COMPUTE!'s SECOND BOOK OF ATARI ST

Program Disk Enclosed

Games, utilities, tutorials, applications, and much more, including the award-winning game *Laser Chess*™. The enclosed disk includes all the programs, ready to run and enjoy on any 520 or 1040 Atari ST.

A **COMPUTE! Books** Publication $18.95

COMPUTE!'s SECOND BOOK OF ATARI ST

Program Disk Enclosed

COMPUTE!™ Publications, Inc.
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

The following articles were originally published in *COMPUTE!'s Atari ST Disk & Magazine*, copyright 1986, COMPUTE! Publications, Inc.: "AstroPanic!," "Encryptor," "Crash Analyzer," "Word Count," and "Why C?" (October); "ST-Graph," "File Hider," "*ST-Shell*™", "NEOview," and "Comparing C to BASIC" (December).

The following articles were originally published in *COMPUTE!'s Atari ST Disk & Magazine*, copyright 1987, COMPUTE! Publications, Inc.: "*Picture Puzzler*™," "Desktop Clock," "File Lister," "Snapshot *NEO/DEGAS*," "Extended Formatter," and "Choosing a Compiler" (February); "*Laser Chess*™," "Desktop Notepad," "Directory Dump," "File Finder," "Customizing the GEM Desktop," and "The C Programming Environment" (April).

The following article was originally published in *COMPUTE!* magazine, copyright 1987, COMPUTE! Publications, Inc.: "Full-Screen Shell for ST BASIC" (February).

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-098-X

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Atari, Atari 520ST, Atari 1040ST, ST, ST BASIC, and TOS are trademarks or registered trademarks of Atari Corporation. *Laser Chess, Picture Puzzler,* and *ST-Shell* are trademarks of COMPUTE! Publications, Inc.

Contents

Foreword ..... *v*

1. The Games 1
Laser Chess™
Mike Duppong 3
AstroPanic!
Charles Brannon 11
Picture Puzzler™
Douglas N. Wheeler 34
Spanish Castles
Robert S. Geiger 38
ST-GO
Kyle Cordes 43

2. Applications 45
ST-Graph
Michael P. Cohan 47
Desktop Clock
David Plotkin 59
Desktop Notepad
Tim Victor 62

3. Disk Utilities 73
File Hider
David T. Jarvis 75
File Lister
Richard Smereka 80
Directory Dump
Marcos Zorola 87
File Finder
Richard Smereka 92

4. Utilities 97
Encryptor
Douglas N. Wheeler 99
Crash Analyzer
George Miller 102
Word Count: A Writer's Accessory
Tony Roberts 107

ST Shell™
Richard Smereka 110
Snapshot *NEO/DEGAS*
Philip I. Nelson 125
Extended Formatter
Richard Smereka 133
Customizing the GEM Desktop
McKendre Haynes 139
NEOview
Philip I. Nelson 150
Full-Screen Shell for ST BASIC
David Lindsley 157

5. C Programming 161
Why C?
Sheldon Leemon 163
Comparing C to BASIC
Sheldon Leemon 168
Choosing a Compiler
Sheldon Leemon 176
The C Programming Environment
Sheldon Leemon 183

Appendix 193
How to Use the Disk 195

Index 198

Foreword

Classic games with new faces, handy desk accessories, practical applications, timesaving utilities, and instructive programming: With *COMPUTE!'s Second Book of Atari ST*, you'll continue to explore your ST's impressive potential. And you'll be more than satisfied, perhaps even surprised, by what you find.

For challenging fun, begin by testing your strategic powers with *Laser Chess*™—the First Prize winner in *COMPUTE!'s Atari ST Disk & Magazine*'s $10,000 programming contest. *Picture Puzzler*™, with three levels of difficulty, exercises the eye (and mind) to whatever degree of keenness suits your patience. Then there's "Snapshot *NEO/DEGAS*," a program that makes clever screen images possible for any prospective ST artist.

To help you keep important numbers balanced at home and at work, "ST-Graph" creates bar graphs, pie charts, line graphs, and more. "Customizing the GEM Desktop" and "Extended Formatter" represent the programs included to insure that your time at the computer is as productive as possible.

Ambitious programmers will welcome the commented source code featured with "AstroPanic!" and "NEOview," and those who are ready to make the transition from BASIC to C will find suggestions to help them choose a C compiler. Even the apprehensive will be eager to learn C when they discover the benefits of using this efficient language.

COMPUTE!'s Second Book of Atari ST brings together the best programs from *COMPUTE!'s Atari ST Disk & Magazine* in addition to those never before published. The accompanying disk contains all the programs presented in the book, so you won't have to worry about typing them in. The programs have been tested and are ready to be run. Put them to work for you today—and rediscover the ST.

CHAPTER ONE

Games

Laser Chess™

Mike Duppong

When COMPUTE!'s Atari ST Disk & Magazine *held its programming contest, the originality and skillful programming in* Laser Chess™ *brought it First Prize. A two-player strategy game patterned after traditional chess,* Laser Chess *features some fascinating new twists. The program runs in the low-resolution screen mode on any ST with a color monitor.*

Laser Chess™, as the name implies, is a chesslike strategy game for two players. The goal is to manipulate a laser-firing piece and various reflective objects to eliminate your opponent's king. As in traditional chess, there are an infinite number of ways to accomplish this.

There are eight basic types of pieces in *Laser Chess*, and each has unique capabilities. Over time, you'll learn each piece's advantages and limitations. Obviously, the more you play *Laser Chess*, the more you'll understand the pieces in your arsenal, which in turn will make you a better player. So let's start with a description of the pieces.

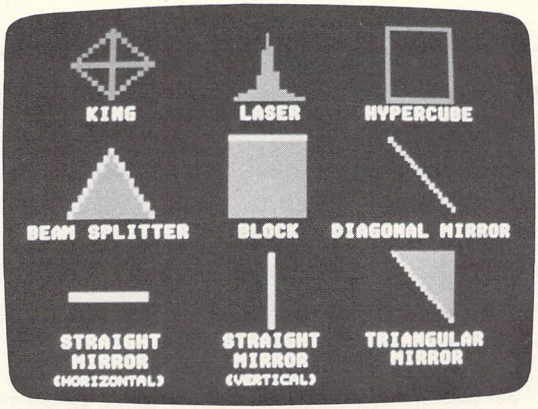
A Geometric Army

Figure 1-1 shows each piece and its name. Notice that certain sides of some pieces appear thickened (or are highlighted on a color display). This indicates a reflective surface. When a laser beam strikes a reflective surface, it bounces off without harming the piece. But if a piece is hit by a laser on a nonreflective surface, it is destroyed.

A piece can also be removed from the board if it is captured by an opposing piece. This is similar to traditional chess; to capture a piece, you simply move one of your own pieces onto its square.

In addition to their ability to move from square to square, pieces with reflective surfaces can also be rotated in place in 90-degree increments. This lets you orient the piece to protect it against opposing laser shots or to set up bounce shots with your own laser piece.

Figure 1-1. The Basic Types of Pieces in *Laser Chess*



The *king* is the most important piece in *Laser Chess*. When a player's king is eliminated, the other player wins the game. Since it has no reflective surfaces, it can be destroyed by a laser from any angle. It can also be captured by an opposing piece. The king is not totally defenseless, however. It can capture any opposing piece by moving onto its square. But this can be done only once per turn.

The second most important piece is the *laser*. This piece is your primary offensive weapon; it's the only piece which can fire a laser shot. When you're ready to take aim, it can be rotated in place at 90-degree angles. But like the king, it is completely vulnerable to enemy laser strikes, because it has no reflective surfaces. If you lose your laser, the game is not over, but only the most skillful (or incredibly lucky) player can overcome its loss.

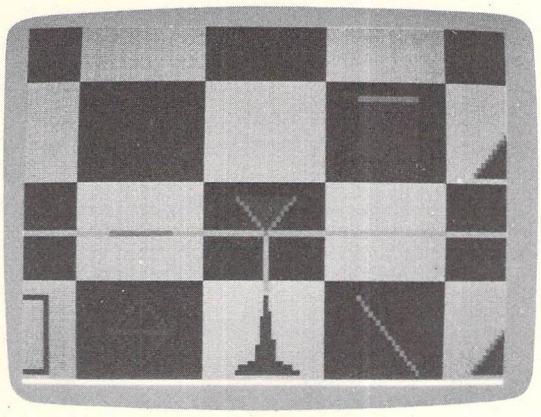
Tricky Pieces

The *hypercube* is an interesting piece. It can't harm an opposing piece directly, but may very well do so indirectly. When the hypercube is moved onto another piece (even your own), that piece disappears from its original position and reappears on a randomly selected empty square. This can be done only once per turn. Thus, the hypercube is a two-edged sword: It may relocate a piece to a vulnerable position, or it can make it possible for the piece to capture an important opposing piece on the next move. The hypercube has no reflective surfaces and cannot be rotated. It is invulnerable to laser shots, however, because it's made of

transparent glass—a laser beam passes right through it. Remember that.

The *beam splitter* is another tricky piece. When a laser beam strikes a splitter's vertex (the point opposite its base), the beam splits in two. The two new beams travel in opposite directions, perpendicular to the original beam's path (Figure 1-2). When a laser shot hits one of the beam splitter's reflective surfaces, it bounces off at a 90-degree angle *without* splitting. If the beam splitter's base is hit by a laser shot, it is destroyed. The beam splitter can be rotated.

Figure 1-2. Beam Splitter (magnified view): Two New Beams, Perpendicular to the Original, Reflected from Its Vertex



The *blocks* are fairly simple pieces. However, they may impose some complex situations. A block can capture any opposing piece by moving onto that piece's square, much like a king. But, unlike a king, a block has one reflective side and can be rotated as the situation demands. Therefore, blocks can be used either offensively or defensively. A laser beam that hits the reflective surface of a block is deflected 180 degrees—bouncing the beam back where it came from.

A *diagonal mirror* cannot be destroyed by a laser, because both of its surfaces are reflective. Diagonal mirrors can be removed from the board only when captured by a block or a king. When a laser beam strikes a diagonal mirror, the beam is deflected 90 degrees. Diagonal mirrors can be flipped to their opposite diagonal, but cannot be rotated to face horizontally or vertically.

The *horizontal mirrors* and *vertical mirrors* (known collectively as *straight mirrors*) are also invulnerable to lasers due to their reflective surfaces. When a laser hits a straight mirror on its flat surface, the beam is deflected 180 degrees. But if the laser hits a straight mirror edgewise, the beam passes straight through it. (Look closely at Figure 1-2; a laser beam is passing through a horizontal mirror just to the left of the beam splitter.) Straight mirrors can be rotated to become either horizontal or vertical mirrors, but not diagonal mirrors.

The *triangular mirrors* deflect laser beams just as diagonal mirrors do, but they are vulnerable to hits on their two non-reflective sides. A triangular mirror can be rotated in 90-degree increments.

Making Moves

All game functions are controlled with the mouse. Each player trades off the mouse after each turn. If you have a color monitor, you'll notice that the mouse pointer changes color to show whose turn it is.

The red player (at the bottom of the screen, unless you've reoriented the board as described below) always gets the first move. There's no particular advantage or disadvantage to moving first.

A turn consists of two moves. The number of moves remaining in a turn is indicated by the number of boxes in the rectangle on the left side of the screen. (See Figure 1-3.)

Before you move or rotate a piece, you must select it. This is done by pointing to the desired piece with the mouse and clicking the left mouse button. You don't have to point directly at the piece; the mouse pointer may be anywhere within the square. When a piece is selected, its square is highlighted.

If you accidentally select the wrong piece, you can deselect it by clicking the left mouse button again while the pointer is within the highlighted square. (This won't cost you a move.) Deselecting is usually done after you've rotated a piece.

After you've selected a piece, your next decision is whether to move or rotate it. To move a piece, simply point to the destination square and click the left mouse button. Moving a distance of one square takes one move; moving two squares takes two moves (although you can move a piece two squares in one step). Since you have only two moves per turn, the maximum distance a piece can be moved in one turn is two squares. If you

try to move too far, the computer beeps to signal your error.

Pieces can be moved forward, backward, left, or right, but not diagonally. You can effectively move a piece diagonally by using two moves—forward and right, for instance. You can do this in a single action by simply pointing to the adjacent diagonal square and clicking the left mouse button; if there's a clear path, the program moves the piece to the square and charges you two moves (one full turn).

You cannot move a piece through other pieces. The only exceptions are captures with blocks and kings, and moves of the hypercube as described above.

Rotating a Piece

To rotate a piece, select it and firmly press the right mouse button. If it's not legal to rotate that particular piece, the program beeps. Otherwise, the piece rotates 90 degrees (a quarter-turn) clockwise. You may continue rotating the piece to any desired position before deselecting it. Rotating a piece to face any direction takes only one move, and the move is subtracted after the piece is deselected. If you deselect the piece in its original position, no move is subtracted.

You can combine a rotation and a move in a single action. First, select the piece. Then rotate it to the direction you wish it to face. Finally, point to any adjacent square (except a diagonal), and click the left mouse button. The piece moves to that square and faces in the direction you chose. Since rotating a piece and moving a piece each take one move, this uses up your turn.

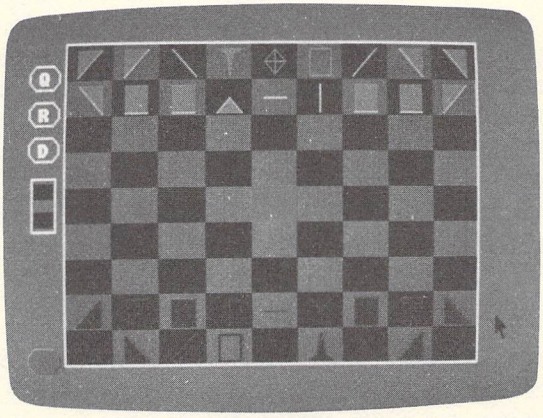
As mentioned above, the number of moves remaining in your turn is represented by the boxes inside the rectangle on the left side of the screen. If you wish to forfeit your entire turn or any remaining moves, just move the mouse pointer inside the square and click either mouse button.

Special Features

At the center of the 9 × 9 board is a special square called a *hypersquare.* It absorbs laser beams and acts as a stationary hypercube. That is, if you try to move a piece onto it, the piece disappears from its original position and reappears randomly on an empty square. This may be done only once per turn, however.

Along the board on the left side of the screen are four octagonal shapes. The top three octagons are labeled with letters *Q*, *R*, and *D*. The four octagons are screen buttons, similar to those

Figure 1-3. Full-Screen View of *Laser Chess*, with Its 9 × 9 Board Grid and Game Controls



found in both dialog and alert boxes on the ST. They can be pressed by moving the mouse pointer inside the octagon and clicking the left mouse button.

The top octagon, labeled Q, is the Quit button. When it's selected, a dialog box appears and requests confirmation. If you confirm that you want to quit, the game is aborted and the program returns you to the GEM desktop.

Beneath the Quit button is the Restart button. This lets you start a new game without finishing the current game. (For instance, a player may be so hopelessly behind that he or she wants to resign.) Again, a dialog box appears, requesting that you confirm this choice.

Below the Restart button is the Direction of Play button, labeled D. Each time you press this button, the entire board rotates 90 degrees, so you can play in a left/right direction or place the red pieces at the top of the screen instead of at the bottom.

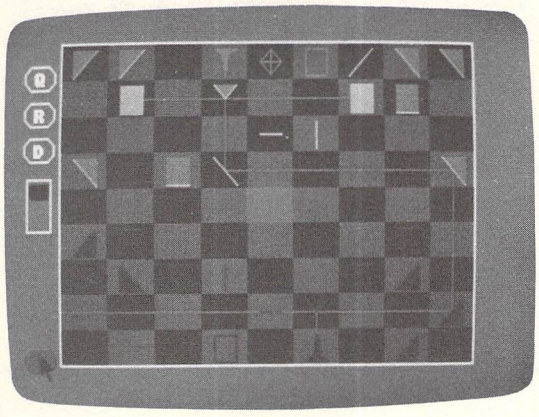
Firing the Laser

The last octagonal button, which is unlabeled, is the laser trigger. When it's your turn, you can select this button to fire your laser. If your laser piece has been captured or destroyed, the laser button won't appear on the screen during your turn.

The laser beam flashes on the screen when you press the left mouse button, and remains there until you release the left mouse button. (See Figure 1-4.) It's usually a good idea to hold the button down for a few seconds, so you can see the effect of

your shot. If you click the button too quickly, the beam may disappear before you can comprehend a complex bounce pattern.

Figure 1-4. Using the Bounce Pattern to Advantage: The Beam Splitter (near the top) Allows a Laser to Destroy Two Blocks



Firing your laser takes only one move, but can be done only once per turn. Therefore, you may want to use your first move in a turn to aim the laser, rotate a reflecting piece to set up a bounce shot, or move another piece into position. Of course, you won't necessarily be firing the laser on every turn. Much of the strategy in *Laser Chess* involves moving and rotating your pieces to set up complex shots.

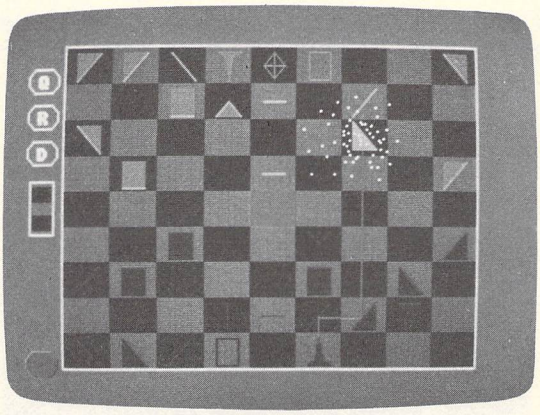
It's critically important to realize that *any* laser hit on a piece's nonreflective or nontransparent surface will destroy that piece (Figure 1-5). You can just as easily destroy your own pieces as well as your opponent's. You can even zap your own laser, particularly if you fire directly into the 180-degree reflective surface of a straight mirror or block, or if you fail to anticipate the effects of a beam splitter. Be forewarned.

Advice on Play

Get your mirrors out early. Use them to gain the fullest potential of your laser. Try to position mirror networks on both sides of the beam splitter so you can inflict as much damage as possible.

Take advantage of the blocks. Since they "control" an area around them with their threat of capture, no other pieces can safely move within their range. Make your opponent work to

Figure 1-5. A Triangular Mirror—Hit by a Laser Beam on a Vulnerable Surface—Exploding into Bits



displace them. Remember to rotate the reflective side of a block to the most probable direction of laser fire. If you can prevent a laser from destroying the block, your opponent will most likely have to gang up on it with two or more of his or her own blocks.

Use mirrors to protect your king. If you surround your king with straight and diagonal mirrors, there is no way it can be hit by a laser. As a result, your opponent will have to break through your defense with blocks. (This is a pretty dirty trick, because if your opponent loses all of his or her blocks, your king is almost invulnerable.) Defending your king with blocks is also a good strategy.

The hypercube should be used sparingly, since you have no idea where a relocated piece will reappear. Most players use the hypercube as a last resort—if another piece is going to be destroyed anyway, it doesn't hurt to take a chance and relocate it with the hypercube. Also, if your opponent's king is encircled with mirrors, you can march right in with your hypercube, followed by a block. This tactic may displace your opponent's defense, forcing evacuation of the enemy king from its mirrored fortress. Escorting the hypercube with an adjacent block prevents the opponent from attacking the hypercube with his or her king. Your opponent's only options will be to flee or be displaced.

AstroPanic!

Charles Brannon

Alien ships weave about, bobbing and diving. Don't let them hypnotize you, though—it's your duty to stop the cosmic horde from achieving total dominance of your monitor screen. This entertaining action game works on any 520ST or 1040ST with either a color or monochrome monitor in all three screen resolutions.

Just when you're beginning to think life is a picnic, here they come. That's right—the aliens—strange, wicked creatures from another world (or who knows, perhaps another dimension altogether). They have entered earth orbit, and their six-ship attack squadrons have managed to penetrate earth's orbital defense system, one wave after another. You're earth's last hope, the hottest laser jock yet to graduate from Defense Command's rigorous training program.

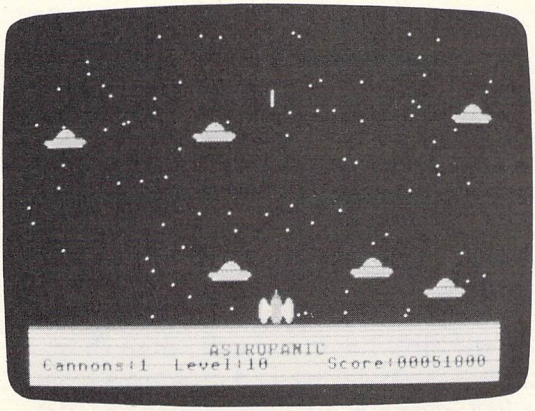
Via a video link, you control the massive neutron-beam cannon, an instrument of fury that hurls a devastating bolt of matter-shredding energy. Since no mirror system can deflect this beam without itself being destroyed, the neutron cannon is shuttled back and forth at high speed across a magnetic levitation (*maglev*) track. The aliens know that the cannon is too heavily shielded to be attacked by energy weapons, so they use the only tactic possible—a *kamikaze* strike.

The aliens bounce about (in an attempt to evade your shots while calculating the best collision trajectory), then careen in for a confrontation. The experts at Defense Command have anticipated even this mad strategy, so at horrendous cost they have manufactured *three* neutron cannons, each popping up to replace the previous one. But after the three cannons are vaporized, there are no more chances left—the invaders will finally achieve their victory.

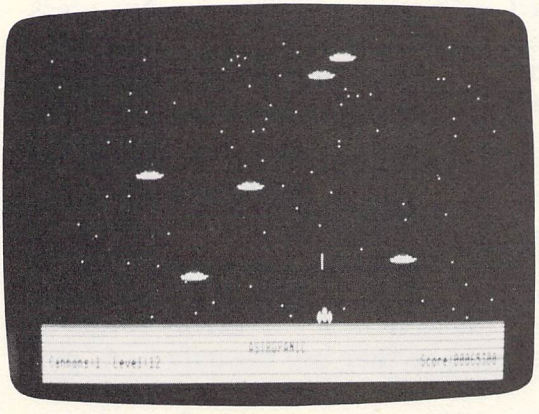
Playing AstroPanic!

Choose the screen resolution you want with Set Preferences; then run PANIC.PRG from the disk menu or from the desktop by double-clicking on the file. The game automatically adjusts itself to the screen resolution you have selected.

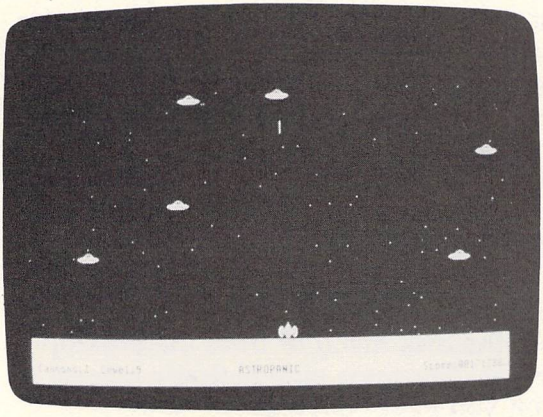
Figure 1-6. Three Games in One: AstroPanic!'s Challenge Changes with the Resolution



Large, finely detailed alien saucers crowd the screen in the low-resolution game.



In medium resolution, saucers are smaller in relation to the sky, so they're harder to hit.



High resolution produces tiny saucers moving through the big sky on the monochrome screen.

"AstroPanic!" is almost like three games in one. In low resolution, the saucers are large, multicolored objects that reveal lots of detail and crowd the screen. In medium resolution, the objects are smaller and show less detail. Since the saucers are smaller in relation to the screen, the "sky" seems bigger, so the saucers are harder to hit. In high resolution, the saucers are even smaller, and the sky is even bigger. If you're lucky enough to have both a color and a monochrome monitor, try playing the game in all three resolutions to experience the differences.

When you run AstroPanic!, a *Let's Play!* button appears in the center of the screen. Click on it and get ready to start shooting—the aliens immediately swarm into action. Use the mouse to move your cannon left and right across the bottom of the screen, and press the *right* mouse button to fire.

The awesome energies of the neutron cannon aren't unleashed casually—you can only fire one bolt at a time. If you see that your shot has already missed, though, you can fire another bolt immediately, canceling the previous shot. Keep your cannon moving (to avoid destruction), and watch out for the edges of the screen, where the saucers ricochet off the sides. When you have destroyed all six ships, you move on to the next level.

A status display at the bottom of the screen shows the number of cannons remaining, the current level, and your score. The closer the alien is to the bottom of the screen, the more points you get for hitting it. Every time you hit a saucer, the remaining ones move a little bit faster. The speed of the survivors gradually increases as you move through each level of the game, and all of the saucers speed up after each level. By the time you reach level 20, you're bound to lose. Try it; you'll see.

How It Works

Accompanying this discussion of AstroPanic! is a copy of the source code interspersed with extra comments (in bold). If you're interested in writing your own ST games or simply learning more about C, you should find the source code very educational.

Perhaps the most remarkable aspect of the game is its speed, considering that it is written in a high-level language. In addition, the executable program is only about 13K long. This is due partly to the exceptional efficiency of the compiler. We examined a disassembly of the compiler's object code and found that it

would be difficult to write much better code by hand in machine language. For example, integer variable assignments such as SCORE=0 can translate into a single 68000 instruction. A program such as a game can exploit this efficiency by using only integer math, thus avoiding some of the larger C library modules.

The ST video hardware doesn't have any provision for sprites. On machines like the Atari 400/800/XL/XE, Commodore 64, and Amiga, sprites greatly simplify game programming (or any programming that employs movable objects). Sprites exist on a separate video plane, so they don't interfere with an underlying background display. Since the video hardware merges the sprites with the video at hardware speed, sprites can be moved quickly without tying up the microprocessor. On the other hand, the 68000 has power to spare—it can easily simulate sprites by virtue of its high-speed memory-moving capabilities.

ST Sprites?

The best way to simulate sprites on the ST would be to write your own routines in machine language. Yet AstroPanic! is written completely in C, using only documented operating system routines. The core of the animation is based on a function called *vrocpyfm( )*, which can be found in the Virtual Device Interface (VDI) library. It's used to copy a rectangular block from one area of memory to another. It can be used to copy one part of the screen to another, or to copy a shape from a memory buffer to any part of the screen.

These memory buffers are supported through a C language structure called a *memory form definition block*, or MFDB. The contents of an MFDB include a pointer to the memory containing the shape data; variables specifying the width, height, and number of bit planes (range of allowable colors) in the shape; as well as a flag specifying whether the format of the shape data conforms to the GEM standard or is machine-specific, using the same memory organization expected by the video hardware.

You can use two methods to animate an object without erasing the background graphics. The first method is to preserve and restore the background as the shape passes over it. Before drawing a shape, save in a buffer the rectangular area that would be overlapped by the shape. When you move the shape to the next position, you then restore the overwritten area from the buffer.

This works fine for one shape or for shapes that don't pass through each other. But imagine what happens when these kinds

of shapes do pass over each other. Each shape first saves the image of the shape it overlaps. After the shapes pass through each other, they have both restored the area they overlapped, leaving behind images of the shapes.

Another method relies on a special binary mathematical operation known as *exclusive OR* (XOR). The binary truth table for XOR is (0 XOR 0 = 0, 0 XOR 1 = 1, 1 XOR 0 = 1, 1 XOR 1 = 0). If you know something about binary math, you can see that XOR works much like binary OR, or even normal addition—except that when you XOR two 1's together, you get a 0. (Interestingly, binary addition yields the same result, but with a *carry* of 1 that must be added to the bit to the left.) When you copy a shape to the screen, you can specify the way the bits in the shape are combined with the bits in the background image.

A Magic Stamp

Let's use a simple example. On a monochrome ST system, white is represented by 0 and black by 1 (the opposite of most computers—the ST monitor displays its screen in reverse to simplify programming). If you XOR a black shape (1) against a white background (0), you see the shape 0 XOR 1 as 1 (black). On the other hand, if screen memory is filled with 1's (black), and you attempt to XOR a shape made out of 0's (white), you will see nothing, since 0 XOR 1 is 1 (black).

But notice what happens if you put a black shape against a white background, then copy the black shape back on top of itself. The first operation is 1 XOR 0 = 1. When you XOR the black shape on top of itself, though, the operation is 1 XOR 1 = 0—the shape has removed itself. This method works no matter what the background data is; XOR is a reversible operation.

One way to think of XOR animation is that you're using a rubber stamp inked with a magical negative ink—an ink that reverses the color of whatever it touches. Naturally, stamping twice is the same as not stamping a shape down at all. If you are careful, you can stamp two different shapes so that they overlap. Then, when you restamp these shapes, the background will be completely restored. The only problem is that the area where the shapes overlap is reversed. The 1's in the shapes XOR together in the overlapped area to give white.

It's a little more complicated with a color display, since the XOR is performed on the binary screen data. A binary pattern of 11 XORed with a binary pattern of 10 gives a result of 01. Two different colors, when overlapped, give a third color. Despite this color variation, though, using XOR is fast and effective as a technique for sprite simulation. When shapes are moving quickly, you rarely notice the strange overlap effects.

AstroPanic! uses the XOR method of animation. When a saucer moves, the program draws the shape, moves it to the next position, and erases the old shape. XOR lets us perform this erasure without cutting holes in the background display (the star field).

Simulated Simultaneity

The inner loop of AstroPanic! updates all of the moving objects in the game: the saucers, the cannon, and the missile in flight. Of course, all of these objects aren't really moving simultaneously—they just seem to be because the program alternates the animation quickly and smoothly. First, sprite 1 gets to move one notch, then sprite 2, then sprite 3, and so on. Then the program checks to see if the cannon should move and if the missile has been fired.

When a saucer is destroyed, the program stops drawing it, skipping to the remaining shapes instead. Since drawing the shapes consumes the most time, skipping a saucer makes it possible for the others to move that much faster. When there is just one saucer left, it moves six times faster than when there are six saucers being animated. To ease this, the program uses a delay loop to simulate the time taken to draw a shape when one of the saucers is no longer being displayed.

To prevent unsightly flicker, the cannon is redrawn only when it needs to move to a new position. Therefore, the saucers slow down somewhat if you keep moving the cannon. Since drawing and erasing the missile takes some time, too, it can slow down the animation loop. Despite all this, AstroPanic! is still quite effective.

If you are familiar with GEM, and have your GEM reference manuals handy, this program is fairly easy to follow, with a good sprinkling of comments and liberal use of meaningful variable and label names. The program is broken up into a number of small modules, which makes it easier to understand than if the whole game were written as a large main loop. Using these

modules is somewhat analogous to using subroutines in BASIC, except every module has its own private variables.

AstroPanic! contains most of the elements found in arcade games, and can serve as a model for your own game programming. Even if you aren't ready to write your own game from scratch, try modifying this game, customizing certain details; it can be very instructive—and fun.

AstroPanic! Annotated Listing

```
/* AstroPanic ST by Charles Brannon */
/* created June 12 1986 */
/* last modified June 27 1986 */
```

These are obligatory include files needed to access GEM and AES routines, and to call the standard input/output (STDIO) functions. You may need to use different header files with your C compiler; include only as many as are needed to eliminate undeclared identifier error messages.

```
#include <define.h>
#include <gemdefs.h>
#include <obdefs.h>
#include <osbind.h>
#include <stdio.h>
```

These are the raster operations that can be performed when copying a shape to the screen. They specify how the bits in the source are to be combined with the bits on the screen. XOR is eXclusive OR; REPLACE overwrites the display; and ERASE clears only the part of the display overlapped by 1-bits in the source image, while TRANSparent sets to 1 only the part of the display overlapped by 1-bits in the source image, leaving the background alone. REVerse TRANSparent *complements* (flips 0's to 1's, and 1's to 0's) the source image before merging the source with the background.

```
#define fdb_XOR 6
#define fdb_REPLACE 3
#define fdb_ERASE 4
#define fdb_TRANS 7
#define fdb_REVTRANS 13
```

This macro is used to extract a pseudorandom number from 0 to ($x-1$), using the XBIOS random number trap.

```
#define rnd(x) (Random( )%(x))
```

Other useful macros for hiding or displaying the arrow pointer. (The pointer doesn't need to be on the screen during the game.)

```
#define HIDE_MOUSE graf_mouse(M_OFF,&dummy)
#define SHOW_MOUSE graf_mouse(M_ON,&dummy)
```

Since we need to adjust the game throughout for color or monochrome mode, this definition makes the source code more readable.

```
#define COLORMODE work_out[35]
```

You can change NUMSPRITES and recompile the game to get more saucers, but this slows down the action. Using six saucers is the best compromise.

```
#define NUMSPRITES 6
```

The height and speed of the missile can also be changed if you want to fine-tune the game. However, since the saucers can move in increments of up to 12 pixels per move, don't make the missile move in too large an increment, or it will sometimes skip over the saucers. The collision routine checks to see whether a saucer is somewhere within the height of the missile, so a tall missile can prevent the fast-moving missile from skipping over a saucer (but also affects the speed of the game). Notice the use of the ternary conditional operator to select one of two values. If the value preceding the question mark is nonzero (true), the first value before the colon (:) is used as the value of the three-part expression; otherwise, the value following the colon is used. I use this technique in many parts of the source code.

```
#define MISSILE_H (COLORMODE? 8 : 16)
#define MISSILE_SPEED (COLORMODE? 7 : 12 )
```

TOPSCREEN is the top border from which the saucers bounce. TEXTBOX is the size of the score box at the bottom, in pixels (this is doubled in monochrome mode).

```
#define TOPSCREEN 4
#define TEXTBOX 32
```

```
/* global variables */
```

These declarations are required for the sake of GEM and the library routines.

```
int dummy,ch,cw;
int work_handle,contrl[12],pxyarray[10];
int intin[128],intout[128],ptsin[128],ptsout[128];
int work_in[11],work_out[57];
```

At least with *Megamax C*, I have to create an MFDB (memory form definition block) "manually." Each shape has an MFDB to specify the bit image and proportions of the object.

```
struct my_fdb
{
    char *fd_addr; /* address of raster */
    int fd_w; /* width in pixels */
    int fd_h; /* height in rows */
    int fd_wdwidth; /* width in words */
    int fd_stand; /* 0 for ST, 1 for standard */
    int fd_nplanes; /* how many planes */
    int fd_r1, fd_r2, fd_r3; /* reserved */
} saucer,screen,cannon;
```

The following comments explain the purposes of these variables.

```
int colortab[16][3]; /* used to save colors */
unsigned long score; /* you know what this is! */
int ships; /* how many cannons are left */
int missile; /* flag for whether missile is in flight or not */
int missile_x,missile_y; /* position of missile in flight */
int cannon_x, cannon_y; /* horizontal & vertical position of cannon */
int xborder,yborder; /* screen boundaries */
int x[NUMSPRITES],y[NUMSPRITES]; /* holds x/y position of sprites */
int xacc[NUMSPRITES],yacc[NUMSPRITES]; /* acceleration factors */
int isdead[NUMSPRITES]; /* is this sprite dead? */
int death_toll; /* saucers shot this round */
int textline; /* line where text box starts */
int round; /* current level of game */
int speed; /* saucer speed */
```

The main( ) routine controls the game. Most routines have meaningful names, so it is pretty easy to follow the "recipe" for AstroPanic!.

```
main( )
{
    int sprite; /* sprite index */
    int prev_x,prev_y; /* stores previous position of a sprite */
    appl_init( );
    init_workstation( );
```

Set screen to black and hide the arrow cursor.

```
    set_colors( );
    HIDE_MOUSE;
```

Fill the screen with stars.

```
    clear_sky( );
```

Wait for the player to click on the alert button to start.

```
    form_alert(1,"[1][AstroPanic!|Charles Brannon|(C) 1986 COMPUTE!][ Let's
        Play! ]");
```

Load the appropriate shapes.

```
    init_shapes( );
```

Here, we adjust the boundaries of the screen according to workstation width (work_out[0]) and height (work_out[1]). The height and width of the saucer are found in saucer.fd_h and saucer.fd_w, members of the my_fdb structure. The value of work_out[31] is 0 for a color screen or 1 for monochrome, so we can select the position of the text lines in the score box. This figure needs to be doubled (by left-shifting it by 1) for the 400-line monochrome mode.

```
    xborder=work_out[0]-saucer.fd_w-4;
    textline=work_out[1]-((COLORMODE)? TEXTBOX : TEXTBOX<<1);
    yborder=textline-cannon.fd_h;
```

Set the vertical position of the cannon.

```
cannon_y=yborder;
```

Set the score to 0 and the number of cannons to 3. Initialize the saucers (ufos) and draw the screen.

```
reset_game( );
/* ye olde main loope */
```

FOREVER is defined in *define.h* as *for(;;;)*, an infinite loop (exited with the Terminate( ) call when the game is over).

```
FOREVER
{
```

Look for a keypress and pause if one is found.

```
    check_for_pause( );
```

Begin looping for all sprites, updating their positions and images.

```
    for (sprite=0;sprite<NUMSPRITES;sprite++)
    {
```

If a sprite has been shot, we don't draw it, but we do delay a bit via dumdum( ) to simulate the time it would have taken to draw the image. That way the surviving saucers don't speed up too much. However, this delay loop is decreased after each wave, so the survivors gradually move faster and faster. The *continue* keyboard breaks out of the current iteration of the loop, but continues with the statement following the above *for* (as opposed to *break*, which terminates the entire loop).

```
        if (isdead[sprite]) { dumdum( ); continue; }
```

We remember the previous position of the "sprite," since we are about to add in the horizontal and vertical acceleration factors (displacement from current position) in order to move the saucer to the next position.

```
        prev_x=x[sprite]; prev_y=y[sprite];
        x[sprite]+=xacc[sprite];
        y[sprite]+=yacc[sprite];
```

If the position exceeds the screen boundaries, we reverse the direction by negating the displacement, and reset the position to the previous position so that the shape doesn't escape from the screen.

```
        if (x[sprite]<4 | x[sprite]>xborder)
            xacc[sprite]=-xacc[sprite],x[sprite]=prev_x;
        if (y[sprite]<TOPSCREEN | y[sprite]>yborder)
            yacc[sprite]=-yacc[sprite],y[sprite]=prev_y;
```

The first put( ) erases the previous image of the saucer; the second updates it at the new position.

```
        put(&saucer,prev_x,prev_y,fdb_XOR);
        put(&saucer,x[sprite],y[sprite],fdb_XOR);
```

If this sprite is within eight pixels of the saucer, we check to see whether the horizontal positions of the sprite and the saucer overlap. If so, the cannon is destroyed and the loop is canceled with *break*, since we will call *init_ufos( )* within the *kill_cannon( )* routine to reset the saucer positions for the next round.

```
                if (cannon_y-y[sprite]<8)
                    if ( (cannon_x>=x[sprite] &&
cannon_x<=x[sprite]+saucer.fd_w) |
                        (cannon_x+cannon.fd_w>=x[sprite] &&
cannon_x+cannon.fd_w<=x[sprite]+saucer.fd_w) )
                    {
                        kill_cannon( );
                        break;
                    }
            } /* end for */
```

We move the cannon and update the missile only after all the saucers have been updated; otherwise, the saucers would move too slowly. This works just fine, though. The missile is updated only if the missile-is-in-flight flag is active.

```
            move_cannon( );
            if (missile) update_missile( );
        }
}
```

End of the main function and the beginning of the supporting modules.

Terminate( ) ends the game when all the cannons have been destroyed. We restore the colors we've changed, close the virtual workstation, tell AES that we've finished, and exit( ) back to the desktop. The flag allows the calling routine to pass an error value back to the operating system (such as −1 for an emergency exit), but we really don't use this feature in this game.

```
Terminate(flag)
int flag;
{
    SHOW_MOUSE;
    reset_colors( );
    v_clsvwk(work_handle);
    appl_exit( );
    exit(flag);
}
```

The endpoint of the loop decreases as the round increases. Unfortunately, I had to use this busy-wait technique, which prevents background programs from running during the delay, since the shortest time waited for by the XBIOS Delay( ) routine (despite the claim of its millisecond resolution) is too long.

```
/* dummy routine, for short delay */
dumdum( )
{
```

```
    int i;
    for (i=0;i++<42-(round<<1));
}
```

We need to reset the game the first time it is run and also after the end of a game.

```
reset_game( )
{
    missile=FALSE; /* kill missile */
    clear_sky( );
    speed=2; /* maximum speed */
    score=round=death_toll=0; ships=3;
    update_scorebox( );
    cannon_x=0;
    put (&cannon,cannon_x,cannon_y,fdb_XOR); /* cannon appears */
    init_ufos( );
}
```

We plot 100 dots (a polyline with two identical coordinates) for a starfield background. Since the XOR animation technique used to move the saucers also preserves the background, it would be a shame not to provide a background to preserve.

```
/* fill sky with stars */
clear_sky( )
{
    int star;
    v_clrwk(work_handle);
    vsl_color(work_handle,1);
    vswr_mode(work_handle,1); /* replace */
    for (star=0;star<100;star++)
    {
        pxyarray[2]=pxyarray[0]=rnd(work_out[0]);
        pxyarray[3]=pxyarray[1]=rnd(work_out[1]);
        v_pline(work_handle,2,pxyarray);
    }
}
/* allow player to pause game by pressing a key */
```

The trick here is that *evnt_multi* normally waits for an event, but we just want to check for a keypress. If there is no keypress, we need to return to the main loop—otherwise the saucers will move only when you have pressed a key. The secret (divulged by Tim Oren in his GEM tutorials) is to wait for both a keypress event *and* a time event. It works because you are waiting for a duration of zero milliseconds, which makes *evnt_multi* return almost immediately. However, if the event that occurred was not a timer event, you know that the other event you were waiting for has happened. It's a little roundabout, but I couldn't find any other way to scan for any key within GEM or AES (I couldn't get *vsm_choice( )* to work, the next best thing). You have to really watch those zeros, though, to make sure the fields you *do* use fall in the right place. If there is a keystroke, we then wait for another keystroke with *evnt_keybd( )* before continuing.

```
check_for_pause( )
{
    int key,which;
    /* poll keyboard by waiting for a null time duration */
    which=evnt_multi(MU_TIMER|MU_KEYBD,0,0,0,0,0,0,0,0,0,0,0,0,0,
        &dummy,0,0,&dummy,&dummy,&dummy,&dummy,&key,&dummy);
    if (which & MU_KEYBD) evnt_keybd( );
}
/* initializes positions and vectors for saucers */
init_ufos( )
{
    int sprite;
    death_toll=0; /* no sprites dead yet */
```

For each sprite (saucer), we choose a random acceleration (from $-speed$ to $+speed$, where *speed* ranges from 2 to 8 or 12). A zero acceleration can't be used, though, or the saucers will move only horizontally, vertically, or not at all (if both xacc and yacc are zero). The saucer is then drawn on the screen so that the animation loop will have a previous image to erase; otherwise, the first put( ) in the animation cycle will leave an image behind.

```
    for (sprite=0;sprite<NUMSPRITES;sprite++)
    {
        isdead[sprite]=xacc[sprite]=yacc[sprite]=0;
        while (xacc[sprite]==0) xacc[sprite]=(speed>>1)-rnd(speed+1);
        while (yacc[sprite]==0) yacc[sprite]=(speed>>1)-rnd(speed+1);
        x[sprite]=8+rnd(xborder-8);
        y[sprite]=8+rnd(yborder-50);
        put (&saucer,x[sprite],y[sprite],fdb_XOR); /* make it appear */
    }
}
```

This one is chock-full of VDI calls. It redraws the score box every time a saucer is hit, so it has to be fast.

```
update_scorebox( )
{
    char temp[20];
    int y,d;
```

First we draw a solid white bar (rectangle) using replace mode.

```
    pxyarray[0]=0; pxyarray[1]=textline;
    pxyarray[2]=work_out[0]; pxyarray[3]=work_out[1];
    vswr_mode(work_handle,1); /* replace */
    vsf_color(work_handle,1); /* white */
    vsf_interior(work_handle,1); /* solid */
    v_bar(work_handle,pxyarray);
```

Now we draw an expanding series of green lines (black in monochrome mode) over the white box, just for looks. (All these colors assume the default color palette.) Transparent mode is initialized so that the background of a text cell doesn't erase the white box or the green lines.

```
    vswr_mode(work_handle,2); /* transparent */
    vsl_color(work_handle,COLORMODE? 3 : 0);
    for (y=d=0;y<TEXTBOX;y+=d++)
    {
        pxyarray[0]=0; pxyarray[1]=y+textline;
        pxyarray[2]=work_out[0]; pxyarray[3]=y+textline;
        v_pline(work_handle,2,pxyarray);
    }
    vsl_color(work_handle,1);
```

That ternary ? operator is really handy. It lets us choose either red or black, depending on the color mode.

```
    /* draw text in red, if possible */
    vst_color(work_handle,COLORMODE? 2 : 0);
```

The title is centered.

```
    vst_alignment(work_handle,1,0,&dummy,&dummy); /* center */
```

The text origin is given as the center of the screen (the width shifted right by 1, faster than dividing by 2). The variable *ch* is the character cell height, defined by *v_opnvwk( )*. The text is positioned one line lower in monochrome mode so that it isn't drawn through the horizontal line effect.

```
    v_gtext(work_handle,work_out[0]>>1,textline+(ch<<1)+
        (COLORMODE? 0 : ch),"ASTROPANIC");
```

In black, we prepare to draw the text with left alignment.

```
    vst_color(work_handle,0); /* draw text in black */
    vst_alignment(work_handle,0,0,&dummy,&dummy); /* left */
```

The sprintf( ) function works like printf( ) except that the output is stored in a string instead of appearing on the display. This makes formatted output possible with any display routine.

```
    sprintf(temp,"Cannons:%d Level:%d",ships,round+1);
    v_gtext(work_handle,cw,textline+ch*3,temp);
```

The score line is right-aligned, and the origin is one character cell width to the left of the screen's right margin.

```
    vst_alignment(work_handle,2,0,&dummy,&dummy); /* right */
    sprintf(temp,"Score:%07lu0",score);
    v_gtext(work_handle,work_out[0]-cw,textline+ch*3,temp);
    vst_alignment(work_handle,0,0,&dummy,&dummy);
    vst_color(work_handle,1);
}
```

The main loop calls this routine when a saucer collides with the cannon.

```
/* when cannon is hit, kill it */
kill_cannon( )
{
    int lum;
    put (&cannon,cannon_x,cannon_y,fdb_XOR); /* remove cannon */
```

In color mode, we'll turn the screen red, and, after the explosion, gradually decrease the luminance back to black while the explosion sound effect decays.

```
    if (COLORMODE) setcolor(0,1000,0,0); /* flash screen */
    else setcolor(0,1000,1000,1000); /* monochrome */
    explode(cannon_x+(cannon.fd_w>>1),cannon_y+(cannon.fd_h>>1)
        ,8,1);
    if (COLORMODE)
        for (lum=1000;lum>=0;setcolor(0,lum--,0,0));
    else setcolor(0,0,0,0);
    missile=FALSE; /* kill missile */
```

One ship less now, so we show this by updating the score box display.

```
    --ships; update_scorebox( );
```

The game is over when all the cannons are used up.

```
    if (ships==0) { end_game( ); return; }
    SHOW_MOUSE;
```

One nice touch is that we distinguish between the next and the last cannon, just to remind the player that this is the last chance.

```
    if (ships==1)
        form_alert(1,"[3][|Last Cannon|][Ready!]");
    else
        form_alert(1,"[3][|Next Cannon|][Ready!]");
    HIDE_MOUSE;
```

We reset part of the game by redrawing the screen and setting up new saucer positions. We can't use *reset_game( )* since it sets the score to 0 and the number of cannons back to 3.

```
    clear_sky( );
    update_scorebox( );
    cannon_x=0;
    put (&cannon,cannon_x,cannon_y,fdb_XOR); /* cannon appears */
    init_ufos( );
}
```

The explosion is used to destroy both the cannon and the saucers. It draws an expanding circle in white, then erases the circle from the inside. Because XOR mode is used, this explosion doesn't erase the background, and it improves the explosion effect, since some of the circles slightly overlap to create the illusion of an expanding shell. The game pauses when the saucer explodes, or else the explosion would be too slow. We need a longer duration for the explosion when the cannon is hit to leave time for the color effect. The sound effect happens in the background, allowing the program to continue immediately after Dosound( ).

```
/* explosion effect radiating from center */
/* flag controls duration of sound effect */
```

```
explode(xcenter,ycenter,radius,flag)
int xcenter,ycenter,radius,flag;
{
    static char boom[ ]=
    {0,0, 1,0, 2,0, 3,0, 4,0, 5,0, 6,63, 7,0xf7, 8,0x10, 9,0,
    10,0, 11,0, 12,10, 13,0, 255,0};
    int r;
    boom[25]=flag? 20 : 10;
    Dosound(boom);
    vswr_mode(work_handle,3); /* XOR */
    vsf_interior(work_handle,0); /* hollow circle */
    for (r=0;r<radius;v_circle(work_handle,xcenter,ycenter,r+=2));
    for (r=0;r<radius;v_circle(work_handle,xcenter,ycenter,r+=2));
    vswr_mode(work_handle,1); /* normal */
}
```

At the end of the game, if the player wants to play again, we reset the game. Otherwise, Terminate( ) cancels the program and returns us to the desktop.

```
end_game( )
{
    SHOW_MOUSE;
    if (form_alert(1,"[2][Play Again?][YES|NO]")==1)
    {
        reset_game( );
        HIDE_MOUSE;
    }
    else Terminate(0);
}
```

We periodically check for cannon movement and missile firing in the main loop.

```
/* moves cannon, checks for fire button */
move_cannon( )
{
    int button,x,y,oldx;
```

The *released* flag is used to make sure the player has released the mouse button before we allow another shot.

```
    static int released=TRUE;
    static char blip[ ]=
    {0,0, 1,0, 2,10, 3,0, 4,0, 5,0, 6,0, 7,0xfd, 8,0, 9,16,
    10,0, 11,0, 12,8, 13,4, 255,0};
```

We save the old position of the cannon and poll the mouse for the horizontal position.

```
    oldx=cannon_x;
    vq_mouse(work_handle,&button,&x,&y);
```

We use the right mouse button to fire the missile. The left mouse button is represented by bit 0, with a value of 1. The right button is bit 1, with a value of 2. (If both buttons are pressed, we get 1+2=3). The binary AND

masks out the unwanted bits. We also check to see whether the button is released. The *released* flag is set only if the right button test fails. If a missile is fired, *released* is set to 0 (FALSE).

```
if (button&2)
{
    if (released)
    {
        if (missile) draw_missile( ); /* erase old missile */
        Dosound(blip);
```

We fire the missile by enabling the missile-is-in-flight flag and giving the missile its horizontal and vertical positions. The first missile is drawn so that the *update_missile( )* routine has something to erase the first time through the loop.

```
        missile=TRUE; missile_x=cannon_x+(cannon.fd_h>>1);
        missile_y=cannon_y;
        draw_missile( );
        released=FALSE;
    }
}
else released=TRUE;
```

The horizontal mouse cursor position is used as the cannon's position, although the cannon can be wider than the cursor, so we use only the legal horizontal coordinates.

```
    cannon_x=(x<work_out[0]-cannon.fd_w)? x :
work_out[0]-cannon.fd_w;
```

We redraw the cannon only if it has moved, to eliminate flicker.

```
    if (cannon_x != oldx)
    {
        put (&cannon,oldx,cannon_y,fdb_XOR);
        put (&cannon,cannon_x,cannon_y,fdb_XOR);
    }
}
```

This is called in the main loop if a missile is in flight. Each call to this routine moves the missile up one notch (defined by MISSILE_SPEED) and then checks this missile position against all the sprites.

```
/* moves missile to next position, if missile is onscreen */
update_missile( )
{
    int sprite;
    draw_missile( ); /* erase old missile */
```

When the missile reaches the top of the screen, it is turned off; otherwise, we draw it at the new position.

```
    if ((missile_y-=MISSILE_SPEED)>TOPSCREEN)
        draw_missile( ); /* draw new missile */
    else missile=FALSE; /* end of mission */
```

The check for the missile collision seems complicated, but we're just checking to see whether the saucer lies anywhere within the bounds of the missile. Even though this loop executes six times every time the missile moves up one notch, it slows down the animation loop only a little.

```
    for (sprite=0;sprite<NUMSPRITES;sprite++)
    {
        if (!isdead[sprite] && missile_x>=x[sprite] &&
missile_x<=x[sprite]+saucer.fd_w)
            if ( (y[sprite]>=missile_y && y[sprite]<=missile_y+MISSILE_H)
                ||(y[sprite]+saucer.fd_h>=missile_y &&
                y[sprite]+saucer.fd_h<=missile_y+MISSILE_H) )
            {
                killsprite(sprite);
                break;
            }
    }
}
```

This routine removes a saucer (sprite) from the screen and from the animation loop.

```
killsprite(which)
int which;
{
    put (&saucer,x[which],y[which],fdb_XOR); /* remove saucer */
    draw_missile( ); /* remove missile */
    missile=FALSE;
```

Explode the poor devil.

```
    explode(x[which]+(saucer.fd_w>>1),y[which],8,0);
    isdead[which]=TRUE;
```

Your score is the vertical position of the saucer when it is hit. However, I see no reason why players in monochrome mode should get twice the score, since there are 400 scan lines in monochrome mode versus 200 in the color modes, so in monochrome mode we right-shift the value by 1 (which quickly divides by 2).

```
    score+=(COLORMODE? y[which] : (y[which]<<1));
    update_scorebox( );
```

We simultaneously increment the counter which keeps track of the number of saucers shot and check to see whether all the sprites have been shot. If so, new saucers are provided, and we move up to the next level of difficulty, to a maximum of 20. The speed of the saucers is allowed to reach 8 pixels per move in color mode or 12 in monochrome mode (since there's more screen real estate to be covered).

```
    if (++death_toll==NUMSPRITES)
    {
        init_ufos( );
        if (round<20)
```

```
        {
            round++; /* next round */
            update_scorebox( );
        }
        if (speed<(COLORMODE? 16 : 24)) speed++;
    }
}
```

This is the core routine for updating the missile, a polyline drawn in the XOR mode so that the missile doesn't erase any background.

```
/* draws missile at missile_x, missile_y, with XOR */
draw_missile( )
{
    vswr_mode(work_handle,3); /* XOR drawing mode */
    vsl_color(work_handle,1); /* white */
    pxyarray[0]=missile_x; pxyarray[1]=missile_y-MISSILE_H;
    pxyarray[2]=missile_x; pxyarray[3]=missile_y;
    v_pline(work_handle,2,pxyarray);
    vswr_mode(work_handle,1); /* replace mode */
    vsl_color(work_handle,1); /* black */
}
```

Before we choose custom colors, we save the existing colors in an array so that it can be restored when the game ends.

```
/* Saves colors in global array colortab[ ] */
save_colors( )
{
    int i;
    for (i=0;i<16;i++)
        vq_color(work_handle,i,0,colortab[i]);
}
```

This routine employs a convenient setcolor( ) routine which simplifies palette redefinition.

```
/* sets colors for this program */
set_colors( )
{
    save_colors( );
    setcolor(0,0,0,0); /* black */
    setcolor(1,1000,1000,1000); /* white */
}
```

It's easier to use this routine than to fill an array every time you want to call *vs_color( ).*

```
setcolor(index,red,green,blue)
int index,red,green,blue;
{
    int rgb_in[3];
    rgb_in[0]=red; rgb_in[1]=green; rgb_in[2]=blue;
    vs_color(work_handle,index,rgb_in);
}
```

The colors are restored when the game is over.

```
reset_colors( )
{
     int i;
     for (i=0;i<16;i++)
          vs_color(work_handle,i,colortab[i]);
}
```

This is one of the longest parts of the program, containing the data for the saucers and cannon. Since every screen resolution has different proportions and color capabilities, as well as varying internal memory layout, we need different shapes for every resolution. A program should always support both color and monochrome modes, but needn't work in both low and medium resolution. It's fun to take advantage of the characteristics of these modes, though: lo res is the most colorful—with big, detailed ships. The smaller ships in medium res make for the fastest game.

```
/* initializes the shapes according to screen resolution */
init_shapes( )
{
     screen.fd_addr=0; /* screen memory */
     switch (work_out[13]) /* number of colors */
     {
     /* hi res, 640 x 400 */
     case 2: ufo_high( );
               cannon_high( );
               break;
     /* medium res, 640 x 200 */
     case 4: ufo_med( );
               cannon_med( );
               break;
     /* lo res, 320 x 200 */
     case 16: ufo_low( );
                cannon_low( );
                break;
     }
}
```

Each word of data represents 16 pixels. The raster array is filled; then the *fd_addr* field of the FDB structure is filled in by pointing to the array. This member is defined as a pointer to a *char*, so we cast the array name (which is similar to "pointer to *int*") to avoid compiler warnings.

```
/* initializes data for hi-res saucer shape */
ufo_high( )
{
     static int ufohigh[ ]=
     {7,0x8000,0x18,0x6000,0x20,0x1000,0x40,0x800,0x1ff,0xfc00,
     0x1e49,0x27c0,0x7fff,0xfff0,0x8000,8,0x6aaa,0xaab0,0x1d55,
     0x55c0,0x3ff,0xfe00,0,0};
     saucer.fd_addr=(char *) ufohigh; /* raster memory */
     saucer.fd_w=29; /* width in pixels */
```

```
    saucer.fd_h=11; /* height in rows */
    saucer.fd_wdwidth=2; /* width in words */
    saucer.fd_stand=1; /* standard FDB? */
    saucer.fd_nplanes=1; /* one plane */
}
cannon_high( )
{
    static int cannonhigh[ ]=
    {16,0,16,0,16,0,0x38,0,0x54,0,0x306c,0x19c0,0x68aa,0x2df0,
    0x68aa,0x2c08,0xc4ba,0x46b0,0xd3ab,0x97c0,0xc8ba,0x2600,
    0xd6aa,0xd600,0xd6aa,0xd600,0xc8ba,0x2600,0xd3ab,0x9600,
    0xc4ba,0x4600,0x68aa,0x2c00,0x6828,0x2c00,0x307c,0x1800,
        0x38,0};
    cannon.fd_addr=(char *) cannonhigh; /* raster memory */
    cannon.fd_w=23; /* width in pixels */
    cannon.fd_h=20; /* height in rows */
    cannon.fd_wdwidth=2; /* width in words */
    cannon.fd_stand=1; /* standard FDB? */
    cannon.fd_nplanes=1; /* one plane */
}
```

In medium res, two words are needed for every 16 pixels. If the second word is placed beneath the first word, then each pixel will take its color from the palette register pointed to by the top and bottom bit of each of the 16 columns in the stacked words. The left pixels of the two-pixel pair required to identify a color from 0 to 3 (00, 01, 10, 11) come from the first word, and the right pixels from the second word.

```
/* initializes data for medium-res saucer shape */
ufo_med( )
{
    static int ufomed[ ]=
    {
        0,0,0xf800,0,0,0x701,
        15,8,0xff00,0x100,0,0x800,
        0x1ff,0x1ff,0xfff8,0xfff8,0x28,0x701,
        0xffff,0x5555,0xffff,0x5555,0xf1c0,0x5800,
        0x3fff,0x3fff,0xffff,0xffff,0x8000,0x8000,
        0,0xff,0,0xffe0,0x8b0,0
    };
    saucer.fd_addr=(char *) ufomed; /* raster memory */
    saucer.fd_w=36; /* width in pixels */
    saucer.fd_h=6; /* height in rows */
    saucer.fd_wdwidth=3; /* width in words */
    saucer.fd_stand=0; /* not a standard FDB */
    saucer.fd_nplanes=2; /* two planes */
}
cannon_med( )
{
    static int cannonmed[ ]=
    {     /* plane zero */
        0x40,0, 0x40,0, 0xa0,0, 0x1f0,0, 0x21f0,0x8000,
```

```
        0x51f1,0x4000, 0xd7fd,0x6000, 0xdfff,0x6000,
        0xd9f3,0x6000, 0x50a1,0x4000, 0x2000,0x8000,
        /* plane one */
        0,0, 0,0, 0xe0,0, 0x1b0,0, 0x21f0,0x8000,
        0x7111,0xc000, 0xf7fd,0xe000, 0xff1f,0xe000,
        0xf9f3,0xe000, 0x70e1,0xc000, 0x2000,0x8000
    };
    cannon.fd_addr=(char *) cannonmed; /* raster memory */
    cannon.fd_w=19; /* width in pixels */
    cannon.fd_h=11; /* height in rows */
    cannon.fd_wdwidth=2; /* width in words */
    cannon.fd_stand=1; /* standard FDB? */
    cannon.fd_nplanes=2; /* two planes */
    vr_trnfm(work_handle,&cannon,&cannon);
}
```

In lo res, it takes four words to define 16 pixels. Again, the words are stacked out, and each column is read top to bottom. The leftmost bit of the color number comes from the first word, and so on.

```
/* initializes shapes for low resolution */
ufo_low( )
{
    static int ufolow[ ]=
    {0,7,0,7,0,0x8000,0,0x8000,
     1,0x1e,0,0x1e,0x8000,0x6000,0,0x6000,
     0,0x3f,0,0x3f,0x4000,0xb000,0,0xb000,
     0,0x7f,0,0x7f,0,0xf800,0,0xf800,
     0x1ff,0,0,0,0xfc00,0,0,0,
     0,1e49,0x1b6,0x1b6,0,0x27c0,0xd800,0xd800,
     0,0x7fff,0x7fff,0x7fff,0,0xfff0,0xfff0,0xfff0,
     0x8000,0,0xffff,0,8,0,0xfff8,0,
     0x6aaa,0x1555,0x7fff,0,0xaab0,0x5540,0xfff0,0,
     0x1d55,0x1d55,0x1d55,0x1d55,0x55c0,0x55c0,0x55c0,0x55c0,
     0x3ff,0x3ff,0,0,0xfe00,0xfe00,0,0};
    saucer.fd_addr=(char *) ufolow; /* raster memory */
    saucer.fd_w=29; /* width in pixels */
    saucer.fd_h=11; /* height in rows */
    saucer.fd_wdwidth=2; /* width in words */
    saucer.fd_stand=0; /* not a standard FDB */
    saucer.fd_nplanes=4; /* four planes */
}
cannon_low( )
{
    static int cannonlow[ ]=
    {0,16,0,0,0,0,0,0,
     0,16,0,0,0,0,0,0,
     0,16,0,0,0,0,0,0,
     0,0x38,0,0,0,0,0,0,
     0x28,0x54,0,0,0,0,0,0,
     0,0x307c,0x3010,0x3000,0x1c0,0x1800,0x1800,0x1800,
     0x1000,0x78fe,0x7854,0x7800,0x11f0,0x3c00,0x3c00,0x3c00,
```

```
    0x1000,0x78fe,0x7854,0x7800,0x1008,0x3c00,0x3c00,0x3c00,
    0x3800,0xfcfe,0xfc44,0xfc00,0x38b0,0x7e00,0x7e00,0x7e00,
    0x2c00,0xefff,0xff55,0xef01,0x69c0,0xee00,0xfe00,0xee00,
    0x3701,0xf7ff,0xff45,0xf701,0xd800,0xde00,0xfe00,0xde00,
    0x2901,0xe9ff,0xff55,0xe901,0x2800,0x2e00,0xfe00,0x2e00,
    0x2901,0xe9ff,0xff55,0xe901,0x2800,0x2e00,0xfe00,0x2e00,
    0x3701,0xf7ff,0xff45,0xf701,0xd800,0xde00,0xfe00,0xde00,
    0x2c00,0xefff,0xff55,0xef01,0x6800,0xee00,0xfe00,0xee00,
    0x3800,0xfcfe,0xfc44,0xfc00,0x3800,0x7e00,0x7e00,0x7e00,
    0x1000,0x78fe,0x7854,0x7800,0x1000,0x3c00,0x3c00,0x3c00,
    0x1000,0x787c,0x7854,0x7800,0x1000,0x3c00,0x3c00,0x3c00,
    0,0x307c,0x3000,0x3000,0,0x1800,0x1800,0x1800,
    0x38,0,0,0,0,0,0,0};
  cannon.fd_addr=(char *) cannonlow; /* raster memory */
  cannon.fd_w=23; /* width in pixels */
  cannon.fd_h=20; /* height in rows */
  cannon.fd_wdwidth=2; /* width in words */
  cannon.fd_stand=0; /* not standard FDB */
  cannon.fd_nplanes=4; /* four planes */
}
```

Here is the primitive for drawing shapes with MFDBs. It fills in the *pxyarray* from the structure, passed through *shape*, a pointer to an MFDB structure. We are copying from (0,0) in the source bitmap (the arrays defined above) to the *x* and *y* position passed through *(xpos, ypos). Copy raster, opaque* is used to do the actual work. This routine is not blindingly fast. If only this routine were rewritten in machine language, the whole game could be speeded up considerably. There is some elegance to a game written entirely in C, however.

```
put(shape,xpos,ypos,mode)
struct my_fdb *shape;
int xpos,ypos,mode;
{
    pxyarray[0]=0; pxyarray[1]=0;
    pxyarray[2]=shape->fd_w-1; pxyarray[3]=shape->fd_h-1;
    pxyarray[4]=xpos; pxyarray[5]=ypos;
    pxyarray[6]=xpos+pxyarray[2];
    pxyarray[7]=ypos+pxyarray[3];
    vro_cpyfm(work_handle,mode,pxyarray,shape,&screen);
}
```

This routine combines all the elements needed to initialize a virtual workstation.

```
init_workstation( )
{
    int i, handle;
    work_handle=handle=graf_handle(&cw,&ch,&dummy,&dummy);
    for (i=0;i<10;work_in[i++]=1); work_in[10]=2;
    v_opnvwk(work_in,&work_handle,work_out);
    if (!work_handle) exit(-1); /* error if we can't open */
}
```

Picture Puzzler™

Douglas N. Wheeler

Looking for an interesting diversion? This program scrambles a NEOchrome- *or* DEGAS*-format picture into a 10 × 10 jigsaw puzzle for you to reassemble on the screen. It also times how long it takes you to solve the puzzle and works in any screen resolution, color or monochrome.*

Nearly every Atari ST user accumulates at least one diskful of pictures created with *NEOchrome, DEGAS,* and other drawing programs. Numerous screens are available from user groups and bulletin board systems. With the popular slide-show programs that are also widely available, it's easy to view these pictures in rapid succession without actually loading them into *NEOchrome* or *DEGAS.*

Now there's something new you can do with your computer art collection. *Picture Puzzler*™ lets you turn any *NEOchrome*- or *DEGAS*-format picture into a fascinating jigsaw puzzle that you reassemble on the screen. It even keeps track of how long it takes you to put the puzzle back together.

Picture Puzzler supports the mouse and works in any screen resolution: low-resolution color, medium-resolution color, and high-resolution monochrome. And because it's written in compiled C, it responds to your commands very quickly.

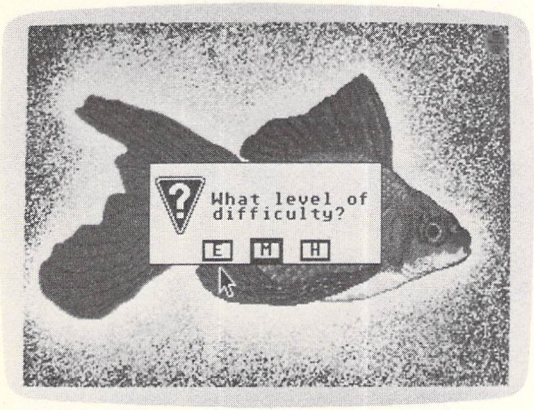
Scrambling a Screen

To get started, simply run PUZZLER.PRG from the disk menu or the GEM desktop in the screen resolution of your choice. After you've clicked on the OK button to acknowledge the copyright message, a standard GEM file selector appears. This works like any other file selector; click on the filename of the picture you want to load, and then click on OK. (As a shortcut, you can double-click on the filename.) If the picture you want to load is on another disk, insert that disk in drive A and click within the file window to display the new directory. You can also change the pathname at the top of the selector window to load pictures from other drives.

Picture Puzzler automatically recognizes a *DEGAS*- or

NEOchrome-format picture by its filename extension. *DEGAS* pictures should end in .PI1 for low resolution, .PI2 for medium resolution, and .PI3 for high resolution. *NEOchrome* pictures should always end in .NEO.

Figure 1-7. Easy: The Best Choice for First-Time Unscramblers



After *Picture Puzzler* has loaded the picture, it asks you to pick a difficulty level: easy, medium, or hard (Figure 1-7). We recommend starting with easy. On this level, the program divides the picture into a 10 × 10 grid and randomly scrambles the resulting 100 pieces. The medium and hard levels scramble the picture into 100 pieces, too, but they also add a twist—literally. On the medium level, about 25 percent of the pieces are flipped upside-down. On the hard level, about 50 percent of the pieces are flipped. When you pick your level, *Picture Puzzler* rapidly scrambles the picture before your eyes. The result can be seen in Figure 1-8.

The Hard Part

So much for the easy stuff. Now it's time to reassemble the picture. To do this, you drag pieces around the screen with the mouse just as you drag files around on the GEM desktop. To pick up a piece, point to it; then click and hold the left mouse button. Drag the piece where you want it; then release the mouse button. Instantly, it's swapped with the piece that formerly occupied that spot.

Figure 1-8. The Newly Created Puzzle

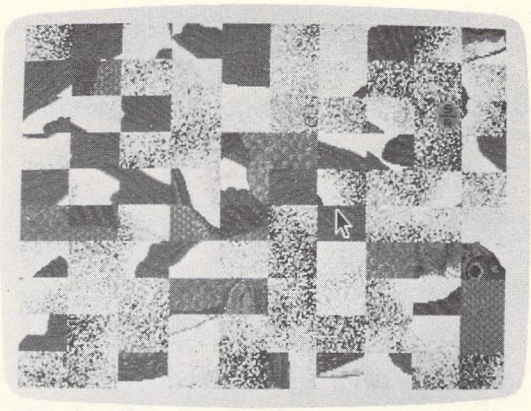
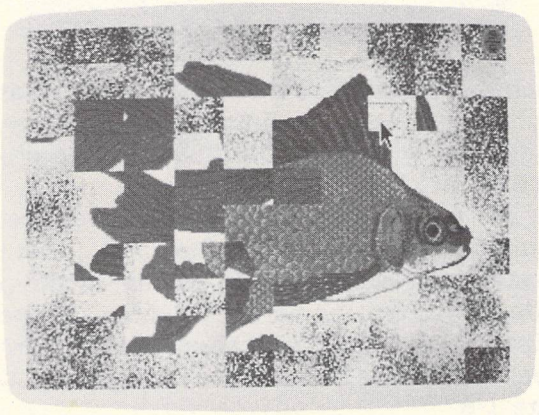


Figure 1-9. Outline Box Showing New Location of Puzzle Piece



If you have chosen the medium or hard level, you may need to flip some pieces over. To do this, hold down either the Shift, Control, or Alternate key when you press the mouse button to drag a piece; when you release the button, the piece is dropped into place and flipped.

Reassembling a picture is not as easy as it looks. To give you some help, *Picture Puzzler* lets you look at the unscrambled picture when you press and hold the *right* mouse button. You can look at the picture as long as you hold down the button. But be careful: You get only three such peeks during each puzzle.

If you get frustrated and want to give up, you can return to the desktop by pressing both mouse buttons simultaneously. A dialog box asks you to confirm this action.

If you persevere and complete the puzzle successfully, *Picture Puzzler* instantly lets you know and displays a dialog box showing how long it took you to finish. At this point, you can either try another picture or exit the program.

Helpful Hints

Some of the same techniques that apply to assembling real jigsaw puzzles also work with *Picture Puzzler.* For example, if the screen has a border, that is always a good place to start when assembling a complex picture. If the picture contains any text, that is another good place to begin.

You may be wondering what happens when a picture contains large areas of solid color or repeating patterns. It would seem to be nearly impossible to reassemble such a picture, because many of the pieces are visually identical. However, *Picture Puzzler* takes this problem into consideration. If two or more pieces really are identical, their positions are interchangeable. But if even one pixel is different, *Picture Puzzler* treats them as separate pieces that must be placed in their original locations.

Despite this feature, occasionally you may assemble a picture which looks correct, but in fact is not. The problem is that more than one palette color may be assigned the same red, green, and blue values, making them indistinguishable on the screen. *Picture Puzzler* knows the difference and won't let you finish until you get it right. This problem can be seen in the picture entitled "Mr. *X*" that comes on the *DEGAS* disk. There's a border on the lower right side of the screen, though you can't see it.

Fortunately, there is a solution. When you drag a piece over one of these areas, the border of the dragged box changes colors. At any rate, keep this problem in mind if you're creating your own pictures for puzzles.

Another problem is encountered when you try to piece together sections of a picture that were spray-painted. For instance, the picture of the comet on the *DEGAS* disk is just about impossible to complete because there is almost no pattern to the stars. Give it a try—but be sure not to waste your three peeks at the correct picture.

Spanish Castles

Robert S. Geiger

"Spanish Castles" is a deceptively simple game of strategy. While it runs in any resolution, the program looks best in lo res, where the first six palette colors are available to brighten the game board which fills the screen.

"Spanish Castles"—like the original, "The Witching Hour"—is a strategy game based on Alquerque, a popular Spanish board game that was first played in ancient Egypt. This ST BASIC version pits you against a friend or the computer, or you can watch the computer play against itself.

The game board for Spanish Castles has 25 squares (which actually look more like octagons, with their rounded corners); each square is marked with a letter of the alphabet. As play begins, 12 Castilian towers confront an equal number of Moorish mosques, so only one square is left vacant. Each player attempts to capture the opponent's pieces by jumping over them. Vertical or horizontal moves are permitted, as are some diagonal ones. Follow the lines connecting the squares to trace the legal diagonal moves. Don't worry if you accidentally charge off in the wrong direction, though—the computer won't allow an illegal move.

Spanish Castles creates a command line at the top of the screen where ST BASIC's menu bar is usually found. When "Moor's move from:" appears on the command line, the player who has chosen to be the Moor enters a letter. The letter tells which square contains the mosque that the Moor is about to move. When the computer responds with the "To:" prompt, the Moor enters the letter of the square the mosque is heading for.

If you're playing against the computer, simply press RETURN when you've finished your move. Press RETURN on every move to watch the computer play against itself.

If you can capture an opponent's playing piece, you must do so; otherwise, you may move to any adjacent empty square. If you have a choice between a single jump and a double or triple, be careful; the jump which appears to carry you further may get you into more trouble.

Capturing a playing piece removes it from the board. The game is over when one player has captured all the pieces from the opposing side. At that point, an alert box pops up proclaiming the winner, and the game can be restarted or exited via the mouse.

Running Spanish Castles

Since there are only two BASIC programs in this book, the main menu program was not designed to load and run BASIC programs, so you'll need special instructions to load and run Spanish Castles. To do so, first insert *ST Language Disk*, which came with your computer. Next, load and run BASIC.PRG by double-clicking on its icon or filename from the desktop. Once BASIC is loaded, insert *COMPUTE!'s Second Book of Atari ST Disk*. Type **LOAD A:\SPANISHC.BAS** and press Return; then type **RUN** and press RETURN. This should bring up Spanish Castles, ready for you to play.

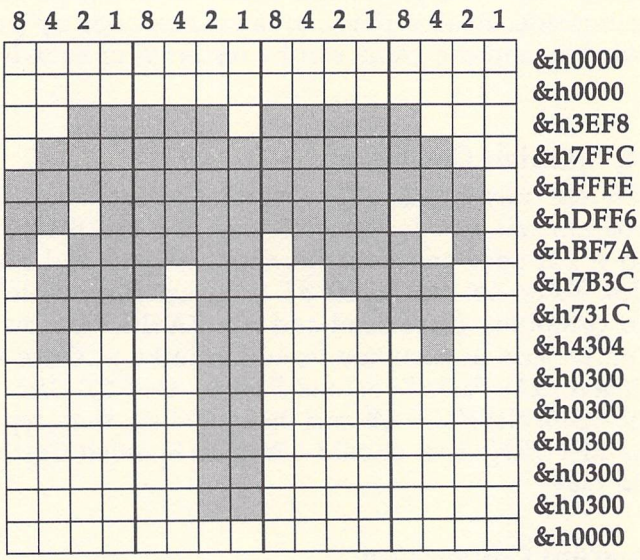
The Program Conversion

The Witching Hour underwent several major changes before its conversion to ST BASIC was complete. If you've done—or would like to do—program translations, you'll find it interesting (and perhaps helpful) to see exactly how these changes worked.

In the original IBM program, the game squares are graphic blocks (31 pixels across by 23 pixels down) which come in three types—an outlined empty square, an outlined square with a witch shape, and an outlined square with a ghost shape. The programmer uses IBM BASIC's GET and PUT commands to place them on the game board. It's possible to duplicate this procedure on the ST using GEM VDI raster operations, but Spanish Castles employs OpCode(112), Set User-Defined Fill Pattern, which is a simpler technique for creating smaller blocks under certain conditions.

Unless you write a utility program to create your individual fill patterns, you'll have to use paper and pencil to make a fill description chart such as the one shown in Figure 1-10. There should be 16 rows × 16 columns—256 tiny squares—in the chart; the 16 columns are divided into four equal sections. Each tiny square represents a single pixel in your fill pattern and a single bit in the computer's memory image. Each bit can be either on or off, so an individual row of 16 bits would look like a confusing string of 1's and 0's.

Figure 1-10. Sample Fill Description Chart—A Palm Tree



Using the hexadecimal numbering system (base 16), however, translates your patterns into something easier to understand. Begin by converting the binary pattern of 1's and 0's for each row into 16 decimal numbers and then into 4 hexadecimal numbers. To do this, label the pixels (in each of the four sections of the chart) 8, 4, 2, and 1, going from left to right. Once you've drawn the pattern for your fill (by shading in the squares which represent the *on* pixels), add up the numbers which correspond to the shaded bits in each section. For example, if all four squares in a section are shaded, the sum is 15 (8 + 4 + 2 + 1 = 15). The sum for each section is then converted to hexadecimal. Do this for each row, and you will have a total of 16 four-digit hexadecimal numbers.

To make the conversion easier, use this table:

decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Notice that in hexadecimal, letters are used to represent the six higher-order digits. The four hex digits for each row are grouped together and, in ST BASIC, are preceded by "&h" to signify a hexadecimal number. See lines 880–900 of the Spanish Castles program for an example fill pattern.

The Big Picture

Now imagine that you are storing a full screen of each pattern somewhere in memory. The data which describes the pattern of the fill consists of 16 two-byte words, with bit 15 of word 1 as the upper left dot of the pattern, and bit 0 of word 16, the lower right dot. You create the pattern's screen by starting at the upper left-hand corner and reproducing the 16 × 16–pixel pattern block across the top of the screen, and then copying row after row until the pattern covers the screen. Then, when you use a BASIC command like PCIRCLE to draw a filled circle, it's as if you have opened a hole in your BASIC OUTPUT window onto the area of memory where your fill pattern's screen is located. The trick is to open a 16 × 16–pixel hole using VDI OpCode(9), Filled Area, but only at locations at which both the x and y values of the top left-hand corner are multiples of 16. This reveals exact copies of the original patterns—either castle towers, mosques, or a solid filled area.

Spanish Castles includes several initial variables which differ from those in the IBM program; these variables insure that the coordinates calculated and stored in the X and Y arrays (which hold the positions where each square is to be PUT) are multiples of 16. The ST program uses offsets with these coordinates—VDI OpCode(11) Id(9) to place the rounded-corner squares, and VDI OpCode(8) Text to place the identifying letters.

In addition, the translation program demanded minor changes to line up all the connecting lines. With all the changes it involved, the completed game board filled the OUTPUT window, leaving no room for the INPUT prompts. This made it necessary to avoid any ST BASIC commands such as LINEF, CIRCLE, ARC, PRINT, and GOTOXY, which output to the screen. Instead of the OUTPUT window, Spanish Castles uses the VDI directly for all output to the screen, which reduces ST BASIC to a set of generic commands. You need a good reference to do any work with VDI, but it's worth the effort.

That ST Look

Finishing touches included changing the first six palette colors and erasing the entire screen to create a sharp display. With AES OpCode(105), which also draws the window's title line instead of the full window, the title of the OUTPUT window became "Spanish Castles." To allow the game board to command the display, the program has the command line at the top of the

screen. A final system alert box gives this translation a typical ST appearance.

Note that the mouse pointer is turned off during the program since it's not needed in a program which relies on keyboard input (and it gets in the way when the game board is drawn). Without a mouse pointer you will not be able to pull down ST BASIC menus, which are still there even though the menu bar showing the titles was erased at the start of the program. As a result, you must exit this program by clicking either mouse button while the mouse pointer is over the right-hand FINISHED bar within the final system alert box.

ST-GO

Kyle Cordes

Here's a game that's easy to learn and fun to play, but don't be fooled: Winning requires concentration and foresight. "ST-GO" runs in medium resolution on any ST with a color monitor.

"ST-GO" is a program which allows two players to compete in the Japanese game of go-moku (sometimes called goban). Although this computer version of go-moku is played on the same kind of "board"—a 19 × 19 grid—used by the game of go, the two should not to be confused. Go is a more complex game; the object of go-moku is simply to place five pieces in a line in any direction. Still, if you and your opponent stay sharp in detecting each other's strategy, ST-GO is far from easy.

On Your Mark...

In order to begin ST-GO, you and your opponent must decide which of you will be the red player and which will be blue. At the bottom corners of the grid are turn indicators—a red and a blue rectangle. In a new game, the red rectangle is always larger, indicating that the red player moves first.

To place a piece, point the mouse arrow near an intersection on the grid and click. If that intersection is open, your playing piece appears there, and your turn ends. If you accidentally move the mouse while clicking (and the piece appears on the wrong intersection), you can undo the incorrect move by clicking on the UNDO box on the left side of the screen (the UNDO key doesn't work). The turn indicators help you make sure that you don't click twice on UNDO and accidentally erase your opponent's last move.
corner of the grid. Every turn involves a move: Neither player can pass.

You may display the instructions for ST-GO at any time by clicking on the INSTRUCTIONS box. You may also restart a game whenever you choose by clicking in the NEW GAME box. (When you click on NEW GAME, RESET, or QUIT, a dialog box appears to verify your choice.) Note that the win counters are

not affected by NEW GAME; if you wish to reset them, click in the RESET box.

When a player successfully places five pieces in a row, the screen flashes and the win counter below his or her turn indicator is incremented. At that point, the players can choose to continue playing on that board or to start over with an empty board.

Continuing on the same board makes the game more difficult, since there are more potential moves to consider. If you do choose to continue, the player who has just lost moves next. If, on the other hand, you wish to compete by seeing who has the lowest average of moves-per-win during a series of games, you must start a new game in order to reset the counter which keeps track of the number of moves.

CHAPTER TWO

Applications

ST-Graph

Michael P. Cohan

Now you can quickly and easily generate graphs to display all kinds of data for home or business. Vertical and horizontal bar graphs, pie charts, line graphs, scatter-dot charts, and numerous variations can be compiled with a few mouse clicks. The program works on all STs in either medium-resolution color mode or high-resolution monochrome.

It's said that a picture is worth a thousand words, and for good reason: Sometimes a powerful photo can convey more information about a dramatic moment than a pageful of prose.

Likewise, a good graph can sometimes reveal more information than a pageful of numbers. Numerical relationships that are lost in columns of figures often pop into sharp focus when displayed in chart form. But until computers came along, constructing graphs was a tedious process that didn't lend itself to instant manipulation and experimentation.

Now you can quickly and easily display graphs on your Atari ST with just a few mouse clicks. "ST-Graph" is an easy-to-use application program that rapidly generates all the common types of graphs based on values you supply. It supports all features of GEM (the Graphics Environment Manager), including drop-down menus, dialog boxes, mouse controls, and adjustable screen windows. It runs in either the medium-resolution color mode or the high-resolution monochrome mode. (It looks best in monochrome, because it was designed for that mode and takes advantage of the greater resolution.) Written in compiled Pascal, ST-Graph is both fast and efficient.

Getting Started

You'll find ST-Graph on the disk as STGRAPH.PRG. An important related file is STGRAPH.RSC. Commonly known as a *resource file,* STGRAPH.RSC contains data required for STGRAPH.PRG to function. Therefore, if you copy ST-Graph to other disks, make sure to copy both STGRAPH.PRG and STGRAPH.RSC to get a fully working program.

You can run ST-Graph from the disk either by selecting it with the menu program or by double-clicking on its icon/filename on the GEM desktop. Note that if you attempt to run ST-Graph in the low-resolution color mode, an alert box informs you that the program does not support this mode, then returns you to the GEM desktop.

When ST-Graph runs, you should see three windows labeled *Bar Graph, Data,* and *Information.* Any of these windows may be dragged to any part of the screen. The Graph window may be resized or expanded to a full screen. The Data window also has what appears to be a Full-Screen button, but this has a different effect that will be explained in a moment.

The Graph window, as its name implies, displays your graph. The name of this window changes to reflect the different types of graphs which can be displayed. If no data is available, the Graph window is empty (as seen when the program first runs).

The Data window displays the data you have entered. Each graph may contain up to 12 items of data. Each item has three parts: a value (an integer from 0 through 999999), a label (up to six characters), and an Enabled/Disabled button (which controls whether this item is displayed on the graph). All three parts are shown in the Data window. If an item is enabled, it is accompanied by a square filled with the pattern that keys it to the matching item on the graph. If the item is not enabled, the space next to the value and label is blank.

If you don't wish to see items which are not enabled, you can click on the Data window's Full-Screen button. The window resizes itself and shows only those items which are enabled. If no items are enabled, the window will be empty. To see all the items again, click once more on the Full-Screen button.

The Information window consists of eight lines of text that you type into a dialog box. This lets you add miscellaneous information (such as the name of the chart) before printing out the screen.

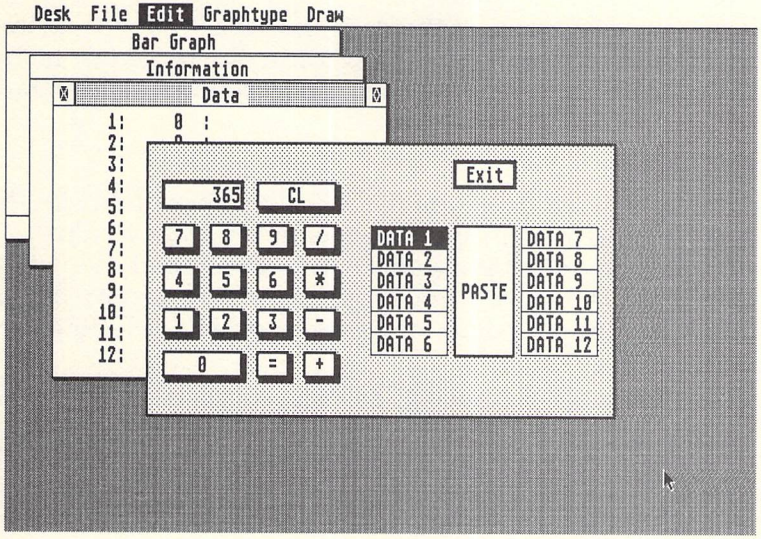
Note: When editing labels (or any dialog box) in ST-Graph, be careful not to move the cursor outside the dialog box, or the program will lock up.

Creating a Graph

Now you can begin constructing a graph. The first step is to enter your data with the pop-up calculator.

Drop down the Edit menu and select the Calculator option. Instantly, a four-function calculator pops open on the screen

Figure 2-1. The Pop-Up Calculator in "ST-Graph"



(Figure 2-1). To enter numbers, point to a button and click the mouse. ST-Graph does not support negative numbers or fractions, but if you want to graph values such as 2.3 and 5.8, all you have to do is mentally shift the decimal point to make them whole numbers. For example, enter 2.3 and 5.8 as 23 and 58, respectively; the proper numerical relationship is reflected on the graph. (Of course, to enter whole numbers such as 23 and 58 on the same graph, you'd have to shift *their* decimal points, too, entering them as 230 and 580.)

On the right side of the calculator are two columns of buttons labeled DATA 1 through DATA 12, and a large button labeled PASTE. These buttons let you copy the number on the calculator's display into the Data window of your chart. Simply click on one of the DATA buttons to indicate the data item (1–12) and then click on the PASTE button. Note that this automatically enables the item so that it appears on the graph. To close the calculator, click on the Exit button.

When you exit the calculator, a graph appears in the Graph window, corresponding to the data you entered. If it isn't exactly the graph you want, don't worry; it can be instantly changed by selecting a menu option, as you'll see shortly. You can also resize the Graph window, making it as small or as large as you want. The graph automatically rescales itself to fit the new window. (Sometimes the graph does not rescale when you make the

window smaller; if this happens, just select Plot Graph from the Draw menu to force a redraw.)

If you need to erase all the data and labels, as well as the contents of the Information window, drop down the File menu and select Clear. An alert box asks you to confirm this operation.

Editing Labels

After you've entered data, you'll probably want to label it. These labels will appear next to the numbers in the Data window. Also, occasionally you may want to disable some of the data you've entered to keep it from appearing on the graph. You can do all this by dropping down the Edit menu and selecting the option called Edit Labels.

This option opens a dialog box with 12 editable text fields. The text fields are for the labels. Point to the field you want to edit and click the mouse. When a thin cursor appears, you can start typing.

You can move within this field with the left and right cursor keys, and you can erase mistakes with the backspace key. To change fields, point to the desired field and click the mouse again.

Each text field has an Enabled button. These buttons control whether each data item is displayed on your graph. If the button is selected (displayed in reverse video—with white letters on a black button), the item is enabled. Click on the buttons to change their state.

The 12 numbered buttons in the row across the top of the dialog box control which of the 12 fields are used by ST-Graph when you exit the dialog box. Normally, when the program starts, all these buttons are selected (displayed in reverse video). This means ST-Graph uses both the labels and Enabled buttons for all 12 fields when you exit. If you click on a numbered button to *deselect* it (it appears in normal video—with black numbers on a white button), ST-Graph ignores the corresponding label and Enabled button. The label and button state is retained the next time you pick the Edit Labels option.

This might seem confusing at first, so here's why the numbered buttons are provided. The label fields always contain whatever you last typed into them, even when you load a previously saved graph with new data and labels. The new labels are shown in the Data window, but not in the label fields. Thus, if you load in a new graph and labels, then want to change only one label, you should select only the button of the label you

want to change so that the others aren't reset also. (If this still sounds confusing, it will become clearer as you work with the program.)

When you've finished editing labels, exit the dialog box by clicking on the Exit button. If you change your mind and decide you'd rather leave the labels and Enabled buttons as they were before you opened the dialog box, click on the Cancel button instead.

Dollars and Cents

Besides editing the labels, there's one other way to modify the display in the Data window. When you drop down the Edit menu and choose the option called Show Data as Dollars, all the numbers you've entered are divided by 100 and are displayed to two decimal places (81 becomes .81, and so on). Therefore, if you want your graph to reflect dollars-and-cents figures, enter an amount such as $23.47 as 2347 and select this option.

To turn off the option, drop down the menu and select it again. A checkmark indicates when the option is active.

Note that Show Data as Dollars affects only the display of the Data window—no decimal point is available on the calculator.

Changing Graph Types

Once you've entered your data and labels, you can specify which type of graph should be displayed in the Graph window. Simply drop down the Graphtype menu and select the type you want. The graph is instantly redrawn and the name of the Graph window changes to reflect your choice. Also, a checkmark on the Graphtype menu shows which graph is currently selected.

Note that you can change any graph option at any time, even when the Graph window is not the active window.

ST-Graph offers the following types of graphs:

- Pie Chart: Your data is translated into slices of the pie in a clockwise direction (Figure 2-2).
- Horizontal Bar: Your data matches the bars from top to bottom (Figure 2-3).
- Vertical Bar: Your data matches the bars from left to right (Figure 2-4).
- Stacked Bar: A variation of the vertical bar; see below.
- Line: Your data matches the points on the line from left to right (Figure 2-5).

- Dot: Your data matches the dots from left to right (similar to Figure 2-5).

Figure 2-2. Pie Chart Created with "ST-Graph"

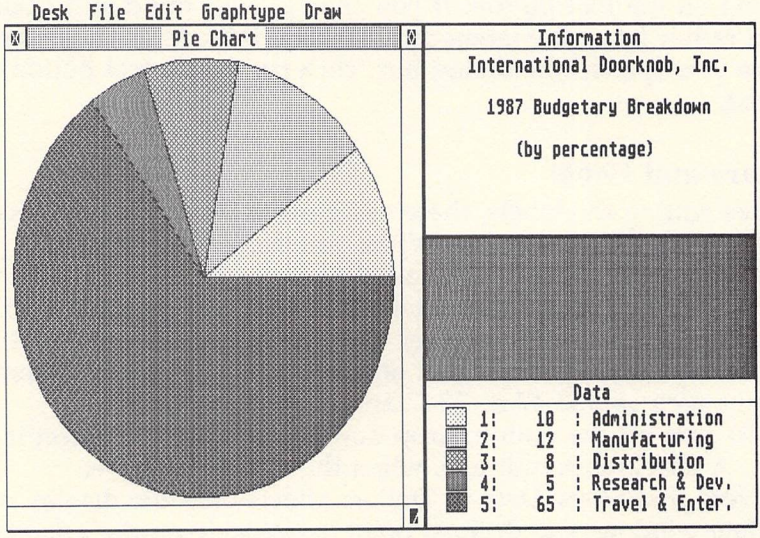


Figure 2-3. Horizontal Bar Chart

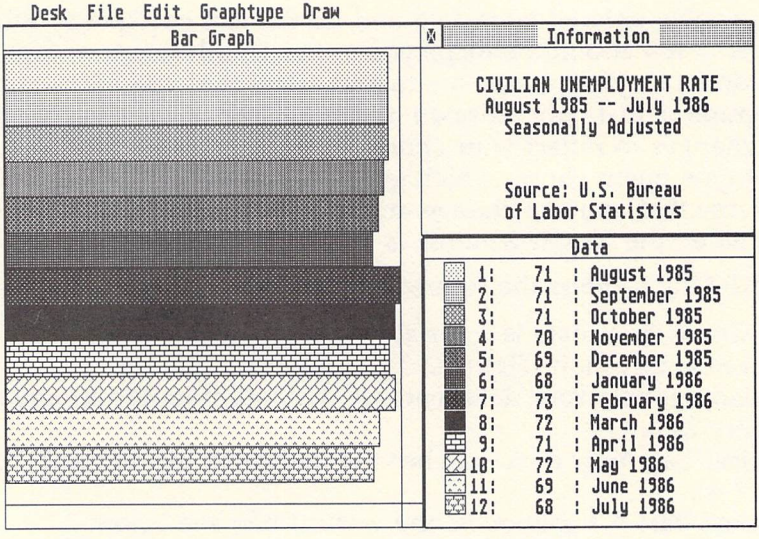


Figure 2-4. Vertical Bar Chart

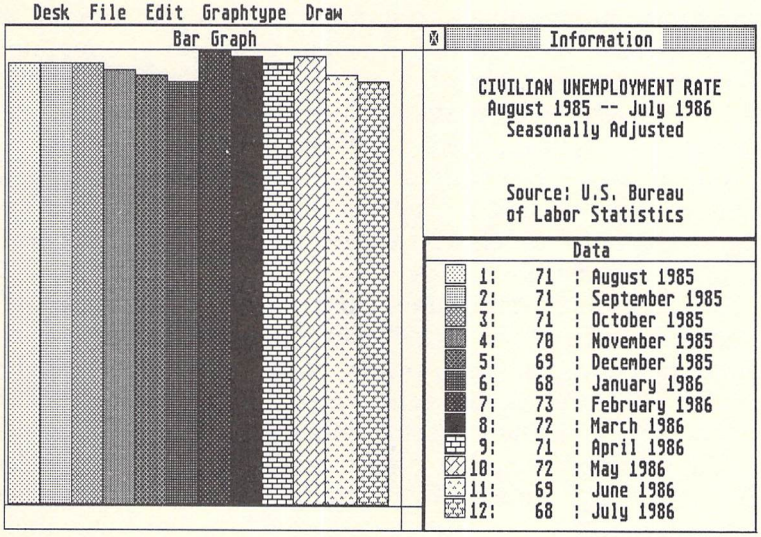
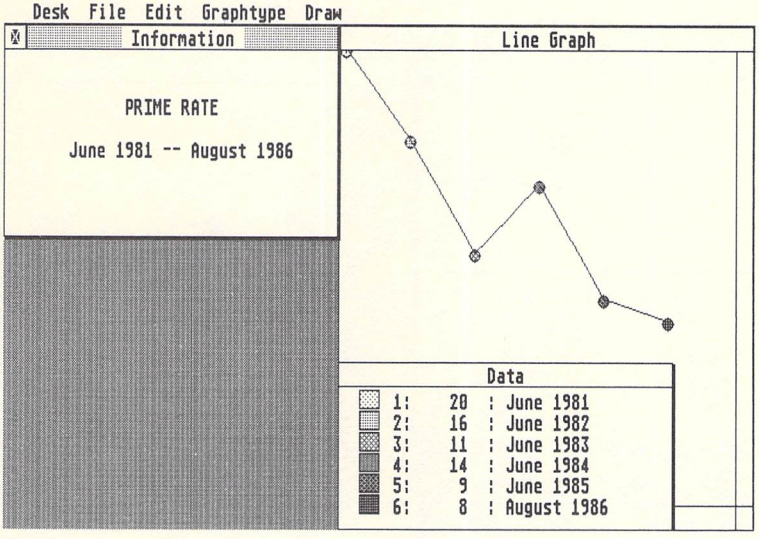


Figure 2-5. Line Graph

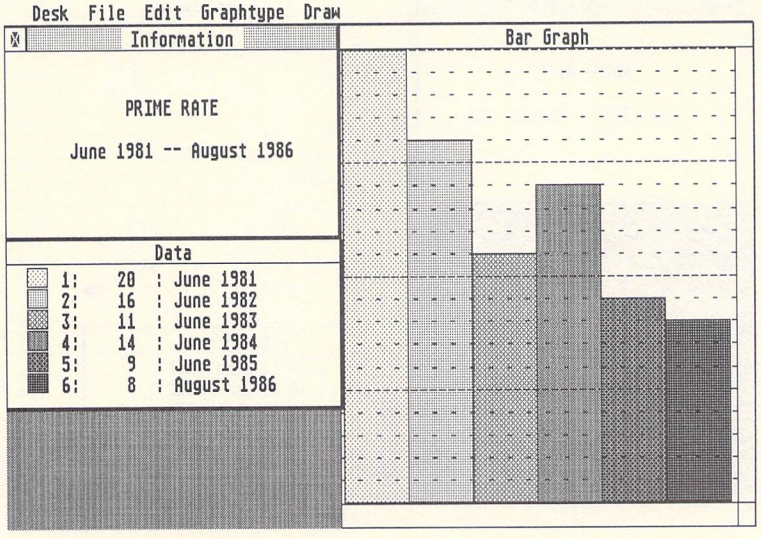


In addition to these different types of graphs, you have several options available for modifying the graphs. One is the Grid option, which superimposes the graph upon a grid. Toggle this option on and off by selecting it from the Graphtype menu; a checkmark indicates when it is active. (Note that Grid is dimmed on the menu when you're displaying a pie chart, since it's not meaningful to superimpose a pie chart on a grid.)

You can adjust the scale of the grid by dropping down the Edit menu and selecting the option called Edit Grid Spacing. A dialog box opens and lets you enter three numbers—for Scale, Grid, and Emphasize.

The Scale value determines the relative size of the chart (but not the actual size of the Graph window). For instance, if you're graphing percentages, you'll probably want a scale of 1 to 100. Normally, ST-Graph scales your graph to the largest data item supplied. That is, if the largest number you enter for a data item is 94, the corresponding bar on a bar graph will extend completely across the window. By changing the value of Scale, you can scale the graph to a different value. Note that this value is ignored if it's *less than* the largest value in the graph.

Figure 2-6. A Scaled and Emphasized Grid



The Grid value controls the spacing of the grid. For instance, if you've entered 100 for the Scale value and want grid lines spaced by increments of 10 (representing values of 10, 20, 30, 40, 50, 60, 70, 80, and 90), enter a Grid value of 10. If you want grid lines spaced by increments of 20 (20, 40, 60, 80), enter 20. If you want to set the Scale but don't want to display a grid, enter 0.

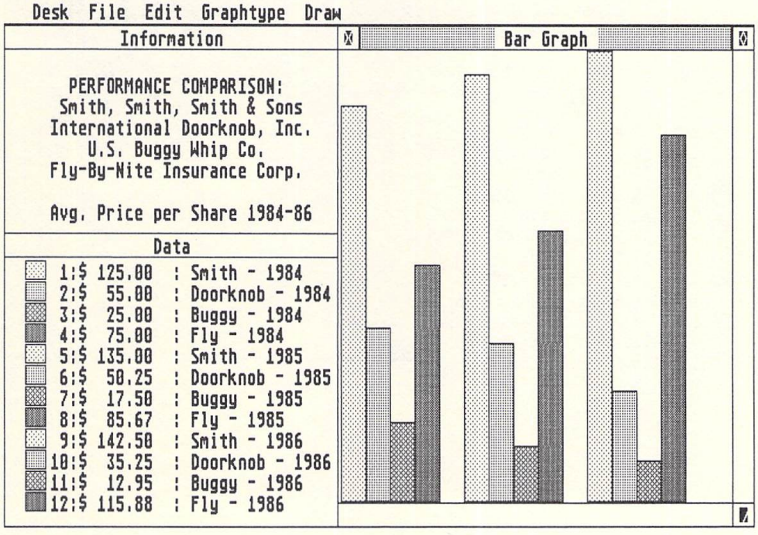
The Emphasize value lets you highlight some of the grid lines. If you enter 5, for instance, every fifth line becomes a solid line instead of a dotted line. If you don't want to emphasize any lines, enter 0.

So, putting this all together, let's say you want a vertical bar chart ranging from 1 to 20, superimposed on a grid spaced by 1's, with every fifth grid line emphasized. You'll enter 20 for Scale, 1 for Grid, and 5 for Emphasize. The result can be seen in Figure 2-6.

Grouping

Beneath the Grid option on the Graphtype menu, you'll see another set of options, labeled Bar Grouping. These options are: None, Groups of Six, Groups of Five, Groups of Four, Groups of Three, and Groups of Two. A checkmark indicates the currently selected option. These options are available only when bar graphs—horizontal, vertical, or stacked—are being displayed.

Figure 2-7. Bar Grouping

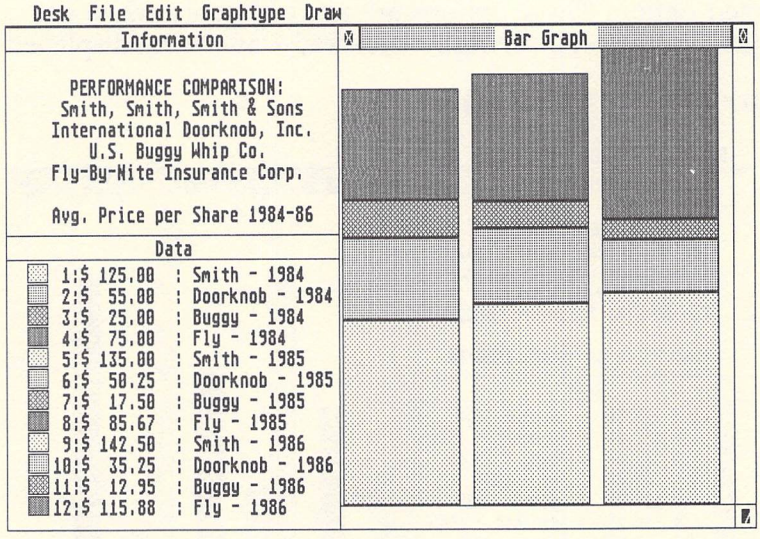


These options let you group the bars together and use the same fill patterns for each group. For instance, let's say you're comparing the performance of four companies over a three-year period. You enter the first-year figures for the companies as data items 1–4, the second-year figures as data items 5–8, and the third-year figures as items 9–12. Then you choose the Groups of Four option with either the horizontal or vertical bar graph. (A grid can be added, too, if you wish.)

The result can be seen in Figure 2-7; the bars are grouped together in three sections, four bars per section, with the same four fill patterns used in each section. The correct fill patterns are also displayed in the Data window.

The bar-grouping feature works with stacked bar charts as well. Continuing with our previous example, select the Stacked Bar option with Groups of Four. Instead of displaying three groups of four bars, ST-Graph displays only three bars. The first four data items are stacked atop one another in the first bar, the next four items are stacked in the second bar, and the last four are stacked in the third bar (Figure 2-8).

Figure 2-8. A Stacked Bar Chart



If you don't have much experience with business graphs, you'll probably have to experiment with this feature awhile to learn why it's useful. Try some different values and grouping options; then compare the graphs that result when you select the Vertical Bar option versus the Stacked Bar option.

Printing the Chart

ST-Graph's Information window lets you add a title and any other information you deem necessary before printing out a finished copy of the chart.

Up to eight lines of text can be displayed in this window. To enter the text, choose the Edit Information option from the Edit menu. A dialog box opens with eight editable text fields. Click on the line you want to edit; then enter the text you want. Click on the Exit button when you've finished, or click on Cancel to retain the previous contents of the Information window.

The row of eight buttons across the top of this dialog box have the same effect on the text fields as the similar buttons in the Edit Data box.

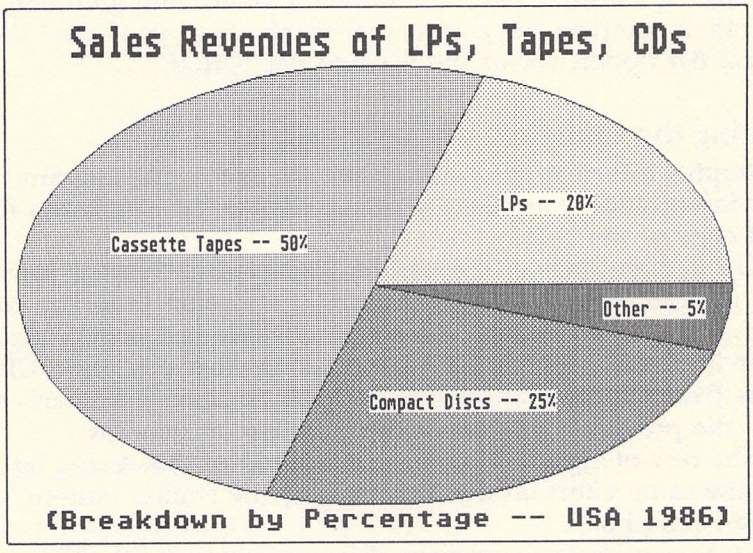
Once you've designed your chart and arranged the Graph, Data, and Information windows on the screen for an attractive display, you can press the Alternate and Help keys simultaneously to dump the screen to a graphics printer. (The ST is set up for an Epson or Epson-compatible, but it's possible to use other printers by making adjustments with the Install Printer desk accessory that comes with the ST.)

Alternatively, you can capture the ST-Graph screen on disk with one of the commercial or public domain snapshot-type programs. One such program, by Russ Wetmore, is available on the Atari Corporation's bulletin board system in Sunnyvale (408-745-5308), and on other BBSs as well. Another, "Snapshot *NEO/DEGAS*," appears later in this book. Most of these snapshot utilities save the screen in a format that can be loaded by graphic-design programs such as *NEOchrome* or *DEGAS*. This lets you modify the chart further before making a printout (Figure 2-9).

Saving and Loading Graphs

Another ST-Graph option lets you save a graph on disk. The actual screen layout won't be saved, however—only the contents of the Data and Information windows. You'll still have to arrange the screen as you want it. To use this option, select Save from the File menu.

Figure 2-9. An "ST-Graph" Chart Captured on Disk with a Snapshot Utility and Modified with a Drawing Program



To bring in a previously saved graph, select Load from the File menu. Both options take advantage of the standard GEM item selector, so you can save and load files on different disks and in different folders. When you select Load, ST-Graph looks for filenames with the extender .DAT, so you might want to append this extender to any graphs you save.

Remember that loading a previously saved graph does not change the text fields in the Edit Information/Edit Labels dialog boxes—only in the Information and Data windows themselves. If you wish to change only one line of text or one label, make sure all the buttons across the top of these dialog boxes are deselected, except for the one corresponding to the field you wish to change.

To leave ST-Graph and return to the GEM Desktop, you can click on the Close buttons of any of the three windows or choose Quit from the File menu. An alert box requests that you confirm the action.

Desktop Clock

David Plotkin

With this desk accessory, you can display a digital clock on your screen while running any other GEM application program. The program works on any ST in any screen mode: low- or medium-resolution color and high-resolution monochrome.

It's easy to lose track of time when you're working with a computer, and nearly everyone has experienced the surprise of discovering that it's suddenly three hours past bedtime. But now it's easy to keep an eye on the clock while working with your word processor, spreadsheet, database, or telecommunications program.

"Desktop Clock" is a simple desk accessory that's always instantly available within any program that supports GEM (the Graphics Environment Manager). When summoned from the Desk menu, it pops open a small window with a digital clock. The clock can be repositioned anywhere on the screen and does not interfere with the main application program running in the background. It operates on a 12-hour cycle and indicates a.m. or p.m. You can make the clock disappear and reappear at will. You can even make it reappear if it is hidden behind another window.

Installing the Clock

The program file for Desktop Clock can be found on the disk under the filename CLOCK.AC. *This is not an executable file*—it cannot be run from the disk or by clicking on the filename or icon from the GEM desktop. Instead, it must be installed on your boot disk as a *desk accessory*, a program that is automatically loaded into memory when you first switch on your ST. A desk accessory remains in memory even when it isn't running. To activate a desk accessory, you must select it from the Desk menu which is present in all application programs that support GEM. (Sometimes the Desk menu is titled with an Atari logo symbol; in any case, it's always the menu at the far left of the screen.)

"Desktop Clock" was written using *Personal Pascal* from Optimized Systems Software.

Installing Desktop Clock requires only a few simple steps:

1. Copy the file CLOCK.AC from *COMPUTE!'s Second Book of Atari ST Disk* to your boot disk—that is, the disk you insert in drive A when you first switch on the computer.
2. Rename CLOCK.AC to CLOCK.ACC by selecting Show Info from the File menu. (If you're not sure how to rename a file, consult the manual that came with the ST.) The .ACC extender is important—programs with filenames ending in .ACC are recognized by the computer as desk accessories and are automatically loaded into memory during boot-up. If you happen to be using the disk-based version of TOS (pre-ROM), you'll have to rename the file DESK*x*.ACC, where *x* is a number between 1 and 6 not being used by some other desk accessory (for example, DESK5.ACC). Make sure that you install no more than six desk accessory programs, or your ST will fail to boot. The Control Panel accessory counts as two (that file installs two accessories: the Control Panel and the printer configuration accessory). The VT-52 emulator accessory also counts as two accessories, placing two options on the Desk menu. This means that if you install the Control Panel and VT-52 emulator, there's only enough room for two more accessories of your own choosing.
3. To install Desktop Clock, turn off the computer and wait a few seconds. Then switch the computer back on, making sure the bootup disk with CLOCK.ACC is in drive A. (Note that desk accessories must always be installed with this *cold start* procedure—a *warm start* triggered by pressing the reset button may not reliably install a new accessory.)

Using the Clock

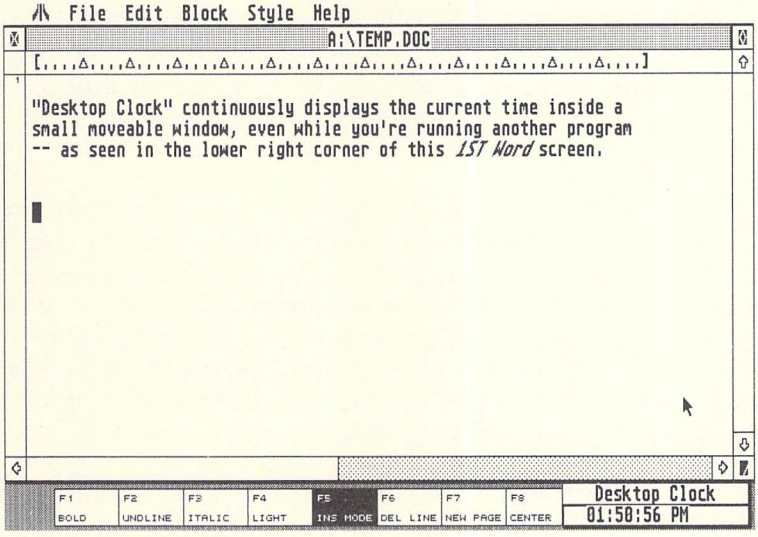
That's all there is to it. When you drop down the Desk menu from the GEM desktop, you should see a new selection labeled Desktop Clock, along with any other accessories you may have previously installed on the boot disk. To open the Desktop Clock, just select Clock from the menu.

Desktop Clock works the same as any other GEM window. You can move it anywhere on the screen by clicking and dragging the title bar. You can make it disappear by clicking on the close gadget in the upper left corner of the window. You can move the clock on top of other windows, or move other windows on top of the clock. However, there are no sizing gadgets on the clock window; it always stays the same size so that it

takes up a minimum amount of space on your screen.

If you close the clock window or hide it behind another window, you can make it reappear by selecting it again from the Desk menu.

Figure 2-10. "Desktop Clock" (lower right corner) in a Screen from the GEM Application *1ST Word*



Desktop Clock has no provisions for setting the system time. Therefore, to set the clock, you'll need to install the Control Panel desk accessory that came with the ST or use a program that prompts you to enter the time and date when booting up. To learn how to use a batch file for this purpose, see the "ST-Shell" article in Chapter 4.

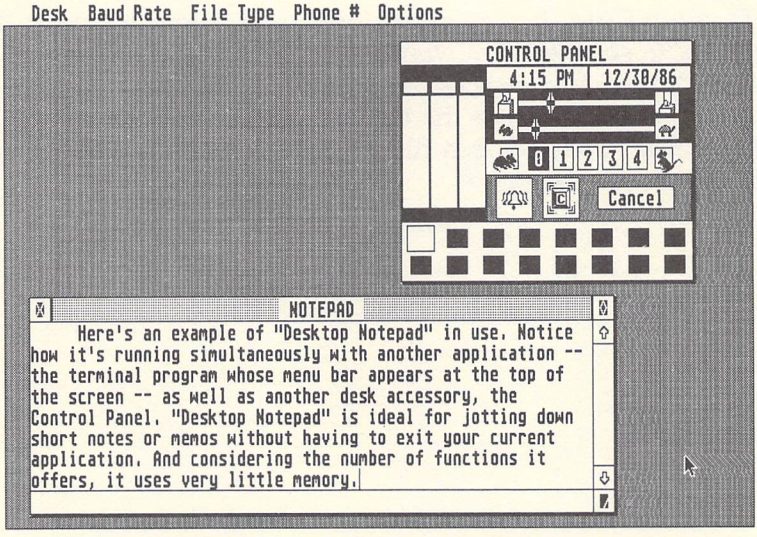
Desktop Notepad

Tim Victor

If you've ever needed to jot down a short note while using your ST, you'll love this desk accessory—a notepad that's instantly available while any other program is running. You can write a note, save or load from disk, and print copies without interrupting the main program you're using. It works on any ST in any screen mode, color or monochrome.

"Desktop Notepad" is a handy desk accessory that nearly every ST user needs at one time or another. Often while working with an application program, you'll need to write yourself a quick note, memo, or reminder. Some word processors let you open another window for this purpose, but what if you're using a spreadsheet, database manager, terminal program, text editor, or compiler? Desktop Notepad lets you jot down your note without interrupting whatever main application you're running.

Figure 2-11. "Desktop Notepad" Running Concurrently with Another Application



Although it's not a full-featured word processor, Desktop Notepad does support basic word processing functions. You can insert text at any point; delete single characters or entire blocks; move or copy blocks of text from one place to another within a document; search for specified strings of text; search and replace strings; create documents of any size up to the limit of available memory; save and load text on disk; and print out copies of documents on a printer. Desktop Notepad takes full advantage of GEM features, including a movable and resizable notepad window, scroll bars, and a cursor that's controllable with the mouse or the keyboard. And, thanks to its instantaneous word-wrapping, you can freely enter text without worries about formatting.

Desktop Notepad will be a significant addition to nearly anyone's software library.

Installing the Program

Desktop Notepad must be installed as a desk accessory. That means the program is automatically loaded into memory when you turn on your ST, and it waits for you to call it into action by selecting it from the Desk menu. Most programs that support GEM make this menu available at the far left side of the menu bar. (Sometimes the Desk menu is labeled with the Atari logo, as in *1ST Word*.) As long as the Desk menu is available, Desktop Notepad is ready to be called up at the click of a button.

To prepare Desktop Notepad for use, simply copy the file NOTEPAD.AC from *COMPUTE!'s Second Book of Atari ST Disk* to your boot disk (the disk you insert in drive A when you switch on your ST). Then rename the file to NOTEPAD.ACC. The .ACC extender tells the ST's operating system to automatically load Notepad as a desk accessory when the computer is switched on. NOTEPAD.ACC must be on the root (main) directory of your boot disk; do not place it in a folder (subdirectory).

There's one exception to this rule: If you're using a hard disk drive, you may place your desk accessory files on the root directory of disk C.

Note: If you have an early 520ST without the TOS operating system in read only memory (ROM), you should rename NOTEPAD.AC to DESK5.ACC after copying it to your boot disk. However, Desktop Notepad may not work properly with some applications if your 520ST lacks TOS in ROM. The TOS upgrade costs about $35 at your local Atari dealer's and is highly recommended. If you already have a DESK5.ACC file on your boot

disk, rename NOTEPAD.AC to DESK*x*.ACC, where *x* is any number less than six.

The current version of TOS allows up to six desk accessories and one primary application program in memory at a time. Note that the Control Panel and VT-52 emulator accessories supplied with your ST count as *two* accessories *each,* placing two options each on the Desk menu. This means that if you install the Control Panel and VT-52 emulator, there's only enough room for two more accessories of your own. Desktop Notepad counts as one accessory.

There is one potential problem in using desk accessories, particularly large ones. Some accessories occasionally interfere with some application programs. If the application you're using is behaving strangely, try disabling all the accessories (perhaps by renaming the extenders from .ACC to .AC) and rebooting the computer. Then try your application again to see if it behaves properly. This shouldn't be a problem with Desktop Notepad, because it's a relatively small accessory. However, it does have the capability of loading very large text files. (See below.)

After placing NOTEPAD.ACC on your boot disk, insert the disk in drive A and switch on the computer. If it seems that the GEM desktop takes slightly longer to appear than usual, don't be alarmed; it takes a few seconds longer for the computer to load desk accessories into memory. When the desktop appears, confirm that Desktop Notepad is installed by dropping down the Desk menu. You should see a selection called Notepad. You can open the Notepad by clicking on this selection.

Full-Screen Editing

When Desktop Notepad opens, the screen should resemble Figure 2-12. Try typing a line of text. Notepad works much like *ST Writer* and *1ST Word;* all alphanumeric keys and cursor control keys are active. However, Notepad is permanently locked in insert mode, so whatever you type pushes existing text ahead of it.

Notice that when the cursor reaches the end of a line, the text automatically continues on the next line, and no words are broken at the right margin of the Notepad window. This feature is called *word-wrapping* or *parsing* and is found in most word processors. As a result, all text entered into Notepad is flushed against the left screen margin, leaving a ragged right margin.

Figure 2-12. "Notepad": A Blank Slate Awaiting Input

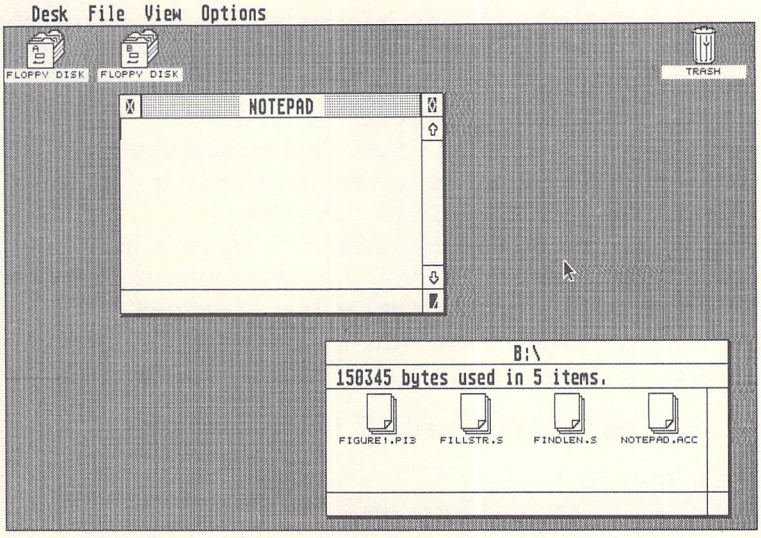
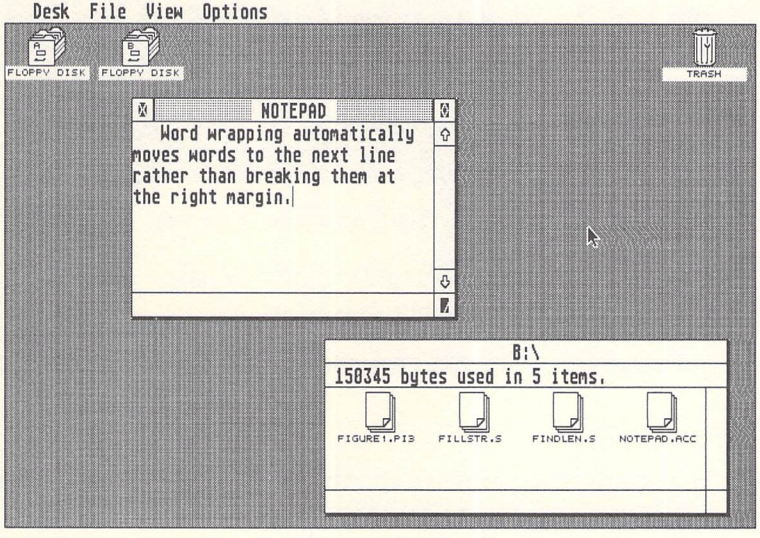


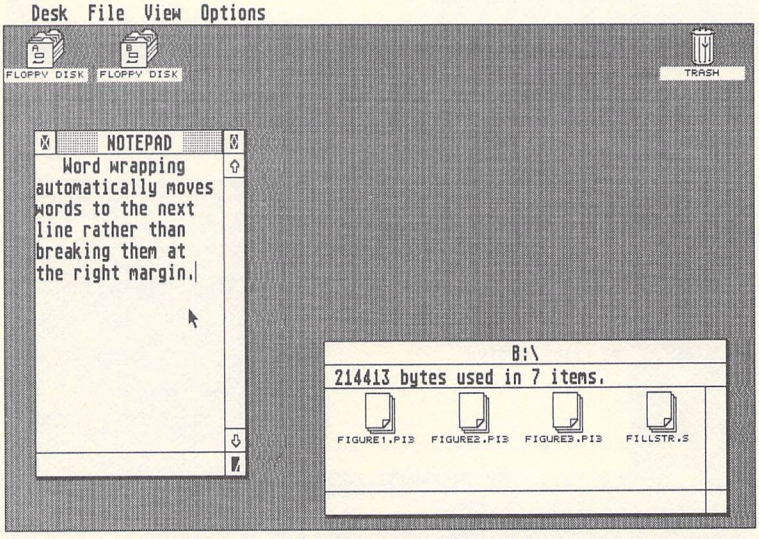
Figure 2-13. Word-Wrapping in "Notepad"



Notice that the Notepad window is equipped with standard GEM gadgets. By pointing at the window's title bar and pressing the left mouse button, you can drag the window to any position on the screen. By dragging the lower right corner of the window, you can resize the Notepad. By clicking on the tiny button in the upper right corner, you can expand the Notepad to full-screen size and shrink it back down again. The vertical slider along the right edge of the window lets you scroll through a note that might be too long to fit in a single window.

One interesting feature of Desktop Notepad is that it automatically reformats the word-wrapping whenever the window is resized. This makes it unnecessary to use a horizontal slider along the bottom edge of the window. Even some full-blown word processors on the ST lack this feature.

Figure 2-14. Text Reformatted for Resized Window



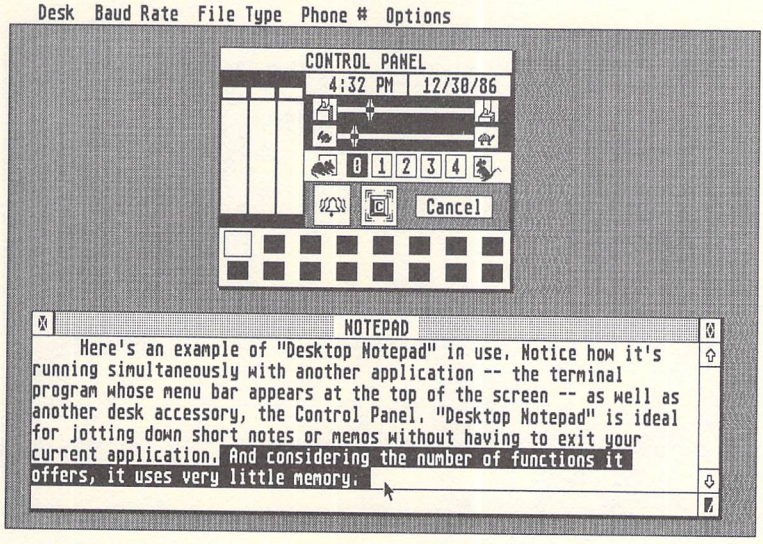
Editing Controls

The keyboard and mouse work much the way you'd expect them to. You can move the cursor a character at a time in any direction with the cursor keys, or move it directly to any point in the text by pointing and clicking the mouse. The Backspace key erases the character immediately behind the cursor. The Return key inserts a carriage return and lets you start a new paragraph.

If you want to delete a lot of text, it can be tedious to keep

pressing the Backspace key. Instead, define the block to be deleted by holding down the left mouse button and dragging either forward or backward. You can define text which doesn't appear in the window by dragging the mouse above or below the window—this automatically scrolls the text in the window. The defined block appears in reverse video. If you change your mind at this point, you can undefine the block by holding down the Control key while clicking the left mouse button.

Figure 2-15. Reverse-Video Block Defined by Dragging the Mouse



Once a block is defined, you can delete it with a single mouse-click. These and other functions are controlled by the special menu seen in Figure 2-16. The special menu appears whenever you drop down the Desk menu and reselect the Notepad item *while the Desktop Notepad window is already open.* The special menu disappears after you've selected a function or when you click the mouse button without selecting a function.

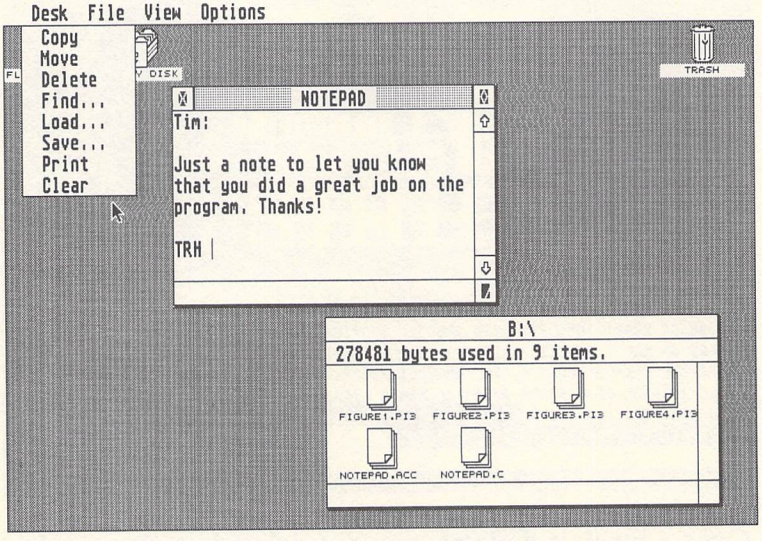
Other items on the special menu let you move or copy defined blocks of text, search and replace strings, save notes on disk, or recall notes that you saved earlier. To move a block of text, first define it as described above. Next, place the cursor at the position within your note where you want to move the block. Then select Move from the special menu. The text block

will be deleted from its original position and moved to the new position.

Copying a block of text to another place in your note is just as easy. Define the block to be copied, move the cursor to the position where you want the text inserted, and select Copy from the special menu. The text block will be copied to the new position and left unchanged in the original position.

To find a string of text or search and replace a string, select Find from the special menu. A dialog box opens to present another collection of functions (Figure 2-17).

Figure 2-16. The "Desktop Notepad" Special Menu



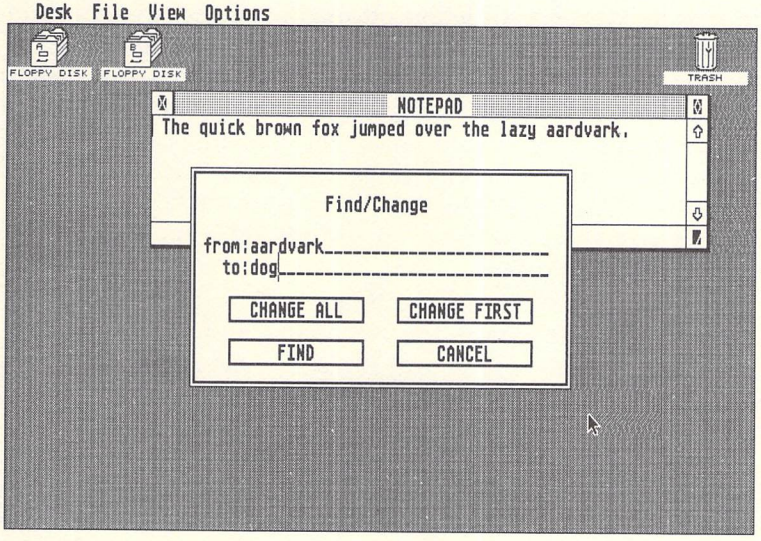
To find a string of text, type the string on the FROM: line. Then click on the FIND button. The cursor will move to the next occurrence of that string within your note. Keep in mind that all search operations begin at the current location of the cursor. If you want to search through the entire note, first move the cursor to the beginning of the text.

To search for a string of text and replace it with another string, first type the string you're searching for on the FROM: line. Next, type the string you want to replace it with on the TO: line. Then click on the CHANGE FIRST button to replace only the first occurrence of the string, or CHANGE ALL to replace all occurrences of the string. Again, remember that search-and-

replace operations begin at the current location of the text cursor.

If you selected the Find function by accident or changed your mind, click on the CANCEL button to exit the dialog box and return to the Notepad window.

Figure 2-17. Functions Available with the Special Menu's Find Option



Saving and Loading

To save or load a note, select the Save or Load functions from the special menu. Desktop Notepad saves all notes in a format known as ASCII (American Standard Code for Information Interchange). That is, the note is saved as a block of text with no special imbedded characters. In this form, the text can be uploaded to another computer via modem or loaded into a word processor or text editor that handles ASCII files. Likewise, Desktop Notepad can load a text file that has been saved in ASCII format by another word processor or text editor.

One note, however: Many word processors and text editors on the ST save ASCII files a little differently from the way similar programs on other computers save them. ASCII files on the ST often have a carriage-return character and a linefeed character at the end of every line of text. The carriage-return character is invisible, but makes it appear that Desktop Notepad's automatic

word-wrapping isn't working; the text breaks in strange places along the right margin of the Notepad window. The linefeed character may appear on the screen as a bell character. Similarly, tab characters inserted by some word processors may appear on the screen as clock characters.

You can ignore these characters when editing a document with Desktop Notepad, especially if you plan to load the file later into the word processor or text editor that originally created it. Alternatively, you can delete all of the carriage return, linefeed, and tab characters with Desktop Notepad's search-and-replace function. To do this, select Find from Notepad's special menu. To enter the search string on the FROM: line, type Control-M for the carriage-return character, Control-J for the linefeed character, or Control-I for the clock character. Don't type anything on the TO: line. When you click on the CHANGE ALL button, the character specified on the FROM: line will be deleted throughout the document.

This operation may take quite some time with a long document. To speed it up, make the Notepad window very small before beginning the search and replace. This reduces the amount of reformatting the program must do each time it deletes a character.

Remember that if you delete all of the carriage returns or linefeeds in this manner, you may have trouble reformatting the document if you load it back into the word processor or text editor that originally created it. Experiment first before altering an important document.

The File Selector

When you select the Save or Load functions from Desktop Notepad's special menu, a standard GEM file-selector window opens up to prompt you for a filename. This works the same as all file selectors on the Atari ST. Simply enter any standard ST filename (up to eight characters with an optional three-character extender), and press Return or click on the OK button. Click on Cancel to abort.

To change a pathname, click on the dotted line at the top of the file-selector window. Press Backspace or Esc to erase the old pathname, type the new pathname, and click on the directory box to display the new directory. For example, if you need to load a text file on drive B, type B: on the pathname line. Consult the manual that came with your ST if you need further infor-

mation on using file selectors and pathnames.

Desktop Notepad's special menu also lets you print the text currently loaded into the Notepad window. Make sure your printer is powered up, connected, and online; then select the Print function. The note will be printed as it appears in the Notepad window, including line breaks. The entire note is printed even if it doesn't fit inside the Notepad window.

Closing and Reopening

As you're writing a note, you may find that you need to check an item in the application you're working with, or you may just want to continue with the application. If so, you can close the Notepad without losing the note it contains.

When you click on the close button in the upper left corner of the Notepad window, the Notepad window goes away. The next time you reopen the Notepad window by selecting Notepad from the Desk menu, it reappears with the text intact. If you want to erase the contents of the Notepad, select the Clear function on the special menu. This lets you start a new note if desired.

CHAPTER THREE

Disk Utilities

File Hider

David T. Jarvis

Use the two programs described here to render disk files invisible on directories to protect them from snoopers; restore them to visibility; call up extended directories of disks; and more. The programs work on any ST in any screen mode: low- or medium-resolution color, and high-resolution monochrome.

"Reinventing the wheel" violates a basic rule of design: *Don't waste time doing something that's already been done well.* But it can be useful to reinvent a wheel now and then. Benjamin Franklin reportedly taught himself to write by rewriting, from memory, works of other writers and then comparing his results to the originals. You can take the same approach with programming. It can be instructive to write programs that perform functions already provided by your computer's operating system; what better way to learn how the system works?

There can also be practical results from this. By rewriting basic system functions, you can tailor them to your own needs or preferences. On the Atari ST, for instance, you might occasionally want to enter system commands with an "old-fashioned" command line interpreter instead of using the mouse controller and GEM (Graphics Environment Manager) desktop. Or, more to the point, you might want to modify the disk directory function so that certain files you'd like to conceal from certain eyes don't appear in a directory window. Writing a custom directory program is an important step toward accomplishing this goal.

Such a program can be found on the disk under the filename XDIR.PRG. As we'll describe in more detail in a moment, "XDIR" provides an extended directory listing of a disk by calling low-level routines within the ST's operating system. It reveals disk information which isn't normally available in a standard GEM directory window.

The other program is "File Hider," stored on the disk as FILEHIDE.PRG. File Hider lets you hide filenames that would normally appear in a directory. In effect, you can protect sensitive data against casual snoops by rendering the file invisible.

Using XDIR

Let's cover the extended directory program first. To get started, you must make sure that XDIR is properly installed as a special type of TOS (Tramiel Operating System) application. Follow these steps:

1. Copy the file XDIR.PRG from *COMPUTE!'s Second Book of Atari ST Disk* to one of your own disks. Although you can run XDIR from the book disk, it's a good idea to keep the original copy as a backup.
2. Click once on the XDIR.PRG icon or filename to highlight it. Don't double-click it at this point.
3. Drop down the Options menu and select Install Application.
4. When the dialog box appears, click on the button labeled TOS Takes Parameters to make sure it is selected. Then click on the OK button.

XDIR is now installed and ready to run. If you want to avoid repeating steps 2 through 4 whenever you reboot the computer, drop down the Options menu again and select Save Desktop. This insures that XDIR will be installed as a TTP (TOS Takes Parameters) application whenever you boot from that disk. Otherwise, you'll have to select Install Application during each session.

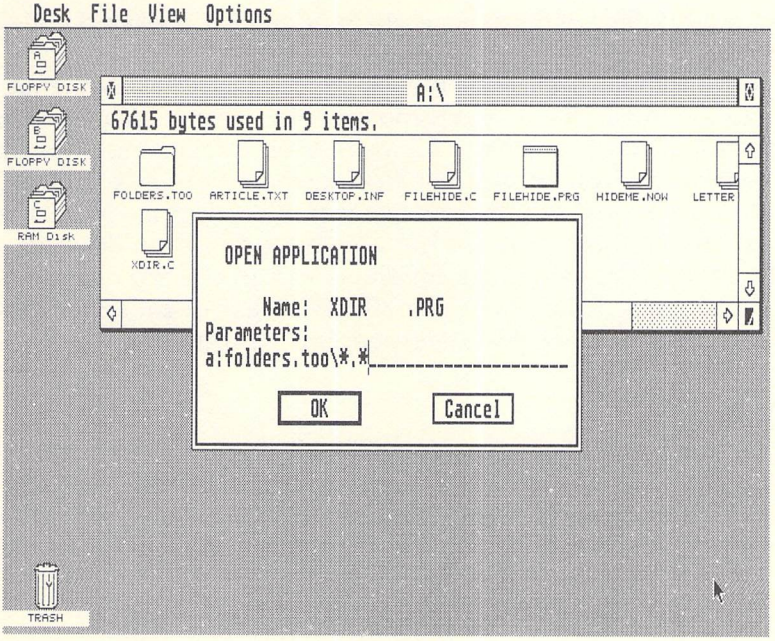
To run XDIR, double-click on the XDIR.PRG icon or filename. A dialog box opens to prompt you for a disk pathname. If you press Return or click on the OK button without typing anything, the default is the usual *.*, which displays every file and folder on the disk. You can change this pathname, of course, just as you would with a GEM item selector. For instance, to view the contents of a folder called FOLDERS.TOO on disk A, you would enter A:FOLDERS.TOO *.* (Figure 3-1). For more information on pathnames, consult the manual that came with your ST.

When you press Return or click the OK button, XDIR clears the screen and displays the disk directory. To exit the directory and return to the GEM desktop, press any key.

File Attributes

You'll notice that the XDIR directory provides a few pieces of information normally missing from a GEM directory window: the amount of free space on the disk, the disk label, and any special *attributes* of each file, as we'll explain in a moment. In addition, folders (subdirectories) are denoted with <DIR> and the letter *D*,

Figure 3-1. Entering the Pathname with "XDIR"



and files concealed with the File Hider utility are marked with an *H*. If you're viewing the contents of a folder, the first few entries in the directory tell you how deep you are in the directory structure. You'll see a period and a <DIR> for each directory level, including the root (main) directory.

File attributes are special characteristics of disk files on the ST. A file's attributes are encoded in one byte; a particular attribute is given to a file by setting the corresponding bit of the attribute byte. With the current version of TOS, a file may possess the following attributes:

Attribute Value	Characteristic
1	Read-only; file cannot be altered.
2	A hidden file; filename won't appear on normal directories.
4	System file.
8	The file is a disk label (always an empty file).
16	The file is a subdirectory (folder).
32	Used for archival purposes.

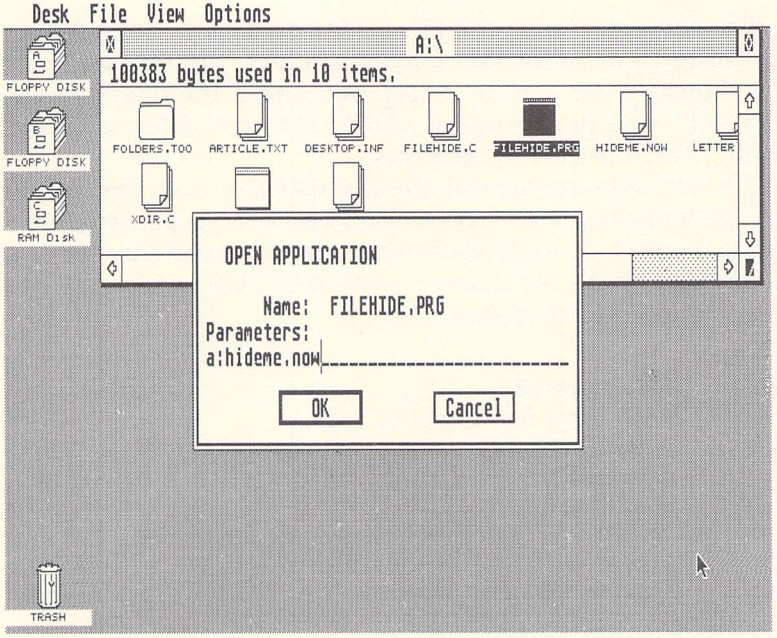
A file can have more than one attribute. For example, a file with an attribute byte containing the value 3 would be a read-only, hidden file. A file whose attribute byte is 0 has no special attributes.

Examining a file's attributes is interesting, but being able to change the attributes is much more useful. You can change the read-only attribute from the GEM desktop by clicking on a file and selecting Show Info; then click on the Read-Only button in the dialog window which appears. But other attributes, such as the one for hiding a file, are not accessible from the desktop. Fortunately, TOS contains a low-level call to read or change a file's attributes. That's how File Hider works—it calls this routine to let you set or clear the corresponding attribute bit.

Hiding Files

Before using File Hider, you must install the program in the same way you installed XDIR. Copy FILEHIDE.PRG to another disk, click once on the icon/filename, select Install Application from the Options menu, click on the button labeled TOS Takes

Figure 3-2. Making a File Invisible with "File Hider"



Parameters, and then click on the OK button. Again, you may want to select Save Desktop to avoid the trouble of reinstalling the application in the future when booting from that disk.

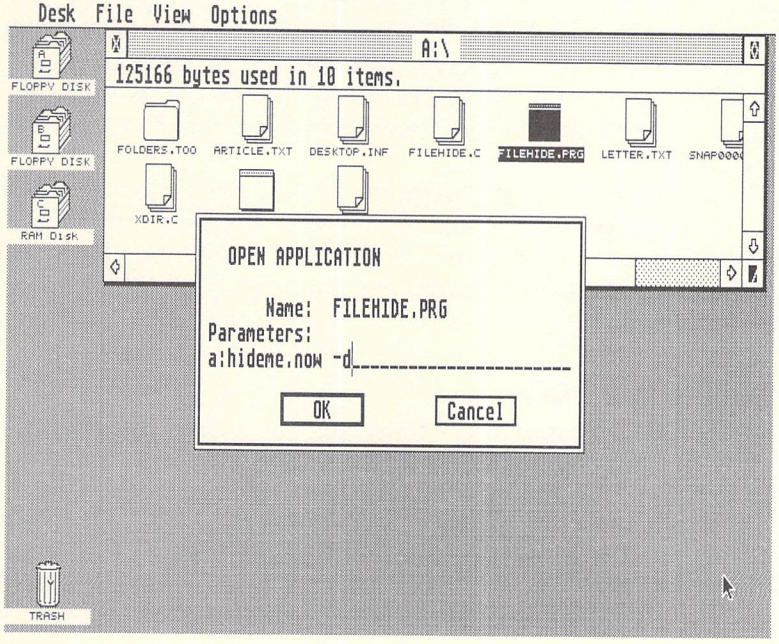
To run File Hider, double-click on the icon/filename. A dialog box opens to prompt you for the pathname and filename of the file you want to hide. For example, to see how to hide a file called HIDEME.NOW on the root directory of drive A, look at Figure 3-2.

After you've entered this information and pressed Return or clicked on the OK button, File Hider clears the screen and attempts to render the file invisible. If it succeeds, you'll see a verification message; press any key to exit back to the GEM desktop. You can confirm that the file is hidden by opening (or reopening) the directory window for that disk. Or you can run XDIR and observe that the hidden filename is denoted with an *H*.

To reverse the process and make a file visible again, rerun File Hider. But when you enter the pathname and filename, add a space character and the parameter *−D* to the end of the filename. See Figure 3-3 for an example.

This restores the file to visibility as if nothing has happened.

Figure 3-3. Making a File Visible Again



File Lister

Richard Smereka

This utility greatly enhances the file-listing functions of the Atari ST. It works in any screen mode and runs as a stand-alone program on the GEM desktop or as a command with ST-Shell. *A printer is optional, but recommended.*

Normally when you want to examine a text file on the Atari ST, you click on the appropriate icon or filename with the mouse, then click on the Show or Print buttons inside the dialog box that appears. This either displays the file on the screen or dumps it to a printer.

Although this built-in function is sufficient for most purposes, the resulting output is raw and unformatted. A quick look at what's available in other operating systems reveals that there's plenty of room for improvement. That's the reasoning behind "File Lister," an enhanced file-viewing utility. It adds several extra features to control the appearance of the final output. The program runs on any ST in any screen mode, but for obvious reasons it works best in the 80-column modes (medium-resolution color and high-resolution monochrome). It can be used as a stand-alone application from the GEM desktop or as a command with a command-driven disk operating system such as *ST-Shell.* (See Chapter 4.)

To generate hardcopy with File Lister, you'll also need a compatible printer. File Lister requires no special printer driver to control the printed listing, but the printer must be capable of interpreting the form-feed character (character code $0C) to advance to the next page.

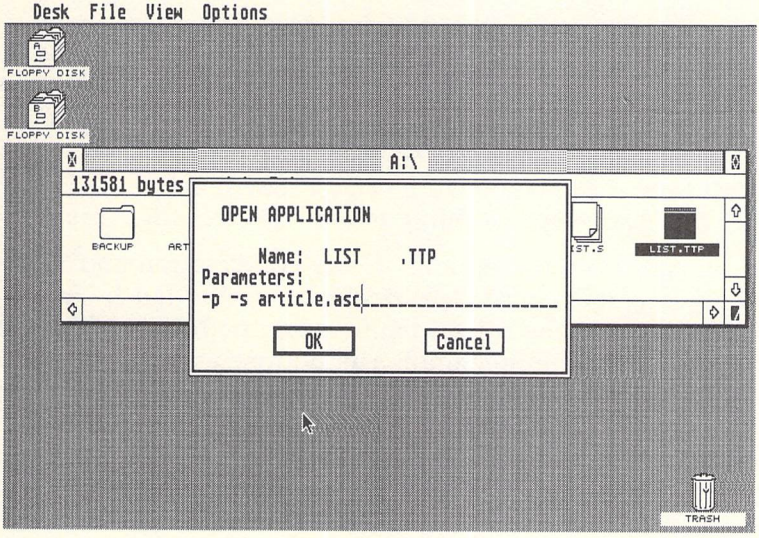
Preparing File Lister

There are two different ways to install File Lister, depending on whether you plan to use it from the GEM desktop or from a command line interface such as *ST-Shell.*

To use it from the desktop, name the file LIST.TTP (it's already stored in this form on the book disk). The .TTP filename extension is important. It stands for *TOS Takes Parameters,* which signals the ST that File Lister is a Tramiel Operating System pro-

gram that requires certain parameters before it can function. When you run a TTP application by double-clicking on its icon, a dialog box pops open on the screen so you can enter these parameters. Figure 3-4 shows an example of this box.

Figure 3-4. Dialog Box Available When "File Lister" Is Run from the GEM Desktop



If you plan to use File Lister from a command-driven DOS such as *ST-Shell,* copy LIST.TTP to your *ST-Shell* disk and rename it LIST.PRG.

File Lister expects these parameters:

[options] [D:][\pathname \]filename.ext

The only parameter that's absolutely required is *filename.ext*—the name of the ASCII text file you wish to view or print. The parameters within brackets are optional and must be separated from each other by at least one space. (Do not type the brackets.) Let's take a look at what these parameters do.

File Lister Options

The most obvious parameters let you specify a disk drive and pathname. If the file you wish to examine is on another drive, substitute the drive identifier for *D:*. If the file is in a folder, enter the *pathname* between backslash characters. For example, if

you're running File Lister from drive A and the file you want to view is called READ.ME in a folder called TEXTFILE on drive B, you enter

B:\TEXTFILE\READ.ME

If you're not sure how to use pathnames, refer to the user's manual that came with your ST.

To specify additional *options*, refer to the list below:

−p printer output (default: printer output off)
−n print line numbers (default: line numbers off)
−z pad line and page numbers with zeros (default: zeros on)
−s screen output (default: screen output on)
−w screen wait after 18 lines (default: wait on)
−f full printer format (default: printer format on)
−t TTP pause before returning to desktop (default: pause off)

You can use these options in any combination and in any order, as long as they're separated from each other by at least one space (except the drive identifier and pathname, which must be together with the filename). The options are like switches—by default they assume an automatic position, on or off. When included in the command line, their normal default state is reversed.

The options may be freely mixed, although some options depend on the state of others. For example, giving the *−f* option on the command line will have no effect unless the *−p* option is on, because there is no sense in changing the printer format if no printer output is requested.

Detailed Examples

Following are some typical ways in which you might use File Lister. Remember to name the program LIST.TTP if you're running it from the GEM desktop, or LIST.PRG if you're running it from *ST-Shell.*

Desktop example: SAMPLE.TXT
***ST-Shell* example: LIST SAMPLE.TXT**

This simply lists the text file SAMPLE.TXT to the screen, pausing every 18 lines for a keypress to continue scrolling.

Desktop example: −p SAMPLE.TXT
***ST Shell* example: LIST −p SAMPLE.TXT**

The *−p* option tells File Lister to send the text file to the

printer *as well as* to the screen. (The *−s* screen option is turned on by default.) Note that normally the printer format option *−f* is also switched on, so you get the full printer format. (Option *−f* is discussed below in more detail.)

If, for any reason, File Lister cannot properly communicate with the printer (for instance, if the printer is not powered up or online), you'll see the error message *Trouble Communicating With Printer,* and the list request will be terminated.

Desktop example: −p −n SAMPLE.TXT
***ST-Shell* example: LIST −p −n SAMPLE.TXT**

This example lists SAMPLE.TXT on the printer *and* prints line numbers at the beginning of each line of text. Among other things, this feature is useful when you're documenting or debugging source code. The line numbers range from 1 to 9999 and are padded with leading zeros (0001, 0002, and so on; see Figure 3-5).

Desktop example: −p −n −z SAMPLE.TXT
***ST-Shell* example: LIST −p −n −z SAMPLE.TXT**

This command lists SAMPLE.TXT on the printer and prints line numbers at the beginning of each line of text, but turns off the extra zeros and pads the numbers with spaces instead.

Desktop example: −p −n −z −s SAMPLE.TXT
***ST-Shell* example: LIST −p −n −z −s SAMPLE.TXT**

This lists SAMPLE.TXT on the printer with line numbers that are padded with spaces instead of zeros, but turns off screen output. Note that since screen output is normally on, the *−s* option turns it off when included in the command line.

Desktop example: −p −n −f SAMPLE.TXT
***ST-Shell* example: LIST −p −n −f SAMPLE.TXT**

This lists SAMPLE.TXT on the printer and the screen with line numbers padded with zeros, but the *−f* option turns off the full printer format. The full printer format consists of 55 lines per page with a header at the top of each page consisting of the filename and page number. Since the full printer format is turned on by default, the *−f* option turns it off when included in the command line.

Desktop example: −w SAMPLE.TXT
***ST-Shell* example: LIST −w SAMPLE.TXT**

This command lists SAMPLE.TXT on the screen *without* waiting for a keypress every 18 lines. In other words, the text

Figure 3-5. A Page of Source Code (dumped to printer)—Line Numbers and Headers Provided by "File Lister"

```
File Lister

File: MOLLY.C                                                    Page 0001

0001
0002 /* MOLLY.C: Atari ST graphics demo for any resolution */
0003 /* by Tim Victor and Philip Nelson, October, 1986     */
0004
0005 #include <osbind.h>
0006
0007 int handle;
0008 int input[11],output[57];
0009 int intin[100],intout[100],ptsin[100],ptsout[100],contrl[12];
0010 int width,height,colors;
0011
0012 int xlate[] = {                /* Translate VDI colors to hardware */
0013      0,2,3,6,
0014      4,7,5,8,
0015      9,10,11,14,
0016      12,15,13,1
0017      };
0018
0019 int pal[] = {                  /* Palette for color cycling        */
0020      0x700,0x720,0x750,0x770,
0021      0x370,0x070,0x072,0x075,
0022      0x077,0x037,0x007,0x207,
0023      0x507,0x707,0x703
0024      };
0025
0026 main()
0027 {
0028      int i,rez,style;
0029      int draw_tone,draw_color;
0030      int pal_phase,pal_tone;
0031      int pts[4],savepal[16];
0032      int xpos,ypos,rad;
0033      int xdir,ydir,rdir;
0034      int most = 20;
0035
0036      for (i=0;i<10;i++)
0037           input[i] = 1;
0038      input[10] = 2;
0039
0040      handle = 1;
0041      v_opnvwk(input,&handle,output); /* Open virtual workstation */
0042      v_hide_c(handle);               /* Hide mouse */
0043
0044      width = output[0];      /* Current workstation width */
0045      height = output[1];     /* Current height */
0046      colors = output[13]-1; /* Number of colors not counting backgrnd */
0047
0048      pts[0]=pts[1]=0;
0049      pts[2]=width;
0050      pts[3]=height;
0051      vs_clip(handle,1,pts);          /* Turn on clipping */
0052
0053      /* Save current color palette */
0054      for(i=0;i<16;i++) savepal[i] = Setcolor(i,-1);
0055
```

scrolls by at full speed. This is useful when you want to quickly scan through a file. Note that if the $-s$ option is off, $-w$ has no effect.

You can also activate this option *after* you've started listing a file on the screen. Normally when File Lister displays a file, it pauses every 18 lines and prints this message: *A = Abort, N = No Wait, Any Other Key to Continue.* Pressing A aborts the listing and returns you to the GEM desktop or *ST-Shell.* Pressing N turns off the wait feature, and pressing any other key resumes the listing.

Desktop example: −t SAMPLE.TXT
***ST-Shell* example: LIST −t SAMPLE.TXT**

This makes File Lister pause after it has finished displaying a file; normally, it's useful only when you're running File Lister as a TTP application from the GEM desktop. If you don't include this option, File Lister exits to the desktop so quickly that you might not see any messages that are generated before it quits. The *−t* option forces a pause. If you're running File Lister from *ST-Shell*, this option is unnecessary because the text remains on the screen when the command prompt reappears.

File Lister Batch Files

If you're using File Lister with *ST-Shell*, you can set up a series of batch files in advance with your favorite listing options. Then, rather than typing in a long list of options on a command line, you can simply execute the appropriate batch file. Note that to take advantage of this you must have a copy of *ST-Shell* or another command-driven DOS with similar batch file capabilities.

For instance, you could have two main batch files: one to list a file to the screen, and another to dump a file to the printer. Look at this one-line batch file:

```
list −n %1
```

Let's call it SCREEN.BAT. To call this batch file from *ST-Shell*, all you'd type at the screen prompt would be SCREEN SAMPLE.TXT. This would display the text file SAMPLE.TXT with zero-padded line numbers, pausing for a keypress every 18 lines.

Here's an example of a one-line batch file for listing to the printer:

```
list −p −n −z −s %1
```

Let's call it PRINT.BAT. When you enter the command PRINT SAMPLE.TXT at the *ST-Shell* prompt, this batch file dumps SAMPLE.TXT to the printer with space-padded line and page numbers, and suppresses screen output.

About the Program

File Lister was written in machine language instead of a high-level compiled language in order to minimize the size of the program. It also runs faster than a program written in a compiled

language, although the limiting factor is more likely to be the speed of the printer and screen-scrolling.

The small size of File Lister is important, because it leaves room on an *ST-Shell* disk for a number of utilities of this type. (See "Extended Formatter" in Chapter 4.) These utilities are really extrinsic or external DOS commands, just like those found in MS-DOS, PC-DOS, CP/M, and UNIX.

If such utilities were written in a compiled language, they might be about 10K each. There would be only enough room on a single-sided disk for about 25–30 of them. But if each utility were a maximum of 2.5K (as this one is), there'd be room for about four times as many on the disk. Of course, shorter utilities also load faster and use less memory.

Directory Dump

Marcos Zorola

A short, useful utility that helps you keep track of your disk files, "Directory Dump" reads the disk directory and sends a formatted copy to a printer, the screen, or another device. It works on any ST in any screen resolution, color or monochrome.

Faced with the question of how to print out a disk directory, many ST owners resort to a rather inelegant solution: Double-click on a disk icon to display the directory window; then press Alt-Help to start a time-consuming graphics screen dump. If the directory is too long to fit on a single screen, several screen dumps are necessary. To complicate matters, there may be folders (subdirectories) on the disk, each of which must be displayed and printed separately. There's no other way to print a disk directory from the GEM desktop.

"Directory Dump" provides a much better alternative. It lets you send a listing of an entire disk directory—as text, not graphics—to a printer or any other legal device or file. Legal devices include the printer, the screen, and even another disk file.

Directory Dump prints out the filenames in the order in which they appear on the disk. In general, this means the files are listed in the order in which they were saved. Deleting a file opens up a slot in the directory, however, so the next file you create appears in the newly available slot.

Folders and their contents are included in the listing, too. The only limitation is that Directory Dump cannot handle disks that have 50 or more folders (including nested folders). In practice, this isn't a severe limitation, even on hard disks up to 20 megabytes in size.

Directory Dump also lets you choose the type of information that is displayed in the directory listing. Coupled with its flexible output, this gives you the ability to customize the listings to suit your requirements.

Running the Program

Directory Dump is on the disk under the filename DIRDUMP.PRG. You can run Directory Dump directly from the disk menu, or

Figure 3-6. Sample Printout from "Directory Dump"

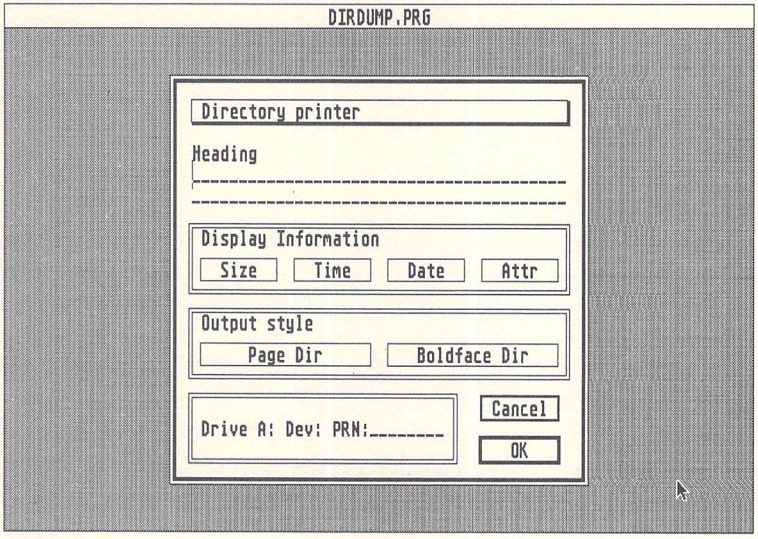
```
     PRNCTL  .ACC          17901      11/20/1985      00:11:36      Read/Write
     PRNCTL  .BTN            275      11/20/1985      01:58:18      Read/Write
     PRNCTL  .HLP           4079      11/20/1985      00:05:56      Read/Write
     PRNCTL  .PRG          17505      11/20/1985      00:11:42      Read/Write
     PRNCTL  .RSC           5488      11/20/1985      00:02:58      Read/Write
     PRNCTLLW.RSC           6036      11/20/1985      00:04:58      Read/Write
     SNAPSHOT.AC            3619      11/20/1985      00:11:16      Read/Write

Total of 54903 byte(s) in 007 file(s) in \.


\NEOCHROM.FLD\

     NEOVIEW .TOS           1132      11/20/1985      00:19:24      Read/Write
     NEOVIEW .TTP            436      11/20/1985      00:20:00      Read/Write

Total of 1568 byte(s) in 002 file(s) in \NEOCHROM.FLD\.


\GAMES.FLD\

     LASERCHS.PRG          28104      11/20/1985      00:28:58      Read/Write

Total of 28104 byte(s) in 001 file(s) in \GAMES.FLD\.


\DIRDUMP.FLD\

     DIRDUMP .MOD          20020      11/20/1985      00:24:04      Read/Write
     DIRDUMP .PRG          16238      11/20/1985      00:24:30      Read/Write

Total of 36258 byte(s) in 002 file(s) in \DIRDUMP.FLD\.


\GAMES.FLD\PUZZLER.FLD\

Total of 0 byte(s) in 000 file(s) in \GAMES.FLD\PUZZLER.FLD\.


\GAMES.FLD\PUZZLER.FLD\PUZZLER.FIN\

     PUZZLER .C            13601      11/20/1985      00:29:42      Read/Write
     PUZZLER .PRG           7974      11/20/1985      00:29:50      Read/Write

Total of 21575 byte(s) in 002 file(s) in \GAMES.FLD\PUZZLER.FLD\PUZZLER.FIN\

Grand total of 142408 byte(s) in 0014 file(s).
```

from the GEM desktop in the usual fashion: Double-click on the DIRDUMP.PRG icon or filename. When the program starts, you'll see a screen like the one shown in Figure 3-7.

In the Heading field, type a name for the disk whose directory you're printing. Whatever you place there will be printed at the top of each directory and folder listing. It's a good idea to type the name of the disk as it appears on your disk label—"BASIC Programs," for example—so when you're looking

through the listings, you can match up disks with their printed directories. You may also want to include the date and whether the disk is formatted for single- or double-sided use.

Figure 3-7. Opening Screen for "Directory Dump"



Customized Listings

Below the Heading field is the Display Information field. The four buttons here control what sort of information is printed. If all four buttons are turned off, the listing contains the filenames only, each printed under the folder in which it's located. By clicking on the buttons, you can select any one (or several) of the following options:

- Size—The size of the file in bytes is listed.
- Time—The time of the file's creation is listed.
- Date—The date of the file's creation is listed.
- Attr—The file's attribute is listed. This may be READ/WRITE or READ ONLY.

If you don't usually use the ST's Control Panel accessory to set the time and date, most files will have very similar date and time stamps, and you'll probably want to omit this information from the directory listing. The READ/WRITE and READ ONLY attributes for a file can be set by selecting Show Info from the File menu on the GEM desktop. If you never write-protect any of

your files, there's no need to print the file attribute.

The second set of options controls the output style:

- Page Dir—Each folder starts printing on a new page.
- Bold Dir—The folder names are printed in boldface.

The Bold Dir option makes the folders stand out when the Page Dir option is not used. Directory Dump sends an escape code for boldface which is recognized by most Epson and Epson-compatible printers. If the boldface feature doesn't work properly with your printer, just ignore this option.

Changing Devices

The Directory Dump window also has two additional fields you can fill in: the Drive field and the Device field. These fields hold default values (drive A and device PRN:) when Directory Dump is first run.

The Drive field specifies which drive will be accessed for the listing. In other words, if the Drive field is set to A, the disk whose directory you want to list should be inserted in drive A. If you use a single-drive system, leave the drive set to A. If you have additional floppy drives or hard drives, you may enter drive specifiers B–P, depending on which drives are currently installed.

It's also possible to use Directory Dump to catalog a RAM disk, although there isn't much point in doing so, since the RAM disk disappears when you turn off the ST.

The Device field lets you specify which device will be used for output. Legal device names are

PRN:	Printer (the default)
AUX:	RS-232 port
***X*:FILE.EXT**	Disk file on drive *X* (A–P)
CON:	Console (screen)
HSS:	MIDI port

Normally you'll leave this field alone. By typing in a different device name, you can redirect output to any available ST output device shown above. If you change it to CON:, for instance, the output goes to the screen. Note that the colons are a required part of the device names.

Creating a Disk File

By sending the directory listing to a disk file, you could edit it with a word processor or text editor and perhaps merge a number of listings together to make a catalog. Type the drive identifier, following it with a filename. The filename must be limited to 12 characters, counting the period and the optional extender. For example, to create a file called DIR1.CAT on drive A, type A:DIR1.CAT. The result is an ASCII file containing the directory listing.

To send the listing to a printer attached to the parallel printer port, leave the Device field set to PRN:. If you have a serial (RS-232) printer that has been installed as such with the Install Printer accessory that came with your ST, it should also work with PRN:. If not, change the Device field to AUX:. Theoretically, you could also send the listing to a modem via AUX:, but it would make more sense to create a disk file and then upload it.

The HSS: device lets you send a directory listing to the MIDI (Musical Instrument Digital Interface) port. This capability wasn't included in the program on purpose; it's just a consequence of the ST's device-oriented input/output flexibility.

When you've made all the selections described above, make sure your printer is connected and turned on (if you're directing output to that device); then click on the OK button. If you decide to exit the program instead, choose the CANCEL button.

File Finder

Richard Smereka

When you need to discover the whereabouts of a file on a crowded disk, simply tell "File Finder" the name of the file you're looking for. Like a bloodhound, it sniffs out every occurrence in every folder on a given disk. It's a must if you own a hard disk drive. The program works on all STs in any screen resolution, color or monochrome.

What do you do when you've forgotten where a certain file is stored? If you have 20 disks, each containing an average of 10 folders, you have 200 folders you might have to look through. If you're lucky, you may find the misplaced program after a few tries. If not, well, you'll have to open a lot of folders.

"File Finder" quickly locates a particular file on a crowded disk. It's very handy for those times when you know the name of the file but have forgotten which folder it resides in. File Finder searches the entire disk, whether it's single-sided, double-sided, a RAM disk, or a hard disk. It reports all occurrences of the specified filename on the disk. As another option, File Finder can search for just the first occurrence of the file on the disk.

Another use for File Finder is to root out redundant files on a disk. Search for every occurrence of a given filename; if it shows up more than once, you may have identical copies of the same file which can be deleted to conserve space. This is especially useful on hard disk systems, where multiple copies of files tend to propagate.

File Finder can be run directly from the GEM desktop as a TTP (TOS Takes Parameters) program, or from the command line of *ST-Shell*. (See Chapter 4 for this disk operating system shell.)

Starting a Search

File Finder is called FIND.TTP on the disk. *If you plan to use File Finder from* ST-Shell, *rename the FIND.TTP file to FIND.PRG.* Don't rename the file if you plan to use it from the GEM desktop.

To run File Finder from the GEM desktop, open or double-click on FIND.TTP. A dialog box pops up with a dotted line. At

this point you should type the name of the file you're looking for, in this format, on the dotted line:

[options] [x:]filename

The command line syntax in *ST-Shell* is similar:

FIND *[options] [x:]filename*

The square brackets indicate optional arguments; do not type the brackets. There are two options, *−f* and *−t*, discussed below. The *x:* represents the optional disk drive identifier (from A: to P:), and *filename* is the name of the file you're searching for.

Each parameter on the command line must be separated by at least one space. It is illegal in this utility to supply a pathname, since the purpose of the program is to search all folders on the disk.

File Finder searches only one disk drive at a time, since you probably have a good idea which disk the file is on, but have forgotten which folder it occupies.

The complete filename must be given without any wildcard symbols (such as question marks or asterisks). This restriction stems from a problem within the GEMDOS function *Pexec( )*. When a new process is created and wildcard symbols are part of the command tail, *Pexec( )* tries to expand the command tail. This usually results in the new process not executing properly.

If the optional disk drive identifier is present in the command line, the disk in that drive is searched. If you don't include a drive identifier, the search operates on the current disk drive—the one from which you loaded File Finder.

The program prints a reminder of the proper syntax if you make a typing mistake or do not enter a filename. You can then try again.

Optional Parameters

You may include two options on the command line:

−f Search for the first occurrence of the file (default = off)
−t TTP program pause after execution (default = off)

When you include the *−f* option, File Finder searches for the first occurrence of the file. Once it finds the file, the program ends. Note that if you're searching a disk with a lot of nested folders, you can decrease the processing time by giving the *−f* option.

Whether you use this option or not depends on whether you want all occurrences of a file, or just the first. If you're sure that there isn't another file on the disk with the same name, you can use the *−f* option. On the other hand, if you want to check to see how many occurrences of a particular file there are on the disk, omit the *−f* option.

The option *−t* is used when you're executing File Finder as a TTP application from the GEM desktop. When you include *−t* on the command line, the program pauses and asks you to press a key before exiting back to the desktop. This allows you to examine the program's output, including any error messages that are generated. This option usually isn't needed when you're executing File Finder from *ST-Shell*, because the *ST-Shell* command prompt reappears immediately after the program's output—there is no return to the desktop.

Note that these two options are like switches that assume an off position by default. By putting them on the command line, you reverse their position. (You turn them on.)

The options may appear on the command line in any order, and you may include one or the other, or both. Remember, though, that all parameters on the command line must be separated by at least one space.

A Few Examples

Following are some examples of how to use File Finder.

TTP example: **−t TEST.C**
ST-Shell example: **FIND TEST.C**

File Finder searches for the file TEST.C on the current drive (because no drive identifier is present). File Finder lists all occurrences of TEST.C. The *−t* parameter in the TTP example makes sure the output will be visible when the program finishes.

TTP example: **−t −f D:CMP.PRG**
ST-Shell example: **FIND −f D:CMP.PRG**

File Finder searches drive D for the file CMP.PRG. Because the *−f* option is included, File Finder lists only the first occurrence of CMP.PRG on drive D.

File Finder's output is fairly straightforward, but let's examine a sample. First, here's the command line:

TTP example: **−t AUTOLOG.BAT**
ST-Shell example: **FIND AUTOLOG.BAT**

This searches for all occurrences of the file AUTOLOG.BAT on the current drive. Here is how the output from that search might appear:

FILE FINDER Version 1.0
Stand by....Searching for AUTOLOG.BAT
Found AUTOLOG.BAT
Path: Root
Path: \STSHELL.SYS
Path: \STSHELL.SYS \BACKUPS \BATFILES \AUTOBATS

In this example, File Finder discovered three occurrences of the file AUTOLOG.BAT on the disk (one for each Path: statement). The first copy of AUTOLOG.BAT is on the root level (the main directory), the second is inside the folder STSHELL.SYS, and the third is nested four levels below the root directory inside the folder AUTOBATS.

As the final example clearly suggests, if you tend to organize your most crowded disks with a lot of nested folders, you'll rate File Finder an especially valuable utility.

CHAPTER FOUR

Utilities

Encryptor

Douglas N. Wheeler

Do you have sensitive information you want to protect from prying eyes? This utility automatically encodes any type of disk file and locks it with a password of your own choosing. It works on a 520ST or 1040ST in any resolution mode: in low or medium resolution with a color monitor or in high resolution with a monochrome monitor.

Security is an important issue in today's world. In fact, everybody is becoming more security-conscious. Businesses have important documents and contracts which should not be viewed by just anybody hanging around the office. Lawyers, doctors, and other professionals have to guarantee their clients complete confidentiality. Even the average teenager may have letters or diaries which he or she would like to keep secret. If you've refrained from storing something with a computer for any of these reasons, this program is for you.

"Encryptor" is a utility which will encode and decode any ST disk file, whether it is your latest programming creation, a database file, a text file made with your favorite word processor, or even your mother's secret recipes. Once a file is encrypted, no one can decrypt it without entering the correct password or cracking the code.

Using Encryptor

Encryptor is very easy to use. Simply double-click on the ENCRYPT.TOS icon to run the program; then answer the two questions it asks you. But before you begin, there are a few things you should know. First, the file you wish to encrypt must be on the disk in the default drive, the same drive which contains ENCRYPT.TOS. Second, there must be enough room on the disk for a copy of the file. Third, the file must not be in a folder; it must be on the disk's root directory.

If the file is in a folder, the easiest way to move it to the root directory is to drag the file's icon to the drive icon. This makes a copy of the file outside of any folders. The original copy of the file remains in the folder, however, so you should delete it

if you want to make sure no unencrypted version of the file remains on the disk.

When you run Encryptor, the first question it asks is the name of the file you wish to encrypt. Simply type in the filename (wildcard symbols are not allowed) and press Return.

The second question Encryptor asks is the password you wish to choose. The password may be up to 40 characters long and may contain any displayable character except the space. Each file you encrypt can be secured with a different password if you like. But don't forget any of these passwords, because you'll need them to decrypt the files later. To decrypt a file, you simply enter the password at this same prompt after rerunning ENCRYPT.TOS. Either memorize the passwords or write them down and keep them in a safe place.

Because of the nature of the encryption scheme, you can superencrypt a file by encrypting it more than once using different passwords. To read such a file, you'd have to decrypt it as many times as you encrypted it using the same passwords in reverse order.

How It Works

Although it is a short program, Encryptor is quite complex. If you're interesting in learning how it works, read on; otherwise you already know everything you need to use Encryptor.

Encryptor begins by opening the file you selected for input and creating a temporary file named *qqqq* for output. The program then reads the input file, encrypts the data, and writes the results to the output file one character at a time. When it's finished, the program deletes the original file and changes the *qqqq* filename to the name of the original file. Most of these operations are pretty straightforward; obviously, the encrypting is where the action is.

As Encryptor reads each character, it matches that character with a corresponding character in the password. That is, the first character in the file is matched with the first character in the password; the second character in the file is matched with the second character in the password; and so on, repeating the password as many times as necessary to match every character in the file.

For example, if the file you're encrypting is a text file which begins *Now is the time for all good men,* and the password is *Help,* Encryptor matches up the characters as follows:

```
Now is the time for all good men
HelpHelpHelpHelpHelpHelpHelpHelp
```

Now picture, if you will, the complete ASCII character set as a continuous line of characters, repeating indefinitely in each direction. For each character to be encrypted, Encryptor effectively reverses the ASCII character set around the corresponding letter of the password. Using the first letter of our sample file and its corresponding password letter, we would have the following:

Original ASCII set:

```
... A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ...
```

New ASCII set:

```
... O N M L K J I H G F E D C B A @? > =< ; : 9 8 7 6 ...
```

As you can see, the ASCII character set has been reversed "around" the letter *H* (the first letter of the password). If you then look up the first letter of the file—*N*, on the top line—you will see that this corresponds to the letter *B*, which would be written to the output file. Encryptor would then continue to the next character in the file and reverse the ASCII set around the next letter of the password.

This process repeats until the complete file is encrypted and written to the output file. Encryptor then deletes the original file and changes the output filename from *qqqq* to the original filename.

Crash Analyzer

George Miller

Advanced programmers: If your ST crashes when running a new program you're writing, run this utility to find out why. It recovers special information preserved in memory by the 68000 microprocessor to give you a report on the cause of the failure. The utility works in all monochrome and color modes on any ST, but some of the information it displays is not visible in the low-resolution color mode.

Sooner or later it happens to everyone. As your carefully crafted program begins to run, bomb icons suddenly appear on the left side of the screen and you find that you've become another victim of a system crash.

But now you can clear away some of that smoke and find out *why* your program crashed the computer. "Crash Analyzer," a utility written in machine language, provides an explanation of those bomb symbols and helps you diagnose what caused your program to fail. It works even after you've rebooted the computer by pressing the reset button. And it's useful to programmers who are working in practically any language: 68000 machine language, C, Pascal, Forth, and even ST BASIC.

Taking Exception

Since computers do exactly what you tell them to do—not always what you want them to do—the system crash is a fact of life faced by every programmer. In the past, on the eight-bit computers, your only choice was to turn off the power and start over. The computer would refuse to respond to any of your attempts to get its attention.

The designers of the Motorola 68000 family of microprocessors felt the chips should be intelligent enough to recognize when they are heading for a system crash, and to take steps to recover from it, if possible. So the designers made that kind of help possible by including a feature called *exception processing*.

Basically, an *exception* is the ability of the chip to interrupt whatever it is doing, do something else, and then return to the

original task. Exceptions fall into two categories: those caused by external sources, such as input/output devices, and those caused by internal operations, like programming errors and TRAP instructions. An exception caused by an external source is called an *interrupt.*

Each 68000 exception is processed by a different routine. Pointers to these routines are stored in the first 1024 bytes in memory in a 68000 system. Each pointer, or *vector,* is stored in a long-word memory location (four bytes). There are 256 possible vectors, numbered from 0 to 255. Therefore, the address of a vector is the vector number multiplied by 4.

Exceptions generated by programming errors use certain vector numbers. Table 4-1 lists the more important preassigned exception vectors. Table 4-2 is a list of the vectors used by the ST.

Here's why the vector numbers are important. As the ST begins to crash, the 68000 senses that something is going wrong. Immediately it stores a copy of the values found in its address and data registers into an area of low memory. This area of memory usually survives the crash, and is not overwritten when you press the reset button on the rear of the ST to reboot. Then, to alert you that something has gone awry, the ST displays a number of bomb icons near the left side of the screen. The number of bombs corresponds to the vector number of the exception encountered. This information can point you to the right track when you are debugging your programs.

Exception Errors

Here's a brief explanation of the more commonly encountered types of exceptions. Refer to any complete guide on 68000 programming for a more detailed explanation.

Resets (exceptions 0 and 1). Vectors 0 and 1 are used when the computer is first powered up (or, on the ST, when the reset button is pressed) to set the initial stack and the program counter.

Bus error (exception 2). This indicates that your program tried to access a nonexistent area of memory.

Address error (exception 3). This may be caused by referencing a long word at an odd-numbered memory address. Long words consist of four bytes and must coincide with even addresses.

Illegal instruction (exception 4). Your program tried to execute an instruction which is not part of the 68000 instruction set. Check your source code for typing errors.

Table 4-1. Preassigned Vectors

Vector	Address	Function
0	$000	Reset initial supervisor stack pointer
1	$004	Reset initial program counter
2	$008	Bus error (nonexistent memory)
3	$00C	Address error
4	$010	Illegal instruction
5	$014	Division by zero
6	$018	CHK instruction
7	$01C	TRAPV instruction
8	$020	Privilege violation
9	$024	Trace
10	$028	Line A emulator
11	$02C	Line F emulator
12–14	$030–$038	Unassigned
15	$03C	Uninitialized interrupt vector
16–23	$040–$05C	Unassigned
24	$060	Spurious interrupt
25–31	$064–$07C	Level 0–7 autovector interrupts
32–47	$080–$0BF	TRAP 0–15 instruction vectors
48–63	$0C0–$0FC	Unassigned
64–255	$100–$3FF	User interrupt vectors

Table 4-2. Vectors Used by the ST

Vector	Function
10	Line A emulator
26	Level 2 interrupts
28	Level 4 interrupts
33	TRAP #1 GEMDOS
34	TRAP #2 GEM
45	TRAP #13 BIOS
46	TRAP #14 XBIOS

(All unused vectors are available to the programmer.)

Division by zero (exception 5). Since the 68000 instruction set includes division instructions, a check is made for the mathematically illegal operation of division by zero.

CHK instruction (exception 6). Caused by the CHK instruction.

TRAPV instruction (exception 7). Caused by the TRAPV instruction.

Privilege violation (exception 8). This happens when a privileged instruction is attempted while the 68000 is not in supervisor mode.

Trace (exception 9). Used by many debugger programs to single-step through a program that is being debugged.

Line A emulator (exception 10). A trap for opcodes using the format $A*xxx*.

Line F emulator (exception 11). A trap for opcodes using the format $F*xxx*.

Safe Deposit Boxes

In addition to reading the exception vectors, you can gather even more information about the conditions within the computer at the time of the crash. Before grinding to a halt, the 68000 saves the vital contents of its registers in some special memory locations which act as safe deposit boxes. After a crash which has been handled by an exception, it's possible to examine the information held in low memory (unless the ST is powered down).

The first place to look is at location $0380. If this address contains the magic number $12345678, then the information about the crash is good.

The data registers D0–D7 are saved beginning at location $0384.

The address registers, A0–A6, and the supervisor stack pointer, A7, are stored starting at $03A4.

The exception number is stored at $03C4 as a long word, and the user register is stored at $03C8. Beginning at $03CC is a list of 16 words saved from where the supervisor stack was pointing.

As you can see, when the 68000 crashes, it tries to help you by putting information where you can find it. However, unless you're using a monitor or a debugger, it's difficult to examine all of these locations after a crash. That's when Crash Analyzer comes to the rescue.

Run the Analyzer (disk filename: ANALYZER.TOS) immediately after rebooting the computer with the reset button. It informs you which exception has been triggered and lists the contents of the data registers (D0–D7) and address registers (A0–A7). It shows you a list of flags which were set at the instant of the crash and displays the starting addresses of both the

supervisor and the user stacks. Finally, it shows the contents of the supervisor stack at the time of the crash.

In effect, Crash Analyzer takes a snapshot of the activity in your ST at the moment of the crash, then lets you examine it at your leisure.

This information—plus a complete guide on 68000 programming—will help you debug your program to correct the condition that triggered the crash.

Word Count

A Writer's Accessory

Tony Roberts

This compact and efficient desk accessory quickly counts the number of words in a text file. It's a useful tool for writers and can be summoned in a flash from within any word processor. It works on any 520ST or 1040ST in any resolution mode: in low or medium resolution with a color monitor or in high resolution with a monochrome monitor.

One of a writer's traditional guideposts is the word count. It measures both what has been accomplished as well as what remains to be done. Writing assignments take shape when an editor specifies how many words are expected, and a writer working past deadline often relies on word-count references to keep the anxious editor at bay.

Although most computers, the ST among them, can report the number of bytes in a file, writers and editors rarely bandy about byte counts. There are various mathematical methods for converting number of bytes to number of words, but they're rather haphazard. More precise counting methods are often painfully slow.

"Word Count" is a desktop accessory program that solves these problems. It provides a fast, convenient word count for all types of text files—including documents in *ST Writer* and *1ST Word* formats as well as plain ASCII files.

Using Word Count

You'll find Word Count on the disk under the filename WRDCOUNT.AC. Don't try to run it from the disk menu. Instead, copy the file to one of your own boot disks and rename it WRDCOUNT.ACC. The .ACC extender tells the ST's operating system to install the program as a desk accessory when the computer is switched on. (To install any desk accessory, you have to

"Word Count" was written using *Personal Pascal* from Optimized Systems Software. Portions of this program (the linked libraries) are copyright 1986 by OSS and CCD. Used by permission of OSS.

boot up the computer by turning on the power; merely pressing the reset button doesn't do it.) Once the computer has been started, it's not necessary to keep the WRDCOUNT.ACC disk in the drive.

After Word Count is installed, it's instantly available from the Desk menu on the GEM desktop. To activate it, simply drop down the Desk menu on the desktop or from within an application program (such as your word processor) and select it.

Once activated, Word Count provides a standard GEM item-selector dialog box which lists the files on your disk. The dialog box displays the files in the current folder, but, as usual, you may switch to other folders to locate the desired file. This is accomplished by clicking on the directory line, editing the line to reflect the desired pathname, then moving the pointer down into the file area and clicking again to inform GEM that you want to view another directory.

Upon receiving a valid filename from the dialog box, Word Count reads through the specified file byte by byte, looking for space characters, tab characters, and line-ending characters, which it assumes are word delimiters. When it finds a delimiter, Word Count increments the counter and continues the search unless the previous character also was a word delimiter, in which case the counter is not incremented.

When the count is complete, another dialog box opens and displays the complete pathname of the file selected and the results of the count. After you've clicked on OK or pressed Return, Word Count retreats back to the desktop, and you can pick up where you left off.

It Has To Be Fast

Word Count works rapidly. The time it takes depends on the length of the text file, of course, but normally it finishes the job in only a few seconds. Because it's a memory-resident desk accessory and because it sets aside a large buffer for reading the text file, time-consuming disk accesses are kept to a minimum. Another advantage of Word Count as a desk accessory is that it's available whenever the Desk menu is displayed—while you are using your word processor, programming in BASIC, or even using a telecommunications program that supports GEM.

One shortcoming of Word Count is that it cannot analyze the document currently in memory until you've stored it on disk. But aside from that, you can ask it to count any disk file, includ-

ing program (.PRG or .TOS) files—but with these, of course, the results are rather meaningless. It's also possible to count BASIC program files and Pascal or C source files, but because most source code is inconsistently spaced, the results are less than accurate.

Word Count ignores the format lines that are stored at the beginning of *1ST Word* and *ST Writer* document files. If you're using another word processor, you might have to experiment to see what effect (if any) its format lines have on the word count.

If you own an early Atari 520ST that has not yet been upgraded with the TOS operating system in ROM, you may have to rename WRDCOUNT.ACC to DESK5.ACC for it to work properly.

Word Count works with text files stored on floppy disks, hard disks, and RAM disks; but a caution is in order if you're using a single-drive floppy system. Because of an operating system bug, the computer may crash if you attempt to switch to Disk B from the file-selector dialog to access a text file on another disk. To circumvent this problem, close the Disk A window, remove the disk from the drive, and insert the disk containing your text file before activating Word Count from the Desk menu.

ST-Shell™

Richard Smereka

Here's a major new feature for your Atari ST—a program that provides disk operating system commands and batch file capabilities. Using more than 30 UNIX-like commands, you can run programs, create and delete files and folders, print screen messages, set the system date and time, change screen colors, customize the cursor, check free memory, set up autoboot sequences, and do much more. The program works on all STs in medium-resolution color and high-resolution monochrome modes.

The Atari ST relies heavily on the desktop metaphor provided by GEM, the Graphics Environment Manager. Instead of typing in cryptic disk operating system commands, you deal with icons, windows, and drop-down menus. For instance, you never have to use a DIR or CATALOG command to find out what's on a disk; instead, you open or double-click the disk icon. You don't type LOAD or RUN to execute programs; you open or double-click the program icon. You don't manipulate files by typing COPY, ERASE, or DELETE; you drag icons from window to window or to the trash can.

Although designed for convenience, the desktop-style interface can sometimes become a minor nuisance, especially when you perform a certain series of actions every time you turn on the computer. For instance, you might want to start each session by setting the system clock, running a RAM disk utility, and copying certain files from drive A to the RAM disk. What's needed in a case like this is an old-fashioned command line DOS that supports batch files.

ST-Shell™ is the answer. Like GEM, it's a program that wedges itself as a shell between you and the computer's underlying operating system. But unlike GEM, it's not a graphics-oriented desktop environment. Instead, it's a command line interpreter similar to MS-DOS, CP/M, DOS XL, and the Amiga CLI. Actually, most of the commands are patterned after those found in UNIX, a popular operating system on minicomputers and powerful micros. With *ST-Shell*, you enter commands at a

DOS prompt to manipulate files, run programs, pass arguments, and execute batch files. You can even set up your system to automatically run a batch file when the computer is first switched on.

Almost any program can be executed from *ST-Shell.* If the program requires arguments such as filenames or additional commands, you can add them to the command line and they'll be passed along. In addition, *ST-Shell* allows batch files of just about any size.

ST-Shell is set up for an 80-column screen, as found in the medium- and high-resolution modes. It's possible to run *ST-Shell* in the lo-res mode, but characters past column 40 will not appear on the screen.

The instructions for using *ST-Shell* can get somewhat involved, and for good reason: *ST-Shell* is much more than the average utility program. It's a fairly complete, yet compact, disk operating system shell. In all, there are 33 commands for managing disk files and setting up your system, and many of these commands have several variations. If you've never used a command-oriented DOS before, it will take some time to fully master this environment.

In addition, keep in mind that the complexity of command-line interpreters like *ST-Shell* is exactly why Atari chose to equip the ST with GEM. Many of the functions provided by *ST-Shell* can be performed more easily with GEM, but some can't be performed at all—such as the batch file processing. *ST-Shell* offers flexibility in return for its complexity.

Preparing *ST-Shell*

ST-Shell is labeled STSHELL.TOS on the disk. You can run it directly from the disk (either from the menu program or the GEM desktop), but we recommend copying it to another disk and saving the original as a backup. When you copy STSHELL.TOS to another disk, also copy the file called HELP.BAT. (We'll explain why later.)

Note that *ST-Shell* is a TOS (Tramiel Operating System) application, as indicated by the .TOS filename extender. TOS applications do not support GEM features such as windows, drop-down menus, and the mouse controller. When you run a TOS application, the screen clears, the mouse pointer disappears, and a text cursor is enabled.

In some cases, you may want to rename *ST-Shell* from STSHELL.TOS to STSHELL.PRG to disable it as a TOS application. *ST-Shell* still won't support GEM features, but it will behave differently in some respects. For instance, if you want *ST-Shell* to run automatically when you turn on the computer, *you must rename it STSHELL.PRG and place it in a folder named AUTO.* The ST checks for an AUTO folder during bootup and runs any non-GEM programs in the folder that have the .PRG filename extension. The programs are executed in the order in which they were placed in the folder. (GEM programs can't be started from the AUTO folder because GEM is not initialized at this stage of the boot-up procedure.)

If you place *ST-Shell* in the AUTO folder as STSHELL.PRG, you'll notice that the ST wakes up in the low-resolution screen mode if you're using a color monitor. That's because the ST boots up in lo res by default unless you Set Preferences for medium res and then select Save Desktop. Even if you've done this, however, the ST still boots up in lo res when running *ST-Shell* from the AUTO folder. Why? Because the DESKTOP.INF file which saves your preferences is not loaded until after all programs in the AUTO folder have finished executing. This is an idiosyncracy of the ST that you'll have to get used to when booting *ST-Shell* from the AUTO folder.

You'll also notice that the text cursor does not appear when *ST-Shell* is booted from the AUTO folder. This is normal; you can turn on the cursor with *ST-Shell*'s CURON command.

It's important to realize that since GEM is not initialized when the ST is running programs found in the AUTO folder, *you can't run a GEM application from* ST-Shell *if STSHELL.PRG has been automatically started from AUTO.* This means that auto-booting batch files cannot run GEM applications—a limitation of the ST, not of *ST-Shell.*

Running GEM Programs

It is possible, however, to run GEM applications from *ST-Shell* if STSHELL.PRG has *not* been started from the AUTO folder. If you run STSHELL.PRG from the desktop, you can launch a GEM application simply by typing its filename at the *ST-Shell* prompt. For example, suppose you want to run *1ST Word* and load a text file called DIARY.DOC. With STSHELL.PRG active, you can type this command:

1ST_WORD DIARY.DOC

or include this line in a batch file. The computer runs *1ST Word* and loads DIARY.DOC. (This demonstrates *ST-Shell's* ability to pass arguments—in this case a filename—to an application program.)

If you try to run a GEM application such as *1ST Word* from *ST-Shell* when it is named STSHELL.TOS (a TOS application), the GEM program comes up on the screen, but lacks a mouse pointer. You'll probably have to reboot to regain control. When you run the GEM program from *ST-Shell* when it is named STSHELL.PRG, the mouse pointer and other GEM features remain available.

You may notice some odd effects when running GEM programs from *ST-Shell,* however. For instance, the text cursor may remain on the screen, superimposed on the GEM application. Although the cursor usually causes no harm, you can prevent this by turning off the cursor with *ST-Shell's* CUROFF command before running a GEM program.

Similarly, when you quit the GEM application and exit back to *ST-Shell,* remnants of the GEM screen and mouse cursor may remain on the *ST-Shell* screen. You can clean this up by entering *ST-Shell's* CLS command.

The *ST-Shell* Screen

When *ST-Shell* first runs, you'll be greeted with a sign-on message, a text cursor, and a command line prompt that looks like this:

A:

This prompt indicates that the current disk drive is drive A. In other words, all disk commands entered at this prompt will affect drive A. *ST-Shell* is now waiting for you to type in a command.

(Note: If you run *ST-Shell* from a drive other than drive A, that drive identifier appears as the default prompt.)

All *ST-Shell* commands can be typed in upper- or lowercase characters. You can even mix upper- and lowercase, since *ST-Shell* converts all input to uppercase before interpreting the command.

An *ST-Shell* command or a program name *must* appear as the first argument on any new command line. Any following arguments (such as the *1ST Word* document filename shown above)

must be separated from the command by one or more spaces. When entering commands at the keyboard, you are limited to one screen line. Commands in a batch file, however, may be longer. (We'll cover batch files in detail below.)

Running Other Programs

To run a program from *ST-Shell*, all you have to do is type the name of the program followed by any arguments that may be optional or required. For example, to run a text editor program named ED.PRG, type

ED *filename*

where *filename* is the name of the text file you wish to edit.

When running a program from *ST-Shell*, do not type in the filename extension (the period and the three characters which follow it). *ST-Shell* automatically searches for files with these extensions:

.PRG (application programs)
.TOS (TOS applications)
.BAT (*ST-Shell* batch files)

ST-Shell searches for the filenames in that order. In other words, if a .PRG file and a .BAT file happen to have the same name (such as 1ST_WORD.PRG and 1ST_WORD.BAT), the .PRG file is found and executed first and the .BAT file is ignored. Therefore, to avoid conflicts, you should make sure your files have unique filenames.

Also, since *ST-Shell* first attempts to interpret anything on the command line as a command, do not use filenames that are the same as *ST-Shell* commands. The accompanying table shows the full command set.

Table 4-3. *ST-Shell* Command Set

Command	Function
Disk commands	
***x*:**	Change prompt to the specified drive (A–P)
DF	Display free space on disk in the current drive
LS	List (display) the disk directory
GD	Get directory: display current folder name
CD	Change directory (current folder)
MKDIR	Make (create) a folder
RMDIR	Remove (delete) a folder
CPDIR	Copy a folder
CP	Copy a file
RM	Remove (delete) a file
MV	Move (rename) a file
Screen commands	
HOME	Home the cursor
CLS	Clear the screen and home the cursor
PLOT	Place the cursor at a designated location
CURON	Turn on the text cursor
CUROFF	Turn off the text cursor
CURFLASH	Turn on flashing text cursor
CURSOLID	Turn on solid (nonflashing) text cursor
CURATE	Change cursor flash rate
TEXT	Change the screen text color
BGROUND	Change the screen background color
RVSON	Turn on reverse video
RVSOFF	Turn off reverse video
ECHO	Print text or blank line on the screen
WRAPON	Wrap text to next screen line if line overflows
WRAPOFF	Do not wrap overflow lines; ignore overflow text
Miscellaneous commands	
MF	Display the amount of free memory (RAM)
BEEP	Beep the monitor speaker
TIME	Get or set current system time
DATE	Get or set current system date
DOC	*ST-Shell* remark statement
BYE	Exit *ST-Shell* and return to GEM desktop
EXIT	Exit *ST-Shell* from a batch file and return to GEM desktop

Command Syntax

In the following sections, we'll list *ST-Shell* commands in uppercase type and required parameters in lowercase italics. Optional parameters are listed in lowercase italics enclosed in brackets. *Do not type the brackets.* For example,

RMDIR ***[x:] [\path\] foldername\ [−P]***

means the command RMDIR (Remove Directory) requires one parameter, the name of a folder to remove. Optional parameters include a drive identifier (*x:*), a pathname (*\path*), and the characters *−P* (which prevent *ST-Shell* from warning you if the folder to be deleted contains any files). Remember that all *ST-Shell* commands and parameters must be separated by spaces.

So, to delete a folder named LETTERS on drive B, and to prevent *ST-Shell* from alerting you if the folder contains any files, you enter

RMDIR B: LETTERS \ −P

If the folder LETTERS is contained within another folder on drive B called TXTFILES, you enter

RMDIR B: \TXTFILES \ LETTERS \ −P

When you're first learning *ST-Shell*, it's a good idea to experiment with the commands on a scratch disk before using a disk with important files.

Disk Commands

x:

This changes the current disk drive and therefore the *ST-Shell* prompt which appears on the screen. Normally, *ST-Shell* defaults to the drive from which it was run, and the screen prompt denotes this drive. You can change this to any drive you want—the *x* parameter stands for any drive identifier in the range A through P. On a single-drive system, your drive is always A. The second floppy drive is always B. If you have a RAM disk installed, or a hard drive, you may use the letters C–P. For this command to work properly, there can be nothing else on the command line or in a batch-file line.

DF

Display Free disk space. This command displays the total amount of disk space available and the amount of space remaining on the disk in the current drive.

LS *[x:] [\path\] [*.*]*

List directory. This command lists to the screen the directory of a disk or folder. The default drive is listed unless you provide a drive identifier (*x*:). Note that the wildcard symbols (*.*) are optional only for listing a directory for the current drive. If you are listing a directory for another drive or folder, you must include the wildcard symbols. Of course, you can change the wildcard symbols to specify that only certain files should be listed. For example, LS B: *.C lists to the screen only those files ending with the extension .C on the disk in drive B.

The optional pathname (*path*\) lets you list the contents of a folder nested to any level. *ST-Shell* has a simple rule regarding folder pathnames. When using the commands MKDIR, RMDIR, and CPDIR, you may omit the backslashes in the pathname if the folder is not nested below the root level. For example, the command RMDIR TEST is a valid command as long as the folder TEST is not nested below the root directory. If you are in doubt, use the backslashes.

GD

Get Directory. This command displays the current folder (directory) or indicates that the current directory is the root directory.

CD *[\foldername\]*

Change Directory. This changes the current directory to the folder specified. If no folder is specified, CD defaults to the root directory.

MKDIR *[x:] [\path\] foldername*

Make Directory. This creates a folder on the default drive, or on the drive indicated (*x*:). *ST-Shell* displays an error message if the folder cannot be created (for example, if a folder of that name already exists).

RMDIR *[x:] [\path\] foldername\ [-P]*

Remove Directory. This removes a folder from the default drive or from the drive specified (*x*:). The folder may contain files and other empty folders. You cannot delete a folder if it contains

other nested folders with files in them. If the folder contains files, *ST-Shell* asks if you're sure you want to delete them. This safety measure may be prevented by using the option *−P.*

CPDIR *[x:] [\path\] olddir\ [x:] [\path\] newdir*

Copy Directory. This command creates a new folder and copies into it the entire contents of an existing folder. The *olddir* parameter specifies the old directory (existing folder) name. The *newdir* parameter specifies the new folder. The new folder cannot already exist.

CP *[x:] [\path\] oldname [x:] [\path\] newname*

Copy a file. Both drives (if they are different) must be separate physical drives or RAM disks. On a single-drive system, you cannot copy a file from drive A to drive B because they are the same physical drive.

RM *[x:] [\path\] filename*

Remove file. This deletes the file specified by *filename* from the current folder in the current drive, or from the drive and/or folder specified by *x:* and *\path*.

MV *[x:] [\path\] oldname [x:] [\path\] newname*

Move (rename) a file. The *oldname* (and optional parameters) refers to the existing filename. The *newname* refers to the new filename. The new filename cannot already exist; if it does, *ST-Shell* reports an error. Files can be moved between folders, but not between disk drives, with this command.

Screen Commands

HOME

Home the cursor. This command places the text cursor in the home position (the upper left corner) without clearing the screen.

CLS

Clear Screen. This command both clears the screen and homes the cursor.

PLOT *row column*

Plot cursor at specified position. With this command, you can move the cursor to any screen position indicated by *row* and *column*. In the medium- and high-resolution screen modes, *row* can

range from 0 to 24, and *column* can range from 0 to 79. Plotting outside these ranges results in an error. PLOT is useful when you are formatting the screen with ECHO commands in a batch file.

CURON *[-B]*

Cursor On. This command makes the text cursor visible if it isn't on the screen. Normally, when processing batch files, *ST-Shell* automatically turns off the cursor. You can suppress this feature by specifying the *-B* option.

CUROFF *[-B]*

Cursor Off. This command makes the text cursor invisible. This is handy for formatting the screen. The *-B* option suppresses the cursor only during batch file processing (the default).

CURFLASH

Cursor Flash. This enables a blinking text cursor, the default.

CURSOLID

Cursor Solid. This stops the text cursor from blinking, displaying it as a solid block.

CURATE *rate*
CURATE −N

Cursor Rate. Both of these commands change the rate at which the cursor flashes. Higher values for *rate* make the cursor blink slower; lower values make it blink faster. The useful range for *rate* is 10–50. If you specify a *rate* of 1, the cursor blinks so fast that it appears to be gray. The command CURATE −N restores the flash rate to *ST-Shell*'s default value, which is approximately equal to CURATE 30. The CURATE command has no effect if CURSOLID has been executed.

TEXT *color*

Change text color. In hi res, *color* can be 0 or 1; in medium res, *color* can range from 0 to 3.

BGROUND *color*

Change background color. Allowable values for *color* are the same as for the TEXT command.

RVSON

Reverse On. This causes all subsequent ECHO commands to print in reverse video. If you follow RVSON with CLS, the entire screen is reversed.

RVSOFF

Reverse Off. This disables reverse-video printing.

ECHO ***[text line to be printed]***

Print text on the screen. This command is generally used in batch files. If no text is specified, ECHO prints a blank line. ECHO is an exception to the rule that *ST-Shell* converts all text to uppercase; when processing batch files, *ST-Shell* prints the text in an ECHO statement as is.

WRAPON

Wrap On. If a line of text overflows a screen line, this command allows the leftover text to be printed on the next screen line.

WRAPOFF

Wrap Off. This command truncates all text after printing on screen-column 80. This is the default when *ST-Shell* is initialized.

Miscellaneous Commands

MF

Memory Free. This displays the number of available bytes of random access memory (RAM) in the ST.

BEEP

Beep the monitor speaker. This is useful in batch files when you want to grab the user's attention.

TIME ***[hh:mm:ss]***
TIME P

Get or set system time. If the TIME command is followed by no arguments, *ST-Shell* displays the current system time. To set a new time, follow TIME with the new setting. You can omit leading zeros (for example, 9:15:00 does not have to be entered as 09:15:00). The TIME P command causes *ST-Shell* to prompt the user to enter the time; this is useful in batch files.

DATE *[mm/dd/yy]*
DATE P

Get or set system date. The same basic rules that apply to the TIME command also apply here: You can enter 8/6/86 instead of 08/06/86 or 08/06/1986. The DATE P command prompts the user to enter the date; again, this is intended for batch files.

DOC *[comment text]*

Document remark. This is like a REM statement in BASIC; it lets you add comments to batch files. *ST-Shell* ignores any text that follows DOC.

BYE

Exit *ST-Shell* to the GEM desktop; this command may not be used in a batch file.

EXIT

Exit *ST-Shell* to the GEM desktop from within a batch file. This command should be on the last line of the batch file.

Batch Files

If you're new to a command-oriented DOS, you may be unfamiliar with the advantages of batch files. A batch file is just a series of DOS commands that are executed in sequence by the computer. The computer reads the commands in the batch file and carries them out, one by one, just as if you have typed them on the keyboard. In effect, a batch file is a program written in the language of DOS commands.

Batch files are extremely useful for automating frequently executed tasks, which is why they were incorporated into the design of *ST-Shell.* And like any good DOS, *ST-Shell* is capable of automatically reading and executing a batch file when the computer is first powered up. (We'll cover this in a moment.)

An *ST-Shell* batch file is identified by the file extension .BAT. A batch file may consist of *ST-Shell* commands and can even run other programs. Consider this example:

```
ECHO ** This is a batch file test **
MKDIR B:USR
CP TEST.C B: \USR \ TEST.C
NUMBER
```

To create this batch file, you enter these lines exactly as they appear above with a text editor or word processor that can save straight ASCII files (such as *1ST Word* with its WP Mode switched off). Suppose you saved this text file on disk with the filename BATCH.BAT. To run the batch file, you'd type BATCH at the *ST-Shell* prompt (remember that you don't have to type the filename extension *.BAT*).

When BATCH.BAT runs, *ST-Shell* executes the four commands in the sequence shown above. The first command ECHOes a simple message on the screen. The second command, MKDIR, creates a folder on drive B called USR. The third command, CP, copies the file TEST.C from the current drive (usually drive A) into the folder USR on drive B. The last line in the batch file tells *ST-Shell* to run the program named NUMBER on the default drive. The program NUMBER can have the file extension .PRG or .TOS.

NUMBER cannot be another batch file, however. A batch file cannot load and run another batch file.

Passing Parameters

ST-Shell lets you pass parameters from the command line to a batch file. For example, you could type this on the command line:

```
TEST HELP.DOC
```

TEST could be a batch file which contained this single line:

```
PRINT %1
```

This would mean you were executing the program PRINT from within the batch file TEST, and you were passing the first (%1) argument found on the *ST-Shell* command line to PRINT. The specifier %1 replaces the characters found in the batch file with the *n*th command line argument. Command line arguments start from zero, which is the actual program name and cannot be used in a batch file.

Unlike *ST-Shell* commands entered at the keyboard, a batch file command can be extended past one screen line. The @ character is called the *line-continuation character.* When placed at the end of a batch line, it tells *ST-Shell* that the next batch line should be treated as a continuation of the first line. You may not continue a line past the second line, however.

This capability is useful if you're programming in a language

like C that requires lengthy commands for compiling or linking. Suppose you are linking a program like this:

```
link68 [u,tem[d:]] %1.68k=gemstart,d:%1,osbind,gemlib,libf,@
  vdibind,aesbind
```

Normally, it would be difficult to get this entire command on one line. But when you're using the line-continuation character, the command does not have to be squeezed on one line. Note where the continuation character is placed—between one subcommand and another. Do not chop a command in the middle.

Also, since *ST-Shell* ignores all leading spaces, notice that you can indent the second line.

Special Notes

Multiple commands on a single line are not permitted in batch files; each line in a batch file may contain only one command or program name.

If you want a batch file to terminate *ST-Shell* and quit to the GEM desktop, keep in mind that you must use the EXIT command, not BYE. The BYE command is valid only when typed on a command line.

Before exiting *ST-Shell*, enter the CUROFF command; this will prevent the cursor from remaining visible when you return to the desktop.

One batch file has a special meaning to *ST-Shell*. When *ST-Shell* initializes, it looks for a batch file called AUTOLOG.BAT. If this file is present, *ST-Shell* automatically executes the file. This lets you create an autoboot sequence that prompts you for the date and time, customizes the screen colors and text cursor, prints a welcome message, or whatever you want.

You'll find two examples of AUTOLOG.BAT files on the disk; a short one is on the root directory, and a longer one is in the folder named STSHELL. The longer AUTOLOG.BAT demonstrates a typical startup sequence that queries for the date and time. This AUTOLOG.BAT is designed to be executed when *ST-Shell* is booted from an AUTO folder. For this purpose, it must be on the root directory. *You must copy it to the root directory of your boot disk, then, along with STSHELL.PRG in an AUTO folder, to see how it works.*

If you're not running *ST-Shell* from an AUTO folder, *ST-Shell* looks for the AUTOLOG.BAT file in the current directory, not in the root directory. That's why the shorter AUTOLOG.BAT

file is executed when you run *ST-Shell* from the disk.

The disk also contains the *ST-Shell* batch file HELP.BAT. This is a help screen for *ST-Shell*. Simply type HELP at the *ST-Shell* prompt to load and run this batch file; it consists of ECHO statements that list the complete set of *ST-Shell* commands. HELP.BAT is also a good example of screen formatting with ECHO. To have this help screen available, you must copy HELP.BAT to any disk on which you copy *ST-Shell*.

Snapshot *NEO/DEGAS*

Philip I. Nelson

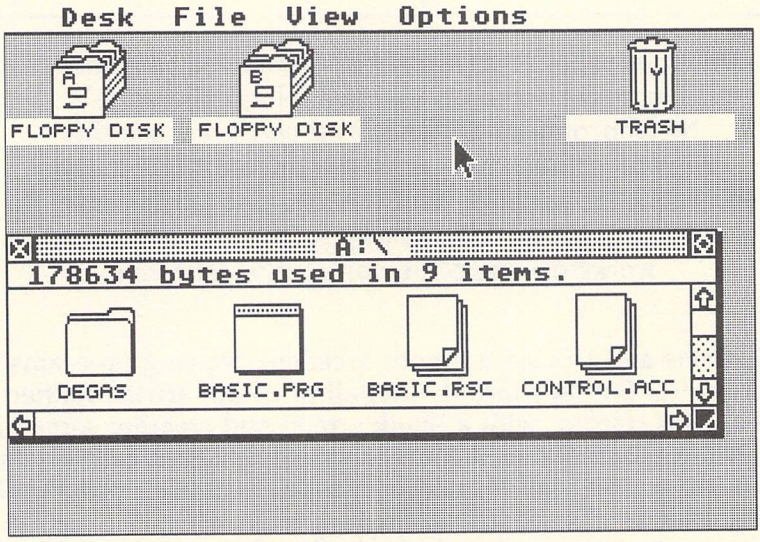
Try "Snapshot NEO/DEGAS" whenever you want to capture a screen image for later use with NEOchrome or DEGAS. You can even use this convenient desk accessory to convert NEOchrome and DEGAS pictures from one format to the other. It adjusts automatically to any screen resolution and works on any ST, color or monochrome.

NEOchrome and *DEGAS* are both excellent drawing programs for the Atari ST. But let's face it—not all of us are artists. Rather than always starting with a blank screen and creating something from scratch, sometimes it's easier to simply capture an existing screen image and load it into your favorite drawing program for modifications.

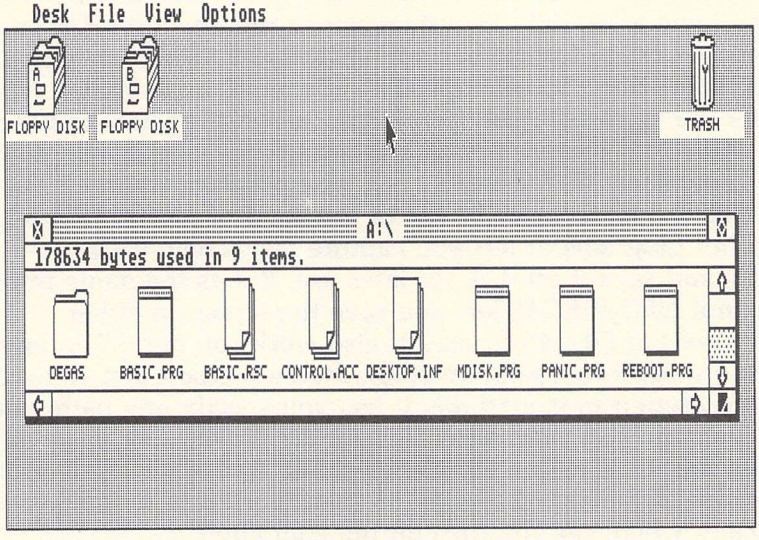
A number of so-called "snapshot" utilities are available both commercially and in the public domain for this purpose. But among those utilities, it's difficult to find the right combination of features. Some work only in certain screen modes; some can save a screen in *DEGAS* format but not *NEOchrome* format, or vice versa; some don't let you specify a filename or pathname when saving the screen on disk; some require *two* programs to function—one to capture a screen and another to save it; and so on.

"Snapshot *NEO/DEGAS*" is a new program written to include all the features you need in this kind of utility. Like other snapshot programs, it lets you capture any screen image instantly and save it on disk for later use. But as the name implies, Snapshot *NEO/DEGAS* lets you save the screen in either *NEOchrome* or *DEGAS* format. It also works on any ST in any screen mode: low-resolution color, medium-resolution color, and high-resolution monochrome. It lets you specify any pathname and filename you want when saving the file on disk. It's available at any time with a simple keypress. And since it's a desk accessory, Snapshot *NEO/DEGAS* installs itself in memory automatically whenever you turn on the computer.

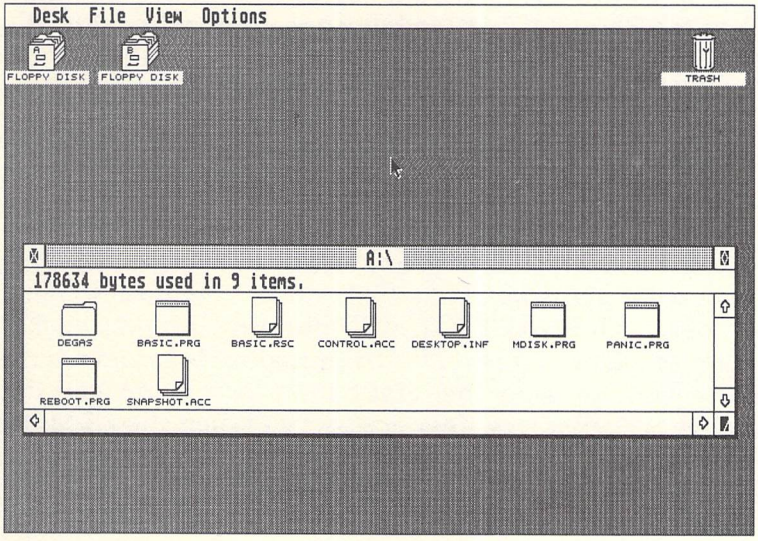
Figure 4-1. "Snapshot *NEO/DEGAS*"—Available in Any Resolution



Lo-res screen



Medium-res screen



Hi-res screen

You can use Snapshot *NEO/DEGAS* to capture graphics screens, GEM screens, high-score game screens, or just about anything. As a bonus, it also lets you convert *NEOchrome* pictures to *DEGAS* format, or vice versa.

Installing Snapshot

You'll find Snapshot *NEO/DEGAS* on the disk under the filename SNAPSHOT.AC. It cannot be run from the disk menu program or the GEM desktop; it must be installed as a desk accessory. (A *desk accessory* is a program that automatically loads into memory when you first switch on your computer, and then idly waits there until called from the Desk menu at the far left of the menu title bar. The Control Panel which comes with every ST is an example of a desk accessory.)

To use Snapshot *NEO/DEGAS*, you must first install it as a desk accessory on your startup disk (the disk you insert in drive A when switching on the computer) by renaming it SNAPSHOT.ACC. If you aren't sure how to do this, follow these steps:

1. Copy the file SNAPSHOT.AC from *COMPUTE!'s Second Book of Atari ST Disk* to your startup disk.
2. Display a directory window for your startup disk.
3. Select the file SNAPSHOT.AC by clicking once on its icon or filename to highlight it.

4. Drop down the File menu and select the Show Info option.
5. When the Show Info dialog box appears, a cursor should appear on the filename line. Change the name of the file to SNAPSHOT.ACC.
6. Exit the Show Info box by pressing Return or clicking the mouse on the OK button. The directory window for your startup disk should confirm that the file is now named SNAPSHOT.ACC.

Snapshot *NEO/DEGAS* is now ready to be installed. Turn off your computer and wait about 15 seconds. Then insert your startup disk with SNAPSHOT.ACC into drive A and switch the computer on again. (This *cold start* procedure is recommended because merely pressing the reset button for a warm start does not reliably install a new desk accessory.) When the GEM desktop appears, drop down the Desk menu. You should see a new selection entitled SNAPSHOT NEO/DEGAS.

Snapping Pictures

As a desk accessory, Snapshot *NEO/DEGAS* hides in memory until you need it, even when you're using applications such as ST BASIC or *1ST Word*. To snap a picture of the current screen, just press Alt-Help. (Hold down the Alt key; then tap the Help key.) Snapshot *NEO/DEGAS* briefly inverts the screen colors to signal that it has stored a complete image of the screen, including the resolution, color palette, and current position and appearance of the mouse pointer.

At this point, the screen is captured in memory, but is not yet saved on disk. If you press Alt-Help again, the captured image will be replaced by a new screen. Unlike most cameras, Snapshot *NEO/DEGAS* can take only one picture on its "film." If you need to capture more than one screen, you must save each image on disk before pressing Alt-Help again.

To save a captured screen on disk, start by dropping down the Desk menu. (Some application programs, such as *1ST Word*, title the Desk menu with the Atari logo symbol, but it still works the same.) Then select SNAPSHOT NEO/DEGAS to activate the desk accessory. A dialog box appears and prompts you to choose the desired format: *NEO* or *DEGAS*. If you decide not to save the screen, you can click on CANCEL. (If you select SNAPSHOT NEO/DEGAS without a captured screen in memory, the accessory informs you with an alert box.)

Since the current version of *NEOchrome* loads only low-resolution files, Snapshot *NEO/DEGAS* warns you if you choose the *NEOchrome* file option when in medium or high resolution. To cancel the save, simply select the CANCEL box. However, Snapshot *NEO/DEGAS* is designed for upward compatibility, so it allows you to save a *NEOchrome*-format image in medium or high resolution in case *NEOchrome* is ever updated to handle those screen formats as well.

After you've selected the file type, Snapshot *NEO/DEGAS* opens a standard GEM file selector box which allows you to choose a filename for the stored screen image. Choose the drive-path and filename you wish to use, just as you would from BASIC or any other ST application. Again, if you choose CANCEL, or if you select OK without entering a filename, Snapshot *NEO/DEGAS* aborts the operation without saving anything to disk. If you select a filename that already exists, Snapshot *NEO/DEGAS* gives you the option to replace the existing file or cancel.

The program also checks to make sure the disk contains enough free space to hold the new file; if there's not enough room, Snapshot *NEO/DEGAS* displays an alert box and aborts without altering the disk. (Keep in mind that a picture file in any resolution in either format requires about 32K.)

Naming Picture Files

When saving pictures, you are responsible for entering a filename with the correct three-character extension for the desired picture format. Every *NEOchrome* filename must end with the .NEO extension. *DEGAS* filenames end with the extension .PI followed by a 1, 2, or 3 to indicate the screen resolution: Use .PI1 for low resolution, .PI2 for medium resolution, and .PI3 for high resolution.

If you're not sure about the extension, look at the path specification in the upper portion of the file selector box. As a convenience, Snapshot *NEO/DEGAS* supplies the correct extension for the format and resolution which you select. The filename extension does not affect the contents of the file; if you accidentally save a picture with the wrong extension, simply rename it with Show Info from the desktop.

Once you've selected a filename, Snapshot *NEO/DEGAS* saves the complete screen image on disk in the desired format, including the screen resolution and color palette which were in effect when you captured the screen. The resulting file can be

loaded into *NEOchrome* or *DEGAS* and manipulated like any other picture file.

As a desk accessory, Snapshot *NEO/DEGAS* is normally available from within any GEM application. However, it's possible for an application to change what's available in the Desk menu. Some programs replace existing menus with menus of their own (or make all accessories unavailable, as in the case of *NEOchrome*), but restore them when you exit the application. If you have previously installed Snapshot *NEO/DEGAS,* it should work even when you're using such a program. Press Alt-Help to store a screen image while the application is running; the screen should blink as usual to signal that the image is captured. After you've exited the application, Snapshot *NEO/DEGAS* should reappear in the Desk menu. At this point, you can save the captured image to disk. The process of returning to the desktop does not disturb the captured image.

If Snapshot *NEO/DEGAS* does not reappear in the Desk menu when you return to the desktop, it has been forcibly removed by the application and cannot be used. It's considered bad GEM etiquette for an application to remove a desk accessory without replacing it, but you should be aware of the possibility.

Additional Notes

Snapshot *NEO/DEGAS* works correctly under circumstances where a program temporarily changes the screen resolution. For instance, *NEOchrome* always runs in low resolution, even if the computer is set for medium resolution before you run *NEOchrome.* If you capture a screen in *NEOchrome,* then exit to a medium-resolution desktop, Snapshot *NEO/DEGAS* remembers the correct resolution and saves the picture in lo-res format.

This does not apply, however, to a resolution change which does not occur under program control. If you switch resolutions from the desktop with the Set Preferences option, the ST reinitializes all desk accessories, effectively erasing any screen that Snapshot *NEO/DEGAS* has captured in memory.

Like most ST programs, Snapshot *NEO/DEGAS* opens the GEM file selector to let you choose a filename and pathname. Sometimes, calling the file selector from a desk accessory can have unexpected consequences. If the file selector box overlays an open disk directory window on the desktop, mouse events may occasionally "leak through" the file selector and affect the underlying window. In such cases, it's possible for GEM to be-

come confused about which activity—the file selector or the disk directory window—has priority in receiving mouse input. To avoid surprises, you can close any directory windows that are likely to lie under the file selector box when it appears on the desktop.

Although Snapshot *NEO/DEGAS* itself is less than 4000 bytes in length, it needs to reserve another 32,000-odd bytes of memory to store the screen image, color palette, and other data. That shouldn't create problems unless you're running a highly memory-intensive application on a 512K machine, or are using several other desk accessories which are very large.

Normally on the ST, pressing Alt-Help activates a graphics screen dump to your printer. Snapshot *NEO/DEGAS* diverts this hardcopy vector in order to capture the screen instead. If you run another program that also tries to divert the hardcopy vector for some other reason, Snapshot *NEO/DEGAS* probably won't work correctly. To avoid conflicts, do not use any other program or utility that relies on Alt-Help while Snapshot *NEO/DEGAS* is installed. If you wish to print a hardcopy image of a screen—either from the desktop or a program like *DEGAS*—turn the computer off and reboot with a startup disk that doesn't contain SNAPSHOT.ACC. Or temporarily rename the program to SNAPSHOT.AC and reboot. (A desk accessory must have the extension .ACC to be recognized by the system.)

Snapshot *NEO/DEGAS* may not work correctly on early 520STs which require you to load TOS (the operating system) from disk. Later 520STs and all 1040STs have TOS in ROM (Read Only Memory). There are many differences between the RAM-based and ROM-based versions of TOS. You can have an early 520ST upgraded with TOS in ROM chips at an authorized Atari service center.

Suggested Applications

Snapshot *NEO/DEGAS* is useful in many situations. You may want to create geometric figures in BASIC or Logo (or any other language, for that matter), save the picture, and then touch it up with *NEOchrome* or *DEGAS*. Many such figures can be created more easily with a program other than *NEOchrome* or *DEGAS*. Students can plot mathematical functions; dabblers in the stock market can track the progress of selected securities; artists can draw crystalline lattices.

Game players may also appreciate the ability to save

screens. When you struggle to reach the all-time high score, it's nice to have a permanent record of your achievement. It also proves that you really did make a certain score, especially if you have friends who may doubt your boasts.

Another idea is to snap screens to accompany newsletter articles. The screens can be printed out with *NEOchrome* or *DEGAS*, or merged into some of the desktop publishing programs available for the ST. The sample screens appearing in Figure 4-1 were captured with Snapshot *NEO/DEGAS*, then uploaded using special software to a minicomputer/typesetter. Some programs with animation can't be paused long enough for the long exposures required to photograph them; with those programs, Snapshot *NEO/DEGAS* captures a frozen image instantly.

Another useful feature of Snapshot *NEO/DEGAS* is its ability to convert a picture from *NEOchrome* to *DEGAS* format or vice versa. To convert a *NEOchrome* picture to *DEGAS* format, simply run *NEOchrome* and load the picture, select the Full Screen display, and then capture the picture by pressing Alt-Help. Exit *NEOchrome* and save the picture from the desktop with a *DEGAS* filename (using the extension .PI1 to indicate low resolution). The picture can then be loaded into *DEGAS*.

To convert from *DEGAS* to *NEOchrome* format, simply reverse the process: Capture the screen from within *DEGAS*, return to the desktop, and save it with a *NEOchrome* filename (using the extension .NEO). Since screen data is structured differently for different resolutions, this conversion works only for low-resolution pictures. You cannot convert between lo-res *NEOchrome* pictures and medium- or hi-res *DEGAS* pictures.

Incidentally, Snapshot *NEO/DEGAS* can capture a *NEOchrome* screen only when you have selected the Full Screen display. If you try to capture a screen that contains the *NEOchrome* tools at the bottom of the screen, the image will be incomplete. This is because *NEOchrome* uses special split-screen techniques to display more than the usual 16 colors in the palette box.

Extended Formatter

Richard Smereka

With this compact machine language utility, you can format blank disks with an option to increase the storage capacity of either a single- or double-sided disk. It works on any ST in any screen mode, either as an ST-Shell command or as a stand-alone application from the GEM desktop.

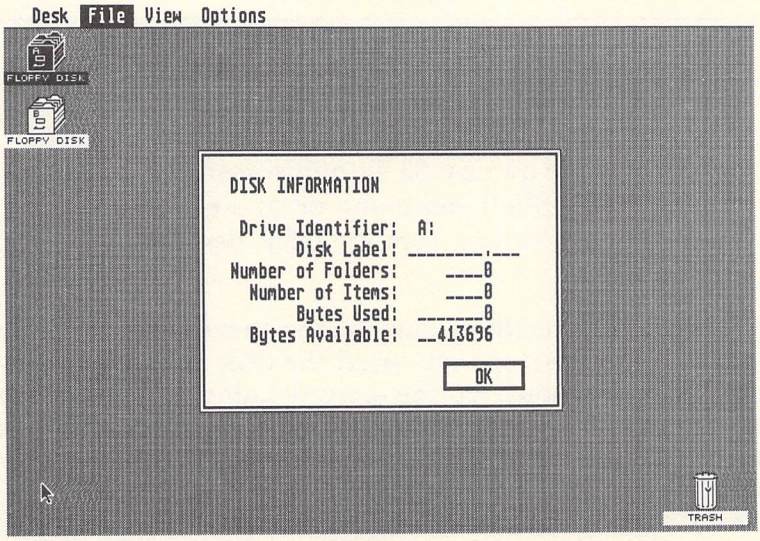
As you know, a formatting command is already built into the Atari ST's GEM desktop. You insert the disk to be formatted into a drive, click the mouse on the corresponding drive icon, drop down the File menu, and select Format. A dialog box pops open to let you choose single- or double-sided formatting. It's all very quick and easy.

Why, then, is a separate utility to format a disk a useful addition to your software library? For one thing, the integral format command is available only from the GEM desktop. If you're using an alternative such as *ST-Shell*—the command-driven disk operating system discussed earlier in Chapter 4—there may be no format command available. In addition, GEM's Format option currently doesn't offer some special features that are possible in a custom formatting utility.

"Extended Formatter" fills both of these voids. First, it provides a handy format command for the *ST-Shell* command line interface. And second, as its name implies, Extended Formatter provides a special formatting option not currently supported by GEM: You can format a single-sided disk to store 404K of data instead of the standard 349K, or a double-sided disk to store 808K instead of the usual 698K. Best of all, disks formatted with this option can be read from or written to without using any special software. They're fully interchangeable with regular disks.

Extended Formatter works on any ST in any screen mode, and it works with both single- and double-sided floppy disk drives.

Figure 4-2. "Extended Formatter" Results—with Single- and Double-Sided Disks—Shown in Number of Bytes Available



Installing on the Desktop

There are two different ways to install Extended Formatter, depending on whether you plan to use it from the GEM desktop or from a command-driven DOS such as *ST-Shell.*

To use it from the desktop, copy the file FORMAT.XXX from the book disk to your own disk and rename it FORMAT.TTP. The .TTP filename extension is important. It stands for *TOS Takes Parameters,* which signals to the ST that Extended Formatter is a Tramiel Operating System program that requires certain parameters before it can function. When you run a TTP application by double-clicking on its icon, a dialog box pops open on the screen so you can enter these parameters. (We've named the program FORMAT.XXX on the disk to guard against accidentally formatting the disk.)

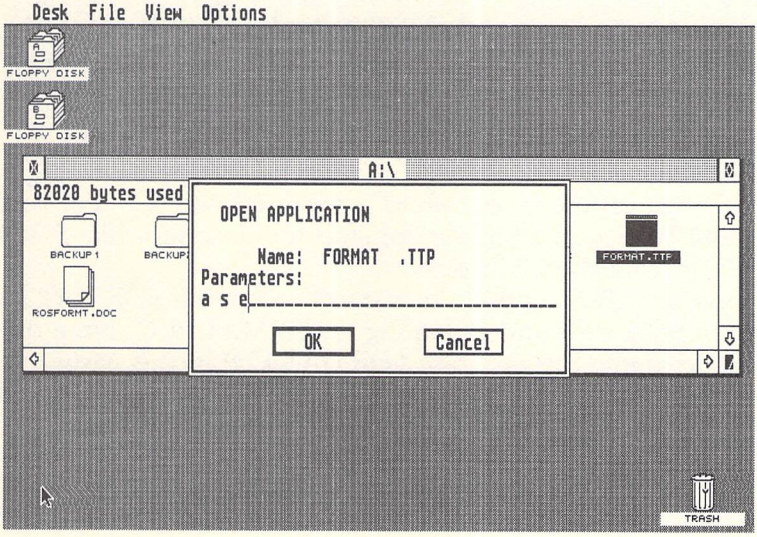
Here are the parameters expected by Extended Formatter:

D S **[E]**

The first two parameters are required. *D* is the drive identifier; substitute A (for drive A) or B (for drive B). *S* indicates the number of sides to be formatted; use S for single-sided or D for double-sided. (You must have a double-sided drive to use D, of course.) The third parameter is optional; if E is entered (do not

type the brackets), the extended-formatting option is selected. The parameters must be separated from each other by at least one space.

Figure 4-3. GEM Dialog Box for Entering "Extended Formatter" Parameters



Here are some examples of what you might type into the dialog box when you run Extended Formatter:

A S

This formats the disk in drive A for single-sided use.

B D

This formats the disk in drive B for double-sided use.

A S E

These are the parameters shown in Figure 4-3; they format the disk in drive A for single-sided use and also select the extended-formatting option (404K).

A D E

This formats the disk in drive A for double-sided use with the extended-formatting option (808K).

After you've entered the parameters and clicked on the OK button or pressed Return, Extended Formatter loads into memory

and then waits for you to press a key. This gives you a chance to swap disks if necessary. (On a single-drive system, naturally, you wouldn't want to format the disk from which you loaded Extended Formatter.)

That's all there is to it. Just remember that formatting erases any previous information that may have been stored on the disk. GEM's Format command warns you about this with an alert box, but Extended Formatter does not. This shouldn't be a problem, however, since the TTP dialog box—which should be warning enough—provides a Cancel button to abort the program.

Formatting from *ST-Shell*

A somewhat different procedure is required when you're running Extended Formatter from a command line interface like *ST-Shell* instead of from the GEM desktop.

First, install Extended Formatter by copying FORMAT.XXX from the book disk and renaming it FORMAT.PRG. Note that you *do not* name the program FORMAT.TTP in this case.

To run Extended Formatter, type the following command line:

FORMAT *D S* **[E]**

The parameters are the same as those used when Extended Formatter is being run from the desktop. *D* is the drive identifier, either A or B; *S* is the number of sides to format, either S for single-sided or D for double-sided; and E is the optional parameter for extended formatting (again, omit the brackets). The parameters must be separated from each other and from the FORMAT command by at least one space. Here are some examples:

FORMAT A S

This formats the disk in drive A for single-sided use.

FORMAT B D

This formats the disk in drive B for double-sided use.

FORMAT A S E

This formats the disk in drive A for single-sided use and also selects the extended-formatting option.

FORMAT A D E

This formats the disk in drive A for double-sided use with extended formatting.

After you've entered the parameters and pressed Return, Extended Formatter loads and then waits for you to press a key. This gives you a chance to swap disks if necessary.

Additional Tips

Extended Formatter works only with microfloppy disks in drives A or B. Attempting to format a disk in any other drive causes an *invalid drive specification* error. Also, do not attempt to format a RAM disk or hard disk with this utility.

When you run Extended Formatter, there must be at least 20,000 bytes of memory available in the computer for the track buffer. If there's not enough memory, an error will result.

Normally a single-sided ST disk has 349K of disk space available and a double-sided disk has 698K. The Extended Format option boosts these capacities to 404K and 808K, respectively, by increasing the number of tracks on the disk as well as the number of sectors per track. There should be no problem reading from or writing to these disks with drives that are properly aligned. (Atari recommends, however, that you don't format your disks to sizes other than those defined by the default system values. While extended-format disks have been used for some time at COMPUTE! without any problem, we recommend that, to avoid trouble, you fill an extended disk with files that can be replaced, and test it awhile before entrusting it with your irreplaceable data.)

When using the Extended Format option, you may have to change the way you copy disks. The Atari ST's disk copy routine—which is called when you drag a disk icon atop another disk icon or use the copy command from *ST-Shell*—will not copy a normally formatted disk onto an extended format disk, or vice versa. That's because the operating system first checks to be sure the disks used in a copy operation are compatible. That is, both disks must have the same total storage space available. If not, the computer informs you that the disks are incompatible. You may have encountered this message when trying to copy a floppy disk to a RAM disk or vice versa.

To copy a normally formatted disk onto an extended format disk, copy the individual files and folders from disk to disk. The easiest way to do this is to select a number of files simultaneously by looping them with the mouse or clicking on them while holding down a Shift key. See your ST manual for more information on extended-selection copying.

It is possible, however, to copy extended format disks onto each other by dragging the disk icon or by using the copy command from *ST-Shell*.

Customizing the GEM Desktop

McKendre Haynes

Have you ever wondered about the meaning of all those numbers in the DESKTOP.INF file? Wonder no longer. This in-depth article takes the mystery out of customizing and saving the GEM desktop. The techniques apply to all STs, color and monochrome.

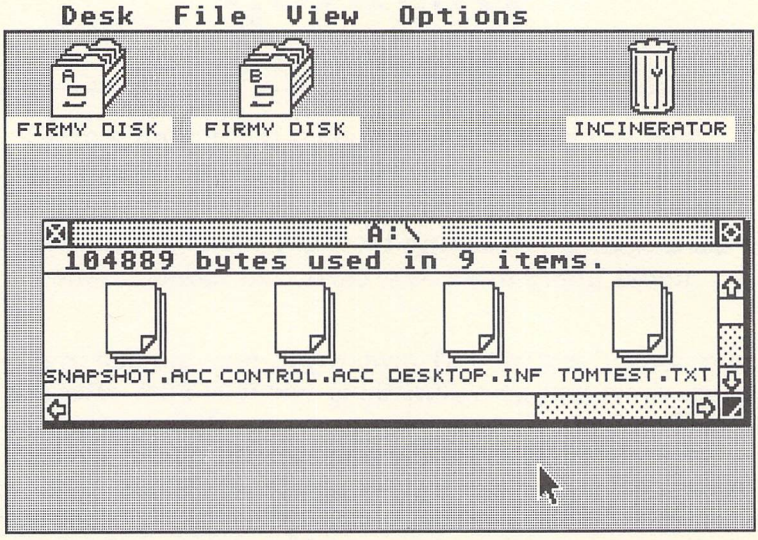
If you're a relatively new ST owner, you may indulge in a power-on ritual that goes something like this: First, after the GEM desktop appears, you drop down the Options menu and select Set Preferences to set the screen resolution to medium (if you have a color monitor) instead of low resolution. Then, perhaps you use the same menu to turn off confirmation of copies or deletes. Next, you drop down the View menu and select Show as Text to replace the file icons with filenames. Moving over to the Desk menu, you open the Control Panel and change the screen background color to something easier on the eyes than fluorescent green. Perhaps you open the Install Printer accessory and adjust the printer parameters, too.

Finally, after dragging the trash can to the lower right-hand corner of the screen, you double-click on the Floppy Drive A icon, resize the directory window, and reposition it slightly. Now the desktop looks the way you want it to look.

Arranging the desktop to suit your taste takes time. And what happens when you turn off the ST? It forgets everything. The next time you turn it on, you have to rearrange everything all over again.

It doesn't have to be that way. After moving things around and setting your preferences, you can drop down the Options menu and select Save Desktop. This writes a file called DESKTOP.INF on drive A. When you next turn on your ST, be sure this disk is in drive A. The computer reads the file and automatically reconstructs the GEM desktop just as you left it.

Figure 4-4. Modifying the DESKTOP.INF File: The Key to a Customized GEM Desktop



Exploring DESKTOP.INF

Not too long ago, I read an article which described how to change the labels of the desktop icons. In the process of modifying the file, I became curious as to what all those other numbers in DESKTOP.INF meant.

The DESKTOP.INF file tells the ST how to configure the GEM desktop whenever the power is switched on or the reset button is pressed. The ST continually monitors the GEM desktop, so it's always ready to save the various values on disk whenever Save Desktop is selected.

To view the contents of the DESKTOP.INF file, double-click on its icon or filename in a directory window. An alert box pops open to offer three options: SHOW, PRINT, and CANCEL. Click on the SHOW button to display the DESKTOP.INF file on the screen; click on PRINT to send a copy to the printer; or click on CANCEL to return to the desktop.

When listed to the screen or printer, a typical DESKTOP.INF file looks something like this:

```
#a030001
#b001000
#c77700070037300700552005055522207705570570577055031110 05
```

```
#d
#E  1B 02
#W  00  00  00  05  50  14  07  B:\*.*@
#W  00  00  0C  04  50  0E  00  @
#W  00  00  0E  09  2A  0B  00  @
#W  00  00  0F  0A  2A  0B  00  @
#M  00  00  00  FF  A FLOPPY DISK@ @
#M  01  00  00  FF  B FLOPPY DISK@ @
#T  07  00  02  FF  TRASH@ @
#F  FF  04  @ *.*@
#D  FF  01  @ *.*@
#G  03  FF  *.APP@ @
#G  03  FF  *.PRG@ @
#F  03  04  *.TOS@ @
#P  03  04  *.TTP@ @
```

Yours will probably look slightly different, depending on how you've configured your desktop.

By changing the data in this file, you can customize your desktop. Changing the data is easy if you load DESKTOP.INF into a text editor or word processor that can save files in plain ASCII format (for instance, *1ST Word* with word processor mode turned off). However, it's also easy to mess up the DESKTOP.INF file if you don't know exactly what you're doing. If you're not sure how to go about this, just continue reading.

How It Works

Each line in the DESKTOP.INF file contains a series of hexadecimal (base 16) numbers, and some lines contain additional symbols. Let's take a look at what these lines do.

The first line in DESKTOP.INF contains values describing the configuration of the RS-232 port:

Digit	Meaning	
1st:	0 = Full duplex	1 = Half duplex
2nd:	0 = 9600 bps	1 = 4800 bps
	2 = 1200 bps	3 = 300 bps
3rd:	0 = No parity	1 = Odd parity
	2 = Even parity	
4th:	0 = 8 bits/char	1 = 7 bits/char
	2 = 6 bits/char	3 = 5 bits/char
5th:	0 = X OFF, Rts/Cts OFF	
	1 = X ON, Rts/Cts OFF	
	2 = X OFF, Rts/Cts ON	
	3 = X ON, Rts/Cts ON	
6th:	0 = Strip bit ON	
	1 = Strip bit OFF	

So, for example, since the first line in the sample DESKTOP.INF file above is #a030001, the RS-232 port will be configured as follows: full duplex, 300 bps (bits per second), no parity, 8 bits per character, X OFF, Rts/Cts OFF, and strip bit OFF.

The next line in DESKTOP.INF contains the printer information you've entered with the Install Printer accessory. The numbers represent the following settings:

Digit	Meaning
1st:	0 = Dot-matrix
	1 = Daisywheel
2nd:	0 = Black and white
	1 = Color
3rd:	0 = 1280 pixels/line
	1 = 960 pixels/line
4th:	0 = Draft quality
	1 = Final quality
5th:	0 = Printer port
	1 = Modem port
6th:	0 = Continuous feed
	1 = Single sheet feed

In our example DESKTOP.INF file, the second line is #b001000, so the printer settings are dot-matrix, black and white, 960 pixels per line, draft quality, printer (parallel) port, and continuous feed. This is a typical setup for many Epson-compatible printers.

The Control Panel

The third line in DESKTOP.INF contains the values from the Control Panel accessory when the desktop was saved. In our example file, this line is

```
#c777 000 700 373 007 005 520 050 555 222 077 055 707 505 770 550
  3111005
```

Note: For illustrative purposes, the first 48 numbers are arranged in groups of 3, separated by spaces. This is *not* the way you would normally see it in the DESKTOP.INF file, where the numbers run together.

The first 48 numbers are the colors set by the Control Panel. Since the ST can display a maximum of 16 colors at any one time, each color is represented by 3 numbers (48 / 16 = 3). Each

of the 3 numbers corresponds to the number displayed under R, G, and B on the Control Panel color selector. R is the red value; G is the green value; and B is the blue value. All 512 colors possible on the ST can be created by combining the various RGB values.

The last seven numbers in this line, 3111005 in the example, are also set by the Control Panel:

Digit	Meaning
1st:	Mouse button response (0–4)
2nd:	0 = Keyclick off
	1 = Keyclick on
3rd:	0 = Bell off
	1 = Bell on
4th & 5th:	Keyboard response (0–46)
6th & 7th:	Character repeat delay (0–21)

Again, looking at our example file, you can see that 3111005 stands for a mouse button response of 3 (thus requiring fairly rapid double-clicks); keyclick on; bell on; a keyboard response of 10 (fairly fast); and a character repeat delay of 05 (a very short delay for fast repeats).

The next line in DESKTOP.INF, labeled #d, does not seem to be used yet. Most likely it's reserved for future use by a new version of the ST.

Icons and Screens

The next line in the file, #E 1B 02 in our example, performs two functions. The first hexadecimal number determines how files are arranged in the directory and if confirmation should be requested before copying or deleting files. The 32 possible values (not all of which are listed here) depend on how the four options are chosen—for example,

1B = View as Icons; Sort by Name; Confirm Deletes; Confirm Copies
03 = View as Icons; Sort by Name; No Confirm Deletes; No Confirm Copies
9B = View as Text; Sort by Name; Confirm Deletes; Confirm Copies

The first two options—View as Icons or View as Text and Sort by Name, Date, Size, or Type—are found under the View menu. The last two—Confirm Deletes and Confirm Copies—can be set when you choose Set Preferences from the Options menu. All of these settings are encoded as bits in the hexadecimal number. Bit 7 is icons/text; bits 6 and 5 are sort by name, date, size,

or type; bit 4 is for confirming deletions, and bit 3 is for confirming copies.

The second number in the #E line, 02 in our example, indicates the screen resolution to which the GEM desktop will default. Low resolution is 01, medium is 02, and high is 03.

Window Info

The next four lines represent the four windows that may be opened (*W* apparently stands for *window*). GEM permits only four windows to be open simultaneously:

```
#W  00  00  00  05  50  14  07  B:\*.*@
#W  00  00  0C  04  50  0E  00  @
#W  00  00  0E  09  2A  0B  00  @
#W  00  00  0F  0A  2A  0B  00  @
```

These hexadecimal values are continuously updated in memory as windows are opened, resized, and repositioned on the desktop. (This is why, when you close a window, the next window opened will appear in its former location and size.) When you select Save Desktop, the current arrangement is saved on disk. The next time you boot up the computer or press the reset button, the window (or windows) will reopen and position itself exactly as it appeared when you selected Save Desktop.

For each line and each window, the first two bytes are related to the slider bar positions, the next two determine the window's desktop location, and the third pair define the window's size. In the first line of our example above, these pairs of bytes are 00 00, 00 05, and 50 14.

The use of the next byte is uncertain, but it appears to be used by GEM as a window handle. This byte is always a value between $07 and $0A, with $07 the first value assigned to a window. The values of this byte seem to vary from ST to ST.

In the first window line, the symbols B:*.*@ represent the pathname of the open window: It is the root directory (not a folder) of disk drive B, and all files will be displayed. If a window had been opened for a folder in drive B, the line would end like this: B:\FOLDER*.*@, where FOLDER is the folder name. As each successive folder is opened, the latest folder name is added to the line.

The windows are opened in the order listed, so if you have A listed before B, the directory for B will be opened last and will come up as the currently active window. Changing the path-

name allows you to customize the directory that appears. For example, if you make B: *.*@ into B: *.BAS@, only BASIC programs will be listed in the directory window for drive B when you boot or reset your ST. Closing and reopening the window restores the normal directory.

Disk Drives and Trash Cans

The next three lines in DESKTOP.INF list information about the disk drive and trash can icons:

```
#M  00  00  00  FF  A FLOPPY DISK@ @
#M  01  00  00  FF  B FLOPPY DISK@ @
#T  07  00  02  FF  TRASH@ @
```

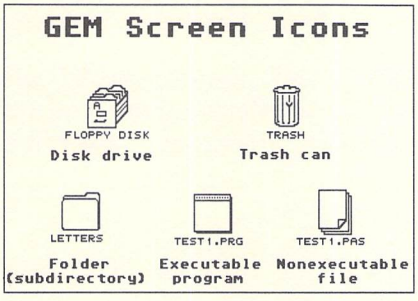
The M and T lines tell the ST where to locate the icons on the desktop, what the icons should look like, and how the icons should be labeled.

The first byte in each line is the horizontal location of the icon; the second is the vertical location; and the third byte tells the ST which GEM icon to use for each device.

There are five icons built into the ST, numbered as follows:

00 = File drawer
01 = Folder
02 = Trash can
03 = Program
04 = File

Figure 4-5. GEM's Built-In Icons



You can change the icon numbers for the disk drives and trash can to any of the above values, but the results will likely be confusing. (A trash can that's disguised to look like a disk drive could be hazardous to the health of your files.) If an icon specifier greater than 04 is used, the system will not boot.

The text portions of the lines which appear after the FF delimiter are the desktop icon labels. These, too, can be edited. You might, for example, relabel your disk drive icons as FIRMY DISKS since the 3½-inch floppies are not really very floppy, and rename the trash can icon to INCINERATOR, since—unlike the Macintosh and Amiga—the ST provides no simple way to recover a discarded file. The resulting lines in the DESKTOP.INF file would look like this:

```
#M  00  00  00  FF  A FIRMY DISK@ @
#M  01  00  00  FF  B FIRMY DISK@ @
#T  07  00  02  FF  INCINERATOR@ @
```

Some people with two-drive 1040ST systems change their icon labels to read INTERNAL for drive A and EXTERNAL for drive B, since the 1040ST has a built-in floppy drive.

By the way, there's an easier method of changing a disk drive icon than by editing the DESKTOP.INF file with a text editor or word processor. Simply click once on the icon to highlight it, drop down the Options menu, and select Install Disk Drive. The dialog box that appears lets you change the drive identifier and the icon label. You shouldn't mess with the drive identifier (this is for installing a RAM disk or hard disk on your system), but you can change the label quite easily. Press the Tab key to move the cursor to the line with the label, and then press the Esc key to erase the line. Now you can type in any new label you want (up to 12 characters). Click on the INSTALL button to effect the change, or the CANCEL button to restore the original label.

File Icons

The rest of the DESKTOP.INF file tells the ST which icons to use for different types of files within windows and how to deal with those types of files. The example above looks like this:

```
#F  FF  04  @ *.*@
#D  FF  01  @ *.*@
#G  03  FF  *.APP@ @
#G  03  FF  *.PRG@ @
#F  03  04  *.TOS@ @
#P  03  04  *.TTP@ @
```

The #F and #D lines may represent folders and disks; their function is uncertain. The next four lines describe the four types of programs that can run. Here, G files run under the Graphics Environment Manager (GEM), and F and P files are Tramiel Operating

System (TOS) programs which run with or without parameters.

There's no useful purpose to modifying these lines, and they should be left untouched to avoid problems.

Installing Applications

Occasionally you'll see a DESKTOP.INF file that has lines like this:

```
#G  03  04  BASIC.PRG@ *.BAS@
#G  03  04  1ST_WORD.PRG@ *.DOC@
```

These indicate applications that have been installed. In GEM desktop parlance, *installing an application* has a special meaning. Usually when you double-click on a file represented by a file (04) icon—such as an ST BASIC program or a *1ST Word* file—an alert box tells you that you can only SHOW or PRINT the file; it's not an executable program. In the case of BASIC programs and *1ST Word* documents, the files are identified with the filename extensions .BAS and .DOC, respectively.

But by installing the application, you can run the application program merely by clicking on one of its nonexecutable files. For example, let's say you want to install *1ST Word*. Click once on the *program icon* for *1ST Word* (the icon you'd double-click to run the program). When it's highlighted, drop down the Options menu and select Install Application. A dialog box appears and asks you for a Document Type. Enter DOC as the document type; then click the OK button.

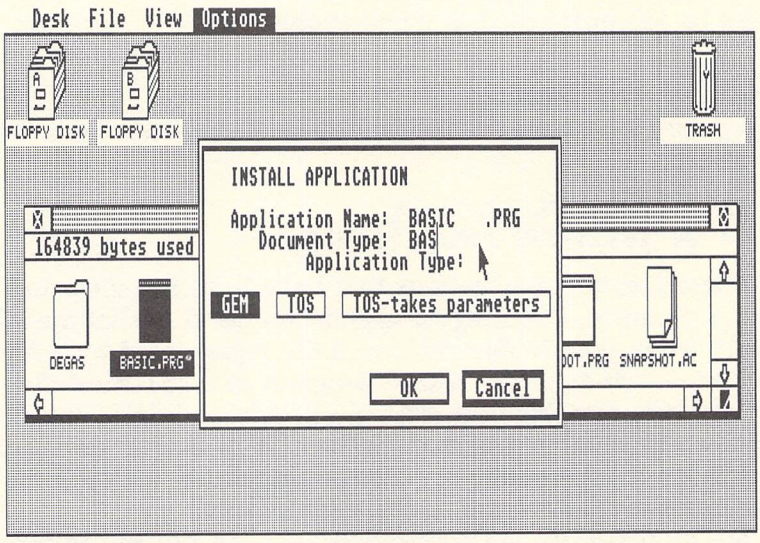
Now you can load and run *1ST Word* just by double-clicking on any *1ST Word* document file (that ends with the filename extension .DOC). Not only is *1ST Word* loaded, but the text file you click on is automatically loaded, too. Of course, the *1ST Word* program file must be on the same disk as the document file for this to work.

If you reboot or reset the ST, the application will be uninstalled. To keep from having to install it each time you use the computer, simply Save Desktop.

Saving Room for Accessories

A desk accessory is a special type of program that automatically loads into memory when an ST is booted, then waits "in the background" until selected from the Desk menu. At that point it runs instantly, since it doesn't have to be loaded from disk like other programs. Also, a desk accessory can be called anytime the

Figure 4-6. Installing an Application from the Desktop



Desk menu is available, even when you're running another application program. (Sometimes, as in *1ST Word*, the Desk menu is represented by the Atari logo symbol.)

Desk accessory files end with the .ACC extension and must be located on the root directory in order for you to be able to load them. Normally, they'll load from drive A. If you have a hard disk, accessories load from drive C. The most common desk accessory is the Control Panel (CONTROL.ACC) that comes with every ST system. Another free accessory is the VT-52 emulator (EMULATOR.ACC).

Accessories can be useful, but they consume memory and lengthen the time required for the computer to boot up. Also, the current version of TOS limits the ST to a maximum of six accessories that may be installed at one time. The Control Panel, although it is a single accessory, uses up two of these slots because it includes the Install Printer program.

If you want to free up some memory or some slots on the Desk menu for other accessories, you can Save Desktop after configuring your system with the Control Panel, Install Printer, and VT-52 emulator. Then you can prevent these accessories from loading in the future by deleting them from your boot disk or renaming them CONTROL.AC and EMULATOR.AC. Since all the

information supplied to your ST by these accessories is preserved in the DESKTOP.INF file when the desktop is saved to disk, it's not really necessary to keep them around.

The only function you might miss is the system clock on the Control Panel. Several programs are available both commercially and in the public domain which let you set the clock during bootup. This could eliminate your need for the Control Panel/ Install Printer accessory, freeing up those two slots for more important accessories.

NEOview

Philip I. Nelson

Here are two programs—one written in machine language and the other in C—which let you display NEOchrome *pictures without loading* NEOchrome. *Programmers will want to study the accompanying source code, which demonstrates how to load* NEOchrome *pictures into your own programs.*

NEOchrome, the graphics-design program supplied with the ST, is a powerful tool for artists and doodlers alike. But it can also be a powerful tool for programmers. For instance, if you're writing arcade-style games or other graphics applications, it saves time and effort to draw the background with *NEOchrome* and then just load it into screen memory, rather than to write a complex routine to create a similar background.

But how do you go about displaying a *NEOchrome* picture within a program of your own? The accompanying programs provide the answer—in both C and machine language. You should be able to convert them into routines for your own programs with little trouble. And the techniques they illustrate are valuable for anyone interested in writing programs that involve *NEOchrome*-format files.

In addition, these programs are useful in their own right for anyone who wants a quick look at a *NEOchrome* picture without running *NEOchrome*. You don't have to be a programmer to use them.

Viewing with NEOview

Before running either version of "NEOview," make sure the computer is in the low-resolution screen mode. This is the only mode currently supported by *NEOchrome*.

The machine language version of NEOview is stored on the disk as NEOVIEW.TTP and runs as a *TTP* (TOS Takes Parameters) application from the desktop. When you run a TTP application, the ST opens a dialog box in which you may type information to be passed to the program. In this case, enter the name of the *NEOchrome* picture you wish to display, using the full drive specifier, pathname, and filename.

For example, to display a picture called MYPIC.NEO on the disk in drive A:, enter A:MYPIC.NEO in the dialog box and press Return. NEOview displays the picture until you press Return again. If a disk error occurs, the program prints an error message and waits for you to press Return. If you have a single-drive system, make sure that NEOVIEW.TTP is on the same disk as the *NEOchrome* picture you want to display.

The C version of NEOview is stored on the disk as NEOVIEW.TOS and runs as a TOS (Tramiel Operating System) application. When you run the program, it prompts you to enter the name of the file you wish to display. Again, you should enter the full drive specifier, pathname, and picture filename. For instance, you would enter B:PICS \DOG.NEO to load DOG.NEO from the folder PICS on the disk in drive B:. If a disk error occurs, the program prints an error message; otherwise, it displays the picture and waits for you to press any key.

How It Works

In addition to the executable object files on disk, commented source code for the C version of NEOview—and the source code for the machine language version, as well—follows this discussion of the programs. If you're a C or machine language programmer, you'll want to study this code to learn how to load *NEOchrome* pictures with your own programs.

NEOview was written first in machine language using *AS68*, the 68000 assembler included with the Atari ST development system. It was then translated with *Alcyon C*, the C compiler that comes with the development system. Minor changes may be needed to compile and link the C source code with other C compilers.

To facilitate comparison, both versions of the program perform essentially the same tasks in the same order. After they have accepted the filename, the programs save the current color palette, read the color-palette data from the *NEOchrome* file, and reset the ST's palette with the new colors. Then they read the screen-image portion of the file directly into the computer's screen memory, close the disk file, and wait for the user to press a key. Finally, they restore the original color palette and return to the desktop.

These programs illustrate only the bare essentials needed to get the job done. If you decide to incorporate any of these techniques in a program of your own, you will probably want to add more error-checking or convenience features.

NEOview: Commented C Source Code

```
/* SHOWNEO.C */
/* Display a NEOchrome picture. This program runs as a TOS      */
/* application from the desktop. Enter the filename of the      */
/* NEOchrome file you wish to display. If the file is found,    */
/* the program resets the palette, displays the picture, and    */
/* waits for you to press any key. If the file doesn't exist,   */
/* or if a disk error occurs, the program displays a terse      */
/* error message. Because NEOchrome currently works only in     */
/* low resolution, the program assumes you are already in low   */
/* res before you run. To save space, the program doesn't do    */
/* checks like scanning the extender to make sure it's .NEO     */
/* or checking the file size before reading to make sure it's   */
/* 32128 bytes.                                                 */

#include <osbind.h>

char namebuff[40];
char errormsg[ ] = "A disk error occurred. Press any key....";
char prompt[ ] = "Enter NEO filename: ";

int savepal[16], newpal[16], junkbuff[46], *screen;

main( )
{   int i, filehandle;
    namebuff[0] = 37;
    Cconws(prompt);
    Cconrs(namebuff);
    namebuff[namebuff[1]+2] = '\0';

    if( (filehandle = Fopen(&namebuff[2],0) )<0 ) {      error( );
      return(0);
      }
/* Make the flashing cursor disappear.                          */
    Cursconf(0,0);
/* Save current color palette                                   */
    for(i=0;i<16;i++) savepal[i] = Setcolor(i, -1);
/* Read and throw away first two words.                         */
    if( (i = Fread(filehandle, 4L, junkbuff) ) <4 ) {    error( );
      return(0);
      }
/* Read 16 words of palette data into newpal array.             */
    if( (i = Fread(filehandle, 32L, newpal) ) <32 ) {    error( );
      return(0);
      }
/* Tell system to use the new color palette                     */

    Setpalette(newpal);
```

```
/* Read and throw away 46 words of color cycling data          */
   if( (i = Fread(filehandle, 92L, junkbuff) ) <92 ) {      error( );
      return(0);
      }
/* Find screenbase                                             */
   screen = (int *) Physbase( );
/* Read screen image into screen memory.                       */
   if( (i = Fread(filehandle, 32000L, screen) ) <32000 ) {
      error( );
      return(0);
      }
/* Close the file.                                             */
   Fclose(filehandle);
/* Wait for a keypress                                         */
   Bconin(2);
/* Restore original palette                                    */
   Setpalette(savepal);
} /* main ends here                                           */
/* Print error message and wait for any key to be pressed.     */

error( )
{   Cconws(errormsg);
   Bconin(2);
}
```

NEOview: Machine Language Source Code

```
* SHOWNEO.S

start:

* Enter supervisor mode
clr.l           -(sp)
move.w          #$20,-(sp)
trap            #1
addq.l          #6,sp
move.l          d0,savesp

* Save color palette
move.w          #32,d0
move.l          #savepal,a1
move.l          $ff8240,a0
sv:
move.b          (a0)+,(a1)+dbra     d0,sv

* Disable cursor
move.w          #0,-(sp)
move.w          #21,-(sp)
trap            #14
addq.l          #6,sp
```

```
* Find command tail
lea             start,a0
sub.l           #128,a0
clr             d0
move.b          (a0),d0
adda.l          #1,a0

* Reject null name
tst.w           d0
beq             exit

* Zero-terminate tail
move.b          #0,0(a0,d0)

* Open for read
move.w          #0,-(sp)
move.l          a0,-(sp)
move.w          #$3d,-(sp)
trap            #1
move.w          d0,filehandle
add.l           #8,sp
tst.w           d0
bmi             errout

* Discard rez info
move.l          #junkbuff,-(sp)
move.l          #4,-(sp)
move.w          filehandle,-(sp)
move.w          #$3f,-(sp)
trap            #1
add.l           #12,sp
tst.w           d0
bmi             errout

* Read palette data
move.l          #newpal,-(sp)
move.l          #32,-(sp)
move.w          filehandle,-(sp)
move.w          #$3f,-(sp)
trap            #1
add.l           #12,sp
tst.w           d0
bmi             errout

* Use new palette
move.l          #newpal,$45a

* Discard cycling info
move.l          #junkbuff,-(sp)
move.l          #92,-(sp)
```

```
move.w          filehandle,-(sp)
move.w          #$3f,-(sp)
trap            #1
add.l           #12,sp
tst.w           d0
bmi             errout

* Get screen base
move.l          $44e,savescr

* Read screen image
move.l          savescr,-(sp)
move.l          #32000,-(sp)
move.w          filehandle,-(sp)
move.w          #$3f,-(sp)
trap            #1
add.l           #12,sp
tst.w           d0
bmi             errout

* Close file
move.w          filehandle,-(sp)
move.w          #$3e,-(sp)
trap            #1
add.l           #4,sp
tst.w           d0
bmi             errout

* Wait for Return
wait:
move            #1,-(sp)
trap            #1
add.l           #2,sp

* Restore palette
exit:
move.l          #savepal,$45a

* Back to user mode
move.l          savesp,-(sp)
move.w          #$20,-(sp)
trap            #1
add.l           #6,sp

* Exit to desktop
move.w          #0,-(sp)
moveq.l         #0,d0
trap            #1
```

```
* Error handler
errout:
move.l          #errormsg,-(sp)
move.w          #9,-(sp)
trap            #1
addq.l          #6,sp
bra             wait

errormsg:
.dc.b 13,10,"A disk error occurred. Press Return...",0
.bss
savepal:
.ds.w 16
savesp:
.ds.l 1
savescr:
.ds.l 1
screen:
.ds.l 1
filehandle:
.ds.w 1
newpal:
.ds.w 16
junkbuff:
.ds.w 46
```

Full-Screen Shell for ST BASIC

David Lindsley

If you've ever wanted to write an ST BASIC program that isn't confined to the BASIC output window, you'll be glad to see this program. It shows you how to create full-screen graphics that don't depend on the usual window borders.

Windows are integral to the ST BASIC programming environment. Whether you're typing, listing, or running a program, everything occurs within a bordered window. Since ST BASIC provides no commands for monitoring gadgets such as the window scroll bar, the gadgets serve no real purpose in most programs. And in applications such as games, the ever-present borders prevent you from using the full area of the screen.

This program creates a full-screen shell for your own ST BASIC programs. By enclosing a program within this code, you can override BASIC's windowing environment and work with the entire screen surface.

When you run the demonstration program (on the disk), the screen is filled immediately with a graphic design. At the top of the screen, where the ST BASIC menu titles normally appear, is a title bar containing the name of this program. After a short pause, the screen clears and returns to normal, displaying the ST BASIC menu titles which were overdrawn while the program ran.

Running the Program

Before you can run "Full-Screen Shell," though, you must load BASIC into memory. Insert your *ST Language Disk* into the computer; then double-click on the BASIC.PRG icon or filename from the desktop. When BASIC has finished loading, insert *COMPUTE!'s Second Book of Atari ST Disk*. Enter **LOAD A:\FULSCRN.BAS** (and press Return), followed by **RUN** (and another Return). Full-Screen Shell will then display the demo described above.

Enclosed in a Shell

The line numbering of this program is designed to make it easy to merge with your own programs. Lines 10–70 check the current screen resolution and adjust several variables accordingly. Line 80 calls the subroutine PRGNAME, which draws a title bar with the title you designate and fills the screen with the specified pattern.

Lines 100–5000 are reserved for your program. In the demonstration included, line 110 simply delays long enough for you to look at the screen. In a real program, of course, you would substitute your own code. Just remember that your portion of the program should use only line numbers 100–5000.

Instead of terminating with END, your program should fall through to lines 5010–5030. These lines restore the usual ST BASIC menu titles, clear the output window, and reopen it so that you can use BASIC normally. Since the shell code draws on the entire screen, it erases the ST BASIC menu titles. (However, the menus are still active while the program runs, so that you can select Break to stop the program, and so on.) Thus, it's necessary to redraw the menu titles when the program ends. The string *name$* in line 5010 contains the text for these titles, which you can change if you wish. The END statement at the end of line 5030 terminates the program.

Merging

Unless you write your programs with this shell in mind, most programs will need some modification before you merge them with the shell. This is necessary in order to preserve the windowless screen. Once you have cleared the screen completely, you cannot use ordinary BASIC graphics commands such as PCIRCLE, GOTOXY, LINEF, and CIRCLE. If you do, ST BASIC suddenly redraws the right and lower bars of the output window, even though these commands have nothing specifically to do with those window bars.

To avoid such unwanted effects, you must create all graphics with VDISYS commands which aren't tied to windows. This rule also includes text, which must be placed with VDISYS instead of PRINT. VDISYS commands are more complicated to use than most BASIC commands, but they can operate much faster, giving your program the appearance of something written in machine language. Any graphics or text that you create in ST

BASIC can also be created with VDISYS commands. In fact, BASIC itself uses VDI routines to create graphics in the first place.

The simplest way to use the shell program is to delete existing lines 100–110 and MERGE it with your own program code. Here are the steps to follow before you attempt the merge: First, renumber your program if necessary, so that its line numbers fall in the range 100–5000. Then substitute the name of your program in the string *title$* in line 5090. Delete any CLEARW 2 or FULLW 2 commands from the beginning of your program and rename any variables that conflict with the variable names used in the shell code. Once this is done, you can perform the merge.

Program Notes

Lines 50–70 set several important variables used by subroutines in the shell. The variables *dcx* and *dcy* represent the screen size, and the variables *c* and *s* indicate colors.

The PRGNAME subroutine beginning at line 5050 specifies the screen coordinates and color according to the current resolution and passes those values to the RECT subroutine. Lines 5100–5120 draw the top menu bar in the color specified by the variable *s*.

Lines 5130–5160 call a VDI routine which places text at the designated screen coordinates. Line 5150 centers the text on the screen. Line 5160 places the text 8 lines below the top border in low- and medium-resolution modes or 16 lines down in high-resolution mode. Lines 5170–5190 POKE the necessary information into memory prior to the VDISYS call. You can place the title lower on the screen by changing lines 5100 and 5160. You may want to include additional VDISYS calls to enlarge the lettering or create special text effects. Or you can eliminate the title altogether by deleting lines 5090–5190.

The RECT subroutine calls a VDI routine which fills the specified screen rectangle with the designated color and pattern.

The MENU subroutine is similar to the PRGNAME routine, but it's designed to clear the screen back to white, the usual background color (5300). The LEN function used in line 5150 is omitted in lines 5330 and 5360 because the number of characters in the string *name$* (including spaces) is now known to be 28. If you change the length of *name$*, change the 27 in line 5360 to match the new length.

CHAPTER FIVE

C Programming

Why C?

Sheldon Leemon

If you're curious about C, this discussion of what it's like as a programming language will answer some of your questions. In particular, you'll see why the language has become so popular on next-generation microcomputers like the Atari ST series.

If you're coming to the ST from the world of eight-bit Atari computers, you may feel that C has emerged from nowhere to become the dominant programming language on the ST. After all, on the older eight-bit machines, BASIC and machine language have been far and away the most popular programming languages, almost to the exclusion of any other languages. BASIC has filled the role of the beginner's language, easy to learn and easy to use because of its interactive nature. Since most home computers come with BASIC built into ROM (Read Only Memory), many people equate "making the computer do something" with giving it commands in BASIC.

Assembly or machine language (ML for short), on the other hand, has become the language of choice for commercial programs on eight-bit micros. Since ML is the only code that the computer really understands without the need for translation, ML programs can be smaller and faster than any other kind. They are, however, much more difficult to write than BASIC programs.

To many current Atari owners, these two languages have proven to be sufficient. So where did C come from, and why, all of a sudden, is it the language that you have to learn in order to program the ST effectively?

The Balance of Power

Contrary to what the frustrated programmer might think, C is not a new language that somebody dreamed up to spring on the long-suffering microcomputer hobbyist who has finally mastered BASIC and grudgingly come to terms with ML. Rather, its recent popularity reflects a natural evolution in software that has paralleled the rapid transformation of low-cost computer hardware into high-powered gear like the ST.

Choosing the right programming language for any computer involves weighing tradeoffs between the level of performance that the finished program must meet, and the time required to develop and maintain that program. Programs that are written in ML execute quickly and take up relatively little memory, but they are relatively difficult to write and maintain. It's much easier to program in higher-level languages, but these languages generally don't offer the same performance as ML, and the programs are generally much larger in size than comparable ML programs.

On the older eight-bit computers, there isn't much weighing to do. After all, you can't run a 100K Pascal program on a computer that has only 64K of memory. These machines usually don't have enough memory or mass storage to obtain reasonable performance from programs written in higher-level languages, so ML is more or less a necessity.

Minicomputers and mainframes, on the other hand, have so much power that they can achieve excellent performance from programs written in higher-level languages like FORTRAN and Pascal. As a result, the emphasis is on writing a functional program in a reasonable amount of development time.

The ST computers fit somewhere between these two extremes. They have a much faster microprocessor than do the older micros, along with large mass-storage capacity and lots of main memory. Still, they're not quite in the same league as mainframes.

A Mid-Level Language

That's where C comes in. It represents a good compromise between high-level languages like FORTRAN that are used extensively on mainframes and the machine-level programming that is a must for micros. C supports most of the fundamental features of high-level languages, but it also supports lower-level functions like bit manipulation of memory. In addition, it interfaces easily with ML programs, so time-critical portions of a C program can be written in ML.

Although C may not offer features like sophisticated handling of text strings, it does generate fairly compact code that executes rapidly enough to achieve good performance. In fact, most of GEM—which controls the windowing environment on the ST—is written not in ML, but in C. (This is another reason that C is the natural choice for those who wish to make use of GEM's features.)

Besides the strictly pragmatic considerations of the size and performance of C programs, the language itself has many features that make it well-suited for software development. It's a modern, *structured* language, the philosophy of which is based on the use of small subprograms called *functions*. Each function is a self-contained program that performs a particular task. The use of functions allows the programmer to break down the overall task into small, manageable pieces. Each piece can be independently tested and debugged, and then incorporated into larger functions that perform more complex tasks.

This modularity not only makes it easier to write and maintain properly working programs, but also helps to eliminate duplication of effort. For instance, assume you're writing several programs, each of which requires the user to enter some specific kind of information, such as an amount expressed in dollars and cents. You can create one general module that prompts the user and accepts the input, and then include that same module in each of your programs. In fact, libraries of such commonly used modules are available commercially. In C programming, it is quite common to obtain the skeleton of an application like a telecommunications program or database manager from a commercial or public domain source, and then expand and customize the program to meet your own needs.

Portable Programming

Another aspect of C is the kind of output it generates. C compilers create machine language code, just like machine language assemblers. (In fact, some C compilers generate assembly source code that is later assembled.) Such programs generally run faster than those compiled by languages which generate semi-interpreted code or pseudocode (*p-code*). Furthermore, compiled C programs can be executed without requiring any special support programs or runtime packages, so they're easy to operate and easy to distribute. The programs created by a C compiler can also be fairly compact, since they include only those parts of the language that are actually used in the program.

Finally, C offers a fair degree of portability—you can usually translate a C program to another computer without completely rewriting it. Although there isn't an official standard version of C, in practice, most implementations of the language are very similar.

Of course, programs that include any kind of graphics or windowing generally use very hardware-specific display methods, which makes it harder to convert them for use on computers with different types of display hardware. But by isolating these display routines into a small group of distinct functions, C programmers need to convert only these functions to enable their programs to operate on another machine. This makes it much easier to convert a C program written for the IBM PC to one for the Atari ST (particularly if it uses the GEM interface) than it would be to convert a similarly complex program from IBM BASIC to ST BASIC.

Worth the Wait

Despite these many advantages, there are still several reasons that newcomers might feel put off or intimidated by C. For one thing, it's a compiled, rather than an interpreted, language. Using an interpreted language like BASIC is an interactive experience. The language has a built-in screen editor, and after you've entered a line of code, you need only type RUN to see the program execute. Some BASICs (such as eight-bit Atari BASIC) even provide syntax checking on entry, so you get instant feedback if you make a typing mistake. If you run the program and discover it doesn't function properly, you just list the lines that are to blame, make some changes, and run the program again.

Compiled languages such as C are not so easy to use. First, you must compose the source code using a separate text editor. Then you use the compiler program to convert the source code into object code. The compiler may be a single program, or it may be composed of two or more programs, each of which handles a different phase of the compilation process. Finally, you must use a linker program to combine the object code with portions of the C library and convert it into an executable format.

If an error occurs at any stage of this process (due to a typing mistake in the source code, for instance), the whole procedure must be repeated until the program compiles and links successfully. Only then can you actually run the program to determine whether or not it does what you want it to do. If it doesn't, you've got to load up your text editor and try again. This is a far cry from BASIC, where you type PRINT "HELLO", press RETURN, and the computer prints HELLO.

Fortunately, many products have appeared which make the

process of generating C programs much less tedious. Very sophisticated text editors are now available, and some can even perform syntax checking so that you don't have to wait until compiling time to discover syntax errors. Many DOS shell programs are available for the ST which let you run batch programs to automate the process of compiling and linking into a single step. Some will return you to your text editor if the compiling process fails and give you error messages that help you locate the problem.

There are even some interpreted versions of C (though not yet for the Atari ST). These allow a programmer to develop programs in an interactive environment that is much more like BASIC. The difference is that once programs have been written using the interpreter, they can then be compiled and run with a speed that BASIC can't touch.

Memory Leverage

Thankfully, the power of the ST computers is a big help in compiling C programs. Even with compiled (rather than interpreted) C, the large amount of memory available in the 520ST and 1040ST provides a luxurious environment in which to work. Using one of many programs available either commercially or in the public domain, you can partition some of that memory as a RAM disk, then run your text editor, compiler, and linker from memory. The amount of time saved by not having to compile and link from floppy disk can dramatically increase your productivity. In fact, some compilers work so fast from a RAM disk that they hardly take longer than interpreters.

The main point to keep in mind is that deep down, a programming language is a programming language. Though C has its eccentricities of syntax and style, it still incorporates the basic concepts of conditional branching (IF-THEN-ELSE), loops (FOR-NEXT, DO-WHILE), and so on. If you've learned to program in BASIC, you can learn to program in C. And once you get used to it, you may really enjoy creating programs that run rings around eight-bit machine language without the tedium of machine language programming.

Now that's what I call "Power without the price."

Comparing C to BASIC

Sheldon Leemon

While C has a distinct look, BASIC programmers will recognize a number of features that it shares with the more familiar language. And adapting to the peculiarities that do exist—in C formatting, for example, or in its variable and function definitions—brings payoffs in programming efficiency that are well worth the effort.

Some BASIC programmers are hesitant to try C because of its image as a difficult language to learn. Though it is true that the process of writing a C program is substantially different from writing a program in BASIC, the two languages still have quite a bit in common. Once you get accustomed to C's distinctive syntax and style, it is possible to begin writing C programs in just a short time.

Perhaps the best way to demonstrate that C isn't so mysterious is to show an example of what a C program looks like. Following are listings of two programs, one in C and one in BASIC. Each produces a list of the prime numbers from 2 to 50. (In case you've forgotten, a prime number is only evenly divisible by itself and 1.) The list that is produced looks like this:

```
2
3
5
7
11
```

and so on (up to 47).

First comes the version written in C:

```
/* Sieve.c—Finds the prime numbers from 2 to SIZE */

main( )
{
  int num, x, count;          /* declare & initialize variables */
  #define SIZE 50
  char flags[SIZE+1];
```

```
num = 2;

    for (x = num; x <= SIZE; x = x+1)
        flags[x]=1;          /* set all flags */

    while (num <SIZE/2){
        for (x = 2*num; x <= SIZE; x = x+num)
            flags[x] = 0;     /* eliminate multiples */
    num = num +1;
    }
    for (x = 2; x <= SIZE; x = x+1)
        if (flags[x])
            printf("%2d \n",x); /* print the primes */
}
```

Now, here is the same program in ST BASIC.

```
100 REM Sieve.bas—Finds the primes between 2 and size
110 '
120 DEFINT a-z 'declare and initialize variables
130 size=50
140 DIM flags(size+1)
150 num=2
160 '
170 FOR x=num TO size
180 flags(x)=1 'set all flags
190 NEXT
200 '
210 WHILE (num<size/2)
220 FOR x=2*num TO size STEP num
230 flags(x)=0 'eliminate all multiples
240 NEXT x
250 num=num+1
260 WEND
270 '
280 FOR x=2 TO size
280 IF flags(x) THEN PRINT USING "##";x 'print the primes
300 NEXT x
```

As you can see, the two programs are not all that different. Let's compare them statement by statement.

Flexible Formatting

First of all, you'll notice the C program has no line numbers. A single statement can take up one line or many lines. The compiler doesn't get confused because each statement in C ends with a semicolon (;), and multiple statements grouped together as a

single block are enclosed in curly braces ({ }). The programmer decides how to arrange the statements on each line to make the program neat and readable. It is customary, however, to group the various parts of a C program together in a way that makes them visually distinct from one another.

The first statement, which starts with the slash and asterisk characters (/*), is a remark—identical to the REM statement in line 100 of the BASIC program. In C, a remark can extend over many lines until the closing */ characters. It is especially important to include many remarks in a C program, because the language is compact and each statement can do a lot of work. Without comments, it can be difficult to remember what a line of C code actually does. Furthermore, remarks don't affect the size of the final program in C, since the compiler ignores them. With a BASIC interpreter, REM statements consume memory.

Next comes the line *main( )*. This marks the start of a function named main. All C programs are made up of functions, which are small subprograms. Every C program has at least one function (called main) where program execution begins. The fact that the name *main* is followed by parentheses, but no semicolon, shows that it is a function definition. Some functions use values (called parameters) that are passed to them by other functions, and such functions contain the names of these variables listed within the parentheses. Since main( ) is the first function to execute, the only values it can be passed are those in the list of parameters that the user types in when starting a TTP (TOS Takes Parameters) type of program. The example program above is a TOS program, without parameters, so the parentheses after main are empty.

After the name of the function comes the curly brace character, {. Curly braces are plentiful in C programs—they mark the beginning and end of function definition blocks, and the beginning and end of compound statements within a function. As shown here, most programmers indent the lines to make the source code more readable—indentations help to visually match up left braces with their corresponding right braces.

Defining Variables

After the initial brace come three strange-looking statements:

```
int num, x, count;          /* declare & initialize variables */
#define SIZE 50
char flags[SIZE+1];
```

The first is roughly equivalent to the DEFINT statement in line 120 of the BASIC program. It declares that the variables named *num, x,* and *count* will be integers, and reserves space for these variables. To tell the truth, though, the equivalent BASIC line was added for instructive purposes rather than out of necessity. BASIC isn't a strongly typed language. Most of the time, you don't have to worry about whether a simple numeric variable is stored internally as an integer or a floating-point value (although most BASICs give you the option of specifying which should be used, for those times when it's necessary). Unless you specifically declare a variable type with a DEF statement, BASIC assumes a default type and reserves storage space for the variable on its own.

With C, however, declaration statements are not optional. You *must* take responsibility for deciding how much storage space will be allotted for each variable, and you can even specify the particular memory location to be used to store a variable. Each time you want to use a variable in C, you must declare ahead of time whether it should be stored as a long or short integer, in single- or double-precision floating-point math, or as text characters. These declarations are usually made in a block at the top of the function definition.

The only case in which BASIC really requires you to declare a variable ahead of time, the way C does, is when a subscripted array will have more than ten elements. In this respect, the DIM statement in line 140 of the BASIC program is very similar to the C declaration of the flags array.

The Preprocessor

The middle statement in the trio of C lines shown above is somewhat more complicated to explain. Where the BASIC program assigns the value of 50 to a variable called *size,* the C program uses the *#define* statement to define a macro called *SIZE* as the number 50. This is because C has a feature known as the *preprocessor.* This allows you to define symbolic names which are replaced by a larger expression when found in the program by the compiler.

In these two programs, the terms *size* and *SIZE* refer to the size of the group of numbers in which you are looking for primes. This makes it easy to change the size of the group; you need only change the value of the size term.

In BASIC, the size value must be assigned to a variable—

even though its value stays constant throughout the program—because BASIC has no other symbolic way to represent a number. But in C, you can use the #define operator to create the symbol SIZE. Every time the compiler sees the word SIZE, it will substitute the number 50. This allows us to assign a symbolic meaning to SIZE, making the program easier to read without wasting storage space in our program by creating a variable for this constant value.

This example, however, gives only the smallest clue to the power of the preprocessor, which can be used for much more sophisticated types of substitutions.

Building Loops

When you compare the bodies of the programs, you find that there are only small differences. The first is that the form of the loop used by each language is somewhat different. The BASIC format declares a loop variable, the starting value of that variable, the terminating value, and an increment value, separated by the keywords TO and STEP. The increment parameter is optional and defaults to 1 if omitted.

In C, the starting condition, termination condition, and the condition repeated each time through the loop are enclosed in parentheses and separated by semicolons. Although in this particular program each condition is related to the variable *x*, it is interesting to note that in C, unlike BASIC, the three conditions do not all have to relate to the same variable. You could declare a loop that begins by setting the value of *y* to 0, and ends when *z* is equal to 50, changing the value of *x* each time through the loop. Also, it is possible to leave one or more terms empty; the expression *for(;;)* can be used to set up an endless loop.

The second difference is that BASIC uses the NEXT statement to mark the end of a FOR loop, while C expects the loop to consist of either a single statement or a compound statement enclosed in curly braces. This compound statement may be composed of any number of lines. The same is true of the *if* conditional statement. The compound if statement may stretch over several lines—unlike the BASIC if statement, which must take up only one line.

Likewise, where BASIC uses the WEND statement to define the end of the WHILE loop, C accomplishes the same thing by enclosing the whole body of its while statement within curly braces.

Screen Output

Another difference is the ways in which the two programs print their results. The C program uses a function called *printf( )*, which is not part of the language proper. Instead, printf( ) is part of the standard library of input/output routines that must be *linked* with the program after it is compiled. During the linking stage, these routines are attached to the object code generated by the compiler.

Printf( ) also is an example of a function that takes parameters. The text and variables upon which a function operates appear within the parentheses that follow the function name. The printf( ) function performs roughly the same task as the PRINT USING command in BASIC. The % and *d* characters specify that a decimal number is to be formatted, and the number 2 is used to specify that the numbers are to be printed with digits before the decimal place and none after. In BASIC, the PRINT USING template ## does roughly the same thing. However, the C printf( ) function allows for multiple substitutions, while separate BASIC statements would be required for each formatted column.

This example should make it pretty clear that once you get past the formal requirements of function names, curly braces, and variable declarations, C is not as strange as you might have thought. Of course, you should not take this to mean that C is just BASIC in disguise. C has a number of powerful features that distinguish it quite clearly from BASIC. But, thankfully, there are enough similarities so that beginning programmers can produce working code right away and can learn to take advantage of C's special features a little at a time.

C Shortcuts

For most BASIC programmers, C's extra features will be quite welcome. For example, C has a multitude of powerful math and logical operators. The statement x += num; may be less recognizable at first than x = x + num, but it requires a lot less typing over the course of a long program. C allows you to use either form.

As mentioned above, C has a number of features that let you pack a lot into one line. For example, you can make multiple assignments using the = operator. The statement

```
a=b=c=d=0;
```

is just fine in C. In most BASICs, it would require four separate statements.

Assignments can also be made to a value that is the result of a function, as well as to a constant value, as in this statement:

```
a=b=c=d=getchar( );
```

Here, *getchar( )* reads the character from the keyboard and assigns it to four different variables.

You can even make assignments at the same time you make comparisons. For example, the statement

```
if ((a=b)<c) DoThis( );
```

first assigns the value of *b* to *a*, then compares that value to *c*, and calls the function *DoThis( )* if the new value of *a* is less than that of *c*.

Admittedly, the C prime-number program shown earlier was to some extent written to look as much as possible like the BASIC program. Here is another version that is a bit more C-like:

```
/* Sieve1.c—Finds the prime numbers from 2 to SIZE */

#define SIZE 50
main( )
{
  int num = 2, x, count;    /* declare & initialize variables */
  char flags[SIZE+1];

      for (x = num ;x <= SIZE; x++)
          flags[x]=1;          /* set all flags */

      while (num++ <SIZE/2)
          for (x = 2*num; x <= SIZE;x += num)
              flags[x]=0;          /* eliminate multiples */

      for (x = 2; x<= SIZE; x++)
          if (flags[x])
              printf("%2d \n",x); /* print the primes */
}
```

This one takes several C shortcuts. First, the variable *num* is assigned a value in the line in which it is declared. As stated above, in C you can assign a value to a variable just about anywhere. Also, it uses the += operator as explained above. Finally, in three places it uses the ++ increment operator. With this operator, you can say x++ instead of x=x+1.

Note also that the ++ operator can be used to increment

one of the variables being compared as part of the condition of the *while* statement. The ++ following the variable num means that *after* the comparison has been made to determine whether the while loop should continue, the value of the variable should be increased by 1. If the ++ came before the variable name, its value would be increased *before* the comparison was made.

Learning C is a big step up from learning BASIC, though perhaps not as big a step as learning to program in machine language. But like any big programming job, writing a C program can be broken down into smaller, more manageable steps. Once you try C, you may discover that taking full advantage of the power of your Atari ST is not as difficult as you once thought.

Choosing a Compiler

Sheldon Leemon

The best version of C for your time and money offers a lot more than just a compiler. Consider the merits and shortcomings of the familiar packages presented here.

If you think you'd like to try C programming, you'll have to find the C compiler that suits you best. You have several versions of C to choose from on the ST.

Unfortunately, the process of comparing compilers is complicated by the fact that the compiler itself does not make up a complete C programming environment. In order to effectively program in C on the ST, you need not only a C compiler, but also a text editor, linker, command processor shell, resource file construction program, and documentation of the GEM (Graphics Environment Manager) and TOS (Tramiel Operating System) functions. You might also want an assembler, disassembler, debugger, program librarian, and make utility. Each of the C compilers currently being sold contains some, but not all, of these features. Therefore, to evaluate C compilers, you must consider how close each comes to providing everything that's needed to start programming on the ST.

C compilers for the ST are available from Alcyon, Megamax, MetaComCo, and GST. Another package, *Haba Hippo C*, has been discontinued, though copies are probably still available from existing stock. (Still another, *Mark Williams C*, has recently become available; this package is popular on the IBM PC.)

Among the compilers which have been around awhile, the two lowest-priced entries, *Haba Hippo C* and *GST C*, have some major drawbacks. *Hippo C* was the first C compiler available for the ST other than Atari's own development system (*Alcyon C*). *Hippo C* was lacking in such areas as floating-point math and GEM support, and had so many other problems that it was ultimately withdrawn from the market.

GST C is not nearly as problem-ridden as *Hippo C*, and in fact has many positive features. For a very low price it offers not only the compiler and linker, but also an assembler, text editor, and menu-driven command shell. It also offers complete GEM

support (though it doesn't include GEM documentation). The major problem with *GST C* is that it's not really a complete implementation of the C language. It lacks such major features as floating-point math, casts, and structures. This is not to say that you can't develop significant programs with *GST C*; GST reportedly used it in-house to develop *1ST Word*, the word processor included with every ST. But if you're just learning to program on the ST, you're confronted with adapting to both a new language and a complex operating system, and it's extremely difficult to work around the eccentricities of a nonstandard compiler at the same time.

Alcyon C: Wheat and Chaff

The first C compiler to appear for the ST was *Alcyon C*, which is included in the kit which Atari sells to software developers. The $300 developer's kit is a package deal, however, so you can't buy *Alcyon C* without also paying for everything else in the kit—mostly documentation. This makes *Alcyon C* the most expensive C package and probably the most extensive as well.

Atari's GEM and ST documentation represents both the best and the worst available. It's the best because it contains the most GEM information you can get in one place, and it's the worst because precious little of it was written specifically for the ST.

Take, for example, the GEM documentation. Basically it consists of poor photocopies of Digital Research's preliminary GEM manuals for the IBM PC, complete with 8088 machine language examples that have nothing to do with the 68000-based Atari ST. Atari didn't add any material on how GEM differs on the ST, nor did it try to eliminate the large quantity of irrelevant material that relates only to the PC. It's up to the reader to separate the wheat from the chaff. So while the documentation starts out as a stack of about 2000 loose sheets of paper, by the time you get rid of the IBM GEM installation manual, material on CP/M–68K, and more information than you'll ever want to know about the Kermit communications transfer protocol, you're left with a much smaller pile.

The good stuff consists mainly of the *Hitchhiker's Guide to the BIOS*, the *Line A Document*, a GEMDOS manual, some hardware specifications, and miscellaneous loose ends. This material is very helpful, but is incomplete, not entirely free from errors, and poorly organized (some of it exists only as disk files that you must print yourself).

Probably just as helpful as the printed documentation, or more so, is the support that Atari provides in the Atari Developers' Forum on the CompuServe Information Service. To reach this area, log on and type GO ATARIDEV. Atari representatives are available online to answer programming questions, and they also provide programming examples and timely updates to the manuals. Although this service goes a long way toward filling in the gaps left by the written documentation, it's not free.

The *Alcyon C* Compiler

Alcyon C consists of a three-pass compiler that generates machine language source code, plus an assembler. After you've run those four programs, you still have to put your object code through the linker and Relmod utility to convert it into a form that can be loaded by GEMDOS (GEM Disk Operating System). To make running these six programs a little more bearable, Atari includes a minimal batch utility program that lets you set up text scripts which describe a sequence of programs to be run with one command.

Although the batch utility makes compiling and linking more convenient, it doesn't do much to speed up the process. *Alcyon C* produces good results, but having to load and run so many programs makes it slower for development than any other C compiler. It's theoretically possible to run this compiler with just one single-sided drive, but it's not something you'll want to try if your time is worth more to you than 30 cents an hour. A hard disk drive—or better still, a very large RAM disk—is the only way to go with this package.

The compiler itself is solid and professional. It offers several compile-time options that can be invoked with flags in the command line, including one which specifies a search path for include files. It has good support for floating-point math, and the library of standard functions is quite adequate.

As usual with this package, however, the compiler documentation is not really specific to the ST version of *Alcyon C*. Instead, you get photocopies of the Digital Research CP/M–68K C documentation, along with Alcyon's generic Motorola development system manual. It's up to you to figure out what applies to the system you're working with. Nevertheless, all of the material is there, somewhere.

Valuable Extras

The *Alcyon C* package includes a large collection of auxiliary software. Digital Research's *Resource Construction Set* (RCS) is almost indispensable for creating GEM program resources such as menus and icons. For creating source code, there is the *MicroEMACS* editor, a non-GEM command-driven text editor that is also available in many public domain versions. The *AR68 Program Librarian* helps manage system library files. *SID* is a symbolic assembly-level debugger.

There's also a simple, usable-but-buggy command processor shell called COMMAND.TOS that operates something like the MS-DOS interface. (You may find the Michtron *DOS Shell* program more complete and reliable.) And to compensate for the disorganized documentation, the program disks include good source code for a sample GEM application and desk accessory.

In summary, the Atari developer's kit contains everything you need in order to write great GEM software, but finding it can sometimes resemble a high-tech adventure game.

Although Atari doesn't limit the sale of its developer's kit as some manufacturers do (Atari's definition of a developer is somebody who is willing to spend $300), the prospective buyer should exercise discretion. This package is mostly for those who are seriously dedicated to producing commercial applications. For those people, the resources in this package—especially Atari's technical and marketing support—are absolutely essential, and they far outweigh the problems associated with sketchy and poorly organized documentation.

Megamax C: Complete and Concise

The *Megamax C* compiler package provides a development system almost as complete as Atari's, but in a much more attractive and usable format and at a more affordable price ($200). The GEM documentation isn't as extensive as the Digital Research material from Atari, but that's mostly because it doesn't contain any extraneous information.

Instead, each GEM library call is summarized on its own page, complete with an example of the syntax, a full explanation of the function, and its input and output parameters. Brief overview sections provide a little insight into how to put the calls together. Similar concise explanations are offered for BIOS (Basic Input/Output System), XBIOS (eXtended BIOS), and GEMDOS routines. In addition, there are chapters covering system global

variables, keyboard codes, and system error codes.

In short, Megamax has taken all of the most useful ST information and summarized it in a convenient and attractive format, complete with a table of contents and index. The documentation for the compiler itself is also neatly laid out. Its most serious flaw is that there's no list explaining the compiler or linker error messages, which can make it quite difficult to figure out where you've gone wrong.

Megamax's previous 68000 compiler was created for the Macintosh, and, for good or ill, that experience has shaped the ST version of C. On the positive side, Megamax is obviously used to dealing with a mouse-driven windowing environment, and it shows in the way in which *Megamax C* takes advantage of the user interface.

For example, while it's convenient to use the GEM desktop icons to run a single program, it isn't so convenient when you have to edit, compile, link, and test an application over and over again. So Megamax provides a shell program from which you can easily edit, compile, link, and run the program you're writing. It even has a built-in make utility that lets you compile and link in one step. Moreover, when a compile fails, you end up back in the text editor with your source code and a list of the compile errors in separate windows.

The 32K Ceiling

But Megamax's Macintosh background also has some drawbacks. Macintosh programs use position-independent code, which limits program code and data segments to a maximum size of 32K. While Mac programmers are used to this by now, it seems to have thrown the ST world for a loop. Whenever I mentioned the Megamax compiler to any of its competitors, they almost always said the same thing: It's a nice, fast compiler for small applications, but it isn't really useful for serious work because it limits you to 32K programs.

Of course, if that were true, there wouldn't be any Macintosh software. The 32K limit on program segments means only that any single function must be less than 32K. To create programs larger than 32K, you simply string the 32K sections together.

Likewise, the 32K data-section limit means you can't declare an array with more than 32K of elements. You can, however, work with larger data arrays by using the *malloc* function to allo-

cate the memory, and then declaring a pointer to that memory block. So while these limitations may create slight portability problems, they don't really limit the usefulness of programs written with *Megamax C*.

As a matter of fact, the standalone version of *Thunder!*—a spelling checker from Batteries Included that keeps a 50,000-word dictionary in memory—was written with *Megamax C*. (The desk-accessory version was compiled with *Alcyon C*.) As an added bonus, the format in which *Megamax C* compiles its object code results in smaller and faster programs.

Swift Compilation

The *Megamax C* compiler itself is a fast and simple one-pass compiler. True, this simplicity does limit your options somewhat. For example, since there are no compile-time directives, all header files must either be in the same folder as the source file, or in the HEADERS folder within the MEGAMAX folder. The MEGAMAX folder, in turn, must be on the disk's root directory, which irritates some hard disk users. Also, *Megamax C* directs compiler error messages to a disk file without consulting you for your opinion.

In general, *Megamax C* is very compatible with *Alcyon C* source code. It uses 16-bit integers, which simplify GEM programming. As many published benchmarks have shown, the object code produced by *Megamax C* tends to be smaller and faster than that produced by *Alcyon C*—in some cases, significantly so.

A more important distinction is the time and trouble required to compile a program with each package. Unlike the large and unwieldy collection of programs required for *Alcyon C*, all of the necessary *Megamax C* programs fit neatly on one single-sided disk with room left over for source code. And the Megamax compiler works so quickly that it's actually much faster to compile and link a program using *Megamax C* with a floppy drive than it is to use *Alcyon C* on a hard disk. If you use *Megamax C* with a hard disk or RAM disk, it's almost like working with an interpreter rather than a compiler.

The auxiliary programs in the Megamax package are outstanding. It is the only C package besides Atari's that comes with a resource construction set. Resource files are all but essential to creating GEM applications that use drop-down menus, dialog boxes, and icons, and it is almost prohibitively tedious to create them manually. A resource construction set, therefore, is practically a necessity for serious GEM programmers.

The *Megamax C* linker is intelligent enough to load only the modules necessary to resolve external references in your source code, which reduces the size of executable object files. Also, because it automatically searches the system library, your command line merely has to specify the name of your object module.

Sorry, No Assembler

Along with the linker there's a librarian and a code improver that performs branch optimization. The text editor is nice, but probably too Mac-like for most ST users—it won't let you move the cursor with the cursor keys, and it's limited to 32K files. Of course, you're free to use any other standard ASCII text editor or word processor to create your source code.

There's no assembler in the Megamax package, and some would say that none is necessary since the compiler accepts inline assembly commands and thus doubles as an assembler. Still, some people like a compiler that generates assembler source code so they can optimize sections of the program. This isn't possible with *Megamax C* unless you use its disassembler to break down your program and then reconstruct it as source code.

Finally, the Megamax package contains even more example programs than Atari's, including the same application and desk accessory program source.

If you suspect that I favor *Megamax C*, you're right. Megamax has provided a complete ST development system for a reasonable price (as C compilers go). The only other viable C product I've seen that's cheaper is MetaComCo's *Lattice C*. While this is a good, full C compiler with an excellent standard library, it suffers from some serious problems. First, the *int* data type is 32 bits long instead of 16, which causes portability problems with *Alcyon C*. Second, it doesn't include a resource construction set, which puts a damper on GEM programming. And third, it doesn't include any GEM documentation. By the time you finish buying the extra books you need, your investment will equal the cost of *Megamax C*.

But the real clincher is the compiling time. Unless you're trying to finish *War and Peace* while your programs compile and link, you'll find *Megamax C*'s speed to be a lifesaver.

The C Programming Environment

Sheldon Leemon

Learn how to make the C programming environment work in your favor: You'll save yourself time and frustration as you make the transition from programming in BASIC.

For the person who has no prior C programming experience, the prospect of learning the language on the ST can be quite intimidating. Naturally, there's the problem of learning the syntax of the language. But at the same time, the programmer also must become familiar with the ins and outs of the various parts of the ST operating system—the GEM AES, VDI, GEMDOS, BIOS, and XBIOS. Calls to these collections of routines are necessary to implement the friendly icon and menu interface that GEM provides, since this kind of interface goes beyond the scope of the traditional C input/output scheme.

As difficult as these hurdles appear, the prospective C programmer faces an even more fundamental problem. He or she must first learn the mechanics of writing a C program, compiling it, and getting it to run. To someone coming from a BASIC programming background, this may sound strange. After all, if you want to write a BASIC program that puts the words "My program works!" up on the screen, you simply load up BASIC, enter the program line

10 PRINT "My program works!"

and then type RUN. In fact, you don't even have to enter a program. You can just type

? "My program works!"

in direct mode, and the words will appear on the screen. The advantage of an interpreted language like BASIC is that it's highly interactive. The program source code (the text that makes up the program) stays in memory at all times, there's always a text editor available for changing the source code, and when you want to see the result of the changes, all you have to do is type RUN.

Delayed Gratification

C, on the other hand, is a compiled language. It produces stand-alone programs that don't need an interpreter in memory in order to run. Such programs run much more quickly than interpreted BASIC programs, but they generally take longer to develop.

In C, the program text editor and the means of making the program executable aren't part of the same cozy program development environment (as they are in BASIC). Instead, the programmer must create a source code file with a text editor, and then run a series of programs that create a working program out of that file. If approached properly, this process can be almost as quick and easy as programming in BASIC. But if you concentrate only on learning the rules of C programming and don't spend the time required to set up a convenient programming environment, you may find that creating the simplest C program can be quite a chore.

Let's go through the steps required to create a C program which, like the BASIC program above, just prints the statement "My program works!" on the screen. The first step is to create a text file that contains the program source code. To do this, you'll need to run a text editing program.

Many C compilers include such a program. Atari, for example, includes Mark Williams's *MicroEMACS* editor with the *Alcyon C* compiler in its development package. The *Lattice C* compiler comes with MetaComCo's *ED* program, and the *Megamax C* compiler has a GEM-style editor with windows, drop-down menus, and a mouse-controlled cursor. There are also a number of public domain text editors, including variations on the *MicroEMACS* editor, that may be downloaded from bulletin boards and information services.

It may also be possible to create source code files with your favorite word processor. The file that's produced must be straight ASCII text, with no embedded printer-control characters. And, unlike most documents produced with a word processor, the file should include a carriage return at the end of every line. These requirements are met in *1st Word* when the word processor mode is switched off, but other word processors may require some maneuvering. With *ST Writer*, for example, you must print the text to a disk file.

Compiling the Source Code

Once you've started your editor from the GEM desktop, you must type in the text of the program. For this modest example, all you have to type is

```
main( )
{
    puts("My program works! \n");
}
```

This program creates a function called main( ) which contains a single statement, the purpose of which is to print a string of text on the screen. After you've finished typing it in, you must save the source code on disk. TEST.C will be the name of this file. (Most C compilers require that the name of source code files end with the .C extender.)

The next step is to compile the source code into object code. This process involves running a compiler program (or programs). The compiler reads the source code file, translates the C commands into equivalent machine language instructions, and creates another disk file to store these instructions.

The simplest case is that of a single-pass compiler, like *Megamax C*, which requires only one program to convert the source code to object code. To compile the test program with *Megamax*, you run the program MMCC.TTP (for *Megamax C* compiler). This is a TOS Takes Parameters type of program, which means that when you start it from the desktop, a dialog box appears with a dotted line for entering parameters. In this case, the parameters are instructions which tell the program what source code file to compile. With some compilers, these parameters may include optional instructions as to how the source code should be compiled, what the output file should be named, and so on.

For this example, enter the name of the source file, TEST.C, as the sole parameter. This means that you want the compiler program to use the source file TEST.C as input, and create the object code file TEST.O as output.

The Missing Link

Assuming that the program compiles successfully, you can go on to the final stage. (If it doesn't successfully compile, go back to step 1 with the text editor.) Some beginning C programmers find the next step a bit puzzling. For, if the compiler has already converted the program into machine language, why can't it be run?

The answer is that the object code file isn't a complete program. It's missing certain preparatory routines that perform housekeeping functions—setting up a program stack and certain pointers, and directing program execution to the function named main( ), which is where every C program starts. The program may also require the addition of some standard subroutines which aren't part of the C language proper.

For example, this sample program uses an input/output function called puts( ). This function is not an integral part of the C language. Like all other I/O functions in C, it is actually implemented as a separate program. A number of such programs are collected into an object code file (or files) known as the standard C *library.* Whenever a C program refers to an external function like puts( ), the compiler can only note its usage in the object code file, since it doesn't know the machine code instructions needed to perform the function. It's up to a program called the *linker* to pull those external programs out of the system library and join them with the object code, along with the startup code, to create an executable program. For this reason, the job of the linker is said to be *resolving external references.*

To link the *Megamax* object code file, you need to run a program called MMLINK.TOS. The parameter line to enter in the dialog box is

TEST.O −O TEST.PRG

which tells the linker to use the TEST.O file for input and to create the program file TEST.PRG as output. Since the only external routines needed for this program are located in the SYSLIB file, which the linker automatically checks, you don't have to include the name of any other object code files or library files with which to link TEST.O.

If the link process succeeds, you end up with the program file TEST.PRG, which can be run from the desktop by double-clicking its icon. This program prints the message "My program works!" briefly at the top of the screen before ending. At that point, you may wish to delete the intermediate file TEST.O.

Multipass Compilers

This may seem a lot of work for such a small program, but the Megamax compiler is actually the simplest to use. The most complex, the *Alcyon C* compiler that comes with Atari's developer's kit, requires several additional steps.

First of all, it is a three-pass compiler, which means that three separate programs are needed to process the source code. After having created the original source code file, you must run a program called CP68.PRG, which is the C preprocessor. Since all of the *Alcyon C* programs have the .PRG extender, you must install them as TOS Takes Parameters applications from the desktop to add a command line. After doing so, you can run CP68 with the command line TEST.C TEST.I.

After you've created the TEST.I file, run the parser program, C068.PRG, with the command line TEST.I TEST.1 TEST.2 TEST.3. This takes the TEST.I file and creates two more intermediate files, PRINT.1 and TEST.2. Next you run the code generator, C168.PRG, with the command line TEST.1 TEST.2 TEST.S. Unlike *Megamax C*, the *Alcyon C* compiler does not turn the C source code directly into machine language instructions. Instead, it creates an assembly language source code file—in this case, TEST.S, which must be run through an assembler to be converted into machine code.

So the next step is to run the assembler, AS68.PRG, with the command line −L −U TEST.S, which creates the object code file TEST.O.

This brings you to the linking stage. Run the linker, LINK68 .PRG, with the command line [U] TEST.68K=GEMSTART, TEST.O, GEMLIB. This creates a file called TEST.68K using the startup code from the file GEMSTART, the program code from TEST.O, and the C library code from the file GEMLIB. Unlike *Megamax C*, which uses only one large library file for all of the C libraries, *Alcyon C* breaks them down into files like GEMLIB, VDIBIND, AESBIND, OSBIND, and LIBF. Which of these you must include in your link depends on which kinds of functions (GEM AES, GEM VDI, and so on) you use in your program.

It may seem that once you've performed the link you should be finished, but there's one more step. The linker provided with *Alcyon C* creates a relocatable load file in a format that is incompatible with the ST. This load file must be modified with a program called RELMOD.PRG. In this case, run RELMOD with the command line TEST, which means that it takes the file called TEST.68K as input and produces a file called TEST.PRG.

It is TEST.PRG that you finally run to print the message briefly on the screen. Once you've done that, you'll probably want to clean up all of the intermediate files that you've left lying around—TEST.I, TEST.1, TEST.2, TEST.S, TEST.O, and TEST.68K.

Menu-Driven Shells

As you can see, if you had to run each of these programs from the desktop and remember the proper command line to enter for each, you might soon lose your enthusiasm for C altogether. Fortunately, there are programs which allow you to delegate these boring, repetitive tasks to the computer.

The first type is a menu-driven shell program. Both the *Megamax C* and *Lattice C* compilers include this type of program, and the shell *Menu+*, packaged with the latter, is also available separately. These programs allow you to associate the various programs used in the development process with menu items on the shell window. Then, when you want to run one of the programs, all you have to do is select the menu item.

When you run the compiler from the *Megamax* shell, a file-selector window pops up which lets you click on the name of the source code file to compile. You may also set up default command lines. The *Megamax* shell lets you use a simple *make* facility, in which the compilation and linking processes are run according to a script that you save in a file. For example, the script file

```
TEST.C
MMLINK TEST.C -O TEST.PRG
```

gives the shell instructions to compile and link your test program in just one step. And if there were an error in the compilation process, the shell would automatically switch back to the editor program and load both the source file and the error file showing what the compiler problems were.

Batch-Processing Shells

The Atari development package takes a slightly different approach. It includes a program called BATCH.TTP that emulates the batch-processing capabilities of operating systems like MS-DOS. Batch processing means that you can put a batch of command lines in a text file, and the operating system executes them one after the other automatically. It also includes the concept of parameter substitution. This means that you can type in filenames on the command line of the batch program, and these names may be substituted for parameters in the batch file.

The advantage of parameter substitution is that the same batch file can be used to perform the same general operation on different input files. When you want to specify that a name

should be substituted by one that is entered on the command line, you place a percent sign followed by a number in the batch file.

For example, if you put the command CP68 %1.C %1.I in the batch file, the %1 parameter is replaced by the first parameter on the batch file command line. If the batch file is called C and the input file you want compiled is MYPROG, you use the command line C MYPROG, and the batch program will read the line CP68 %1.C %1.I as CP68 MYPROG.C MYPROG.I. If the command line is changed to C YOURPROG, the batch program interprets it as CP68 YOURPROG.C YOURPROG.I.

In this way, you can build up generalized command files. For example, the following CC.BAT file contains all of the steps needed to compile and link a C program:

```
CP68 %1.C %1.I
C068 %1.I %1.1 %1.2 %1.3
RM %1.I
C168 %1.1 %1.2 %1.S
RM %1.1
RM %1.2
AS68 -L -U %1.S
RM %1.S
LINK68 [U] %1.68K=GEMSTART,%1,
  VDIBIND,OSBIND,AESBIND,LIBF,
  GEMLIB
RM %1.o
RELMOD %1
RM %1.68K
```

If you run the BATCH.TTP program with the command line CC TEST, it automatically creates a TEST.PRG file from your TEST.C source file. The RM commands refer to a program that ReMoves (deletes) the unwanted intermediate files after the compiler has finished with them.

Command Line Shells

From a batch-processing program, you're just a short step to using a command line shell. Such a program provides the type of command line interface found in UNIX or MS-DOS. Instead of clicking on files to run them, you type their names. Instead of dragging files to the trash can to delete them, you type in a command like RM or DEL.

Command shells generally include batch file processing, so if you type a filename that ends in the extender .BAT, the shell

automatically executes the commands contained in that text file. For example, if you have commands in a file CC.BAT, you type CC to execute them. If the batch file uses parameter substitution, you add the command line to the end of the filename (for example, CC FILE1 FILE2).

There are many such command shells available, both commercially and in the public domain. In fact, the CC.BAT file shown above would work well with *ST-Shell* (see Chapter 4) since it contains its own version of the RM command.

Optimizing for Speed

Another important part of setting up a C programming environment is organizing your work disks. At the very least, you should try to set up your work disks so that all the programs you need are on one floppy disk. This avoids time-consuming disk swaps. But for real efficiency, put all of the necessary programs on a hard disk or a RAM disk.

A hard disk offers a lot of speed and convenience, but it also requires the outlay of a fair amount of cash, something not everyone has. But with the ST's large amount of memory, everyone should be able to use a RAM disk. RAM disk programs (available both commercially and in the public domain) make the computer think that a section of its random access memory is a disk drive. Since a RAM disk isn't subject to the mechanical limitations of a real disk drive, it's very fast—as fast as, or faster than, the best hard disks.

Even with 512K of RAM, you should have enough memory for a fair-sized RAM disk. But with a one-megabyte 1040ST or an expanded 520ST, you should have enough room to put all of your files in RAM. This can make a huge difference when you're compiling and linking programs.

Keep in mind, though, that a RAM disk disappears when you pull the plug. Try to set up your batch file to write a copy of your finished program, as well as any altered source files, to floppy disk just in case.

As you've seen, compiling a C program can be a laborious, time-consuming process that involves running many programs and entering many command lines by hand. Or, it can be a matter of typing a single command that sets a series of events in motion at lightning speed. It's all a matter of how you set up your programming environment. So don't be in such a hurry to start writing code that you doom yourself to needless drudgery. Take the time needed to arrange things so that each compile takes the least time and the least effort possible. Each second you knock off the compiling process through careful planning will be saved hundreds of times over as you program in the coming months.

Appendix

How to Use the Disk

To use the disk, simply insert it in a drive and click on the appropriate file-drawer icon to display the directory window. If you wish, you may boot up your ST with this disk by inserting it in drive A and switching on the computer, but normally it contains no active desk accessories.

There are two ways to access programs and files on the disk. You can simply run or examine the files from the GEM desktop as usual. Or you can use the custom disk menu program on the disk that contains descriptions of each file as well as special instructions. To run the menu program, double-click on the file named DISKMENU.PRG. It works in all screen modes, color or monochrome.

DISKMENU.PRG displays a directory of files on the disk, one screen at a time. Click on the lower buttons labeled *Prev* or *Next* to display the previous or next screen.

At the top of the disk menu are three buttons labeled *Description, QUIT,* and *Run program.*

The *Description* button calls up a screen which describes the program or file. At the bottom of this screen are the filename and two buttons labeled *MENU* and *RUN*. Clicking on the *MENU* button returns you to the disk menu. Clicking on the *RUN* button loads and runs the program. However, if this particular file is not an executable program (for example, a source code or data file), the *RUN* button is dimmed and disabled.

You can also run a program directly from the disk menu by clicking on the *Run program* button at the upper right of the screen. However, if this particular file is not an executable program, you'll be alerted to this fact.

Note that several files on the disk require special instructions or an explanation; please refer to the corresponding article in the book before attempting to run a program or access a file.

Clicking on the *QUIT* button on the disk menu returns you to the GEM desktop.

There are four files on the disk which are required for the disk menu program: DISKMENU.PRG, DISKMENU.RSC, MONOMENU.RSC, and CONTENTS.MAR. These files do not appear on the disk menu itself. Do not delete them if you intend

to use the disk menu. If you plan to use the disk menu, be sure these files are copied when you back up the disk.

Our disk is not copy-protected. You are encouraged to make a backup of the disk as soon as possible. However, the contents of the disk are copyrighted and may not be used by anyone other than the owner of *COMPUTE!'s Second Book of Atari ST.* Since the writers and programmers whose work appears on this disk are paid, in part, according to the volume of sales, we ask that you respect the copyright.

Index

= operator 173–74
/* (characters). *See* slash and asterisk
Alcyon C 151, 177–79, 181
application, installing 147
ASCII 69
assembly language 163
AS68 151
"AstroPanic!" annotated listing 17–33
Atari Corporation's bulletin board system 57
attributes, file 77–78
BASIC 163
batch file 121
batch processing 188
batch-processing shells 188–89
BIOS 179
boot disk 63
braces 170
cold start 60, 128
command line shells 189–90
compiled languages 166, 184
compiler 176–82
COMPUTE!'s Second Book of Atari ST Disk, using. *See* disk, how to use
Control Panel desk accessory 61, 142–43, 148
DEGAS 34, 125
desk accessory 59, 63, 127, 147–49
"Desktop Clock," installing 60
disk drive 145–46
disk, how to use 195–96
DOS Shell 179
errors, exception 103–5
exception 102–5
exception processing 102
exclusive OR (XOR) 15
folder 63
function 165, 170
function definition 170
GEM (Graphics Environment Manager) 47, 59, 75
  desktop directory, how files are arranged 143–44
  disk operating system (GEMDOS) 178
Haba Hippo C 176
hexadecimal numbering system 40
icons 146–47
indentations 170
Install Printer accessory 142
interpreted language 166
interrupt 103
Lattice C 182
library 186
line-continuation character 122
line numbers 169
linker 186
loops 172
machine language 163–64
main( ) 170
Mark Williams C 176
Megamax C 179–82
memory buffers 14
memory form definition block (MFDB) 14
menu-driven shells 188
musical instrument digital interface (MIDI) 91
NEOchrome 34, 125
"NEOview" commented C source code 152–53
"NEOview" machine language source code 153–56
parameter 81, 170
parsing 64
pathname 76, 81
preprocessor 171–72
prime number 168
"Prime.BAS" 169
"Prime.TOS" 168–69
pseudocode 165
RAM disk 190
remark 170
resource construction set (RCS) 179
resource file 47
ROM 163
root directory 63
RS-232 port 141–42
screen output 173
screen resolution 144
semicolon 169, 172
slash and asterisk 170
source code 183
source code, compiling 185
sprites 14–17
structured language 165
ST-Shell 80
  batch files 121–22
  Command Set (table) 115
  command syntax 116
  disk commands 116–18
  miscellaneous commands 120–21
  passing parameters 122–23
  preparing 111–12
  screen 113–14
  screen commands 118–20

text editor 166–67, 184
TOS operating system 63, 76, 80, 111, 151
TOS Takes Parameters (TTP) 76
Tramiel Operating System (TOS) 76
trash can 145–46
TTP 76, 150, 170
variables, defining 170–71
vector 103
Vectors, Preassigned (table) 104
Vectors Used by the ST (table) 104
virtual device interface (VDI) 14
vrocpyfm 14
warm start 60, 128
window 144–45, 157
word-wrapping 64
XBIOS 179
"XDIR" 75–78

COMPUTE! Books

Ask your retailer for these **COMPUTE! Books** or order directly from **COMPUTE!**.

Call toll free (in US) **800-346-6767** (in NY 212-887-8525) or write COMPUTE! Books, P.O. Box 5038, F.D.R. Station, New York, NY 10150.

Quantity	Title	Price*	Total
_____	Machine Language for Beginners (11-6)	$16.95	_____
_____	The Second Book of Machine Language (53-1)	$16.95	_____
_____	COMPUTE!'s Guide to Adventure Games, Revised (67-1)	$14.95	_____
_____	Computing Together: A Parents & Teachers Guide to Computing with Young Children (51-5)	$12.95	_____
_____	COMPUTE!'s Personal Telecomputing (47-7)	$12.95	_____
_____	BASIC Programs for Small Computers (38-8)	$12.95	_____
_____	Programmer's Reference Guide to the Color Computer (19-1)	$12.95	_____
_____	Home Energy Applications (10-8)	$14.95	_____
	The Home Computer Wars: An Insider's Account of Commodore and Jack Tramiel		
_____	Hardback (75-2)	$16.95	_____
_____	Paperback (78-7)	$ 9.95	_____
_____	The Book of BASIC (61-2)	$12.95	_____
_____	The Greatest Games: The 93 Best Computer Games of All Time (95-7)	$ 9.95	
_____	Investment Management with Your Personal Computer (005)	$14.95	_____
_____	40 Great Flight Simulator Adventures (022)	$10.95	_____
_____	40 More Great Flight Simulator Adventures (043-2)	$12.95	_____
_____	100 Programs for Business and Professional Use (for IBM PC and Apple Computers) (017-3)	$24.95	_____
_____	From BASIC to C (026)	$16.95	_____
_____	The Turbo Pascal Handbook (037)	$14.95	_____
_____	Electronic Computer Projects (052-1)	$10.95	_____
	Flying on Instruments with Flight Simulator		
_____	perfect bound (091-2)	$ 9.95	_____
_____	wire bound (103-X)	$12.95	_____
	Jet Fighter School		
_____	perfect bound (092-0)	$ 9.95	_____
_____	wire bound (104-8)	$12.95	_____
_____	The Complete Desktop Publisher (065-3)	$21.95	_____
_____	I Didn't Know You Could do *That* with a Computer! (066-1)	$14.95	_____
_____	Flight Simulator Adventures: For the Macintosh, Amiga, and Atari ST (100-5)	$12.95	_____

* Add $2.00 per book for shipping and handling. Outside US add $5.00 air mail or $2.00 surface mail.

NC residents add 5% sales tax. _____
NY residents add 8.25% sales tax _____
Shipping & handling: $2.00/book _____
Total payment _____

All orders must be prepaid (check, charge, or money order).
All payments must be in US funds.
☐ Payment enclosed.
Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. _____ Exp. Date _____
(Required)

Name _____

Address _____

City _____ State _____ Zip _____

*Allow 4–5 weeks for delivery. Prices and availability subject to change. Current catalog available upon request.

A Winning Combination

Whether you and the ST are an old team or a new one, you'll like the way *COMPUTE!'s Second Book of Atari ST* has you working—and playing—together. This book/disk combination eliminates typing and lets you dive right into a unique collection of programs and information. And the variety of games, special applications, utilities, and programming tips makes this a resource you'll reach for time and again.

Here's a sample of what's inside:

- *Laser Chess™*, the spectacular $10,000 First Prize winner in *COMPUTE!'s Atari ST Disk & Magazine* programming contest.
- *ST-Shell™*—a full command line interpreter—offers batch file processing and an efficient alternative to GEM.
- "AstroPanic!"—your opportunity to save the world from a frenzied alien assault. Commented source code documents a C program operating at machine language speed.
- Using "ST-Graph," you'll get a clearer perspective on home and business facts and figures.
- "Desktop Notepad" features word processing capabilities no matter what kind of application you're already running.

Plus:

- Disk utilities for complete control of file access.
- Extended applications for popular commercial drawing programs.
- A word count accessory to gauge your writing progress.
- Fundamentals of programming in C.

That's just a sample of the quality ST material collected here. Combine that quality with clear explanation and helpful illustrations, and the result is an ideal complement to the powerful ST.

All of the programs from the book are on the accompanying disk; all have been carefully tested to insure that they will execute smoothly. So when you've bought *COMPUTE!'s Second Book of Atari ST* and hurried back to your computer, there'll only be one thing left to do: Enjoy.

ISBN 0-87455-098-X

COMPUTE!'s Second Book of ATARI ST

COMPUTE!
Books