



ADD 512K RAM TO YOUR 1 MB A500 (FOR ABOUT \$30)

AC's TECH AMIGA

For The Commodore

Volume 1, Number 2
US \$14.95 Canada \$19.95

CAD APPLICATION DESIGN

Implementing an On-line Help Facility

Intuition and Graphics in ARexx Scripts

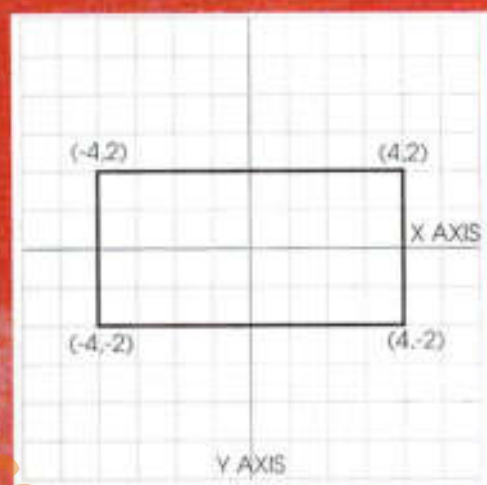
Programming the Amiga's GUI in C

Using Boolean Gadgets
from Absoft's FORTRAN

UNIX and the AMIGA

Interfacing Your Assembly Language
Program to ARexx

Introduction to 3-D Graphics
Programming



Introducing *Grand Slam* The Ultimate AMIGA Multi-Function Card

Trumpcard Professional Gate Array
achieves SCSI transfer rates up to 1.9
Mbytes/sec as measured by DPERF2.

Memory Controller Gate Array
includes memory arbitration and smart
refresh logic to guarantee lowest
possible power consumption...less
than 0.9 amps with 8 megs!

Fast RAM expansion in
0, 2, 4, 6, or 8 Mbyte
increments using easy
to install SIMM
memory modules.

Multi layer PC board for noise-free
operation.



INTERACTIVE VIDEO SYSTEMS

High quality RGB output for your Amiga

These images are **completely unretouched** photos taken from a stock 1084s RGB monitor. They are pure RGB, not smeary composite. No other graphics expansion device offers so much performance and costs so little! And all the software to run it is **free**. Even upgrades! There's not enough room to cover all the great features of this system, so here are just a few.

System Features:

- PaintL reader, ext ip a/w
- 18/24 bit "pure" modes
- 256/512 color register modes
- RGB pass through
- Screen overlay/underlay
- Screens pull up/down & go front/back
- View with any IFF Viewer
- Animate via ANIM or Page Flipping
- Works with Digiview™
- Completely blitter-compatible
- NTSC encoder compatible
- PAL & NTSC compatible
- Uses **only** RGB port
- FCC Class B, UL Listed
- Works w/ all Amiga monitors
- Does **not** use Amiga power

Paint Features:

- Custom brushes use blitter
- RGB, HSV, HSL, CMY palette
- RGB and HSV spreads
- Extensive ARexx™ support
- 10 Color Cycle/Glow ranges
- Range pong, reverse, stop
- Smooth zoom, rotate or scale
- Area, Edge, outline fill, overfill
- Dithered 24 bit fill mixing
- Anti-alias with any tool or brush
- Loads, shows GIF™ **exactly**
- "C" source code available free
- Upgrade from BBS 24 hrs/day
- Color or 256 greys painting
- 256 color stencils
- Matte/color anti-alias/cycle draw
- Prints via printer device
- Auto enhance and IFF palettes
- Writes IFF24, GIF™ HAM-E

Image Compatibility:

- 24 bit IFF, 24 bit IFF with CLUT chunks; 2 to 256 color standard IFF, half bright, HAM, DHB and GHT trace; RGB8 and RGBN; Targa™; GIF™; Dynamic HDRes™; SEAM, ARZO, ARZ1, AHAM, 18 bit ScanLab™, UPBS brushes; All of the 12 different HAM-E format image tile types.
- Images may be scaled and converted to 24 bit IFF files.
- Image processing software supplied provides edge enhancement, blur, various convolutions, and much more.

BLACK BELT SYSTEMS

Call (406) 367-5509 for more information. 398 Johnson Rd., Glasgow, MT 59230
SALES: (800) TR-AMIGA International Sales: (406) 367-5513
BBS: (406) 367-ABBS FAX: (406) 367-AFAX

HighView™ New Tek. Amiga™ Commodore Business Machines™ IFF™ Computerware™ Dynamic HDRes™ NewTek™ ScanLab™ AMIG, ARZO, ARZ1™ AMIG, HAM-E™ Black Belt Systems.
Eight Image Copyright True Vision. 1994-1995 Commodore, AHAM, ARZ1, ARZ2™ AMIG, HAM-E™ Black Belt Systems.

Circle 118 on Reader Service Card

What are you missing if you don't have the AC's TECH Premiere Issue?

Advanced Dissassembling: Magic Macros with ReSource
AmigaDOS, EDIT, and Recursive Programming Techniques
Building the VidCell 256 Grayscale Digitizer
An Introduction to Interprocess Communication with ARexx
An introduction to the ilbm.library
Creating a Database in C, Using dBC III
Using Intuition's Proportional Gadgets from FORTRAN 77
FastBoot: A Super BootBlock
AmigaDOS for Programmers
Adpating Mattel's Power Glove to the Amiga



Don't Miss Out!

AC's TECH Premiere Issue is in short supply, so order your copy today!

Call 1-800-345-3360 from anywhere in the US or Canada.
(credit cards only, please)



Half length card allows space for "hard card" configurations.

Dual 50 pin SCSI connectors for flawless SCSI data transfers even at top speeds.

Output only parallel port connects to printer allowing simultaneous operation of printer and any audio or video digitizer connected to Amiga's parallel port.

SCSI ID jumpers for use in exclusive IVS SCSI-SHARE SCSI networking environments.

Grand Slam includes card, disk mounting brackets, cables, TCUTILS 2.0 hard drive formatting/network configuration utility, comprehensive memory test software and parallel port configuration/patcher utility software.

Grand Slam 500 is available for Amiga 500 owners.

Naturally, Trumpcard, Trumpcard 500, Trumpcard Pro and Trumpcard 500 Pro owners can upgrade. Call IVS for details.

Amiga is a Trademark of Commodore Business Machines

Grand Slam List Price \$349.95

7245 Garden Grove Blvd., Suite E • Garden Grove, CA 92641
Voice: (714) 890-7040 • Fax: (714) 898-0858

Circle 140 on Reader Service Card

Contents

Volume 1, Number 2

- 8 CAD Application Design: Part I—World and View Transforms** *by Forest W. Arnold*
A detailed look at the mathematics and programming techniques used in CAD system design. Use these basics to construct the building blocks of a two-dimensional CAD program.
- 22 Interfacing Assembly Language Applications to ARexx** *by Jeff Glatt*
How to add an ARexx implementation to a program, demonstrated by adding a complete ARexx implementation to a 68000 assembly language paint program.
- 30 Adding Help to Applications Easily** *by Philip S. Kasten*
Implement a context-sensitive 'on-line' help facility in your applications using this powerful, yet easy-to-use, arsenal of functions...Just call help()!
- 48 Programming the Amiga's GUI in C—Part I** *by Paul Castonguay*
Getting started in C programming on the Amiga. Includes a presentation of the first programming concept in the Amiga Intuition environment: the opening of libraries.
- 60 Intuition and Graphics in ARexx Scripts** *by Jeff Glatt*
Using the ARexx function library, rx_intui.library, which adds a few dozen ARexx commands that allow an ARexx script to utilize Intuition and Graphics library routines.
- 64 UNIX and the Amiga** *by Mike Hubbart*
An introduction to UNIX for the Amiga programmer.
- 70 A Mega and a Half on a Budget** *by Bob Blick*
Add 512K of RAM to your 1 MB Amiga 500 for about \$30!
- 76 Accessing Amiga Intuition Gadgets from a FORTRAN Program**
Part II—Using Boolean Gadgets *by Joseph R. Pasek*
Using a direct interface to the Amiga's ROM Kernel to access Intuition boolean gadgets. Use these techniques to create a Jupiter's Moons' Simulator.
- 88 ToolBox Part I: An Introduction to 3-D Programming** *by Patrick Quaid*
The first in a series of articles dedicated to presenting the solutions to common Amiga programming problems. This time, we look at 3-D programming concepts.

Departments

- 5 Editorial**
51 Source and Executables ON DISK!
67 List of Advertisers


```
printf("Hello");
```

```
print "Hello"
```

```
JSR printMsg
```

```
say "Hello"
```

```
writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga.

Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PIM Publications, Inc.
One Currant Place
Fall River, MA 02722

AC's TECH / AMIGA

ADMINISTRATION

Publisher:	Joyce Hicks
Assistant Publisher:	Robert J. Hicks
Circulation Manager:	Doris Gamble
Asst. Circulation:	Traci Desmarais
Corporate Trainer:	Virginia Terry Hicks
Traffic Manager:	Robert Gamble
International Coordinator:	Donna Viveiros
Marketing Manager:	Ernest P. Viveiros Sr.
Marketing Associate:	Grag Young
Programming Artist:	E. Paul

EDITORIAL

Managing Editor:	Don Hicks
Editor:	Ernest P. Viveiros, Jr.
Associate Editor:	Elizabeth Fedorzyn
Hardware Editor:	Ernest P. Viveiros Sr.
Technical Editor:	J. Michael Morrison
Technical Associate:	Aimée B. Abren
Copy Editors:	John Rozendes Jeff Gamble Paul Larrivee
Video Consultant:	Frank McMahon
Art Director:	William Fries
Photographer:	Paul Michael
Illustrator:	Brian Fox
Research & Editorial Support:	Melissa A. Torres

ADVERTISING SALES

Advertising Manager: Donna Marie

1-508-678-4200
1-800-345-3360
FAX 1-508-675-6002

SPECIAL THANKS TO:

Richard Ward & RESCO
Steve at Printer's Service
Swansea One Hour Photo
Pride Offset, Warwick, RI
Mach 1 Photo

AC's TECH For The Commodore Amiga™ (ISSN 1053-7929) is published quarterly by PIM Publications, Inc., One Currant Road, P.O. Box 869, Fall River, MA 02722-0869.

Subscriptions in the U.S., 4 issues for \$44.95; in Canada & Mexico surface, \$52.95; foreign surface for \$56.95.

Application to mail at Second-Class postage rates pending at Fall River, MA 02722.

POSTMASTER: Send address changes to PIM Publications Inc., P.O. Box 869, Fall River, MA 02722-0869. Printed in the U.S.A. Copyright© 1990, 1991 by PIM Publications, Inc. All rights reserved.

First Class or Air Mail rates available upon request. PIM Publications, Inc. maintains the right to refuse any advertising.

PIM Publications Inc. is not obligated to return unsolicited materials. All requested returns must be received with a Self Addressed Stamped Mailer.

Send article submissions in both manuscript and disk format with your name, address, telephone, and Social Security Number on each to the Editor. Requests for Author's Guides should be directed to the address listed above.

AMIGA™ is a registered trademark of Commodore-Amiga, Inc.

STARTUP-SEQUENCE

Amiga in the Spotlight

The Amiga and the Video Toaster have been stirring up a lot of interest lately in the MS-DOS and Macintosh communities! It appears that NewTek and the Video Toaster attracted big crowds at a recent MacWorld Expo. Also, there have been several small articles in various MS-DOS trade publications, including a mention in John Dvorak's column in *PC Magazine*. This is fantastic, but there's a small quirk. These people, who are awed by the Amiga's capabilities, really don't recognize the Amiga itself. Instead, they refer to the Amiga/Video Toaster combination as a video computer that comes with an Amiga, or a great video peripheral that's soon to be controlled from an IBM PC. Crazy, right?

Right! But that's OK. Let the MS-DOS community think whatever it wants. The fact is that it is being mesmerized by the only hardware platform that could support the Video Toaster or anything like it—the Amiga! Now this is where it gets good. Some of these big cash MS-DOS software vendors are going to want part of the Video Toaster action (that's market share to the pinstripers). So they get their best, hot-shot programmers, give them an Amiga and a stack of books and technical notes, and let them go. This is where Joe-MS-DOS-hot-shot programmer discovers the Amiga's powerful operating system. He shows the suits some snazzy Amiga programming, the suits see dollar signs, and the company dumps a lot of money into Amiga software development.

This is good for everyone involved, because it creates a larger market. The larger the market, the more potential customers to buy your software. The more software you sell, the more development you can invest in to create new products. Your new products create a larger market, and so on. It's a rolling snowball, and it's a win-win situation.

DISCOVERIES

Everyone's heard of CDTV. This has got to be one of the most awaited Amiga products of all time (after the Toaster, of course.) Commodore, with Nolan Busnell, is nursing this baby to make sure nothing goes wrong. It's sure to be a best seller.

Many Amiga programmers and developers don't realize the incredible opportunity that CDTV provides. With the CDTV will come an enormous customer base just looking to buy software for their new high-tech home appliance. Educational software is sure to be a big seller, along with entertainment and home resources software. The market is wide open.

Plus, there are several developmental advantages to CDTV. First, the user can't illegally duplicate and distribute the disc. No piracy. Also, the duplication of the discs is very cost-effective. There have been rumors of some duplication houses charging a \$500-\$600 mastering fee, and \$.75/piece for duplication.

The disadvantages are few. There are very strict style guidelines for producing CDTV software, including the fact that windows are not allowed. This simple fact means that most software cannot be just ported, but will require a little redesign. Kudos to Commodore for finally putting some style control on the Amiga software. It's definitely a necessary evil.

For you techies who want to get a jump on CDTV technology, Commodore reports that CDTV's will be for sale at *The World of Commodore*, New York (April 5-7). Get 'em while their hot!

The Amiga's future looks bright. Be a part of it. For more information on Commodore's developmental programs, contact:

Commodore Business Machines
CATS—Developer Programs
1200 Wilson Drive
West Chester, PA 19380

Sincerely,



Ernest P. Viveiros, Jr.
Editor

AC's TECH MessagePort

Dear AC's TECH,

Congratulations on your first issue of AC's TECH for the Commodore Amiga! I loved it! My favorite articles were the 'ilbm.library' and 'AmigaDOS for Programmers.' I originally wrote a file-identification program called What Is in Modula-2 and after getting SAS/C I was excited about converting it to C. The 'Introduction to the ilbm.library' article has been an extreme help in converting my code, and now I am going to make full use of this shared library. The 'AmigaDOS for Programmers' story was a nice introduction to file locks. Thanks for the help!

I liked the VidCell digitizer hardware project article. Though I am completely hardware-illiterate and did not understand it all, I liked the reading anyway. Perhaps in the future you could include an article that is a brief explanation of hardware concepts so that this sort of article would be easier to follow, and maybe even give me the courage to try on my own.

What I liked best about the magazine was the on-disk examples. Having this code was a great way to learn and useful as a springboard for my own programs.

In response to the 'To 2.0 or not to 2.0' quandary, I suggest this: why not develop two versions? Each piece of software could easily be developed in 1.3 and then include an enhanced version for 2.0. Both versions could be included in the same box. This would be like the 68000 and 68020/030 versions now available for some programs.

I love the magazine, I love the info, and I love the style! I can't wait for the next issue, keep up the good work.

Ryan Sheftel
Rosemont, PA

Dear Ryan,

Thank you for writing! Your enthusiasm typifies the average AC's TECH reader. We 'techies' continue to be excited about the Amiga! We aren't the type to jump on the most-popular-computer-of-the-year bandwagon—but we've stuck loyally with the Amiga, even through the tough times. In a sense when the Amiga makes the big win, you won't be sorry!

In regards to your idea about packaging 1.3 and 2.0 versions together. I think this is a good idea...as a short-term solution. Everyone's favorite workhorse program has to work

under AmigaDOS release 2 before we even consider the system software upgrade. However, the long-haul plan is not only to update the 1.3 programs to run (bug-free, of course) under AmigaDOS release 2, but to redesign the application to take advantage of the new features that release 2 brings the programmer. Let's show what the Amiga can do!

—EPVjr

Dear AC's TECH,

I have recently purchased the Premiere issue of AC's TECH, and was quite surprised with its content. By its description, I had assumed it would be a technical guide to using and controlling the Amiga. I didn't expect about 30-50% of the magazine to be about programming and the rest on 'how the Amiga does this, that, and the other thing', but alas it was 90% programming. I did read and enjoy it a great deal, even though I am NOT a programmer.

I have read and enjoyed Amazing Computing and will continue to do so in the future, but I would really like to see some articles about board specifications, the exact boot-strap process, is BUSTER a 32-bit chip?, how does the auto-configure process work?, etc...

I know I will be creating my application software in the near future, and your magazine will help me along.

Greg Bastow
Surrey, BC Canada

Do you have a comment about AC's TECH or anything else in or around Amiga programming/development? Redirect your voice to your pen & paper port, then mail your message to:

AC's TECH MessagePort
P.O. Box 869
Fall River, MA 02722-0869

If we publish your letter, you will receive a certificate good for five free public domain disks from PiM Publications, Inc.

(All letters become the property of PiM Publications, Inc. The AC's TECH editors reserve the right to edit all letters for length and clarity.)

Turn up a Volume!

**The Complete Amazing Computing library
which now includes Volume 5**

is available at incredible savings of 50% off!

◆ **Volume 1 is now available for just \$19.95*!** ◆

(A \$45.00 cover price value, the first year of AC includes 9 info-packed issues.)

◆ **Volumes 2, 3, 4, & 5 are now priced at just \$29.95* each!** ◆

(Volumes 2,3, & 4 include 12 issues each, and are cover priced at \$60.00 per volume set.)

Subscribers can purchase freely redistributable disks at bulk rate discount prices!
This unbeatable offer includes all Fred Fish, AMICUS, and AC disks**

(see the *Fall/Winter'90 AC's Guide* for a complete index of all current freely redistributable disks).

Pricing for subscribers is as follows:

- ◆ 1 to 9 disks: \$6.00 each
- ◆ 10 to 49 disks: \$5.00 each
- ◆ 50 to 99 disks: \$4.00 each
- ◆ 100 disks or more: \$3.00 each

(Disks are priced at \$7.00 each and are not discounted for non-subscribers)

**To get FAST SERVICE on volume set orders, freely redistributable disks,
or single back issues, use your Visa or MasterCard**

call 1-800-345-3360

Or, just fill out the order form on page 98.

* Postage & handling for each volume is \$4.00 in the U.S., \$7.50 for surface in Canada and Mexico, and \$10.00 for all other foreign surface.

** AC warrants all disks for 90 days. No additional charge for postage and handling on disk orders. AC issues Mr. Fred Fish a royalty on all disk sales to encourage the leading Amiga program anthologist to continue his outstanding work.

CAD APPLICATION DESIGN

PART I—WORLD AND VIEW TRANSFORMS

by FOREST W. ARNOLD

INTRODUCTION

With modern computer-aided design (CAD) systems, models of real or imagined objects can be quickly and easily created, displayed on a monitor, or plotted on a plotter or printer. Model objects can be readily changed, and the effect of the modifications can be instantly seen and evaluated.

Objects can be moved around, made bigger or smaller, and turned around and upside down. We can create and experiment with models of objects smaller than atoms or as large as the universe - all on a computer monitor. We can 'zoom in' on part of an image to examine small details, then 'zoom out' to see the image in its entirety.

Since we can create and manipulate complex models with CAD systems, we might expect CAD software to be complex. However, no matter how complex the model or the CAD system, all CAD systems are built around a surprisingly small number of relatively simple software procedures.

This is the first of two articles which will explore the basic procedures used in vector-based two-dimensional CAD systems. In this article, we'll take a look at the mathematics which makes CAD systems possible. We'll then develop a library of useful, general-purpose transformation procedures for manipulating two-dimensional graphical objects. Also, we'll see how drawings are displayed in a window or plotted

on a plotter at any scale or rotation angle. In a later article, we'll take a look at direct manipulation techniques used in CAD systems, and combine these techniques with our transform procedures to create the basic part of a two-dimensional drawing program.

CARTESIAN COORDINATES AND VECTORS

Let's start out by seeing how CAD systems model the world and the geometric objects in it. One of the simplest ways to represent geometric objects is with Cartesian coordinates. A two-dimensional Cartesian coordinate system is formed by imagining a vertical line and a horizontal line in a plane which cross each other, then measuring distances along the lines from the place where the two lines cross. The vertical line is called the y axis and the horizontal line is called the x axis. Distances along the x axis are called x coordinates and distances along the y axis are called y coordinates. The units for measuring distances along the axes can be inches, meters, light-years, or whatever unit of measure is appropriate for the objects to be described.

Any point can be located relative to the two coordinate axes by specifying its x coordinate and its y coordinate. The usual way coordinates are written is as an ordered pair of

MATHEMATICS & TECHNIQUES FOR DESIGNING AND PROGRAMMING CAD SOFTWARE FOR THE AMIGA™

numbers inside parentheses, with the x coordinate written first, followed by the y coordinate. The coordinates where the axes cross each other is (0,0). This point is called the origin of the coordinate system. X coordinates to the right of the origin are positive, and are negative on the left side of the origin. Likewise, y coordinates above the origin are positive, and are negative below the origin. So (1,0) represents a point on the x axis which is 1 unit to the right of the origin, and (0,-1) represents a point on the y axis which is 1 unit below the origin.

After a coordinate system has been set up and an object is drawn in the coordinate system, any point on the object can be described in terms of its Cartesian coordinates. Figure one (see page 10) shows how a rectangle can be drawn in a Cartesian coordinate system. The rectangle in figure one is described by specifying the location of the corners of the rectangle in Cartesian coordinates. The sides of the rectangle are line segments which connect the corners. In a program, one of the simplest data structures for storing and using the Cartesian coordinates defining a set of connected points is an array.

The (x,y) pairs are stored in contiguous array locations. Each point is implicitly connected to the next point in the array by a line segment. Any point in the figure can be chosen as the

starting point. For the rectangle in figure one, an array which can hold 10 coordinates is needed (for closed figures, the last coordinate pair and the first coordinate pair are usually the same). If the upper, left corner is chosen as the starting point, the array of coordinates defining the rectangle is:

Index:	0	1	2	3	4	5	6	7	8	9
Coordinate:	x	y	x	y	x	y	x	y	x	y
Values:	-4	2	4	2	4	-2	-4	-2	-4	2

Coordinate pairs representing single points are also called vectors. Vectors are very similar to Cartesian coordinates. Both can be represented using the same notation, (x,y), and both specify distances and directions. The difference between vectors and coordinates is that coordinates always define directed distances from the origin along the coordinate axes, and vectors define directed distances from ANY point, which is not necessarily the origin.

The coordinate system used to describe geometric objects is called a WORLD coordinate system. The coordinate system used to draw the images of the objects is called a VIEW coordinate system. World coordinate systems and view



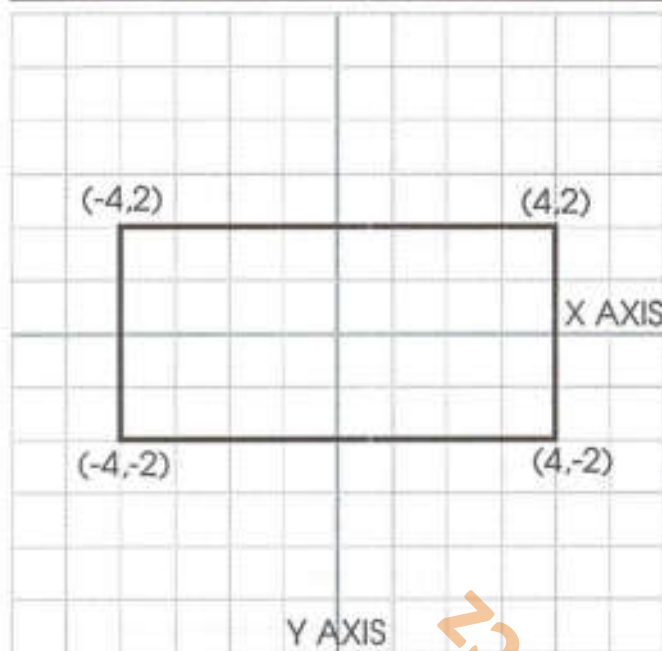


Figure One

A rectangle drawn in a Cartesian coordinate system.

geometric objects. Without this ability, CAD programs wouldn't be able to convert world coordinate descriptions of objects to window coordinates and correctly display images of the objects. Nor could they provide us with the ability to edit graphical images.

Points and line segments constitute almost all of the geometric objects used in CAD systems. To manipulate points and lines, we need to know how to:

- 1.) Add and subtract vectors.
- 2.) Multiply and divide a vector by a number.
- 3.) Multiply a vector by a matrix.
- 4.) Multiply two matrices.
- 5.) Find distances between points in the plane.
- 6.) Find the equation of a line which passes through two points.
- 7.) Determine the point where two lines intersect.

We'll start out with the rules for vectors and matrices, since these are the critters used to transform geometries in CAD systems and since the transformation procedures use the rules. We'll get to 5, 6, and 7 in a little while.

I stated above that vectors were similar to Cartesian coordinates. So (x,y) can be either two-dimensional Cartesian coordinates or a two-dimensional vector. Another notation for

coordinate systems are almost always different. In intuition and most other window systems, the view coordinate system has its origin in the upper, left corner of the windows, and y coordinate values are positive below the origin.

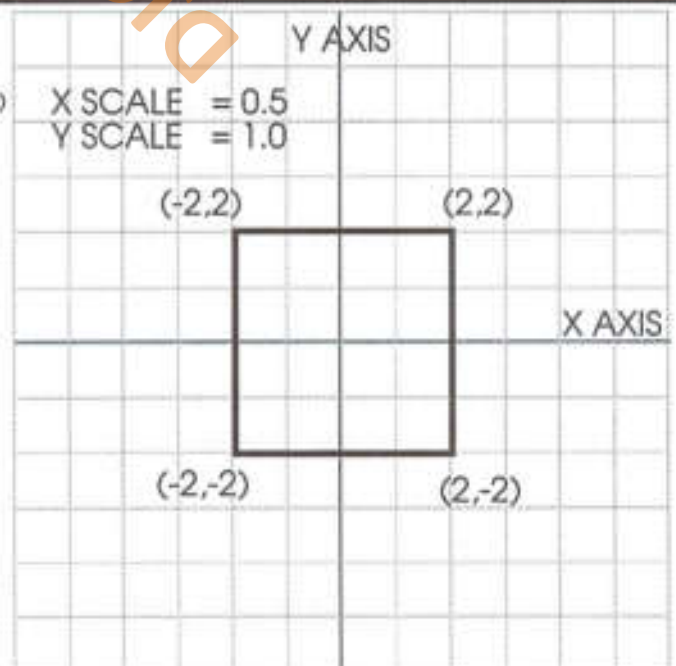
Two of the basic tasks which any CAD system has to perform are converting world coordinates to view coordinates, and converting view coordinates back to world coordinates. Three other basic operations CAD systems apply to geometric objects are scaling, translations, and rotations. Scaling is used to "zoom in" and "zoom out". Scaling makes objects appear bigger or smaller. Translations are how objects are moved from one place to another. "Panning" is also accomplished using translations. Rotations turn objects around, over, and upside down. Figures two, three, and four, are representations of these three operations. All of these basic tasks are performed using geometric transformations on vectors. To understand how transformations work, we need to look at a few mathematical formulas and techniques.

CAD SYSTEM MATHEMATICS

Cartesian coordinate systems are very useful for determining where points are located, determining distances between locations, and describing geometric objects in terms of points on the object. But they do much more than just this. They also permit us to apply algebraic techniques and formulas to

Figure Two

A rectangle (from figure one) scaled in a Cartesian coordinate system



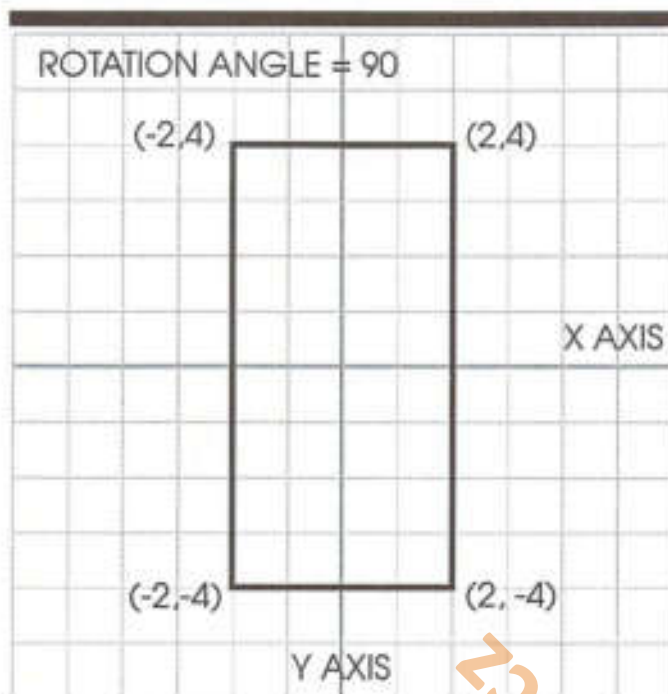


Figure Three

A rectangle (from figure one) rotated in a Cartesian coordinate system

To multiply (divide) a vector by a number: Multiply (divide) each element of the vector by the number.

Using this rule,

$$a * [x1 \ y1] = [a*x1 \ a*y1]$$

and

$$[x1 \ y1] / a = [x1/a \ y1/a]$$

Pretty simple so far! Let's complicate things by looking at matrices and the rules for them. A matrix is a table of values, organized into rows and columns. A matrix is also a collection of vectors. Here is a 2x2 (two rows by two columns) matrix:

	Col 1	Col 2
Row 1	m11	m12
Row 2	m21	m22

Where m11 is the element at row 1, column 1; m12 is the element at row 1, column 2; m21 is the element at row 2, column 1; and m22 is the element at row 2, column 2. Notice that the elements of a matrix are subscripted: the subscripts indicate the row (first subscript) and column (second subscript) where the element is located.

representing vectors is $[x \ y]$. In this notation, the vector is called a row vector, since the elements defining the vector are all on the same row. An alternate representation for the same vector is

$$\begin{bmatrix} x \\ y \end{bmatrix}$$

This is called a column vector, since the elements are all placed in a single column. The two notations are equivalent, and convenience or mathematics dictates which is used at any given time.

Suppose we have two row vectors, $[x1 \ y1]$ and $[x2 \ y2]$. Here is the rule for adding and subtracting vectors: To add (subtract) two vectors: Add (subtract) the corresponding elements of each vector.

Using this rule,

$$[x1 \ y1] + [x2 \ y2] = [x1+x2 \ y1+y2]$$

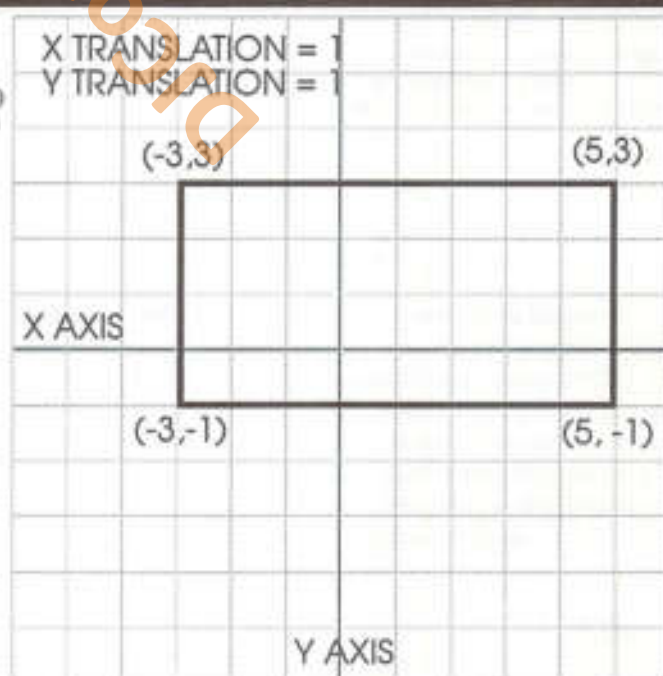
and

$$[x1 \ y1] - [x2 \ y2] = [x1-x2 \ y1-y2]$$

The rule for adding and subtracting vectors may lead you to believe that vectors can be multiplied and divided in about the same way. This is not true! However, it is easy to multiply or divide a vector by a number. Here is the rule:

Figure Four

A rectangle (from figure one) transformed in a Cartesian coordinate system



Now suppose we have a matrix, A, and another matrix, B, and we want to multiply the matrices. The first requirement is that we can multiply the matrices. Matrix multiplication is only defined if the number of columns in the first matrix is the same as the number of rows in the second matrix. So if A has 2 rows and 3 columns, and B has 3 rows and 2 columns, then $A * B$ is legal. The product of two matrices is another matrix. The number of rows in the product matrix will be equal to the number of rows of the first matrix, and the number of columns in the product matrix will be equal to the number of columns in the second matrix. So if $C = A * B$, then C will have 2 rows and 2 columns. A is a 2x3 (two by three) matrix, B is a 3x2 matrix, and the product, C is a 2x2 matrix. Let's see how matrix multiplication is actually done:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$$\begin{aligned} c_{11} &= a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} \\ c_{12} &= a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} \\ c_{21} &= a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} \\ c_{22} &= a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} \end{aligned}$$

As you can see from the example, each element of the product matrix is the sum of the products of elements from a row of the first matrix and the corresponding column of the second matrix. For this reason, matrix multiplication is called "row-column multiplication". Rather than state a rule for matrix multiplication, here is an algorithm for multiplying two matrices:

input: matrix A which is $m \times n$,
matrix B which is $n \times k$
output: product matrix C which is $m \times k$

```
do for i = 1, i <= m
  do for j = 1, j <= k
    set c(i,j) = 0
    do for l = 1, l <= n
      set c(i,j) = c(i,j) + a(i,l) * b(l,j)
```

It should be apparent from this algorithm that computers can multiply two matrices a lot easier than humans can!

Multiplying a matrix times a vector is done the same way as multiplying two matrices: we write the vector as a column vector and treat it as a matrix with n rows and one column. So, if a vector with 3 elements is multiplied by a 2x3 matrix, the product will be a 2x1 matrix, which is a column vector with 2 rows. Here is how the multiplication looks:

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} m_{11}v_1 + m_{12}v_2 + m_{13}v_3 \\ m_{21}v_1 + m_{22}v_2 + m_{23}v_3 \end{bmatrix}$$

We only need to know a few more things about matrix multiplication before seeing how matrices are used to transform geometric objects. First, the order in which matrices are multiplied is important. If we have a matrix M and another matrix N, $M \times N$ is almost never equal to $N \times M$. In fact, unless both M and N have the same number of rows as columns, one of the multiplications is not even defined! Second, there is no such thing as division of one matrix by another matrix or vector. So, if we multiply M times a vector v and get a vector c, we can't just divide M by c to find the vector we started with. To do this, we have to find a matrix called an INVERSE matrix.

An inverse matrix, M' , is defined to be a matrix which, when multiplied by another matrix M, will produce a special matrix called an IDENTITY matrix. The identity matrix is a square matrix: it has the same number of rows and columns. All entries of the identity matrix are zero, except for the diagonal elements, which are equal to 1. Here is a 3x3 identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix is called an identity matrix because, if another matrix is multiplied by it, the matrix is not changed by the multiplication. Identity matrices are frequently denoted by just writing "I" to mean any identity matrix. For a matrix to have an inverse, the matrix has to be a square matrix. It is important to keep in mind that even if a matrix is square, it may not have an inverse matrix. But, if there are two matrices, A and B, both $n \times n$, and $A * B = I$, then A is the inverse of B, and B is the inverse of A.

If we can find an inverse matrix, we can solve the problem of reversing a transformation which was applied to a vector. So if a vector has been transformed with a matrix A ($c = A * v$), we can find out what the original vector was by multiplying c by the inverse of A ($v = A' * c$).

TRANSFORMING WORLD COORDINATES

Three types of matrices are used in CAD systems to transform world coordinates. The three transformation matrices are a scale matrix, a rotation matrix, and a translation matrix.

Figure two shows how the rectangle in figure one will look after it is scaled. Looking at the rectangle in figure two, we can see that multiplying the coordinates of an object to be scaled will either "stretch" the object or "shrink" the object. Multiplying coordinates by a number greater than one scales it up, and multiplying by a number between zero and one scales it down. The number that is used to multiply a coordinate is called a scale factor. So if we have a pair of coordinates we want to scale, the way to do it is to multiply the x coordinate

by an x scale factor, and multiply the y coordinate by a y scale factor. Both scale factors can be the same value, but we want a technique which will let us apply different scale factors to each coordinate. The easiest way to do this is with a scaling transformation matrix, which looks like this:

$$\begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix}$$

If a set of coordinates are multiplied by this matrix, the result will be the scaled coordinates. Here is the formula for the multiplication:

$$\begin{bmatrix} sx & 0 \\ 0 & sy \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} sx * x \\ sy * y \end{bmatrix}$$

Let's see how the coordinates of the rectangle in figure one were scaled so that the width of the rectangle became half its original width, and the height stayed the same. The x scale factor to do this is .5, and the y scale factor is 1. Plugging these values into the scale matrix and multiplying the vectors corresponding to the left and right top corners of the rectangle yields:

$$\begin{bmatrix} .5 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} -4 \\ 2 \end{bmatrix} = \begin{bmatrix} -2 + 0 \\ 0 + 2 \end{bmatrix} = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} .5 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 + 0 \\ 0 + 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Here is how the array of coordinates for the rectangle looks after the scaling is complete:

Index:	0	1	2	3	4	5	6	7	8	9
Coordinate:	x	y	x	y	x	y	x	y	x	y
Value:	-2	2	2	2	2	-2	-2	-2	-2	2

Rotation matrices are constructed using formulas from trigonometry. Rotating a point is equivalent to moving the point in a circle around some other point. If we have a circle in a Cartesian coordinate system with radius, r, and the center of the circle is located at the origin, (0,0), then the formulas for the (x,y) coordinates of any point on the circle are:

$$x = r * \cos(ang1), y = r * \sin(ang1)$$

where ang1 is the angle of inclination of the point in radians with respect to the x axis.

Now, if the point is rotated counterclockwise by an angle, ang2, by adding the angle to the original angle and using a trigonometric identity for the cosine and sine of the sum of two angles, the formulas become:

$$x = r * \cos(ang1+ang2) = x * \cos(ang2) - y * \sin(ang2)$$

$$y = r * \sin(ang1+ang2) = x * \sin(ang2) + y * \cos(ang2)$$

The expanded formulas are exactly what we need to build a rotation matrix which will rotate a point by any angle! Here it is:

$$\begin{bmatrix} \cos(a) & -\sin(a) \\ \sin(a) & \cos(a) \end{bmatrix}$$

If a column vector is multiplied by this matrix, the result will be a vector containing the rotated coordinates. Here is the multiplication:

$$\begin{bmatrix} \cos(a) & -\sin(a) \\ \sin(a) & \cos(a) \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x * \cos(a) - y * \sin(a) \\ x * \sin(a) + y * \cos(a) \end{bmatrix}$$

Let's rotate the rectangle of figure one by 90 degrees to see how rotation matrices work. Doing this will produce the coordinates for the rectangle in figure 3. The cosine of 90 degrees is 0, and the sine of 90 degrees is 1. Plugging these values into our rotation matrix and multiplying the two top corner coordinates of the rectangle yields:

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} -4 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 - 2 \\ -4 + 0 \end{bmatrix} = \begin{bmatrix} -2 \\ -4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 - 2 \\ 4 + 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 4 \end{bmatrix}$$

This is the array of coordinates after each point is rotated:

Index:	0	1	2	3	4	5	6	7	8	9
Coordinate:	x	y	x	y	x	y	x	y	x	y
Value:	-2	-4	-2	4	2	4	2	-4	-2	-4

The only other transformation matrix we need for a complete set of transforms is a translation matrix. Translation is just a different word for moving an object. It should be pretty obvious from figure four that all we need to do to move a point is to add offsets (translations) to the coordinates of the point. The formulas for doing this are:

$$x = x + x \text{ translation}, y = y + y \text{ translation}$$

→

The only problem with this is that adding translations to coordinates doesn't fit in with multiplication of 2x2 matrices. To solve this problem, we increase the dimension of our matrix and use what is known as HOMOGENEOUS coordinates. Instead of using just [x y] to represent a point, we use an "extra" coordinate and ignore it. The extra coordinate is equal to 1 (it doesn't have to be, but for two-dimensional graphics, 1 works just fine). So a point becomes [x y 1] in homogeneous coordinates. Since the point is represented with three elements, the transformation matrices for homogeneous coordinates have to be 3x3 matrices. If we add a zero term in the formulas for translation, we can see how our translation matrix should look:

$$\begin{aligned}x &= x + 0 + x \text{ translation} \\y &= y + 0 + y \text{ translation} \\1 &= 0 + 0 + 1\end{aligned}$$

Writing these formulas as a matrix gives:

$$\begin{bmatrix} 1 & 0 & xtrans \\ 0 & 1 & ytrans \\ 0 & 0 & 1 \end{bmatrix}$$

If a homogeneous coordinate vector is multiplied by this matrix, the coordinate will be moved by "xtrans" along the x axis and by "ytrans" along the y axis. The multiplication looks like this:

$$\begin{bmatrix} 1 & 0 & xtrans \\ 0 & 1 & ytrans \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + 0 + xtrans \\ y + 0 + ytrans \\ 0 + 0 + 1 \end{bmatrix}$$

Multiplying the top, corner coordinates of our rectangle by a translation matrix with xtrans = 1 and ytrans = 1 will give us the translated coordinates, as follows:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} -4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -4 + 0 + 1 \\ 2 + 0 + 1 \\ 0 + 0 + 1 \end{bmatrix} = \begin{bmatrix} -3 \\ 3 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 + 0 + 1 \\ 2 + 0 + 1 \\ 0 + 0 + 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}$$

Ignoring the extra coordinate, this is the array of coordinates after each point is translated:

Index:	0	1	2	3	4	5	6	7	8	9
Coordinate:	x	y	x	y	x	y	x	y	x	y
Value:	-3	-1	-3	3	5	3	5	-1	-3	-1

We have now derived all the matrices needed to manipulate two-dimensional objects, and have seen how to use each of them to perform the three basic CAD operations of scaling, rotating, and translating world coordinates. Rewriting the scale matrix and the rotation matrix for homogeneous coordinates, the three matrices are:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

All of the examples have used single transformations. If we want to scale, rotate, and translate a point, do we have to apply the transformations separately? Do we have to first multiply the point by a scale matrix, then multiply the scaled point by a rotation matrix, and finally, multiply the scaled and rotated point by a translation matrix? No, we don't; the three transformation matrices can be multiplied together, and then the point to be transformed can be multiplied by the overall matrix. The scale matrix is created and multiplied by a rotation matrix, then the product matrix is multiplied by a translation matrix. The final product matrix combines all three transformations into a single matrix. Then, when a vector is multiplied by the overall transformation matrix, the vector will be scaled, rotated, and translated all at once!

So far we've developed the transform matrices to convert a geometric object modeled with vectors from one world coordinate representation to another (transformed) world coordinate representation. To display the model in an Intuition window and to interact with the model using a pointing device, we need to know how to represent the model in the window's coordinate system. We'll now see how the world coordinate representation of a geometric object is transformed to a window coordinate representation, and vice versa.

WINDOWS, VIEWPORTS, AND VIEWING TRANSFORMATIONS

Suppose you are designing a house with a CAD system. Naturally, you would like for the dimensions of your design to be in feet and you would like to work with feet instead of pixels while creating and modifying your design. In CAD terms, you want to work in object space and you want the dimensions of your design to be in world coordinates. However, the unit of measure for Intuition screens and windows is pixels, and the image of the design which is displayed is in pixel coordinates. The image of the design is in image space and the image's dimensions are in view coordinates.

Computers can't display geometric objects in object space, only in image space. So CAD systems have to convert the world coordinate model to an image in view coordinates to display it. The conversion is performed with a viewing transformation. Figure five depicts the overall transform process which occurs in CAD systems.

Designers work with a model in world coordinates: moving parts, scaling parts, and rotating parts using world coordinate transforms. The CAD system then applies a viewing transformation to the world coordinates to create an image of the object in view coordinates. The image is then displayed or plotted. To enable objects to be picked with a pointing device, the CAD system has to reverse this process: the pick coordinates are view coordinates which have to be converted back to world coordinates.

Viewing transformations map a rectangular area in world coordinate space onto a rectangular area in view coordinate space. The rectangular areas are called windows. The area in world coordinates is a world window, and the corresponding area in view coordinates is a view window. Figure six shows how a world window corresponds to a view window being displayed in an Intuition window. Viewing transforms are calculated from the coordinates of the world window and the coordinates of the view window by mapping the corners of the world window to the matching corners of the view window. A view transform for figure six would be calculated so that (wxl,wyb) is transformed to (vxl,vyb) and (wxr,wyt) is transformed to (vxr,vyt) .

Let's use some actual numbers for the coordinates in figure six and see what is involved in creating a view transform. Suppose the house is designed using feet, and the lower, left world window coordinates are (1,2) and the upper, right coordinates are (21,42). The height of the world window is then 20 feet, and the width is 40 feet. Suppose the Intuition window is 80 pixels high and 220 pixels wide, and that the lower, left view window coordinates are (10,70) and the upper, right view window coordinates are (210,20). The view window is then 50 pixels high and 200 pixels wide. Here is a summary of the coordinates and dimensions:

World window: $(wxl,wyb) = (1,2)$, $(wxr,wyt) = (21,42)$
 world width = 20, world height = 40
View window: $(vxl,vyb) = (10,70)$, $(vxr,vyt) = (210,20)$
 view width = 200, view height = 50

To make the dimensions of the world window the same as the dimensions of the view window, we need to find width and height scale factors which can be multiplied times the world dimensions to yield the view dimensions. Here is the formula we want:

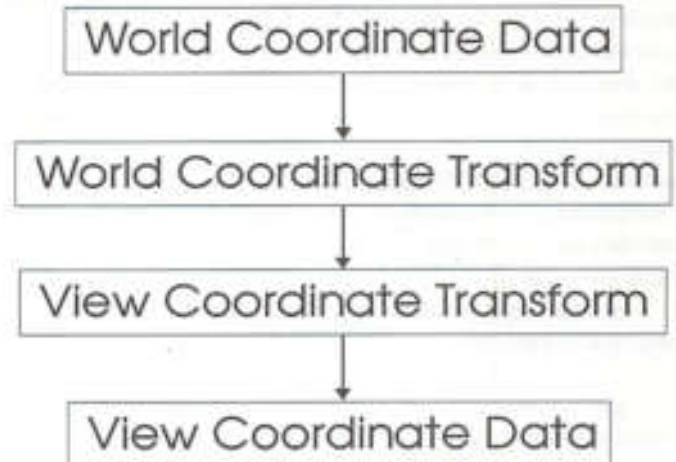
view dimension = world dimension * scale factor

Solving this formula for the scale factor gives

scale factor = view dimension / world dimension

Figure Five

The Transform Pipeline



Multiplying the scale factor times world dimensions will produce the desired view dimensions. The only problem with this formula is that our world y coordinates increase in the "up" direction and our view y coordinates increase in the "down" direction: wyt is greater than wyb , but vyt is less than vyb . We can include information about direction in our scale factors by using the following formulas:

width scale factor = $(vxr - vxl) / (wxr - wxl)$
 height scale factor = $(vyl - vyt) / (wyt - wyb)$

We can find the scale factors for our example by plugging the example values into these formulas:

width scale factor = $(210 - 10) / (21 - 1) = 10$
 height scale factor = $(20 - 70) / (42 - 2) = -5/4$

These scale factors will map directed distances relative to the world coordinate origin onto directed distances relative to the view coordinate origin. They will also make directions in view coordinates correspond to directions in world coordinates.

Take another look at figure six (see page 17). The world window is not located at the world coordinate origin, and the view window is not located at the Intuition window origin. So if we just multiply world coordinates by these scale factors, we won't get the correct view coordinates! However, if we first move the world window so that it is located at the world coordinate origin BEFORE multiplying by the scale factors, we'll get correctly scaled view coordinates, relative to the view coordinate origin. Since the view window is not located at its origin (the top, left corner of the Intuition window), we then move the view coordinates to their location inside the view window.

The translation values to move the world window so that it is located at the world coordinate origin can be found by subtracting the lower, left world window coordinates from the window corner coordinates. If the world coordinates are first translated then scaled, the resulting view window will have its lower, left view window coordinates located at the Intuition window origin. By adding the view window lower, left coordinates to the calculated view coordinates, the view coordinates will be moved to their correct position in the Intuition window.

Summarizing the above discussion, three steps are required to transform world coordinates to view coordinates:

Step 1: world coordinates are moved so that the specified world window is located at the world coordinate origin. This step is accomplished with a translation matrix, using the lower, left world window coordinates for the translation values. The translation matrix is:

$$\begin{bmatrix} 1 & 0 & -wxl \\ 0 & 1 & -wyb \\ 0 & 0 & 1 \end{bmatrix}$$

Step 2: world coordinates are converted to view coordinates, relative to the view window origin. This step is accomplished with a scale matrix:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where ...

$$sx = (vxr - vxl) / (wxr - wxl) \\ sy = (vyt - vyb) / (wyt - wyb)$$

Step 3: view coordinates are moved so that the view window is positioned at the desired location in the Intuition window. This step is accomplished with a translation matrix, using the lower, left view window coordinates for the translation values:

$$\begin{bmatrix} 1 & 0 & vxl \\ 0 & 1 & vyb \\ 0 & 0 & 1 \end{bmatrix}$$

The above three matrices can be combined by matrix multiplication, resulting in the following overall view transformation matrix:

$$\begin{bmatrix} sx & 0 & -sx * wxl + vxl \\ 0 & sy & -sy * wyb + vyb \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix will work just fine as long as view distances along the x axis and the y axis are equal, and as long as the width and height scale factors are equal.

Unfortunately, this is not usually the case: 100 horizontal Intuition pixels is not the same length as 100 vertical Intuition pixels, and the scale factors are usually not equal because the world window and the view window are generally not squares.

We need to correct the scale factors to produce equal scaling in x and y, and to compensate for different horizontal and vertical pixel resolutions. The scale factors can be made equal by simply using the smaller of the two scale factors, and the difference between horizontal and vertical view distances can be compensated for by using a value called an aspect ratio. The x aspect ratio is the number of pixels per unit in x divided by the number of pixels per unit in y, and the y aspect ratio is the number of pixels per unit in y divided by the number of pixels per unit in x.

Where do we get the values needed to calculate the aspect ratio? Thanks to the Amiga software designers, the values are stored in the GfxBase structure in the member variables named "NormalDPMX" and "NormalDPMY". These variables contain the number of dots per meter along the x and y axes for an interlaced screen. After a GfxBase structure pointer has been set by opening the graphics library, the x aspect ratio for Intuition screens and windows can be found by dividing NormalDPMX by NormalDPMY. For non-interlaced screens, NormalDPMY needs to be halved before the aspect ratio is determined. The y aspect ratio is just the reciprocal of the x aspect ratio.

We can now specify how the translation and scale values for the view transform are calculated. The translation values are calculated by scaling the world coordinate translations and combining them with the view coordinate translations:

$$xtrans = vxl - wxl * (vxr - vxl) / (wxr - wxl) \\ ytrans = vyb - wyb * (vyt - vyb) / (wyt - wyb)$$

The scale values are calculated with the following algorithm, which finds the smallest scale factor in x or y and corrects the appropriate value using the x aspect ratio:

Input: world window coordinates,
view window coordinates,
x aspect ratio
Output: translation values and scale values
for a view transform

Step 1: Find directed view and window distances:

```
Set vheight = vyt - vyb
Set vwidth = vxr - vxl
Set wheight = wyt - wyb
Set wwidth = wxr - wxl
```


Step 2: Find absolute scale factors

```
If ABS(vheight) is less than ABS(vwidth),
    Set yscale = ABS(vheight/wheight)
    Set xscale = yscale * xaspect
Else
    Set xscale = ABS(vwidth/wwidth)
    Set yscale = xscale * xaspect
```

Step 3: Correct the sign for the scale factors

```
If vwidth and wwidth have different signs,
    Set xscale = xscale * -1.0
If vheight and wheight have different signs,
    Set yscale = yscale * -1.0
```

We then plug the calculated values into a view transform matrix:

$$\begin{vmatrix} \text{xscale} & 0 & \text{xtrans} \\ 0 & \text{yscale} & \text{ytrans} \\ 0 & 0 & 1 \end{vmatrix}$$

THE TRANSFORMATION AND MATRIX PROCEDURES

Listing one (all listings are on the AC's TECH disk) contains procedures which implement all of the above transformations, manage a transform structure, and handle the details of transforming coordinates. Listing one has almost all of the procedures needed for creating vector-oriented two-dimensional graphics programs. Listing two is the include file which defines the transform structure, declares the transform procedures, and defines some useful macros. The procedures in listing one can be compiled and linked with a graphics program, or can be placed into an object module library for later linking.

Here is a description of how the procedures work, and how to use them. The linkages are documented with the procedure listings.

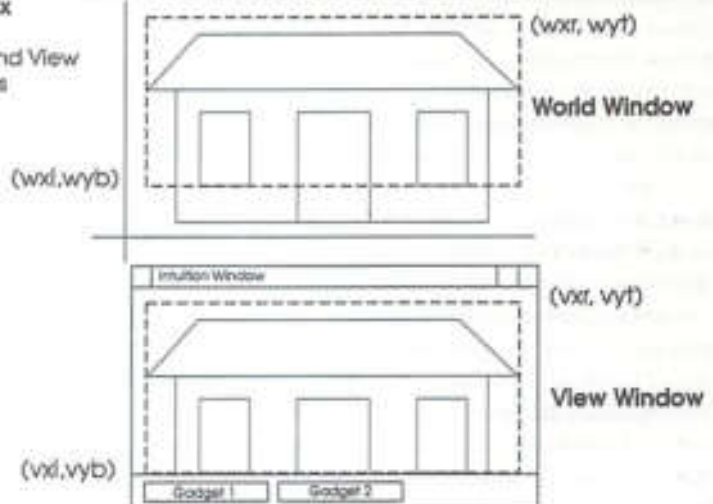
The basic transform data structure is defined in "transform.h". It is typedef'd as "transform2_t". It stores the scale values, translation values, and rotation value for a transform, and also stores the transformation matrix and its inverse. A pointer to a transform2_t structure is required as input for the transform procedures. The transform procedures are setWorldTransform(), getTransform(), point2dForward(), and point2dInverse(), worldWindowToViewWindow(), and setViewTransform().

SetWorldTransform() initializes the transform2_t structure, creates a world transform matrix, and finds the inverse transform matrix. The transform matrix is created so that coordinates will first be scaled, then rotated, and finally translated. You may want to experiment with creating the transform in a different order; for example, rotating first, then scaling. The results may surprise you!

GetTransform() is a simple utility procedure which just returns the current transform values stored in the transform2_t structure. Of course, you can access these values directly, but it

Figure Six

World and View Windows



is good programming practice to isolate access and update of data structures in a single place. This way, if the structure changes, the impact on your program will be minimized!

Point2dForward() and point2dInverse() are the two work-horse procedures for transforming coordinates. Point2dForward() transforms a set of coordinates and returns the transformed coordinates. The transform is performed by multiplying the coordinates by the transform matrix. Point2dInverse() converts a set of transformed coordinates back to the original coordinates (there may be round-off or truncation error, depending on the transform and the coordinates, so the coordinates may not be exactly the same as the original ones). Point2dInverse() is just like point2dForward(), except the transformed point is multiplied by the matrix which is the inverse of the transform matrix.

WorldWindowToViewWindow() and setViewTransform() are view transform procedures. The view translation and scale values which will transform a world coordinate window to a view coordinate window are calculated in worldWindowToViewWindow(), using the algorithm described above. The calculated values are then sent to setViewTransform(), which uses the values to initialize a transform structure and build a view transform matrix and its inverse matrix. In addition to the view translation and scale values, a rotation angle can also be sent to setViewTransform(). The view rotation angle will cause the entire view window to be rotated. After the transform structure has been initialized in setViewTransform(), it can be used with point2dForward() to transform world coordinates to view coordinates, with point2dInverse() to transform view coordinates back to world coordinates, and sent to getTransform() to access its values.

To use the transform procedures, include the file "transform.h" in your program. Allocate a transform2_t structure, either statically or dynamically. Call

setWorldTransform(), sending it the transform and transform values. Then, when you are ready to transform a point, call point2dForward(), sending it the transform2_t structure. It will do the work for you, and return the transformed coordinates. To convert the transformed point back to original coordinates, call point2dInverse().

The transform matrix and its inverse matrix are created using the matrix procedures mat2dInit(), mat2dScale(), mat2dTranslate(), mat2dRotate(), and matInvert(). These procedures can be used without the transform procedures, if you are interested in experimenting with them, or creating your own transform structures. The matrices are actually stored in singly-dimensioned arrays. For two-dimensional homogeneous transforms, nine elements are needed, since the matrix is a 3x3 matrix. So you can understand the code, here is how the matrix is stored (numbering from 0 so indexes correspond to "C" indexes):

	col 0	col 1	col 2
row 0	array[0]	array[1]	array[2]
row 1	array[3]	array[4]	array[5]
row 2	array[6]	array[7]	array[8]

Any two-dimensional matrix can be stored this way. The array index for an entry at row *i*, col *j* (matrix[i][j]) is *i* * number of columns + *j*. So the array index for the entry in matrix[1][2] is 1 * 3 + 2 = 5.

Mat2dInit() initializes the transform matrix by creating a 3x3 identity matrix:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The overall transform matrix is then created by successive matrix multiplications of the transform matrix by scale, rotation, and/or translation matrices.

Mat2dScale() multiplies the transform matrix by a scaling matrix:

$$\begin{vmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} t00 & t01 & t02 \\ t10 & t11 & t12 \\ 0 & 0 & 1 \end{vmatrix}$$

Mat2dTranslate() multiplies the transform matrix by a translation matrix:

$$\begin{vmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} t00 & t01 & t02 \\ t10 & t11 & t12 \\ 0 & 0 & 1 \end{vmatrix}$$

Mat2dRotate() multiplies the transform matrix by a rotation matrix:

$$\begin{vmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} t00 & t01 & t02 \\ t10 & t11 & t12 \\ 0 & 0 & 1 \end{vmatrix}$$

Finally, matInvert() is a matrix inversion procedure which can find the inverse of an *n* x *n* matrix (if the matrix has an inverse). I'm not going to describe matrix inversion, but you can find the algorithm in any textbook about linear algebra. The algorithm used is called "Gauss-Jordan elimination with partial pivoting".

THE TRANSFORM DEMONSTRATION PROGRAMS

Demonstration programs which show how to use the transform procedures are presented in listing three and listing four. The program in listing three shows how the transforms for scaling, rotating, and translating model geometric objects are used. The program in listing four shows how to set up and use view transforms.

The program in listing three, tform1.c, displays a set of coordinate axes and a polygon centered in an Intuition window. Four gadgets, labeled "Translate", "Rotate", "Scale", and "Reset" are displayed in the window, and each time a transform gadget is selected, that transform is applied to the polygon.

The "Translate" gadget is used to move the polygon in increments of 100 world coordinate units. Each time the gadget is selected, the polygon will be moved right, left, up, or down. Selecting the "Rotate" gadget rotates the polygon counter-clockwise by 30 degree increments until it turns all the way around. It then rotates the polygon clockwise in -30 degree increments. The "Scale" gadget is used to scale the polygon using scale values between .4 and 2.0, in increments of .4. The *x* and *y* scale values are calculated separately. Selecting the "Reset" gadget restores the polygon to its original, untransformed position and size. The program demonstrates what effect each separate transformation has, and also shows the effect of combinations of transformations.

Tform2.c, the program in listing four, is similar to tform1.c, except all transformations are performed with a view coordinate transform instead of world coordinate transforms. The entire model world, instead of individual objects, is transformed.

The program starts by displaying three geometric objects and a set of coordinate axes centered in a view window. Two pushbutton gadgets are displayed in the lower, left corner of the Intuition window. One gadget is labeled "Full Display", and the other is labeled "Set Transform".

When the "Full Display" gadget is selected, the program recalculates the data for the view transform so that the entire world window will be displayed in the Intuition window. The

"Full Display" scale for both axes is set to scale 1. This is not the actual scale value for the view. Instead, it is a scale value relative to the size of the current view window. Thus, it is a base value for increasing or decreasing the view scale. This is done by multiplying the actual x scale and y scale values by the relative scale value.

When the "Set Transform" gadget is selected, the transform requester "pops up" in the Intuition window. The transform requester contains labeled string gadgets for displaying current transform values and for entering new transform values.

It also contains an "Accept" pushbutton and a "Cancel" pushbutton. Transform values are modified by selecting the appropriate string gadget and typing in the new value. When the "Accept" button is selected, the requester is "popped down", the transform values are read from the string gadget buffers, the view transform is updated, and the view is redisplayed using the updated view transform. When the "Cancel" button is selected, the program simply removes the requester. The program is stopped by selecting the window "Close" gadget.

The executable programs are created by compiling and linking the programs with the transform procedures. The Lattice commands to compile and link the programs are:

```
"lc -lm tform1.o transform.o"
"lc -lm tform2.o transform.o"
```

HOW THE PROGRAMS MANAGE DATA AND HANDLE EVENTS

Both programs use several neat programming "tricks". First, linked, hierarchical-style data structures are used for describing and organizing the program's data. Second, indirectly executed (via function pointers) "event handler" procedures are used with the gadgets. These features are described in the next several paragraphs.

The program data and state information is organized in a hierarchy. The top level of the hierarchy is a "world_t" structure. A pointer to this structure is stored in the "UserData" field of the Intuition window in which the data is being displayed. Storing the top-level structure pointer with the window allows us to access all of our data and state information through the Window pointer.

The world_t structure contains an overall description of the entire set of world coordinates, and a pointer to the view transform structure. It also contains a pointer to a linked list of all the world coordinate geometric objects. In addition to the list of world objects, this structure contains the minimum and maximum extents of the world coordinated system. The extents are used to determine how the world coordinates will be scaled to fit in the window.

The objects are stored in an "object_t" structure, which is simply a polygon descriptor record. The actual coordinates for each geometric object are stored in a singly-dimensioned array

of sequential x,y coordinate pairs. The object_t structure contains a pointer to the object's coordinate array, and the dimension of the array. The object_t structure used in tform1 also contains a pointer to an individual world transform for each object.

Function pointers are used to process gadget events. A pointer to the procedure which is called from the event processing loop when a gadget is selected is stored in the gadget's "UserData" structure. Whenever a "GADGETUP" event occurs, the input handler retrieves the function pointer and calls the procedure using the function pointer. Each gadget handler procedure has only one call argument: the pointer to its window. Since the window pointer contains a pointer to the world_t structure, the handler procedure can access all the information it needs to do its job from the window pointer.

To make casting the pointer from an Amiga APTR to a function pointer easier, a typedef for the function pointer is used. By using event handler function pointers, we can simplify event processing and eliminate "switch" type code which requires detailed knowledge of the program's control and data structures.

PROGRAM CONTROL FLOW AND DRAWING PROCEDURES

Both programs work basically the same way. The main() procedure initializes data structures, opens an Intuition window, calculates the initial view transform for the window, displays the model world, and then calls handleInput().

Window gadgets are initialized and function pointers are attached to the gadgets in initWinGadgets(). Tform2 also calls initTfRequester() to create the 'Set Transform' requester and its gadgets. Both initWinGadgets() and initTfRequester() are straight-forward, except for the "static" structures defined in initTfRequester().

The structures are "static" to prevent them from being overwritten after the procedure returns, since they contain data for the requester and its gadgets used throughout tform2. Remember that local variables in a procedure which are not static are allocated from the stack when the procedure is entered, and the space is made available for reuse when the procedure returns!

In tform1, the view transform is calculated in findViewTransform(). This procedure is called the first time the window is displayed, and then again each time a 'refresh window' event is received. In tform2, the view transform is calculated in fullDisplay() each time the "Full Display" gadget is picked.

The input handler receives input events, retrieves the event handler function pointer, and calls the event handler procedure. This process continues until the window 'close' gadget is picked. In tform1, HandleInput() is only concerned

with three events: CLOSEWINDOW, GADGETUP, and REFRESHWINDOW. In tform2, two additional events, REQSET and REQCLEAR, are processed.

When the first requester in a window is popped up, Intuition sends out a REQSET event, and when the last one is popped down, Intuition sends out a REQCLEAR event. Of course, to receive these events, we have to request them by setting the IDCMP flag in the NewWindow structure when we open a window.

The reason we want to get requester events is to prevent the window from being closed while a requester is displayed. Otherwise, the program could end and leave the requester just hanging around. If handleInput() receives a CLOSEWINDOW event, and no requester is being displayed, the input loop is ended, any messages are reply'ed to, and the procedure returns to main(). If a requester is being displayed, the CLOSEWINDOW event is just ignored. When a REFRESHWINDOW event is received, drawAll() is called to redisplay the window. Finally, if a GADGETUP event is received, handleInput() gets the function pointer to the gadget handler and calls the procedure using the function pointer.

The drawing procedures are drawAll(), eraseAll(), drawAxis(), and drawObject(). DrawAll() sets up and installs a clip region in the window, calls drawAxis(), then traverses the list of objects stored with the window, calling drawObject() to actually transform and display each object.

Clip regions and clipping procedures are part of the Amiga software. The structures related to clipping are defined in the Amiga graphics include files "clip.h", "regions.h", and "layers.h", and all but one of the clipping procedures are in "graphics.library". The exception is InstallClipRegion(), which is in "layers.library".

A clip region is basically a collection of rectangles for restricting drawing; that is, no drawing will occur outside the clipping rectangles. The clipping procedures are very useful and are documented in the "Libraries and Devices" Rom Kernel Reference Manual. The reason drawAll() uses clip regions is to prevent drawing from occurring on top of the window's gadgets. EraseAll() just erases everything in the window by calling the graphics procedure RectFill().

DrawObject() retrieves the array defining an object's coordinates and loops through the array, transforming the coordinates and drawing lines connecting each coordinate pair. In tform1, the coordinates are transformed twice: first the world transform attached to the object is used to translate, rotate, or scale the object; then the view transform attached to the window is used to convert the coordinates to pixels.

In tform2, only the view transform is used. Point2dForward() is called to perform both the world and view transforms. Notice that the world coordinates for the object being drawn in tform1 are never modified by the transformation: we keep the original coordinates and just use the transformed coordinates for drawing into the window.

DrawAxis() draws the coordinate axes through the world center coordinates, (0,0). Point2dForward is called to transform the world origin to pixel coordinates using the window's view transform. In tform1, drawAxis() simply draws horizontal and vertical lines for the axes.

In tform2, the procedure is much more complicated. All the world information, including the coordinate axes, may be rotated by the view transform. When the axes are rotated, we don't know their starting and ending pixel coordinates. To find the start and end coordinates, we need to know where each axis intersects the sides of the view window. We have to use a little algebra and perform some coordinate transformations to find the intersect points.

If we know two points, (x1,y1) and (x2,y2), we can find the equation of the line passing through the points. The general equation of a line through two points is:

$$y = (y2 - y1) / (x2 - x1) * x - (y2 - y1) / (x2 - x1) * x1 + y1$$

If $x2 - x1$ is 0, the line is vertical, and if $y2 - y1$ is 0, the line is horizontal. If the line is not vertical, it intersects the y axis at the point (0,y) and if it is not horizontal, it intersects the x axis at the point (x,0). So, to find where it intersects the y axis, we set x equal to zero and solve for y:

$$y = y1 - (y2 - y1) / (x2 - x1) * x1$$

To find where the line intersects the x axis, we set y equal to zero and solve for x:

$$x = x1 - (x2 - x1) / (y2 - y1) * y1$$

The above two equations are used in the version of drawAxis() in tform2 to find the points where the world coordinate axes intersect the sides of the view window rectangle.

But the coordinate axes are in world coordinates and the sides of the view window rectangle are in view coordinates! The points have to be in the same coordinate system for the equations to work right. We solve this problem by transforming the corner coordinates of the view window rectangle to world coordinates. A view transform converts world coordinates to view coordinates, but its inverse transform will convert view coordinates to world coordinates. We just happen to have the inverse transform stored in our transform structure, and we also have a procedure to use it.

The corner coordinates of the view window are transformed to world coordinates by calling point2dInverse(). DrawAxis() then determines if the sides of the view window rectangle are horizontal and vertical in the world coordinate system. If so, the world coordinate origin is transformed to view coordinates and the axes are drawn as horizontal and vertical lines through the transformed origin.



Continue the Winning Tradition With the SAS/C® Development System for AmigaDOS™

Ever since the Amiga® was introduced, the Lattice® C Compiler has been the compiler of choice. Now SAS/C picks up where Lattice C left off. SAS Institute adds the experience and expertise of one of the world's largest independent software companies to the solid foundation built by Lattice, Inc.

Lattice C's proven track record provides the compiler with the following features:

- ▶ SAS/C Compiler
- ▶ Global Optimizer
- ▶ Blink Overlay Linker
- ▶ Extensive Libraries
- ▶ Source Level Debugger
- ▶ Macro Assembler
- ▶ LSE Screen Editor
- ▶ Code Profiler
- ▶ Make Utility
- ▶ Programmer Utilities.

SAS/C surges ahead with a host of new features for the SAS/C Development System for AmigaDOS, Release 5.10:

- ▶ Workbench environment for all users
- ▶ Release 2.0 support for the power programmer
- ▶ Improved code generation
- ▶ Additional library functions
- ▶ Point-and-click program to set default options
- ▶ Automated utility to set up new projects.

Be the leader of the pack! Run with the SAS/C Development System for AmigaDOS. For a free brochure or to order Release 5.10 of the product, call SAS Institute at 919-677-8000, extension 5042.

SAS and SAS/C are registered trademarks of SAS Institute Inc., Cary, NC, USA.

Other names and product names are trademarks and registered trademarks of their respective holders.



SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

Circle 146 on Reader Service card.

Otherwise, the above equations are used to find the points where all four view window sides intersect the x axis, and the points where all four sides intersect the y axis. The intersect coordinates are then sorted to find those closest to the origin. The intersect coordinates are transformed to view coordinates by calling `point2dForward()`. The transformed intersect coordinates are then used to draw the axes.

EVENT HANDLER PROCEDURES

The event handler procedures which are called in `tform1` when a gadget is selected are `rotate()`, `scale()`, `translate()`, and `reset()`. These procedures erase the contents of the window, modify the world transform for the object, then redisplay the contents of the window.

`Tform2` also has four gadget event handler procedures. They are `fullDisplay()`, `setNewTransform()`, `acceptValues()`, and `cancelValues()`. When the 'Full Display' gadget is picked, `fullDisplay()`, which was described above, is called. When the 'Set Transform' gadget is picked, `setNewTransform()` is called. This procedure updates the view transform requester, pops up the requester, and returns control to `handleInput()`. If the requester 'Cancel' gadget is selected, `cancelValues()` is called, and just pops down the requester. If the requester 'Accept' gadget is picked, `acceptValues()` is called. this procedure reads

the new view transform values, pops down the requester, updates the view transform, and then erases and redisplay the window.

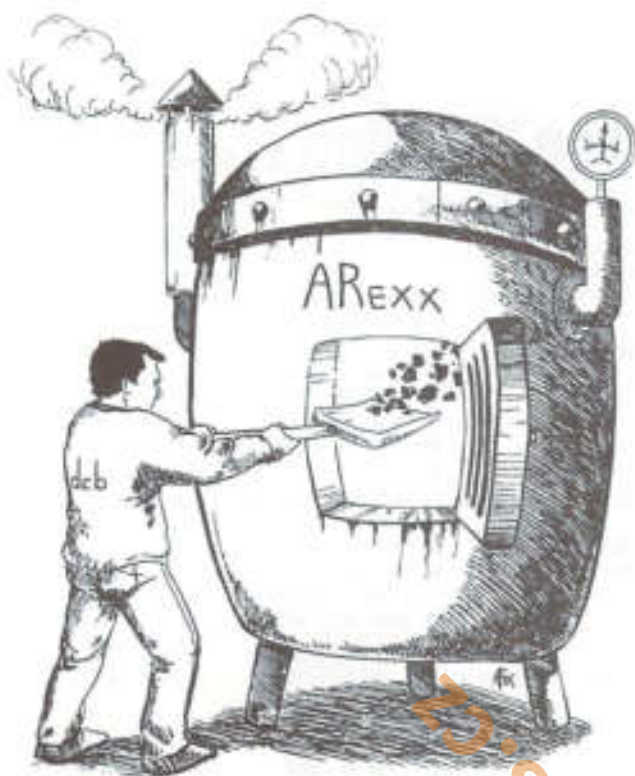
Notice the relationship between `setNewTransform()`, `acceptValues()`, and `cancelValues()`. `setNewTransform()` is an event handler which is setting up the program for a call to another event handler, either `acceptValues()` or `cancelValues()`. The gadget events are being 'chained together' through event handlers. We'll explore event chaining in more detail next time.

SUMMARY AND PREVIEW

We have now developed a data structure for storing two-dimensional transform data and a complete set of procedures for creating, updating, accessing, and using two-dimensional transforms. We have also developed techniques for processing input events using function pointers to 'event handler' procedures. We saw how to create and use world coordinate transforms and view transforms.

Where do we go from here? Well, next time, we're going to see how to use event handlers with menus and the mouse so we can move, rotate, and resize objects by 'picking and dragging' them. We'll also implement a menu 'zoom' function, and implement 'panning' with a proportional gadget.





Interfacing Assembly Language Applications to ARexx

by Jeff Glatt

Amiga owners are hearing a lot of talk about ARexx. They've been told that ARexx can be used to add a sophisticated macro language to any program, or turn several applications from various companies into a single, integrated package that is more powerful than the individual components working alone. There have been numerous articles describing how an end-user can use ARexx, but unfortunately very little information that shows a programmer how to interface his program to ARexx. This article strives to do the following:

- 1) Describes the ARexx server from the point of view of a programmer.
- 2) Demonstrates how to add an ARexx implementation to a program (i.e. how to communicate with the ARexx server using ARexx structures). This demonstration is accomplished by describing a complete (but surprisingly simple) ARexx implementation added to my example 68000 assembly program.

What's ARexx?

ARexx is a programming language. In describing ARexx, I like to compare it to BASIC because most programmers are familiar with BASIC, and also because, like BASIC, ARexx is an interpreted language. Therefore, an ARexx program (often called a macro or script) is really just a text file which must be parsed and "executed" by some interpreter. Just as you need the AmigaBASIC interpreter to run your BASIC script, you need the ARexx interpreter (called the ARexx server) to run an ARexx script. The ARexx server comes in the form of a library called "rexsyslib.library" and a program called REXXMast which loads and starts up a self-running process that can load and "execute" ARexx scripts. Furthermore, there is a program

which the user can invoke from a CLI in order to start an ARexx script's execution. Amiga ARexx is written and marketed by Bill Hawes. An end-user must obtain this product to use ARexx, either by itself or in conjunction with his own program.

From hereon in, I will refer to the ARexx server/library as simply ARexx.

Unlike AmigaBASIC, an ARexx script can be written using any text editor. The script consists solely of ascii characters, and each line ends with a newline character (10). There are no imbedded codes in the script that are required by the interpreter.

There is also no "output window" for the interpreter. ARexx prints error messages to the CLI process window from where the script was started. From WorkBench, ARexx will create a DOS window when the script starts, and close it when the script is finished.

ARexx has a built-in set of commands. For example, the SAY command prints text to the standard out handle (usually the CLI window). There are commands for constructing loops, performing tests, evaluating mathematical expressions, reading/writing files, etc. The two areas where ARexx lacks are that it has no graphics and no audio commands whatsoever. The extent of its graphics is printing text to the CLI. It has no intuition interface (i.e. can't open windows, post requesters, or support the mouse).

The two most important differences between BASIC and ARexx are in the way that variables are treated, and in the fact that ARexx can send messages to any program that has a public port, and therefore control that program.

All ARexx variables are in the form of strings. For example, if you set a variable, MyNum = 10, internally ARexx stores the value of MyNum not as a LONG, WORD, or BYTE value, but as the ASCII, null-terminated string of '10'. In other words, MyNum would be stored as the following 3 bytes: MyNum dc.b '1','0','0'; or 49,48,0. Note that ARexx has an internal mechanism for marking this as a numeric variable even though it is stored as a string. So the following two statements in an ARexx script are NOT the same:

```
MyNum = 10
MyNum = '10'
```

Although ARexx stores them both as identical strings, it marks the first as a numeric variable and the second as a string variable, two entirely different things. (Note that strings in an ARexx script are enclosed in quotes. Also note that unlike BASIC, ARexx string variables do not require any \$ or other special symbol to distinguish them. ARexx intelligently determines the type of variable when it executes the script, so the user never declares variable types.)

For details about ARexx's set of built-in commands, and information about using variables in an ARexx script, consult the ARexx Users Manual that is included with that product.

How ARexx Controls Your Program

As mentioned, ARexx can communicate with any program that has a public port, sending it "commands" that ARexx encounters in a user-written script. A program can also send "commands" and data to ARexx. Because of these capabilities, a program can use ARexx as an easy way to implement a powerful (but slow) user customizable macro language. ARexx scripts can be invoked from the program so that the user never even has to deal with the server directly, and it appears as if the macro features are "part of the program." Also, self-

running demos of programs can be made by writing an ARexx script to control the program, then invoking ARexx upon this script within a startup-sequence. Finally, two or more programs can pass data directly between each other via ARexx, creating an integrated environment.

To allow ARexx to communicate with your program, the program must add a public Exec port (i.e. a port with a name) to exec's list of ports via Exec's AddPort function. You can also use the CreatePort function as listed in the Exec RKM on page B-5, but be sure to pass a portname string. Additionally, before creating and adding the port, you should make sure that there isn't already a port with this name in exec's list. You do this with the exec function, FindPort. If so, your program should be able to dynamically change its port name to something unique. Note: You cannot use the portname of 'REXX'. This is reserved for the ARexx server's own port.

Once your program has added its port to exec's list, an ARexx script can send commands and arguments (data) to the program. ARexx does this by sending a message structure called a REXXMsg to your port. The REXXMsg contains an imbedded Exec message and looks like this: REXXMsg:

```

/* The standard Exec message part
dc.1 0 ;LN SUCC (address of the previous message linked to this port)
dc.1 0 ;LN FREQ (address of the next message linked to this port)
dc.b 5 ;LN TYPE (-WT MESSAGE)
dc.b 0 ;LN PRT
dc.1 ;LN NAME, could be the address of the null-terminated name of your port]
dc.1 ;LN ADDR (the address of your program's public port)
dc.w 128 ;the size of the REXXMsg
/* The REXX extension
dc.1 ;the address of the ARexx task]
dc.1 ;the address of rexxsyslib base]
dc.1 ;the re Action field]
dc.1 ;the re Result1 field]
dc.1 ;the re Result2 field]
dc.1 16 ;re strings, 16 addresses for upto 16 Result strings
dc.1 ;the address of your program's public port]
dc.1 ;the address of your port's name string]
dc.1 ;an ASCII null-terminated string to be appended to script names]
dc.1 ;standard input handle]
dc.1 ;standard output handle]
dc.1 ;the re avail field]
```

How does an ARexx script tell ARexx to direct REXXMsgs to your program? It does so with the ARexx command:

```
ADDRESS portname
```

where portname is the name of your program's public port. For example, if your program adds a port named, 'myPort', then an ARexx script can cause REXXMsgs to be sent there with the line

```
ADDRESS 'myPort'
```

This is case-sensitive, so the ARexx script must exactly match the name of your program's port.

When does ARexx send a message to your program? It does so whenever it encounters a line in an ARexx script which does not contain one of ARexx's built-in commands. For example, if an ARexx script contained the following line:

```
SAY 'hello'
```

then ARexx would print the word `hello` to the CLI window. `SAY` is a built-in ARexx command so it would be handled entirely by ARexx. There is no need to send an ARexx message to your program. On the other hand, assume that there is this line in a script:

```
'BLORT' 'hello' 10
```

Note that `'BLORT'` is not flush against the left margin (i.e. there are some leading spaces at the beginning of the line). Because of this, ARexx does not interpret it as a variable, but rather as some command (since it is the first element). Also note that I quoted `'BLORT'`. Although this is not necessary, it ensures that ARexx sees it as a command, just in case there also happens to be a variable by the same name earlier in the script. ARexx determines that it has no built-in command `'BLORT'`, so it packs up this entire line in a `RexxMsg` and passes the `RexxMsg` to your public port. The line is sent as a null-terminated string with the address stored in the first `rm_Args` field of the `RexxMsg` (at an offset of 40 bytes). When your program gets the `RexxMsg` from the port (via Exec's `GetMsg`) and gets the address in the first `rm_Args` field, this is the string that you find at this address:

```
BLORT hello 10
```

Note that the command and the two arguments, `hello` and `10`, are all passed in one string. Also note that although the `10` in the ARexx script was not placed in quotes, ARexx sends it as an ASCII string nevertheless. It is up to your program to parse this string, separating the command from its arguments, and convert any numeric argument (the `'10'` in this example) to a real value. Then, your program analyzes the command, and does something based upon it, perhaps using the arguments. (Your program better know what to do when it receives the `'BLORT'` command.) Note that the command and arguments may be separated by one or more spaces, but all quotes are removed from around string arguments. Unfortunately, this makes it possible to misinterpret a string with imbedded spaces as several arguments. If, for example, a user wanted the second arg to be `'hello world'`, the best approach would be to use this line in the script:

```
'BLORT' "'hello world'" 10
```

Then, the `RexxMsg`'s `rm_Args` string would be

```
BLORT "hello world" 10
```

ARexx removes the single quotes around the `"hello world"`, but leaves the imbedded double quotes which your program can use to isolate the single, string arg. You'll probably encounter several problems with respect to the confusing and unpredictable manner in which ARexx treats single quotes and whitespace.

Let's assume that our program uses the `'BLORT'` command to mean "open a window, print the first argument in a window, DOS Delay() for the second argument, then close the window." When we received this `RexxMsg`, we would detect `'BLORT'` as the command, open a window, print `"hello"`, Delay() for a factor of 10, then close the window.

After your program handles the `RexxMsg`, you must set up the two `rm_Result` fields, and reply it via Exec's `ReplyMsg`. The `rm_Result1` field should always be 0 if there were no errors in handling the `RexxMsg`. If there was some problem (for example, we couldn't open our `BLORT` window), then this should be set to an error severity level (i.e. not 0). A good guideline is 5 for warnings, 10 for errors, and 20 for fatal errors. The `rm_Result2` is used for passing any strings back to the ARexx script. If you have no string to send back, then set `rm_Result2 = 0`. Otherwise, it is the address of some `RexxArg` string (to be discussed shortly). This is useful for commands where you might need to return some data to the ARexx script. Note that if you set `rm_Result1` to some error (not 0), then you should always set `rm_Result2` to 0. Do not return a `RexxArg` string when an error occurs. Also, you should not return a `RexxArg` if the `rm_Action`'s `RESULT` bit is not set. We'll discuss `rm_Action` later. A script must have the following line if it wants you to send back a `RexxArg`:

Option Results

Note that your program receives a `RexxMsg` for every individual line in the ARexx script that has a command which is not one of ARexx's built-in commands. Therefore, your program can receive numerous `RexxMsgs` per script. Each `RexxMsg` is just one "step" of the script (i.e. one command and any arguments for it). ARexx sends these "unknown" commands to the port identified with the last `ADDRESS` command. If an ARexx script has several `ADDRESS` commands, ARexx sends messages to the portname of the last encountered `ADDRESS` command. This is so that different parts of an ARexx script can send messages to different applications. The `ADDRESS` command can even be placed before each line of the script to ensure which program that line is sent to. For example, if your portname is `'myPort'`, then this would ensure that your program gets that `'BLORT'` command:

```
ADDRESS 'myPort' 'BLORT' 'hello' 10
```


ARexx normally passes REXXArg strings. For example, that command string in the first `rm_Args` field is really a REXXArg. A REXXArg looks just like a normal null-terminated string, except that it has eight bytes PREPENDED to it; i.e., at a negative offset from the address that ARexx gives you. These prepended bytes contain info about the total length of the string (not counting the end null byte), some flags for ARexx's use, and the total size of the REXXArg. Let's see what our 'BLORT' REXXArg really looks like: REXXArg:

```
dc.l 23
;the total size of this particular REXXArg
dc.w 14
;the length of the string
dc.b [ARexx's flags]
dc.b [ARexx's hash count for the string] myStr
dc.b 'BLORT hello 10',0
```

So, the address in the REXXMsg's `rm_Args` field would be the label `myStr`, not `REXXArg`. If you wanted to find the length of the string, you would grab the word at an offset of -4 from this address.

How Your Program Controls ARexx

Your program can cause ARexx to load and execute some ARexx script. This script could then send REXXMsgs to another program, or even your program, like a macro key or self-running demo. To do this, you must allocate a REXXMsg, set the `rm_Action` and `rm_Args` fields, store your port's address in the `MN_REPLYPORT` and `rm_Passport` fields, store the port's name in the `MN_NAME` and `rm_CommAddr` fields, optionally set up the `rm_FileExt`, `rm_Stdin`, and `rm_Stdout` fields, and send this REXXMsg to the ARexx server's public port which is named 'REXX'. You send the REXXMsg by using Exec's FindPort to locate the address of the 'REXX' port, then use Exec's PutMsg.

Since the ARexx server is a library, your program can open it and call functions that allocate and free REXXMsgs. This is probably the best approach since ARexx then takes care of most of the setup of the REXXMsg, and freeing of resources. ARexx's CreateREXXMsg allocates a REXXMsg and sets up all fields to default values (usually 0). Then, you only need to set the `rm_Action` and `rm_Args` yourself.

Normally, you set the first `rm_Args` field to the address of a REXXArg string. (You can create this REXXArg using the ARexx library's CreateArgstring.) This string is the name of the ARexx script file that you want ARexx to load and execute. In addition, you may supply an extension in the REXXMsg's `rm_FileExt` field if desired. This will be appended to the filename. The `rm_Action` field (a long) contains certain subfields. The high byte of this long is a command field. You'll set it to RXCOMM (1) to indicate that this is a command level invocation. It may be set to other values instead if you wanted

F-BASIC 3.0™

Original Features:

- Enhanced, compiled BASIC
- Extensive control structures
- True Recursion & Subprograms
- FAST Real Computations
- Easy To Use For Beginners
- Can't Be Outgrown By Experts

Version 2.0 Added:

- Animation & Icons
- IFF Picture Reader
- Random Access Files
- F-Basic Linker
- Improved Graphics & Sound
- RECORD Structures
- Painters

Version 3.0 Added:

- Integrated Editor Environment
- Q20/Q30 Support
- IFF Sound Player
- Built In Complex/Matrices
- Object Oriented Programs
- Compatible with 500, 1000, 2000, 2500, or 3000

F-BASIC™ With User's Manual & Sample Programs Disk
—Only \$99.95—

F-BASIC™ With Complete Source Level Debugger
—Only \$159.95—

F-BASIC™ Is Available Only From:
DELPHI NOETIC SYSTEMS, INC.

Post Office Box 7722
Rapid City, SD 57709-7722

Send Check or Money Order or Write For Info
Credit Card or C.O.D. Call (605) 348-0791

F-BASIC is a registered trademark of DMI, Inc.
REXX is a registered trademark of International Business Machines Corporation.

Circle 192 on Reader Service card.

to add/remove function libraries, hosts, or open/close the trace console. Normally, you don't need to do these things from a program so this article will skip those topics.

For RXCOMM, ARexx will load and execute the entire script, firing off REXXMsgs to other applications as need be. When all of ARexx's REXXMsgs are returned and the script is complete, ARexx returns your initial REXXMsg to your public port. The `rm_Result1` field will be 0 if there were no errors. Otherwise, this field will be an error severity level. The `rm_Result2` field will be some REXXArg string that was sent back to you from either ARexx or another application; i.e., maybe you're expecting some data back. Your program should then free all `rm_Arg` REXXArgs and the REXXMsg (via ARexx library's DeleteArgstring and DeleteREXXMsg functions).

There are a few other features you can utilize by setting bits in the `rm_Action` field before sending off the initial REXXMsg. These bits are set in conjunction with `rm_Action's` high byte = RXCOMM. For example, you can set the TOKEN bit (19). This will cause ARexx to parse and separate each argument of the script line, and create separate REXXArgs for each.

Let's assume that we have the following two lines:

```
number = 400
'BLORT' 'hello world' 'hi' number
```

ARexx will separate 'BLORT' into one REXXArg, storing the address in the first rm_Args field (at an offset of 40). Then, 'hello world' will be separated into one REXXArg with the address stored in the second rm_Args field (at an offset of 44). This solves the problem of imbedded spaces in arguments since ARexx is doing the parsing for you. 'hi' will be the third rm_Args string. The fourth rm_Args string will be the value of the variable "number". Since number was assigned equal to 400, ARexx substitutes this before parsing to produce a fourth rm_Arg string of '400'. The lowest nibble of the rm_Action field will be set to the number of arguments. In this example, that's 3. (A REXXMsg can hold up to 15 tokenized arguments since it has 16 rm_Args fields. The first rm_Arg field is always reserved for the command.) The one problem with setting up your ARexx implementation to expect tokenized arguments is that the user may write and invoke a script that doesn't tokenize — the more common scenario. Of course, you can always check the rm_Action's nibble to see if any arguments were tokenized, or if you have to parse, args from the one rm_Args string yourself (i.e., rm_Action's low nibble = 0).

If you set the RESULT bit (17) of rm_Action, this means that you allow ARexx (or some other application) to return a REXXArg in the REXXMsg's rm_Result2 field. You MUST set this bit if you expect some returned string there. Otherwise, no one else should ever return a REXXArg to you. You are responsible for freeing any returned REXXArg even though you may not have allocated it.

If you set the NONRET bit (20), then this means that you don't want ARexx to ever return your REXXMsg. ARexx will free the REXXMsg and all allocated REXXArgs on your behalf. Of course, this means that you'll never know if the REXXMsg was handled successfully since you won't be able to examine the returned rm_Result1 field. Also, you can't get any returned REXXArg in the rm_Result2 field.

The NOIO bit (16) causes ARexx to suppress its rm_Stdin and rm_Stdout. This can be used to prevent ARexx from printing messages to the CLI window. This may have to be done if your program is run from Workbench and has no default tool window. ARexx is not too intelligent about recognizing whether a host program was CLI or Workbench launched, and may even crash from Workbench.

If you set the STRING bit (18) of rm_Action, this means that the first rm_Arg is not the name of some ARexx script to invoke, but instead a complete script to execute. In other words, you can directly pass an entire ARexx script to ARexx by placing the address of the ASCII script in the first rm_Args field, setting the STRING bit of rm_Action, and sending the REXXMsg to ARexx. Each line of the script must be separated by a semi-colon or newline. This is useful, for example, if your

program has a set of macro keys. You could allow the user to write and bind an ARexx script to each macro key. Then, instead of requiring ARexx to access some disk in order to load a script, you can pass the script directly, eliminating any disk swapping by the user or load delays.

If your program both accepts REXXMsgs from ARexx and also can create and send REXXMsgs to ARexx, you can use one public port provided that you can recognize whether a REXXMsg queued there is a reply to one that you created, or simply one that ARexx wants you to implement. The best way to tell is to examine the REXXMsg's MN_REPLYPORT (at an offset of 14) to see if it is the same address as your port. If so, this is your own REXXMsg being returned and you must free it now. If not, then some ARexx script has just sent a command to your program and expects you to handle it, then reply the REXXMsg.

Alternately, you might want to add two public ports, having one for ARexx scripts to ADDRESS to, and the other for your program to get back REXXMsgs that you create and send to ARexx. Note that you should NEVER alter the rm_Args, rm_Action, or most other fields of REXXMsgs that are sent to you from ARexx. In this case, you are only allowed to modify the rm_Result fields.

WARNING: The ARexx server stomps all over structures which are not its own. It may alter fields of your task control block, and will definitely alter fields in a REXXMsg that you send it while the message is in its possession. (For example, ARexx alters your MN_REPLYPORT field while using the REXXMsg).

AN EXAMPLE

Now that you know how ARexx works, and how your program communicates with it, I'd like to introduce my own rexapp library. This library makes adding an ARexx implementation to any program very easy. It helps you setup a unique public port that you can Wait() for REXXMsgs to arrive at. It takes care of allocating and freeing all REXXMsgs and REXXArgs. It also deciphers whether a REXXMsg is one that you sent out, or one sent from ARexx. For ones that you sent out, it determines whether the result was an error or success, and calls one of two routines in your program accordingly. For ones sent from ARexx, it checks a "list" of your program's recognized commands against the received command. If found, it calls a "dispatch" routine in your program, passing the address of the function that you've attached to that command. It takes care of replying REXXMsgs. It also has a routine to close your port down. In short, it completely manages almost all of the ARexx details on your behalf via just a half-dozen, easy-to-use functions. The only thing that your program needs is a special data structure for the library, a list of your command strings (along with the addresses of functions that are associated with those commands), and the actual functions.

To demonstrate how to add an ARexx implementation (using my rexxapp library), I'm going to take an existing program and add the extra ARexx code. The program that I've chosen is an assembly listing that appears in a series of previous *Amazing Computing* articles which I wrote. I chose this program because it has several features that can be controlled over ARexx. Therefore, I can add an ARexx implementation with several commands. Also, since the text of those articles appears on this disk, you can get a detailed description of that program by reading the file, "Paint.txt", and this article can concentrate solely on the ARexx code. Read "Paint.txt" now, and study the original source.

The first step in adding an ARexx implementation is to decide upon the name of your public port. I'll choose 'Paint'.

The next step is to decide what features of the program you wish to be accessible over ARexx. For example, my Paint program allows the user to select one of three pen numbers for the drawing color. Let's allow this to be done over ARexx. We need to decide upon some command string. I'll arbitrarily choose 'COLOR' for this command. Furthermore, I'll expect one argument — the pen number. So maybe the following line will appear in an ARexx script:

```
ADDRESS 'Paint' 'COLOR' 1
```

ARexx will send a RexxMsg to my Paint port with the first rm_Args field being a RexxArg string as so:

```
COLOR 1
```

The rexxapp library will look through my list of commands, find the 'COLOR' command, and call its associated function (to set the drawing pen), passing the argument '1'. My function will then set the pen to 1.

For my Paint program, I'll also have the commands, 'TOFRONT' and 'TOBACK' to move the window to the front and back respectively. The 'QUIT' command will cause the program to exit. The 'LINE' command will draw a line between two points. This command will take 4 arguments: the x start position, y start position, x end position, and y end position. The 'BOX' command will draw a filled rectangle. It has 4 arguments: left x position, top y position, right x position, bottom y position.

Finally, the 'VERSION' command will cause the Paint program to return a RexxArg string to the script. The string will be the title of my program. Note that my script must have OPTION RESULTS for this to work. ('VERSION' is the only one of my commands that returns a RexxArg string. The others do not.)

Now that I've decided upon the ARexx commands that my program understands, taking care not to name a command the same as one of ARexx's built-in commands, I'm ready to add code to my program.

I need to write a function that implements each command. For example, I wrote rexxtofront to be called whenever I receive the 'TOFRONT' command. The function rexcline is called whenever I receive the 'LINE' command. If you've written your code with ARexx in mind, features that can be accessed both by the user and over ARexx can often share the same code. RexxPaint.asm is my program with the ARexx implementation added.

Now, let's examine how the rexxapp.library works. Make sure that this file is copied to your LIBS: drawer.

This library operates upon what I call a RexxData structure. There is an instance of this at the label, _rexPort. This structure has an imbedded port that I'll use for ARexx communication, plus various other fields as described. It also has what I call an association list. This is a series of LONGS. The first long is the address of one of my ARexx commands, and the next long is the address of the function associated with that command. This is used by rexxapp lib to determine if a received RexxMsg has a command that my program handles, and if so, what routine is to be executed as a result of receiving that command.

When a match is found, the lib will call a function in my program called myDispatch, passing the address of the received RexxMsg, the string containing any arguments, and the address of the function which is associated with that command. It is up to myDispatch to do any initial setup if needed, call that function, and setup the RexxMsg Result fields upon return. There is a doc file for using the rexxapp.library called "RexxAppLib.doc". You should read that now.

First, I need to open the rexxapp.library and set up my port, named 'Paint'. Since this is only done once, I added the call to the open_libs routine at label RX1. I store the lib base address at _rexBase. At RX2, I put the base of my RexxData structure in a5. All of the rexxapp lib routines require that this structure's address be in a5. The lib function SetRexxPort completely sets up the port. After this call, I'm ready to receive or send RexxMsgs. Note that my RXname has an extra null byte at the end. The reason is that the lib resolves any port name collisions by appending a number to the name. The first copy of my program adds a port called 'Paint'. If I run a second copy of the program, its port name is 'Paint2'. If there's an error setting up the port, SetRexxPort will set the z-flag and return an error message in a0. I simply print that message to my window.

It's always safe to call any other rexxapp lib routine even if the port is not setup. In this case, the lib routines will do nothing.

Now, in my main IDCMP loop, I have to Wait() on both my Intuition window port AND my ARexx port. At RX7, I do this by creating a mask of both the Intuition port's SigBit and

the mask that rexxapp lib made for me when I called SetRexxPort. This mask is in my RexxDATA structure (at the label RexxMask).

At RX6, I've added a call to ReceiveRexx. This is the lib function that will handle any received RexxMsgs by calling my dispatch function. It also handles any returned RexxMsgs that I sent to ARexx, calling my ErrorVector (printRexxErr) or ResultVector (doResult) depending upon whether there was an error or successful result respectively. It handles all RexxMsgs that are currently queued at the port. When there are no messages at the port, it returns.

Note that when I wake up from Wait() at label E6, I could add code to check the mask in d0 to see whether I woke up because of an IntuiMessage and/or a RexxMsg received. Then I could choose whether to GetMsg(), or ReceiveRexx(), or both. Instead, I just check both ports anyway because this doesn't take too much time.

Let's assume that some ARexx script is ADDRESSing our port. When I call ReceiveRexx, this causes the library to call myDispatch. First, I clear a few variables. Then I save my window's current PenA setting, and change it to the last value used with ARexx, stored at rexxPenA. This is just so that if I receive an ARexx 'COLOR' command, this doesn't change the user's Pen choice too. Note that I can find a window's current PenA value by getting it from the RastPort. Next, I call the vector that the lib passed me. This will be one of my ARexx handlers depending upon which command was received. Let's assume that the command was 'LINE 10 10 20 30', so I'm calling rexxline.

In rexxline, I need to parse the command line for 4 additional numeric args. The lib has passed me the '10 10 20 30' part of the command line. I call GetBoxCoords to parse those 4 args, convert them to actual values, stuff them into some variables, and do some error checking that the values make sense and are within my window borders. I Move() to the location specified by the first 2 args, and Draw() to the location specified by the last 2 args. This renders a line in the color of rexxPenA.

Upon return to myDispatch, I restore the PenA to what the user selected. Then, I setup the RexxMsgs rm_Result1 and rm_Result2 fields via SetupResults. Note that if I encountered some error, I set the variable error to some non-zero value. This prompts me to return a secondary error in d1 of 10 instead of 0. If I had a Result string to return, I stored the address in ResultString. (rexxversion is my only handler that returns a result string.) If there's an error, the library won't send the result string anyway. Note that after calling SetupResults, I can dispose of my result string if desired.

Remember that each call to myDispatch represents only one command of an ARexx script (one receipt of a RexxMsg). It's entirely possible that the user may be operating some part of the program, i.e., Intuition messages are being sent simultaneously. This could result in main alternating between handling a RexxMsg and an IntuiMsg. A good example of this is to execute the ARexx script, DrawLines. While the script is executing — drawing a series of lines in the window — note that you can be drawing with the mouse. It almost appears as if two separate processes are operating on the window. In fact, two processes are simultaneously sending messages to my ports: Intuition and ARexx. Of course, my program is handling each as it arrives. Because of this, it may be necessary to keep separate variables for ARexx and Intuition's use such as I did with the PenA value, and change/restore parameters within the dispatch routine. Otherwise, the user's actions might disrupt an executing ARexx script and vice versa.

I've added the facility to invoke an ARexx script from my program. If the user types a script name in my window string gadget, I'll send this to ARexx. I do this in the gadget_up routine, when the user finishes using the string gadget. At U2, I call the rexxapp lib function, ASyncRexxCmd, passing the user entered text in a0 (StrBuffer). This function allocates and sets up a RexxMsg, creates a RexxArg based upon the contents of StrBuffer, and sends the message to ARexx. Upon return, the z-flag is set if an error, and an error message is in a0. Note that I can now change the contents of StrBuffer if desired. An important thing to note is that ARexx may not even have started the script when ASyncRexxCmd returns. My program is simply telling ARexx to invoke the script, and returns even while ARexx is loading and proceeding to execute that script. For example, you can execute the DrawLines script by typing its complete path in the Paint window's string gadget and pressing return. This is an example of a program invoking a script which controls itself. Make sure that the ARexx server is started via RexxMast!

Finally, I demonstrate how you can send a string file (instead of a script invocation) to ARexx. I added a new "ARexx" menu item to the Project menu. When this is selected, I send a small, string file that I constructed at the label myStringFile. Note that this doesn't have to be a RexxArg structure, but should be null-terminated. Also, both the RXCOMM and RXSTRING bits are set. This string file just draws a small rectangle in the upper left window corner.

When the program exits, I have to remove my port and close the rexxapp library. This is done in the quit_all routine at the label RX5.



Collectible Disks!

The Fred Fish Collection

Choose from the entire Fred Fish collection and get your disks quickly and easily by using our toll free number: **1-800-345-3360**.

Our collection is updated constantly so that we may offer you the best and most complete selection of Fred Fish disks anywhere.

Now Over 400 Disks!

Disk prices for AC's *TECH* subscribers:

- 1 to 9 disks – \$6.00 each
- 10 to 49 disks – \$5.00 each
- 50 to 99 disks – \$4.00 each
- 100 disks or more – \$3.00 each

(Disks are \$7.00 each for non-subscribers)

**You are protected by our no-hassle,
defective disk return policy***

To get **FAST SERVICE** on Fred Fish disks, use your Visa or MasterCard and

call 1-800-345-3360.

Or, just fill out the order form on page 95.

*AC's *TECH* warrants all disks for 90 days. No additional charge for postage and handling on disk orders. AC's *TECH* issues Mr. Fred Fish a royalty on all disk sales to encourage the leading Amiga program anthologist to continue his outstanding work.



Adding Help to Applications Easily

*Implement an 'on-line' help facility in your applications using this powerful, yet easy-to-use, arsenal of functions...
Just call help()!*

by Philip S. Kasten

Since the Amiga's introduction, a key labelled "Help" has been prominently displayed on the keyboard. However, very few applications make use of the key. Of course, there are many other ways to allow the user to request help; a help menu or gadget are two mechanisms that quickly come to mind.

Although the Help key is easy for a user to access, perhaps it has been inconvenient for programmers to integrate into code. Applications that use VANILLAKEY instead of RAWKEY don't get IDCMP messages when the user presses the Help key. However, it is strongly suggested that effort be taken by Amiga programmers to provide some means for the user to get "on-line" help from applications. For this very reason, the following arsenal of functions has been written; these functions are invoked behind the scenes when an application makes a simple call to help().

The major goals set for the design were:

- 1) It shall be easily invoked by applications.
- 2) All help text shall be in a file separate from the application and should not require anything to be recompiled when the help text changes. This allows for last minute editing of the help file without impacting the application code itself. There cannot be any side effects in the application due to editing the help text file. Also, by having a separate file for help text, the application code can be made language independent (English, French, etc.). In fact, the application can use "help text" files to determine the strings for Intuition menus, gadgets, etc.
- 3) The application can specify which font the help text is to be displayed in. The help facility must work with ANY font, including proportionally spaced ones.

"It is strongly suggested that effort be taken by Amiga programmers to provide some means for the user to get 'on-line' help from applications."

- 4) The help text file(s) shall be keyed, so that the application would request help for a certain string or topic and the help facility will automatically display the appropriate text.
- 5) Neither the application nor the help text file itself are to be bothered about formatting the help messages. Word wrap shall happen automatically, based on font and window sizes.
- 6) Different type styles, drawing modes, and text colors shall be easily specified by the author of the help text file.
- 7) The help text file can specify the location and size of the window for each help text entry.

It is believed that all of the above goals have been met, and I would encourage readers to examine the code, use it, modify it, and/or learn from it. At the end of this article, other uses for the code and possible future enhancements to it will be described.

The help utility was written and compiled using Manx Aztec C 3.6a. It was written to be compiler non-specific (except for compiler-specific header files), but it has not yet been verified that the code will compile using any other compiler.

Design Overview

The design was started by identifying the above goals and then deciding how help text files should look, how the user interface should work, and how the application should invoke the help facility. Each will be quickly described below.

Format of Help Text Files

The help text files are ASCII files. They are composed of key lines and help text. An example of a help text file is shown in Listing One. Key lines start with the '@' character, which is immediately followed (no intervening spaces) by a key string. Following the key string is another '@' and then an optional set

of values used to specify the location, size, and sizing limits of the window that will display the help text. Default values for the window will be used for any of the above that are not specified by the key line.

Following the key line are the lines of text that are to be displayed when the application requests help for the key in the associated key line. These lines of text may contain special escape sequences that allow the author of the file to specify drawing mode (JAM1, JAM2, COMPLEMENT), text style (italics, bold, underlined, plain) and front and back pen colors. Because word wrap occurs automatically, and extra white space is normally stripped out, an escape sequence to force a space into the help display is provided. Another escape sequence forces a new line to occur. Table One lists the available escape sequences. The two character sequence '\@' is used to indicate the end of each help text message.

The User Interface

When the help facility is invoked, the user will be presented with a window containing the help text. If there is more help text than will fit in the window at one time, the user can scroll through the text using the mouse and a proportional gadget or with the cursor keys. Table Two lists the usage of the cursor keys alone as well as with the SHIFT and ALT qualifiers. It was felt that the user should be able to completely access the features of the help facility from the keyboard as well as from the mouse since some applications, such as text editors, may be keyboard intensive. The user can resize the help window within the limits specified by the key line for the message; the key line can also prevent the user from resizing the window.

The Application Interface

Aside from opening intuition.library and graphics.library, the only things the application needs to do is open the help text file for read access and call help() with the correct values for its four arguments: the file pointer; a pointer to the screen where the help window should open (or NULL if the help

Table One
Escape Sequences Available

\B	Bold on
\b	Bold off
\U	Underline on
\u	Underline off
\I	Italics on
\i	Italics off
\N	Bold off, Underline off, Italics off
\J1	JAM1 Drawing Mode
\J2	JAM2 Drawing Mode
\X	COMPLEMENT (XOR) Drawing Mode
\Cn	Color number to use for the Front Pen (n can range from 0 to the maximum number of colors in your palette).
\Rn	Color number to use for the Back Pen. (\B was already used — \R was chosen for "Rear Pen").
\s	This will cause a space to be displayed without allowing the token to be split across two lines. This will allow for uses like P.\s.\sKasten — the entire name will be treated as a single token. Multiple spaces can be forced (\s\s\s will create three spaces).
\\	Use this to put a backslash in your help message.
\n	This will force a new line.
\@	This indicates the end of a message.

Table Two
Cursor Keys

The keyboard, as well as the mouse, can be used to scroll through help messages.

Up	Scroll up one line
Down	Scroll down one line
Shift-Up	Scroll up one page
Shift-Down	Scroll down one page
Alt-Up	Go to the first page of the message
Alt-Down	Go to the last page of the message

In addition, the ESC key can be used to close the help window.

window is to open on the WorkBench screen); a pointer to the null-terminated string to be used as the key and a pointer to the TextFont structure used to display the help text (or NULL if the default RastPort font is to be used). If a font is specified, it must already be opened by the application. Once called, the help() function won't return to the application until the user has closed the help window.

As can be seen from the above descriptions, all three "interfaces" are simple to use. Once the look and feel of the design was chosen, the actual algorithm design took place. The design was broken into three main steps:

- 1) Locate the text for the specified key.
- 2) Read all the help text into a set of data structures that accounts for window size and text styles.
- 3) Present the user with the formatted text.

The second step is really the workhorse of the project and is broken down into many smaller steps and, therefore, functions. Since each of these functions needed access to similar information, a HelpInstance data structure was created. A pointer to this structure is passed to all the functions. This allowed the avoidance of global variables, and kept all the control data in one place. The next few sections will describe the three major steps in greater detail. However, to gain a complete understanding of the project, the reader is referred to the heavily commented source code.

Locating the Key String/Help Message

The format of the help text files greatly simplifies the search for the correct help text. Entire lines are read in one at a time. If the line that is read doesn't start with '@', the line is discarded and the search continues. If a line is found that starts with '@', the next characters are compared with the key string. If a match is found, this line is the key line and all subsequent lines (until a special delimiter, or the end of the file, is found) become the text that will be displayed to the user. The special delimiter is the two character sequence '\@'. If the end of the file is reached before a match is found, the screen will flash and help() will return.

A design decision was made to claim that a match is found if the specified key matches the beginning (but not necessarily the entire) string on the key line. For example, if the help text file contained a key line that read:

```
#FunctionKeys#
```

and help() was invoked as in:

```
help(file, NULL, "Fun", NULL)
```


Then a match would be considered to have been found. This design decision was made anticipating applications which allow the user to specify the desired key string — the user only has to type in enough characters to get the unique help text entry. However, note also that the comparison is case sensitive. It is quite easy for an application and help text file author to create a case-insensitive environment for the user.

The key line can contain (optional) window values. The complete format of the key line is:

```
#key string:left/top/width/height/min_width/min_height/max_width/max_height
```

Note that a '@' character is required between the key string and the window values. The key string may contain any printable characters (including spaces) except for the '@'.

The length of the entire key line cannot exceed 80 characters. Default values will be used if the window values are not specified. The window values must be specified in the order shown, but not all need to be specified. If it is desired that the default minimum and maximum values for window size be used, those values need not be placed on the key line. However, a value cannot be skipped; i.e., if the programmer needs to specify the left edge and the width of the window, the top value must be specified. For those cases, placing a value of -1 in the location for the value you don't want to specify will cause the default value to be used. Default values for the window specifications are shown in Table Three. Note that the key string will become the title of the help window.

Comments (text that will never be displayed by the help utility) can be placed in the help text at the beginning of the help text file, at the end of the file, and between messages in the file. Listing one illustrates the use of comments in help text files.

Reading and Processing the Help Text

Once the key string has been located and the window specifications determined, the help window is opened on the screen specified by the application. Next, a data structure named *HelpInstance* is allocated and initialized.

The *HelpInstance* data structure is a repository for "global" information. For example, it holds pointers to the help window, screen and font, and values of window size, drawing mode, pen colors and softstyle. It also keeps track of line indexes and buffers for token processing. Other values stored in the *HelpInstance* data structure are as follows: the number of pixels that can be in one line of help text (calculated from the window width), the number of pixels used in the current line, the number of characters in the current token, the total number of help lines in the help message, the line number of the top line displayed in the window, the number of help lines displayable in the window (based on font height and window height), some flags, and an *Intuition RememberKey*. Using the

Table Three
Default Window Values

Top:	20
Left:	0
Height:	custom screen height/2 or GfxBase->NormalDisplayRows/2
Width:	custom screen width or GfxBase->NormalDisplayColumns
Min Height:	The above value for Height
Min Width:	The above value for Width
Max Height:	-1 (maximum window height)
Max Width:	-1 (maximum window width)

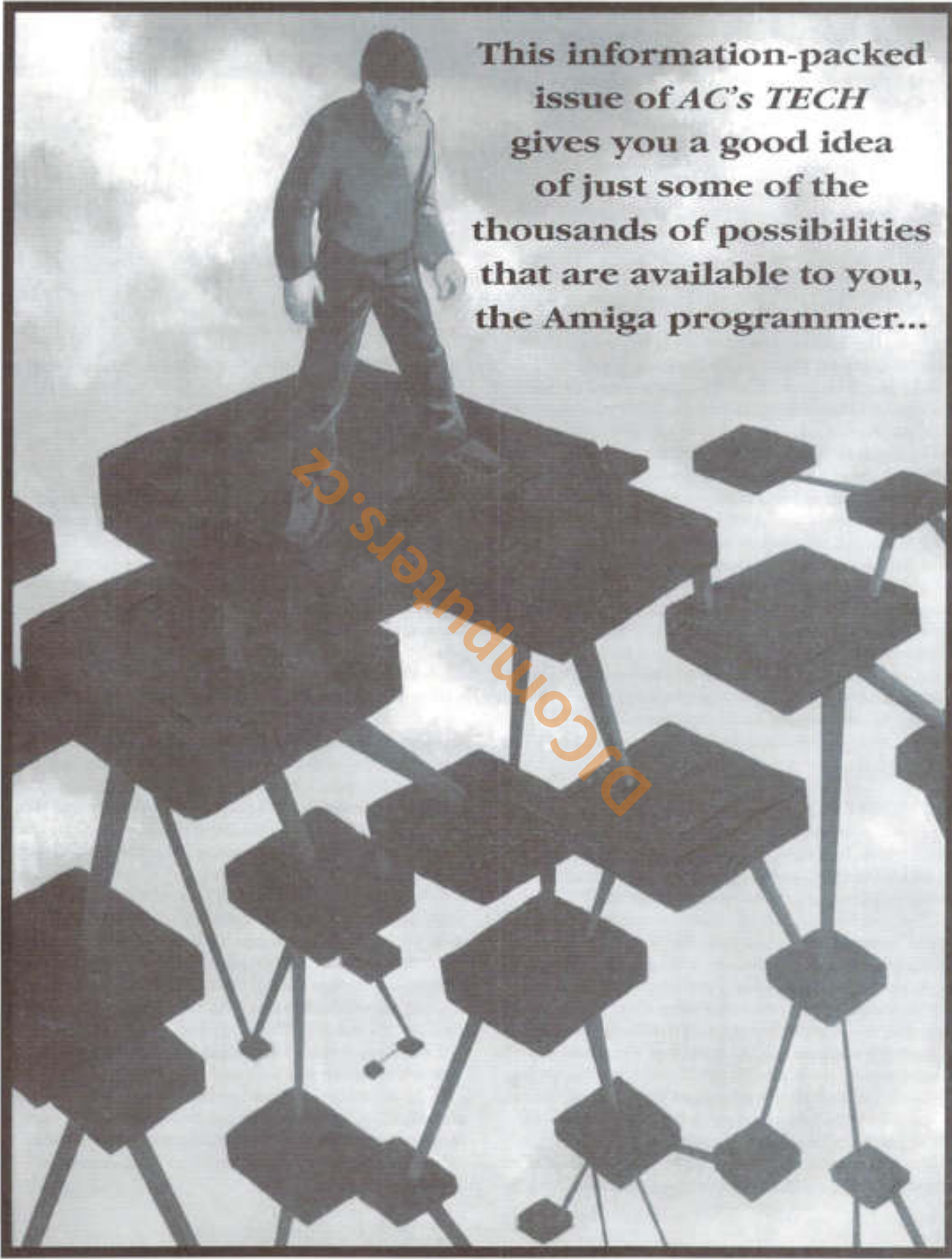
In addition, the window title will be the key string, as specified by the help text file.

HelpInstance data structure eliminated the need to use global variables. A pointer to this data structure is passed to every function comprising the help facility.

The general approach taken to process the help text is as follows. The number of lines of text that can fit within the help window, as well as the number of pixels in a line, are calculated from the window size and font size. The help text is built one line at a time, each line being composed of one or more segments. A segment is a piece of text that is to be displayed in a single drawing mode, pen color and softstyle (from here on, these three characteristics will be called the text style). A segment is composed of one or more tokens. A token is usually a word (a string of characters delimited by white space in the help file). However, if a "word" in the help file is made up of multiple text styles, or if the word is larger than the size of the buffer used for tokens, the word will be broken into multiple tokens. The entire help file is read in and processed before the help text is displayed in the window.

A line is simply a linked list of segment structures (*HelpSegmentStruct*). A line index (*HelpLineIndexStruct*) is used to point to the first segment of each line. Line indexes are fixed in size; if there are a greater number of lines in the help text than can be kept track of in a single line index, additional line indexes will be allocated. Line indexes are linked together such that the line to be displayed after the last line in index N is the first line in index N+1. Figure One should greatly clarify how all the data structures fit together.

During the course of reading and processing the help text, many data structures are allocated. The *Intuition RememberKey* facility is used to track all memory allocated except for the memory allocated for the *HelpInstance* data structure.

A man in a dark jacket and trousers is walking across a path made of various computer hardware components, including circuit boards and connectors, which are arranged like stepping stones. The background is a bright, cloudy sky. The overall image has a vintage, slightly grainy aesthetic.

**This information-packed
issue of *AC's TECH*
gives you a good idea
of just some of the
thousands of possibilities
that are available to you,
the Amiga programmer...**

DiComputers.cz

Since you are dedicated to
mastering your Amiga and using it
to its fullest potential – be sure to
maximize your possibilities
when it comes to required reading, too!

That is easily accomplished by
subscribing to the “Amazing” family of
high-quality AC publications, including
the *original* monthly Amiga magazine,
the *only* complete Amiga product guide,
and now, the *first* disk-based,
all-technical Amiga publication!

Amazing Computing.
AC's GUIDE.
AC's TECH.

When you consider the many
possibilities confronting you today –
AC publications represent Amazing values
you simply can't afford to overlook.

call 1-800-345-3360

(credit card orders only; please have Visa or Mastercard ready)

Displaying the Help Text

Once all the text has been read and processed, it is displayed to the user. As mentioned earlier, the user can scroll through the text and resize and drag the window. If the window width is changed, the help text is read and processed again before being displayed. The reason is that the number of pixels per line has changed and word wrap will likely need to occur in different places. (Because all the segments in a line are linked together, it is actually relatively simple to recalculate the display without re-reading the file. Perhaps a future version of this utility will use such an approach.)

The line indexes allow for quick updating of the display when the user scrolls the text. By processing and formatting the text prior to displaying it, very little work needs to be done to display the text.

Using help() In Your Applications

The purpose of this set of functions is to provide a simple, consistent method for applications to incorporate context-sensitive, on-line help. In order for an application to use these routines, the application must do the following:

- 1) Open intuition.library.
- 2) Open graphics.library.
- 3) Open the appropriate help text file for read access.
- 4) Call help() with the appropriate arguments. A prototype of the help call is:

```
BOOL help(FILE *helpfile, struct Screen *screen,  
char *helpkey, struct TextFont *font)
```

If a pointer to a custom screen is given, the screen must already be open. NULL can be passed in for the screen argument if the help window is to open on the WorkBench screen. If a non-NULL pointer to a TextFont structure is passed, the font must already be open. Note that help() will return FALSE if the specified helpkey was not located in the helpfile; otherwise, it will return TRUE. It should be obvious that there is very little work the application programmer need do to use this code in his or her product.

Caveats

Of course, this utility cannot be everything for everybody; in fact, it wasn't meant to be. It was designed to be both easy to use and full of features, but each additional feature increases the size and complexity of the tool. As is usually the case, certain compromises were made in the design; these compromises and design decisions are listed below. Of course, any or all of these may be addressed by modifying the existing code.

- 1) Tokens with multiple text styles may be broken across lines. For example, a token such as "Partially\Uunderlined\u" may end up being displayed with the word "Partially" at the end of one line and the word "underlined" at the beginning of the next (of course, "underlined" will be underlined). A simple way to circumvent this problem is to force a line break to occur prior to the token ("\\nPartially\Uunderlined\u").
- 2) Similarly, long tokens (even of a single text style) may be broken across lines. This is relatively unlikely, since with the values currently chosen in the source code such a token would need to exceed 30 characters.
- 3) If a window is so narrow that a token is found that cannot fit on its own line, the remainder of the help text (beginning with that token) will not be displayed. For example, suppose the window is 50 pixels wide and a token is found that is 60 pixels long. This token cannot fit on an otherwise empty line. All help text processed up to this token will be displayed; all help text beginning with this token through the end of the message will not be displayed. If the window is resized so that the token can fit, the help text will be displayed.
- 4) Allowing the user to change the width of the help window can cause a carefully laid out display to look different than planned.

The bottom line with all these caveats is that the help text file should be considered to be part of your application, requiring careful designing, testing and debugging. However, since no code needs to be recompiled, the turnaround time is extremely short. Once you have entered all your text in the help text file, start your application and invoke each help message. Make sure that the colors are all well chosen, that the text lines up as desired, that all the text is displayed, that window sizing has no negative effects, etc.

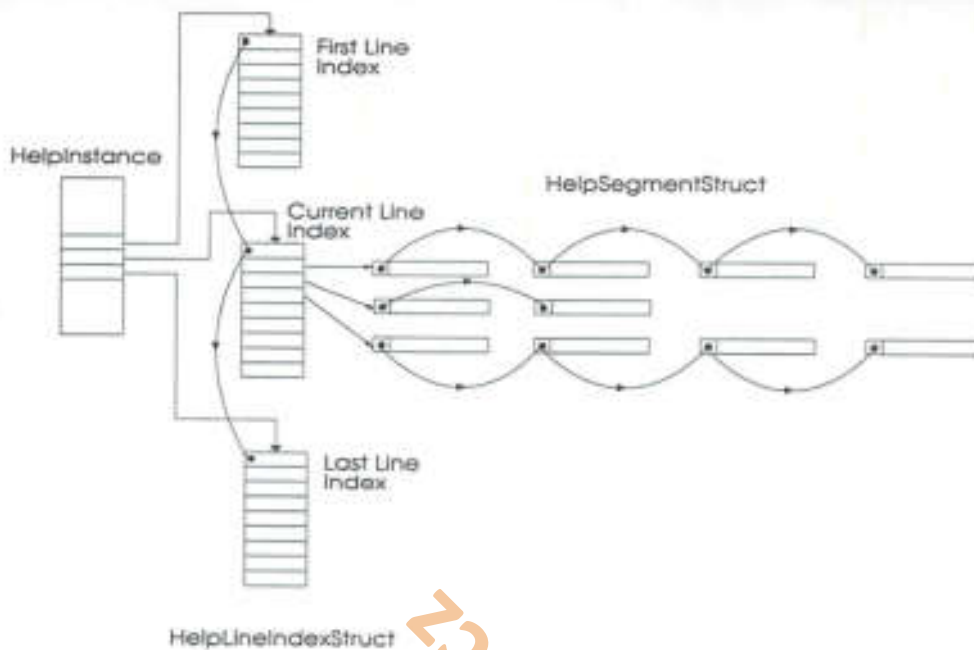
Future Enhancements

No software project is ever truly complete. Below are some ideas for possible future work:

- 1) A change in the window's width won't require the help text file to be re-read. Since most help messages are short, it was felt that it was no real problem to have the message re-read. However, since all the text is already processed and all the segments are linked together, it is a fairly straightforward job to rearrange the segments and tokens in place.

Figure One

How the Help data structures 'fit together.'



- 2) If a help file contains many long messages, it would be better to be able to immediately locate the position in the file where the message begins. This can be done via another file, an index file, which would be created by a tool run on the help text file. This index file would indicate the file position to seek to for each key string in the help text file.
- 3) Allow for multiple help windows. Perhaps an asynchronous interface would be nice so that applications can have multiple help messages displayed simultaneously.
- 4) If enough applications use this facility, perhaps creating an Amiga shared library (help.library?) would keep the code size overhead low.

Other Suggestions for Use

Although it has constantly been referred to as a "help" utility, there are many other uses for the code. In fact, by simply using the GetHelp command included on disk (the rather short listing appears in Listing Two) it is trivial to create these applications. Three examples are a phone book, a recipe file and a CLI command help utility. All that would need to be done would be to create the help text files (for example Phone.help, Recipe.help and CLI.help). If these files are placed in the S: directory, the following could be done:

```
Alias Phone GetHelp S:Phone.help
Alias Recipe GetHelp S:Recipe.help
Alias CLIShelp GetHelp S:CLI.help
```

Finally, you can get your recipes for pancakes or apple pie by typing:

```
Recipe pancakes
Recipe "Apple Pie"
```

Notice the quotes surrounding "Apple Pie" in the above example; they are needed because of the embedded space in the key string.

The GetHelp command also allows the user to specify a font and a size. If a font is specified, the size must also be specified. The font name must be the complete name (e.g., ruby.font).

Listing One can be used as a help text file. It is included on disk in a file named Listing1. Used in conjunction with the GetHelp command, the reader can easily experiment with the various features supported by the help utility.

Conclusion

It is hoped that many of you will find this useful for your own applications. Remember that the software you create is for the user; anything that makes the user's experience with your product easier is a plus. Requiring the user to search through pages of documentation to (re)discover how to perform at least the basic functions of your application can be avoided with a set of subroutines such as those presented here.

[SAS/Lattice 5.10 compatible versions of the help listings are supplied on the AC's TECH Disk—Ed]

[illegible]

10

```

static struct NewWindow NewWindow;
struct Window *window;
struct WindowVitalsStruct winvitals;

if (!file)
    return FALSE; /* If we were called with a file that didn't open
                  * we will return. This is in case the application
                  * (apparently) didn't check the result of the fopen()
                  * call.
                  */

/* Here we establish some defaults for the help window. If the
 * application passed a NULL screen pointer, we will open the window
 * on the Windows screen. Defaults are:
 * top: 11 pixels down.
 * left: Flush with the left edge of the screen.
 * height: Half the screen height for a custom screen, or half the
 * height specified by GDIBase->NormalDisplayRows.
 * width: The width of the custom screen or the width specified by
 * GDIBase->NormalDisplayColumns.
 * title: The key string. It will be initialized by the
 * helpInstanceKey() function.
 * min height and width: Same as the height and width.
 * max height and width: The size of the screen.
 */
winvitals.top = 11;
winvitals.left = 0;
winvitals.height = (screen ? screen->Height/2 :
    GDIBase->NormalDisplayRows/2);
winvitals.width = (screen ? screen->Width :
    GDIBase->NormalDisplayColumns);
winvitals.min_width = winvitals.width;
winvitals.min_height = winvitals.height;
winvitals.max_width = -1;
winvitals.max_height = -1;

/* Attempt to locate the help key in the help text file. If the
 * key is found, the winvitals structure will be updated to reflect
 * key specific values and the file will be positioned at the help
 * text for the key read. If the key is not found, flash the screen
 * and return. Note that we haven't allocated anything yet, so we
 * don't have to free anything.
 */
if (HelpLocateKey(file, key, &winvitals) == FALSE)
    DisplayBeepScreen();
return FALSE;

}

filepos = Tell(file); /* Remember where the help text for this key
                      * started.
                      */

/* Open the help window.
 */
NewWindow.LeftEdge = winvitals.left;
NewWindow.TopEdge = winvitals.top;
NewWindow.Width = winvitals.width;
NewWindow.Height = winvitals.height;
NewWindow.DetailPen = 0;
NewWindow.BlackPen = 1;
NewWindow.Flags =
    SINGLE_REFRESH | ACTIVATE |
    WINDOWCLOSE | WINDOWMAX |
    WINDOWMIN | WINDOWMOVE |
    WINDOWRESIZE | WINDOWZORDER;

NewWindow.IDCFlags =
    HELP_IDCW;
NewWindow.FirstGadget =
    NULL;
NewWindow.CheckFlag =
    NULL;
NewWindow.Screen =
    SCREEN0;
NewWindow.BitMap =
    NULL;
NewWindow.Type =
    (screen ? CUSTOMSCREEN : WINDOWSCREEN);
NewWindow.MinWidth =
    winvitals.min_width;
NewWindow.MinHeight =
    winvitals.min_height;
NewWindow.MaxWidth =
    winvitals.max_width;
NewWindow.MaxHeight =
    winvitals.max_height;
NewWindow.Title =
    GETTEXT("winvitals.title");

window = OpenWindow(&NewWindow);

/* If we couldn't open the window, return. We still haven't allocated
 * anything.
 */
if (window == NULL) return TRUE;

/* Now that the window is open, let's attach the prop gadget to the
 * right window border.
 */
HelpPropInfo.Flags = AUTOCLOSE | FREEWINDOW;
HelpPropInfo.VertPos = 0;
HelpPropInfo.VertSize = WINDOW0;
HelpPropGadget.LeftEdge = window->BorderRight + 1;
HelpPropGadget.TopEdge = window->BorderTop + 1;
HelpPropGadget.Width = window->BorderRight - 1;
HelpPropGadget.Height = window->BorderBottom - window->BorderTop - 1;
AddGadgetToWindow(&HelpPropGadget, -1);
RefreshList(&HelpPropGadget, window, NULL, 1);

/* We now allocate the helpInstance structure. This is our repository
 * for global information needed throughout the help process. If
 * we can't allocate it, we will close the window and return.
 */
hi = HelpAllocateHelpInstance(file, screen, window, key, &winvitals,
    100);

if (!hi)
    CloseWindow(window);
return TRUE;

```

```

/* Now we start doing some interesting work. We will stay in this
 * loop until the user has had enough help.
 */
for (i = 0; i < MAX_LINES_FEB_INDEX; i++)
{
    /* Position the file to where the help text for this key begins.
     * The first time we get here, this is redundant. However, we can
     * get here again if the user resizes the width of the help window.
     */
    rewind(file, SEEK_SET, 0);

    /* We don't know how many lines of text are in the help message.
     * We allocate a line index capable of keeping track of
     * MAX_LINES_FEB_INDEX lines. If the index gets full, we'll
     * allocate another one.
     */
    helpAllocateLineIndex(hi);

    /* At first it might appear that I haven't verified that I
     * actually allocated the index. But the for() loop below will
     * terminate immediately if we couldn't since hi->currentIndex
     * will be NULL (see helpAllocateLineIndex()).
     */
    for (indexOffset = 0; hi->currentIndex; indexOffset++)
    {
        if (indexOffset == MAX_LINES_FEB_INDEX)
        {
            /* If the index is full, allocate another one and restart
             * the index count. All the indexes are linked together
             * so we won't lose any.
             */
            indexOffset = -1;
            helpAllocateLineIndex(hi);
            continue;
        }

        /* Here we will build a whole display line. This line will
         * contain one or more "segments". The line length and the
         * number of segments needed are dependent on the font used,
         * the size of the window and the number of text style changes.
         */
        rc = HelpBuildLine(hi, indexOffset);

        switch (rc)
        {
            case HELP_CONTINUE:
                /* There are more lines to create. */
                continue;

            case HELP_DONE:
                /* There are no more lines to create. */
                break;

            case HELP_FATAL:
                /* We ran out of resources somewhere along the line. */
                break;

            default:
                /* There is a bug in the software! */
                CHECK(0);
                break;
        }

        /* At this point, we have read in the help message and created
         * a complete set of linked data structures. We now are ready to
         * display the help message. Since we built our data structures
         * based on the window size, if we allow the user to change the
         * window size while we are rendering the text we might trash the
         * window buffers. Therefore, I use the SILENTIFY flag as a
         * semaphore until I clear the flag, I own the window. When
         * I clear the flag, the user can resize the window. Note that
         * I couldn't set the SILENTIFY flag earlier since we must not
         * use any of the verify flags while accessing DOS. (Read the FPM).
         */
        ModifyIDCW(hi->window, HELP_IDCW | SILENTIFY);

        rc = HelpDisplay(hi);

        ModifyIDCW(hi->window, HELP_IDCW);

        /* If HelpDisplay() returns anything but HELP_CONTINUE, the user
         * is done. If HELP_CONTINUE is returned, the user resized the
         * window and we have to re-calculate the data structure.
         */
        if (rc != HELP_CONTINUE) break;

        /* First off, return all tracked resources to the system. The
         * only memory we allocated that won't be returned here is the
         * memory we allocated for the helpInstance structure. Of course,
         * the window structure is excluded from this, too. Also,
         * reset all helpInstance values.
         */
        FreeMemory(hi->ScreenData, (LONG)TRUE);
        hi->ScreenData = NULL;
        hi->FirstIndex = NULL;
        hi->LastIndex = NULL;
        hi->currentIndex = NULL;
        hi->screen = 0;
        hi->screenBuffer[0] = '\0';
        hi->screenBuffer[1] = '\0';
        hi->open_line_so_far = 0;
        hi->total_lines = 0;
        hi->total_width = 0;
        hi->flags = 0;
        hi->total_lines = 0;
        hi->open_line = 0;
    }
}

```



```

/* Clear the window. */
SetAPen(hi->window->Port, 0);
SetDither(hi->window->Port, JAMI);

RectFill(hi->window->Port, (LONG)hi->window->BorderLeft,
(LONG)hi->window->BorderTop,
(LONG)hi->window->BorderRight - 1,
(LONG)hi->window->Height - hi->window->BorderBottom - 1);

/* Do it all over again. */
}

/* We are all done. Free all resources, including the HelpInstance
structure and close the Window.
*/
HelpFreeAll(hi);
return TRUE;

```

```

/*-----
static struct HelpInstance *HelpAllocateHelpInstance(FILE *file,
struct Screen *screen,
struct Window *window, char *key,
struct WindowVitalStruct *wv,
struct TextFont *font);
Description: This function is used once per help() session. It
allocates and initializes the HelpInstance data
structure, a pointer to which MUST be passed to all
help() related functions.
Inputs: file: A pointer to the opened (read access) help
text file.
screen: A pointer to the screen that the help
window has been opened on. This can be
NULL if the window has been opened on the
Workbench screen.
window: A pointer to the opened help window.
key: A pointer to the null terminated string the
calling application wants to use as the
help key.
wv: A pointer to the initialized
WindowVitalStruct data structure.
font: A pointer to the TextFont structure of the
ALREADY OPENED font. This can be NULL if
help() should use the default MacFont font.
Returns: A pointer to the allocated and initialized HelpInstance
data structure, or NULL if the data structure could not
be allocated. This is the ONLY data structure that is
allocated using AllocMem() instead of AllocMemner().
As such, this must be explicitly FreeMem()'ed.
-----*/

```

```

static struct HelpInstance *HelpAllocateHelpInstance(FILE *file, screen, window,
key, wv, font);
FILE *file;
struct Screen *screen;
struct Window *window;
char *key;
struct WindowVitalStruct *wv;
struct TextFont *font;

struct HelpInstance *hi;

hi = (struct HelpInstance *) (AllocMem((LONG)sizeof(struct HelpInstance),
MEMF_CLEAR));

if (hi) {
hi->file = file;
hi->screen = screen;
hi->window = window;
hi->key = key;
hi->win_top = wv->top;
hi->win_left = wv->left;
hi->win_width = wv->width;
hi->win_height = wv->height;
hi->win_title = (CONST) "wv->title;
hi->keyMemberKey = NULL;
hi->current_drawmode = JAMI;
hi->current_fontpen = 1;
hi->current_backpen = 0;
hi->current_softstyle = hi->current_softstyle enable = 0;
hi->spn_per_line = RIGHT_MARGIN - LEFT_MARGIN + 1;

if (font != NULL) {
hi->font = font;
void SetFont(hi->window->Port, font);
} else {
hi->font = hi->window->Port->Font;
}

return (hi);
}

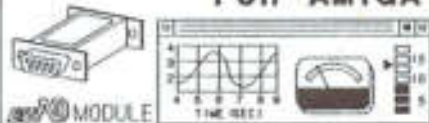
```

```

/*-----
static void HelpAllocateLineIndexStruct HelpInstance *hi);
Description: This function is used to allocate an index for a
set of lines. Each line contains one or more
segments, which are linked together. An index can
keep track of MAX LINES PER INDEX lines. If the
help message contains more lines, another line
index needs to be allocated. This function
automatically links all allocated line indexes into
the HelpInstance data structure.
Inputs: hi: a pointer to the HelpInstance data structure.
Returns: Nothing. hi->currentIndex will always point to the
allocated index. If hi->currentIndex is NULL, the
allocation failed. The data structure
is allocated using AllocMemner(), so that it will
be freed when AllocFree() is called.
-----*/

```

EZAD™ DATA ACQUISITION SYSTEM FOR AMIGA



LOW COST SYSTEM FOR MONITORING EVENTS WITH YOUR AMIGA. MEASURE, GRAPHICALLY DISPLAY AND RECORD TEMPERATURE, PRESSURE, LIGHT INTENSITY, ETC. NO SEPARATE POWER SUPPLY REQUIRED. CONNECTS TO SECOND GAMEPORT. DOES NOT INTERFERE WITH PARALLEL OR SERIAL PORT OPERATION. MULTI TASKING SOFTWARE RUNS FROM WORKBENCH. COMPLETE HARDWARE AND SOFTWARE SYSTEMS STARTING AT \$79.95. THIS PRODUCT SHOWS THE TRUE POWER OF THE AMIGA.

BOONE TECHNOLOGIES
P O BOX 15052, RICHMOND, VA 23227
WRITE FOR INFORMATION AND DEMO DISK

Circle 194 on Reader Service card.

```

static void HelpAllocateLineIndex(hi)
struct HelpInstance *hi;

struct HelpLineIndexStruct *hli;

hli = (struct HelpLineIndexStruct *)
AllocMemner(hi->MemberKey,
(LONG)sizeof(struct HelpLineIndexStruct),
MEMF_CLEAR);

if (hli) {
if (hi->firstIndex == NULL) {
hi->firstIndex = hli;
hi->lastIndex = hli;
} else {
hi->lastIndex->next = hli;
hi->lastIndex = hli;
}

hi->currentIndex = hli;
}

```

```

/*-----
static struct HelpSegmentStruct *HelpAllocateSegment(
struct HelpInstance *hi);
Description: This function allocates a segment that is to be
linked into the current line.
Inputs: hi: a pointer to the HelpInstance data structure.
Returns: A pointer to the allocated HelpSegmentStruct or
NULL if the allocation fails. The data structure
is allocated using AllocMemner(), so that it will
be freed when AllocFree() is called.
-----*/

```

```

static struct HelpSegmentStruct *HelpAllocateSegment(hi)
struct HelpInstance *hi;

struct HelpSegmentStruct *hs;

hs = (struct HelpSegmentStruct *)
AllocMemner(hi->MemberKey,
(LONG)sizeof(struct HelpSegmentStruct),
MEMF_CLEAR);

return hs;

```

```

/*-----
static BOOL HelpLocateKey(FILE *file, char *key,
struct WindowVitalStruct *wv);
Description: This function searches a file for the specified
help key. If the key is found, the
WindowVitalStruct is modified to reflect any key
specific values.
Inputs: file: A pointer to the opened (read access) help
text file.
key: A pointer to the null terminated string the
calling application wants to use as the
help key.
wv: A pointer to the default initialized
WindowVitalStruct data structure.
Returns: TRUE if the key was not found in the help text
file.
-----*/

```

```

/*
 * TRUE otherwise. In this case, *wv is modified to
 * reflect any key specific values. Also,
 * the entire key string (as specified by the
 * help text file) is copied into wv->title[]
 */
static BOOL helpProcessKey(FILE, Key, wv)
FILE *file;
Key *key;
struct WindowVitalsStruct *wv;
{
    char buffer[81]; /* 80 chars max., plus '\n' and '\0' */
    char *bufptr;
    USHORT keylength;
    SHORT temp;
    USHORT left = -1, top = -1, width = -1, height = -1,
        min_width = -1, min_height = -1;

    /* Read in the file. */
    fseek(file, 0L, 0);

    /*
     * Notice that if the key is "abc" and the file contains
     * an entry keyed by "abcdef", we declare a match. This
     * is done anticipating applications that wish to allow the user
     * to enter only enough characters to make a unique match. I
     * don't look for a unique match per se, but I do key to the user's
     * input.
     */
    keylength = strlen(key);

    while (!feof(buffer, 82, file)) {
        if (!buffer || !key) continue;
        if (strncmp(buffer, key, keylength) == 0) continue;

        /*
         * The key was found. We will now determine the window values
         * and title.
         */
        buffer[strlen(buffer)-1] = '\0'; /* Change the '\n' to a '\0'. */

        /*
         * Keep scanning the buffer until we reach the key string delimiter
         * ('\0') or the end of the buffer ('\0').
         */
        for(bufptr = &buffer[keylength-1];
            (*bufptr != '\0') && (*bufptr != '\0');
            bufptr++, keylength++);

        /*
         * We now copy the key string into the wv->title[] buffer. The
         * above loop made keylength equal to the length of the key string.
         */
        strcpy(wv->title, &buffer[keylength]);
        wv->title[keylength] = '\0';

        /*
         * If we found the key string delimiter ('\0'), we scan the buffer
         * for the window values.
         */
        if (*bufptr == '\0') {
            bufptr++;
            sscanf(bufptr, "%d/%d/%d/%d/%d/%d",
                &left, &top, &width, &height,
                &min_width, &min_height);
            wv->max_width = wv->min_width;
            wv->max_height = wv->min_height;

            /*
             * If any of the local variable are no longer -1, we want to
             * override the default values.
             */
            if (left != -1) wv->top = top;
            if (left != -1) wv->left = left;
            if (width != -1) wv->width = width;
            if (height != -1) wv->height = height;
            if (min_width != -1) wv->min_width = min_width;
            else wv->min_width = wv->width;
            if (min_height != -1) wv->min_height = min_height;
            else wv->min_height = wv->height;

            if ((wv->max_width != -1) && (wv->min_width > wv->max_width)) {
                temp = wv->min_width;
                wv->min_width = wv->max_width;
                wv->max_width = temp;
            }

            if ((wv->max_height != -1) && (wv->min_height > wv->max_height)) {
                temp = wv->min_height;
                wv->min_height = wv->max_height;
                wv->max_height = temp;
            }

            return TRUE;
        }

        /* The key wasn't found. */
        return FALSE;
    }

    /*
     * static void helpFreeAll(struct HelpInstance *hi)
     *
     * Description: This function returns all system resources
     * allocated by the help utility.
     *
     * Inputs: hi: a pointer to the HelpInstance data structure.
     *
     * Returns: Nothing.
     */
    static void helpFreeAll(hi)
    struct HelpInstance *hi;
    {
        if (hi) {
            if (hi->window) CloseWindow(hi->window);
            if (hi->memberKey) FreeMemberKey(hi->memberKey, (LONG)TRUE);
            FreeMem((char *)hi, (LONG)sizeof(struct HelpInstance));
        }
    }
}

```

```

}
return;

/*
 * static SHORT helpBuildLine(struct HelpInstance *hi,
 * SHORT indexoffset)
 *
 * Description: This function creates one line of help text. The
 * line isn't displayed yet, only read into memory and
 * processed.
 *
 * Inputs: hi: a pointer to the HelpInstance data structure.
 * indexoffset: the offset into the current line
 * index that this line should be assigned to.
 *
 * Returns: HELP_FATAL if we ran out of resources trying to
 * build this line.
 * Whatever helpBuildSegment() returns.
 */
static SHORT helpBuildLine(hi, indexoffset)
struct HelpInstance *hi;
SHORT indexoffset;
{
    struct HelpSegmentStruct *segment;
    SHORT rc;

    /* Assume we aren't going to be able to do this. */
    rc = HELP_FATAL;

    /*
     * Increment the number of lines so far. The line width in pixels
     * is zero at this point (we haven't added any characters yet).
     */
    hi->total_lines++;
    hi->pix_line_so_far = 0;

    /*
     * If there is a token left over from the previous line, we have to
     * check if one or two spaces are needed to follow the token.
     * HELP_SPACE_FLAG and HELP_EXTRA_SPACE_FLAG each indicate we need
     * one space. If either of these are set, we set the corresponding
     * "HOLD" flag: HELP_HOLD_SPACE_FLAG and HELP_HOLD_EXTRA_SPACE_FLAG.
     * Then we clear the current flags.
     */
    if ((hi->token_length > 0) && (hi->flags & HELP_SPACE_FLAG)) {
        hi->flags |= HELP_HOLD_SPACE_FLAG;
    } else {
        hi->flags &= ~HELP_HOLD_SPACE_FLAG;
    }
    if ((hi->token_length > 0) && (hi->flags & HELP_EXTRA_SPACE_FLAG)) {
        hi->flags |= HELP_HOLD_EXTRA_SPACE_FLAG;
    } else {
        hi->flags &= ~HELP_HOLD_EXTRA_SPACE_FLAG;
    }
    hi->flags &= ~HELP_SPACE_FLAG | HELP_EXTRA_SPACE_FLAG;

    /*
     * Lines are made up of one or more segments. The for() loop will
     * automatically terminate if a segment couldn't be allocated. For
     * each segment we allocate, we link it into the line and build the
     * segment. As long as helpBuildSegment() returns HELP_CONTINUE,
     * another segment is needed for the current line.
     */
    for (segment = hi->currentIndex->line[indexoffset] =
        helpAllocateSegment(hi));
        segment && !rc=helpBuildSegment(hi, segment) == HELP_CONTINUE;
        segment = segment->next = helpAllocateSegment(hi));

    /*
     * If we exited the for() loop because we couldn't allocate a segment,
     * return HELP_FATAL. Otherwise, return the result code returned by
     * helpBuildSegment().
     */
    if (!segment) return HELP_FATAL;
    return rc;
}

/*
 * static SHORT helpBuildSegment(struct HelpInstance *hi,
 * struct HelpSegmentStruct *segment)
 *
 * Description: This function builds a single segment. It will
 * return if there is no room left for more characters
 * in the segment. If the help text file indicates
 * that a change of text style is needed or if we
 * reached the end of the help text message.
 *
 * Inputs: hi: a pointer to the HelpInstance data structure.
 * segment: a pointer to the current segment.
 *
 * Returns: HELP_CONTINUE if this segment was built and another
 * segment is needed for this same line.
 * HELP_EOF if the end of file has been reached in the
 * help text file.
 * HELP_TCM if the end of the help message has been
 * reached.
 * HELP_EOL if this line is to be terminated.
 */
static SHORT helpBuildSegment(hi, segment)
struct HelpInstance *hi;
struct HelpSegmentStruct *segment;
{
    SHORT rc;

    /*
     * Each segment has its own text style. By this I mean: font pen,
     * back pen, draw mode and style (italic, bold, underlined, plain).
     * The HelpInstance keeps track of the current set of values so that
     * each segment can determine its set.
     */
    segment->frontpen = hi->current_frontpen;
    segment->backpen = hi->current_backpen;
    segment->drawmode = hi->current_drawmode;
    segment->softstyle = hi->current_softstyle;
    segment->softstyle_enable = hi->current_softstyle_enable;
}

```



```

/*
 * If there is no token left over from the previous segment, we look
 * to see if there are any text style changes being requested in the
 * help text file. Since we haven't put any text into this segment
 * yet we can change our text style values from ones we just set up
 * without violating the rule that each segment is made up of
 * text that has all the same text style.
 */
if (hi->token_length == 0) helpAddToken(hi->token, hi->file, segment);

/*
 * Now we keep adding tokens to the segment until helpAddToken()
 * returns anything but HELP_CONTINUE.
 */
while((rc = helpAddToken(hi, segment)) == HELP_CONTINUE);

switch (rc) {
case HELP_EOS:
/*
 * This means that a new text style is needed or that there
 * is no room left in this segment for a new token. We return
 * HELP_CONTINUE so that helpBuildLine() will allocate a new
 * segment for the current line and call this function again.
 */
return HELP_CONTINUE;
case HELP_EOF:
case HELP_EOL:
case HELP_EOL:
/*
 * We reached the end of the file, end of the message or
 * we know we reached the end of the line. The calling functions
 * will know what to do with these results.
 */
return rc;
default:
/* Software bug! */
CHECK_FOR_ERRORS();
}
/*NOTREACHED*/
}

```

```

/*
 * static SHORT helpAddToken(struct HelpInstance *hi,
 * struct HelpSegmentStruct *segment)
 *
 * Description: This function adds a single token to the current
 * segment.
 *
 * Inputs:
 * hi: a pointer to the HelpInstance data structure.
 * segment: a pointer to the current segment.
 *
 * Returns:
 * HELP_CONTINUE if a token was successfully added
 * and another token might be able to be put
 * in this segment/line.
 * HELP_EOF if the token just added is to be the last
 * for the current line.
 * HELP_EOL if next token couldn't fit in the current
 * segment.
 * Anything else helpGetToken() may return.
 */
static SHORT helpAddToken(hi, segment)
struct HelpInstance *hi;
struct HelpSegmentStruct *segment;
{
SHORT rc;
BYTE space_adjust;
BYTE *cursor;
CHECK_FOR_ERRORS;
char *buffer;
char *buffer;
SHORT flags;

/*
 * Assume that we will be able to add the next token to the
 * current segment.
 */
rc = HELP_CONTINUE;

buffer = segment->buffer;
cursor = segment->cursor;
flags = hi->flags;

/*
 * space_adjust will be set to the number of spaces we need to
 * add to the next token.
 */
space_adjust = ((flags & HELP_SPACE_FLAG) ? 1 : 0) +
((flags & HELP_EXTRA_SPACE_FLAG) ? 1 : 0) +
((flags & HELP_HOLD_SPACE_FLAG) ? 1 : 0) +
((flags & HELP_HOLD_EXTRA_SPACE_FLAG) ? 1 : 0);

/*
 * If there is no token outstanding (a token that was previously
 * read but couldn't be fit into a previous segment), we will read
 * the next token from the help text file.
 */
if (hi->token == '\0') {
rc = helpGetToken(hi, segment);

switch (rc) {
case HELP_LAST_TOKEN:
/*
 * The token we have just read is to be the last token
 * in the current line. This happens when a help text
 * line is terminated with "\n".
 */
hi->flags |= HELP_LAST_TOKEN_FLAG;
rc = HELP_EOF;
/* fall through */
case HELP_CONTINUE:
case HELP_EOL:
break;
default:
return (rc);
}
}
}

```

EXPAND!	EXPAND!	EXPAND!	EXPAND!	EXPAND!
MEMORY/DISK	UNIT	2ND	4TH	8TH
1MB/70/80 SIMM	\$ 53.00	110	210	400
256KB-80	6.50	104	200	384
512K-80	8.50	104	200	384
256K1-80	1.75	112	192	384
256K4-80 5C ZIP	9.50	152	296	576
1MB4-80 5C ZIP	45.00	180	360	680
100 AdRAM 540	109	199	299	-
100 AdRAM 2000	119	199	279	429
RAMAGES 2000	109	189	269	429
4VS META 4	-	259	349	-
GAIP II 4C	329	329	429	609
ADRESI 2080/40M 40	449	549	649	829
AE HO 3.5" DRIVE.....189	100 ASIDE.....119			
SL EXPRESS/RAP/FAX....209	ADIDE/TEAC 40MB...399			
SUPRA 2400 MODEM.....105	ADISPEED.....229			
TRUMP300 FPD/40M HO...499	FLICK FREE V.....329			
SUPRASOX100/40M/512K...999	AdRAM 5400/2MB...189			
ORDERS: 800-735-2633 VISA/MC/COD				
INFORMATION: 408-626-2633 FAX: 408-626-0552				
VISIONSOFT PO BOX 22517, CARMEL CA 93922				

Circle 183 on Reader Service card.

```

/*
 * char count is the number of characters that the token contains
 * plus the number of spaces that must be added to it.
 *
 * pix_count is the number of pixels wide the token is (including
 * the spaces).
 */
char count = space_adjust + hi->token_length;
pix_count = space_adjust*hi->font->dx +
TextLength(hi->window->font, hi->token,
LONG(hi->token_length));

/*
 * If the pixel size of this one token is larger than the number
 * of pixels we can fit in a whole line, we will prematurely terminate
 * the help message. It is up to the author of the help text file
 * to make sure that the user cannot size the help window too small.
 */
if (pix_count > hi->pix_per_line) {
return HELP_EOF;
}

/*
 * If the pixel size of this one token won't fit on the current
 * line (even if it at least fit on its own line from the above
 * conditional), we'll terminate the current line. This token
 * will then become the first one used on the next line.
 */
if ((pix_count + hi->pix_line_wd_fact) > hi->pix_per_line) {
return HELP_EOL;
}

/*
 * This token will go into the current line. But if the current segment
 * itself cannot hold this token, we'll terminate the current segment.
 * This token will become the first token of the next segment, which
 * will be the next segment for the current line.
 */
if (char count + *cursor > CHARS_PER_BUFFER) {
return HELP_EOF;
}

/*
 * If either HELP_SPACE_FLAG or HELP_EXTRA_SPACE_FLAG are set, the
 * previous token (on this line) needed to be followed by spaces before
 * this token is inserted into the line.
 */
if (flags & HELP_SPACE_FLAG) {
buffer[*cursor++] = ' ';
}

if (flags & HELP_EXTRA_SPACE_FLAG) {
buffer[*cursor++] = ' ';
}

/*
 * Add the token to the current segment.
 */
strcat(buffer, hi->token);
cursor += hi->token_length;

/*
 * If either HELP_HOLD_SPACE_FLAG or HELP_HOLD_EXTRA_SPACE_FLAG are set,
 * the current token must be followed by spaces on the current line.
 */
if (flags & HELP_HOLD_SPACE_FLAG) {
buffer[*cursor++] = ' ';
hi->flags |= HELP_HOLD_SPACE_FLAG;
}
}

```

```

if (!flags & HELP_HOLD_EXTRA_SPACE_FLAG) {
    buffer[(*curDoc)+1] = '\0';
    hi->flags |= ~HELP_HOLD_EXTRA_SPACE_FLAG;
}

/*
 * Adjust the pixel count for the line.
 */
hi->pix_line_so_far += pix_count;

/*
 * Since we used the token, clear out the holding buffer for it.
 */
(void)memset(hi->token, MAX_TOKEN_CHARS+1, 0);
hi->token_length = 0;

/*
 * If this token had to be the last one for this line, return HELP_EOL.
 */
if (hi->flags & HELP_LAST_TOKEN_FLAG) {
    hi->flags |= ~HELP_LAST_TOKEN_FLAG;
    return HELP_EOL;
}

return 0;
}

/*-----*/
static SHORT helpGetToken(struct HelpInstance *hi,
                        struct HelpSegmentStruct *segment)
/*
 * Description: This function reads a single token from the help
 * text.
 *
 * Inputs: hi: a pointer to the HelpInstance data structure.
 * segment: a pointer to the current segment.
 *
 * Returns: HELP_CONTINUE if a token was successfully read.
 * HELP_EOF if the end of file was reached before any
 * characters were read.
 * HELP_EOL if last token couldn't fit in the current
 * segment.
 * HELP_EOM if the end of the help text message was
 * reached.
 */
static SHORT helpGetToken(SHORT segment)
struct HelpInstance *hi;
struct HelpSegmentStruct *segment;
FILE *file = hi->file;
int c, c2;
SHORT i = 0;
SHORT rc;
BOOL done, lastWasAOct = FALSE;

hi->token_length = 0;
hi->flags |= ~HELP_SPACE_FLAG | HELP_EXTRA_SPACE_FLAG;

for(done=FALSE; !done) {
    /*
     * Read the next character from the file.
     */
    c = getc(file);

    switch (c) {
        case EOF:
            /*
             * If we have reached the end of the file, then
             * we have to see if we have read any characters for
             * this token.
             */
            if (i > 0) {
                /*
                 * This token is not empty. We will return
                 * HELP_CONTINUE. The last time helpGetToken() is
                 * called, we will return HELP_EOF.
                 */
                rc = HELP_CONTINUE;
            } else {
                /*
                 * This token is empty. Just return HELP_EOF.
                 */
                rc = HELP_EOF;
            }

            /*
             * In either case, we are done with the processing for
             * this token.
             */
            done = TRUE;
            break;

        case '\\':
            /*
             * If the character just read is the special "escape"
             * character (not the ASCII ESC character), we have some
             * special work to do. First, we will put the character
             * back into the file. Then we will process the escape
             * sequence. There are some sequences that don't cause
             * us to terminate the current segment.
             */
            (void)ungetc(c, file);
            if ((rc=helpHandleEscape(hi, segment, file, &i))
                == HELP_CONTINUE)
                continue;

            /*
             * We have reached one of the following conditions:
             * 1) end of segment
             * 2) end of message
             * 3) end of token (no more room in current token buffer).
             */
            done = TRUE;

            if (rc == HELP_EOF) {
                /*
                 * If there is any white space following this token,
                 * move the file pointer past it and remember that we

```

```

        * read a space.
        */
        if (helpSkipAllWhiteSpace(hi, file))
            hi->flags |= HELP_SPACE_FLAG;
    } else if (rc == HELP_EOF) {
        /*
         * If helpHandleEscape() returned HELP_EOF it determined
         * that there was no more room in the current token. We
         * are done getting this token.
         */
        rc = HELP_CONTINUE;
    }

    break; /* EOF, EOM, EOT */
}

default:
    /*
     * We've already handled most of the special cases. We
     * still have to handle whitespace, periods (two spaces)
     * follow periods under most circumstances) and, of course,
     * "normal" characters.
     */
    if (!isspace(c)) {
        if ((c == '\n') && lastWasAOct) {
            (void)helpSkipAllWhiteSpace(hi, file);
            hi->flags |=
                HELP_SPACE_FLAG | HELP_EXTRA_SPACE_FLAG;
            rc = HELP_CONTINUE;
            done = TRUE;
            break;
        }

        if ((c == '\t') && lastWasAOct) {
            c2 = getc(file);
            (void)ungetc(c2, file);
            if (c2 == '\n') {
                (void)helpSkipAllWhiteSpace(hi, file);
                hi->flags |=
                    HELP_SPACE_FLAG | HELP_EXTRA_SPACE_FLAG;
                rc = HELP_CONTINUE;
                done = TRUE;
                break;
            }
        }

        (void)helpSkipAllWhiteSpace(hi, file);
        hi->flags |= HELP_SPACE_FLAG;
        rc = HELP_CONTINUE;
        done = TRUE;
        break;
    }

    lastWasAOct = ((c == '\t') ? TRUE : FALSE);

    /*
     * Store the character in the token buffer.
     */
    hi->token[i++] = c;

    if (i == MAX_TOKEN_CHARS) {
        /*
         * If there is no more room in this token for more
         * characters, we'll return and let the calling routine
         * call us again.
         */
        rc = HELP_CONTINUE;
        done = TRUE;
        break;
    }

    continue;
}

hi->token_length = i;
return (rc);
}

```

```

/*-----*/
static BOOL helpSkipAllWhiteSpace(struct HelpInstance *hi,
                                FILE *file)
/*
 * Description: This function adjusts the file so that when this
 * function returns the next character read from the
 * file will not be a white space character (the file
 * may be positioned at the end-of-file).
 *
 * Inputs: hi: a pointer to the HelpInstance data structure.
 * file: a pointer to the FILE used for help text.
 *
 * Returns: TRUE if any white space was skipped.
 * FALSE if no white space was skipped or if the
 * file was at the End-Of-File when this function was
 * called.
 */
static BOOL helpSkipAllWhiteSpace(hi, file)
struct HelpInstance *hi;
FILE *file;
int c;
BOOL rc = FALSE;

while ((c = getc(file)) != EOF) {
    if (!isspace(c)) {
        (void)ungetc(c, file);
        break;
    } else rc = TRUE;
}

return rc;
}

```

```

/*-----*/
static void helpHandlePreEscape(struct HelpInstance *hi,
                                FILE *file,
                                struct HelpSegmentStruct *segment)
/*
 * Description: This function processes help text escape sequences

```



```

/*
 * static SHORT helpDisplay(struct HelpInstance *hi)
 *
 * Description: This function is called to actually display the
 * help text message for the help key. The message
 * has already been read and processed into the
 * necessary data structures. If the width of the
 * help window changes, the message will have to be
 * reprocessed before this function is called again.
 * However, while in this function, the user will be
 * able to scroll the help message (via keyboard or
 * a proportional gadget).
 *
 * Inputs: hi: a pointer to the HelpInstance data structure.
 *
 * Returns: HELP_EXIT if the user wants to quit the help
 * message.
 *         HELP_CONTINUE if the user changed the width of the
 *         help window.
 */
static SHORT helpDisplay(hi)
struct HelpInstance *hi;
{
    struct IntMsg *img;
    ULONG class;
    USHORT code, qualifier;
    BOOL redisplay, quit;
    WORD verPos, verBody;
    hidden;
    overlap = 0;
    totalLines = hi->total_lines;
    visibleLines;
    topLine = 0;

    hi->lines_per_page = visibleLines =
        (SHORT) (hi->rows - TOP_MARGIN + ROW_HEIGHT/2) / ROW_HEIGHT;

    /* The following code segment comes from FPM Intuition: gadgets page 85 */
    /* Find the number of hidden lines, those that don't fit in the
     * visibleLines portion. It turns out to be useful in further
     * calculations.
     */
    hidden = MAX(totalLines - visibleLines, 0);

    /* If topLine is so great that the remainder of the line won't even
     * fill the displayable area, reduce topLine!
     */
    if (topLine > hidden)
        topLine = hidden;

    /* Body is the relative size of the proportional gadget's body. It's
     * size in the container represents the fraction of the total area is
     * in view. If there are 50 lines hidden, then body should be
     * full-size (MAXBODY). Otherwise, body should be the fraction of
     * the number of displayed lines - overlap over
     * (the number of total lines - overlap).
     */
    if (hidden > 0) {
        verBody = (ULONG) (visibleLines - overlap) * MAXBODY / (totalLines - overlap);
    } else {
        verBody = MAXBODY;
    }

    /* For is the position of the proportional gadget body, with zero
     * meaning that the scroll gadget is all the way up (to left),
     * and full (MAXBODY) meaning that the scroll gadget is all the way
     * down (to right). If we can see all the lines, For should be zero.
     * Otherwise, For is the top displayed line divided by the number of
     * unseen lines.
     */
    if (hidden > 0) {
        verPos = (ULONG) ((ULONG) topLine + MAXBODY / hidden);
    } else {
        verPos = 0;
    }

    /* End of excerpt from FPM */

    HelpPropInfo.Flags = ACTIONED | FREEVENT;
    HelpPropInfo.VerPos = verPos;
    HelpPropInfo.VerBody = verBody;

    NewModifyProp(&HelpPropGadget, hi->window, NULL,
        (ULONG) HelpPropInfo.Flags,
        -1L, (LONG) verPos, -1L, (LONG) verBody, 1L);

    /*
     * The proportional gadget has been calculated. Now display all the
     * lines that will fit in the window.
     */
    helpDisplayVisibleLines(hi);

    for(redisplay = FALSE, quit = FALSE; (redisplay || !quit); {
        SendWait((hi->window->UserPort->Msg_SigBit));
        while (img = (struct IntMsg *) GetMsg(hi->window->UserPort)) {
            class = img->Class;
            code = img->Code;
            qualifier = img->Qualifier;
            ReplyMsg((struct Message *) img);

            /*
             * If the redisplay or quit flags are set, we reply to all
             * messages in the message queue, but ignore them. Once
             * all the messages have been replied to, the subsequent for()
             * loop will terminate.
             */
            if (redisplay || quit) continue;

            /*
             * Neither the redisplay or quit flag is set. Therefore, we need
             * to process this message.
             */
            switch (class) {
                case CLOSINGDOWN:
                    /*
                     * The user has had enough. We set the quit flag so that
                     * this loop will terminate.
                     */
                    quit = TRUE;
                    continue;

                case GADGETUP:
                    /*
                     * The user has released the proportional gadget. We
                     * probably need to redisplay the help text.
                     * We need to prevent the user from resizing the window
                     * until the text is redisplayed. We want to turn off
                     * NONRESIZE messages, though.
                     */
                    ModifyIntMsg(hi->window, HELP_EXIT | HIDEVERIFY);
                    /* Fall through */

                case NONRESIZE:
                    /*
                     * If the case is NONRESIZE (if we didn't fall through
                     * from the above case) we know that the user is still
                     * playing with the proportional gadget with the left
                     * mouse button down and dragging. Since the prop gadget
                     * is in use, we know that the HIDEVERIFY option is
                     * still in effect.
                     *
                     * Now we need to calculate the prop gadget values (see
                     * NON and possibly redisplay the text.
                     */
                    hidden = MAX(totalLines - visibleLines, 0);
                    topLine =
                        ((ULONG) hidden +
                         ((struct PropInfo *)
                          (HelpPropGadget.SpecialInfo->VerPos)) +
                         (MAXBODY/2)) / MAXBODY;

                    if (topLine != hi->top_line) {
                        hi->top_line = topLine;
                        helpDisplayVisibleLines(hi);
                    }
                    break;

                case REFRESHWINDOW:
                    /*
                     * Since we opened a SINGLE REFRESH window, we need to
                     * handle REFRESHWINDOW messages.
                     */
                    Refresh(hi->window);
                    helpDisplayVisibleLines(hi);
                    EndRefresh(hi->window, (LONG) TRUE);
                    break;

                case RESIZE:
                    /*
                     * If the size of the window hasn't actually changed, we
                     * have nothing to do.
                     */
                    if ((hi->win_height == hi->window->Height) &&
                        (hi->win_width == hi->window->Width))
                        break;

                    /*
                     * Otherwise, we need to determine the new window
                     * information. Also, since the width of the window may
                     * have changed, we need to recalculate the pixel size of
                     * a line.
                     */
                    hi->win_height = hi->window->Height;
                    hi->win_pixels_line = RIGHT_MARGIN - LEFT_MARGIN + 1;
                    hi->win_top = hi->window->YofEdges;
                    hi->win_left = hi->window->XofEdges;

                    /*
                     * If the window width changed, we cannot just redisplay
                     * the text - we need to read it all in again. We set
                     * the redisplay flag so that this loop will terminate.
                     */
                    (hi->win_width != hi->win_width) {
                        hi->win_width = hi->window->Width;
                        redisplay = TRUE;
                        continue;
                    }

                    /*
                     * Since only the height changed we can just redisplay
                     * the help text based on the new number of lines in the
                     * window. We need to calculate the prop gadget values
                     * again (see NON).
                     */
                    hi->lines_per_page = visibleLines =
                        (SHORT) (hi->rows - TOP_MARGIN) / ROW_HEIGHT;
                    hidden = MAX(totalLines - visibleLines, 0);
                    if (topLine > hidden) topLine = hidden;
                    if (hidden > 0) {
                        verBody = (ULONG)
                            (visibleLines - overlap) *
                            MAXBODY / (totalLines - overlap);
                    } else {
                        verBody = MAXBODY;
                    }
                    if (hidden > 0) {
                        verPos = (ULONG) ((ULONG) topLine + MAXBODY / hidden);
                    } else {
                        verPos = 0;
                    }

                    NewModifyProp(&HelpPropGadget, hi->window, NULL,
                        (ULONG) HelpPropInfo.Flags,
                        -1L, (LONG) verPos, -1L, (LONG) verBody, 1L);

                    helpDisplayVisibleLines(hi);
                    break;

                case GADGETDOWN:
                    /*
                     * The prop gadget has been selected. As long as the user
                     * keeps dragging it, we want to get NONRESIZE messages.
                     * We will be redisplaying the text as the mouse moves, so
                     * we don't want the user to resize the window. It would
                     * be a neat trick for the user to resize the window and
                     * anyway since the prop gadget is being used!
                     */

```



```

GetAdventer, 0);
GetDungeon, 0);
ScrollUp, 1);
    (LONG)hi->window->borderLeft,
    (LONG)hi->window->borderTop,
    (LONG)hi->window->width - hi->window->borderRight - 1),
    (LONG)hi->window->height - hi->window->borderBottom - 1));

/*
 * There may several line index tables. We need to find the
 * line index that points to the segments for the top line we are
 * to display. When the for() loop is done, hli will point to the
 * correct helpLineIndex struct.
 */
which_hli = hli = hli->top_line/MAX_LINES_PER_INDEX;
for (hli = hi->firstIndex; which_hli > 0; hli = hli->next, which_hli--)

/*
 * endIndex is one more than the number of lines we are to display
 * in this window.
 */
endIndex = MIN(hi->lines_per_page, hi->total_lines - hi->top_line);

/*
 * Here we display the lines. We may cross into other line indexes.
 */
for (count = 0, lineIndex = hi->top_line - (hli->MAX_LINES_PER_INDEX;
    count < endIndex);
    if (lineIndex == MAX_LINES_PER_INDEX) {
        lineIndex = 0;
        hli = hli->next;
        continue;
    }
    MoveUp, LEFT_MARGIN, TOP_MARGIN + ROW_HEIGHT*count);
    helpDisplayLineToChk, hli->line[lineIndex]);
    count++;
    lineIndex++;
}

```

```

void HelpDisplayLine(HI, segment)
{
    HelpTextLine *hl;
    HI->HelpSegmentStruct *segment;

    struct HelpSegmentStruct *hs;
    struct TextFont *tfp = HI->Window->TFont;

    /*
     * For every segment in this line, set the fontstyle, drawing mode and
     * pass in the segment values and print the text.
     */
    for (hs=segment; hs != hs->Next; hs=hs->Next) {
        HelpTextLine *hl=hs->TextLine;
        hl->fontstyle = hs->fontstyle;
        hl->drawmode = hs->drawmode;
        hl->fontcolor = hs->fontcolor;
        hl->fontsize = hs->fontsize;
        Text(HI, hl->Text, tfp, HI->Window->YCoord+hs->YCoord);
    }
}

```

```

static void helpDisplay(FinishLineStruct HelpInstance *H)
Description: This function will display all lines that fit in
the help window, starting with the top line
specified in HelpInstance structure.
**NOTE** The help window's TOP flag must have the
STAYOPEN flag set before this function is called!
Otherwise, it is possible that this function will
trash the window borders.
Input:    h: a pointer to the HelpInstance data structure.
Returns:   Nothing.

```

```

struct HelpLineWindowProc "HLP:
HWND hwnd, WPARAM, LPARAM, HWNDparent:
TRYIN which all, hold:
struct HelpText "cp = h1-window+0x00000000
/*
 * As a precaution, we won't render into the window if the window width
 * changed since the HelpLine structure was last updated. This
 * routine MUST be called with the HIDEWINDOW flag turned on for the
 * window's IDHWND. That will prevent the window from being realized
 * while we're in this routine.
 */
if (h1-win_width == h1-window+0x00000000) {
    /*
     * First we will clear the help text area in the window.
     */

```

```
static void __cdecl _File, int line, char *func, ULONG rc,
struct _HelpInstance *hi)

/* Description: A simple debugging function. When the CHECK()
macro is called, this function gets called with the
filename, line number and function name where
the CHECK() macro was invoked. _wassert() also is
passed a result code that is specified by the
caller of CHECK(). This function will free all
resources allocated by the help utility and then
exit. NOTE that any resources allocated prior to
calling _wassert() will not be freed. This function
should never get called. It is left in for
debugging purposes only.

/* Inputs: file: a null terminated ASCII string, representing
the filename where the bug occurred.
line: the line number of the buggy line in the
file.
func: a null terminated ASCII string, representing
the function name where the bug occurred.
rc: a value that is meaningful only in context of
the buggy line.
hi: a pointer to the HelpInstance data structure.

/* Returns: Nothing. Actually, it terminates the program.
Read the description above.
```

```
static void sw_err(FILE, line, sum, re, hi)
char *file, *Fname;
int line;
double re;
struct helpInstance *hi;
{
    printf("sw_err: %s:%d: re=%f\n", file, line, sum, re, hi);
    helpFree(hi);
    exit(1);
}
```

Programming the Amiga's GUI in C

• Part I •

by Paul Castonguay

This is the beginning of a series that will help you take advantage of many of the custom features of your Amiga using the C Programming Language. In this issue you will find:

- 1.) A description of the skills that you will need in order to program your Amiga in C.
- 2.) A detailed installation procedure (on disk) of the SAS version 5.10a compiler for Hard Drive as well as Floppy systems. The installation of 512K, 1MB, as well as 2MB "Two Floppy Systems" is optimized by special installation programs.
- 3.) Some background material (on disk) for understanding compilers and linkers.
- 4.) An overview (on disk) of the SAS/C environment.
- 5.) A detailed description (on disk) of how to use SAS/C, complete with an example program to test the validity of your installation.
- 6.) A presentation (in this article) of the first programming concept in the Intuition environment, the opening of libraries.

The example program (on disk) is called Rosettes and it demonstrates the kind of programs I intend to present throughout this series. It is not so complex that it will seem impossible to learn, yet hopefully it is interesting enough that you will want to find out how to write similar programs yourself. It should at least convince you that you can indeed design, compile, and execute C programs that interact with the custom features of your Amiga, even on a 512K two floppy system.

SAS/C Version 5.10a

This is the compiler that I will be using. If you have purchased the recent 5.10, please be advised that you are now eligible for a free update simply by calling the "Book Sales Department" of SAS at 919-677-8000, extension 5042 and giving them your user registration number. In fact, if you have a "Two Floppy System" you may have had trouble getting version 5.10 of SAS/C to work on it. There were some simple but unfortunate errors in their "read.me" file that made it difficult for an uninitiated, first time

user to successfully install the product on a floppy system. The new version 5.10a has corrected them. In addition, the installation programs and valuable background material in this issue of AC's TECH (on disk) will just about guarantee that you not only get SAS/C up and running, but also configure it in a way that takes maximum advantage of your system. There are in fact three separate installations for three different sizes of "Two Floppy Systems," 512K, 1MB, and 2MB.

WHY C?

C is the most natural high level language to program the Amiga with. Many of the Amiga's system functions expect to be called in a form that is easily implemented in C. But as an average hobbyist, you may be feeling a bit overwhelmed by the apparent complexity of such programs written for the Amiga. Even if you have already taken a C programming course, you may be unsure of how to apply that knowledge. Just the simple act of opening a graphics window may look at first like a formidable programming challenge.

Successful programming in C is largely a matter of organizing your work so that it is useful in the greatest number of situations. A program fragment that performs a particular task, like opening a window, should be written in such a way that it becomes useful to any programming projects that need one. In this series I will emphasize techniques to help you do that. Naturally you will need some prerequisites and I will list them in the next section. But even without them you should have no trouble following the detailed instructions on disk and getting your compiler up and running, even compiling the Rosettes program.

PREREQUISITE #1, LEARN TO USE YOUR COMPUTER

You need to know how to do such things as make backup copies of your system diskettes, run programs, initialize (format) diskettes, produce documents using either a word processor or editor, organize your work into drawers (directories), ... etc. These operations and many more are generally referred to as "using your computer at its system or command level." You cannot seriously sit down to program a computer in any language unless you know how to do that.



Getting starting in programming the Amiga in C... Including a presentation of the first programming concept in the Intuition environment, the opening of libraries.

The Amiga has two modes of operation at the system level, the interactive icon/pull-down menu mode, available from the Workbench tool (application), and the command line mode called AmigaDOS available from the SHELL window. You should develop the skill of using both. You may accomplish this quickly if you have had previous experience on other systems, or not so quickly if the Amiga is your first computer. Whatever your pace, don't rush it. Learn to use your computer comfortably.

I'd like to also add that although the Workbench tool seems to need no explanation, don't underestimate its powers. Read the manual. For example many users complain that it is too clumsy when duplicating a series of diskettes on a two-drive system because it requires a disk-swap each time to pick up the DiskCopy program from the boot-up diskette. Not so. The DiskCopy program can be operated from the RAM Disk using icons. Here is the procedure:

- 1.) Open the Workbench/System window and drag the DiskCopy icon to RAM:
- 2.) Open the RAM: window so that the DiskCopy icon is visible.
- 3.) Place the diskette you wish to back up in any drive and a blank one (or otherwise not needed) in the other.
- 4.) Left-select the disk you want to back up.
- 5.) Press the SHIFT key and hold it down throughout the remainder of this procedure.
- 6.) Left-select the destination disk.
- 7.) Double-click the RAM:DiskCopy icon. You may now lift the SHIFT key. The DiskCopy procedure continues, presenting you with the usual dialogue box and without any swaps.

Thus you may conveniently back up a series of diskettes, like the ones that came with your SAS/C Compiler. Note that this procedure changes their names from "SAS_C_5.1.X" to "Copy of SAS_C_5.1.X," a feature that is very desirable when backing up your own work. How often have you forgotten which disk was the original and which was the backup? The above feature saves you the trouble of looking inside each disk and comparing date stamps on the various files to find that out. But if you want to use the above disks to install SAS/C on your system you will have to change their names back to "SAS_C_5.1.X."

For those who want to learn AmigaDOS I recommend the latest editions of either "COMPUTE's AmigaDOS Reference Guide," by Arian R. Levitan and Sheldon Leemon, Compute! Publications, Inc., 1986, or "The Amiga Companion," by Rob Peck, IDGC/Peterborough Publications, 1988. In addition to learning all the usual commands for copying files, creating directories, and organizing your work, you need to understand the assignment of logical devices. Make sure you know what is meant by a system logical device, like C:, S:, LIBS:, FONTS:, and SYS:, as well as user defined logical devices. For example:

Assign BK: My_Work:My_C_Progs/Window_Examples

This command assigns the device name BK: to a directory where you would like to save your work. The device name BK: is a convenient way of specifying that location. Suppose, for instance, your current working directory is a drawer on the RAM Disk, as is usually the case in SAS/C for smoothest operation. You can backup your work with the simple command:

Copy #1 to BK:

The system will automatically ask you to insert the "My_Work" disk, and then copy all files from your current directory to the above specified location.

The assigning of logical devices cannot be done from the Amiga's Workbench tool, although you can design the feature in by using the IconX program in conjunction with a script file and an icon of your own design.



A C ' s T E C H UPDATE

o n d i s k i n u p d a t e . l z h

FastBoot DBI Idiosyncrasies (AC's TECH 1.1)

Some readers have been experiencing some problems with the DBI program included for use with the FastBoot story in AC's TECH V1.1. We have included a fresher copy of the DBI program on the AC's TECH 1.2 disk (Included in Updates.lzh). We hope this solves any problems you may have encountered. Thanks to the readers who brought this to our attention. An additional thanks goes to Dan Babcock for his help. Also, please note that DBI is shareware. If you find DBI to be useful, please send the nominal \$5.00 shareware fee to the author, Jorgen Kjæsgaard. His address is listed in dbi.doc.

VidCell Digitizer Questions (AC's TECH 1.1)

Oops! We incorrectly listed the Digi-Key part number for the high-speed 555 timer in the VidCell digitizer story. The correct part number is LM555CN. If you have a hard time locating that part, use ICM7555IPA, which is also available from Digi-Key.

Additionally, some readers have reported some difficulties in locating the KAD0820BCN A/D converter. Digi-Key reportedly does not stock that part any longer. Another verified source is Allied Electronics. Their order number is (800) 433-5700. Many thanks to Doug Austin for this information. Jameco sells a verified equivalent A/D converter, ADC0820CCN, for \$12.00. Jameco's number is (415) 592-8097. Please note that Jameco enforces a \$50.00 minimum order. In any case, if you prefer you can purchase the A/D converter directly from GT Devices for \$12.00.

There were also minor errors in the schematic drawing on page 28. The polarity symbols (+ -) at C4 should have been reversed. If you follow the schematic symbol, you'll be fine. Thanks to Alastair Bor for pointing out this problem.

Finally, due to popular demand the VidCell PCB only is available from GT Devices, for \$20.00. For more information on VidCell, or to order, write to GT Devices, P. O. Box 2098, Pasco, WA, 99301.

Where's SuperView? (AC's TECH 1.1)

Some readers have commented on the empty superview.docx directory which was accidentally left on the AC's TECH 1.1 master. SuperView was never meant to appear on the AC's TECH disk, so you didn't miss a thing!

Missing ARExx? (AC's TECH 1.1)

The lharc file for 'An Introduction to Interprocess Communication with ARExx' on the AC's TECH Disk 1.1, was accidentally omitted from a master diskette. We have included the file as IPC.lzh. We apologize for the error.

Please let us know if you make any major upgrades/modifications (for the better!) to any of the software/hardware projects that appear in AC's TECH. We will send a copy of your changes to the author, and maybe even distribute it via AC's TECH Updates.

Send your comments, suggestions, and program updates to:

PiM Publications, Inc.
AC's TECH UPDATES
P.O. Box 869
Fall River, MA 02720-0869

Presenting the AC's TECH Disk—Volume 1, Number 2

A few notes before you dive into the disk!

- You'll need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.
- In order to fit as much information as possible on the AC's TECH Disk, we archived most of the files using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

To unarchive a file *foo.lzh*, type *lharc x foo*
For help with lharc, type *lharc ?*



**HAPPY
PROGRAMMING!**

ENJOY!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PIM Publications, Inc., their distributors, and/or its retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas.)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are Copyright ©1990, 1991 by PIM Publications, Inc. and may not be duplicated in any way. The purchaser, however, is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. Neither PIM Publications, nor its agents will be held responsible for any damages incurred while attempting these projects.

PREREQUISITE #2, LEARN BASIC

That's right, I recommend that before learning to program your Amiga in C, you do so in BASIC. I give this advice to both novice and experienced programmers alike, although to each for different reasons.

Beginning programmers need to learn the fundamental concepts of programming, the standard building blocks (constructs) around which all programs are built, regardless of language. BASIC allows you to do that within an environment that is easy to use. I also recommend True BASIC because it is a compiler, and because it supports many of the concepts that are usually attributed to C, like local variables, external functions, library files (similar to header files and pre-compiled modules in C), and recursion. It also allows access (through the Toolkit) to the same system functions that we will be using here in C. And most important for beginners, True BASIC is a well-established teaching language with excellent textbook support. It will help ease your graduation to C.

Experienced programmers, even experienced C programmers, need to become familiar with the machine they are programming, and most importantly its display. You can't create much of a program if you can't show anything on the screen. AmigaBASIC is an easy-to-use environment that supports the underlying concepts of screens and windows. Play around with the SCREEN and WINDOW statements. Find out about bit planes and how they are used to produce 32 colors. Try a few simple graphic images. You don't have to devote a lot of time and energy to this, just enough to make yourself feel comfortable with how the system works. An excellent text is "Advanced AmigaBASIC," by Tom Halfhill & Charles Brannon, COMPUTE! Publications, Inc., 1986. Often, when you are about to try a feature on the Amiga for the first time in C, it is informative to see first how that same thing is implemented in AmigaBASIC.

PREREQUISITE #3: LEARN STANDARD C

Standard C is the language that you learn in a C programming course at any community college or university. (Anyone questioning my use of the term "Standard C" should refer to the editorial in the January, 1991 issue of "The C Users Journal," R&D Publications, Inc. 2601 Iowa, Lawrence, KS 66046, (913) 841-1631). The syntax of Standard C is in accordance with the now famous book "The C Programming Language, Second Edition," by Brian Kernighan and Dennis Ritchie (the creators of the C Language), Prentice Hall 1988, which I refer to throughout this series as K&R. However, if you've never studied C before and if you're trying to do so on your own, you may prefer an easier text. A good one is "C Primer Plus," by Mitchell Waite, Stephen Prata, and Donald Martin, Howard W. Sams & Co., 1985. Buy the latest edition.

Note that Standard C doesn't interact with the Amiga's system of graphic windows, menus, sprites, etc. Rather it is limited to treating the computer as if it were a plain, vanilla-type terminal (DEC VT-100 or equivalent), the kind that you see many

of when you visit the computing center at any university. Screen I/O (input/output) usually consists of standard format, text strings within a scrolling type window. On the Amiga such programs use a SHELL window. It is in fact an important feature of the Amiga that it can run any Standard C program, with perhaps a few trivial exceptions, from a SHELL window. That means you can take any course in C and execute all your assignments right on your Amiga. It also means you can use your Amiga to learn all about C programming on your own, at home. Naturally, to do this you will have to install your compiler and learn how to use it. As I mentioned earlier, you can find detailed instructions on how to do that, along with some valuable background material, on the disk that came with this magazine.

THE AMIGA'S GUI, INTUITION

In case you haven't already guessed there are really two ways to program the Amiga in C. The first, mentioned above, interacts with the system from an ordinary SHELL window. To make such programs accept data you use functions like:

```
scanf("%s", unsigned char *s);
getchar();
```

To make them produce output you use functions like:

```
printf("%s", unsigned char *s);
putchar(int c);
```

Note that it is my style to occasionally show the data types of certain arguments in a function call by using the same form as the new ANSI prototype declarations. I do this whenever I think it will help in their understanding. For example, the above "unsigned char *s" shows you that "s" is a pointer to type "unsigned char". In a real program you would not use that notation for a function call. Rather, you would first declare s, then use it as an argument, alone like this:

```
unsigned char *s = "Hello out there!!\n";
printf("%s", s);
```

The second and more exciting way to program the Amiga is to interact with it through an environment called a GRAPHICAL USER INTERFACE (GUI) that supports the use of windows and pull-down menus. The Amiga's GUI is called "Intuition."

I want to mention at this point that there are other computers besides the Amiga that have "Graphical User Interfaces." The UNIX operating system has XWindows, the Macintosh has the Toolbox, and IBM has Microsoft Windows Version 3.0 (and a few others such as HP's New Wave). The irony is that while those systems can cost you a lot more, Intuition on the Amiga completely outperforms them on similar hardware. And, the Amiga's windowing system multitasks without four megabytes of RAM and an 80386 microprocessor.

Programming in the Intuition environment, or any other GUI for that matter, is very different from programming in Standard C. GUI's require that you become familiar with a large

number of functions, each of which may require knowledge of a multitude of details before you can use them. For example, to send text to a graphics window requires (but don't try to understand it yet) the function:

```
Test(struct RastPort *rp, unsigned char *s, strlen(char *s))
```

In addition, Intuition's functions are specific to the Amiga. Programs written in it will not execute on a UNIX or MS-DOS machine and visa versa. Still the general operation of all GUI's is similar enough so that experience in one is an advantage when trying to learn another.

The standard reference for the Amiga is the "ROM KERNEL REFERENCE MANUAL: LIBRARIES & DEVICES," Addison Wesley 1990. But any hobbyist who has seen that book knows how overwhelming it can be. Does that mean you have to be a trained computerscientist with XWindows experience to program the Amiga in C? Hey, the Amiga is supposed to be a hobbyist's machine, remember? Surely there must be some way the average user can access its great potential as well. There is, and it is the purpose of this series of articles to present exactly how that is done. Once you get up and running with a few of the basic concepts, you will find yourself much more able to deal with that ROM Kernel Manual.

THE AMIGA'S HEADER FILE SYSTEM

Remember from Standard C how you were required to reference certain files that contained definitions particular to the system you were using? You did that by putting at the top of your source code one or more #include statements (actually called preprocessor directives) that made reference to one of these files, traditionally called header files (K&R, pages 33, 88). For example:

```
#include<stdio.h>
#include<string.h>
#include<math.h>
```

You probably learned from experience that there were times when one of your programs didn't work because it was trying to perform an operation requiring one of these (K&R, 241). And perhaps at the time you didn't care what the particular problem really was, or what exactly was in those files. You just wanted your program to work, so you blindly added #include statements at the top of your source code until it did. Such an approach won't work on the Amiga.

The Amiga's system of header files contains a formidable number of declarations and definitions that you must be aware of, and in some cases be intimately familiar with, in order to take advantage of its custom features. In SAS/C there are actually two sets of such files. The first, called "Compiler Headers," is located in the directory SAS_C_5.1.4:Compiler_Headers, in dh0:LC/Compiler_Headers on a hard drive system. These files are for reference. They are for you to inspect and study whenever you need to know a particular detail about some feature of the Amiga.

AAMIGA WAREHOUSE

The Memory Specialist

FOR SOFTWARE GO TO THE REST. FOR HARDWARE CALL THE BEST!!!

*A Power User's Lunch
Memory, Accelerators, Flicker Fixer's.
I almost forgot dessert, Hard Drives.
You can get your fill without emptying your wallet!*

*Do you do any Desktop Publishing or Ray Tracing?
Using Amax? Then you know how annoying Interlace
flicker can be. We have packages that can save your eyes.*

*Call for current
pricing on
Memory chips.
We will not be
beat on any
product
we have in stock.*

**Special
Flicker Fixer and VGA
Monitor
\$575**

**Hard Cards
Start as low as
\$353.50 for a
40 meg system**

We carry a full line of Monitors.

**Limited Time
CSA's Mega Midget with CPU and Co Processor for only \$699.50**

**GOLDENIMAGE
ICD
Expansion Systems**

15488 Feldspar Dr.
Chino Hill, CA. 91709
1-800-942-9505
714-283-0499
Major Credit Cards
accepted or Cod.

1-800-942-9505

Circle 151 on Reader Service card.

The second set is called "Compacted Header Files." These are the exact equivalent of the others except that they have been compressed in size for faster access. They were produced from the first when you installed SAS/C on your system and they are located in the INCLUDE: device. These "Compacted Header Files" are for your compiler to read. They are referenced when your program is compiled.

To get a printout of the names of all the header files that come with SAS/C, enter the following command from a SHELL window:

```
DLF >PRT: INCLUDE: OPT A
```

The Amiga's header files fall into two main categories. Those in the first, which I call "Standard Headers," are the same as those that you will find on any implementation of Standard C. They have the familiar names: <stdio.h>, <string.h>, <math.h>, etc. (K&R, 241) The second category of header files is what I call "Amiga Specific Headers." These contain a proliferation of definitions and details that have to do with the Amiga's custom features.

A large part of the secret to programming the Amiga is in becoming familiar with what is in these files. I will slowly begin to do that in these articles. Notice that many of them are divided



into directories with names like: exec, intuition, and graphics. Each contain files that are used to access a particular part of the Amiga's operating system. The particular header files that allow the use of the Amiga's GUI are in the directory called "intuition". But before getting into that, we need to consider some fundamental concepts.

AMIGA'S DATA TYPES

Your first job in programming the Amiga is to become familiar with its data types. You know that in Standard C all variables must have their type declared:

```
int i;  
unsigned char *str;  
float x;
```

But perhaps you have seen listings for the Amiga in various magazines that referred to following data types:

```
WORD w;  
STRPTR str;  
FLOAT x;
```

These are simply new names for some of the usual data types of Standard C. They are in fact the exact equivalents of the first three that I just mentioned.

These redefinitions are made for you by a series of "typedef" statements (K&R, 146) in the header file <exec/types.h>. So to become familiar with the Amiga's data types you must look at that file. You can do that by using your editor. Enter into a SHELL window:

```
USE BAS_C_5.1.4:Compiler-Headers/exec/types.h
```

You will also find there some macro definitions (K&R,88) for the storage classes extern (K&R, 31, 33, 80) and register (K&R,83). You are not forced to use these definitions. After all, the compiler itself must first perform the proper substitutions and convert them all back to Standard C before compiling your program. However, there are certain advantages to using them. They make your programs more understandable. Here is a short example:

```
#include<exec/types.h>  
main()  
{  
    STRPTR s_one, s_two;  
    s_one = "Hello Out there!!!\n";  
    s_two = "Help me I'm stuck in here.\n";  
    printf("kaka", s_one, s_two);  
}
```

Note that STRPTR is defined in <exec/types.h> as "unsigned char *". In the above example the name STRPTR is more informative than "unsigned char *" because it identifies not only that s_one and s_two are pointers of type char (which can point to any location in memory), but also that your intention is to use them specifically to point to strings. Strings are contiguous sections of memory containing text and terminated by a "null character" (K&R, 30).

All this may seem like a lot of talk over a minor point, but it isn't. You see, the Amiga is so complex a machine that knowing the exact purpose of a variable is often crucial to understanding how a program works. In fact, here lies another large part of the secret to programming the Amiga in C. You must write your programs in such a form that they not only work, but also that they are easily understandable to yourself and others. Thus <exec/types.h> is an important inclusion for all your programs. But later we will see that its inclusion is automatic when we include a certain other header file.

THE AMIGA'S RUN TIME LIBRARIES

In most programming environments your compiled programs are linked to whatever system routines are needed to run them. The Amiga does that with its "Startup Code" and "Standard Math Library," and in that way it is no different than any other system. But the Amiga also allows your programs to use other libraries that are not linked to them. Indeed, these libraries are loaded onto the system independently of your programs and are called "Run Time Libraries." The idea is that the system needs to put only one copy of such a library in memory and it will serve the needs of several programs running simultaneously. You can see immediately that the Amiga was designed as a multitasking machine from the ground up. In fact, the Amiga has several of these "Run Time Libraries," each of which is loaded into memory only if a program running on the system requests it. We will be using several "Run Time Libraries" in this series.

The particular "Run Time Library" that is related to the Amiga's GUI is called "intuition.library". Thus the first operation that any of your programs must do (to use the GUI) is send a request to the operating system asking that it load "intuition.library" into memory (open it). This is done by calling the system function "OpenLibrary()."

THE EXEC IS ALWAYS THERE

The "exec.library" is another "Run Time Library," but unlike "intuition.library" this one is always loaded on the system. It happens automatically at boot up time. The "exec.library" contains many low level, system functions for the Amiga, but we need to know only a few of them. It turns out that a lot of the use of the various functions in "exec.library" will be hidden from us by the fact that they are often called directly by functions in other, higher level libraries. Thus we will be protected from many of the computer's lower level details. This is in accordance with a formal principle in computer science called "Information Hiding," which dictates that different programs, indeed even different parts of programs, should not have any unnecessary knowledge of each other. This principle reduces the complexity of our own programs by lowering the number of details that we must be aware of in order to design them. I will refer often to this principle in these articles.

OPENING A LIBRARY

Here is what we must do within our programs to request "intuition.library" from the operating system:

- 1.) Declare a pointer (K&R,93) of the correct type. The "intuition.library" has its own special type of pointer that you must use.
- 2.) Call the `OpenLibrary()` function using as arguments the name of the library and a version number. The function will return to us the address in memory where it successfully loaded the library, or a NULL if it failed.
- 3.) Test that the library was successfully loaded into memory.
- 4.) Assign the address of the library, as returned by the `OpenLibrary()` function, to the previously declared library pointer.

THE LIBRARY POINTER

There are two design considerations that affect how you should declare the library pointer. First it must be available to all sections of your program that need to use functions in the library. Second, reference to it is handled automatically by the system. That is, you yourself don't need to use the library's address as an argument in any of your function calls. You simply call a function by name and the system finds it for you. To be able to do all this requires that this pointer be of global scope (K&R,80) and of a particular type and name. In this case the type must be "pointer to struct `IntuitionBase`" and the name must be "`IntuitionBase`." And be careful, these names are case sensitive. Remember from Standard C that a structure is a data type that can contain items (called members) of different types (K&R, 127).

So, to use the `OpenLibrary()` function your program requires the following global declaration:

```
struct IntuitionBase *IntuitionBase;
```

Well now, if that doesn't sound like a lot of double talk, I don't know what does. Actually it's very simple. First of all, the type "struct `IntuitionBase`" is data type that is declared for you within the header file `<intuition/intuitionbase.h>`. You can look there and inspect its details if you like, but it isn't necessary that you know much about it. You will not be called upon to make any references or assignments to any of its members. The system will do whatever is necessary for you, automatically. However you should realize that this structure does not come magically out of the clouds, but is in fact declared in the header file `<intuition/intuitionbase.h>`, which somehow must get included with your program if you ever want to use "intuition.library."

VERSION NUMBER

The version number represents the release of the operating system you are using. There is a macro definition (K&R,89) in `<exec/types.h>` called `LIBRARY_VERSION` which gives you the version number of the operating system that your compiler is

presently supporting. The actual version of the operating system you are using is available either from a Workbench menu selection, or by entering into a SHELL window the following AmigaDOS command:

```
version
```

My system reports:

```
Kickstart version 34.5. Workbench version 34.20
```

Version numbers are there for your protection, but it is also your responsibility to verify that you are using the correct one. If you use a version of the compiler that is higher than that of the system, you could get into trouble by accidentally using something that is not yet supported on your computer. On the other hand, a compiler that is lower than the operating system may not be able to take full advantage of everything available on the computer. You should look into `<exec/types.h>` and compare your compiler's version number against the one reported by the above "version" command on your system.

The `OpenLibrary()` function protects you against any dangerous incompatibilities between different versions of the operating system by refusing to open a library if you give it a version number that is higher than the one on the system where the program is running. In a minute we will write a program to investigate this, but first I must talk about how the SAS/C compiler links programs that use the Amiga's custom features.

PROTO HEADER FILES

In older versions of the compiler (from Lattice) when your programs made use of any operating system functions, it was necessary to link them to a certain library file called "amiga.lib" (page G10 of your SAS documentation). But recently it has been possible to bypass that requirement by using within your programs certain include statements for what are called "Prototype Header Files." There are some 19 of these files and they serve two purposes. First, they declare all the Amiga's system functions according to the new ANSI standard (K&R, 26). Second, they allow the system to call these functions more directly than was possible before, without the need to link to "amiga.lib". SAS claims that including these "Proto Header Files" improves the execution speed of your programs while also reducing the time required to link them. I also noticed that it takes less memory to link programs that include proto files, an important issue on a 512K machine. I will therefore use the "Prototype Header Files" in all my example programs in these articles.

Note that the floppy installation procedures (on disk) do NOT install the file "amiga.lib" to the SAS_Headers diskette. I do not recommend it (and neither does SAS) because doing so will make the SAS_Headers diskette 100% full, as opposed to 89%. As a programmer you will want some of that room for your own use. As a result, if you try to compile and link programs that use any

→

of the Amiga's "Run Time Libraries" you should have in your code an inclusion to the "Prototype Header Files". Without it the Linker will report:

```
Error 425: Cannot find library LibAmiga.lib
```

LAZY MAN'S PROTO INCLUSION

If you are trying to compile someone else's code, and are not sure which system "Run Time Libraries" the program is using, you can include the single file `<proto/all.h>`. This will cause the inclusion of all prototype declarations for all the Amiga's "Run Time Libraries". The program will then link without "amiga.lib" no matter what.

TESTING THE OPENLIBRARY() FUNCTION

Ok, now it's time to try out our first function call to the Amiga's operating system. But let's take a conservative approach. Here is an example which simply asks the system to load the intuition library and then report its address. It also reports the version number of the operating system that your compiler is supporting. The example will give you a chance to experiment with the `OpenLibrary()` function.

```
#include<exec/types.h>
#include<intuition/intuitionbase.h>
#include<proto/exec.h>
#include<proto/intuition.h>

struct IntuitionBase *IntuitionBase;

main( )
{
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", LIBRARY_VERSION);
    printf("The LIBRARY_VERSION is %d\n", LIBRARY_VERSION);
    printf("The address of intuition.library is %u\n", IntuitionBase);
}
```

That above program on my system reports:

```
The LIBRARY_VERSION is 34
The address of intuition.library is 12596564
```

Note that this program should be run from a SHELL window. If you run it from an icon it executes and closes its window faster than you can read it.

Now, here is some homework for you. I want you to try the following experiments.

- 1.) Change the first letter in the library name to uppercase letters, like this: "Intuition.Library".
- 2.) Change the version argument from `LIBRARY_VERSION` to the number 0.
- 3.) Change the version argument to the number 50.
- 4.) Remove the cast (K&R, 45) from the function call. That is, change it to the following:

```
IntuitionBase = OpenLibrary("intuition.library", LIBRARY_VERSION);
```

Compile, link, and run the program each time. When the function call fails you will get something like this:

```
The LIBRARY_VERSION is 34
The address of intuition.library is 0
```

Make sure you understand what makes the function fail and what allows it to succeed. The name of the library is case sensitive. "Intuition.Library" causes the function to fail. Notice also that it fails whenever you specify a version number higher than the operating system you are using. Finally, notice how the compiler gives you a warning whenever you leave out the cast statement from in front of the function call.

Were you brave enough to ignore the warning and try running the program anyway? It runs normally. So why the warning? The answer lies in the prototype declaration of the `OpenLibrary()` function in the `<proto/exec.h>` header file. The function is declared as type "pointer to struct Library" and we are assigning it to a pointer of type "pointer to struct IntuitionBase". The "intuition.library" has its own personal data type that we must point to. The compiler notices that these data types are different and alerts us to the possibility of an incompatibility. Basically the compiler is saying, "Hey buddy, do you know what you're doing?"

It turns out that we do know what we are doing, and we use the cast statement to let the compiler know. Basically we are saying "Yes buddy, I do know what I'm doing, so stop sending this particular message." But of course you do not want the compiler to suppress all such error messages completely, only in this instance. You still want it to report other occurrences of this same kind of error if they occur elsewhere in your program. Perhaps somewhere else in your program you really did make the mistake of assigning an address to an incorrect pointer type, and if so you certainly will want the compiler to save you from any possible disasters.

It is a common theme throughout your work on the Amiga that a cast statement is used whenever you must assign a value returned by a function to a variable whose type is different from that of the function. And of course, you really must know what you are doing.

VERSION NUMBER PROTECTION

If you compile and run a program on your own computer it is easy to make sure you have the latest versions of both the operating system and compiler. In my case both are 34. But what happens when you give a copy of a program created on your machine to a friend, who, for one reason or another, is still using AmigaDOS version 1.2. Quite simply, your program will not work. The `OpenLibrary()` function will detect the incompatibility and return a NULL address. But be careful! If at that point your program was to try to use a function within that library the machine would then crash.

It is your responsibility as a programmer to design your programs such that they do not use functions in any library that for one reason or another cannot be successfully opened by the operating system (otherwise your friends will have you sent to remedial programming school for crashing their machines). The solution is simple and from your experiments with the above example you have probably already guessed it. Your program must test the value returned by the `OpenLibrary()` function. If it is zero (NULL), your program should exit safely, otherwise it can assume that the library is present and go ahead with normal execution.

Pictures (flow charts) often help solidify your understanding of programming operations. In that spirit I offer the following personal depiction of the opening a "Run Time Library" (see figure one.). Here is our previous example modified to do that:

```
#include<exec/types.h>
#include<intuition/intuitionbase.h>
#include<proto/exec.h>
#include<proto/intuition.h>
#define INTUITION_REV 33

struct IntuitionBase *IntuitionBase;

main()
{
    IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", INTUITION_REV);
    printf("The LIBRARY VERSION is %d\n", LIBRARY_VERSION);
    printf("The address of intuition.library is %u\n", IntuitionBase);

    if(IntuitionBase == NULL)
    {
        printf("The intuition library could not be found.\n");
        printf("Your program must exit safely.\n");
        exit(1);
    }

    printf("You got the library.\n");
    printf("You may continue normal program execution.\n");
    return(0);
}
```

The above program protects the code from crashing on older Amigas. To do this you must specify the version number used in the open library function call. Commodore recommends version 33. (See RKM page 231.) I will do this from now on.

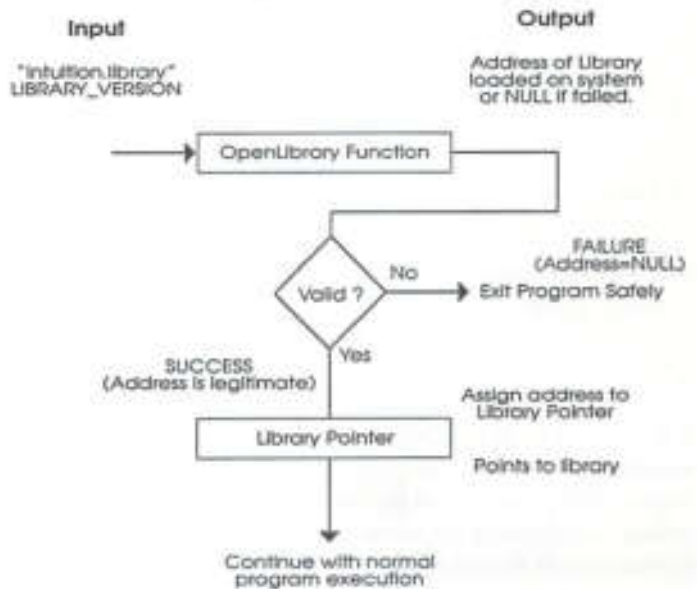
CLOSING LIBRARIES

A constant theme on the Amiga is that resources are loaded into system memory only when they are needed and later purged if they are not. Thus, it is your responsibility as a programmer to report to the operating system whenever one of your programs no longer needs a resource. For a "Run Time Library" you do that by calling the `CloseLibrary()` function. It takes one argument, the same pointer that you used to open the library with in the first place. In the present situation this calls for the following:

```
CloseLibrary((struct Library *)IntuitionBase);
```

A cast statement is needed in front of the argument in the above call because the data type of `IntuitionBase` differs from the one given in the prototype declaration in `<proto/exec>` for the `CloseLibrary()` function.

Figure One
Run Time Library



You may notice that even though you inform the operating system that you no longer need a library, it may not actually get purged immediately. The operating system has its own list of priorities to take care of and it may decide to hang on to it for a while. But regardless of what the system decides to do, the point is that unless you report when your programs are finished with certain resources the system will never be able to clear them up later when it needs to do so.

STRUCTURED PROGRAMMING

Perhaps you are now realizing that the simple act of opening a library has a lot of details surrounding it. Earlier I mentioned that a lot of C programming is based on properly organizing your work such that it becomes reusable in many different programming projects. You don't want to spend a lot of time writing the intricate details of opening libraries every time you start a new project. You want this operation to be streamlined, requiring no more than a very few lines of code. You want it to be both easy to use and easy to remember. Take a look at the following modification to our little example:

```
#include<exec/types.h>
#include<intuition/intuitionbase.h>
#include<proto/exec.h>
#include<proto/intuition.h>
#define INTUITION_REV 33

struct IntuitionBase *IntuitionBase = NULL;
BOOL Open_Libs(VOID);
VOID Close_Libs(VOID);

main()
{
    if(!Open_Libs())
        exit(1);
}
```

```

printf("The address of intuition.library is %u\n", IntuitionBase);
Close_Libs();
}

BOOL Open_Libs(VOID)
{
    if(!IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", INTUITION_REV 33))
    {
        printf("intuition.library could not be loaded.\n");
        return(FALSE);
    }
    return(TRUE);
}

VOID Close_Libs(VOID)
{
    if(IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);
}

```

Here we see quite a few changes. First let me mention that it is traditional in C to consider the number 0 as representing something that is false, and 1 something that is true (K&R, 42). Note that the header file <exec/types.h> continues in that same tradition by defining the macros TRUE and FALSE as 1 and 0 respectively. In the same spirit it is popular to design functions within programs to return a 1 if they complete their task successfully, or a 0 if for any reason they fail to do so. Such functions can then be called by using the convenient and popular style:

```

if(!My_Function())
{
    /* take corrective action */
}
/* continue normal execution */

```

If the value returned by My_Function() is equal to zero (meaning that something went wrong) the '!' (K&R, 42) operator inverts it, causing the if statement to succeed and execute the block within braces to take corrective action. If the value returned by My_Function() is equal to one (meaning that all went well) the if statement fails and normal program execution continues. Traditionally you design My_Function() to return either a TRUE or a FALSE. In the above program example the function Open_Libs() uses that same traditional calling style, exiting the program if Open_Libs() returns a FALSE.

Note that it would be incorrect if, in the event that "intuition.library" failed to open, the program terminated directly from within the Open_Libs() function. The Open_Libs() function may have no idea about the working details of the program that called it. That calling program may in fact have many details that require attention before it may be safely terminated. Do you see what I mean about making your programs usable in most programming situations? The Open_Libs() function is simply not in a position to make such decisions. The only thing it should do is report to its calling program whether or not it succeeded in performing its assigned task.

Notice also that this same calling style is used within the Open_Libs() function to call the OpenLibrary() function itself.

I declared the Open_Libs() function to be type BOOL, a data type borrowed from the PASCAL language which can have only two possible values, TRUE or FALSE (1 or 0). BOOL is another definition conveniently made for you in <exec/types.h>. Although in C that declaration doesn't theoretically prevent the Open_Libs() function from returning integers of value other than 0 or 1, we design Open_Libs() so that those are the only ones it can return. All this helps streamline the future use of the Open_Libs() function. It's a big help if all your functions follow the same conventions.

In my example I added a separate function for closing the library. The if statement verifies whether a the library pointer does indeed contain a valid address, rather than a NULL. To neglect this test would crash the machine if your program accidentally tried to close a library that was never opened successfully in the first place. Yes, I know that in my simple example this is impossible. If the library fails to open, the program terminates before it ever gets to the CloseLibrary() function. But as you will soon see, your programs will be considerably more complex than that. They will have to deal with having to close libraries at different times in their execution, when perhaps a different number of libraries are open. The above idea of testing the pointer to a library before closing it will help us design a streamlined method for dealing with such complexities.

MY FINAL VERSION

So our little test example has now graduated to quite a little program. Would you believe we are not actually finished with it? Did you notice that I named it Libs, not Lib? My intention is to design it so that it opens other libraries as well. You see, I recognize that almost all my programming examples will need more than one library and therefore I prefer to combine the opening of all of them into one convenient, easy-to-use function. My approach may be a bit wasteful for those few times when one of my program does not actually use every library that it opens, but I don't really care about that. This series of articles is for instructional purposes, where such conveniences are of greater value than scrimping for every last byte of available system memory. Besides, these libraries are not very large. Even a 512K machine can accept this kind of overhead without much complaint.

Below I present to you my final Open_Libs() function that will serve us for quite some time.

```

#include<intuition/intuitionbase.h>
#include<proto/exec.h>
#include<proto/intuition.h>
#define INTUITION_REV 33
#define GRAPHICS_REV 33
#define DISKFONT_REV 33

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Library *DiskfontBase = NULL;

BOOL Open_Libs(VOID);
VOID Close_Libs(VOID);

```



```

main()
{
    if(!Open_Libs())
        exit(1);

    printf("The address of intuition.library is %u\n", IntuitionBase);
    printf("The address of graphics.library is %u\n", GfxBase);
    printf("The address of diskfont.library is %u\n", DiskfontBase);

    Close_Libs();
}

BOOL Open_Libs(VOID)
{
    if(!IntuitionBase = (struct IntuitionBase *)
        OpenLibrary("intuition.library", INTUITION_REV))
    {
        printf("intuition.library could not be loaded.\n");
        return(FALSE);
    }

    if(!GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library", GRAPHICS_REV))
    {
        printf("graphics.library could not be loaded.\n");
        return(FALSE);
    }

    if(!DiskfontBase = (struct Library *)
        OpenLibrary("diskfont.library", DISKFONT_REV))
    {
        printf("diskfont.library could not be loaded.\n");
        return(FALSE);
    }

    return(TRUE);
}

VOID Close_Libs(VOID)
{
    if(IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);

    if(GfxBase)
        CloseLibrary((struct Library *)GfxBase);

    if(DiskfontBase)
        CloseLibrary(DiskfontBase);
}

```

Hey! What happened to the #include<exec/types.h>? Take a look in Compiler_Headers (disk4) at intuition/intuitionbase.h. You can use the LSE Editor for this. Enter into a SHELL window:

```
LSE SAS_C_5.1.4:Compiler_Headers/intuition/intuitionbase.h
```

At the very beginning of the file are three lines which test if <exec/types.h> has been previously included, and if not to do so. Well how about that! The header file system is smart enough to know what other files are needed whenever you want to use a particular one. So my previous inclusion of <exec/types.h> wasn't really necessary after all, <intuition/intuitionbase.h> takes care of it for me automatically. I admit now that I included it earlier only to draw your attention to its importance.

AN IMPORTANT AMIGADOS FEATURE

I would like to close this issue with something that has nothing to do with the C language, but which is very important when using a development system like SAS. You are now beginning to realize the importance of header files, right? Well, suppose you need to know what particular header file contains a certain definition. Perhaps you do not even know for sure the exact

spelling of what you are looking for. That problem happens often when programming the Amiga in C.

OK, let's suppose you want to know which header file contains the version number supported by the compiler. How do you find out? Enter into a SHELL window:

```
Search SAS_C_5.1.4:Compiler_Headers VERSION quick all
```

AmigaDOS begins to search all files, reporting their names and any line numbers where the string "version" occurs within them. Eventually you will see the macro LIBRARY_VERSION defined on line 56 of <exec/types.h> as 34. You can press [CTRL]-C to stop the search at any time. Would you like a printout? Enter:

```
Search >PRT; SAS_C_5.1.4:Compiler_Headers VERSION all
```

Note that none of this will work if you have installed the compiler to a floppy system using the SAS installation procedure, but it will work perfectly if you install it using one of the programs that come on disk with this magazine. If you haven't already, I hope you try one of these installations and prove to yourself that your "Two Floppy Amiga" is really a powerful C language development system.

Program Modules, Next Issue

The above program is all well and good, but you are probably wondering exactly how I intend to use it as a module in other real projects. Next issue I will present how functions, or groups of



I Can't Find the Books?

The books referenced in this series are necessary tools for the serious Amiga programmer. Unfortunately, they may be a little hard to find, and a bit costly. A fantastic mail-order source for trade computer books is Communiqué Professional Books. Communiqué is a mail-order only bookseller that provides great service along with great mail-order prices...usually 20% off retail. Communiqué also features Free Shipping by US Priority Mail!—2-Day Shipping)

For more information and a complete price list, write to:

Communiqué Professional Books
4263 North Main Street
Suite 496
Fall River, MA 02720

ask for their Trade Computer Titles Catalog & Price List!

—Ed

Intuition and Graphics in ARexx Scripts

by Jeff Glatt

Using the ARexx function library, rx_intui.library, which adds a few dozen ARexx commands that allow an ARexx script to utilize Intuition and Graphics library routines.

Although ARexx has a built-in set of commands that cover the basic necessities of looping, math operations, file input/output, etc., it has no features that allow it to take advantage of the Amiga's two most notable aspects: Intuition and Graphics. For example, ARexx doesn't have commands for opening windows, attaching menus and gadgets, posting requesters, and interacting with and waiting for user input. Nor does it have commands for drawing lines, boxes, or printing text in various colors and styles. It doesn't support the mouse in any way. What ARexx does have is the facility to add "function libraries" which can add new commands to the default, built-in set. I have written an ARexx function library, rx_intui.library, which adds a few dozen ARexx commands that allow an ARexx script to utilize Intuition and Graphics library routines. This article demonstrates how to use this library to create an ARexx script with an Intuition interface.

The rx_intui.library should be copied to your LIBS: drawer, as well as to the dissidents requester.library, color.library, ilbm.library, and prtspool.library files. The rx_intui.library uses these other libs to implement certain features. I've included a program called "InstallLibs" that will do all of this copying for you. Make sure that you have about 36K free room in your LIBS directory. Either click on the InstallLibs icon, or from the CLI, cd to the directory with InstallLibs and invoke it. Obviously, you also need the ARexx product by Bill Hawes (not included here).

There is a doc file for the rx_intui library called "RexxIntuition." This document describes all of the new ARexx commands that are added, their arguments, and what they return. Too much information to include in this article, you should read that doc file now, and then study the example for each command as described in section A.



Once you've become familiar with what the rx_intui library offers, I wish to show an example of its use. You've probably encountered those "Directory Master" programs that feature a window full of gadgets that allow you to select disk files with the mouse and then copy, delete, rename, etc. The program may also allow you to format, copy, re-label, and make directories on disks by simply clicking on gadgets. This is an example of an Intuition interface. What I intend to do is use the rx_intui library to create a window with labeled gadgets, get mouse and gadget selections from the user (using the FileIO requester to get filename and directory selections), and then use ARexx's ADDRESS COMMAND to call the appropriate C: command to implement the operation upon the user's selections. CLLexx is the resulting ARexx script.

In the first few lines of the script, I open a window, and get two FileIO structures for use in obtaining AmigaDOS file and disk names. Then, I add 11 BOOLEAN gadgets to the window with ID's ranging from 0 to 10. (If you are unfamiliar with the difference between boolean, proportional, and string gadgets, you could consult the text file called Paint.txt accompanying the article "Implementing ARexx".) These gadgets allow me to select one of 11 operations performed on disks or directories. The first three gadgets allow me to COPY, DELETE, or RENAME files (i.e., text files, data files, executables) or directories. These gadgets will be labeled with the string 'FILES' above them. The next four gadgets allow me to create a new directory, or copy, format, or re-label (re-name) a disk. These gadgets will be labeled 'DISKS'. The next two gadgets allow me to read/edit text files by invoking the text editor of my choice, or print text files. These are labeled 'TEXT'. The last two gadgets allow me to display or print picture files. These are labeled 'PICTURES'.

The IDCMP loop is contained in the DO WHILE class > 0 loop. This loop continues until the user clicks on the window's close gadget. Inside the loop are tests for whatever actions the user makes using the 11 gadgets. The ARexx script is "put to sleep" when it calls the WaitMsg() function until the user clicks one of the 11 gadgets. Upon returning, I parse my IDCMP spec into the 4 components of class, part1, part2, and part3. For a gadget event, the class will be equal to 1. So I know

that the user clicked one of the 11 gadgets, and next I have to determine which one. The part1 contains the ID of the selected gadget. If the user selected the FILES COPY gadget, then part1 would be 0. Let's assume this to be the case.

Now, I have to bring up the file requester in order for the user to select which file(s) to copy. By specifying MULTIPLE_FILES, the user can select several files in one directory. Then, I have to bring up the requester again so that he can choose the destination directory. I'm going to copy the file to that directory, keeping the same filename. Note that I'm using a second FileIO so that I don't disturb the selections made using the first FileIO until I'm done with my copying. Additionally, I select the SHOW_DISKS flag so that the user is presented with a list of volumes. After the user selects the destination directory, I zero out the filename field of the second FileIO using the Poke function to make sure that only a directory path is returned (the destination must be a valid directory, not a file). Also note that I check both fname and toname for null ("") before proceeding. This would be the case if the user cancelled, or there was an error with opening the file requester. I display a "wait" mouse pointer using the RedrawWind function to indicate that this copying is now in progress.

Now, since I enabled MULTIPLE_FILES, I need to extract each selected filename from the first FileIO one at a time. The functions FirstEntry and NextEntry do this. These functions return a null filename when there are no more selections. That's why I use the DO WHILE fname > "" loop. Using GetPath, I obtain the complete source filename and the complete destination filename. (FirstEntry and NextEntry only return the filename part, minus its root path). Notice how I pass fname along with the appropriate FileIO. Finally, I'm ready to use ARexx's ADDRESS COMMAND to use the C: command "copy." I format the copy template. Note that I add double quotes around each complete filename. This is in case it contains imbedded spaces. The COPY command (as do most C: commands) requires double quotes around arguments with imbedded spaces. If not, the quotes won't matter anyway. I examine the return value. It should be 0 for success. If not, I post an error requester in the window. Next, I append 'info' to



the filenames and attempt to copy any icon file associated with this copied file. Note that I don't check for an error here because this icon file might not exist (i.e., if the user was copying the serial device). Finally, I get the next selection and repeat the process. I drop out when the last selection is copied. All of code for the other gadgets is skipped. I drop down to where I clear the "wait" mouse pointer. (Even if it was never set, this is safe to call.)

Most of the other gadgets are handled in a similar manner. I use the file requester to obtain the arguments, format some template for a C: command, and invoke the command. Sometimes, as in the RENAME gadget, I don't allow MULTIPLE_FILES, and therefore don't have to extract each selection. For the DISK operations, such as DISKCOPY, I'll invoke the SHOW_DISKS flag so that the requester comes up with a list of mounted volumes or devices instead of filenames in the current directory. (The user can invoke this list by selecting the mouse right button while the requester is open.)

Occasionally, I'll use the Input() command, such as with the RENAME gadget, if I only need to obtain one simple string. This saves having to bring up the file requester and extract one field from it (i.e., just the filename part).

For the MAKEDIR command, I need to make an icon for the newly created directory. Unfortunately, the C: command doesn't allow this option, so I achieve this by copying the "Empty.info" file from the user's boot disk. If he has deleted this file, then no icon will be made for the new drawer.

A special topic arises with the DISKCOPY command. Note that when I format the ADDRESS COMMAND, I'm specifying redirected input from the NIL: device (<NIL:). Normally, when DISKCOPY starts up, a message is posted to the CLI background window asking the user to press RETURN to initiate the copy. Since I have my own window open and active (i.e., all keyboard events go to my ARexx program instead of the CLI), the user never sees this prompt and won't respond to it. By re-directing input from NIL:, this forces that prompt to go away of its own accord, and allows the DISKCOPY to proceed without needing to be initiated by the user pressing RETURN.

The FORMAT command presents a special problem. I use the file requester to obtain the device name to format. The requester does not handle disks that are not formatted, so when the user makes his selection, a DOS requester will appear if an unformatted disk is in the selected drive at this time. Of course, selecting the DOS requester's CANCEL will make it go away, and the operation proceeds normally. An alternative to using the file requester would be to simply "hardwire" some boolean gadgets labeled DFO:, DFI:, etc. for the FORMAT command. A new window could be opened with these gadgets added to it. Each gadget would have a unique ID. The gadgets would be set to ISSUE_CLOSE, so that selecting one would close the window as well as return the

selected ID. A WaitMsg() IDCMP on this window would wait for and return the user's selection. This is an example of "rolling your own" input requester which this library makes possible. Here's an example alternative to the FORMAT code:

```
/* An example of a custom, input "requester" (actually a pop-up window) */
popwind=GetWindow('Select drive to FORMAT',,,,234,40,,608,4102)
IF popwind == '' | popwind == 0 THEN SAY 'Window open error'
errmsg = 'Gadgets open errors'

/* Add a bool gadget for DFO: with ID = 50. Set it to ISSUE_CLOSE */
gadget=AddGadget(popwind,1,'DFO',28,14,50,14,1,50,128)
IF gadget == '' | gadget == 0 THEN SAY errmsg
/* Add a bool gadget for DFI: with ID = 51 */
gadget=AddGadget(popwind,1,'DFI',84,14,50,14,1,51,128)
IF gadget == '' | gadget == 0 THEN SAY errmsg
/* Add a bool gadget for HD0: with ID = 52 */
gadget=AddGadget(popwind,1,'HD0',139,14,50,14,1,52,128)
IF gadget == '' | gadget == 0 THEN SAY errmsg
/* Wait for user event. Parse the 4 parts of spec into separate vars */
class1 = 1
DO WHILE class1 > 0
  spec1=WaitMsg(popwind)
  PARSE var spec1 class1 myPart1 myPart2 myPart3
  /* If a gadget selection, assign the ID to gadgetID */
  IF class1 == 1 THEN gadgetID = myPart1
END
err=EndWindow(popwind)

/* Now figure out which device string matches the ID */
IF gadgetID == 50 THEN tname = 'DFO:'
IF gadgetID == 51 THEN tname = 'DFI:'
IF gadgetID == 52 THEN tname = 'HD0:'
```

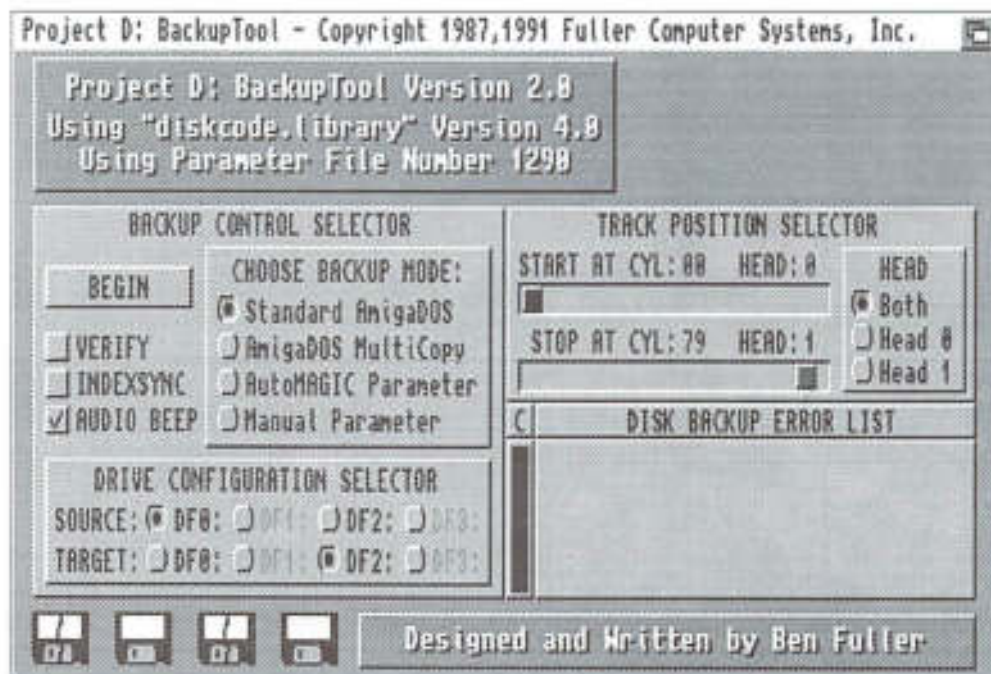
The READ gadget invokes a text editor in the C: directory called "ed." You can change this by altering the ADDRESS COMMAND for that code to the name of your editor of choice.

Note that the SHOW gadget uses the library's IFFLoad routine, allowing the library to open a new window/screen combo for the chosen picture. I do a WaitMsg() to wait for the user to click into that window when he wishes to end the display. I must close that window and screen even though the library opened them.

The PRINT picture code uses the Print() routine to print the picture. This requires that I load the picture into a window first. As soon as the Print() routine returns — immediately if there was enough memory to spool the print job, I can close the window/screen. It doesn't matter if the printing is still in progress (or yet to begin). For the text PRINT, I use the copy command to send the file to prt:, which is the DOS handle for the printer device.

This is one example of what you can do with the rx_intui library. There are many other uses. You could write a text editor in ARexx, or make a "control" window from which to launch a variety of applications. Using the DOS ser: and par: devices, you could write ARexx scripts to send information out of the serial and parallel ports using string or prop gadgets to obtain the data (i.e., a midi patch editor perhaps). Remember that you set such parameters as baud rate, data stop bits, etc. via preferences.





**The
Best
Amiga
Disk
Copier
Just
Got
Better.**



*Say Hello
to Project D 2.0*

Circle 163 on Reader Service Card

Available at fine Amiga dealers everywhere.

FULLER COMPUTER SYSTEMS, INC.

THE UTILITY COMPANY

Orders: (800) 874-DISK

Tech Support: (602) 497-6070

FAX: (602) 497-6071

AMIGADOS
Release

2

compatible

UNIX

and the Amiga

An Introduction to UNIX, for the Amiga Programmer—Part I

By Mike Hubbart

The Amiga's operating system (OS) is often criticized—it's either too slow, too buggy, or lacks certain "necessary" features. Commodore has worked on the reported OS problems and deficiencies by releasing updates, from the original V1.0 to the much heralded V2.0. But if the Amiga's OS is not MS-DOS nor Apple compatible, does it resemble any other OS? The operating system most like the Amiga's seems to be the UNIX operating system.

First For UNIX

There are two classes of UNIX: AT&T's System V and the University of California's BSD. UNIX has been around since the early 1970's, originally designed as a mainframe OS, although it has migrated to 80386/80486 IBM-compatibles. UNIX, now adapted by software vendors such as SCO and Interactive to run on IBM and compatible microcomputers, can replace the limited MS-DOS environment. UNIX is primarily seen at universities, yet is migrating to the business sector.

UNIX is an OS that supports multitasking and multiple users on individual terminals, but requires a much larger hard drive and more system memory than AmigaDOS (like OS2 and Windows 3.0). The complete Interactive 386/ix UNIX, SCO UNIX, and SCO Open Desktop packages each need at least a 110 to 200M hard drive (a 600M SCSI is preferable) plus from 4 to 16M of system memory (preferably fast 32-bit over slower 16-bit memory).

SCO's Xenix, a subsystem of UNIX for 80286/386/486 systems, can operate with a minimum configuration on a system with as little as 15M of hard drive space plus just 2M of memory, much less than UNIX. 12 - 15M is enough for the base Xenix OS, and just for a single user. The hard drive size plus memory requirements for a Xenix and UNIX system are affected by the number of people attached by terminals to the system — adding people will increase the amount of memory and hard drive requirements for a system.

UNIX allows one or many terminals (and hence other users) to connect to a single system, by use of a peripheral called a multiport I/O card. The I/O card plugs into an internal expansion slot in the main PC running UNIX, allowing from one to 64 terminals to connect by external cables to the main system. A smaller setup with the main system using its two serial ports with modems to connect terminals is also acceptable. Each user has his own log-in name and a private password for entry into the system.

A main UNIX system should be a fast (25MHz or better — although 16MHz and 20MHz systems will work adequately on smaller setups) 80386- or 80486-class machine, not on an 8086/8 nor a 80286 system. For those with a 80286 system that want to run a smaller UNIX network, there are versions of Xenix that will handle the job. Xenix will operate on computers with as little as 2M memory, yet most versions of UNIX require a minimum of 4M. For those with 8086/8's, sorry. There are a few software companies that have UNIX/Xenix lookalikes, and these programs will run on 8086/8 systems, even though neither UNIX nor Xenix will. If you want a UNIX clone for a 8086/88 machine, try MKS Toolkit, Coherent, or Minix.

Xenix comes in 80286 and 80386 versions, and both versions run on 80286-, 80386-, and 80486-class machines, with MFM, RLL, ESDI, and/or SCSI hard drives. SCSI hard drives are the best for use with either UNIX or Xenix, with Adaptec's 1542A and 1542B host adaptors as an excellent choice. However, the Adaptec 1522 host adaptor is not recommended at this time for either UNIX or Xenix. There are several vendors of SCSI host adaptors that work fine with UNIX. Check with the UNIX or hardware manufacturer before buying if you have doubts.

One or more UNIX/Xenix applications (a.k.a. programs) can run at the same time on a system used by one or many users. Applications can be large databases, where several people use different terminals to access a common pool of information.

Installing UNIX is a lengthy procedure — several hours at least for just the operating system. Both SCO and Interactive UNIX come on anywhere from 25 to 80 high density diskettes (either 5.25" or 3.5"), and it takes quite a while just to swap all those disks after the information has been read into the system. The number of applications (for example, network support, TCP/IP, and C Programming) chosen affect how many disks are needed, since the disks are organized by application. Both SCO and Interactive check the hard drive for errors before laying down the OS. The hard drive is used for temporary storage of data from system memory, when the system has a greater demand on memory than available from system resources. The hard drive needs to be as error-free as possible to avoid corrupting application data or the OS itself when the data is written to and read from the hard drive into memory.

With SCO's UNIX and Xenix, you can have an MS-DOS partition on the same hard drive as UNIX or Xenix. This is useful when you consider how many DOS business applications are in use today, and how many UNIX users may not want to give up their favorite DOS program. The procedure for a dual OS drive is simple:

- 1) Use DOS's fdisk to remove all DOS partitions on the hard drive;
- 2) Use DOS's fdisk to set the DOS partition first, use the DOS "format /s" to high level format the DOS partition, and then install DOS on the partition;
- 3) Boot the system from the UNIX/Xenix Boot disk;
- 4) Create a UNIX/Xenix partition, starting immediately after the end of the DOS partition;
- 5) Make the UNIX/Xenix partition active (not the DOS partition), and then complete the SCO installation.

When the system is turned on, SCO Boot: is on the screen. Pressing the <Enter> key will boot the system up in UNIX/Xenix, while typing "DOS" and pressing <Enter> will bring the system up in the MS-DOS environment. Your default boot string is stored in the /etc/default/boot file — it is 'DEFBOOTSTR='.

Now For The Amiga

So how does UNIX apply to the Amiga? Commodore is working on releasing a version of UNIX for the Amiga, which shows their support for this useful OS. Prentice-Hall has produced Minex For The Amiga, a mini-UNIX which works on Amiga 500/2000/2500/3000's. Additionally, there have been several UNIX shells for the Amiga — Jay Ts' T-Shell and a few shareware/PD shells found in the Fred Fish library.

Both the Amiga and UNIX can use either a command line environment (the shell) to enter commands, or they both can use a graphic interface. The Amiga and UNIX graphic interface

supports use of a mouse to select applications, saving the key-strokes needed to start the application and all the specified options. The normal Amiga Mouse has two buttons, while three-button mice are recommended for the X*Windows graphic environment. The graphic interface of UNIX is easier to use than typing in commands, since it uses icons just like the Amiga's Workbench. Both UNIX and the Amiga allow access to the command line from within the graphic interfaces — the Amiga has its Shell (just like SCO's Open Desktop) icon available when opening a main Workbench window (at least that's how I set up my hard drive's Workbench).

Both the Amiga and UNIX have several different shells, although UNIX has several different graphic environments — X*Windows, Open Motif, Looking Glass, and Open Desktop. Open Desktop is SCO's product that incorporates their SCO UNIX System V 3.2 Rev 2, into a graphic interface, and is available for around \$1000. Looking Glass and Open Motif are extra interface programs that cost around \$500 each, in addition to the cost of the UNIX package itself (priced from \$600 - \$1500 and up, depending on the applications needed).

Piping, the output of one command into another, is supported on the Amiga and in UNIX environments. An example of piping is: `#dir dh0:c | more`. This line directs the output of the directory of the c directory, into the MORE utility. Since More allows displaying a large file one screen at a time, you have time to see all the files in the C directory if there are more than your screen will hold at one time. Since the screen normally scrolls unless a key is pressed when listing a long directory, this particular example takes the guesswork out of knowing when to stop the scrolling.

Re-direction, an Amiga and UNIX feature, utilized (with the ">" and "<" characters) between commands on the same command line also saves a lot of typing time. Here's an example of Amiga re-direction:

```
# dir > prt:
```

This line re-directs a listing of the current directory out to the printer. This is useful when you have a lot of files on a disk and want a list handy for reference. This method saves time, since systems that fail to support re-direction require an intermediate step to accomplish the same purpose. UNIX does use lp filename to send output to a printer, although re-direction is supported as shown by the next example:

```
ls -l > special_file
```

This line redirects the output of listing the current directory from the screen to a file called special_file.

→

AmigaDOS and UNIX Commands

Copy and CP

Many UNIX commands are similar to Amiga commands. They use CP to copy files from one directory to another for UNIX, while Amiga CLI has the COPY command. COPY dh0:cando/text/testfile dh0:cando/text is same as CP /usr/text/testfile /usr/newdir.

The AmigaDOS Copy command has the following options: All, Quiet, Buffer, Clone, Date, Com, and Nopro. The UNIX CP command lacks these options, since re-direction of input and output is such an integral part of the OS.

CD and Date

CD and DATE are used by both AmigaDOS and UNIX to change/display the current directory, and to provide date information. Both the AmigaDOS and UNIX CD lack any optional flags. The AmigaDOS Date shows or sets the system time and has Date, Time, and To options.

The UNIX Date command is loaded with output display options. +%a uses the 3-letter abbreviation for day of the week; +%D has the date displayed in mm/dd/yy format; +%d day is shown as 1-31; +%H time is in 24 hr clock (0-2300); +%h uses the 3-letter abbreviation for month; +%j shows julian date, from 001 to 366; +%M shows minutes as 00 - 59; +%m shows month as 1-12; +%n inserts a newline, to add a space between the date and the next command line prompt; +%r adds AM or PM time designators; +%S shows seconds as 0 - 59; +%T shows the time as hh:mm:ss, such as 10:42:16; +%t acts to insert a tab, to separate the information; +%w shows the day of the week as a number instead of with letters — it shows a 0 instead of SUN for Sunday; +%y shows the year as 00 - 99.

These options can be chained together for more detailed information. For example, Date "+%D %a" which displays 02/10/91 Sun, for February 10, 1991. If the options were not enclosed within "", only the date 02/10/91 would be displayed. Another example would be, Date "%D%t%T" which displays 02/10/91 14:08:06, for the same information in the previous example, plus the time tabbed away from the date. Note that this example did not require the use of "" to encase the options.

Dir, LS, and LC

The Amiga DIR command gives a listing of the specified directory, while UNIX provides LS (short for List—an AmigaDOS command) and LC (short for List Column). With AmigaDOS, you might use DIR dh0:c or LIST dh0:c for a listing of all files in dh0:c. DIR (in AmigaDOS 1.3) only has a few option switches — A, AL, and D.

LS has many useful options: -a lists all entries, including files starting with "." — such as .profile; -b shows all non-printing characters in octal; -C displays the specified directory, in Columnar form; -c utilizes the "time" field for last modification time; -d lists directory names, but not files; -F has / to show directories & * to show executable files; -f lets you specify several directories at once; -g prints a long list (-l), but without the file owner; -i prints the inode number; -l for Long LiSt, which shows the file sizes, group, links, owner, mode, and when the files were last modified; -m prints out the files across the screen, separating each file with commas; -n prints the long list (-l), but utilize group/numeric id; -o prints the long list (-l), but excludes the file group; -p uses an "/" to indicate the directories; -q prints a "?" to display any non-printing characters; -R shows all files in the directory, as well as the files in subdirectories. -r displays the directory, sorted in reverse order; -s lists the files with their size in blocks; -t lists files sorted by the last time accessed, although the times themselves are not displayed; -u displays files based on the time of last access; -x displays the file across the screen in columnar form.

LC has the same flags as LS, except for the following differences. -A is the same as -a, except it does not show "." files; -l forces the display to single items for each line (just like the plain LS command).

ECHO

Echo is found in both AmigaDOS and UNIX. The AmigaDOS command has three options: Noline, First, and Len. The UNIX version has considerably more options. \b is a backspace; \c causes a line to print without a newline; \f causes a form feed; \n causes a newline; \# sends an octal value out; \r causes a carriage return; \t gives a tab; \\ gives a backslash in the output text.

MORE

More is a utility in AmigaDOS that allows someone to display a text file on the screen without using a text editor or word processor. Environmental variables are supported on the Amiga and in UNIX. If More is started from either the shell or CLI, setting the Amiga EDITOR environmental variable allows bringing up a chosen text editor while within More. Both the AmigaDOS and UNIX versions of More accept input via piping.

The UNIX version of this command has quite a few options. The + /pattern displays the text two lines before the start of the pattern specified; -c clears old then redraws screen for each page of text; -d get a prompt at the end of each screen; -f count lines within the file, using newlines instead of basing the count by number of screen lines (as is normal); -l ignore any formfeeds; -n for window size; +n set the line to start viewing, within the specified file; -r show control characters on the screen; -s elimi-

Recently, we were ordered by U.S. military officials to explain to their complete satisfaction just what a SuperSub is (as we all know, it's the best subscription deal around for Amiga users, since it includes both *Amazing Computing* and *AC's GUIDE*).

☆☆☆☆☆

Then, a prominent Congressman wired to ask us if we would testify before a top-secret subcommittee as to whether or not we can produce a single prototype SuperSub for less than \$500 million (is this guy kidding? – a one-year SuperSub costs just \$36 – and we can produce one for *anybody!*).

☆☆☆☆☆

Finally, a gentleman called us from Kennebunkport and told us to read his lips, but we told him we couldn't, because we don't have a picturephone.

And then he ordered a SuperSub.

**AC's SuperSub –
It's Right For You!
call 1-800-345-3360**

List of Advertisers

Please use a Reader service card to contact those advertisers who have sparked your interest. Advertisers want to hear from you. This is the best way they have of determining the Amiga community's interests and needs. Take a moment now to contact the companies with products you want to learn more about. And, if you decide to contact a advertiser directly, please tell them you saw them in

AC's TECH/AMIGA

Advertiser	Page	Reader Service Number
Amiga Warehouse	53	146
Black Belt Systems	2	118
Boone Technologies	41	194
Delphi Noetic	25	199
Fuller Computer Systems	63	163
ICD	CIV	123
Interactive Video Systems	CII, 1	140
Manx Software Systems	99	187
Memory Location.The	69	107
Memory Management	73	186
SAS Institute, Inc.	21	146
Vision Soft	43	183

nate multiple blank lines, displaying only 1; -u stop underlining or character enhancement; -w give a terminate prompt at the end of the file.

Amiga and UNIX Editors

Ed, an archaic/weak/boring screen editor, is found for the Amiga and UNIX. Both AmigaDOS and UNIX also have the EDIT line editor commands, and both support piping — which connects the output of one process or command into another. Another surprise — Memacs is available on both Amiga and UNIX platforms.

The most powerful UNIX editor is Vi, pronounced "Vi", "Vee-Eye", or "Six" by people in the UNIX crowd. Although having used it for years, I still feel Vi is probably the least intuitive editor I've ever used. I use Vi regularly when C programming in a UNIX environment but would note that it takes a while (or a good help chart on-hand) to learn it well. This editor comes with SCO Xenix & UNIX, and with Interactive UNIX. There are several good pd/shareware versions of Vi for the Amiga in the Fred Fish Library (such as Stevie) for the curious. The Manx C compiler comes with Vi, for those diehard UNIX people that are so used to it and prefer it over Amiga text editors with pull-down menus like TxE+ and CED. Although not provided with standard Amigas, Vi does come with the Amiga 3000UX.

Fred Fish Amiga UNIX Goodies

For Amiga owners wanting to use UNIX commands or utilities on their systems without going to a UNIX OS, there is a way. If you haven't discovered yet, the Fred Fish Library is loaded with goodies for programmers, utility addicts, and UNIXheads. It would take too much space to cover most of the authors that have contributed time and energy to produce UNIX commands and utilities for the Amiga community, but we will mention a few and some of the UNIX items they wrote or ported. Not all the versions of a program or utility by the same or different authors will be mentioned.

For communication setups, William Loftus and Matt Dillon have ported UUCP to the Amiga. UUCP stands for UNIX to UNIX Communication Protocol. William's port is on FF 152 and includes the cron, mail, and compress UNIX commands and utilities; Matt's version of UUCP appears on FF 360. Matt Dillon also wrote DNET, a program that allows linking an Amiga to another system running BSD4.3 UNIX. This program appears on FF 294.

As for UNIX commands and utilities, there are many. UNSHAR, by Eddy Carroll, appears on FF 345, while Gary Glendown's MAN is on FF 241. Justin McCormick brought us LS on FF 236, while George Musser and Paul Kienitz produced a version of WHO that tells what tasks are currently running on the system. Edwin Hoogerbeets wrote MV, a program from FF 219

that does the same as UNIX's MV, CP, and RM commands — it will move files, copy files, or remove files. G.R. Walter released STEVIE (on FF 217), a port of the UNIX VI editor — not a popular one since it is not as intuitive as many would prefer. GNUGREP is Henry Spencer's program that does the same as the UNIX commands grep, egrep, fgrep, and bmgrep. It is on FF 295. Tomas Rokicki's SPELL, on FF 191, is a port of the UNIX spell checker and very useful for those using a PD/shareware text editor to write letters. One last author I'll mention for now is Gary Brant, who released UNIXUTILS on FF 179. UNIXutils have WC, Head, Tail, Tee, Detab, Entab, and Trunc, and I can attest to the usefulness of the WC word counter program for writers.

Amiga UNIX Shells

For the Amiga, there are a couple of choices for UNIX shells. Matt Dillon has Csh, his Csh-like shell for the Amiga out for several years, which he continues to upgrade for us. I started out with Matt's Csh, since I had only a 512k system with a single drive and wanted to do more than swap disks whenever I wanted to run a program from a disk other than the Workbench disk. Csh has built-in commands that are available for use even if the disk is out of the drive, so it is a lifesaver for single drive systems with little memory to waste. The HELP key on the Amiga keyboard brings up a list of the built-in commands for Csh. I'm glad to see someone actually using this key. Csh still gets plenty of use, even though my setup has grown considerably since first purchasing an Amiga. Csh appears on Fred Fish disk 331.

Although it is newer to the Amiga, I'm impressed with Steve Koran's SKsh — a UNIX Korn shell lookalike. Setting up SKsh takes a bit of time — if you try running it without reading the information, you'll definitely visit the guru (unless you always run a 10k stack)! I made a boot disk for SKsh, and also installed it on my hard drive (along with Bill Hawes' Wshell).

SKsh offers several enhancements over the AmigaDOS 1.3 shell. Command substitution, definable shell functions, local variables and local aliases, and EMACS-style line editing are a few of the new features. I/O re-direction, pipes, and ARexx are supported in SKsh. More than one command can be run from the command line by using a semi-colon to separate them. SKsh has both built-in and external commands, and the external commands can be resident. The UNIX wildcards are supported, so feel free to use *, ?, and [] in place of known characters. There is sufficient documentation provided with SKsh, including information on known bugs, built-in and external commands, how to install the shell, and reference information on examples for writing scripts in SKsh.

SKsh needs AmigaDOS 1.3, plus a stack of 10,000 minimum to function. Both .skshinit and .skshrc must be in the S: directory, and arp.library must be in the LIBS: directory. Something to note:

this program has reported errors when running with older versions of arp.library (no dates were given), so use the most current version available. The external commands, located either in the C: or a user-selected directory, are cat, cmp, cp, crc, cut, du, encr, fgrep, grep, head, indent, join, num, srun, strings, tail, tee, view, wc, window, and xd. SKsh is available on Fred Fish disk 342 and is freely re-distributable, but it is not in the public domain since it is copyrighted by Mr. Koran. While Steve does not ask for shareware status, this product is certainly worth a donation so that Steve will continue to upgrade it.

Commercial Amiga UNIX Products


There are currently just a few commercial Amiga Unix products. Jay Ts' T-Shell has been out for several years, although it hasn't been updated for quite a while. The Tshell and all its applications eat up space. I load the help info into RAM, speeding up the display of info from this shell. Loading the help files and some of the applications on a hard drive also speeds up things. When I copied the help files from the floppy into memory, it took 4 min., 17 seconds for the T-Shell to be up and running. When I ran B.A.D. on the copy of the T-Shell disk (never on an original), the loading time dropped to 2 min., 5 seconds, clearly showing the worth of optimizers. T-shell uses UNIX-like commands, and allows the user to also use Amiga commands so learning the shell is painless.

Prentice-Hall's Minex For The Amiga is a new product. Although not advertised, this program deserves some attention. Minex was previously available only for IBM 8086/8 and 80286-class machines. The Amiga version is \$16 third party vendor (Unipress) that has C source code only for the IBM version that is just \$99. Let's hope they bring it over to the Amiga Minex!


Summary

Very few people have claimed UNIX is easy to learn. Yet this powerful OS has many features found in the Amiga, even though the system hard drive and memory requirements are much greater than what the Amiga needs to do the same task. UNIX can be a convenient platform for the Amiga, allowing Amigas to run on networks with other PC's, thereby helping to persuade businesses to go with the Amiga. We already know that the Amiga is an ideal and inexpensive platform for multimedia and animation.

Passing information and files between the Amiga and other platforms easily could spur on sales needed to keep the Amiga a major player in the market for a long time. Seeing how Commodore is pushing the 3000UX at UNIX expos around the world, they (and the IBM/UNIX community magazines seem to agree) obviously feel so. The Amiga 3000UX was shown at UNIX Solu-





The Memory Location



Amiga specialists! Full service Commodore dealer.
Commodore authorized Educational dealer.

Pulsar Power PC
4mb 2630 Card (25mhz 68030)
2232 Multi-Serial port card
1950 Multisync Monitor
AE High density 3.5" drive
Mi Graph Hand Scanner
Sharp JX-300 Color Scanner
Sharp JX-100 Color Scanner
Xapshot still video camera
Canon RP-420 Video printer
Gold Disk Office
Disney Animation Studio
BibleReader
Amix II
Home Front
Wings
Shadow of the Beast II
Black Gold
Heart of the Dragon
Pool of Radiance
Check Mate
Over Run
Second Front









The Memory Location

396 Washington Street
Wellesley, MA 02181
(617) 237 6846

Store hours: Mon.-Thur. 10-6, Fri. 10-8, Sat. 9-5

Commodore authorized repair on-premise. Low flat labor rate, plus parts.

Circle 107 on Reader Service card.

tions at Anaheim, CA, at UNIX Expo in New York, at Comdex in Las Vegas, and at Educom in Atlanta, GA. The 3000UX comes configured an Amiga 3000 with either a 4M/100M (4M memory and 100M hard drive space) or 8M/200M system. CBM recently started publishing Amiga UNIX Tech Notes to keep Amiga UNIX users current on tips and new ideas.

Should UNIX receive more support from Amiga owners or developers at this time? I think anything that would help expand the sale of Amigas, without hurting the Amiga's image as a serious computer, deserves consideration. There are many UNIX tools to examine in the Public Domain but not enough in the commercial marketplace. The Amiga 3000UX, Minex, and Tshell are useful, but I hope we'll see even more UNIX support from many Amiga developers in the near future. The next time you pick up a Fred Fish disk, try out some of the UNIX commands (if any are there) and see how similar they feel when compared to the Amiga's commands. You may then feel like trying out a better shell than the WorkBench 1.3 Shell—one like Steve Kern's SKsh or Matt Dillon's Shell. In the next several UNIX articles here in AC's TECH, we'll look at setting up both the SKsh AND Dillon's Shell, starting with a WorkBench disk and cutting it down to the bare essentials.



A Meg and a Half on a Budget

Add 512K RAM to your 1MB A500 for about \$30

by Bob Blick

Project Description

Here is an inexpensive and easy project for the do-it-yourselfer to boost his Amiga 500 from 1 megabyte of RAM to 1.5 megabytes. Unlike most after-market memory upgrades, this doesn't require you to discard your A501 memory board. With this project you add extra RAM chips to your existing memory board by piggybacking the new chips on top of the old ones, and adding a small circuit board containing the necessary logic to enable the new RAM. The memory auto-configures and no special startup sequences are needed. Total investment: one evening and about \$30.

No special tools or parts are required for this project. The RAM control circuit can be hand-wired on perfboard, or a printed circuit board can be made using the layout at the end of this article. An etched and drilled circuit board is also available. Anyone who has electronic circuit or kit-building experience should be able to complete this project in a few hours. However, this is not something for the faint at heart — you must take your Amiga completely apart and put it back together again, and you'll have to do some soldering. Beware! All warranties on your Amiga will be voided, and you alone will bear responsibility for failure or enjoy the fruits of success. Many of these upgrades have been supervised by the author, but nothing's guaranteed — you must double-check everything and be prepared to undo what you did if you can't get it to work.

System Requirements

The Amiga 500 comes stock with 512k RAM. If you're like most Amiga 500 owners, you bought a Commodore-Amiga A501 memory cartridge. It gives you an extra 512k of RAM for a total of 1 megabyte and has a clock/calendar chip in it. It's housed in a triangular metal box that plugs into the bottom of the Amiga behind a plastic cover. You should check, however, because there are many A501 clones by other manufacturers. The only real difference is usually the omission of the metal covers on the clones. The upgrade will work with almost all boards, but my directions are specific for the A501, so you might need to figure out some things for yourself if you don't have the A501.

There are a few different versions of the A500 motherboard. I've seen only one version in all the computers I have looked into. You'll have to remove the covers to determine your version. Mine has "A500 REV 5" etched into the motherboard between the disk drive and the A501 expansion connector. This upgrade will work on any Amiga 500 that has 512k on the motherboard and an A501-type memory expansion. If your motherboard is not a REV 5, there may be some differences.

The power requirements for this expansion are minimal, so the power supply of the Amiga will not be over-taxed.

The Amiga 500 has a DMA controller chip known as "Fat Agnus." The number for it is 8370 or 8371. If you have upgraded to the 8372 "Super Fat Agnus," also known as "1 Meg Agnus," this project will not work for you. Check in a future issue for a 2-megabyte project for "Super Fat Agnus" owners. If you don't know which Agnus is in your Amiga, boot with a Workbench disk, open a Shell window, and type in "avail" <return>. Leave out the quote marks. You will be shown how much "chip" and how much "fast" memory you have. You should have approximately 512k each. After this RAM expansion, you will have 1 meg of fast and 512k of chip.

Getting Started

Before getting the parts, you should familiarize yourself with your Amiga by taking it apart. Take out the A501 memory board from the bottom of the computer first. Pry off the plastic door and slide the A501 out. If it's a real A501, it is encased in a triangular metal box. The box has four metal tabs bent over and soldered. Pry them back with a soldering iron and open the box. A battery is on the board, so don't set the bare board on a metal surface or otherwise you will short circuit the battery. You should see sixteen chips with sixteen pins each (256-kilobit RAM chips) and one chip with eighteen pins (clock/calendar chip). Other memory boards with sixteen RAM chips are compatible with this project. If you have only four RAM chips with twenty pins each, you have 1-megabit chips. This will require changes in the parts needed and in the modification of the memory board.

"With this project you add extra RAM chips to your existing memory board by piggybacking the new chips on top of the old ones, and adding a small circuit board containing the necessary logic to enable the new RAM. The memory auto-configures and no special startup sequences are needed. Total investment: one evening and about \$30."



There are six T10 Torx screws holding the case of the Amiga together. If you don't have a Torx driver, use a 2 millimeter or 5/64 inch allen wrench. A small screwdriver might also work. The top of the Amiga will snap open easily, exposing the keyboard and shielded metal cover. Disconnect the keyboard by unplugging the ground wire by the disk drive and the multipin connector inside the metal shield. Make a note of the way it plugs in, with the black wire on the left. Bend back the metal tabs, remove the four Torx screws, and the metal shield. You'll see the motherboard, so now is a good time to check whether you have a REV 5 motherboard (printed on the right side of the motherboard near the disk drive). If you have a different motherboard, the directions for the modification of the motherboard may not be right and you'll need to use the schematic as a reference.

Double-check the Fat Agnus chip. It's the big square one and may have a metal clip holding it in. It should have 8370 or 8371 printed on it, as well as other numbers.

If you are satisfied with everything, check the parts list, get the parts you need, and get to work!

So Build the Thing, Already!

There are three steps to expanding your RAM: building the RAM control board, adding RAM chips piggyback-style to your A501 board, and connecting the RAM control board to the Amiga's motherboard. Let's start with the RAM control board.

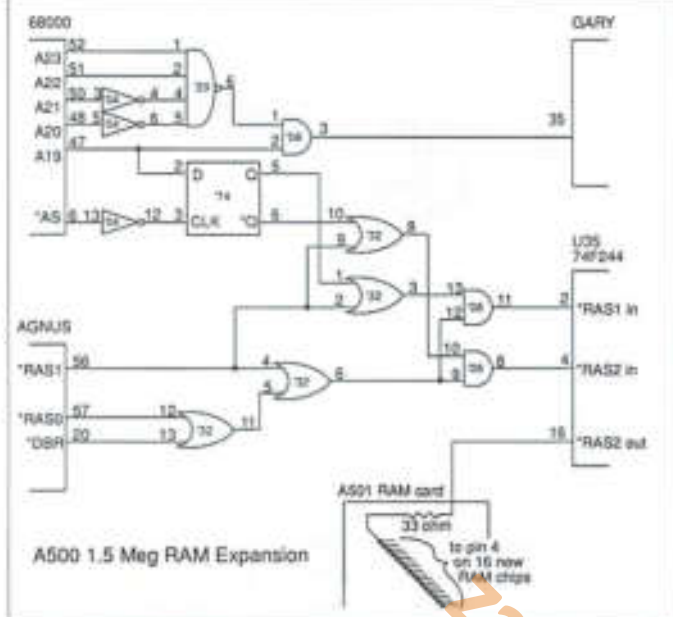
The RAM control has five simple chips on it and sends signals to the RAM chips. You can hand wire the circuit on perf board, but it's easy to make mistakes that way. A printed circuit board is fairly foolproof. The easiest way to obtain the board is to purchase an etched and drilled PC board from the author. If you like to etch printed circuit boards yourself, I recommend using TEC-200 film to transfer the image to copper-clad circuit board. With this process you photocopy the PC board layout onto TEC-200 film and then use a regular dressmaker's iron to transfer the black photocopier toner onto the copper-clad board. It's now ready to etch using ferric chloride etchant. Check the parts list for a source of TEC-200 film. Directions are included with each package.

Follow the parts placement diagram for stuffing the PC board. I recommend using sockets for the chips. Pin one for each chip is indicated on the board with a rectangular pad. Line the notches on the sockets up in the direction of pin one. Solder the sockets to the board. The five 0.1 microfarad capacitors are indicated by ovals in the diagram. Solder them and clip the leads close to the board. The wire you cut off can be used for the six short jumper wires. Solder them in and clip the excess close to the board. Use ribbon cable for the wires that will connect the board to the Amiga. Prepare the ends by stripping and tinning with solder. You need a piece with six wires, one with five wires, one with two wires, and one single piece, all six or eight inches long. Attach them to the RAM control board as shown. Press the chips into their sockets, aligning pin one with the notches. This completes the RAM control board.

Modifying the A501 board involves a lot of soldering — over 240 connections. It's repetitious and boring work and hard to do every solder joint right the first time. But after a little practice it gets pretty easy. You must piggyback the 16 RAM chips on the board with another RAM chip and solder the pins where they touch each other. Test fit the chips to see whether they fit snugly over the chips on the A501 board. If every pin touches, it makes soldering easier. If you need to, use the flat surface of a tabletop to bend in the pins a little. Next you must bend out pin four on all the chips. All of the pins of the piggyback chips, except for pin four, get soldered to the bottom layer. Pin four is *RAS, or Row Address Strobe, and it needs to be connected to the RAM control board. Note: *RAS is also pin four on 1-meg chips with 20 pins.

Take a pair of needlenose pliers and bend the lower part of pin four out to the side on all 16 of the new RAM chips. Don't bend the pin right near the body of the chip, as it may break off. Bend it at the thinner part of the pin. Solder the chips one at a time to the A501 board, starting at the end away from the connector of the A501 board. Use thin solder and a soldering iron with a miniature tip. Getting a good solder connection involves heating both pins. It's hard to get to the lower layer without solder bridging to nearby pins. Let the chip cool off after every few pins if it gets really hot.

A500 1.5MB RAM EXPANSION



After soldering all the chips, connect pin four of all the piggyback chips together. Wire-wrap wire or other small gauge solid wire is useful for this step. With wire-wrap wire it is possible to use just two or three pieces of wire to connect all the chips, daisy-chain style, and still have insulation on the wire between each chip. The insulation on this type of wire will slide along easily, so you can strip inch-long sections of insulation on one long piece of wire. Connect the two rows of eight chips together as shown, along with a 33-ohm resistor to the last chip. The other end of the resistor gets soldered to a previously unused terminal on the connector. In the Amiga schematics it is numbered pin 37 and is the tenth pin from the back end of the connector, on the top row. Double-check your work, but don't put the metal covers back on the board at this time.

The RAM control board must intercept signals between chips in the Amiga. A total of four circuit board traces must be cut on the motherboard. Three cuts are on the bottom, so the motherboard must be removed. After disassembling the Amiga as described earlier, continue by removing the three disk drive screws on the bottom and one inside the Amiga. Disconnect the two disk drive cables from the motherboard, noting the direction they are connected (red towards the disk drive). Remove the two hex-headed screws on the RGB video connector. Pull the motherboard out of the case and take the metal shield and cardboard insulator off the bottom of the board.

Before cutting any traces on the board, look at the inset box on the parts placement diagram. One of the wires from the RAM control board needs to connect to pin 56 of the Agnus chip via a soldered hole in the motherboard. As indicated in the diagram, it is near RAM chip U26 towards the front of the motherboard. If you are unsure, you may want to use an

ohmmeter to double-check. This soldered hole is also connected to pin 35 on the Gary chip. Use the ohmmeter to check the connection from this hole to Gary pin 35.

Locate the Gary chip by the markings on the board. It has 48 pins. The pins are numbered from the top, counterclockwise starting at the notched end. Pin 35 is on the side towards the front of the computer. If it checks out, find pin 35 on the bottom of the board. The thin copper trace leading to the pin must be cut. Use a sharp knife or razor blade and cut the trace. You need remove only enough copper to break the connection. Locate the chip U35. It is near the front of the computer close to the motherboard's RAM chips. It is marked 74F244 and has 20 pins. Do not confuse it with U34, another 74F244 close by. On the bottom of the board, cut the traces leading to pins 2 and 4 of U35.

Turn the motherboard right side up and plug the A501 board into it. Note the pin that connects to the resistor you added to the A501 board. You may remove the board now. On the motherboard you must cut the trace to that pin. It is the tenth pin from the back of the computer. This completes the cutting. Except for the wire to the soldered hole, the connections from the RAM control board to the motherboard are soldered to chips on the motherboard. Before soldering on the wires, tie the pins close to the body of the chips with some solder.

The wires from the RAM control board are long; you may wish to cut them shorter for neatness. The board is meant to be located in front of Agnus and held in place with double-sided foam mounting tape. Five wires connect to U35, two wires to Gary, one connects to the soldered hole. The other group of six wires connects to the 68000. This is the largest chip, at the left edge of the motherboard. It has 64 pins. Five of the wires connect on the far left side. The wires are in reverse order and the ribbon cable will do a flip. Note that pin 49 should not be connected to anything. After all the connections are made, double-check your work for solder globs and stray bits of wire. Your Amiga is ready to test.

Checkout and Reassembly

Before re-assembling the computer, you would be wise to check that everything works as planned. Set the motherboard on a non-metal table and attach the two disk drive cables and one or two screws to keep the disk drive from flopping around. Hook up the monitor, mouse, and power supply. Don't plug the A501 board or keyboard in yet. Turn on the computer and boot a Workbench disk. The computer should boot and show 400000 or so of RAM. If the computer doesn't work, chances are good you have connected one or more RAM control board wires to the wrong place or have a short circuit, solder bridge, or cut more than four traces. It should be easy to find the problem.

If the computer works, turn it off and plug the A501 board in. It is possible to offset it by one pin, so pay attention. Turn the computer on and Workbench should boot and show about 1400000 free memory. Make sure windows, icons, and menus work properly, and turn the computer off and try again just to

make sure. If the computer doesn't work, check your A501 board carefully for solder bridges. If the computer works but is flaky or shows less than the right amount of memory, check the A501 board for pins that aren't really soldered. Check the RAM control board to make sure the chips are in the correct sockets. You also might have forgotten to put the chips in!

This completes the testing. You may re-assemble the computer. The metal cover for the A501 board will need a few taps with a hammer before re-assembly in order to fit over the piggybacked chips. You could leave the cover off, but the computer will produce more radio interference without it. Make sure the insulating sheet goes on with the bottom cover. Before shutting the case on the computer, make sure the disk drive is connected properly and check that the plastic sheet is in place on the back of the keyboard, and remember to re-plug the keyboard. After assembling the computer, test it again before putting your whole system back together again.

How It Works

The circuit on the RAM control board has two main features. One part tricks the Gary chip and the other part tricks the Agnus chip. One of the functions of the Gary chip is decoding RAM addresses. The 74F20 detects when RAM is being addressed in the 512k block just above the normal expansion block of RAM. When this happens, the Gary chip is fooled into thinking that normal expansion RAM is being addressed. The 74F74 flip flop detects which bank of RAM is being addressed and locks the other bank out.

Kickstart 1.2 and 1.3 automatically check for RAM at these addresses, so this RAM is auto-configuring as FAST RAM. Any memory that connects to the A501 connector shares the same bus as CHIP RAM. However, the Amiga only distinguishes between CHIP RAM and FAST RAM. This type of memory is sometimes called SLOW RAM. The only real way to have FAST RAM is to connect to the processor bus. That is the connector at the far left edge of the Amiga, hidden by a small plastic cover. Memory connected there is really FAST RAM and can speed the Amiga up by as much as 35%. The additional required circuitry is significant, so most manufacturers make memory expansions that plug into the A501 connector.

MM Memory Management, Inc.

Amiga Service Specialists

Over four years experience!
Commodore authorized full service center. Low flat rate plus parts.
Complete in-shop inventory.
Memory Management, Inc.
396 Washington Street
Wellesley, MA 02181
(617) 237 6846.

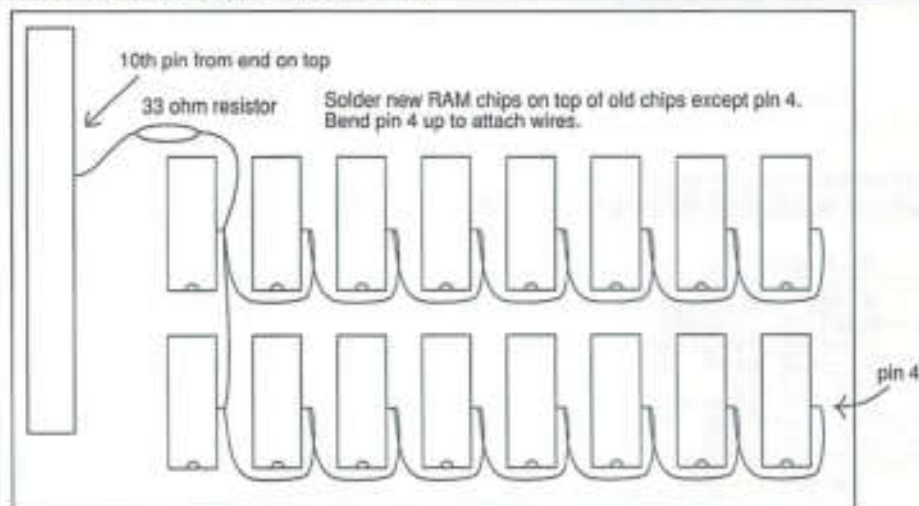
Circle 186 on Reader Service card.

During RAM refresh all RAM (chip, expansion, and piggyback) must be enabled. The rest of the circuit takes care of this. When Agnus asserts pins 20, 56, and 57, both expansion and piggyback RAM are enabled. There is an unused section of the 74F244, U35 on the motherboard. It is used to buffer the CAS signal to the piggyback RAM.

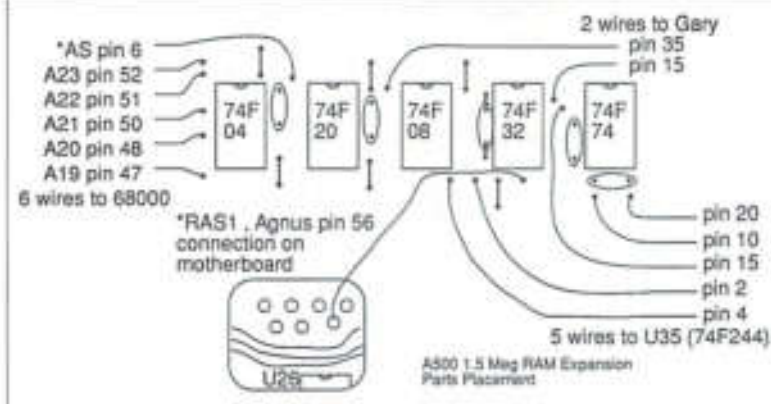
Substituting Parts

Dynamic RAM chips are made by many companies and have many different part numbers for equivalent parts. The ones you want are 256k by 1 bit. The access time should be 150

A501 CARD MODIFICATION DETAIL



PARTS PLACEMENT



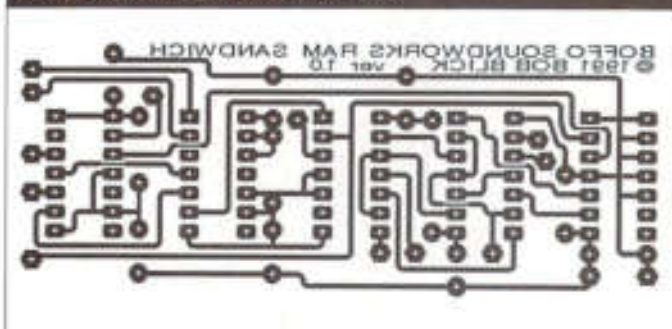
nanoseconds or less. Faster will not make your system run faster, just cost more. These chips are typically called 41256-15, and any company that sells RAM for IBM-type machines will be quite familiar with them. The prices are quite competitive, so shop around. If you have a memory board with four 20-pin RAM chips, you need four 1-meg (256k by 4 bits each) RAM chips of the 514256 style.

In order to prevent timing errors caused by signal delay in the added logic, 74F (as in Fast) series chips are specified. They have a typical propagation delay of 2 nanoseconds. 74LS series chips have a propagation delay of approximately 10 nanoseconds. Some signals must go through three gates, allowing only a 6 nanosecond delay with 74F series, or 30 nanoseconds with 74LS series chips.

When you purchase your parts, it is unlikely that you will find all the 74F chips. It is sometimes hard to find 74F20s. No matter, I have substituted all the chips with 74LS series chips as a test and it works. You may substitute 74HCT, 74S and 74LS for all parts except the 74F32. Don't, however, substitute a 74S32; the load on pin 56 of the Agnus chip would be too high.

This project has been tested on a normal two-floppy drive A500 system with many combinations of parts. If you add a hard drive or expand your Amiga in other ways, the extra 24 nanoseconds from using all 74LS chips might cause your system to be marginal. This will show up as random Guru messages at bootup. You should try to use as many 74F chips

RAM SANDWICH PCB PATTERN



as possible to prevent this. Don't let this scare you. I couldn't get this circuit not to work. Go by the book as much as possible and you'll have no problems.

Parts List

- 16-256k dynamic ram chips,
- 150 nanosecond 41256-15 or equivalent
- 1-74F04 hex INVERTER
- 1-74F08 quad 2 input AND gate
- 1-74F20 dual 4 input NAND gate
- 1-74F32 quad 2 input OR gate
- 1-74F74 dual D FLIP FLOP
- 5-0.1 microfarad ceramic disc capacitors
- 1-33 ohm resistor, 1/8 watt or greater
- 5-14 pin solder tail chip sockets
- 1-printed circuit board or 3 inch piece of perf board
- scrap piece of ribbon cable, 6 to 8 inches long
- rap wire or hookup wire
- ided mounting tape

Notes

An etched and drilled printed circuit board is available from the author for \$10 postpaid. Send a check or money order to Bob Blick, Box 916, Mendocino, CA 95460.

10 sheets of TEC-200 image film for making printed circuit board transfers with a laser printer or photocopier is available for \$5.95 plus \$3 p/h from DC Electronics, Box 3203, Scottsdale, AZ 85271. (800)423-0070.

Substitute for 74F series chips in this order:
74HCT, 74LS, 74ALS. Exception: do not use 74S32 in place of 74F32.

A501 clone memory boards with four 20-pin 1-meg chips will require four of the same generic type of RAM chips (514256 type) instead of sixteen 41256 RAM chips.

About the Author

Bob Blick is an electronics teacher who also does technical work for audio and video studios. He plays drums and enjoys watching tennis. Bob has been building projects for the Amiga since 1986. He can be reached on MCAB-BBS (707) 937-3056 as BOB BLICK and on PLINK as BOFFO*BOB.





AC's *TECH* *AMIGA*

Technically Speaking,
It's The First.

Now That You Know
It's Also The Best,
Don't Just Sit There –
SUBSCRIBE!

Hurry!

Special Charter Subscription Offer –

4 Big Issues – Just \$39.95

(limited time only)

Use the convenient sub card on page 33
or call 1-800-345-3360

Next issue of **AC'S TECH** available July 1991.

Accessing Amiga Intuition Gadgets from a FORTRAN program

Using Boolean Gadgets

by Joseph R. Pasek

PLINK: OWR821
CompuServe: 74555,1117

Introduction

The implementation of a program on the Amiga can be done as simply as the programmer wants — a quick "hack" where program inputs and outputs are conveyed by the Amiga's CLI. This is O.K. when the programmer wants a quick check or something for his personal use. Or maybe the program is just better suited if used from the CLI. For those programming gems, that may mean many things to a wider circle of users. All the obvious indications point to using the Amiga's Graphical User Interface (GUI); it should be used as fully and as correctly as needed for the application.

The availability of the Amiga's CLI makes the quick hack almost too easy to accomplish. But a piece of software that is destined to be used by others and used often should be carefully considered as a candidate to incorporate the features provided by Intuition. The worthwhile Amiga program written in FORTRAN should be no exception to this rule. This brings us to the main topic of this article, interfacing a FORTRAN program to Amiga's ROM Kernel routines.

The intent of this article is several-fold. First, it is to continue an effort featured in the last issue of the magazine that shows how to use proportional gadgets from a program written in FORTRAN. Here, I will address the use of Intuition's Boolean gadgets from a program written in FORTRAN. Second, I will show an alternative way of accessing the Amiga's ROM Kernel routines more directly than the means provided by Absoft's `amiga()` routine. In a number of cases this more direct access enhances the performance of the FORTRAN-coded application. And third, I will show some programming examples done in FORTRAN that may be useful to programmers in general.

Example Code Description: Jupiter's Moons' Simulator

The vehicle through which I will demonstrate access to the Intuition gadgets is achieved with a moderately-sized FORTRAN-code example. Actually, after seeing the code, you will no doubt be quite happy that this magazine comes with a disk.

This example will provide the more astronomically-inclined Amiga-user a reasonably accurate representation of the Jupiter's Galilean moons' motion over the next 28 days or so days.

The Jupiter's moons' motion model that is used here is taken from Chapter 36 of Jean Meeus' book *Astronomical Formulae for Calculators*, 4th Edition, Willmann-Bell Inc., 1988. The described method permits the calculation, for any given instant of time, of the positions of the four great satellites of Jupiter with respect to the planet, as seen from the Earth. Quoting Jean Meeus: "The results are good, but not extremely accurate, and therefore may not be used for accurate calculations." A more accurate version of the program is in the works.

The program generates a diagram that is similar to the one shown in the astronomical calendar sections of such magazines as *Sky and Telescope*, *Astronomy*, and astronomical almanacs from various sources (Figure 1 provides an example of such a diagram). The program completes its depiction of the expected motion over the next 28 days. The region to the left of the moons' motion graph shows, during the start-up of the program, four dots representing moons in motion around a stripped disk that is supposed to represent Jupiter's disk. The disk shown is a bit larger than the actual perceived disk as seen through a telescope (a picture of the screen generated is shown in Figure 2).

This program, written in Absoft's FORTRAN 77, will show how to access Intuition's Boolean gadgets, based on the newly defined structures in the Premiere Issue of AC's Tech / Amiga. In addition, I will describe a machine-language-based direct interface to the Amiga's ROM Kernel that is accessible from the FORTRAN. Also, it is hoped that the code provided will be a reasonable example of writing a structured program in FORTRAN.

The Jupiter Moons' Simulation Program Description

The program is divided into a main program and a number of subroutines and functions. The main program

Using an alternative way of accessing the Amiga's ROM Kernel routines more directly than the means provided by Absoft's amiga() routine. In a number of cases this more direct access enhances the performance of the FORTRAN-coded application.

JMoGraph5 is the main point of control of the processing. Program JMoGraph5 is also where most of the graphical interface is defined and generated, and from where the user input is interpreted. Subroutine Makepics() is also used to generate the pictures of Jupiter and the various positions of the moon for a given date.

All graphics generated by the program are done entirely by the main program and the subroutine Makepics(). Figure 2 describes the main features of the interface generated by this program.

The subroutine or functions called can be divided into four categories:

- Direct and Indirect Interface to Amiga's System Routines
- Astronomical Application Routines
- Application-Oriented Routines
- Language System Provided Routines

In the first category, System Interfacing Routines are all those routines that allow the FORTRAN program to access the Amiga's OS and GUI. This includes the means provided by Absoft, the amiga() routine. The amiga() routine can be used as either as a subroutine or a function. Its use is as follows:

```
result = amiga(SysRoutineName, arg1, arg2, ..., arg n)  or
call amiga (SysRoutineName, arg1, arg2, ..., arg n)
```

where...

```
SysRoutineName = Name of Amiga ROM Kernel Routine
arg1, arg2, ..., argn = arguments of a given Amiga ROM Kernel Routine
```

The other system interfacing routines that more directly access the ROM Kernel are written in M68000 assembly language. Some of the routines produced in an ongoing effort to implement for FORTRAN programmers a more direct access to the Amiga will be described later. The use of these routines serves a couple of purposes. The first is

April, 1991 Configuration of Jupiter's Satellites

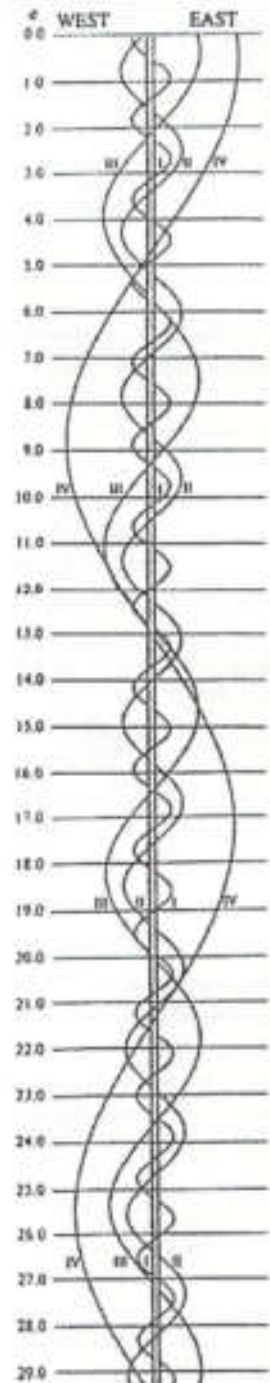


Figure One
Sample Astronomical
Calendar

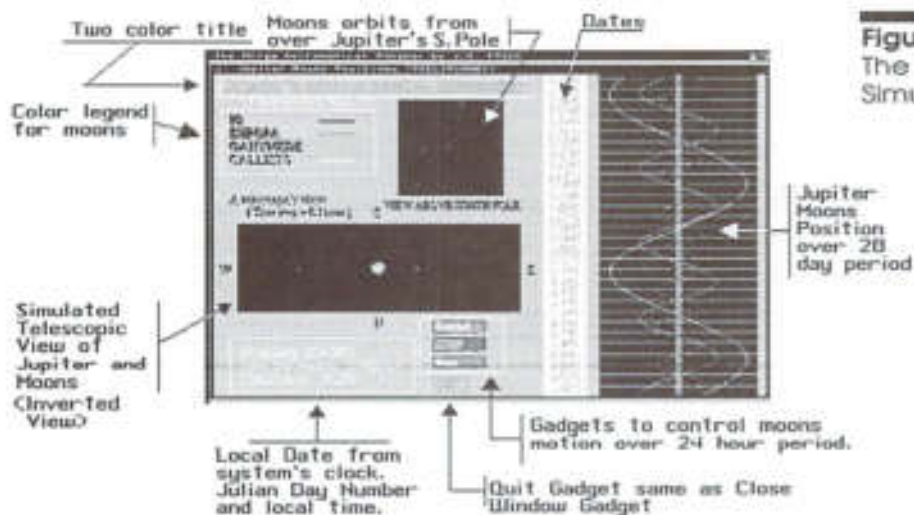


Figure Two
The Jupiters Moons'
Simulation Program

access to the Amiga's system routines with a lot less overhead, and, from an esthetic point of view, the ability to eventually produce code with a less cluttered appearance that sometimes can result by repeated use of the `amiga()` routine.

Some Jupiter Moons' Simulation Program Details

FORTRAN is a high-level applications programming language. It is possible, however, to write FORTRAN code that is capable of interfacing to the various system functions of the Amiga and in some ways assume the role of a system language like C-language. For those who are familiar with the preceding article in this series, it should be clear that FORTRAN 77 language is capable of performing some of the system-level functions that most of us have seen written in C-language.

In this section I will spend some time to describe some of the high-level features of this program. Starting at the beginning of the `JMoGraph5`, for code we see the following line:

```
implicit none
```

This command instructs the compiler to make sure that all variables used are data typed. To gain access to the Amiga's routines files containing relevant information about the systems C-structures, look to constant and routines in the include files. The references to the include files in this example's code are:

```
include ewool.inc
include graph.inc
include intuitl.inc
include deskfmt.inc
```

The system data structures defined in the include files are treated as "templates" — they are used to set up a required structure. After the structure is defined, the contents of the

"templates" arrays are copied to another byte array. The include files "templates" arrays are thus used again and again.

Since a copy must be made of all the structures initialized by the example software, a large number of byte (or integer*1) arrays are defined. Two integer*2 arrays are defined; they will hold gadget button background (`ButtonBack(0:250)`) and a representation

of the disk of Jupiter (`ImageArray(0:51)`).

It was decided that information concerning the "Time Zone" of the user should be passed as arguments to the executable version of the program. This is achieved by using Absoff's `args()` routine. The `args(cmdline)` routine passes to the program user's arguments entered at the CLI. The subroutine's argument `cmdline`, a character string, contains the arguments the user has entered from the CLI. In a more sophisticated application, more code is required to provide better extraction of the contents of `cmdline` than was provided here.

A simple button background (`ButtonBack`) is defined with data to provide a two-color simple button background for the gadgets. This array will be later attached to an Amiga Image structure. The Image structure will be later attached to three separate Gadget structures — one for each gadget.

The program gains access to the system's Graphics and Intuition libraries by calling `GfxBase` and `IntuitionBase` using the function `amiga()`.

Access to the systems Topaz ROM-based font is achieved by setting up a `Text_Attr` structure which is in turn used by `OpenFont()` to open the font. The next part of the code sets up the `NewScreen` structure.

The `NewScreen` structure is defined in order to instruct Intuition to provide this application with a 640 x 400, hi-resolution interlaced screen. To this screen is attached the `Text_attr` structure for the Topaz font. Also, the screen title defined by the string variable `s_title` is attached to the `NewScreen` structure. The function `loc()` is used to return the address of a variable that is given as its argument. The `NewScreen` structure is then used by the `OpenScreen()` routine. If this call is successful, the system returns a pointer to the `Screen` structure.

Early in this description, two arrays that contained the gadgets background and an image of Jupiter's disk were

defined. The contents of these arrays may not be accessible to graphics chip, thus the program must get the information in these arrays into CHIP RAM. Thus the need for the process described in the next paragraph.

Allocation for CHIP RAM is made in order to store the image data for the gadgets and Jupiter's disk model. A call is made to AllocMem() for each image array, providing the size needed and making the provision that it be from CHIP RAM. AllocMem() for each call returns an address, if successful, to a region in the CHIP RAM that has been allocated to these image-defining arrays. The image data is then written to the allocated CHIP RAM. Word(), a FORTRAN provided function, permits the program to read and write two byte chunks of memory at a time. The FORTRAN system also provides byte() and long() functions to allow the programmer to directly read and write memory location in one-byte and four-byte chunks.

A good portion of the code from this point on is spent on setting up the necessary Image, Gadget, Border, and IntuiText structures for graphic items that will appear in the window. A familiarity with Amiga's C structures as defined in the Amiga's ROM Kernel Reference Manual is all that is needed to understand what is shown in the code.

The NewWindow structure is next defined for this application and once defined is used by the OpenWindow() routine to open the window. If successful, a pointer to the newly-defined window structure is passed back to the calling program.

The routines date() and time() are called to get the current date and the time in seconds from midnight as defined by the system's clock. This time is the starting point of the calculation of Jupiter's moons' positions. The current date as provided by the Amiga's system clock requires the users to provide their time_zone as an argument to this program's executable form; i.e., jupmoons time_zone = 8, for Pacific Standard Time zone.

The next section of code generates the graphic the user will see. Most of the window's graphics are generated "on-the-fly." The main title — "Jupiter's Moons' Motion Simulator" is done using 25-point Times font that is normally found on the extras disk and should be in the user's fonts directory. The Times fonts is a disk-based font and the system's OpenDiskFont() is needed to access it. The OpenDiskFont() is accessed using the Absoft's amiga() routine.

The SetFont() routine is used to assign the font to this applications window's RastPort. SetRGB() is called to re-define the contents of the screen's color registers 0 and 2. SetDrMd() is then used to set the drawing mode for the window, in this case "JAM1". The variable "string" is assigned the character string that makes up the title.

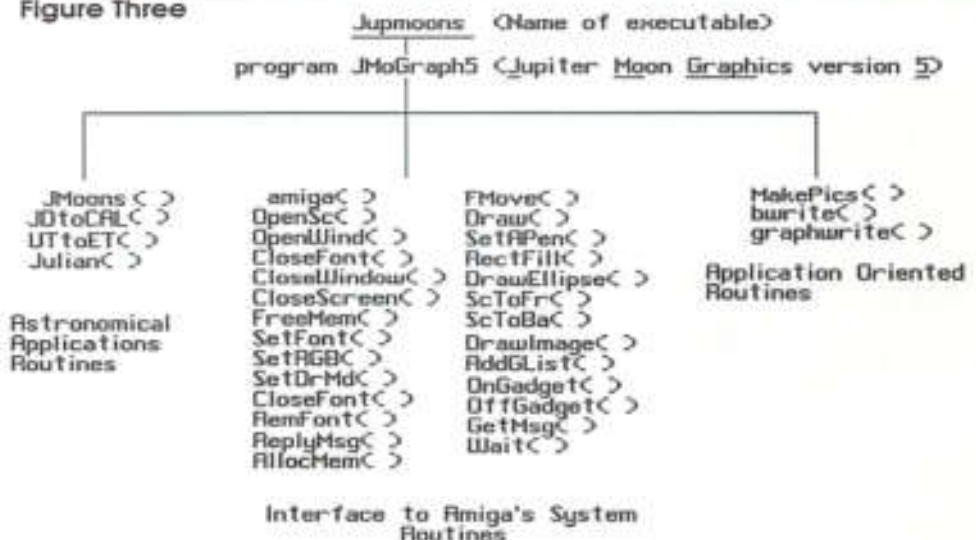
An example of an application-oriented routine is the FORTRAN subroutine called bwrite(). Subroutine bwrite() is a specialized routine that is capable of generating either a "balloon," "outline," or two-color font effect for a given font. The outline of the font would be in one color and the main body in another. Use of the shift argument could give a balloon effect. It is not very sophisticated but does a nice job. The bwrite() code is now given to review:

```
subroutine bwrite (port, string, xloc, yloc, shift,
+ colorback, colorfront)

* port = Current window rast pointer
* string = character string being passed
* xloc, yloc = coordinates of position
* shift = number of shifts the character string is
* spread around
* colorback = color register number for outline color
* colorfront = color register number for main body color

implicit none
include graph.inc
include intui.inc
character*(*) string
integer*4 shift, colorback, colorfront, xloc, yloc,
+ x, yy, n, i, j
n = xloc
yy = yloc
n = len(trim(string))
call SetAPen (long(port),colorback)
do i = -shift, shift
do j = -shift, shift
call Move (long(port), x-1, yy-j)
call Text (long(port), string,n)
enddo
enddo
call SetAPen (long(port),colorfront)
call Move (long(port), xloc, yloc)
call Text (long(port), string,n)
return
end
```

Figure Three

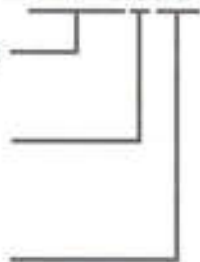


```
integer RectFill
parameter (RectFill = z'0007F233')
```

This field describes ROM Kernel routine's argument sequence.

This field points to the library routine, in this case the library is the graphics library

This field provides the offset from library base to routine.



The font is removed from the window by calling `CloseFont()` and then removed from the system font list by calling `RemFont()`.

The font associated with the window is again changed, this time to 15 pt. Times. More text is generated, but the text generated is accompanied by a shadow effect. This effect is generated using another FORTRAN-based routine — `graphwrite()`. Since `graphwrite()` is a relatively short program, it too will be shown here for the reader's review:

```
subroutine graphwrite (port, string, xloc, yloc,
+   shift, colorback, colorfront)

* This routine generates a shadow-like effect for a
* given font.
* port = window's rasterpointer
* string = character string containing the text to be displayed
* xloc, yloc = window coordinates where the text
*   will be placed (ULHC)
* shift = length of shadow
* colorback = number of color reg. that is used for
*   the shadow
* colorfront = color reg. number that is used for
*   the font.

implicit none

include exec.inc
include graph.inc
include intuit.inc

character*(*) string
integer*4 shift, colorback, colorfront, xloc, yloc,
+   xx, yy, n, i

xx = xloc
yy = yloc

n = len(trim(string))
call SetAPen (long(port),colorback)
do (i = shift,1,-1)
  call FMove (long(port), xx-1, yy-1)
  call Text (long(port),string,n)
enddo
call SetAPen (long(port),colorfront)
call FMove (long(port), xloc, yloc)
call Text (long(port),string,n)

return
end
```

Figure Three
Absoft's Encoding
Scheme
for `amiga()`
arguments.

Examining the `JMcGraph5` routine, the reader can see that repeated use of the above subroutine is done to set text down on the screen.

The user should have noted that some additional routines in some cases have names that are identical to ROM Kernel routines. These assembly language based routines are used to directly call the ROM Kernel routines. `Move()`, `Draw()`, `RectFill()`, `SetAPen()`, `DrawEllipse()` are comparable to the Amiga's Graphics library routines. The argument list for these FORTRAN interface routines are identical to the similarly named versions described in the ROM Kernel Reference Manual.

After generating some more text on the screen, the variable `time_since_midnight`, which stores the time in units of seconds, is changed to units of hours. The correction for ephemeris time is done using the year value (Subroutine `UTtoET()`). Combining the day, with ET corrected time, and `Time_Zone` yields the corrected day (`rday`). Julian day (JD) number is generated from the date and time (subroutine `Julian()`).

Variables are initialized. The range of days (`JD_begin` and `JD_end`) that the moons will be plotted over is also determined. Again, all times are done in terms of the Julian Day number. Graphics depict the moons' motion as viewed from a simulated telescopic view and the motion of the moons as viewed from below the moon's orbital plane (viewing the moons' motion from over Jupiter's south pole). This bit of processing is accomplished with the FORTRAN's while-repeat loop.

The while-repeat loop is used to control the initial graphics animation. The time-increments between each frame drawn is set to one hour. The subroutine `JMoons()` provides positions of Jupiter's moons for a given time — time again is in terms of the Julian Day number.

Let's examine some of the other components of the while-repeat loops. As the moons' trajectories are plotted, the corresponding date and 0 hour line is generated. The subroutine `JDtoCAL()` is used to convert the JD number to a calendar date. The four moons' positions (X, Y, Z) are plotted using the appropriate color, set by calling `SetAPen()` and drawing segments of the trajectories using repeated calls to `Move()` and `Draw()`.

In the area to the left of the moons' trajectories, plots of the moons' positions are shown as viewed from above the planet's south pole (upper) and as viewed through a telescope at low power (lower). The planet's disk is re-drawn each time using the routine `DrawImage()`. The moons are depicted as color-coded 2 x 2 rectangles and are drawn and undrawn using the `RectFill()` routine. This is how the initial display of the program is generated.

After completion of Jupiter moons' 28-day trajectory plot, the window gadgets defined earlier are added to the window with a call to the function `AddGList()`. To activate the added gadget in the window, one performs a call to the system routine `RefreshGadgets()`. Four gadgets should be displayed: "Rewind," "Stop," "Start," and "Quit." Except for the "Stop" gadget, all of the new gadgets are active. The Stop gadget is ghosted and inactive. This is achieved by calling the system routine `OffGadget()`. Another defined gadget coincides with the area of the moons trajectory plots.

The do-while loop that follows contains all of the interface code to the user's gadget selections. Since the window has a close gadget - class = `CLOSEWINDOW`, a click there is indicative that the user desires to close it all down. A message with a message class corresponding to `GADGETDOWN` starts the processing to identify the selected gadget by means of its assigned `GadgetID`.

For instance, the large gadget that corresponds to most of the area of the moons' trajectory plot, if selected, generates a `GadgetID = 1`. If this gadget is selected, the mouse position (`mx,my`) is extracted from the window structure. From the "my" component of the mouse-pointer position, the date and time are computed. A call to `MakePics()` is done to update the moon positions in the left two displays. The "Stop," "Rewind," and "Start" gadget appearance are now adjusted accordingly — "Rewind" and "Start" are active and "Rewind" is inactive. This is signified by the gadgets' ghosted appearance.

The "Rewind" gadget has a `GadgetID = 2`. If the "Rewind" gadget is selected, the current Julian Day number is initialized to 0 hour UT for that day. Each additional click on this gadget backs up the JD number by 24 hours. The `MakePics()` routine is called again to modify the views in the left two windows.

If the "Start" gadget (`GadgetID = 4`) is selected, a sequence of images is generated to show the moons' positions for a 24-hour period. Each image is formed at 0.1 hour intervals — a total of 240 images. Thus we have an animation of the moons. During the period of this animation, the "Rewind" and "Start" are inactive (gadgets are ghosted) and the "Stop" gadget is made active.

If the "Stop" gadget (`GadgetID = 3`) is selected, the animation is stopped. The "Start" and "Rewind" gadgets are again made active and the "Stop" gadget is made inactive.

If the user does not elect to select the "Stop" gadget during the animation and allows it to finish, the control gadgets are again made active and inactive as needed. A code segment from the example shows a little of what is being done here:

```

else if ( GadgetID = 4 ) then !User selected Start gadget?
call OnGadget ( loc(Gadget_Stop), Window, 0)
call OffGadget ( loc(Gadget_Rewind), Window, 0)
call OffGadget ( loc(Gadget_Start), Window, 0)

! show moons positions over 24 hour period starting within
! date resolved from trajectory plot. Time step = 0.1 hrs.

do ( i = 1, 240 )
call MakePics (port, ImagePtr, JD, X, Y, Z, OldXK,
OldY, OldZ)
message = GetMsg (long(Window+wd_UserPort))
if ( message <= 0 ) then ! User has selected a gadget
call OnGadget (loc(Gadget_Rewind), Window, 0)
call OnGadget (loc(Gadget_Start), Window, 0)
call OffGadget (loc(Gadget_Stop), Window, 0)
class = long(message+in_Class)
code = word(message + in_Code)
GadgetPtr = long(message + in_Address)
GadgetID = word(GadgetPtr + 32)
call ReplyMsg (message)
if (class = GADGETDOWN) then ! User hit STOP gadget?
if ( GadgetID = 3 ) exit ! if so exit do-loop
endif
endif
JD = JD + 0.00416667 ! increment by an 1/10 of hour
enddo
call OnGadget ( loc(Gadget_Rewind), Window, 0) ! Enable and
call OnGadget ( loc(Gadget_Start), Window, 0) ! disable
call OffGadget ( loc(Gadget_Stop), Window, 0) ! gadgets endif

```

If the "Quit" or "CloseWindow" gadgets is selected, the do-while loops is exited by toggling the logical variable "Waiton" from `TRUE` to `FALSE`. The calls to `CloseWindow()` and `CloseScreen()` close the window and screen respectively. Calls to the `FreeMem()` routine returns the earlier allocated memory back to the system. The fonts are removed from the system font list using a call to the `RemFont()` routines. This concludes the description of the main program.

A Modified Interface to the Amiga ROM Kernel routines

As alluded to earlier and as the reader should have observed while looking at the example code, some of the calls to the system routines come in one of two ways — the more direct means, call `Draw(port, x, y)` or the indirect way, call `amiga(Draw, port, x, y)`. The `amiga()` routine was provided by the maker of this FORTRAN compiler — Absoft.



The `amiga()` routine is a really nice bit of coding — machine coding, that is. The routine essentially takes care of interfacing the FORTRAN to the Amiga's ROM Kernel. The Absoft include-files provides descriptions of structures and system constants, and takes ROM Kernel routine names and assigns each one with a unique number. Each include-file contains something like the following list:

```
integer CBump      ;parameter (CBump =z'00009230')
integer InitView   ;parameter (InitView =z'0000923C')
integer SetDrMd    ;parameter (SetDrMd =z'0021923h')
integer SetBPen    ;parameter (SetBPen =z'0021923A')
integer SetAPen    ;parameter (SetAPen =z'0021923B')
integer PolyDraw   ;parameter (PolyDraw =z'02219238')
integer Flood      ;parameter (Flood =z'8C259237')
integer WritePixel ;parameter (WritePixel =z'04619236')
integer ReadPixel  ;parameter (ReadPixel =z'04619235')
integer BltPattern ;parameter (BltPattern = z'0009F234')
integer RectFill   ;parameter (RectFill =z'0007F233')
```

Let's examine a single line from this list. On the left side of the list, we see variable names data typed as integers (32 bit). To the right of each typed variable appears a parameter statement in which the typed variable to the immediate left is defined with some hexadecimal number.

The listed variable name corresponds to the name of a ROM Kernel routine. The value assigned to it by the parameter statement is an encoding of everything that is needed by the `amiga()` routine in order for it to interface correctly with ROM Kernel routine (see Figure 3 for a closer look at the meaning of the numbers assigned by the parameter statement).

Each time a ROM Kernel routine is called using the `amiga()` routine, a significant number of operations is required before the actual `amiga()` routine is called. This approach is a reasonable one if the call is made infrequently, such as to `OpenScreen()`. Often, the need is for calling a ROM Kernel routine repeatedly at a high rate, say while performing certain graphic functions. In the example discussed above, calls to the ROM Kernel graphic routines `RectFill()` and `DrawImage()` are made very often in some parts of the main program and again in the `MakePic()` subroutine. In these cases a more direct approach using FORTRAN's capability to access machine language subroutines and functions is better.

In the example, a number of ROM Kernel routines were called in the more direct manner. However, few of the calls were accomplished using Absoft's `amiga()` routine.

"So just what do these nifty, more direct calls procedure look like?" you ask. Well, here are some examples (more are located in the file `F77Interface.asm`). The first example shows the `RectFill()` routine which interfaces to the similarly named ROM Kernel Routine.

```
* FORTRAN calling sequence
* call RectFill (rp, x, y, noise, ysize)
*
* assembly language source for a FORTRAN callable routine
* that calls the Amiga's graphic RectFill() routine
```

```
_RectFill equ -306

RectFill: movem.l a0-a2/a6, -(a7)
movem.l 4(a0), a6
movem.l 36(a7), a2 ; load BP pointer
movem.l 32(a7), a1 ; x position in RastPort
move.l (a1), d0
movem.l 28(a7), a1 ; y position in RastPort
move.l (a1), d1
movem.l 24(a7), a1 ; x LRAMC
move.l (a1), d2
movem.l 20(a7), a1 ; y LRAMC
move.l (a1), d3
move.l (a2), a1

;sr _RectFill(a6)

movem.l (a7)+, a0-a2/a6
rts
end
```

`Rectfill()` is a graphic library routine. Absoft's FORTRAN77 when-if calls a subroutine first, puts the program counter on the stack followed by the first argument — in this case the `rp` (`RastPort`) — then followed by the coordinate pairs of the Upper Left Hand Corner and Lower Right Hand Corner of the rectangle object desired. When the `RectFill()` routine is finally entered, some of the registers used by this routine are also placed on the stack. The next instruction sets up register `a6` with the appropriate library base address. The remaining steps access the contents of the stack to get the addresses of the various parameters passed through the calling sequence. The locations at these addresses are accessed and the contents are transferred to the appropriate register as required. Once the last step is finished, a call to the ROM Kernel routine is done. Before the return from this routine, the content of the registers being saved on the stack are restored.

Another example shows the assembly language version of the routine used to direct access the ROM Kernel's `DrawImage()` routine.

```
* FORTRAN calling sequence
* call DrawImage(rp, Image, LeftOffset, TopOffset)
*
*
* assembly language source for a FORTRAN callable routine
* that calls the Amiga's Intuition's DrawImage routine
```

```
_DrawImage equ -114

DrawImage: movem.l a0-a3/a6, -(a7)

movem.l 8(a0), a6 ; Set up base for Intuition Library
movem.l 36(a7), a2 ; load BP pointer
```


end

```

read readlin ("call. 13") Time_Sec_Sec, Time_Sec_Minute and Line
? Information
Write ("P,(A,150") Time_Sec_Sec, Time_Sec

nne = 1

do ii = 1, 130 ? Set-up Button Background
ButtonBack((i-1)*4) = 'fffff'
ButtonBack((i-1)*4+1) = 'fffff'
ButtonBack((i-1)*4+2) = 'fffff'
ButtonBack((i-1)*4+3) = 'fffff'
enddo
ButtonBack(60)=0; ButtonBack(61)=0; ButtonBack(62)=0
ButtonBack(63)=0
do ii = 17, 200
ButtonBack((i-1)*4) = 'fffff'
ButtonBack((i-1)*4+1) = 'fffff'
ButtonBack((i-1)*4+2) = 'fffff'
ButtonBack((i-1)*4+3) = 'fffff'
enddo
ButtonBack(116)=0; ButtonBack(117)=0; ButtonBack(118)=0
ButtonBack(119)=0
do ii = 31, 600
ButtonBack((i-1)*4) = 0
ButtonBack((i-1)*4+1) = 0
ButtonBack((i-1)*4+2) = 0
ButtonBack((i-1)*4+3) = 0
enddo

IDstring = 'Program by: J. A. PAREK, 3210 S. Diamond St.,'
IDstring = IDstring// 'Santa Ana, CA 92704'

i = anigs(00base) ? open "graphics.library"
if (i=0) stop "'00base' failed"

i = anigs(00base)
if (i=0) stop "'00base' failed"

* = set up the screen text style

f_name = "topaz.font"//char(0)

ta_name = loc(f_name)
ta_style = 0
ta_weight = FE_NORMAL
ta_flags = FE_NORMAL

font = anigs (OpenFont.TextAttr)

if (font = 0) stop "Woop, Topaz font not available."

topazfont = font

* = set up the NewScreen data block and allocate the screen

s_title = "The Aniga Astronomical Almanac by J.A. PAREK"//char(0)

ss_LeftEdge = 0
ss_TopEdge = 0
ss_Width = 540
ss_Height = 400
ss_BackfillPen = 0
ss_BlockPen = 0
ss_ViewWindow = NINEF wrt. LACE
ss_Type = CUSTOMSCREEN
ss_Font = loc(TextAttr)
ss_TextTitle = loc(s_title)
ss_Jedgets = 0
ss_CurtBkMap = 0

Screen = OpenSc ( NewScreen) ? Open Screen
if (Screen=0) stop "'OpenScreen' failed"

ta_style = FE_BOLD ? Define Font

do ( i = 1, 8 ) ? Copy TextAttr
TopazFontAttr(i) = TextAttr(i)
enddo

* ? call SetBk (Screen) ? Place Screen to back of rest ScreenFndam

size = 500 ? Allocate CHIP RAM for Jupiter Image
flags = MEMF_CHIP
ImageBlock = Allocate (size, flags)
if ( ImageBlock = 0 ) stop "Unable to Allocate memory"

size = 400 ? Allocate CHIP RAM for Saturn Backgrounds
ImageBlock2 = Allocate (size, flags)
if ( ImageBlock2 = 0 ) stop "Unable to allocate memory"

do ( i = 0, 31 ) ? write Image data to chip ram
write(ImageBlock+2*i) = ImageArray(i)
enddo

```

```

40  i = i + 0.25f;
void ImageBack(i=1) + ButtonBack(i)
endif

c - set-up image structure for Rewind, Stop, Start buttons

ig_LeftEdge = 0
ig_TopEdge = 0
ig_Width = 40
ig_Height = 15
ig_Depth = 4
ig_ImageData = ImageBackDr
ig_FlagsWrk = 1f
ig_FlagsWrkOff = 0
ig_VertImage = 0

do ( i = 1, 30 )           ! Copy Image structure template to ImageBack
    ImageBack(i) = Image(i)
endif

* - set-up the mouse's trajectory window gadget structures.
* The whole area is one big boolean gadget

gg_StartGadget = 0
gg_LeftEdge = 444
gg_TopEdge = 1
gg_Width = 150
gg_Height = 166
gg_Flags = GADGEXOR
gg_Activation = RELVERIFY .or. GADIMMEDIATE
gg_GadgetType = BOULGADGET
gg_SelectHeader = 0
gg_SelectHeader = 0
gg_ActualInclude = 0
gg_GadgetText = 0
gg_SpecialInfo = 0
gg_GadgetID = 1

do ( i = 1, 44 )           ! Copy Gadget structure template to Chart
    Chart_Gadget(i) = Gadget(i)
end do

* - Set-up another Close or Quit Gadget in the Window

bd_LeftEdge=1; bd_TopEdge=1           ! Quit Gadget's Border structure
bd_FrontPen=0; bd_BackPen=0
bd_DrawMode=JML; bd_Count=0
bd_XY = loc(BorderVectorsQ)
bd_SelectHeader = 0

do ( i = 1, 16 )           ! Make copy of Border structure
    Border(i) = Border(i)
end do

* - Set-up gadget to Rewind to 0 hours 00

RewdString = "Rewind"/char(0)           ! Rewind gadget's InitText structure

it_FrontPen = 7; it_BackPen = 0; it_DrawMode=JML
it_LeftEdge = 3; it_TopEdge = 1
it_FrontFont = loc(TopTextFontStr)
it_Text = loc(RewdString)
it_BackText = 0

do ( i = 1, 20 )           ! Copy InitText structure
    RewText(i) = InitText(i)
end do

gg_WndGadget = loc(Chart_Gadget)           ! All Gadgets are linked together
gg_LeftEdge = 150; gg_TopEdge = 280 ! This is the Rewind Gadget structure
gg_Width = 60; gg_Height = 15
gg_Flags = GADGEXOR .or. GADGMAXI
gg_Activation = RELVERIFY .or. GADIMMEDIATE
gg_GadgetType = BOULGADGET; gg_GadgetHeader = loc(ImageBack)
gg_SelectHeader = 0; gg_GadgetText = loc(RewText)
gg_ActualInclude = 0; gg_SpecialInfo = 0
gg_GadgetID = 1

do ( i = 1, 44 )           ! Make copy of gadget structure
    Gadget_Rewind(i) = Gadget(i)
end do

* - Set-up gadget Stop

StopString = "STOP"/char(0)           ! Following is InitText structure
! for Stop gadget

it_FrontPen = 7; it_BackPen = 0; it_DrawMode=JML
it_LeftEdge = 3; it_TopEdge = 1
it_FrontFont = loc(TopTextFontStr)
it_Text = loc(StopString)
it_BackText = 0

do ( i = 1, 20 )           ! Copy InitText structure
    StopText(i) = InitText(i)
end do

```



```

end do

gg_NextGadget = loc(Gadget_BoxWin) ! This is the final gadget structure
gg_LeftEdge = 250; gg_TopEdge = 300
gg_Width = 40; gg_Height = 15
gg_Flags = GADGEMENU .or. GADGIMMEDIATE
gg_Activation = HELPERIFF .or. GADGIMMEDIATE
gg_GadgetType = BOULGADGET; gg_GadgetHeader = loc(ImageBack)
gg_SelectHeader = 0; gg_GadgetText = loc(StartTest)
gg_Miscellaneous = 0; gg_SpecialInfo = 0
gg_GadgetID = 3

do i = 1, 44 ! Make copy of gadget structure
  Gadget_Step(i) = Gadget(i)
end do

* - Set-up gadget Start

StartString = "Start"//char(0) ! Start gadget's intaitest structure

it_FrontPen = 1; it_BackPen = 0; it_FrontMode=JAG
it_LeftEdge = 3; it_TopEdge = 1
it_CTextFont = loc(TopatTestAttr)
it_CText = loc(StartString)
it_NextTest = 0

do i = 1, 20 ! Make copy of gadget intaitest
  StartTest(i) = StartTest(i)
end do

gg_NextGadget = loc(Gadget_Flags) ! Start gadget's structure
gg_LeftEdge = 250; gg_TopEdge = 225
gg_Width = 40; gg_Height = 15
gg_Flags = GADGEMENU .or. GADGIMMEDIATE
gg_Activation = HELPERIFF .or. GADGIMMEDIATE
gg_GadgetType = BOULGADGET; gg_GadgetHeader = loc(ImageBack)
gg_SelectHeader = 0; gg_GadgetText = loc(StartTest)
gg_Miscellaneous = 0; gg_SpecialInfo = 0
gg_GadgetID = 4

do i = 1, 44 ! Make copy gadget structure
  Gadget_Start(i) = Gadget(i)
end do

QuitString = "Quit"//char(0) ! Quit gadget intaitest structure

it_FrontPen = 3; it_BackPen=0; it_FrontMode=JAG
it_LeftEdge = 1; it_TopEdge = 1
it_CTextFont = loc(TopatTestAttr)
it_CText = loc(QuitString)
it_NextTest = 0

do i = 1, 20 ! Make copy of intaitest structure
  Quit(i) = Quit(i)
end do

gg_NextGadget = loc(Gadget_Start) ! Quit gadget's structure
gg_LeftEdge = 250
gg_TopEdge = 345
gg_Width = 40
gg_Height = 15
gg_Flags = GADGEMENU
gg_Activation = HELPERIFF .or. GADGIMMEDIATE
gg_GadgetType = BOULGADGET
gg_GadgetHeader = loc(Header)
gg_SelectHeader = 0
gg_GadgetText = loc(Quit)
gg_Miscellaneous = 0
gg_SpecialInfo = 0
gg_GadgetID = 5

do i = 1, 44 ! Make copy of gadget structure
  Gadget_Quit(i) = Gadget(i)
end do

* - set up the NewWindow data block

w_title = "Jupiter Moon Position (PMLECHUNK)"// char(0)

nw_LeftEdge = 1
nw_TopEdge = 15
nw_Width = 400
nw_Height = 360
nw_DetailPen = 0
nw_BackPen = 1
nw_Title = loc(w_title)
nw_Flags = WINDOWCLOSE .or. SHUT_SCREEN .or. ACTIVATE
      .or. SHUTWINDOW
nw_ICOFlags = CLOSEWINDOW .or. GADGEMENU
nw_Type = CONTINUOUS
nw_FirstGdgt = 0 ! loc(Chart_Gadget)
nw_ChartMark = 0
nw_Screen = Screen
nw_RotMag = 0
nw_MinWidth = 100
nw_MinHeight = 25
nw_MaxWidth = 640

```

```

nw_MaxHeight = 360

Window = OpenWind (NewWindow) ! Open Window on custom screen
if (Window=0) then
  call CloseScreen (Screen)
  stop "'OpenWindow' failed"
endif

port = Window+nd_HPort ! Set pointer to window HeaderPort

* - set-up image structure

ig_LeftEdge = 0 ! Jupiter Image - Image structure
ig_TopEdge = 0
ig_Width = 18
ig_Height = 13
ig_Depth = 4
ig_ImageData = ImageBlock
ig_PlanePick = 'r'f'
ig_PlaneCoeff = 0
ig_NextImage = 0

ImagePtr = loc(Image) ! Set pointer to Jupiter Image structure

call date (month, day, year) ! Get current date
call time (time_place, midnight) ! Get current time from system

tstep = 1
GdStep = 5

* - set up the text style

* call CloseFont (font)

f_name = "Times.font"//char(0) ! Define Text/Font Attribute structure

ta_Name = loc(f_name)
ta_Size = 24
ta_Style = FB_BOLD
ta_Flags = FB_CLOSEFONT

font = assign (OpenDiskFont, TextAttr) ! Open a disk based font

if (font = 0) then ! If unable to open disk font exit
  call CloseWindow (Window)
  call CloseScreen (Screen)
  call FreeMem (ImageBlock, size)
  stop "Stop, Times 24 not available."
endif

fontID = font

call SetFont (longiport, font) ! Assigns font to window

call SetRGB (Screen+var_ViewPort,0,16,16,16) ! Re-define color register 0
call SetRGB (Screen+var_ViewPort,1,1,1,1) ! " " " " " 1

call SetRGB (longiport, JAG) ! Set Drawing Mode
string = "Jupiter's Moon Motion Simulator"
call Write (font, string, 11, 11, 1, 1, 1) ! Routine generates two
! color font

* - set up the text style

call CloseFont (font) ! Close Times 24 font
call ReadFont (font) ! Removes Times 24 font from font list

f_name = "Times.font"//char(0) ! Set-up Text Attr. Struct for
! Times 24 font
ta_Name = loc(f_name)
ta_Size = 15
ta_Style = FB_NORMAL
ta_Flags = FB_CLOSEFONT

font = assign (OpenDiskFont, TextAttr) ! Open disk based font

if (font = 0) then ! If not available clean-up, exit
  call CloseWindow (Window)
  call CloseScreen (Screen)
  call FreeMem (ImageBlock, size)
  stop "Stop, Times 15 font not available."
endif

fontID = font

call SetFont (longiport, font) ! Assigns font to window

string = "Dates are for 9 Hours UT."
call graphwrite (port, string, 40, 34, 0, 1, 7) ! Routine generates
! shadow-like font
string = "A telescope's view"
call graphwrite (port, string, 20, 150, 3, 1, 7)

string = "( Time step = 1 hour )"
call graphwrite (port, string, 40, 164, 3, 1, 7)

call RectFill (longiport, 512, 3, 512, 360) ! Interface to system
! RectFill() routine

```



```

locm = 185-int(8*X(1))
lomy = 225-int(8*Y(1))
call RectFill (long(pict), locm, lomy, locm+1, lomy+1)
endif
OldX(1) = X(1)
OldY(1) = Y(1)
OldZ(1) = Z(1)
call SetApen (long(pict),1*3)
locm = 185 - int(8*X(1))
locr = 83 - int(8*Y(1))
call RectFill (long(pict), locm, locr, locm+1, locr+1)
repeat

* After moon trajectories are plotted, the window's gadget list
* is assigned to window by calling system's ADDLIST() and made
* viewable by calling system's RefreshGadgets()

gad_position = AddList (Window, loc(Gadget_Quit), 0, -1, 0)
call erase (RefreshGadgets, loc(Gadget_Quit), Window, 0)

call SetApen (long(pict), 0)
call RectFill (long(pict), 30, 152, 179, 166)
string = "Time step = 0.1 hour"
call graphwrite (pict, string, 40, 164, 3, 1, 7)

string = "
oldstring = string

* The following line of code disables the STOP gadget

call OffGadget (loc(Gadget_Stop), Window, 0)

* The following code segment allows for and responds to user inputs
* as defined by the gadgets user actions.

Waiton = .TRUE.

do while (Waiton)
  message = GetMsg (long(Window+ed_UserPort)) ! DCMG message available
  if (message = 0) then ! No message, go to sleep.
    itemp = byte(long(Window+ed_UserPort) + NO_SLEEP)
    result = shift (time, itemp)
    call Wait (result)
  else
    class = long(message-in_Class) ! There is a message, get class
    code = word(message-in_Code) ! get code
    GadgetPtr = long(message-in_InAddress) ! Get GadgetPtr
    GadgetID = word(GadgetPtr + 3) ! Get program defined ID
    call ReplyMsg (message) ! Tell system we got message
    if (class = CLASSMENU) then ! It tries close window gadget selected
      Waiton = .FALSE. ! exit from while loop
    end if
    if (class = GADGETDOWN) then ! User defined gadget selected?
      if (GadgetID = 1) then ! Yes, is it from moon traj. plot?
        mx = word(Window+ed_MouseX) ! Yes, get mouse click position
        my = word(Window+ed_MouseY)
        JD = JD begin + (my-16)/12.0 ! Resolve from position the date and time
        call MakePics (pict, ImagePtr, JD, X, Y, Z, OldX,
          OldY, OldZ) ! Make new pictures reflecting corresponding
          moon positions for date and time
        call OffGadget (loc(Gadget_Stop), Window, 0) ! Enable and disable gadgets
        call OnGadget (loc(Gadget_Quit), Window, 0)
        call OnGadget (loc(Gadget_Start), Window, 0)
        else if (GadgetID = 2) then ! User selected QUIT gadget?
          Waiton = .FALSE.
        else if (GadgetID = 3) then ! User selected back to 0 h UT
          if (JD - airt(JD) > 0.5) then
            JD = airt(JD) + 0.5
          else
            JD = airt(JD) - 0.5
          endif
          call MakePics (pict, ImagePtr, JD, X, Y, Z, OldX,
            OldY, OldZ) ! Make new pictures for 0 h of given date
        else if (GadgetID = 4) then ! User selected Start gadget?
          call OnGadget (loc(Gadget_Stop), Window, 0)
          call OffGadget (loc(Gadget_Back), Window, 0)
          call OffGadget (loc(Gadget_Start), Window, 0)
        end if
      show moon positions over 24 hour period starting within
      date resolved from trajectory plot. Time step = 0.1 hours.

      do (i = 1, 140)
        call MakePics (pict, ImagePtr, JD, X, Y, Z, OldX,
          OldY, OldZ)
      *
      message = GetMsg (long(Window+ed_UserPort))
      if (message <> 0) then ! User has selected a gadget
        call OnGadget (loc(Gadget_Back), Window, 0)
        call OnGadget (loc(Gadget_Start), Window, 0)
        call OffGadget (loc(Gadget_Stop), Window, 0)
        class = long(message-in_Class)
        code = word(message-in_Code)
        GadgetPtr = long(message-in_InAddress)
        GadgetID = word(GadgetPtr + 3)
        call ReplyMsg (message)
        if (class = GADGETDOWN) then ! User selected STOP gadget?

```

```

if (GadgetID = 3) then ! if we wait for loop
  endif
endif
JD = JD + 0.00416667 ! increment by an 1/10 of hour
endif
call OnGadget (loc(Gadget_Back), Window, 0) ! Enable and disable gadgets
call OnGadget (loc(Gadget_Start), Window, 0)
call OffGadget (loc(Gadget_Stop), Window, 0)
endif
endif
repeat

* We get here if user selects either QUIT gadget or window's close gadget

call CloseWindow (Window)
call CloseScreen (Screen)
call FreeMem (ImageBlock, size)
call FreeMem (ImageBackBt, size)
call CloseFont (font)
call RectFill (font)

* - Force all the timer fonts opened by program if system fonts list
* mostly for Version 1.32 of the Amiga OS
call RectFill (font1)
call RectFill (font17)
call RectFill (font20)

stop
end

* following subroutine is used to draw Jupiter and moon configurations
* pictures.

subroutine MakePics (pict, ImagePtr, JD, X, Y, Z, OldX,
  OldY, OldZ)
  implicit none
  Again from us to data type all variables

  real*8 JD
  real*4 rday, temp
  integer*4 month, year, oldlsec, oldlsecy, locm, lomy, ImagePtr
  integer*4 locs, oldlsec, i, port
  real*4 uc(4), OldX(4), OldY(4), OldZ(4), X(4), Y(4), Z(4)
  character*20 oldstring, string

  * the following case is done to preserve variables till next call to this routine

  save oldstring, string

  call JDToCAL (JD, rday, month, year)
  call graphwrite (pict, oldstring, 30, 160, 0, 1, 1)
  write (oldstring, "(L,/,12,/,12,/,12)"/ month, int(rday),
    year-1900)
  call graphwrite (pict, oldstring, 30, 160, 0, 1, 1)
  temp = (day - int(day))*14.0
  call graphwrite (pict, string, 40, 160, 0, 1, 1)
  write (string, "(G,1)"/ temp
  string = "Time (UT) "/temp
  call graphwrite (pict, string, 40, 160, 0, 1, 1)
  call JDToCAL (JD, X, Y, Z, old)
  call DrawMoon (long(pict), ImagePtr, 179, 164)
  do (i = 1, 140)
    if (.not. (abs(OldX(i)) < 1.0) .and.
      (OldX(i) < 0.0)) then
      call SetApen (long(pict), 1)
      oldlsec = 185 - int(8*OldX(i))
      oldlsecy = 225 - int(8*OldY(i))
      call RectFill (long(pict), oldlsec, oldlsecy, oldlsec+1,
        oldlsecy+1)
      endif
      call SetApen (long(pict), 1)
      oldlsec = 185 - int(8*OldX(i))
      oldlsecy = 83 - int(8*OldY(i))
      call RectFill (long(pict), oldlsec, oldlsecy, oldlsec+1,
        oldlsecy+1)
    end if
    if (.not. (abs(X(i)) < 1.0) .and. (X(i) > 0.0))
      * then
      call SetApen (long(pict), 1*3)
      locm = 185 - int(8*X(i))
      lomy = 225 - int(8*Y(i))
      call RectFill (long(pict), locm, lomy, locm+1, lomy+1)
    endif
  enddo

  return
end

```



Toolbox— Amiga Programming Solutions

An Introduction to 3-D Programming

by Patrick Quaid

Raisin Debt

Suppose you're writing a program that, say, organizes your soup cans. You've got your Intuition interface, your file requesters, your AReXX commands, but what you don't have is a routine to alphabetically sort the cans. So what do you do, sit down and write one from scratch? You could — it's not all that hard. For all I know, you might write the greatest sort algorithm known to mankind. On the other hand, hundreds of programmers have spent decades developing and refining sort routines to the point that any further improvement, while possible, is bloody unlikely. Put more positively, your efforts would be more profitably spent directed toward other areas of your program.

This idea is often summarized by the aphorism "Why reinvent the wheel?" The answer, of course, is that if no one had done so, they'd still be made of stone. So although you are unlikely to come up with a better sort routine, other algorithms are much closer to the stone wheel stage, and, of course, the majority fall somewhere in between.

The point of all this is that you should spend your time creating new stuff, rather than recreating old stuff. Fine, but in order to do that you need to know the old stuff. Well, that's where this column comes in. In each issue, we'll go over a different area of programming, looking at some existing solutions and discussing their pros and cons. In this first column, for example, we take a look at 3-D drawing techniques. Later we'll go over multitasking — how to do it, when to do it, what to wear. We'll investigate random numbers and graph traversals, and maybe even discuss some of those finely-tuned sort routines.

Since few problems have one "best" solution, we'll normally look at several different approaches. Each algorithm will be illustrated by a complete, working example program. These examples will be written specifically for the Amiga, but in many cases the underlying ideas apply to computers in general. The programs will be written in C, Modula-2, Assembly, AReXX — whatever language best suits the job at hand.

The goal of the examples, and the column as a whole, is to give you ideas, algorithms, and even complete subroutines that you can include in your code. I hope they'll provide a map through the known, so you can begin blazing trails through the unknown.

Perspective and Depth

The world of computer graphics ranges from simple wire-frame outlines to full photo-realistic rendering. This article will cover the entire spectrum in excruciating detail. Hey — just kidding — but we will introduce the wire-frame part, and later we'll expand to a flexible, filled-surface drawing system.

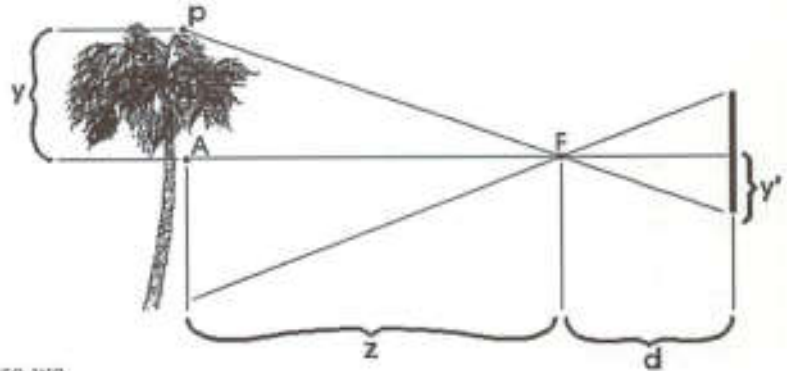
But first some fundamentals. The equations governing perspective graphics are easily derived from high school geometry and a passing familiarity with photography. Take a look at the "pinhole camera" model shown in Figure 1. You can see that we have an object to look at (a palm tree, in this case), the camera's pinhole or focal point (labelled F), and the two dimensional image, which in our case will be the screen. Friendly little photons shoot from the top of the tree, through the pinhole, to leave their marks on the bottom of the screen.

The image on the screen ends up upside down, but that's no big deal — we could, after all, just define our objects backward. Instead we'll get rid of the problem by flipping the screen to the same side of F as the object, as shown in Figure 2. This setup more clearly shows the derivations of the equations we'll need.

That queasy feeling in your stomach signals geometry, dead ahead. First of all, let's define what we want out of all this, to wit: given a point P specified by the 3-D coordinates (x, y, z), where on the screen show we draw its image P'? Consider the right triangle formed by the points F, A, and P. Done? OK, now look at the triangle formed by the points F, C, and P'. Prove that these triangles are similar (answer: they look a lot alike. QED). Since they are in fact similar, the ratio of y' to y is the same as that of d to z. Algebraically that looks like $y'/y = d/z$, which when solved

Figure One

The Basic
Pin-hole
Camera
Model



for y' becomes: $y' = yd/z$. We're not quite done, because we actually wanted the position of P' . To get that, we take what we just got and add it to the y component of C, the center point of the screen. You can relax — the derivations are over for now.

If you tilt your head to the side and read the previous paragraph again, you'll see that you can use the same equations to figure out the x component of P' . Thus for any point in space, the associated screen position is given as follows:

$$(x', y') = (Cx + xd/z, Cy + yd/z)$$

The constant d in this equation defines the distance between the focal point F and the screen. In Figure 2 you can see that the angle v , the field of view, can be defined in terms of d, and vice versa. Small values for d result in fish-bowl effects, like a wide-angle camera lens, while larger values tend to flatten out distances like a telephoto lens.

Since straight lines in space become straight lines on the screen, perspective drawing reduces to converting 3-D points in space to 2-D screen coordinates, then drawing lines between all the endpoints. It's all so incredibly simple.

Well, maybe not. For one thing, we have to be sure z doesn't stray too close to zero or the x' and y' values will blow up. When z is negative, we have another embarrassing problem: the points still show up on the screen, turned inside out. That's all easy enough to handle, I guess. The real problem is the load of limitations this scheme imposes — it makes you stick your camera on a tripod and won't let you move it. But we're dynamic, '90's sort of people, so we don't like limitations. We want to be able to move the camera anywhere, point it at anything, and tilt it to any fashionable angle. We want to set the focal length of lens, and we want to take pictures of gorgeous babes. I can handle all but one of those — if you can help on the other, let me know. In any case, we won't address these issues until next time.

For this column we'll work within our limits to develop a very simple set of graphic display routines. We'll open a window, define some default values, and draw... what? Well, to add some measure of flexibility to this first program, we'll read an object definition from a text file. Let's take a deeper look at it.

First of all, why a text file? It's going to be much bigger and slower than an equivalent binary file, but it has two important advantages. For one thing, it's easy to see what's going on, which is the point of this article after all, and for another you won't need a special program just to create the data file. If we were going to develop a commercial program, we would of course invent our own proprietary, encoded, impossible-to-modify file format in order to feel like professionals.

Example object descriptions are offered in Listing #2 (entitled "Two Boxes") and Listing #3 ("Self Portrait"). The first line is a constant z offset, which will be added to each of the points in the object. This allows us to define data points around the origin, which I find easiest, then push them all far enough away to see.

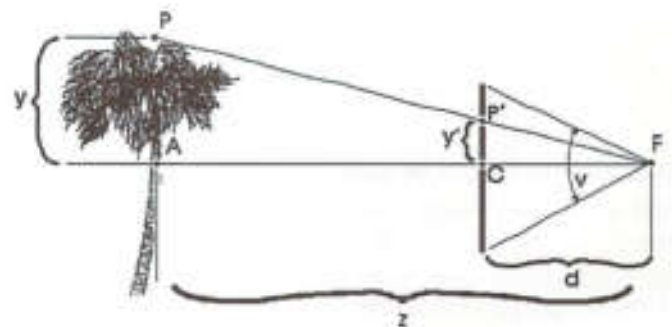


Figure Two The Model Used to Derive Our Equations

Listing Two Two Boxes

```

80
22

 70 100 -30
110 100 -20
120 90 -10

 30 100 -30
-10 100 -20
-20 90 -10

 30 95 -30
 50 20 -60
 30 30 -30
 70 30 -30

 10 -40 -10
 25 -50 -20
 75 -50 -20
 90 -40 -10

 80 -100 0
 20 -100 0
-30 -50 0
-30 150 0
 20 200 0
 80 200 0
130 150 0
130 -50 0

18

0 1
1 2

3 4
4 5

6 7
7 8
7 9

10 11
11 12
12 13

14 15
15 16
16 17
17 18
18 19
19 20
20 21
21 14

```

Listing Three Self Portrait

```

200

16

-100 -100 -100
 100 -100 -100
 100 100 -100
-100 100 -100

-100 -100 100
 100 -100 100
 100 100 100
-100 100 100

-400 -400 -100
-200 -400 -100
-200 -200 -100
-400 -200 -100

-400 -400 100
-200 -400 100
-200 -200 100
-400 -200 100

24

0 1
1 2
2 3
3 0

4 5
5 6
6 7
7 4

0 4
1 5
2 6
3 7

8 9
9 10
10 11
11 8

12 13
13 14
14 15
15 12

8 12
9 13
10 14
11 15

```

We could equivalently add the offset to each individual point, but keeping it separate makes it easy to change. The second line holds the total number of data points. Following that are the data points themselves, one to a line, specified as the triple x, y, and z. After all that, we get the total number of lines that we will define. The

rest of the file consists of these line definitions, each of which refers to its endpoints using the order in which the points were read (e.g., the first data point is point 0). Why define the lines separately? It's not terribly important in this example, but in future articles each data point will typically participate in several lines, so using this method cuts down on the number of points we need to calculate.

We read these values into a few arrays, which for now are of fixed size. Then we convert the 3-D points to 2-D screen positions, all at once, and use those values along with the connectivity array to draw a bunch of lines on the screen. To flaunt our mastery of special effects, we'll simulate actual motion by erasing the image and redrawing it slightly farther away. You won't believe your eyes. To keep the image from flickering too badly, I've used a naïve method of double-buffering. It simply uses two screens, drawing on the hidden one, then flipping it to the front. Not user-friendly, but effective.

The program itself is written in C, and compiles with DICE or Sozobon C. You can get by without the standard Amiga include files if you define NewWindow and NewScreen directly, but in the future we will increasingly rely on the includes and the standard library, so it would be a good idea to send away for them. In addition to the C headers you get assembly includes, documentation for the library functions, and all sorts of other fun stuff. To get all this, send \$20 to CATS (Commodore Application and Technical Support) and ask for the AmigaDOS 1.3 Native Developers Update (at this writing, the AmigaDOS 2.0 update is not yet available). The address for CATS is:

CATS
1200 Wilson Drive
West Chester, PA 19380

The system we've created this time works fine, and if you can arrange your data points to suit this method, you're all set. Nonetheless, it certainly has its problems. For one thing, we use normal integers to specify points. This leaves us open to round-off errors, so we're going to need to use fractional numbers somehow. Plus we have the constraints of the model to deal with: the camera is always fixed at the origin, pointing along the positive z axis. The positive y axis always points up. Next time around, we'll overcome all of these problems. Heck, we'll invent extra problems, just to overcome them. The price, as usual, will be additional calculations.

Listing One 3D.c

```

/*
 3D.c

This program reads an object specified in
3-dimensional coordinates and displays it
using perspective rendering.

Usage: 3d InputFile

*/

#include <stdio.h>
#include "intuition/intuition.h"

/* Maximum number of data points we can read. */
/* We'll also use this to determine the max */
/* number of line segments, which we'll set */
/* arbitrarily at MAXPOINTS*2 */

#define MAXPOINTS      30

/* This is the distance between the focal */
/* point and the screen. A small value here */
/* creates fish-bowl pictures. Larger values */
/* minimize the effects of perspective. */
/* Use a power of two here if possible - that */
/* way the compiler will convert the */
/* multiplication below to a much faster shift */

#define DISTANCE      32

/* Define constants for our screen size. If you */
/* set MAXX to 640 or MAXY to 400, the screen */
/* automatically be opened in HIRIS and LACE */
/* modes (see OpenDisplay()) */

#define MAXX          320
#define MAXY          200
#define HALFX         (MAXX / 2)
#define HALFY         (MAXY / 2)

/* 3D_Point : One point from the input file. */

typedef struct {
    long X,Y,Z;
} Point_3D;

/* 2D_Point : The screen image of a 3D_Point */

typedef struct {
    long X,Y;
} Point_2D;

/* LineSeg : One line segment from input file */

typedef struct {
    long start,end;
} LineSeg;

/* The libraries we'll need: */

void *IntuitionBase = 0,
     *GfxBase = 0;

/* To get rid of flicker, we'll actually open two */
/* screens, then flip back and forth to simulate */
/* double buffering. This is an easy, but user- */
/* unfriendly method. To see why, try doing */
/* anything else while this demo is running. */

struct Window *window1 = 0,
              *window2 = 0,
              *frontwindow;

struct Screen *screen1 = 0,
              *screen2 = 0;

/* Regardless of which screen is in front, this */
/* rastport will always be the correct one in */
/* which to draw. */

struct RastPort *rp;

/* Our input file: */

FILE *ObjectFile = 0;

/* The message that will tell us to get out: */

struct Message *quitmsg;

long z_offset, /* Current offset */
    initial_offset; /* From the input file */

long TotalPoints, /* Number of data points */
    TotalLines; /* Number of line segments */
/* defined in input file */

/* Actual object data read from the input file */

Point_3D Points[MAXPOINTS];

/* 3D points converted to screen coordinates */

Point_2D Display[MAXPOINTS];

/* Line segment definitions from input file */

LineSeg Lines[MAXPOINTS*2];

/* If for some reason we can't open a screen or */
/* something else goes haywire, we call this */
/* routine to notify the user and bug out cleanly */

void Quit(msg)

char *msg;

{
    if (ObjectFile)
        fclose(ObjectFile);

    if (window1)

```

```

    CloseWindow(window1);
if (screen1)
    CloseScreen(screen1);

if (window2)
    CloseWindow(window2);
if (screen2)
    CloseScreen(screen2);

if (IntuitionBase)
    CloseLibrary(IntuitionBase);
if (GfxBase)
    CloseLibrary(GfxBase);

puts(msg);
exit(0);
}

/* This routine is fairly straightforward. It
/* reads the data from the file, expecting integer*/
/* numbers in the following order: */

/*      Initial_Z_Offset      */
/*      Number_Of_Data_Points */
/*      Data_X Data_Y Data_Z  */
/*      ...                   */
/*      Number_Of_Line_Segments */
/*      Start_Point End_Point */
/*      ...                   */

/* Note that by cleverly using standard C IO
/* routines, we have allowed ourselves the freedom
/* to insert as much or as little white space as
/* we like between the numbers. Therefore the
/* order of the values is important, but the
/* layout is up to you. */

void ReadObjectFile(fname)

char *fname;

{
    char *ErrorMsg =
        "Error reading object definition";
    long Coord;
    short Seg, i;

    ObjectFile = fopen(fname, "r");
    if (ObjectFile) {
        if (fscanf(ObjectFile, "%ld",
&initial_offset)==EOF)
            Quit(ErrorMsg);

        if (fscanf(ObjectFile, "%ld",&TotalPoints)==EOF)
            Quit(ErrorMsg);

        if ((TotalPoints<1) || (TotalPoints > MAXPOINTS))
            Quit("Point count out of range");

        for (i=0; i<TotalPoints; i++)
            if (fscanf(ObjectFile,"%ld %ld %ld",
&Points[i].X,&Points[i].Y,&Points[i].Z)==EOF)
                Quit(ErrorMsg);

        if (fscanf(ObjectFile,"%ld",&TotalLines)==EOF)
            Quit(ErrorMsg);

        if ((TotalLines<1) || (TotalLines > MAXPOINTS*2))

```

```

        Quit("Line count out of range");

        for (i=0; i<TotalLines; i++)
            if (fscanf(ObjectFile, "%ld %ld",
&Lines[i].start, &Lines[i].end)==EOF)
                Quit(ErrorMsg);

        fclose(ObjectFile);
    } else
        Quit("Could not open input file");
}

/* Open a couple of screens and windows. We'll
/* need two of each for double buffering. Also,
/* this routine jump-starts the double buffering
/* process. */

void OpenDisplay()
{
    struct NewWindow NW;
    struct NewScreen NS;

    NS.LeftEdge    = 0;
    NS.TopEdge     = 0;
    NS.Width       = MAXX;
    NS.Height      = MAXY;
    NS.Depth       = 1;
    NS.DetailPen    = 0;
    NS.BlockPen     = 0;
    NS.ViewModes   = ((MAXX > 320) ? HIRRES : 0) |
        ((MAXY > 200) ? LACE : 0);
    NS.Type        = CUSTOMSCREEN;
    NS.Font        = NULL;
    NS.DefaultTitle = (UBYTE *) "";
    NS.Gadgets     = NULL;

    if (!(screen1 = (void *) OpenScreen(&NS)))
        Quit("Could not open screen");
    ShowTitle(screen1, FALSE);

    if (!(screen2 = (void *) OpenScreen(&NS)))
        Quit("Could not open screen");
    ShowTitle(screen2, FALSE);

    NW.LeftEdge    = 0;
    NW.TopEdge     = 0;
    NW.Width       = MAXX;
    NW.Height      = MAXY;
    NW.DetailPen    = 0;
    NW.BlockPen     = 0;
    NW.IDCMPFlags   = VANILLAKEY;
    NW.Flags       = BACKDROP | BORDERLESS | ACTIVATE;
    NW.FirstGadget = NULL;
    NW.CheckMark    = NULL;
    NW.Title       = (UBYTE *) "";
    NW.Screen      = screen1;
    NW.Type        = CUSTOMSCREEN;

    if (!(window1 = (struct Window *) OpenWindow(&NW)))
        Quit("Could not open the window");

    NW.Screen      = screen2;
    NW.IDCMPFlags   = 0;
    NW.Flags       = BACKDROP | BORDERLESS;

    if (!(window2 = (struct Window *) OpenWindow(&NW)))
        Quit("Could not open the window");
}

```



```

frontwindow = window1;
ScreenToFront(screen1);

rp = window2->RPort;

}

/* Shows the screen we have, presumably, just
/* finished drawing on. This routine also sets
/* up the correct rasterport, so if we always draw
/* into rp we never need to know the details of
/* which screen is in front.
*/

void SwapBuffers()
{
    rp = frontwindow->RPort;

    if (frontwindow == window1) {
        ScreenToFront(screen2);
        frontwindow = window2;
    } else {
        ScreenToFront(screen1);
        frontwindow = window1;
    }
}

/* Given the 3D coordinates, calculate the screen
/* position. Note that we actually switch the
/* screen coordinate to go from bottom to top, to
/* match our model. Also note that we avoid
/* division by zero problems, but I left in the
/* negative Z problem so you could see what it
/* looked like. Normally you'd just refuse to
/* draw any point behind the camera.
*/

void CalculateDisplay()
{
    long i, ActualZ;

    for (i=0; i<TotalPoints; i++) {
        ActualZ = z_offset + Points[i].Z;
        if (!ActualZ)
            ActualZ = 1;

        Display[i].X =
            (Points[i].X * DISTANCE) / ActualZ + HALFX;
        Display[i].Y =
            HALFY - (Points[i].Y * DISTANCE) / ActualZ;
    }
}

/* Having figured out all the correct screen
/* positions, display the object in the given
/* color.
*/

void ShowObject(color)

    int color;

{
    short i;
    long startpoint, endpoint;

    SetAPen(rp, color);

```

MISSED AN ISSUE?

CHANGE
OF ADDRESS?

SUBSCRIPTION
PROBLEMS?

CALL US!

1-800-345-3360

```

for (i=0; i<Totallines; i++) {
    startpoint = Lines[i].start;
    endpoint = Lines[i].end;

    Move(rp, Display[startpoint].X,
        Display[startpoint].Y);

    Draw(rp, Display[endpoint].X,
        Display[endpoint].Y);
}

```

```

/* In the main routine, we initialize all the
/* globals, as well as open the screen and read
/* the object from the input file. We then
/* begin displaying the object, varying the z
/* offset as we go to simulate motion. If we
/* wanted to get tricky we could do similar
/* things to the x and y axes, too.
*/

```

```

main(argc, argv)

    int argc;
    char *argv[];

{
    long increment;

    if (argc < 2)

```

```

Quit("Usage: 3d_objectfile");

if (!(IntuitionBase = (void *)
    OpenLibrary("intuition.library", 0L)))
    Quit("No Intuition");

if (!(GfxBase = (void *)
    OpenLibrary("graphics.library", 0L)))
    Quit("No Graphics");

OpenDisplay();

ReadObjectFile(argv[1]);

z_offset = initial_offset;
increment = (initial_offset / 100) + 1;

CalculateDisplay();

while (!(quitmsg =
    (void *) GetMsg(window1->UserPort))) {

    for (z_offset = initial_offset;
        z_offset < (initial_offset << 2);
        z_offset += increment) {

        CalculateDisplay();
        SetAPen(rp, 0);
        RectFill(rp, 0, 0, MAXX, MAXY);
        ShowObject(1);
        SwapBuffers();
    }
}

```

```

for (z_offset = initial_offset << 2);
    z_offset > initial_offset;
    z_offset += increment) {

    CalculateDisplay();
    SetAPen(rp, 0);
    RectFill(rp, 0, 0, MAXX, MAXY);
    ShowObject(1);
    SwapBuffers();
}

ReplyMsg(quitmsg);

CloseWindow(window1);
CloseScreen(screen1);
CloseWindow(window2);
CloseScreen(screen2);

CloseLibrary(IntuitionBase);
CloseLibrary(GfxBase);
}

```



WE INTERRUPT THIS HIGHLY INFORMATIVE ARTICLE FOR A VERY SPECIAL PROGRAM ANNOUNCEMENT!

We want to publish *your* very special program in an upcoming issue of AC.

Or, your highly informative article on *any* topic of interest to Amiga users of all skill levels!



The fact is, *Amazing Computing* has always published the most unique, most detailed Amiga programming articles and tutorials found anywhere! AC pays competitive per-page rates to its authors, but publishes *more and longer programming articles per issue* than any other Amiga monthly. That's great news not only for our readers, but also for those of you who are thinking about achieving fame and fortune as freelance writers! And now, the more complex works of high-level programmers are considered for publication in AC's *TECH*. The fact is, AC's *TECH* is the #1 all-technical, disk-based Amiga journal.

Whatever your areas of greatest interest or proficiency on the Amiga, there are probably any number of tips, techniques and tricks you can communicate to Amiga users worldwide, in the pages of *Amazing Computing*.

Even if you have never been published before, you should consider writing for AC. Our knowledgeable, experienced editors are the most helpful in the business.

Call our editorial offices during normal business hours at 1-800-345-3360 to have the *Amazing Computing Author's Guide* information packet sent to you TODAY!

AC'S Back Issue Index

Vol. 1 No. 1, Premier, 1986

Highlights include:
 "Super Spheres", An Amiga Graphics Program, by Kelly Kaufman
 "Duke Virus", by J. Foust
 "EZ-Term", An Amiga terminal program, by Kelly Kaufman
 "Mega Mania", Programming fixes & mouse care, by P. Kivolvitz
 "Inside CLP", A guided insight into AmigaDOS, by G. Musser

Vol. 1 No. 2, 1986

Highlights include:
 "Inside CLP Part Two", Investigating CLP & ED, by G. Musser
 "Online and the CTS Fabite 2424 ADH Modem", by J. Foust
 "Superterm V 1.0", A terminal program in Amiga Basic, by K. Kaufman
 "A Workbench 'Mouse' Program", by Rick Wirth

Vol. 1 No. 3, 1986

Highlights include:
 "Forth", A tutorial
 "Deluxe Draw", An AmigaBASIC art program, by R. Wirth
 "AmigaBASIC", A beginner's tutorial
 "Inside CLP Part 3", by George Musser

Vol. 1 No. 4, 1986

Highlights include:
 "Build Your Own 5 1/4" Drive Connector", by E. Viveiros
 "AmigaBASIC Tips", by Rick Wirth
 "Scrimper Part One", A program to print Amiga screens, by P. Kivolvitz

Vol. 1 No. 5, 1986

Highlights include:
 "The HSI to RGB Conversion Tool", Color manipulation in BASIC, by S. Pietrowicz
 "Scrimper Part Two", by Perry Kivolvitz
 "Building Tools", by Daniel Kary

Vol. 1 No. 6, 1986

Highlights include:
 "Mailing List", A basic mail list program, by Kelly Kaufman
 "Printer Image Editor", by Stephen Pietrowicz
 "Scrimper Part Three", by Perry Kivolvitz
 "Optimize Your AmigaBASIC Programs For Speed", by Steve Pietrowicz

Vol. 1 No. 7, 1986

Highlights include:
 "Try 3-D", An introduction to 3-D graphics, by Jim Meadows
 "Window Requesters in Amiga Basic", by Steve Michel
 "I C What I Think", A few C graphic progs, by R. Petersen
 "Your Menu Sift", Programming AmigaBASIC menus, by R. Cooley
 "Linking C Programs with Assembler Routines", by G. Hull

Vol. 1 No. 8, 1986

Highlights include:
 "Using Fonts from AmigaBASIC", by Tim Jones
 "Screen SaVer", Monitor protection program in C, by P. Kivolvitz
 "A Tale of Three EMACS", by Steve Poling
 ".bmap File Reader in AmigaBASIC", by T. Jones

Vol. 1 No. 9, 1986

Highlights include:
 "The Loan Information Program", A BASIC program for your financial options, by Brian Catley
 "Starting Your Own Amiga-Related Business", by W. Simpson
 "Keep Track of Your Business Usage for Taxes", by J. Kummer
 "Using Fonts from AmigaBASIC Part Two", by Tim Jones
 "68000 Macros On The Amiga", by G. Hull

Vol. 2 No. 1, January 1987

Highlights include:
 "AmigaBASIC Titles", by Bryan Catley
 "A Public Domain Modula-2 System", by Warren Block
 "One Drive Compile", by Douglas Lowell
 "A Megabyte Without Megabucks", An internal megabyte upgrade, by Chris Irving

Vol. 2 No. 2, February 1987

Highlights include:
 "The Modem", Efforts of a BBS synop, by Joseph L. Rothman
 "The ACD Project...Graphic Teleconferencing on the Amiga", by S. Pietrowicz
 "A Disk Librarian in AmigaBASIC", by John Kerman
 "Creating And Using Amiga Workbench Icons", by C. Hansel
 "Build Your Own MIDI Interface", by Richard Rae
 "AmigaDOS Operating System Calls and Disk File Management", by D. Haynes

Vol. 2 No. 3, March 1987

Highlights include:
 "Subscripts and Superscripts in AmigaBASIC", by I. Sestih
 "AmigaTtr", Amiga shortcuts, by W. Block
 "Intuition Gadgets", by Harriet Maybeck Tolly
 "Forth", Put sound in your Forth programs, by Jon Bryan
 "Assembly Language on the Amiga", by Chris Martin

Vol. 2 No. 4, April 1987

Highlights include:
 "Jim Sachs Interview", by S. Hull
 "The Mouse That Got Restored", by Jerry Hull and Bob Rhodes
 "Household Inventory System in AmigaBASIC", by B. Catley
 "Secrets of Screen Dumps", by Nathan Okun
 "AmigaTtr II", More Amiga shortcuts, by Warren Block

Vol. 2 No. 5, May 1987

Highlights include:
 "Writing a SoundScape Module", Programming with MIDI, Amiga and SoundScape in C, by T. Fay
 "Programming in 68000 Assembly Language", by C. Martin
 "Using FutureSound with AmigaBASIC", Programming utility with real digitized STEREO, by J. Meadows
 "Waveform Workshop in AmigaBASIC", by J. Shields
 "Intuition Gadgets Part II", by H. Maybeck Tolly

Vol. 2 No. 6, June 1987

Highlights include:
 "Modula-2 AmigaDOS Utilities", by S. Falwiszewski
 "Amiga Expansion Peripherals", by J. Foust
 "What You Should Know Before Choosing an Amiga 1000 Expansion Device", by S. Grant
 "68000 Assembly Language Programming", by Chris Martin

Vol. 2 No. 7, July 1987

Highlights include:
 "Video and Your Amiga", by Grant Sands
 "Quality Video from a Quality Computer", by G. Sands
 "All About Printer Drivers", by Richard Biefak
 "68000 Assembly Language", by Chris Martin

Vol. 2 No. 8, August 1987

Highlights include:
 "Modula-2 Programming",
 "Assembly Language"
 "Disk-2-Disk", by Matthew Leeds
 "Skinny C Programs", by Robert Riemann, Jr.

Vol. 2 No. 9, September 1987

Highlights include:
 "Modula-2 Programming", Raw console dev. events, by S. Falwiszewski
 "AmigaBASIC Patterns", by Brian Catley
 "Programming with SoundScape", by T. Fay
 "Bill Valk, Vice-President Argis Development", interview by Steve Hull
 "Jim Goodnow, Developer of Manx 'C'", interview by Harriet M Tolly

Vol. 2 No. 10, October 1987

Highlights include:
 "Max Headroom and the Amiga", by John Foust
 "Taking the Perfect Screen Shot", by Keith Cordoni
 "Amiga Artist: Brian Williams", by John Foust
 "All About On-line Conferencing", by Richard Rae
 "Amiga BASIC Structures", by Steve Michel
 "Quick and Dirty Bebs", by Michael Swinger
 "Fast File I/O with Modula-2", by Steve Falwiszewski
 "Window I/O", by Read Prelmore

Vol. 2 No. 11, November 1987

Highlights include:
 "Modula-2 Programming", Devices, I/O, & serial port, by S. Falwiszewski
 "68000 Assembly Language", by Chris Martin
 "The AMICUS Network", by John Foust
 "C Animation: Part II", by Mike Swinger
 "SoundScape Part III", VU Meter and more, by Tudor Fay
 "File Browser", by Bryan Catley

Vol. 2 No. 12, December 1987

Highlights include:
 "The Sony Connection", by Stewart Cobb
 "CLI Arguments in C", by Paul Castonguay
 "MIDI Interface Adapter", by Barry Massoni
 "Modula-2", Command line calculator, by S. Falwiszewski
 "Animation for C Rookies: Part III", by M. Swinger
 "The Big Picture", Assembly language programming, by Warren Ring
 "Insider/Kwikstart Review", RAM & ROM expansion: Comments & installation tips, by Ernest P. Viveiros, Sr.
 "Forth", DumpRFont utility for your Multi-Forth toolset, by Jon Bryan

Vol. 3 No. 1, January 1988

Highlights include:
 "C Animation: Part IV", by Michael Swinger
 "Forth", Sorting out Amiga CHIP and FAST memory, by John Bryan
 "The Big Picture", CLI system calls and manipulating disk files, by Warren Ring
 "68000 Assembly Language Programming", Create a multi-color screen without using Inception routines, by Chris Martin
 "Modula-2 Programming", by S. Falwiszewski
 "FormatMaster: Professional Disk Formatting Engine", by C. Mann
 "Spread", Full featured AmigaBASIC spreadsheet, by Bryan Catley

Vol. 3 No. 2, February 1988

Highlights include:
 "Laser Light Shows with the Amiga", by Patrick Murphy
 "Photo Quality Reproduction with the Amiga and DigView", by Stephen Lebars
 "Solutions To Linear Algebra Through Matrix Computations", by Robert Ellis
 "Modula-2 Programming", Catching up with Calc, by Steve Falwiszewski
 "68000 Assembler Language Programming", by Chris Martin
 "AIRT", Icon-based program language, by S. Falwiszewski

Vol. 3 No. 3, March 1988

Highlights include:
 "The Hidden Power of CLI Batch File Processing", by J. Rothman
 "Perry Kivolvitz Interviewed", by Ed Berowitz
 "Jean 'MacBlas' Giraud Interviewed", by Ed Fadigan
 "FAL Help", A1000 expansion reliability, by Perry Kivolvitz
 "Boolean Function Minimization", by Steven M. Hart
 "Amiga Serial Port and MIDI Compatibility for Your A1000", by L. Ritter and G. Rente
 "Electric Network Solutions the Matrix Way", by Robert Ellis
 "Modula-2 Programming", The gameport device and simple sprites in action, by Steve Falwiszewski
 "The Big Picture", Unified Field Theory by Warren Ring

■ Vol. 3 No. 4, April 1988

Highlights include:

- "Writing A SoundScape Patch Libraries", by T. Fay
- "Upgrade Your A1000 to AS09/2000 Audio Power", by H. Baines
- "Gels in Multi-Forth", by John Bushakra
- "Macabatters", Easing the trauma of Assembly language programming, by Patrick J. Horgan
- "The Big Picture, Part II: Unified Field Theory", by W. Ring

■ Vol. 3 No. 5, May 1988

Highlights include:

- "Interactive Startup Sequence", by Udo Pernisz
- "AmigaTris III", by Warren Block
- "Proletariat Programming", Public domain compilers, by P. Quaid
- "The Companion", Amiga's event-handling capability, by P. Gosselin
- "The Big Picture, Unified Field Theory: Part III", by W. Ring
- "Modula-2", Termination modules for Benchmark and TDI compilers, by Steve Fawcett
- "68000 Assembly Language", Peeling away the complication of display routines, by Chris Martin

■ Vol. 3 No. 6, June 1988

Highlights include:

- "Reassigning Workbench Disks", by John Kerman
- "An IFF Reader in Multi-Forth", by Warren Block
- "Basic Directory Service Program", Programming alternative to the CommuZeroZero, by Bryan Catley

■ Vol. 3 No. 7, July 1988

Highlights include:

- "Roll Those Presses!", The dandy, demanding world of desktop publishing, by Barney Schwartz
- "Linked Lists in C", by W. E. Gattmiller
- "C Notes from the C Group", The unknown "C" of basic object and data types, by Stephen Kemp

■ Vol. 3 No. 8, August 1988

Highlights include:

- "The Developing Amiga", A gaggle of great programming tools, by Stephen R. Pietrowicz
- "Modula-2 Programming", Libraries and the FFT and IIR math routines, by Steve Fawcett
- "C Notes from the C Group: Arrays and pointers unmasked", by Stephen Kemp
- "TrackMouse", Converting a standard Atari trackball into a peppy Amiga TrackMouse, by Darryl Joyce
- "Amiga Interface for Blind Users", by Carl W. Mann
- "Tumbler's Tots", Assembly language program, by D. Ashley

■ Vol. 3 No. 9, September 1988

Highlights include:

- "Speeding Up Your System", Floppy disk caching, by Tony Preston
- "Computer-Aided Instruction", Authoring system in AmigaBASIC, by Paul Castonguay
- "Gels in Multi-Forth, Part II: Screenplay", by John Bushakra
- "AmigaNotes: How IFF sound samples are stored", by Richard Ray
- "C Notes from the C Group", Operators, expressions, and statements in C uncovered, by Stephen Kemp

■ Vol. 3 No. 10, October 1988

Highlights include:

- "The Command Line/NEWCLI: A painless way to create a new console window", by Rich Falconburg
- "On The Crafting of Programs", Optimization kicks off our series on programming savvy, by David J. Harkins
- "Bob and Ray Meet Frankenstein", Create, animate, and metamorphose graphics objects in AmigaBASIC, by R. D'Auto
- "Digital Signal Processing in AmigaBASIC", Perform your own digital experiments with Fast Fourier Transforms, by Robert Ellis
- "HAM & AmigaBASIC", Pack your AmigaBASIC programs with many of the Amiga's 4096 shades, by Bryan Catley
- "CAI—Computer Aided Instruction: Part II", by Paul Castonguay

■ Vol. 3 No. 11, November 1988

Highlights include:

- "Structures in C", by Paul Castonguay
- "On The Crafting of Programs", Speed up your programs, by D. Harkins
- "More Linked Lists in C: Techniques and Applications", Procedures for managing lists, storing diverse data types in the same list, and putting lists to work in your programs, by Forest W. Arnold
- "BASIC Linker", Combine individual routines from your program library to create an executable program, by B. Zupke

■ Vol. 3 No. 12, December 1988

Highlights include:

- "Converting Patch Libraries Files", by Phil Saunders
- "The Creation of Den Blath's Dragon's Lair", by R. Linden
- "Easy Menus in [Forth]", by Phil Burk
- "Extending AmigaBASIC", The use of library calls from within AmigaBASIC, by John Kerman
- "Getting Started in Assembly", by Jeff Glati
- "C Notes From The C Group: Program or function control coding", by Stephen Kemp
- "AmigaDOS, Assembly Language, And FileNotes", Weapons in the war against file overload; accurate, descriptive file naming, by Dan Huith

■ Vol. 4 No. 1, January 1989

Highlights include:

- "Desktop Video", by Richard Starr
- "Industrial Strength Menus", by Robert D'Auto
- "Scrolling Through SuperBMap Windows", by Read Prodmore
- "Sync Tips: Dot crawl, the Amiga and composite video devices", by Oran J. Sands
- "Stop-Motion Animation On The Amiga", by Brian Zupke
- "The Command Line: New and Improved Assembly Language Commands", by Rich Falconburg
- "Painters, Function Pointers, and Pointer Declarations in C", by Forest W. Arnold
- "Death of a Process", Developing an error-handling module in Modula-2, by Mark Cashman

■ Vol. 4 No. 2, February 1989

Highlights include:

- "A Common User Interface for the Amiga", by Jim Bayless
- "SPY: Programming Intrigue In Modula-2", by Steve Fawcett
- "Sync Tips: Getting inside the genlock", by Oran Sands
- "On the Crafting of Programs: A common standard for C programming", by D. J. Harkins
- "The Command Line: Your Workbench Screen Editor", by Rich Falconburg
- "An Introduction to ARexx programming", by Steve Fawcett

■ Vol. 4 No. 3, March 1989

Highlights include:

- "Fractal Fundamentals", by Paul Castonguay
- "Image Processing With Photosynthesis", by Gerald Hull
- "Benchmark 3: Fully Utilizing The MC68010", Part I: Turbocharging the savage benchmark, by Read Prodmore
- "Breaking the Bmap Barrier", Streamline AmigaBASIC library access with Quick-Lib, by Robert D'Auto
- "Double Play", AmigaBASIC program risks double vision, by Robert D'Auto

■ Vol. 4 No. 4, April 1989

Highlights include:

- "Adding the Net-So-Hard Disk", by J. P. Twanley
- "The Hard Drive Kit", A hard drive installation project, using Palomax's Max kit, by Donald W. Morgan
- "Sync Tips: A clearer picture of video and computer resolutions", by Oran J. Sands
- "Passing Arguments", Step-by-step on how to pass data from the CLI to AmigaBASIC, by Brian Zupke
- "Creating a Shared Library", by John Bazz

■ Vol. 4 No. 5, May 1989

Highlights include:

- "Building Your Own Stereo Digitizer", by Andre Theberge
- "MIDI Out Interface", by Br. Seraphim Winslow
- "Digitized Sounds in Modula-2", by Len A. White
- "Sync Tips: The secrets hidden beneath the flicker mode", by Oran J. Sands

■ Vol. 4 No. 6, June 1989

Highlights include:

- "AI Your Request: Design your own requesters in AmigaBASIC", by John F. Wiederhorn
- "Exploring Amiga Disk Structures", by David Martin
- "Diskless Compile in C", by Chuck Rudzinski
- "Programming the '801 Part II", How to calculate Mandelbrot & Julia sets, by Read Prodmore

■ Vol. 4 No. 7, July 1989

Highlights include:

- "Adapting Analog Joysticks to the Amiga", by David Singer
- "Using Coordinate Systems: Part II of the fractals series addresses the basis of computer graphics", by P. Castonguay

■ Vol. 4 No. 8, August 1989

Highlights include:

- "Getting Started in Video", by Richard Starr
- "Executing Batch Files in AmigaBASIC", by Mark Aydelunde
- "Building a Better String Gadget", by John Bushakra
- "On Your Alert: Using System Alerts from BASIC", by John F. Wiederhorn

■ Vol. 4 No. 9, September 1989

Highlights include:

- "Digitizing Color Slides And Negatives on the Amiga", by Ron Gull
- "Improving Your Graphics Programming", by E. Martin
- "Cell Animation in Modula-2", by Nicholas Crasella
- "More Requesters in AmigaBASIC", by John F. Wiederhorn
- "DeluxePaint III — The Inside Story", EA's Dan Silva tells how DeluxePaint III evolved, by Ben & Jean Means
- "Amiga In Desktop Presentation", Presentation techniques to enhance your meetings and seminars, by John Steiner
- "Multitasking in Fortran", by Jim Locker
- "Gels in Multi-Forth: Part III", by John Bushakra

■ Vol. 4 No. 10, October 1989

Highlights include:

- "Better TrackMouse", A true one-handed trackball mouse, by Robert Katz
- "APL & The Amiga", by Henry Uppert
- "Saving 16-color pictures in high-resolution", Part Three of the Fractals Series, by Paul Castonguay
- "More requesters in AmigaBASIC", by John F. Wiederhorn
- "Glat's Gadgets", Adding gadgets in Assembly, by Jeff Glati
- "Function Evaluator in C", by Randy Finch
- "Typing Tutor", by Mike 'Chip' Morrison

■ Vol. 4 No. 11, November 1989

Highlights include:

- "The Amiga Hardware Interface", by John Irvine
- "APL & The Amiga, Part II", by Henry Uppert
- "FastPlay", A faster pixel-drawing routine for the Amiga C compiler, by Scott Steinman
- "64 Colors in AmigaBASIC", by Bryan Catley
- "Fast Fractals", Generate Mandelbrot Fractals at lightning speed, by Hugo M.H. Lypkens
- "Multitasking in Fortran", by Jim Locker

■ Vol. 4 No. 12, December 1989

Highlights include:

- "The MIDI Must Go Thru", by Br. Seraphim Winslow
- "View From the Inside: Bars&Pipes", A tour of Blue Ribbon Bakery's music program, by Melissa Jordan Grey
- "ARexx Part II", by Steve Gilmor
- "A CLI Beginner's Questions Answered", by Mike Morrison
- "Trees and Recursion", by Forest W. Arnold
- "Amiga Circuits", The techniques required to input information via the parallel port, by John Irvine

■ Vol. 5 No. 1, January 1990

Highlights include:

- "The Making Of The 1989 BADGE KILLER Demo Contest Winner, The Sentinel", by Bradley W. Schenk
- "Animation! BASiCally!", Using Cell animation in AmigaBASIC, by Mike Morrison
- "Menu Builder", Building menus with intuition, by T. Preston
- "Facing the CLI", Disk structures and startup sequences, by Mike Morrison
- "Dual Demo", Programming an arcade game, by Thomas Eshelman
- "Scanning The Screen", Part Four in the Fractals Series, by Paul Castonguay
- "It's Colder Than You Think", Calculating the wind chill temperature, by Robert Klimaszewski

■ Vol. 5 No. 2, February 1990

Highlights include:

- "A Beginner's Guide to Desktop Publishing On The Amiga", by John Steiner
- "Resizing the shell/CLI Window", by William A. James
- "Call Assembly Language from BASIC", by Martin F. Combs
- "You Too Can Have A Dynamic Memory", Flexible string gadget requester using dynamic memory allocation, by Randy Finch
- "An Amiga Conundrum", An AmigaBASIC program for a puzzle-like game, by David Serger
- "View From The Inside: ScanLab", ASDG's President shares the development of ScanLab, by Perry Kivolevitz

■ Vol. 5 No. 3, March 1990

Highlights include:

- "Screen Aid", A quick remedy to prolong the life of your monitor, by Bryan Catley
- "The Other Guys' Synthia Professional", review by David Duerham
- "Famper's Master Tracks Pro vs. Blue Ribbon Bakery's Bars&Pipes", by Ben Means
- "MicroIllusions' Music-X", review by Rob Bryanton
- "MusicTiler", Generating a tiled display to accompany the audio on a VCR recording, by Brian Zupke

Vol. 5 No. 4, April 1990

Highlights include:

"Handling MS-DOS Files", Adapting your Amiga to MS-DOS using a 5.25" disk drive, by Jim Locker
 "Bridging the 3.5" Chasm", Making Amiga 3.5" drives compatible with IBM 3.5" drives, by Karl D. Belam
 "Bridgeboard Q & A", by Marion Deland
 "Handling Gadget & Mouse IntuitionEvents", More gadgets in Assembly, by Jeff Galt
 "Ham Bites", Programming in HAM mode in AmigaBASIC, by Robert D. Aiso
 "Gambling with your video, Amiga-style", Problems with trading genlocks with your friends, by Oren Sands

Vol. 5 No. 5 May 1990

Highlights include:

"Commodore's Amiga 3800", preview
 "Newtek's Video Toaster", preview
 "Do It By Remote", Building an Amiga-operated remote controller for your home, by Andre Thierberg
 "Turn Your Amiga 1000 Into A ROM-based Machine", by George Gibson Jr. & Dwight Blumhugh
 "Super Bitmaps in BASIC", Holding a graphics display larger than the monitor screen, by Jason Cahill
 "Rounding Off Your Numbers", by Sedgewick Strum Jr.
 "Faster BASIC Mouse Input", by Michael S. Fahrion
 "Print Utility", by Brian Zagke

Vol. 5 No. 6, June 1990

Highlights include:

"Convergence", Part 5 of the Fractal series, by P. Castonguay
 "C++: An introduction to object-oriented Amiga programming", by Scott B. Steinman
 "APL and The Amiga: Primitive Functions and Their Execution", by Henry T. Lippert
 "Amiga Turtle Graphics", by Dylan McNamara
 "Building A Rapid Fire Joystick", by John Iovine
 "The AM 512", Upgrade your A500 to a 1 megabyte machine, by James Bentley

Vol. 5 No. 7, July 1990

Highlights include:

"Commodore Announces CDTV"
 "Apples, Oranges, and MIPS: 68030-based Accelerators For The Amiga 2000", by Ernest P. Viveiros, Jr.
 "Exceptional Conduct", Quick response to user requests, through efficient program logic, by Mark Cashman
 "Poor Man's Spreadsheet", A simple spreadsheet program that demonstrates manipulating arrays, by Gerry L. Penrose
 "Tree Traversal and Tree Search", Two methods for traversing trees, by Forest W. Arnold
 "Crunchy Frog II", by Jim Fiore
 "Getting in the Point: Custom Intuition Pointers in AmigaBASIC", by Robert D. Aiso
 "Synchronicity: Right & Left Brain Lateralization", by John Iovine
 "Snap, Crackle, & POP!", Fixing a monitor bug on Commodore monitors, by Richard Landry

Vol. 5 No. 8, August 1990

Highlights include:

"Mimetic's FrameBuffer", review by Lonnie Watson
 "The VidTech Scanlock", review by Oren Sands
 "Amigas in Television", The Amiga in a cable television operation, by Frank McMahon
 "Desktop Video in a University Setting", The Amiga at work at North Dakota State University, by John Steiner
 "Credit Test Scroller", review by Frank McMahon
 "Graphic Suggestions", Other ways to use your Amiga in video production, by Bill Burkett
 "Title Screens That Shine: Adding light sources with DeluxePaint III", by Frank McMahon
 "The Amiga goes to the Andys", by Curt Kass
 "Breaking the RAM Barrier", Longer, faster, smoother animations with only one meg of RAM, by Frank McMahon
 "Fully Utilizing the 68881 Math Coprocessor Timings and Turbo_Pixel functions", by Read Prodmore
 "APL and the Amiga Part IV", by Henry T. Lippert
 "Sound Quest's MIDIQuest", review by Hal Belden

Vol. 5 No. 9, September 1990

Highlights include:

"Dr. T's Keyboard Controlled Sequencer 2.0", review Phil Saunders
 "Acting On Impulse", A visit to Impulse, by John Steiner
 "2-D Professional", review by David Duberman
 "Programming In C on a Floppy System", Yes even a stock A500 with a 512K RAM expander, by Paul Miller
 "Time Out", Accessing the Amiga's system timer device via Modula-2, by Mark Cashman
 "Stock Portfolio", An original program to organize your investments, music library, mailing lists, etc., by G.L. Penrose

"Voice-Controlled Joystick", by John Iovine

"FrameGrabber", review by Lonnie Watson

"Gradient Color Dithering on the Amiga Made Easy", by Francis Gardino
 "Sculpt Script", by Christian Aubert
 "The Art Department", review by R. Shamus Mortier
 "Breaking the Color Limit with PageRender3D", review by R. Shamus Mortier

Vol. 5 No. 10, October 1990

Highlights include:

"Notes on PostScript Printing with Dr. T's Copyist", by Hal Belden
 "BioMetal", Make the Amiga flex its first electric muscle, by John Iovine
 "Atlanta 1996", Will Atlanta host the 1996 Summer Olympics? Their best salesperson is an Amiga 2500.
 "CAD Overview: X-CAD Designer, X-CAD Professional, IntroCAD Plus, Aegis Draw 2000, UltraDesign", by Douglas Bullard
 "Saxon Publisher", review by David Duberman
 "AutoPrompt", review by Frank McMahon
 "Sound Tools for the Amiga", Sunrise Industries' Perfect Sound and MichTron's Master Sound, reviews by M. Kevelson
 "Stripping Layers Off Workbench", Remove unused files on your Workbench to make room for other programs, by Keith Cameron
 "Audio Illusion", Produce fascinating auditory illusions on your Amiga, by Craig Zagke
 "Call Assembly Language From Modula-2", Integrating small, fast machine language programs into BASIC, by Martin Combs
 "Koch Flakes", Using the preprocessor to perform selective compilation, by Paul Castonguay
 "C Notes from The C Group", A program that examines an archive file and removes any files that have been extracted, by Stephen Kemp

Vol. 5 No. 11, November 1990

Highlights include:

"Getting A Lot For A Little", A comparison of the available Amiga archive programs, by Greg Epley
 "Amiga Vision", review by John Steiner
 "High Density Media Comes to the Amiga", Applied Engineering's AEHD drive, review by John Steiner
 "Filing The Flicker", MicroWay's Advanced Graphics Adaptor 2000, by John Steiner
 "The KCS Power PC Board", If you have an Amiga 300, and need IBM PC/XT software compatibility, the KCS Power PC Board can help, by Ernest P. Viveiros, Jr.
 "Build An Amiga 2000 Keyboard For The Amiga 1000", Get a better-feeling keyboard for under \$7.00, by Philip R. Combs
 "Looking Beyond The Saud Rate", The Saud Bandit 2400 & Bandit MNP/Levi 5 Plus modems, by R. P. Viveiros, Jr.
 "C Notes From The C Group", Programming with definitions known as "enumerated" data types, by Stephen Kemp
 "ASIC Compiler", review by Bruce M. Drake
 "Mindware's 3D Test Animator", review by Frank McMahon
 "A Little Closer to Excellence", Micro-Systems Software's excellent 3D, review by Kim Schaller

Vol. 5 No. 12, December 1990

Highlights include:

"Twin Peaks Amiga Show Report", AC traveled to AmiEXPO in Anaheim, CA and World of Amiga in Chicago, IL to report on the newest and brightest Amiga products.
 "Information X-Change", Keeping up to date on the latest news via hardware, software, and cable TV, by Rick Broda
 "Stepper Motors", Part One of three part series on building a simple stepper motor, by John Iovine
 "C Notes From The C Group", A discussion on cryptography, by Stephen Kemp
 "Pro Video Post", review by Frank McMahon
 "Feeding The Memory Monster", the ICD AdRAM 540 and AdRAM 560D, review by Ernest P. Viveiros, Jr.
 "McGee & McGee Visits Katie's Farm", review by Jeff James
 "Wings", review by Rick Broda
 "MathVisian 2.0", review by R. Shamus Mortier
 "Making A Name For Yourself", Creating logos on the Amiga, by Frank McMahon
 "Hard Disk Primer For Floppy Users", Taking the sting out of the transition from floppies to hard drive, by Rob Hays
 "Shotgun Approach To Programming With AmigaBASIC", Bringing the fundamentals of AmigaBASIC programming into perspective, by Mike Morrison

Vol. 6 No. 1, January 1991

Highlights include:

"On The Road", coverage of Germany's Amiga '90, COMDEX in Nevada, and The World of Commodore Amiga in Toronto, Canada
 "Electronic Color Splitter", an inexpensive way to grab images off video sources, by Greg Epley
 "SketchMaster", review by Ernest P. Viveiros, Jr.
 "Professional Draw 2.0", review by R. Shamus Mortier
 "Spell-A-Fair", review by Jeff James
 "Programming in AmigaBASIC", by Mike Morrison
 "ZoomBos", by John Leonard
 "Medley", AC's music column discusses MIDI, by Phil Saunders
 "Bug Bytes", a few problems with PageStream 2.0 and Quarterback Tools is now shipping, by John Steiner
 "The Animation Studio", Disney's classic approach in a character animation program, by Frank McMahon
 "Toenail Animation", the Amiga helps out in the courtroom, by Andrew Lichtman
 "Cartoon Animation", back to the basics, by D. L. Richardson
 "Animation Chart", twenty-two animation packages and features
 "Memory & Animation", even 512K users can animate, by Chris Boyer

Vol. 6 No. 2, February 1991

Highlights include:

"Xeler's CDx-450", CD-ROM technology for the Amiga, by Lonnie Watson
 "Distant Suns Libraries", Distant Suns expansion disks, by Jeff James
 "ANIMagic", A graphics tool to spice up your presentations, by Rajesh Godi
 "Sharing Your Amiga Hard Drive With The Bridgeboard", Partition your hard drive to run both AmigaDOS and MS-DOS systems, by Gene Rawls
 "More Ports For Your Amiga", Building an I/O Expansion Board, by Jeff Lavin
 "Medley", A look at different types of music software available, by Phil Saunders
 "C Notes From The C Group", Creating a reminder program, by Stephen Kemp
 "Bug Bytes", New upgrades are in the works for PageStream and Professional Page, by John Steiner
 "The 9-to-5 Amiga", by Daryell Sipper
 "Gold Disk Office", by Chuck Raudonis
 "dataTAX", by Daryell Sipper
 "Gold Disk's Desktop Budget", by Chuck Raudonis
 "BGraphics", by Chuck Raudonis

Vol. 6 No. 3, March 1991

Highlights include:

"Winter '91 CES", CDTV developers demonstrate upcoming releases and Amiga games developers present their latest creations in Las Vegas, Nevada
 "NewTek's Video Toaster: A New Era in Amiga Video", a complete tour of the Video Toaster, by Frank McMahon
 "Ultrasonic Ranging System", the sonar system project continues with the assembly of an ultrasonic ranging system, by John Iovine
 "Writing Faster Assembly Language", the discussion on how to speed up programs with assembly is completed, by Martin F. Combs
 "Programming in AmigaBASIC: Conditionals", using the IF/THEN statement in AmigaBASIC, by Mike Morrison
 "New Products And Other Neat Stuff", an advanced ray-tracing module for 3-D Professional, RandsdTypes gets a price reduction, and DCTV is released, by John Rozendek
 "Bug Bytes", more workarounds for some popular programs, by John Steiner
 "Roomers", Is NewTek getting a run for their money with Digital Creations' V-Machine?, by The Bandito
 "Diversions", Night Shift, James Bond: The Stealth Affair, Wolf Pack, PowerMonger, and Harpoon are reviewed
 "Medley", learn how to load and modify MIDI files with your sequencer, by Phil Saunders
 "FD Serendipity", create your own muntis to save to the bootblock with MenuWriter, or convert IFF pictures to C or assembly with IFF2Source, by Almee B. Abren
 "C Notes From The C Group", working with functions in C, by Stephen Kemp
 "Spirit Technology's HDA-506", a less expensive alternative for Amiga 1000 & 500 owners, by Mike C. Corbett
 "Mambo Paint", Lake Forest Logic's Dynamic hi-res, by R. Shamus Mortier
 "An Impulse To Imagine", review by R. Shamus Mortier
 "Top Feet", Designing Minds' dedicated forum publisher, by Jeff James
 "Quarterback Tools", a disk and file repair program to help fix system crashes and accidental file deletions, by John Steiner

Name _____
 Address _____
 City _____ State _____ ZIP _____
 Charge my ☐ Visa ☐ MC # _____
 Expiration Date _____ Signature _____



Please circle to indicate this is a **New Subscription** or a **Renewal**

PROPER ADDRESS REQUIRED: In order to expedite and guarantee your order, all large Public Domain Software orders, as well as most Back Issue orders, are shipped by United Parcel Service. UPS requires that all packages be addressed to a street address for correct delivery.
 PAYMENTS BY CHECK: All payments made by check or money order must be in US funds drawn on a US bank.

One Year of Amazing!	Save over 49% 12 monthly issues of the number-one resource to the Commodore Amiga, <i>Amazing Computing</i> at a savings of over \$23.00 off the newsstand price!	<input type="checkbox"/> \$24.00 US <input type="checkbox"/> \$44.00 Foreign Surface <input type="checkbox"/> \$34.00 Canada and Mexico
One Year of AC SuperSub!	Save over 46% 12 monthly issues of <i>Amazing Computing</i> PLUS <i>AC's GUIDE / AMIGA</i> 2 Product Guides a year! A savings of \$31.30 off the newsstand price!	<input type="checkbox"/> \$36.00 US <input type="checkbox"/> \$64.00 Foreign Surface <input type="checkbox"/> \$54.00 Canada and Mexico
Two Years of Amazing!	Save over 59% 24 monthly issues of the number one resource to the Commodore Amiga, <i>Amazing Computing</i> at a savings of over \$56.80 off the newsstand price!	<input type="checkbox"/> \$38.00 US (sorry no foreign orders available at this frequency)
Two Years of AC SuperSub!	Save over 56% 24 monthly issues of <i>Amazing Computing</i> PLUS <i>AC's GUIDE / AMIGA</i> 4 Complete Product Guides! A savings of \$75.60 off the newsstand price!	<input type="checkbox"/> \$59.00 US (sorry no foreign orders available at this frequency)

Please circle any additional choices below:

(Domestic and Foreign air mail rates available on request)

Back Issues: \$5.00 each US, \$6.00 each Canada and Mexico, \$7.00 each Foreign Surface.

1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 2.10
 2.11 2.12 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 3.10 3.11 3.12 4.1 4.2 4.3 4.4 4.5
 4.6 4.7 4.8 4.9 4.10 4.11 4.12 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 5.10 5.11 5.12
 6.1 6.2 6.3

Back Issue Volumes: Volume 1-\$19.95* Volume 2-\$29.95* Volumes 3, 4, or 5-\$29.95* each

*All volume orders must include postage and handling charges: \$4.00 each set US, \$7.50 each set Canada and Mexico, and \$10.00 each set for foreign surface orders. Air mail rates available.

NEW! AC's TECH / AMIGA Single issues just \$14.95 each: *Pictures (V1-1)* V1.2

Order a One-Year Subscription to *AC's TECH* Now - Get 4 **BIG** Issues!

Charter Rate Offer: \$39.95 (limited time offer - US only)!

Canada & Mexico: \$43.95

Foreign Surface: \$47.95

Call or write for Air Mail rates!

Freely Distributable Software - Subscriber Special (yes, even the new ones!)

1 to 9 disks \$6.00 each
 10 to 49 disks \$5.00 each
 50 to 99 disks \$4.00 each
 100 or more disks \$3.00 each

\$7.00 each for non subscribers (three disk minimum on all foreign orders)

Amazing on Disk: AC#1 ...Source & Listings V2.8 & V3.9
 AC#2 ...Source & Listings V4.5 & V4.6
 AC#5 ...Source & Listings V4.9
 AC#7 ...Source & Listings V4.12 & V5.1
 AC#9 ...Source & Listings V5.4 & V5.5
 AC#11 ...Source & Listings V5.8, 5.9 & 5.10

InNOCKulation Disk: IN#1 ...Virus protection

AC#2 ...Source & Listings V4.3 & V4.4
 AC#4 ...Source & Listings V4.7 & V4.8
 AC#6 ...Source & Listings V4.10 & V4.11
 AC#8 ...Source & Listings V5.2 & 5.3
 AC#10 ...Source & Listings V5.6 & 5.7
 AC#12 ...Source & Listings V5.11, 5.12 & 6.1
 AC#13 ...Source & Listings V6.2 & 6.3

Please list your Freely Redistributable Software selections below:

AC Disks _____

(numbers 1 through 12)

AMICUS _____

(numbers 1 through 26)

Fred Fish Disks _____

(numbers 1 through 460; FF395 is currently unavailable. Please remember Fred Fish Disks 57, 80, & 87 have been removed from the collection)

Complete Today, or Telephone 1-800-345-3360

Subscription: \$ _____

Back Issues: \$ _____

AC's TECH: \$ _____

PDS Disks: \$ _____

Total: \$ _____

(subject to applicable sales tax)

Please complete this form and mail with check, money order or credit card information to:

P.I.M. Publications, Inc.
 P.O. Box 869
 Fall River, MA 02722-0869

Please allow 4 to 6 weeks for delivery of subscriptions in US.

AC's TECH/AMIGA Reader Service Card

Name _____
 Street _____
 City _____ State _____ ZIP _____
 Country _____

AC's TECH Volume 1.2

Valid until 6/15/91

See page 64 for reference numbers

A. At your option, please provide the following data:

1. Male ☐ 2. Married ☐
 3. Female ☐ 4. Single/never married ☐
 5. Separated/divorced ☐

B. What is your age?

- 0 6. Under 18 ☐ 8. 25-49 ☐
 7. 18-24 ☐ 9. 50-64 ☐
 8. 25-34 ☐ 10. 65 or over ☐

C. Which of the following do you now own?

- (Please check all that apply)
 012. Amiga 2000 ☐ 015. Amiga 1000 ☐
 013. Amiga 500 ☐ 016. Amiga 500 ☐
 014. Amiga 2500 ☐ 017. do not own an Amiga ☐

D. Where do you use your Amiga(s), and about how many hours per week do you use an Amiga at each location?

018. At home _____ hours per week
 019. Home office _____ hours per week

020. At work _____ hours per week
 (type of business _____)
 (type of business _____)

021. At school _____ hours per week
 (specifications _____)

- TOTAL _____ hours per week

E. Please indicate the primary and secondary applications for which you use your Amiga(s).

22. Primary _____
 23. Secondary _____

F. Please indicate the level at which you now consider yourself to be programming the Amiga:

024. Beginner ☐ 026. Advanced ☐
 025. Intermediate ☐ 027. Do not program ☐

G. What language do you most often program in?

28. _____
 How much money are you likely to spend on an Amiga product purchases this year?

- (all personal and business/education for which you have final decision, combined)
 029. Less than \$250 ☐ 034. \$2501-\$5000 ☐
 030. \$251-\$500 ☐ 035. \$5001-\$7500 ☐
 031. \$501-\$1000 ☐ 036. \$7501-\$10000 ☐
 032. \$1001-\$1500 ☐ 037. Over \$10000 ☐
 033. \$1501-\$2500 ☐

H. How did you obtain this issue of AC's TECH?

038. Preordered this issue prior to its publication.
 039. Ordered this issue after seeing it somewhere.
 040. Ordered a charter subscription.
 041. Purchased it at my Amiga dealer.

I. How many pages (not including inserts) have read?

042. _____ others, in addition to myself
 Overall, how would you rate this issue of AC's TECH in the areas indicated?

- Editorial content: _____
 043. Excellent ☐ 047. Excellent ☐
 044. Good ☐ 048. Good ☐
 045. Fair ☐ 049. Fair ☐
 046. Poor ☐ 050. Poor ☐

J. Do you regularly read or subscribe to Amazing Computing?

051. I am a subscriber to AC ☐
 052. I read, but do not subscribe to AC ☐
 053. Do not read or subscribe to AC ☐

K. Have you ever purchased a copy of AC's Guide?

054. Yes ☐ 055. No ☐

101	102	103	104	105	221	222	223	224	225
106	107	108	109	110	226	227	228	229	230
111	112	113	114	115	231	232	233	234	235
116	117	118	119	120	236	237	238	239	240
121	122	123	124	125	241	242	243	244	245
126	127	128	129	130	246	247	248	249	250
131	132	133	134	135	251	252	253	254	255
136	137	138	139	140	256	257	258	259	260
141	142	143	144	145	261	262	263	264	265
146	147	148	149	150	266	267	268	269	270
151	152	153	154	155	271	272	273	274	275
156	157	158	159	160	276	277	278	279	280
161	162	163	164	165	281	282	283	284	285
166	167	168	169	170	286	287	288	289	290
171	172	173	174	175	291	292	293	294	295
176	177	178	179	180	296	297	298	299	300
181	182	183	184	185	301	302	303	304	305
186	187	188	189	190	306	307	308	309	310
191	192	193	194	195	311	312	313	314	315
196	197	198	199	200	316	317	318	319	320
201	202	203	204	205	321	322	323	324	325
206	207	208	209	210	326	327	328	329	330
211	212	213	214	215	331	332	333	334	335
216	217	218	219	220	336	337	338	339	340

AC's TECH/AMIGA Reader Service Card

Name _____
 Street _____
 City _____ State _____ ZIP _____
 Country _____

AC's TECH Volume 1.2

Valid until 6/15/91

See page 64 for reference numbers

A. At your option, please provide the following data:

1. Male ☐ 2. Married ☐
 3. Female ☐ 4. Single/never married ☐
 5. Separated/divorced ☐

B. What is your age?

- 0 6. Under 18 ☐ 8. 25-49 ☐
 7. 18-24 ☐ 9. 50-64 ☐
 8. 25-34 ☐ 10. 65 or over ☐

C. Which of the following do you now own?

- (Please check all that apply)
 012. Amiga 2000 ☐ 015. Amiga 1000 ☐
 013. Amiga 500 ☐ 016. Amiga 500 ☐
 014. Amiga 2500 ☐ 017. do not own an Amiga ☐

D. Where do you use your Amiga(s), and about how many hours per week do you use an Amiga at each location?

018. At home _____ hours per week
 019. Home office _____ hours per week

020. At work _____ hours per week
 (type of business _____)
 (type of business _____)

021. At school _____ hours per week
 (specifications _____)

- TOTAL _____ hours per week

E. Please indicate the primary and secondary applications for which you use your Amiga(s).

22. Primary _____
 23. Secondary _____

F. Please indicate the level at which you now consider yourself to be programming the Amiga:

024. Beginner ☐ 026. Advanced ☐
 025. Intermediate ☐ 027. Do not program ☐

G. What language do you most often program in?

28. _____
 How much money are you likely to spend on an Amiga product purchases this year?

- (all personal and business/education for which you have final decision, combined)
 029. Less than \$250 ☐ 034. \$2501-\$5000 ☐
 030. \$251-\$500 ☐ 035. \$5001-\$7500 ☐
 031. \$501-\$1000 ☐ 036. \$7501-\$10000 ☐
 032. \$1001-\$1500 ☐ 037. Over \$10000 ☐
 033. \$1501-\$2500 ☐

H. How did you obtain this issue of AC's TECH?

038. Preordered this issue prior to its publication.
 039. Ordered this issue after seeing it somewhere.
 040. Ordered a charter subscription.
 041. Purchased it at my Amiga dealer.

I. How many pages (not including inserts) have read?

042. _____ others, in addition to myself
 Overall, how would you rate this issue of AC's TECH in the areas indicated?

- Editorial content: _____
 043. Excellent ☐ 047. Excellent ☐
 044. Good ☐ 048. Good ☐
 045. Fair ☐ 049. Fair ☐
 046. Poor ☐ 050. Poor ☐

J. Do you regularly read or subscribe to Amazing Computing?

051. I am a subscriber to AC ☐
 052. I read, but do not subscribe to AC ☐
 053. Do not read or subscribe to AC ☐

K. Have you ever purchased a copy of AC's Guide?

054. Yes ☐ 055. No ☐

101	102	103	104	105	221	222	223	224	225
106	107	108	109	110	226	227	228	229	230
111	112	113	114	115	231	232	233	234	235
116	117	118	119	120	236	237	238	239	240
121	122	123	124	125	241	242	243	244	245
126	127	128	129	130	246	247	248	249	250
131	132	133	134	135	251	252	253	254	255
136	137	138	139	140	256	257	258	259	260
141	142	143	144	145	261	262	263	264	265
146	147	148	149	150	266	267	268	269	270
151	152	153	154	155	271	272	273	274	275
156	157	158	159	160	276	277	278	279	280
161	162	163	164	165	281	282	283	284	285
166	167	168	169	170	286	287	288	289	290
171	172	173	174	175	291	292	293	294	295
176	177	178	179	180	296	297	298	299	300
181	182	183	184	185	301	302	303	304	305
186	187	188	189	190	306	307	308	309	310
191	192	193	194	195	311	312	313	314	315
196	197	198	199	200	316	317	318	319	320
201	202	203	204	205	321	322	323	324	325
206	207	208	209	210	326	327	328	329	330
211	212	213	214	215	331	332	333	334	335
216	217	218	219	220	336	337	338	339	340



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 36 FALL RIVER, MA

Postage Will Be Paid By Addressee:

AC^{ts} TECH / AMIGA

P.i.M. Publications, Inc.

P.O. Box 869

Fall River, MA 02722-0969



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 36 FALL RIVER, MA

Postage Will Be Paid By Addressee:

AC^{ts} TECH / AMIGA

P.i.M. Publications, Inc.

P.O. Box 869

Fall River, MA 02722-0969



SAVE! SAVE! SAVE!

If AC's TECH is just what you've been looking for in an Amiga publication, don't just sit there –

SUBSCRIBE!

Also consider two more great reasons to be a subscriber – AC and AC's GUIDE! Just fill out, clip and mail this card along with your payment to pick up on the

SAVINGS!

YES! The Amazing AC publications give me 3 GREAT reasons to save!
Please begin the subscription(s) indicated below immediately!

Name _____

Address _____

City _____ State _____ ZIP _____

Charge my ☐ Visa ☐ MC # _____

Expiration Date _____ Signature _____

Please circle to indicate this is a New Subscription or a Renewal



1 year of AC's TECH	4 big issues of AC's TECH! Limited Time Charter Offer!	US \$39.95 <input type="checkbox"/>
		Canada/Mexico \$43.95 <input type="checkbox"/>
		Foreign Surface \$47.95 <input type="checkbox"/>
1 year of AC	12 big issues of Amazing Computing! Save over 59% off the cover price!	US \$24.00 <input type="checkbox"/>
		Canada/Mexico \$34.00 <input type="checkbox"/>
		Foreign Surface \$44.00 <input type="checkbox"/>
1-year SuperSub	AC + AC's GUIDE – 14 issues total! Save more than \$31 off the cover prices!	US \$36.00 <input type="checkbox"/>
		Canada/Mexico \$54.00 <input type="checkbox"/>
		Foreign Surface \$64.00 <input type="checkbox"/>
2 years of AC	24 big issues! Save over 59%! US only.	US \$38.00 <input type="checkbox"/>
2-year SuperSub	28 big issues! Save more than \$75! US only.	US \$59.00 <input type="checkbox"/>

Check or money order payments must be in US funds drawn on a US bank; subject to applicable sales tax.

Please return to:

AC's TECH/AMIGA

P.i.M. Publications, Inc.

P.O. Box 869

Fall River, MA 02722-0869

Please place this order form in an envelope
with your check or money order.



*Captain's Log 217:2055
Time shift equalization routines recoded in
Aztec C. All systems go. It's good to be back home.*

New Releases

Aztec C v5

- Amiga
- Macintosh
- MS-DOS
- 80x86, 680x0 ROM

**Call Today for
Product, Promotional
and Pricing Information**

Aztec C is available for Amiga, Macintosh, MS-DOS, Apple II, TRS-80, and CPM/80. Native and cross development.

Aztec Embedded C hosts on MS-DOS and Macintosh. Targets include: 68000 family, 8086 family, 8080/280/64180, & 6502

Aztec C includes - C compiler, assembler, linker, librarian, editor, source debugger, and other development tools.

MANX  **800-221-0440**

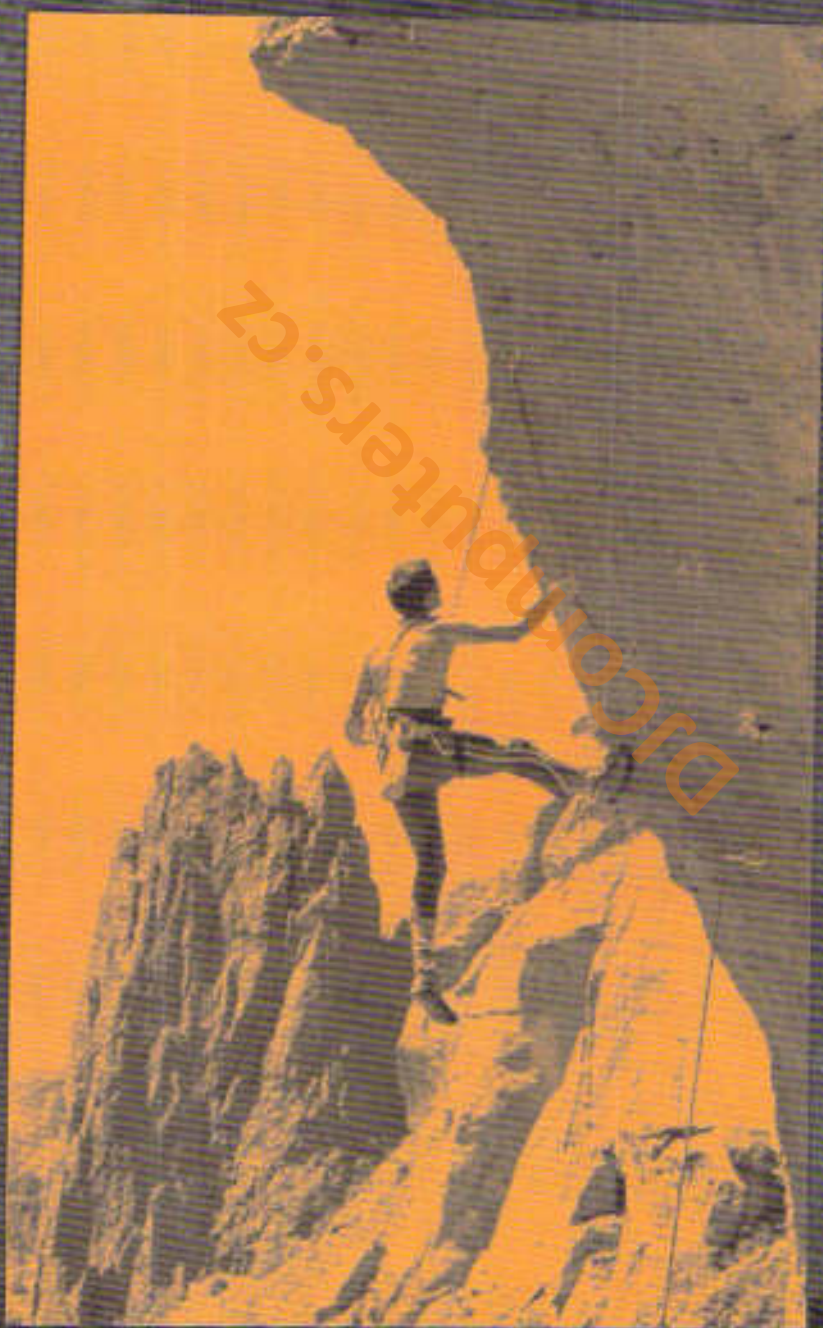
• Outside USA: 908-542-2121 FAX: 908-542-8386 • Visa, MC, Am Ex, C.O.D,
domestic & international wire • one and two day delivery available

Manx Software Systems • 160 Avenue of the Commons • Box 55 • Shrewsbury NJ 07702

Circle 187 on Reader Service card.

It's truly incredible.

**You've reached your current level of Amiga
mastery simply by reading a few issues of
Amazing Computing.**



Now – to accelerate your ascent – become an AC subscriber!

Once you've made the commitment to scale the heights of personal computing with your Amiga, it's important to allocate your time and your money efficiently. You also want to come to play with as many quality tools as possible on a daily basis.

A one-year subscription to *Amazing Computing* fulfills these requirements completely and intelligently: AC is sent to your door monthly, and you pay **just \$2.00 per issue!** And as an AC subscriber, you get **maximum, timely coverage** of important new products, technologies, and trends. You get fully detailed reviews, techniques, projects and analyses that help you develop and refine your skills on the Amiga every day.

Include *AC's GUIDE* in your subscription for an informative and unsurpassed overview of the growing and ever-changing Amiga marketplace. At **320 pages**, *AC's GUIDE* towers as the world's largest and most complete reference to everything presently available for the Amiga. And no other Amiga publication gives you **complete contact information** for every known product developer and hundreds of users groups worldwide!

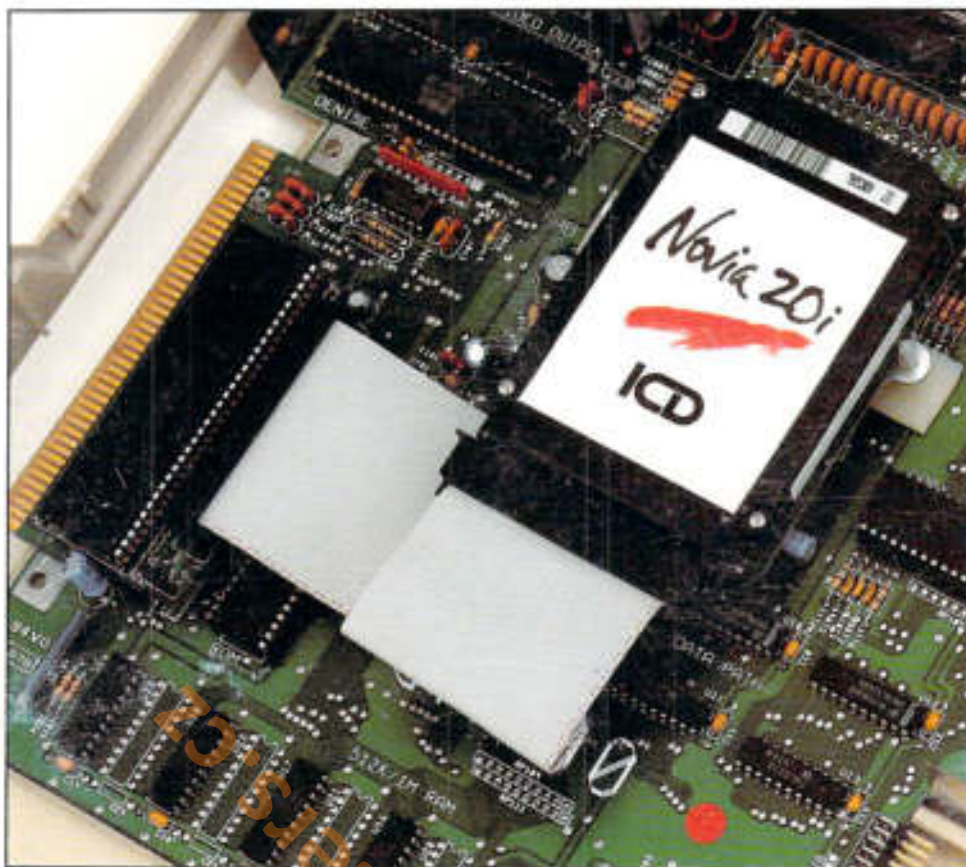
And when you're ready to broaden the scope of your knowledge and ability to the maximum degree, expert technical proficiency is also within your grasp, thanks to *AC's TECH*. It's the first and best **disk-based, all-technical, applications-intensive** Amiga journal!

Gather the skills and sharpen the tools you need to get the most from your Amiga as you accept the challenge of ascending to new heights! Subscribe to *Amazing Computing*, *AC's GUIDE* and *AC's TECH* – today's incredible power trio of Amiga publications.



Be an AC subscriber – and have the climb of your life!

To order *Amazing Computing*, *AC's GUIDE* and *AC's TECH*, call toll free 1-800-345-3360 (credit card orders only). Or, send us your payment along with one of the convenient subscription cards or order forms appearing in this issue.



Novia 20i



The smallest hard drive and interface in the world is now available for your Amiga 500 computer! This rugged little sweetheart mounts completely inside your computer allowing 20 megabytes of high speed storage that takes absolutely no desk space. The advanced features include autobooting from FastFileSystem partitions, high speed caching, auto-configuring, and A-Max II support. Novia 20i comes with complete instructions and all the hardware necessary for a simple, clean, no-solder installation. Available today at Amiga dealers everywhere.

Novia is a trademark of ICD, Incorporated. Amiga is a registered trademark of Commodore-Amiga, Inc. A-Max II is a trademark of ReadySoft.

ICD

ICD, Incorporated 1220 Rock Street Rockford, IL 61101
USA

(815) 968-2228 Phone (815) 968-6888 FAX

Circle 123 on Reader Service Card