited use; I can't think of any place where I've seen it in action.

Finally, the two remaining functions, `graf_slidebox()` and `graf_watchbox()` are used with object trees. The first is the function that allows sliders, such as those found in windows, to work; and the second allows us to track the mouse in and out of a particular rectangle while the mouse button is held down. These functions would be useful should we ever want to write our own custom dialog box (`form_do()`) routines.

### Put on the Coffee

This completes our tour of the AES Graphics Manager library. You should now have an even clearer idea of how some of GEM's routines work, and you should be able to construct many handy GEM-like routines with the information you've learned.

## Program Listing #1

```
/******  
/*C-manship, Listing 1*/  
/* CHAPTER 24 */  
/*Developed with Megamax C*/  
/******  
#include <obdefs.h>  
#include <gemdefs.h>  
  
#define TRUE1  
#define FALSE 0  
#define SOLID 1  
#define PATTERN 2  
#define LEFT1  
#define RIGHT 2  
  
/* GEM arrays */  
int work_in[11], work_out[57], contrl[12], intin[128],  
    ptsin[128], intout[128], ptsout[128];  
  
int handle, /* Application handle. */  
    dum; /* Dummy storage.*/  
  
/******  
* Main program.  
*****/  
main ()  
{  
    appl_init (); /* Init application. */  
    open_vwork (); /* Open virtual workstation. */  
    do_box (); /* Go do our thing.*/  
    v_clsvwk ( handle ); /* Close virtual workstation.*/  
    appl_exit (); /* Back to the desktop.*/  
}  
  
/******  
* do_box ()  
*  
* Calls screen setup function and handles the mouse,  
* calling the appropriate box functions based on the  
* mouse's coordinates and button state.  
*****/  
do_box ()  
{  
    int mouse_x, /* Mouse X coordinate. */  
        mouse_y, /* Mouse Y coordinate. */  
        mouse_but, /* Mouse button state. */  
        box_x, /* Selected box X coord. */  
        box_y, /* Selected box Y coord. */  
        box_width, /* Selected width of box.*/  
        box_height, /* Selected height of box. */  
        exit, /* Program exit flag.*/  
        coords_ok; /* Proper coordinates flag.*/  
  
    setup_scrn ();  
    coords_ok = FALSE;  
    graf_mouse ( ARROW, 0L );
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
/* Wait for box coordinates within the boundary. */
while ( !coords_ok ) {

    /* Poll for left button press. */
    mouse_but = 0;
    while ( mouse_but != LEFT )
        graf_mkstate ( &mouse_x, &mouse_y, &mouse_but, &dum );

    /* Get coordinates for the box. */
    box_x = mouse_x;
    box_y = mouse_y;
    graf_rubberbox(box_x,box_y,20,20,&box_width,&box_height);

    /* Allow only a box whose size fits within the */
    /* boundary to be drawn. */
    if ( box_x > 20 && box_x + box_width < work_out[0]-21 &&
        box_y > 20 && box_y + box_height < work_out[1]-21 ) {
        draw_box ( box_x, box_y, box_width, box_height );
        coords_ok = TRUE;
    }
}
exit = FALSE;
while ( !exit ) {
    mouse_but = 0;

    /* Wait for press of left or right mouse button. */
    while ( mouse_but != LEFT && mouse_but != RIGHT )
        graf_mkstate ( &mouse_x, &mouse_y, &mouse_but, &dum );

    if ( mouse_but == LEFT )

        /* If the mouse was on the sizing button, */
        /* allow the user to resize the box.*/
        if ( chose_size ( mouse_x, mouse_y, box_x, box_y,
            box_width, box_height ) )
            size_box ( box_x, box_y, &box_width, &box_height );

    /* If the mouse was anywhere else in the */
    /* box, allow the user to move the box.*/
    else if ( chose_move ( mouse_x, mouse_y, box_x, box_y,
        box_width, box_height ) )
        move_box ( &box_x, &box_y, box_width, box_height );

    if ( mouse_but == RIGHT )
        exit = TRUE;
}

}

/*****
* draw_box ()
*
* Draws a shaded box with a button in the lower right
* corner.The input is the X and Y coordinates of
* the box's upper left corner and its width and height.
*****/
draw_box ( x, y, w, h )
int x, y, w, h;
{
    int pxy[4];
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
graf_mouse ( M_OFF, 0L );

/* Draw the main body of the box. */
vsf_interior ( handle, PATTERN );
vsf_style ( handle, 5 );
vsf_color ( handle, BLACK );
pxy[0] = x;
pxy[1] = y;
pxy[2] = x + w - 1;
pxy[3] = y + h - 1;
v_bar ( handle, pxy );

/* Draw the box's sizing button. */
vsf_interior ( handle, SOLID );
pxy[0] = x + w - 10;
pxy[1] = y + h - 10;
v_bar ( handle, pxy );

graf_mouse ( M_ON, 0L );
}

/*****
* size_box ()
*
* Resizes a box. The input is the X and Y coordinates
* of the box and pointers to its width and height. The
* function returns the new width and height by way
* of the pointers, thus replacing the old values of
* the width and height.
*****/
size_box ( x, y, w, h )
int x, y, *w, *h;
{
    int old_w, old_h;
    int pxy[4];

    old_w = *w;
    old_h = *h;

    /* Get the new box size. */
    graf_rubberbox ( x, y, 20, 20, w, h );

    /* Don't allow the new box to exceed the boundary. */
    if ( x + *w > work_out[0]-20 | y + *h > work_out[1]-20 ) {
        *w = old_w;
        *h = old_h;
    }

    /* If the size is okay, draw the box. */
    else {
        draw_box ( x, y, *w, *h );

        /* Erase the leftover portions (if */
        /* any) of the old box. */
        graf_mouse ( M_OFF, 0L );
        vsf_interior ( handle, SOLID );
        vsf_color ( handle, WHITE );
        if ( *w < old_w ) {
            pxy[0] = x + *w;
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
        pxy[1] = y;
        pxy[2] = x + old_w - 1;
        pxy[3] = y + old_h - 1;
        v_bar ( handle, pxy );
    }
    if ( *h < old_h ) {
        pxy[0] = x;
        pxy[1] = y + *h;
        pxy[2] = x + old_w - 1;
        pxy[3] = y + old_h - 1;
        v_bar ( handle, pxy );
    }
    graf_mouse ( M_ON, 0L );
}

/*****
* move_box ()
*
* Repositions a box. The input is a pointer to the
* box's X coord., a pointer to the box's Y coord., and
* the box's width and height. The new X and Y
* coordinates are returned from the function by way of
* the pointers, thus replacing the old X and Y values.
*****/
move_box ( x, y, w, h )
int *x, *y, w, h;
{
    int old_x, old_y;
    int pxy[4];

    old_x = *x;
    old_y = *y;
    graf_mouse ( FLAT_HAND, 0L );

    /* Get new location for the box. */
    graf_dragbox ( w, h, *x, *y, 21, 21,
        work_out[0]-41, work_out[1]-41, x, y );

    /* Erase the old box. */
    graf_mouse ( M_OFF, 0L );
    vsf_color ( handle, WHITE );
    vsf_interior ( handle, SOLID );
    pxy[0] = old_x;
    pxy[1] = old_y;
    pxy[2] = old_x + w - 1;
    pxy[3] = old_y + h - 1;
    v_bar ( handle, pxy );

    /* Draw the new box. */
    draw_box ( *x, *y, w, h );

    graf_mouse ( M_ON, 0L );
    graf_mouse ( ARROW, 0L );
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* chose_size ()
*
* Returns a boolean value based on whether the mouse
* button was pressed while over the box's sizing
* button. The input is the X and Y coordinates of
* the mouse, the X and Y coordinates of the box, and
* the width and height of the box.
*****/
chose_size ( mx, my, bx, by, bw, bh )
int mx, my, bx, by, bw, bh;
{
    if ( mx>bx+bw-10 && mx<bx+bw && my>by+bh-10 && my<by+bh )
        return ( TRUE );
    else
        return ( FALSE );
}

/*****
* chose_move ()
*
* Returns a boolean value based on whether the mouse
* button was pressed while over an area of the box
* other than the sizing button. The input is the X
* and Y coordinates of the mouse, the X and Y
* coordinates of the box, and the width and height of
* the box.
*****/
chose_move ( mx, my, bx, by, bw, bh )
int mx, my, bx, by, bw, bh;
{
    if ( mx>bx && mx<bx+bw && my>by && my<by+bh )
        return ( TRUE );
    else
        return ( FALSE );
}

/*****
* setup_scrn ()
*
* Prepares the screen by clearing the workstation and
* drawing a border.
*****/
setup_scrn ()
{
    int pxy[10];

    graf_mouse ( M_OFF, 0L );

    /* Erase the screen. */
    v_clrwk ( handle );

    /* Draw the border. */
    pxy[0] = 20;
    pxy[1] = 20;
    pxy[2] = work_out[0] - 20;
    pxy[3] = 20;
    pxy[4] = work_out[0] - 20;
    pxy[5] = work_out[1] - 20;
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
pxy[6] = 20;
pxy[7] = work_out[1] - 20;
pxy[8] = 20;
pxy[9] = 20;
v_pline ( handle, 5, pxy );

graf_mouse ( M_ON, 0L );
}

/*****
* open_vwork ()
* Opens a virtual workstation.
*****/
open_vwork ()
{
    int i;

    /* Get graphics handle, initialize the GEM arrays and open*/
    /* a virtual workstation. */

    handle = graf_handle ( &dum, &dum, &dum, &dum);
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}
```

## CHAPTER 25 - THE MYSTERY OF COMPILE AND LINK

A while ago, someone came up to me at a users' group meeting and told me that he was confused about what actually goes on during a compilation and link, as well as about the different types of files we must manipulate when programming in C -- specifically, .O and .H files. Luckily for you (he said, tongue in cheek), I couldn't resist turning that question into a chapter for C-manship. So we're going to take a short break from programming and dig into the innards of these wonderful programs we call compilers.

### Stating the Obvious

There is one thing that we all have to know before we can go any further with this topic. To some what I'm about to say may be an obvious fact, to others it may come as a revelation. But whatever group you may fall into, this fact is essential in understanding how your C compiler actually works.

Fact: Every computer understands only one language, machine language, and every program, no matter what language it's written in, must sooner or later be reduced to machine language. Of course, to completely understand the above fact, we must know exactly what machine language is. If you were to get a listing of a machine-language program, what you would have would be a long list of numbers. There would be no variable names, no labels of any kind, no strings of characters. Nothing but numbers. Those numbers represent the instructions the machine understands and the data it needs to perform those instructions. And if we wanted to get very literal about all this, the numbers in our list would all be binary numbers -- that is, consisting of nothing but zeros and ones. Usually, to make things easier for the programmer, "memory dumps" produce listings in hexadecimal format.

How a program is converted to machine code varies with the language we're using. For example, when we run an uncompiled BASIC program, each statement in the program is converted into machine language as it's encountered, rather than the whole program being converted at once. This on-the-fly conversion is why BASIC programs are so slow. BASIC is an example of an "interpreted" language.

Assembly-language programs are as close to machine language as you can get. Each assembly-language statement represents a single machine-language instruction. For this reason, many people confuse the terms "assembly language" and "machine language," but they are really not the same. Assembly language uses "mnemonics" (easy-to-remember names) for each of the machine-language instructions to make it easier for programmers to remember them. An assembly-language program is not interpreted; it is "assembled." During the assembly process, each of the mnemonics is converted to its machine-language equivalent.

Finally, we get to "compiled" languages, of which C is one. When a program is compiled, all the instructions in the source code are converted into machine language, so that we end up with a runnable program, one that doesn't need to be interpreted. The difference between a language like C and a language like assembly is that C source code does not have the direct one-to-one relationship with machine language that assembly source code does. A single statement in C may be compiled into several machine language instructions. Still, C programs run much faster than BASIC programs, because the entire conversion is done before the program is run.

## Compilation

What exactly goes on during a compilation depends on the compiler you're using. There are no set rules, except that it's the compiler's responsibility to take the source code and turn it into object code, the machine-language version of the program. To accomplish this, some compilers make several "passes" over the source code, while others, such as Megamax C, make only one pass.

The one-pass compiler is much faster than the others, but that speed comes with certain disadvantages. For instance, a multi-pass compiler usually converts the source code into assembly code, then assembles the assembly code into the object code. (The Alcyon compiler works this way.) One of the advantages of this multi-step process is that the assembly code that is produced by the compiler can be modified by the programmer before it is assembled and linked. This way, the programmer can do some code optimizing on sections of the program that may not run as fast as he'd like. In addition, the assembly-language listings produced by the compiler can be helpful in locating hard-to-find bugs in the program (assuming, of course, that you are familiar with 68000 assembly language).

The Megamax compiler is a one-pass compiler. It takes our source code and converts it directly into a machine-language module. Because no assembly-language file is created during the compilation, we don't have the option of "tweaking" the program. However, to make up for this, Megamax allows us to place assembly-language code directly into our source code, allowing us to speed up sections of our programs that may need optimizing. In addition, we can use a disassembler to turn the object module into assembly code.

Another important thing we need to know about the compiler is that it can substitute machine-language instructions only for text within the source code that it recognizes as C keywords or C operations. Generally, the process goes something like this: the compiler grabs a line of source code and compares what it finds there to a list of instructions it's able to handle. If it finds a match, it writes to the object file it's creating the machine-language code that represents the C instruction it found. If it doesn't find a match, it sets aside the instruction and goes on to the next.

For example, let's say the compiler has just read in this line:

```
for (x=0; x<10; ++x)
```

This is a standard for/next loop, and the compiler knows exactly what to do with it. The keyword `for` will be in its list of acceptable instructions and the values to use in the loop are found within the source line itself. The only stumbling block is the variable `x`. If `x` has been defined properly, its address will be found in a table of addresses the compiler has built. If `x` isn't found in the table, the compiler will generate an error.

Now let's say the compiler reads in this line:

```
v_bar(handle, pxy);
```

The compiler can check for the variables `handle` and `pxy` to make sure they're in its table. If they're found in the table, the compiler is satisfied. If they're not in the table, an error is generated. But what about the label `v_bar()`? It's a function, not a keyword, so it won't be found in the compiler's list of instructions. The compiler has no idea of what to do with `v_bar()`, so it just assumes that it'll run

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

across the label for this function somewhere else in the program. It leaves a space for its address and moves on to the next line.

If `v_bar()` happened to be one of our own functions, the compiler would come across it sooner or later and store its address in the space it reserved for that address. (This is called "back patching," and not all compilers do this. Sometimes patching in the address is left to the linker.) But, as you know, `v_bar()` is a VDI function. The function itself will not be found in our source code. Does this little problem upset the compiler? Nope. The compiler couldn't care less about the absence of a function. It'll just assume that the function we're calling will be found in another module, and pass the problem along to the linker.

### Linking

It's important to realize that the code produced by the compiler, even though it's in machine-language form, is not executable. In that object module are many "references" that need to be resolved, such as `v_bar()` from the above example. Essentially, what the compiler has passed on to the linker is an object module containing all the machine code generated from our source code, but missing much of the machine-language code it needs to become executable.

When the compiler came across our call to `v_bar()`, for instance, it didn't know where the code for this mysterious function was -- so it left a blank for the linker to handle. When we link the program, the linker will add the code needed to perform `v_bar()` and patch the address of that code into the blank space left by the compiler.

What is the address of `v_bar()`? Well, we don't know. All (well, almost all) of the programs that run on an ST must be "relocatable" -- that is, they must be able to run anywhere in your ST's memory. This causes a problem for the linker when it comes to addresses, because the addresses of functions and data will change depending on where the program is loaded in memory. I said the linker must supply the addresses, right? How can the linker supply an address for a relocatable program that has yet to be loaded in memory?

In a way, it can't. All the addresses generated during the compile and link process are actually offsets from the beginning of the program. The beginning of the program is given the address of zero. When you load an executable program into your ST's memory, the program loader replaces these offsets with real addresses. It sounds tricky, but there's really nothing to it. All the loader has to do is add the offsets already generated during the compile and link to the address the program is being loaded at. This sum will be an absolute address. Simple, eh?

Although we don't know at link time the absolute address of `v_bar()` (or any other function), we do know where the code for calling this function on a machine-language level can be found: it's in Megamax's system library, SYSLIB. In fact, SYSLIB contains the code for calling all the GEM and TOS functions listed in your Megamax manual. (Other compilers have a similar system library, but its name may be different.)

Notice I said above that SYSLIB contains the code for calling all the functions. The machine-language code that actually performs `v_bar()` and the other system functions are built in to your ST's operating system; it's part of GEM. The code found in SYSLIB "binds" the code generated by the compiler to the OS routines. This binding is necessary because the ST's operating system requires a lot of special handling. For instance, a VDI call needs to have some arrays filled in before it can do its work. When

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

programming in C, these arrays are invisible to us. But if we were programming in assembly language, we'd have to handle these arrays ourselves.

So the linker takes the code that was generated by the compiler and attempts to resolve all the missing addresses. In its attempt to do this, the linker will search through any other files to which we may be linking, as well as its own system files. When the linker finds the proper label in its table, it adds the machine code for the function to our already existing object module and patches in the address of the code. This continues, with the linker constantly adding code and resolving addresses, until it gets to the end of the object code module, at which point, we have a complete program.

### The File Types

Some people may be confused about all the different file types we encounter when putting together a program in C. There are really only three we need to be concerned with: .O, .H files and libraries.

The .O files are the object files we've been talking about. They are in machine-code form, but are not as yet executable. They need to be combined by the linker with the code that will make them complete programs.

When developing a program in C, it is advantageous to compile finished portions of the program into separate .O modules. This technique greatly speeds up compile time as our program gets bigger and bigger, since the code we've written previously doesn't need to be compiled every time; it just has to be linked to our new code.

Let's write a simple program that will illustrate some of the things we've been talking about. First, type in the following code under the filename TEST.C and compile it:

```
main ()
{
    print_text ( "This is a test." );
    gemdos (0x1);
}
```

After compilation you should have the file TEST.O on your disk. This file contains the machine-code equivalent of the C program shown above. The compiler has converted everything in the source code except the call to `print_text()`. The compiler can't do anything about this function because it doesn't know where or what it is. Did the compiler complain? Did you get an error? No. The compiler assumed you know what you're doing and left the missing-function problem for the linker to solve.

Now try to link TEST.O. What happened? After searching through all its libraries in vain, the linker told us that it didn't know anything about a function called `print_text()`. The linker passed the problem back to us. We have to solve the problem by writing the code for `print_text()`. Type the following under the filename PRINT.C and compile it:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
print_text ( string )
char *string;
{
    printf ( "%s\n", string );
}
```

You should now have on your disk the files TEST.O and PRINT.O. All we have to do to get an executable program is to link these two files together. Do that now, and then run the resultant program. Hurray! It works!

(Of course, the linker did more than just put together our two object modules; it also added other necessary code, such as the printf() routines from the system libraries.)

Megamax's libraries (SYSLIB, DOUBLE.L, and ACC.L) are really the same thing as .O files. They each contain the object code necessary to perform certain functions. We already talked about SYSLIB; you know what it is. The file DOUBLE.L is a machine-language module that, when linked into your program, replaces the regular floating point math routines with more accurate ones, allowing you to get greater precision (while sacrificing a little on speed). The ACC.L file needs to be linked to your program whenever you're writing a desk accessory, since desk accessories have to be initialized differently than regular programs.

Finally, we have the .H files. There is really no mystery here. These "header" files are included with your compiler as a convenience. Because there are hundreds and hundreds of standard names for various GEM parameters, as well as various standard structures that are used by GEM programmers, it would be silly to have to type all that stuff in every time we want to write a program. To save wear and tear on our keyboards (as well as our patience), all the commonly used data structures and names are provided for us. All we have to do is "include" them into our code.

We can do the same sort of thing when writing our own programs. To keep down the size of each module of our program, we can take all the #defines and global data declarations normally found at the top of your program and place them into a separate file. Traditionally, this type of file is given the .H extension. Let's say your main source code file is called MYPROG.C. You would then name the header file containing the data mentioned above into a file called MYPROG.H. Then, at the top of your program, you would have the line #include MYPROG.H so that the compiler would know where the code really belongs. Take a look at the .H files that came with your compiler, and you'll see that they are really nothing more than a collection of #defines and data declarations.

### Moving Along

For some of you, this excursion into the world of compilers and linkers was a rehash of information you were already familiar with. If there was nothing here for you, I apologize. But I know that there are many of you who have been taking the compilation process for granted, and many of you may have run into problems that you couldn't understand because you didn't know what was going on with your compiler. I hope this discussion cleared some of the clouds. If nothing else, I'm sure you gained some appreciation of what marvelous feats of programming compilers and linkers are.



### CHAPTER 26 - SIMPLE ANIMATION TECHNIQUES

Performing animation on the ST is a little bit tougher than it is on the 8-bit Atari computers, if for no other reason than the ST does not have player/missile graphics. The programmer is responsible for every step of the animation, getting little help from the hardware itself. Even so, coming up with a simple animation sequence is not particularly difficult -- as long as you are willing to do some preliminary work and are competent in handling MFDB's (Memory Form Definition Blocks) and raster operations.

In this chapter, we will study the creation of an animation sequence through each step of the process. But we will not review previously covered material; therefore, if you are not comfortable with MFDBs and the `vro_cpyfm()` function, I strongly advise that you review the chapter on raster operations.

#### The Program

Once you have the program compiled and linked, go ahead and run it. (Color only.) The screen will go blank, after which you should press the left mouse button. A space ship will appear on the right side of the screen, and a photon missile will start moving toward it from the left. When the missile collides with the ship, the ship will explode, and you'll once again be faced with a blank screen. Either press the left button to see the animation again, or the right button to exit back to the desktop.

#### The First Step

Before we program an animation, we must, of course, have something to animate. In other words, we must first take out a program like D.E.G.A.S. and draw the various figures that will make up the animation sequence.

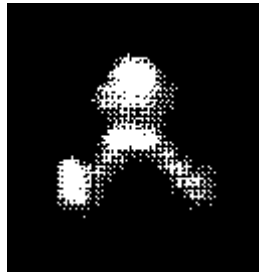
For example, if we wanted to have an exploding ship, we would start with the ship itself as frame one. Then we would take that ship and add a yellow glow to its center; this would be frame two. For frame three, we would expand the yellow glow. Frame four would show the ship completely engulfed by the glow, and in the last frame we would have the ship disintegrating into pieces.

In our sample animation, we also have a photon missile moving across the screen and hitting the alien ship, causing it to explode. The photon animation is made up of three frames.

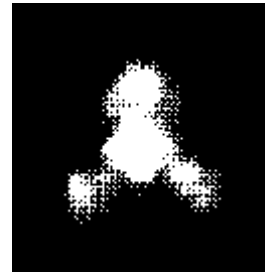
### EXPLODING SHIP ANIMATION FRAMES



Frame 1



Frame 2



Frame 3

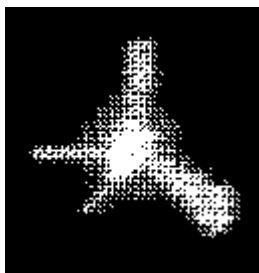


Frame 4

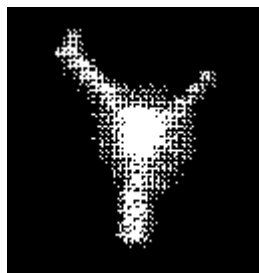


Frame 5

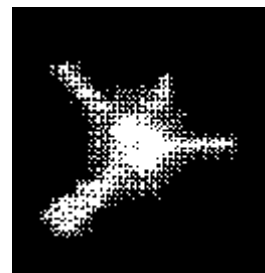
### PHOTON ANIMATION FRAMES



Frame 1



Frame 2



Frame 3

Once we have created all the figures we need for the animation, we must convert the graphics into numerical data. We do this using a sprite editor, such as Raster Sprite Editor from Issue 16 of ST-Log. These editors allow the user to "cut" a section of a picture and convert the cut section into the proper form of data. All we must do then is place the data into our program.

If you look at the top of Listing 1, you will see the graphics data for our sample animation. The data labeled `alien[]` is the alien ship. The data blocks labeled `expl1[]`, `expl2[]`, `expl3[]`, and `expl4[]` are the alien ship in its various stages of disintegration. Finally, the blocks of data labeled `photon1[]`, `photon2[]`, and `photon3[]` are the figures for our photon animation sequence.

### Programming the Animation

Now that we have all our figures converted to data, we must come up with a program that'll move that data on and off the screen in such a way as to create the actual animation. In the case of the

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

photon, we must copy each figure to the screen in sequence, while at the same time moving the photon toward the alien ship. The exploding alien ship will be a little easier to do, since we don't have to move the ship itself.

Let's take a look at the program listing. First turn your attention to the function `main()`. Here we initialize the application and open a virtual workstation. Then we check the resolution to make sure that the user is not trying to run the program in medium or high resolution. If we're in the wrong resolution, we bring up an alert box that informs the user of his error, after which we close the workstation and exit back to the desktop. If the screen resolution is okay, program execution goes to the function `do_animate()`. In this function, all we really do is monitor the mouse buttons. If the left button is pressed, we perform the animation, then come back to wait for another button press. If the right button is pressed, we exit the program.

### The Photon

When the left button is pressed, program execution jumps to the function `photon()`, where the actual animation begins. In `photon()` we first store the address of each image of our photon animation into the array of pointers, `ph[]`. We then turn off the mouse and draw the image of the alien ship on the right side of the screen using our own function, `draw_icon()`.

Next we initialize the X and Y coordinates of the photons, as well as the color stored in `pen`. The while loop that follows this initialization will repeat until `pen` is changed to one of the colors that make up the alien ship. Within the loop, we draw each stage of the photon animation, moving it slightly to the right each time. To animate the photon, we first draw the initial figure in the sequence, using `draw_icon()`. Then, in order to control the speed of the animation, we call the AES function, `evnt_timer()` like this:

```
evnt_timer( low, high );
```

Here, `low` is the low word of the number of milliseconds to pause and `high` is the high word of the number of milliseconds to pause. This function always returns a 1.

After the pause, we redraw the sprite in the same position. Because we are using a writing mode of 6, which Exclusive ORs the source and destination values, this second drawing of the sprite erases the first from the screen, leaving the screen exactly as it was before we drew the sprite. The only problem with this method of drawing a sprite over a background is that when the sprite is first drawn, it will appear transparent; that is, any graphics behind it will show through. In this case, because our display consists of nothing more than a black screen, we experience no problems when we Exclusive OR a sprite with the background.

After erasing the first sprite, we add 4 to `x`, the sprite's X coordinate, so that the next photon sprite will be drawn farther to the right, closer to the alien ship. Then we increment `p`, so the next time we use `p` to index our array of pointers, `ph[]`, we'll be pointing to the next image in the photon sequence. Finally, we get the color of the screen at our current location and call `evnt_timer()` to pause 50 milliseconds before going on to the next frame of the animation.

To retrieve the color of the pixel at our current screen location, we use a call to a VDI function, `v_get_pixel()`:

## C-MANSHIP COMPLETE – by CLAYTON WALNUT

```
v_get_pixel( handle, x, y, &pixel, &pen );
```

Here, handle is the virtual workstation handle, x and y are the X and Y coordinates of the pixel we wish to examine, &pixel is the address of an integer that will be set to 1 if the pixel being examined is set and set (unset?) to 0 if it's not, and &pen is the address of an integer that will contain the color value of the pixel being examined.

If the color detected is one of the colors that make up our ship (in this case, we're looking for RED or DRK\_RED), we know that the missile has collided with the ship. We drop out of the loop, after which program execution jumps to the function kill\_alien(), where we will perform the exploding-ship animation.

### Kaboom!

Because we don't need to actually move the ship, the explosion sequence is much easier. All we must do is draw the four frames of the animation one after the other, providing a short delay between them and erasing each image before drawing the next. This time around, we're erasing the images by drawing a black circle on top of them, rather than using the Exclusive OR method described above.

Once the main sequence is completed, we drop into the code that draws the "sparkles" on the screen -- just to add a little pizzazz. Here we're doing nothing more than using v\_pmarker() to draw 40 crosses in random locations within the area that the ship occupied. Because we're drawing them so fast, you would swear that there were many of them on the screen at the same time! When we're through in kill\_alien(), it's back to photon() to turn the mouse back on and then back one more step to do\_animate() to wait for the next mouse button click. A left click will repeat the animation, while a right click will exit the program.

### The End Again

As you can see, simple animation on the ST is not difficult. However, be aware that, unlike its 8-bit little brother, the ST is not well designed for animations involving many objects at once. The lack of player/missile graphics makes programming arcade-type action games a real chore. And there's really no way to do it in C; you have to have the speed of assembly language. Still, the techniques presented here can be effective and fun in simple applications. Experiment a bit, and see what you come up with. You might surprise yourself.

## Program Listing #1

```
/******  
*C-manship, Listing 1  
*CHAPTER 26  
*Developed with Laser C  
*****/  
#include <stdio.h>  
#include <osbind.h>  
  
#define BLACK0  
#define RED2  
#define WHITE8  
#define DRK_RED9  
#define TRUE 1  
#define FALSE0  
#define NONE 0  
#define LEFT 1  
#define RIGHT2  
#define LOW0  
#define REPLACE1  
#define DOT2  
#define M_OFF256  
#define M_ON 257  
  
/* GEM arrays */  
int work_in[11], work_out[57], contrl[12], intin[128],  
    ptsin[128], intout[128], ptsout[128];  
  
int desk_palette[16]; /* Desktop color palette. */  
  
/* Our own color palette. */  
int my_palette[16] = { 0x000,0x700,0x060,0x770,  
                      0x007,0x707,0x333,0x666,  
                      0x400,0x444,0x373,0x773,  
                      0x337,0x003,0x377,0x400 };  
  
int handle, /* GEM graphics handle */  
    dum; /* A dummy variable for storage of values */  
  
/* Data for sprites. */  
long alien[] = {  
    0x00000000,0x00000000,0x10000000,0x00000800,  
    0x00000000,0x00000000,0x20000000,0x00000400,  
    0x00000000,0x00000000,0x60000000,0x00000600,  
    0x00000000,0x00000000,0x78000000,0x00000600,  
    0x00000000,0x00000000,0x00000000,0x00003C00,  
    0x00000000,0x00000000,0x30000000,0x00000C00,  
    0x00000000,0x00000000,0x30000000,0x00004E00,  
    0x00000000,0x00000000,0x42000000,0x00002400,  
    0x00020000,0x00000000,0x83400040,0x00404040,  
    0x00030000,0x00000000,0x01C00040,0x00408040,  
    0x00030000,0x00000000,0x00800000,0x00000040};  
int alien_w = 32, alien_h = 10;  
  
long expl1[] = {  
    0x00000000,0x00000000,0x0C000000,0x00000000,  
    0x00000000,0x00000000,0x1E000000,0x00000000,  
    0x00000000,0x00000000,0x3F000C00,0x00000C00,
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
0x00000000,0x00000000,0x3F000000,0x00000000,
0x00000000,0x00000000,0x1E000C00,0x00000C00,
0x00000000,0x00000000,0x1E000C00,0x00000C00,
0x00000000,0x00000000,0x3F000C00,0x00000000,
0x00000000,0x00000000,0x3F000C00,0x00000C00,
0x00000000,0x00000000,0x61800000,0x00000000,
0x00010000,0x00000000,0xC0E00000,0x00000000,
0x00010000,0x00000000,0xC0E00000,0x00000000};
int expl_w = 32, expl_h= 10;

long expl2[] = {
    0x00000000,0x00000000,0x18000000,0x00000000,
    0x00000000,0x00000000,0x3C000000,0x00000000,
    0x00000000,0x00000000,0x7E001800,0x00001800,
    0x00000000,0x00000000,0x7E001800,0x00001800,
    0x00000000,0x00000000,0x3C003800,0x00003800,
    0x00000000,0x00000000,0x3C001800,0x00001800,
    0x00000000,0x00000000,0x7E001800,0x00001000,
    0x00000000,0x00000000,0x3C001800,0x00001800,
    0x00000000,0x00000000,0xFF003C00,0x00003C00,
    0x00030000,0x00000000,0x81C00000,0x00000000,
    0x00030000,0x00000000,0x81C00000,0x00000000};

long expl3[] = {
    0x003C0018,0x000000018,0x00000000,0x00000000,
    0x007E003C,0x00000003C,0x00000000,0x00000000,
    0x007E003C,0x00000003C,0x00000000,0x00000000,
    0x013D0019,0x000000019,0x00000000,0x00000000,
    0x07BD0799,0x00000799,0x00000000,0x00000000,
    0x03FF0199,0x00000181,0x80000000,0x00000000,
    0x01FF01FF,0x000001FF,0xC000C000,0x0000C000,
    0x00FF006D,0x0000006D,0x00000000,0x00000000,
    0x03B90038,0x00000038,0xC0000000,0x00000000,
    0x03AD002C,0x0000002C,0xC0000000,0x00000000,
    0x00260026,0x00000026,0x00000000,0x00000000};

long expl4[] = {
    0x01350004,0x00000000,0x00000000,0x00000000,
    0x02CA0282,0x00000082,0x00000000,0x00000000,
    0x01340010,0x00000000,0x80000000,0x00000000,
    0x04B500B1,0x00000031,0x00000000,0x00000000,
    0x0BAA0B00,0x00000A00,0x80008000,0x00000000,
    0x06190210,0x00000210,0x40000000,0x00000000,
    0x00FC00B0,0x00000000,0x80008000,0x00000000,
    0x04A604A0,0x000000A0,0x00000000,0x00000000,
    0x01490009,0x00000000,0x00000000,0x00000000,
    0x07330020,0x00000020,0x80000000,0x00000000,
    0x07030000,0x00000000,0x80000000,0x00000000};

long photon1[] = {
    0x00000000,0x00000000,0x00C00000,0x00000000,
    0x00000000,0x00000000,0x00C00000,0x00000000,
    0x00000000,0x00000000,0x00C00000,0x00000000,
    0x00000000,0x00000000,0x00C00000,0x00000000,
    0x00000000,0x00000000,0x00800000,0x00000340,
    0x00000000,0x00000000,0x00800000,0x00000760,
    0x00000000,0x00000000,0x01800000,0x00000E70,
    0x00000000,0x00000000,0x03C00080,0x00000C30,
    0x00000000,0x00000000,0xFFC00100,0x00000030,
    0x00000000,0x00000000,0x01F00100,0x00000E00,
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
0x00000000,0x00000000,0x023C0000,0x000005C0,
0x00000000,0x00000000,0x041F0000,0x000003C0,
0x00000000,0x00000000,0x080F0000,0x00000000,
0x00000000,0x00000000,0x00070000,0x00000000,
0x00000000,0x00000000,0x00020000,0x00000000,
0x00000000,0x00000000,0x00000000,0x00000000};
int photon_h = 15, photon_w = 32;
```

```
long photon2[] = {
0x00000000,0x00000000,0x20000000,0x00000000,
0x00000000,0x00000000,0x60000000,0x00000000,
0x00000000,0x00000000,0x30000000,0x00000000,
0x00000000,0x00000000,0x18040000,0x00000000,
0x00000000,0x00000000,0x0C080000,0x000003C0,
0x00000000,0x00000000,0x04100000,0x000003E0,
0x00000000,0x00000000,0x03E00100,0x00000C10,
0x00000000,0x00000000,0x03C00080,0x00000C30,
0x00000000,0x00000000,0x03C00100,0x00000C30,
0x00000000,0x00000000,0x01800000,0x00000E70,
0x00000000,0x00000000,0x01000000,0x000006E0,
0x00000000,0x00000000,0x01000000,0x000002C0,
0x00000000,0x00000000,0x03000000,0x00000000,
0x00000000,0x00000000,0x03000000,0x00000000,
0x00000000,0x00000000,0x03000000,0x00000000,
0x00000000,0x00000000,0x03000000,0x00000000};
```

```
long photon3[] = {
0x00000000,0x00000000,0x60000000,0x00000000,
0x00000000,0x00000000,0x30200000,0x00000000,
0x00000000,0x00000000,0x18200000,0x000003C0,
0x00000000,0x00000000,0x0C400000,0x000003A0,
0x00000000,0x00000000,0x03800000,0x00000C70,
0x00000000,0x00000000,0x03C00100,0x00000C30,
0x00000000,0x00000000,0x03FF0040,0x00000C00,
0x00000000,0x00000000,0x03800080,0x00000C70,
0x00000000,0x00000000,0x06000000,0x000001E0,
0x00000000,0x00000000,0x1C000000,0x000003C0,
0x00000000,0x00000000,0x78000000,0x00000000,
0x00000000,0x00000000,0xF0000000,0x00000000,
0x00000000,0x00000000,0x60000000,0x00000000,
0x00000000,0x00000000,0x00000000,0x00000000,
0x00000000,0x00000000,0x00000000,0x00000000,
0x00000000,0x00000000,0x00000000,0x00000000};
```

```
typedef struct fdbstr
{
    longfd_addr;
    int fd_w;
    int fd_h;
    int fd_wdwidth;
    int fd_stand;
    int fd_nplanes;
    int fd_r1;
    int fd_r2;
    int fd_r3;
} MFDB;
```

```
/*
* Main program.
*/
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
main()
{
    appl_init ();
    open_vwork ();
    if ( Getrez () != LOW )
        form_alert ( 1, "[1][This demo must be run|in low \
                        resolution][OK]" );
    else {
        do_animate ();
        Setpalette ( desk_palette );
    }
    v_clsvwk ( handle );
    appl_exit ();
}

/*****
* do_animate ()
*
* Main program loop. Reads the mouse buttons and goes to
* the appropriate functions based on the button presses.
*****/
do_animate ()
{
    int repeat, button, x, y;

    set_colors ();
    repeat = TRUE;
    while ( repeat ) {
        button = 0;
        while ( button == NONE )
            vq_mouse ( handle, &button, &x, &y );
        if ( button == LEFT )
            photon ();
        else if ( button == RIGHT )
            repeat = FALSE;
    }
}

/*****
* photon ()
*
* Performs the photon animation. The function checks for
* a "hit" by getting the color of the screen from the
* current photon position and comparing it to the alien
* ship's color.
*****/
photon ()
{
    int x, y, pixel, pen, p;
    long *ph[3];

    p = 0;
    ph[0] = photon1;
    ph[1] = photon2;
    ph[2] = photon3;
    graf_mouse ( M_OFF, 0L );
    draw_icon ( alien, 7, 250, 100, alien_w, alien_h );
    x = 20;
    y = 100;
```



## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
pen = BLACK;
while ( pen!=RED && pen!=DRK_RED ) {
    draw_icon ( ph[p], 6, x-6, y-6, photon_w, photon_h );
    evnt_timer ( 10, 0 );
    draw_icon ( ph[p], 6, x-6, y-6, photon_w, photon_h );
    x += 4;
    if ( ( p+=1 ) > 2 )
        p = 0;
    v_get_pixel ( handle, x+19, y+4, &pixel, &pen );
    evnt_timer ( 50, 0 );
}
kill_alien ();
graf_mouse ( M_ON, 0L );
}

/*****
* kill_alien ()
*
* Performs the exploding ship animation.
*****/
kill_alien ()
{
    int x, y, rx, ry, i;
    int pxy[2];

    x = 250;
    y = 100;
    vsf_color ( handle, BLACK );

    draw_icon ( expl1, 7, x, y, expl_w, expl_h );
    evnt_timer ( 50, 0 );
    v_circle ( handle, x+20, y+7, 10 );
    draw_icon ( expl2, 7, x+1, y, expl_w, expl_h );
    evnt_timer ( 50, 0 );
    v_circle ( handle, x+20, y+7, 10 );
    draw_icon ( expl3, 7, x+7, y, expl_w, expl_h );
    evnt_timer ( 50, 0 );
    v_circle ( handle, x+20, y+7, 10 );
    draw_icon ( expl4, 7, x+7, y, expl_w, expl_h );
    evnt_timer ( 50, 0 );
    v_circle ( handle, x+20, y+7, 10 );

    x += 13;
    y -= 4;
    vsm_type ( handle, DOT );
    vsm_height ( handle, 1 );
    for ( i=0; i<40; ++i ) {
        rx = rnd ( 16 );
        ry = rnd ( 16 );
        pxy[0] = x + rx;
        pxy[1] = y + ry;
        vsm_color ( handle, WHITE );
        v_pmarker ( handle, 1, pxy );
        evnt_timer ( 10, 0 );
        vsm_color ( handle, BLACK );
        v_pmarker ( handle, 1, pxy );
    }
}
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```

/*****
* draw_icon ()
*
* A general function for drawing icons. It may be used
* in other programs as long as the header file GEMDEFS.H
* has been included at the top of the program. The input
* to the function is the address of the icon's data, the
* desired drawing mode, the X and Y coords at which the
* icon should be drawn, and the width and height of the
* icon. For low resolution only.
*****/
draw_icon ( data, mode, dx, dy, width, height )
long data;
int mode, dx, dy, width, height;
{
    MFDB s_m, scr_m;
    int pxy[8];

    s_m.fd_addr = data;
    s_m.fd_w = width;
    s_m.fd_h = height;
    s_m.fd_wdwidth = width / 16;
    s_m.fd_stand = 0;
    s_m.fd_nplanes = 4;
    scr_m.fd_addr = 0;
    pxy[0] = 0;
    pxy[1] = 0;
    pxy[2] = width;
    pxy[3] = height;
    pxy[4] = dx;
    pxy[5] = dy;
    pxy[6] = dx + width;
    pxy[7] = dy + height;
    vro_cpyfm ( handle, mode, pxy, &s_m, &scr_m );
}

/*****
* set_colors ()
*
* This function stores the original desktop colors, and
* then installs the program's palette.
*****/
set_colors ()
{
    int x;

    graf_mouse ( M_OFF, 0L );
    for ( x=0; x<16; desk_palette [x++] = Setcolor ( x, -1 ) );
    v_clrwk ( handle );
    Setpalette ( my_palette );
    graf_mouse ( M_ON, 0L );
}

/*****
* open_vwork ()
*
* This function opens a virtual work station.
*****/
open_vwork ()
```

## C-MANSHIP COMPLETE - by CLAYTON WALNUT

```
{
    int i;
    handle = graf_handle ( &dum, &dum, &dum, &dum );
    for ( i=0; i<10; work_in[i++] = 1 );
    work_in[10] = 2;
    v_opnvwk ( work_in, &handle, work_out );
}

/*****
* rnd ()
*
* This function is used to get a random number from 0 to
* n-1. Its input is "n" and its output is the random number.
*****/
rnd ( n )
int n;
{
    int r;

    r = ( int ) Random ();
    r = abs ( r ) % n;
    return ( r );
}
•
```